Intermediate Quantitative Economics with Python

Thomas J. Sargent & John Stachurski

Jul 26, 2026

CONTENTS

I Tools and Techniques **7**

1 Modeling COVID 19 **9**
- 1.1 Overview . . . 9
- 1.2 The SIR Model . . . 10
- 1.3 Implementation . . . 11
- 1.4 Experiments . . . 12
- 1.5 Ending Lockdown . . . 17

2 Linear Algebra **21**
- 2.1 Overview . . . 21
- 2.2 Vectors . . . 22
- 2.3 Matrices . . . 30
- 2.4 Solving Systems of Equations . . . 32
- 2.5 Eigenvalues and Eigenvectors . . . 37
- 2.6 Further Topics . . . 40
- 2.7 Exercises . . . 42

3 QR Decomposition **47**
- 3.1 Overview . . . 47
- 3.2 Matrix Factorization . . . 47
- 3.3 Gram-Schmidt process . . . 47
- 3.4 Some Code . . . 49
- 3.5 Example . . . 50
- 3.6 Using QR Decomposition to Compute Eigenvalues . . . 52
- 3.7 QR and PCA . . . 53

4 Circulant Matrices **55**
- 4.1 Overview . . . 55
- 4.2 Constructing a Circulant Matrix . . . 55
- 4.3 Connection to Permutation Matrix . . . 57
- 4.4 Examples with Python . . . 58
- 4.5 Associated Permutation Matrix . . . 62
- 4.6 Discrete Fourier Transform . . . 64

5 Singular Value Decomposition (SVD) **71**
- 5.1 Overview . . . 71
- 5.2 The Setting . . . 71
- 5.3 Singular Value Decomposition . . . 72
- 5.4 Four Fundamental Subspaces . . . 73
- 5.5 Eckart-Young Theorem . . . 76

5.6 Full and Reduced SVD's . . . 77
5.7 Polar Decomposition . . . 80
5.8 Application: Principal Components Analysis (PCA) . . . 80
5.9 Relationship of PCA to SVD . . . 82
5.10 PCA with Eigenvalues and Eigenvectors . . . 83
5.11 Connections . . . 84
5.12 Exercises . . . 87

6 VARs and DMDs **89**
6.1 First-Order Vector Autoregressions . . . 89
6.2 Dynamic Mode Decomposition (DMD) . . . 92
6.3 Representation 1 . . . 92
6.4 Representation 2 . . . 93
6.5 Representation 3 . . . 95
6.6 Source for Some Python Code . . . 99

7 Using Newton's Method to Solve Economic Models **101**
7.1 Overview . . . 101
7.2 Fixed point computation using Newton's method . . . 102
7.3 Root-Finding in one dimension . . . 108
7.4 Multivariate Newton's method . . . 110
7.5 Exercises . . . 119

II Elementary Statistics **125**

8 Elementary Probability with Matrices **127**
8.1 Sketch of basic concepts . . . 128
8.2 What does probability mean? . . . 128
8.3 Representing probability distributions . . . 130
8.4 Univariate probability distributions . . . 130
8.5 Bivariate probability distributions . . . 132
8.6 Marginal probability distributions . . . 132
8.7 Conditional probability distributions . . . 133
8.8 Transition probability matrix . . . 133
8.9 Application: forecasting a time series . . . 134
8.10 Statistical independence . . . 134
8.11 Means and variances . . . 135
8.12 Matrix representations of some bivariate distributions . . . 135
8.13 A continuous bivariate random vector . . . 142
8.14 Sum of two independently distributed random variables . . . 151
8.15 Coupling . . . 152
8.16 Copula functions . . . 153
8.17 Exercises . . . 160

9 Some Probability Distributions **167**
9.1 Some Discrete Probability Distributions . . . 167
9.2 Geometric distribution . . . 167
9.3 Pascal (negative binomial) distribution . . . 168
9.4 Newcomb–Benford distribution . . . 169
9.5 Univariate Gaussian distribution . . . 171
9.6 Uniform Distribution . . . 171
9.7 A Mixed Discrete-Continuous Distribution . . . 172
9.8 Drawing a Random Number from a Particular Distribution . . . 173

10 LLN and CLT **181**
10.1 Overview . . . 181
10.2 Relationships . . . 182
10.3 LLN . . . 182
10.4 CLT . . . 186
10.5 Exercises . . . 192

11 Two Meanings of Probability **199**
11.1 Overview . . . 199
11.2 Frequentist Interpretation . . . 200
11.3 Bayesian Interpretation . . . 206
11.4 Comparing the two interpretations . . . 214
11.5 Role of a Conjugate Prior . . . 215

12 Multivariate Hypergeometric Distribution **217**
12.1 Overview . . . 217
12.2 The Administrator's Problem . . . 217
12.3 Usage . . . 221

13 Multivariate Normal Distribution **227**
13.1 Overview . . . 227
13.2 The multivariate normal distribution . . . 228
13.3 Bivariate example . . . 231
13.4 Trivariate example . . . 235
13.5 One dimensional intelligence (IQ) . . . 236
13.6 Information as surprise . . . 240
13.7 Cholesky factor magic . . . 242
13.8 Math and verbal intelligence . . . 242
13.9 Univariate time series analysis . . . 245
13.10 Stochastic difference equation . . . 250
13.11 Application to stock price model . . . 251
13.12 Filtering foundations . . . 253
13.13 Classic factor analysis model . . . 257
13.14 PCA and factor analysis . . . 260
13.15 Exercises . . . 265

14 Fault Tree Uncertainties **275**
14.1 Overview . . . 275
14.2 Log normal distribution . . . 276
14.3 The Convolution Property . . . 277
14.4 Approximating Distributions . . . 278
14.5 Convolving Probability Mass Functions . . . 282
14.6 Failure Tree Analysis . . . 284
14.7 Application . . . 285
14.8 Failure Rates Unknown . . . 285
14.9 Waste Hoist Failure Rate . . . 286

15 Introduction to Artificial Neural Networks **291**
15.1 Overview . . . 292
15.2 A Deep (but not Wide) Artificial Neural Network . . . 292
15.3 Calibrating Parameters . . . 293
15.4 Back Propagation and the Chain Rule . . . 294
15.5 Training Set . . . 295
15.6 Example 1 . . . 298
15.7 How Deep? . . . 299

15.8 Example 2 . . . 300

16 Randomized Response Surveys **303**
16.1 Overview . . . 303
16.2 Warner's Strategy . . . 303
16.3 Comparing Two Survey Designs . . . 305
16.4 Concluding Remarks . . . 311

17 Expected Utilities of Random Responses **313**
17.1 Overview . . . 313
17.2 Privacy Measures . . . 313
17.3 Zoo of Concepts . . . 313
17.4 Respondent's Expected Utility . . . 315
17.5 Utilitarian View of Survey Design . . . 319
17.6 Criticisms of Proposed Privacy Measures . . . 322
17.7 Concluding Remarks . . . 327

III Bayes Law 329

18 Non-Conjugate Priors **331**
18.1 Overview . . . 331
18.2 The coin-flipping model . . . 332
18.3 MCMC reproduces the conjugate posterior . . . 334
18.4 Non-conjugate priors . . . 336
18.5 Variational inference . . . 341
18.6 Where to next . . . 345

19 Posterior Distributions for AR(1) Parameters **347**
19.1 Overview . . . 347
19.2 Estimation . . . 349
19.3 Conclusion . . . 359

20 Forecasting an AR(1) Process **361**
20.1 A Univariate First-Order Autoregressive Process . . . 362
20.2 Implementation . . . 363
20.3 Predictive Distributions of Path Properties . . . 368
20.4 A Wecker-Like Algorithm . . . 369
20.5 Using Simulations to Approximate a Posterior Distribution . . . 369
20.6 Calculating Sample Path Statistics . . . 370
20.7 Original Wecker Method . . . 372
20.8 Extended Wecker Method . . . 373
20.9 Comparison . . . 377
20.10 Conclusion . . . 377

IV Statistics and Information 379

21 Statistical Divergence Measures **381**
21.1 Overview . . . 381
21.2 Primer on entropy, cross-entropy, KL divergence . . . 382
21.3 Two Beta distributions: running example . . . 383
21.4 Kullback–Leibler divergence . . . 384
21.5 Jensen-Shannon divergence . . . 385
21.6 Chernoff entropy . . . 387

21.7 Comparing divergence measures . . . 388
21.8 KL divergence and maximum-likelihood estimation . . . 390
21.9 Related lectures . . . 391

22 Likelihood Ratio Processes **393**
22.1 Overview . . . 393
22.2 Likelihood Ratio Process . . . 394
22.3 Nature permanently draws from density g . . . 396
22.4 Peculiar property . . . 399
22.5 Nature permanently draws from density f . . . 399
22.6 Likelihood ratio test . . . 401
22.7 Hypothesis testing and classification . . . 410
22.8 Markov chains . . . 421
22.9 Related lectures . . . 424
22.10 Exercises . . . 424

23 Heterogeneous Beliefs and Financial Markets **427**
23.1 Overview . . . 427
23.2 Review: likelihood ratio processes . . . 428
23.3 Blume and Easley's setting . . . 430
23.4 Nature and agents' beliefs . . . 430
23.5 A socialist risk-sharing arrangement . . . 431
23.6 Social planner's allocation problem . . . 431
23.7 If you're so smart, ... 433
23.8 Competitive equilibrium prices . . . 433
23.9 Simulations . . . 436
23.10 Related lectures . . . 442
23.11 Exercises . . . 442

24 Likelihood Processes For VAR Models **465**
24.1 Overview . . . 465
24.2 VAR model setup . . . 466
24.3 Likelihood ratio process . . . 469
24.4 Example 1: two AR(1) processes . . . 470
24.5 Example 2: bivariate VAR models . . . 471
24.6 Application: Samuelson multiplier-accelerator . . . 475

25 Mean of a Likelihood Ratio Process **481**
25.1 Overview . . . 481
25.2 Mathematical expectation of likelihood ratio . . . 482
25.3 Importance sampling . . . 484
25.4 Selecting a sampling distribution . . . 485
25.5 Approximating a cumulative likelihood ratio . . . 486
25.6 Distribution of sample mean . . . 488
25.7 Choosing a sampling distribution . . . 490

26 A Problem that Stumped Milton Friedman **495**
26.1 Overview . . . 495
26.2 Source of the Problem . . . 496
26.3 Neyman-Pearson formulation . . . 497
26.4 Wald's sequential formulation . . . 499
26.5 Links between A, B and α, β . . . 501
26.6 Simulations . . . 502
26.7 Related lectures . . . 513
26.8 Exercises . . . 513

27 A Bayesian Formulation of Friedman and Wald's Problem **521**
27.1 Overview . . . 521
27.2 A Dynamic Programming Approach . . . 523
27.3 Implementation . . . 528
27.4 Analysis . . . 530

28 Exchangeability and Bayesian Updating **537**
28.1 Overview . . . 537
28.2 Independently and Identically Distributed . . . 538
28.3 A Setting in Which Past Observations Are Informative . . . 539
28.4 Relationship Between IID and Exchangeable . . . 540
28.5 Exchangeability . . . 541
28.6 Bayes' Law . . . 541
28.7 More Details about Bayesian Updating . . . 542
28.8 Appendix . . . 545
28.9 Sequels . . . 552

29 Likelihood Ratio Processes and Bayesian Learning **553**
29.1 Overview . . . 553
29.2 The Setting . . . 554
29.3 Likelihood Ratio Processes and Bayes' Law . . . 555
29.4 Another timing protocol . . . 560
29.5 Behavior of Posterior Probability $\{\pi_t\}$ Under Subjective Probability Distribution . . . 563
29.6 Initial Prior is Verified by Paths Drawn from Subjective Conditional Densities . . . 570
29.7 Drilling Down a Little Bit . . . 571
29.8 Related Lectures . . . 572

30 Blackwell's Theorem on Comparing Experiments **573**
30.1 Overview . . . 573
30.2 Experiments and stochastic transformations . . . 574
30.3 Three equivalent criteria . . . 577
30.4 The main theorem . . . 579
30.5 Kihlstrom's Bayesian interpretation . . . 579
30.6 Simulating the Blackwell order with many states . . . 584
30.7 The DeGroot uncertainty function . . . 587
30.8 Connection to second-order stochastic dominance . . . 590
30.9 Application 1: product quality information . . . 592
30.10 Application 2: sequential experimental design . . . 593
30.11 Summary . . . 596
30.12 The Data Processing Inequality and Coarse-Graining . . . 596
30.13 Relation to Bayesian likelihood-ratio learning . . . 599

31 Information and Market Equilibrium **601**
31.1 Overview . . . 601
31.2 Setup . . . 602
31.3 Price revelation . . . 604
31.4 Invertibility and the elasticity of substitution . . . 605
31.5 Bayesian price expectations in a dynamic economy . . . 612
31.6 Simulating Bayesian learning from prices . . . 614
31.7 Exercises . . . 619

32 Incorrect Models **627**
32.1 Overview . . . 627
32.2 Sampling from compound lottery H . . . 630
32.3 Type 1 agent . . . 632

32.4 What a type 1 agent learns when mixture H generates data . . . 633
32.5 Kullback-Leibler divergence governs limit of π_t . . . 635
32.6 Type 2 agent . . . 640
32.7 Concluding remarks . . . 642
32.8 Exercises . . . 642

33 Bayesian versus Frequentist Decision Rules **647**
33.1 Overview . . . 647
33.2 Setup . . . 648
33.3 Frequentist Decision Rule . . . 651
33.4 Bayesian Decision Rule . . . 657
33.5 Was the Navy Captain's Hunch Correct? . . . 664
33.6 More Details . . . 666
33.7 Distribution of Bayesian Decision Rule's Time to Decide . . . 666
33.8 Probability of Making Correct Decision . . . 670
33.9 Distribution of Likelihood Ratios at Neyman-Pearson's t . . . 672

34 Merging of Opinions: The Blackwell–Dubins Theorem **675**
34.1 Overview . . . 675
34.2 Probability measures on sequence spaces . . . 676
34.3 The likelihood-ratio martingale . . . 678
34.4 The Blackwell–Dubins theorem . . . 679
34.5 The Beta–Bernoulli model . . . 680
34.6 Failure of merging: mutual singularity . . . 688
34.7 Kakutani's theorem: when does merging hold? . . . 689
34.8 Extension to continuous time . . . 694
34.9 Applications . . . 695
34.10 The rate of merging . . . 696
34.11 Summary and extensions . . . 698

35 Survival and Long-Run Dynamics under Recursive Preferences **701**
35.1 Overview . . . 701
35.2 Environment . . . 702
35.3 Planner's problem . . . 703
35.4 Survival conditions . . . 706
35.5 Wealth dynamics decomposition . . . 707
35.6 Survival regions . . . 710
35.7 Three survival channels . . . 713
35.8 Varying the IES . . . 714
35.9 Asymptotic results . . . 715
35.10 The separable case . . . 717
35.11 Asset pricing implications . . . 718
35.12 Optimistic and pessimistic distortions . . . 720
35.13 Long-run consumption distribution . . . 721
35.14 Summary . . . 723

V Linear Programming **725**

36 Optimal Transport **727**
36.1 Overview . . . 727
36.2 The Optimal Transport Problem . . . 728
36.3 The Linear Programming Approach . . . 729
36.4 The Dual Problem . . . 736
36.5 The Python Optimal Transport Package . . . 738

37 Von Neumann Growth Model (and a Generalization) 743
37.1 Notation . . . 748
37.2 Model Ingredients and Assumptions . . . 749
37.3 Dynamic Interpretation . . . 751
37.4 Duality . . . 752
37.5 Interpretation as Two-player Zero-sum Game . . . 753

VI Introduction to Dynamics 761

38 Finite Markov Chains 763
38.1 Overview . . . 763
38.2 Definitions . . . 764
38.3 Simulation . . . 766
38.4 Marginal Distributions . . . 768
38.5 Irreducibility and Aperiodicity . . . 770
38.6 Stationary Distributions . . . 773
38.7 Ergodicity . . . 777
38.8 Computing Expectations . . . 778
38.9 Exercises . . . 779

39 Inventory Dynamics 787
39.1 Overview . . . 787
39.2 Sample paths . . . 788
39.3 Marginal distributions . . . 791
39.4 Exercises . . . 794

40 Linear State Space Models 799
40.1 Overview . . . 799
40.2 The Linear State Space Model . . . 800
40.3 Distributions and Moments . . . 806
40.4 Stationarity and Ergodicity . . . 813
40.5 Noisy Observations . . . 818
40.6 Prediction . . . 819
40.7 Code . . . 820
40.8 Exercises . . . 820

41 Samuelson Multiplier-Accelerator 823
41.1 Overview . . . 823
41.2 Details . . . 825
41.3 Implementation . . . 828
41.4 Stochastic shocks . . . 837
41.5 Government spending . . . 839
41.6 Wrapping everything into a class . . . 841
41.7 Using the LinearStateSpace class . . . 846
41.8 Pure multiplier model . . . 852
41.9 Summary . . . 855

42 The Acceleration Principle and the Nature of Business Cycles 857
42.1 Overview . . . 857
42.2 Empirical foundation for the acceleration principle . . . 859
42.3 Acceleration enables oscillations . . . 860
42.4 A linear system with shocks . . . 863
42.5 From autocovariances to spectra . . . 864
42.6 Spectral peaks in the Hansen-Samuelson model . . . 865

42.7 Real roots can produce peaks in general models . . . 869
42.8 A calibrated model in the frequency domain . . . 872
42.9 Summary . . . 882
42.10 Exercises . . . 883

43 Kesten Processes and Firm Dynamics **891**
43.1 Overview . . . 891
43.2 Kesten processes . . . 892
43.3 Heavy tails . . . 895
43.4 Application: firm dynamics . . . 897
43.5 Exercises . . . 898

44 Wealth Distribution Dynamics **905**
44.1 Overview . . . 905
44.2 Lorenz Curves and the Gini Coefficient . . . 906
44.3 A Model of Wealth Dynamics . . . 909
44.4 Implementation . . . 910
44.5 Applications . . . 913
44.6 Exercises . . . 917

45 A First Look at the Kalman Filter **921**
45.1 Overview . . . 921
45.2 The basic idea . . . 922
45.3 Convergence . . . 929
45.4 Implementation . . . 930
45.5 Exercises . . . 930

46 Another Look at the Kalman Filter **937**
46.1 A worker's output . . . 938
46.2 A firm's wage-setting policy . . . 939
46.3 A state-space representation . . . 939
46.4 An innovations representation . . . 940
46.5 Some computational experiments . . . 941
46.6 Future extensions . . . 953

47 The Kalman Filter and Vector Autoregressions **955**
47.1 Overview . . . 955
47.2 The state space system . . . 956
47.3 The Kalman filter . . . 956
47.4 The Gram-Schmidt process . . . 958
47.5 Hidden Markov model . . . 959
47.6 Estimation . . . 959
47.7 Vector autoregressions and the Kalman filter . . . 960
47.8 Spectral factorization identity . . . 961
47.9 Python implementation . . . 962
47.10 An example . . . 967
47.11 Summary . . . 972
47.12 Exercises . . . 974

48 Organization Capital **981**
48.1 Overview . . . 981
48.2 The basic idea . . . 982
48.3 Personnel information as organization capital . . . 982
48.4 Industry equilibrium . . . 989
48.5 Bayesian screening simulation . . . 991

48.6 Team information . . . 993
48.7 Firm-specific human capital . . . 995
48.8 Costs of transferring organization capital . . . 996
48.9 Summary and implications . . . 998

49 Two Models of Measurements and the Investment Accelerator 1001
49.1 Overview . . . 1001
49.2 The economic model . . . 1003
49.3 Measurement errors . . . 1007
49.4 A classical model of measurements initially collected by an agency . . . 1008
49.5 A model of optimal estimates reported by an agency . . . 1014
49.6 Simulation . . . 1019
49.7 Summary . . . 1025

VII Search 1027

50 Job Search I: The McCall Search Model 1029
50.1 Overview . . . 1030
50.2 The McCall Model . . . 1030
50.3 Computing the Optimal Policy: Take 1 . . . 1033
50.4 Computing an Optimal Policy: Take 2 . . . 1040
50.5 Continuous Offer Distribution . . . 1042
50.6 Volatility . . . 1044
50.7 Exercises . . . 1049

51 Job Search II: Search and Separation 1053
51.1 Overview . . . 1054
51.2 The model . . . 1054
51.3 Solving the model . . . 1055
51.4 Code . . . 1056
51.5 A simplifying transformation . . . 1060
51.6 Implementation . . . 1061
51.7 Impact of parameters . . . 1062
51.8 Exercises . . . 1065

52 Job Search III: Search with Separation and Markov Wages 1069
52.1 Model setup . . . 1070
52.2 Code . . . 1071
52.3 Improving efficiency . . . 1075
52.4 Sensitivity analysis . . . 1077
52.5 Employment simulation . . . 1078
52.6 Ergodic property . . . 1081
52.7 Cross-sectional analysis . . . 1082
52.8 Lower unemployment compensation (c=0.5) . . . 1086
52.9 Exercises . . . 1086

53 Job Search IV: Fitted Value Function Iteration 1089
53.1 Overview . . . 1089
53.2 Model . . . 1090
53.3 Solution method . . . 1090
53.4 Implementation . . . 1093
53.5 Simulation . . . 1098
53.6 Exercises . . . 1105

54 Job Search V: Persistent and Transitory Wage Shocks **1109**
54.1 Overview . . . 1109
54.2 The model . . . 1110
54.3 Implementation . . . 1111
54.4 Unemployment duration . . . 1115
54.5 Exercises . . . 1117

55 Job Search VI: Modeling Career Choice **1119**
55.1 Overview . . . 1119
55.2 Model . . . 1120
55.3 Implementation . . . 1122
55.4 Exercises . . . 1127

56 Job Search VII: On-the-Job Search **1133**
56.1 Overview . . . 1133
56.2 Model . . . 1134
56.3 Implementation . . . 1135
56.4 Solving for Policies . . . 1138
56.5 Exercises . . . 1140

57 Job Search VIII: Search with Learning **1145**
57.1 Overview . . . 1145
57.2 Model . . . 1146
57.3 Take 1: Solution by VFI . . . 1150
57.4 Take 2: A More Efficient Method . . . 1155
57.5 Another Functional Equation . . . 1155
57.6 Solving the RWFE . . . 1155
57.7 Implementation . . . 1156
57.8 Exercises . . . 1157
57.9 Solutions . . . 1157
57.10 Appendix A . . . 1159
57.11 Appendix B . . . 1161
57.12 Examples . . . 1165

VIII Reinforcement Learning **1177**

58 Inventory Management via Q-Learning **1179**
58.1 Introduction . . . 1179
58.2 The Model . . . 1180
58.3 Solving via Value Function Iteration . . . 1181
58.4 Q-Learning . . . 1185

59 Risk-Sensitive Inventory Management via Q-Learning **1193**
59.1 Introduction . . . 1193
59.2 The Model . . . 1194
59.3 Solving via Value Function Iteration . . . 1194
59.4 Interpreting the Outcomes . . . 1199
59.5 Q-Learning . . . 1200
59.6 Conclusion . . . 1207

60 Job Search IX: Search with Q-Learning **1209**
60.1 Overview . . . 1209
60.2 Review of McCall Model . . . 1210
60.3 Implied Quality Function Q . . . 1213

60.4 From Probabilities to Samples . . . 1214
60.5 Q-Learning . . . 1215
60.6 Employed Worker Can't Quit . . . 1223
60.7 Possible Extensions . . . 1224

IX Introduction to Optimal Savings 1225

61 Optimal Savings I: Cake Eating 1227
61.1 Overview . . . 1227
61.2 The model . . . 1228
61.3 The value function . . . 1229
61.4 The optimal policy . . . 1231
61.5 The Euler equation . . . 1232
61.6 Exercises . . . 1235

62 Optimal Savings II: Numerical Cake Eating 1239
62.1 Overview . . . 1239
62.2 Reviewing the Model . . . 1240
62.3 Value Function Iteration . . . 1240
62.4 Exercises . . . 1249

63 Optimal Savings III: Stochastic Returns 1255
63.1 Overview . . . 1255
63.2 The Model . . . 1256
63.3 Computation . . . 1260
63.4 Exercises . . . 1268

64 Optimal Savings IV: Time Iteration 1271
64.1 Overview . . . 1271
64.2 The Euler Equation . . . 1272
64.3 Implementation . . . 1274
64.4 Exercises . . . 1280

65 Optimal Savings V: The Endogenous Grid Method 1283
65.1 Overview . . . 1283
65.2 Key Idea . . . 1284
65.3 Implementation . . . 1285

66 Optimal Savings VI: EGM with JAX 1291
66.1 Overview . . . 1291
66.2 Implementation . . . 1292
66.3 Exercises . . . 1296

X Household Problems 1301

67 The Income Fluctuation Problem I: Discretization and VFI 1303
67.1 Overview . . . 1303
67.2 Set Up . . . 1304
67.3 Code . . . 1305
67.4 Exercises . . . 1309

68 The Income Fluctuation Problem II: Optimistic Policy Iteration 1313
68.1 Overview . . . 1313

68.2 Model and Primitives . . . 1314
68.3 Operators and Policies . . . 1314
68.4 Value Function Iteration . . . 1316
68.5 Optimistic Policy Iteration . . . 1317
68.6 Timing Comparison . . . 1318
68.7 Exercises . . . 1321

69 The Income Fluctuation Problem III: The Endogenous Grid Method 1325
69.1 Overview . . . 1325
69.2 The Household Problem . . . 1326
69.3 Computation . . . 1328
69.4 NumPy Implementation . . . 1329
69.5 JAX Implementation . . . 1332
69.6 Simulation . . . 1340

70 The Income Fluctuation Problem IV: Transient Income Shocks 1343
70.1 Overview . . . 1343
70.2 The Household Problem . . . 1344
70.3 NumPy Implementation . . . 1345
70.4 JAX Implementation . . . 1349
70.5 Simulation . . . 1356
70.6 Wealth Inequality . . . 1358
70.7 Exercises . . . 1361

71 The Income Fluctuation Problem V: Stochastic Returns on Assets 1365
71.1 Overview . . . 1366
71.2 The Model . . . 1366
71.3 Solution Algorithm . . . 1368
71.4 Implementation . . . 1369
71.5 Simulation . . . 1372
71.6 Wealth Inequality . . . 1375
71.7 Exercises . . . 1376

XI LQ Control 1381

72 LQ Control: Foundations 1383
72.1 Overview . . . 1383
72.2 Introduction . . . 1384
72.3 Optimality – finite horizon . . . 1387
72.4 Implementation . . . 1389
72.5 Extensions and comments . . . 1394
72.6 Further applications . . . 1396
72.7 Exercises . . . 1404

73 Lagrangian for LQ Control 1413
73.1 Overview . . . 1413
73.2 Undiscounted LQ DP Problem . . . 1414
73.3 Lagrangian . . . 1415
73.4 State-Costate Dynamics . . . 1416
73.5 Reciprocal Pairs Property . . . 1416
73.6 Schur decomposition . . . 1417
73.7 Application . . . 1418
73.8 Other Applications . . . 1423
73.9 Discounted Problems . . . 1423

74 Eliminating Cross Products **1427**
74.1 Overview . . . 1427
74.2 Undiscounted Dynamic Programming Problem . . . 1427
74.3 Kalman Filter . . . 1428
74.4 Duality table . . . 1429

75 The Permanent Income Model **1431**
75.1 Overview . . . 1431
75.2 The Savings Problem . . . 1432
75.3 Alternative Representations . . . 1439
75.4 Two Classic Examples . . . 1442
75.5 Further Reading . . . 1445
75.6 Appendix: The Euler Equation . . . 1446

76 Permanent Income II: LQ Techniques **1447**
76.1 Overview . . . 1447
76.2 Setup . . . 1448
76.3 The LQ Approach . . . 1450
76.4 Implementation . . . 1451
76.5 Two Example Economies . . . 1454

77 Robust Permanent Income and Pricing **1465**
77.1 Overview . . . 1465
77.2 Risk-sensitive recursive preferences . . . 1466
77.3 A preference for robustness . . . 1469
77.4 The permanent income economy . . . 1469
77.5 Observational equivalence . . . 1470
77.6 Estimation . . . 1473
77.7 Asset pricing and the market price of risk . . . 1476
77.8 A risk-sensitive regulator . . . 1478
77.9 Exercises . . . 1481
77.10 Related lectures . . . 1484

78 Certainty Equivalence **1485**
78.1 Overview . . . 1485
78.2 A central problem of empirical economics . . . 1486
78.3 A formal setup . . . 1486
78.4 Estimated rules are not enough . . . 1487
78.5 An optimization problem . . . 1487
78.6 A linear-quadratic DP problem and certainty equivalence . . . 1488
78.7 A trouble with ad hoc expectations . . . 1491
78.8 Exercises . . . 1491
78.9 Concluding remarks . . . 1492

79 Certainty Equivalence and Model Uncertainty **1493**
79.1 Overview . . . 1493
79.2 Recap: ordinary certainty equivalence . . . 1494
79.3 Model uncertainty and robustness . . . 1496
79.4 Value function under robustness . . . 1498
79.5 Risk-sensitive preferences . . . 1499
79.6 Application: permanent income model . . . 1499
79.7 Summary . . . 1504
79.8 Exercises . . . 1504

80 The LQ Permanent Income Model **1509**

80.1 Overview . . . 1509
80.2 The standard LQ permanent income model . . . 1510
80.3 Exercises . . . 1516

81 Consumption Smoothing with Incomplete and Complete Markets 1523
81.1 Overview . . . 1523
81.2 A brief review . . . 1524
81.3 Spreading consumption cross sections . . . 1525
81.4 A borrowers and lenders economy . . . 1527
81.5 Consumption smoothing with complete markets . . . 1529
81.6 Exercises . . . 1531

82 Robust Consumption Smoothing and Precautionary Savings 1537
82.1 Overview . . . 1537
82.2 A brief review . . . 1538
82.3 A robust permanent income model . . . 1539
82.4 Exercises . . . 1553

83 Production Smoothing via Inventories 1557
83.1 Overview . . . 1557
83.2 Example 1 . . . 1562
83.3 Inventories Not Useful . . . 1563
83.4 Inventories Useful but are Hardwired to be Zero Always . . . 1563
83.5 Example 2 . . . 1565
83.6 Example 3 . . . 1566
83.7 Example 4 . . . 1567
83.8 Example 5 . . . 1569
83.9 Example 6 . . . 1570
83.10 Exercises . . . 1573

XII Bounded Rationality in Macroeconomics 1579

84 A Peculiar Definition of Bounded Rationality 1581
84.1 Overview . . . 1581
84.2 Rational expectations as a fixed point . . . 1584
84.3 Money and prices . . . 1590
84.4 Two currencies . . . 1593
84.5 Where this leaves us . . . 1594
84.6 Exercises . . . 1595

85 Adaptive Agents in an Overlapping Generations Monetary Economy 1599
85.1 Overview . . . 1599
85.2 The environment . . . 1600
85.3 Part 1: a stochastic deficit . . . 1601
85.4 Part 2: a constant deficit and two steady states . . . 1607
85.5 A government learning the Phillips curve . . . 1614
85.6 Concluding remarks . . . 1618
85.7 Exercises . . . 1619

86 Learning, Approximation, and Equilibrium Computation 1627
86.1 Overview . . . 1627
86.2 A continuous-state functional equation . . . 1629
86.3 A benchmark solution . . . 1630
86.4 Parameterized expectations . . . 1631

86.5 Learning is equilibrium computation . . . 1633
86.6 Learning without a functional form . . . 1635
86.7 Concluding remarks . . . 1637
86.8 Exercises . . . 1637

87 Exchange Rate Indeterminacy, Learning, and Experiments 1641
87.1 Overview . . . 1641
87.2 The two-currency economy . . . 1642
87.3 Newton–Raphson learning . . . 1643
87.4 Evidence from the laboratory . . . 1648
87.5 A genetic algorithm economy . . . 1648
87.6 Concluding remarks . . . 1652
87.7 Exercises . . . 1653

88 Genetic Algorithms and Classifier Systems 1657
88.1 Overview . . . 1657
88.2 The perceptron: an agent as a discriminant function . . . 1658
88.3 Associative memory: the Hopfield network . . . 1661
88.4 The genetic algorithm . . . 1664
88.5 Classifier systems . . . 1668
88.6 Evolutionary programming . . . 1670
88.7 Concluding remarks . . . 1671
88.8 Exercises . . . 1671

89 Money as a Medium of Exchange among Artificially Intelligent Agents 1677
89.1 Overview . . . 1677
89.2 The Kiyotaki-Wright environment . . . 1678
89.3 Classifier systems . . . 1680
89.4 Implementation . . . 1683
89.5 Economy A1.1: does a medium of exchange emerge? . . . 1693
89.6 Economy A2.1: when theory predicts speculation . . . 1696
89.7 Economy B.1: a different production pattern . . . 1698
89.8 The genetic algorithm . . . 1700
89.9 Economy A1.2: learning from random rules . . . 1703
89.10 Economy C: fiat money . . . 1705
89.11 Economy D: five goods, five types . . . 1707
89.12 Concluding remarks . . . 1710
89.13 Exercises . . . 1711

90 1993 Prospects for Bounded Rationality in Macroeconomics 1717
90.1 Overview . . . 1717
90.2 The quest, restated . . . 1718
90.3 The debit side: how much we must hard-wire . . . 1718
90.4 Why the econometricians have not returned the compliment . . . 1719
90.5 The credit side: selection, computation, and a returned gift . . . 1724
90.6 Reading the ledger against the quest . . . 1725
90.7 Postscript: what became of the program . . . 1725

XIII Phillips Curve Tradeoffs 1729

91 The Rise and Fall of U.S. Inflation 1731
91.1 Overview . . . 1731
91.2 Facts . . . 1734
91.3 The Phillips curve in the data . . . 1734

91.4 Two interpretations . . . 1738
91.5 Ignoring the Lucas Critique . . . 1739
91.6 A premature summary: triumph or vindication? . . . 1740
91.7 Data patterns after 1999 . . . 1742
91.8 Exercises . . . 1746

92 The Credibility Problem **1749**
92.1 Overview . . . 1749
92.2 A one-period economy . . . 1750
92.3 Nash and Ramsey outcomes . . . 1751
92.4 Best response dynamics . . . 1754
92.5 Least squares learning converges to Nash . . . 1756
92.6 More foresight . . . 1758
92.7 Appendix: stochastic approximation . . . 1758
92.8 Exercises . . . 1759

93 Credible Government Policies **1761**
93.1 Overview . . . 1761
93.2 The repeated economy . . . 1763
93.3 Recursive strategies and promised values . . . 1766
93.4 The Abreu–Pearce–Stacchetti method . . . 1767
93.5 The best and the worst equilibrium . . . 1769
93.6 Examples of recursive equilibria . . . 1771
93.7 Multiplicity . . . 1773
93.8 Numerical example . . . 1775
93.9 Interpretations . . . 1776
93.10 Exercises . . . 1778

94 Adaptive Expectations and the Phelps Problem **1783**
94.1 Overview . . . 1783
94.2 Adaptive expectations . . . 1784
94.3 The Phelps problem . . . 1785
94.4 Disinflation paths . . . 1787
94.5 The general Phelps problem . . . 1789
94.6 Testing the natural-rate hypothesis . . . 1790
94.7 Sacrifice ratios and the subversion of the Phelps model . . . 1790
94.8 Exercises . . . 1791

95 Optimal Misspecified Beliefs **1793**
95.1 Overview . . . 1793
95.2 An experiment in Bray's lab . . . 1794
95.3 Optimal misspecification . . . 1795
95.4 Comparing the true and forecasting models . . . 1797
95.5 Lessons . . . 1799
95.6 Exercises . . . 1801

96 Self-Confirming Equilibria **1805**
96.1 Overview . . . 1805
96.2 Objects in the Phelps problem . . . 1806
96.3 The actual Phillips curve . . . 1806
96.4 Self-confirming equilibria . . . 1806
96.5 The special case solved by hand . . . 1807
96.6 Equilibrium with misspecified beliefs . . . 1810
96.7 Exercises . . . 1816

97 Adaptive Learning and Escape Dynamics **1819**
97.1 Overview . . . 1819
97.2 A primer on recursive algorithms . . . 1820
97.3 The adaptive model . . . 1826
97.4 Least squares learning converges . . . 1829
97.5 Constant gain and escape dynamics . . . 1830
97.6 The escape route and the induction hypothesis . . . 1831
97.7 Relation to equilibria under forecast misspecification . . . 1834
97.8 Role of the discount factor . . . 1834
97.9 Anticipated utility . . . 1835
97.10 Conclusions . . . 1835
97.11 Exercises . . . 1836

98 Escaping Nash Inflation **1839**
98.1 Overview . . . 1839
98.2 The model . . . 1840
98.3 The self-confirming equilibrium . . . 1841
98.4 Adaptation and the mean dynamics . . . 1843
98.5 Escape dynamics as a control problem . . . 1843
98.6 The dominant escape path . . . 1844
98.7 The race of four ODEs . . . 1846
98.8 Mean dynamics reinforce the escape . . . 1847
98.9 The experimentation trap . . . 1849
98.10 Escape frequency and model richness . . . 1849
98.11 Escaping volatile inflation . . . 1849
98.12 Exercises . . . 1851

99 Priors, Escapes, and Learning Cycles **1853**
99.1 Overview . . . 1853
99.2 The static model . . . 1854
99.3 The self-confirming equilibrium . . . 1855
99.4 Drifting beliefs and the Kalman filter . . . 1856
99.5 Mean dynamics and E-stability . . . 1857
99.6 Learning cycles . . . 1858
99.7 Escape dynamics and the direction of escape . . . 1860
99.8 Sims's nonconvergence . . . 1861
99.9 Conclusion . . . 1865
99.10 Exercises . . . 1866

100 The Lost Conquest: Fed Policy in the 2020s **1869**
100.1 Overview . . . 1869
100.2 The three elements . . . 1870
100.3 The Fed's drifting-coefficients beliefs . . . 1871
100.4 The Fed's Phelps problem . . . 1873
100.5 Why the Fed was slow: a counterfactual . . . 1875
100.6 A self-confirming equilibrium: nature's New Keynesian model . . . 1876
100.7 Exercises . . . 1878

101 Drifts and Volatilities **1881**
101.1 Overview . . . 1881
101.2 Bad policy or bad luck? . . . 1882
101.3 A VAR with drifting coefficients and stochastic volatility . . . 1883
101.4 The data . . . 1885
101.5 Priors . . . 1888
101.6 A Metropolis-within-Gibbs sampler . . . 1890

101.7 What the data say . . . 1899
101.8 Another quarter-century of evidence . . . 1923
101.9 Bad policy or bad luck? A verdict . . . 1950
101.10 Exercises . . . 1950

XIV Optimal Growth 1955

102 Cass-Koopmans Model 1957
102.1 Overview . . . 1957
102.2 The Model . . . 1958
102.3 Planning Problem . . . 1959
102.4 Shooting Algorithm . . . 1963
102.5 Setting Initial Capital to Steady State Capital . . . 1967
102.6 A Turnpike Property . . . 1969
102.7 A Limiting Infinite Horizon Economy . . . 1972
102.8 Stable Manifold and Phase Diagram . . . 1973
102.9 Concluding Remarks . . . 1975

103 Cass-Koopmans Competitive Equilibrium 1979
103.1 Overview . . . 1979
103.2 Review of Cass-Koopmans Model . . . 1980
103.3 Competitive Equilibrium . . . 1981
103.4 Market Structure . . . 1982
103.5 Firm Problem . . . 1982
103.6 Household Problem . . . 1983
103.7 Computing a Competitive Equilibrium . . . 1985
103.8 Yield Curves and Hicks-Arrow Prices . . . 1993

104 Cass-Koopmans Model with Distorting Taxes 1995
104.1 Overview . . . 1995
104.2 The Economy . . . 1995
104.3 Equilibrium . . . 1997
104.4 No-arbitrage Condition . . . 1997
104.5 Household's First Order Condition . . . 1998
104.6 Computing Equilibria . . . 1999
104.7 Python Code . . . 1999
104.8 Some functional forms . . . 2003
104.9 Computation . . . 2004
104.10 Exogenous growth . . . 2027

105 Two-Country Model with Distorting Taxes 2039
105.1 Overview . . . 2039
105.2 A Two-Country Cass-Koopmans Model . . . 2039

XV Multiple Agent Models 2053

106 A Lake Model of Employment and Unemployment 2055
106.1 Overview . . . 2055
106.2 The model . . . 2056
106.3 Implementation . . . 2058
106.4 Dynamics of an individual worker . . . 2065
106.5 Exercises . . . 2068

107 Lake Model with an Endogenous Job Finding Rate **2075**
107.1 Overview . . . 2075
107.2 Set Up . . . 2076
107.3 Fiscal policy . . . 2081
107.4 Exercises . . . 2085

108 Rational Expectations Equilibrium **2089**
108.1 Overview . . . 2089
108.2 Rational Expectations Equilibrium . . . 2092
108.3 Computing an Equilibrium . . . 2095
108.4 Exercises . . . 2097

109 Stability in Linear Rational Expectations Models **2103**
109.1 Overview . . . 2103
109.2 Linear Difference Equations . . . 2104
109.3 Illustration: Cagan's Model . . . 2106
109.4 Some Python Code . . . 2107
109.5 Alternative Code . . . 2109
109.6 Another Perspective . . . 2111
109.7 Log money Supply Feeds Back on Log Price Level . . . 2114
109.8 Big P, Little p Interpretation . . . 2117
109.9 Fun with SymPy . . . 2119

110 Markov Perfect Equilibrium **2123**
110.1 Overview . . . 2123
110.2 Background . . . 2124
110.3 Linear Markov Perfect Equilibria . . . 2125
110.4 Application . . . 2128
110.5 Exercises . . . 2131

111 Uncertainty Traps **2141**
111.1 Overview . . . 2141
111.2 The Model . . . 2142
111.3 Implementation . . . 2145
111.4 Results . . . 2146
111.5 Exercises . . . 2147

112 The Aiyagari Model **2155**
112.1 Overview . . . 2155
112.2 The Economy . . . 2156
112.3 Implementation . . . 2158
112.4 Exercises . . . 2168

113 The Aiyagari Model with Endogenous Grid Method **2175**
113.1 Overview . . . 2175
113.2 The Economy . . . 2176
113.3 Implementation with EGM . . . 2178
113.4 Computing Aggregate Capital by Simulation . . . 2182
113.5 Computing Equilibrium . . . 2183
113.6 Wealth Distribution . . . 2185
113.7 Summary and Comparison . . . 2187
113.8 Exercises . . . 2188

114 Transitions in an Overlapping Generations Model **2189**
114.1 Introduction . . . 2189

114.2 Setting . . . 2190
114.3 Production . . . 2191
114.4 Government . . . 2191
114.5 Activities in Factor Markets . . . 2191
114.6 Representative firm's problem . . . 2192
114.7 Individuals' problems . . . 2192
114.8 Equilbrium . . . 2193
114.9 Next steps . . . 2194
114.10 Closed form solution . . . 2194
114.11 A computational strategy . . . 2206

115 A Long-Lived, Heterogeneous Agents, Overlapping Generations Model 2217
115.1 Overview . . . 2217
115.2 Environment . . . 2218
115.3 Production . . . 2219
115.4 Government . . . 2219
115.5 Activities in factor markets . . . 2219
115.6 Representative firm's problem . . . 2220
115.7 Households' problems . . . 2221
115.8 Population dynamics . . . 2221
115.9 Equilibrium . . . 2221
115.10 Implementation . . . 2222
115.11 Computing a steady state . . . 2225
115.12 Transition dynamics . . . 2232
115.13 Experiment 1: Immediate tax cut . . . 2237
115.14 Experiment 2: Preannounced tax cut . . . 2242

116 Two Computations to Fund Social Security 2247
116.1 Overview . . . 2248
116.2 Model . . . 2249
116.3 Two experiments . . . 2254
116.4 Computation strategy . . . 2255
116.5 Calibration . . . 2257
116.6 Individual optimality . . . 2262
116.7 Transition path computation . . . 2271
116.8 Experiment 1: compensation through debt . . . 2283
116.9 Experiment 2: government capital accumulation . . . 2297
116.10 Distribution surfaces . . . 2303

XVI Asset Pricing and Finance 2311

117 Asset Pricing: Finite State Models 2313
117.1 Overview . . . 2313
117.2 Pricing Models . . . 2314
117.3 Prices in the Risk-Neutral Case . . . 2315
117.4 Risk Aversion and Asset Prices . . . 2320
117.5 Exercises . . . 2329

118 Stochastic Consumption, Risk Aversion, and the Temporal Behavior of Asset Returns 2335
118.1 Overview . . . 2336
118.2 Euler equation . . . 2336
118.3 The Euler equation under lognormality . . . 2337
118.4 The restricted system and its likelihood . . . 2338
118.5 Likelihood implementation . . . 2340

118.6 Preference parameters and likelihood ratio tests . . . 2351
118.7 Predictability and the R-squared restriction . . . 2354
118.8 Return-difference tests . . . 2355
118.9 Empirical MLE estimation . . . 2357
118.10 Residual diagnostics . . . 2361
118.11 Connection to the equity premium puzzle . . . 2363
118.12 Another approach . . . 2364

119 Estimating Euler Equations by Generalized Method of Moments **2365**
119.1 Overview . . . 2365
119.2 The economic model . . . 2366
119.3 From conditional to unconditional moments . . . 2367
119.4 GMM criterion and asymptotic theory . . . 2373
119.5 Covariance estimation and the choice of instruments . . . 2374
119.6 Empirical GMM estimation . . . 2381
119.7 Summary . . . 2383

120 Competitive Equilibria with Arrow Securities **2385**
120.1 Introduction . . . 2385
120.2 The setting . . . 2386
120.3 Recursive Formulation . . . 2387
120.4 State Variable Degeneracy . . . 2388
120.5 Markov Asset Prices . . . 2388
120.6 General Equilibrium . . . 2390
120.7 Finite Horizon . . . 2394
120.8 Python Code . . . 2395
120.9 Examples . . . 2399

121 Heterogeneous Beliefs and Bubbles **2411**
121.1 Overview . . . 2411
121.2 Structure of the Model . . . 2412
121.3 Solving the Model . . . 2414

122 Speculative Behavior with Bayesian Learning **2423**
122.1 Overview . . . 2423
122.2 Structure of the model . . . 2424
122.3 Information and beliefs . . . 2425
122.4 Source of heterogeneous priors . . . 2425
122.5 Beta priors . . . 2426
122.6 Market prices with learning . . . 2427
122.7 Two Traders . . . 2428
122.8 Concluding remarks . . . 2435
122.9 Exercise . . . 2435

123 Rational Learning and Rational Expectations **2439**
123.1 Overview . . . 2439
123.2 The economy . . . 2440
123.3 The rational learning equilibrium . . . 2442
123.4 Computing the equilibrium . . . 2443
123.5 Posterior concentration . . . 2446
123.6 Equilibrium trades and prices . . . 2448
123.7 Convergence of posterior assessments . . . 2449
123.8 Sharpening the convergence result . . . 2450
123.9 Obstacles to convergence . . . 2452
123.10 Learning within versus learning about a rational expectations equilibrium . . . 2453

123.11 Summary . . . 2454
123.12 Exercises . . . 2454

124 Least Squares Learning in Self-Referential Models **2459**
124.1 Overview . . . 2459
124.2 The self-referential structure . . . 2460
124.3 Why a differential equation governs the limit . . . 2461
124.4 The governing ODE . . . 2462
124.5 Five examples . . . 2467
124.6 Example 1: Ordinary linear stochastic difference equations . . . 2467
124.7 Example 2: Bray's cobweb model . . . 2468
124.8 Counterexample: Unstable Bray feedback . . . 2471
124.9 Example 3: Bray-Savin supply-shifter model . . . 2473
124.10 Example 4: Hyperinflation / asset prices . . . 2476
124.11 Example 5: Investment under uncertainty . . . 2479
124.12 Connection to rational learning . . . 2482
124.13 Summary . . . 2482
124.14 Exercises . . . 2482

125 Affine Models of Asset Prices **2489**
125.1 Overview . . . 2489
125.2 The model . . . 2490
125.3 Pricing risky assets . . . 2492
125.4 Affine term structure of yields . . . 2494
125.5 Risk premiums . . . 2502
125.6 Risk-neutral probabilities . . . 2505
125.7 Distorted beliefs . . . 2507
125.8 Concluding remarks . . . 2510

126 The Recovery Theorem **2511**
126.1 Overview . . . 2511
126.2 Model setup . . . 2512
126.3 The recovery theorem . . . 2515
126.4 Numerical example . . . 2518
126.5 Natural vs. risk-neutral distributions . . . 2522
126.6 Effect of risk aversion . . . 2524
126.7 Recovering the discount rate . . . 2525
126.8 Tail risk: natural vs. risk-neutral probabilities of catastrophe . . . 2526
126.9 From option prices to transition prices . . . 2527
126.10 Testing efficient markets . . . 2528
126.11 Limitations and extensions . . . 2529
126.12 Exercises . . . 2530

127 Long-term Risk: An Operator Approach **2535**
127.1 Overview . . . 2535
127.2 Multiplicative functionals . . . 2537
127.3 Semigroups . . . 2539
127.4 The generator . . . 2542
127.5 Principal eigenfunctions . . . 2546
127.6 A finite-state Markov chain . . . 2549
127.7 The affine diffusion example . . . 2557
127.8 Long-run risk prices . . . 2566
127.9 Assumptions behind the scenes . . . 2575
127.10 Summary . . . 2577
127.11 Exercises . . . 2577

128 Misspecified Recovery **2587**
128.1 Overview . . . 2587
128.2 Three transition matrices . . . 2588
128.3 Martingale Component . . . 2592
128.4 From matrices to the general framework . . . 2594
128.5 When the recovery fails . . . 2600
128.6 Additional state vector . . . 2615
128.7 Measuring the martingale component . . . 2617
128.8 Lessons . . . 2617
128.9 Exercises . . . 2617

XVII Data and Empirics 2621

129 Pandas for Panel Data **2623**
129.1 Overview . . . 2623
129.2 Slicing and Reshaping Data . . . 2624
129.3 Merging Dataframes and Filling NaNs . . . 2629
129.4 Grouping and Summarizing Data . . . 2634
129.5 Final Remarks . . . 2640
129.6 Exercises . . . 2640

130 Linear Regression in Python **2645**
130.1 Overview . . . 2645
130.2 Simple Linear Regression . . . 2646
130.3 Extending the Linear Regression Model . . . 2652
130.4 Endogeneity . . . 2653
130.5 Summary . . . 2658
130.6 Exercises . . . 2658

131 Maximum Likelihood Estimation **2661**
131.1 Overview . . . 2661
131.2 Set up and assumptions . . . 2662
131.3 Conditional distributions . . . 2666
131.4 Maximum likelihood estimation . . . 2667
131.5 MLE with numerical methods . . . 2669
131.6 Maximum likelihood estimation with `statsmodels` . . . 2675
131.7 Summary . . . 2679
131.8 Exercises . . . 2680

132 A Linear Model of Unemployment **2683**
132.1 Overview . . . 2683
132.2 The natural rate versus hysteresis . . . 2684
132.3 The data . . . 2684
132.4 A linear model of unemployment . . . 2686
132.5 A random walk would wander off . . . 2687
132.6 Monthly versus annual . . . 2689
132.7 What the model misses . . . 2689
132.8 Exercises . . . 2693

133 Unemployment Dynamics with Asymmetric Shocks **2697**
133.1 The data . . . 2698
133.2 A model with asymmetric shocks . . . 2698
133.3 Bayesian estimation . . . 2701
133.4 Comparing models with cross-validation . . . 2703

133.5 Conclusion . . . 2707
133.6 Exercises . . . 2707

XVIII Auctions 2709

134 First-Price and Second-Price Auctions 2711
134.1 First-price sealed-bid auction (FPSB) . . . 2711
134.2 Second-price sealed-bid auction (SPSB) . . . 2712
134.3 Characterization of SPSB auction . . . 2712
134.4 Uniform distribution of private values . . . 2713
134.5 Setup . . . 2713
134.6 First price sealed bid auction . . . 2713
134.7 Second price sealed bid auction . . . 2714
134.8 Python code . . . 2714
134.9 Revenue equivalence theorem . . . 2716
134.10 Calculation of bid price in FPSB . . . 2718
134.11 χ^2 Distribution . . . 2720
134.12 Code summary . . . 2723
134.13 References . . . 2727

135 Multiple Good Allocation Mechanisms 2729
135.1 Overview . . . 2729
135.2 Ascending Bids Auction for Multiple Goods . . . 2729
135.3 A Benevolent Planner . . . 2730
135.4 Equivalence of Allocations . . . 2730
135.5 Ascending Bid Auction . . . 2730
135.6 Pseudocode . . . 2731
135.7 An Example . . . 2733
135.8 A Python Class . . . 2741
135.9 Robustness Checks . . . 2749
135.10 A Groves-Clarke Mechanism . . . 2761
135.11 An Example Solved by Hand . . . 2762
135.12 Another Python Class . . . 2765

XIX Other 2773

136 Troubleshooting 2775
136.1 Fixing Your Local Environment . . . 2775
136.2 Reporting an Issue . . . 2776

137 References 2777

138 Execution Statistics 2779

Bibliography 2785

Proof Index 2803

Index 2809

This website presents a set of lectures on quantitative economic modeling.

- Tools and Techniques
  - *Modeling COVID 19*
  - *Linear Algebra*
  - *QR Decomposition*
  - *Circulant Matrices*
  - *Singular Value Decomposition (SVD)*
  - *VARs and DMDs*
  - *Using Newton's Method to Solve Economic Models*
- Elementary Statistics
  - *Elementary Probability with Matrices*
  - *Some Probability Distributions*
  - *LLN and CLT*
  - *Two Meanings of Probability*
  - *Multivariate Hypergeometric Distribution*
  - *Multivariate Normal Distribution*
  - *Fault Tree Uncertainties*
  - *Introduction to Artificial Neural Networks*
  - *Randomized Response Surveys*
  - *Expected Utilities of Random Responses*
- Bayes Law
  - *Non-Conjugate Priors*
  - *Posterior Distributions for AR(1) Parameters*
  - *Forecasting an AR(1) Process*
- Statistics and Information
  - *Statistical Divergence Measures*
  - *Likelihood Ratio Processes*
  - *Heterogeneous Beliefs and Financial Markets*
  - *Likelihood Processes For VAR Models*
  - *Mean of a Likelihood Ratio Process*
  - *A Problem that Stumped Milton Friedman*
  - *A Bayesian Formulation of Friedman and Wald's Problem*
  - *Exchangeability and Bayesian Updating*
  - *Likelihood Ratio Processes and Bayesian Learning*
  - *Blackwell's Theorem on Comparing Experiments*
  - *Information and Market Equilibrium*

  - *Incorrect Models*
  - *Bayesian versus Frequentist Decision Rules*
  - *Merging of Opinions: The Blackwell–Dubins Theorem*
  - *Survival and Long-Run Dynamics under Recursive Preferences*
- Linear Programming
  - *Optimal Transport*
  - *Von Neumann Growth Model (and a Generalization)*
- Introduction to Dynamics
  - *Finite Markov Chains*
  - *Inventory Dynamics*
  - *Linear State Space Models*
  - *Samuelson Multiplier-Accelerator*
  - *The Acceleration Principle and the Nature of Business Cycles*
  - *Kesten Processes and Firm Dynamics*
  - *Wealth Distribution Dynamics*
  - *A First Look at the Kalman Filter*
  - *Another Look at the Kalman Filter*
  - *The Kalman Filter and Vector Autoregressions*
  - *Organization Capital*
  - *Two Models of Measurements and the Investment Accelerator*
- Search
  - *Job Search I: The McCall Search Model*
  - *Job Search II: Search and Separation*
  - *Job Search III: Search with Separation and Markov Wages*
  - *Job Search IV: Fitted Value Function Iteration*
  - *Job Search V: Persistent and Transitory Wage Shocks*
  - *Job Search VI: Modeling Career Choice*
  - *Job Search VII: On-the-Job Search*
  - *Job Search VIII: Search with Learning*
- Reinforcement Learning
  - *Inventory Management via Q-Learning*
  - *Risk-Sensitive Inventory Management via Q-Learning*
  - *Job Search IX: Search with Q-Learning*
- Introduction to Optimal Savings
  - *Optimal Savings I: Cake Eating*
  - *Optimal Savings II: Numerical Cake Eating*

  - *Optimal Savings III: Stochastic Returns*
  - *Optimal Savings IV: Time Iteration*
  - *Optimal Savings V: The Endogenous Grid Method*
  - *Optimal Savings VI: EGM with JAX*
- Household Problems
  - *The Income Fluctuation Problem I: Discretization and VFI*
  - *The Income Fluctuation Problem II: Optimistic Policy Iteration*
  - *The Income Fluctuation Problem III: The Endogenous Grid Method*
  - *The Income Fluctuation Problem IV: Transient Income Shocks*
  - *The Income Fluctuation Problem V: Stochastic Returns on Assets*
- LQ Control
  - *LQ Control: Foundations*
  - *Lagrangian for LQ Control*
  - *Eliminating Cross Products*
  - *The Permanent Income Model*
  - *Permanent Income II: LQ Techniques*
  - *Robust Permanent Income and Pricing*
  - *Certainty Equivalence*
  - *Certainty Equivalence and Model Uncertainty*
  - *The LQ Permanent Income Model*
  - *Consumption Smoothing with Incomplete and Complete Markets*
  - *Robust Consumption Smoothing and Precautionary Savings*
  - *Production Smoothing via Inventories*
- Bounded Rationality in Macroeconomics
  - *A Peculiar Definition of Bounded Rationality*
  - *Adaptive Agents in an Overlapping Generations Monetary Economy*
  - *Learning, Approximation, and Equilibrium Computation*
  - *Exchange Rate Indeterminacy, Learning, and Experiments*
  - *Genetic Algorithms and Classifier Systems*
  - *Money as a Medium of Exchange among Artificially Intelligent Agents*
  - *1993 Prospects for Bounded Rationality in Macroeconomics*
- Phillips Curve Tradeoffs
  - *The Rise and Fall of U.S. Inflation*
  - *The Credibility Problem*
  - *Credible Government Policies*
  - *Adaptive Expectations and the Phelps Problem*

  - *Optimal Misspecified Beliefs*
  - *Self-Confirming Equilibria*
  - *Adaptive Learning and Escape Dynamics*
  - *Escaping Nash Inflation*
  - *Priors, Escapes, and Learning Cycles*
  - *The Lost Conquest: Fed Policy in the 2020s*
  - *Drifts and Volatilities*
- Optimal Growth
  - *Cass-Koopmans Model*
  - *Cass-Koopmans Competitive Equilibrium*
  - *Cass-Koopmans Model with Distorting Taxes*
  - *Two-Country Model with Distorting Taxes*
- Multiple Agent Models
  - *A Lake Model of Employment and Unemployment*
  - *Lake Model with an Endogenous Job Finding Rate*
  - *Rational Expectations Equilibrium*
  - *Stability in Linear Rational Expectations Models*
  - *Markov Perfect Equilibrium*
  - *Uncertainty Traps*
  - *The Aiyagari Model*
  - *The Aiyagari Model with Endogenous Grid Method*
  - *Transitions in an Overlapping Generations Model*
  - *A Long-Lived, Heterogeneous Agents, Overlapping Generations Model*
  - *Two Computations to Fund Social Security*
- Asset Pricing and Finance
  - *Asset Pricing: Finite State Models*
  - *Stochastic Consumption, Risk Aversion, and the Temporal Behavior of Asset Returns*
  - *Estimating Euler Equations by Generalized Method of Moments*
  - *Competitive Equilibria with Arrow Securities*
  - *Heterogeneous Beliefs and Bubbles*
  - *Speculative Behavior with Bayesian Learning*
  - *Rational Learning and Rational Expectations*
  - *Least Squares Learning in Self-Referential Models*
  - *Affine Models of Asset Prices*
  - *The Recovery Theorem*
  - *Long-term Risk: An Operator Approach*

  - *Misspecified Recovery*
- Data and Empirics
  - *Pandas for Panel Data*
  - *Linear Regression in Python*
  - *Maximum Likelihood Estimation*
  - *A Linear Model of Unemployment*
  - *Unemployment Dynamics with Asymmetric Shocks*
- Auctions
  - *First-Price and Second-Price Auctions*
  - *Multiple Good Allocation Mechanisms*
- Other
  - *Troubleshooting*
  - *References*
  - *Execution Statistics*

Part I

Tools and Techniques

CHAPTER

ONE

MODELING COVID 19

Contents

- *Modeling COVID 19*
  - *Overview*
  - *The SIR Model*
  - *Implementation*
  - *Experiments*
  - *Ending Lockdown*

1.1 Overview

This is a Python version of the code for analyzing the COVID-19 pandemic provided by Andrew Atkeson.

See, in particular

- NBER Working Paper No. 26867
- COVID-19 Working papers and code

The purpose of his notes is to introduce economists to quantitative modeling of infectious disease dynamics.

Dynamics are modeled using a standard SIR (Susceptible-Infected-Removed) model of disease spread.

The model dynamics are represented by a system of ordinary differential equations.

The main objective is to study the impact of suppression through social distancing on the spread of the infection.

The focus is on US outcomes but the parameters can be adjusted to study other countries.

We will use the following standard imports:

```
import matplotlib.pyplot as plt
import numpy as np
from numpy import exp
```

We will also use SciPy's numerical routine odeint for solving differential equations.

```
from scipy.integrate import odeint
```

This routine calls into compiled code from the FORTRAN library odepack.

1.2 The SIR Model

In the version of the SIR model we will analyze there are four states.

All individuals in the population are assumed to be in one of these four states.

The states are: susceptible (S), exposed (E), infected (I) and removed ®.

Comments:

- Those in state R have been infected and either recovered or died.
- Those who have recovered are assumed to have acquired immunity.
- Those in the exposed group are not yet infectious.

1.2.1 Time Path

The flow across states follows the path $S \to E \to I \to R$.

All individuals in the population are eventually infected when the transmission rate is positive and $i(0) > 0$.

The interest is primarily in

- the number of infections at a given time (which determines whether or not the health care system is overwhelmed) and
- how long the caseload can be deferred (hopefully until a vaccine arrives)

Using lower case letters for the fraction of the population in each state, the dynamics are

$$\begin{aligned} \dot{s}(t) &= -\beta(t)\, s(t)\, i(t) \\ \dot{e}(t) &= \beta(t)\, s(t)\, i(t) - \sigma e(t) \\ \dot{i}(t) &= \sigma e(t) - \gamma i(t) \end{aligned} \tag{1.1}$$

In these equations,

- $\beta(t)$ is called the **transmission rate** (the rate at which individuals bump into others and expose them to the virus).
- σ is called the **infection rate** (the rate at which those who are exposed become infected)
- γ is called the **recovery rate** (the rate at which infected people recover or die).
- the dot symbol $\dot{y}$ represents the time derivative dy/dt.

We do not need to model the fraction r of the population in state R separately because the states form a partition.

In particular, the "removed" fraction of the population is $r = 1 - s - e - i$.

We will also track $c = i + r$, which is the cumulative caseload (i.e., all those who have or have had the infection).

The system (1.1) can be written in vector form as

$$\dot{x} = F(x, t), \qquad x := (s, e, i) \tag{1.2}$$

for suitable definition of F (see the code below).

1.2.2 Parameters

Both σ and γ are thought of as fixed, biologically determined parameters.

As in Atkeson's note, we set

- $\sigma = 1/5.2$ to reflect an average incubation period of 5.2 days.
- $\gamma = 1/18$ to match an average illness duration of 18 days.

The transmission rate is modeled as

- $\beta(t) := R(t)\gamma$ where $R(t)$ is the **effective reproduction number** at time t.

(The notation is slightly confusing, since $R(t)$ is different to R, the symbol that represents the removed state.)

1.3 Implementation

First we set the population size to match the US.

```
pop_size = 3.3e8
```

Next we fix parameters as described above.

```
γ = 1 / 18
σ = 1 / 5.2
```

Now we construct a function that represents F in (1.2)

```
def F(x, t, R0=1.6):
    """
    Time derivative of the state vector.

        * x is the state vector (array_like)
        * t is time (scalar)
        * R0 is the effective transmission rate, defaulting to a constant

    """
    s, e, i = x

    # New exposure of susceptibles
    β = R0(t) * γ if callable(R0) else R0 * γ
    ne = β * s * i

    # Time derivatives
    ds = - ne
    de = ne - σ * e
    di = σ * e - γ * i

    return ds, de, di
```

Note that `R0` can be either constant or a given function of time.

The initial conditions are set to

```
# initial conditions of s, e, i
i_0 = 1e-7
```

(continues on next page)

(continued from previous page)

```
e_0 = 4 * i_0
s_0 = 1 - i_0 - e_0
```

In vector form the initial condition is

```
x_0 = s_0, e_0, i_0
```

We solve for the time path numerically using odeint, at a sequence of dates `t_vec`.

```
def solve_path(R0, t_vec, x_init=x_0):
    """
    Solve for i(t) and c(t) via numerical integration,
    given the time path for R0.

    """
    G = lambda x, t: F(x, t, R0)
    s_path, e_path, i_path = odeint(G, x_init, t_vec).transpose()

    c_path = 1 - s_path - e_path       # cumulative cases
    return i_path, c_path
```

1.4 Experiments

Let's run some experiments using this code.

The time period we investigate will be 550 days, or around 18 months:

```
t_length = 550
grid_size = 1000
t_vec = np.linspace(0, t_length, grid_size)
```

1.4.1 Experiment 1: Constant R0 Case

Let's start with the case where `R0` is constant.

We calculate the time path of infected people under different assumptions for `R0`:

```
R0_vals = np.linspace(1.6, 3.0, 6)
labels = [f'$R0 = {r:.2f}$' for r in R0_vals]
i_paths, c_paths = [], []

for r in R0_vals:
    i_path, c_path = solve_path(r, t_vec)
    i_paths.append(i_path)
    c_paths.append(c_path)
```

Here's some code to plot the time paths.

```
def plot_paths(paths, labels, times=t_vec):

    fig, ax = plt.subplots()

    for path, label in zip(paths, labels):
```

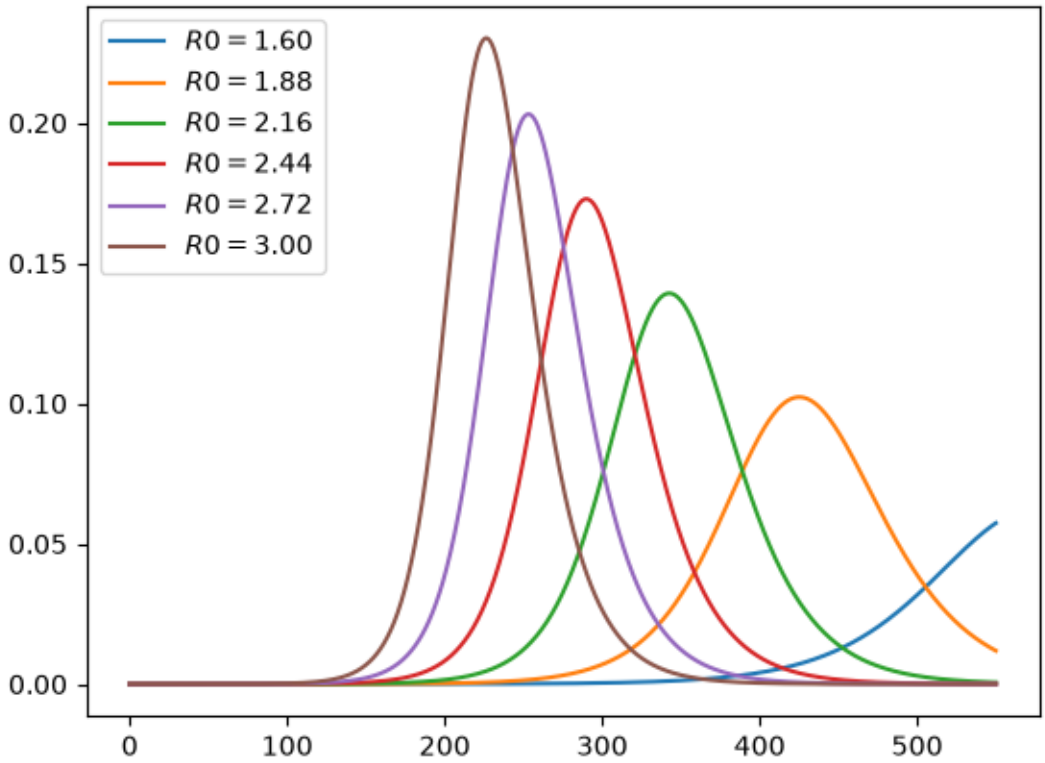
(continues on next page)

(continued from previous page)

```
        ax.plot(times, path, label=label)

    ax.legend(loc='upper left')

    plt.show()
```

Let's plot current cases as a fraction of the population.

```
plot_paths(i_paths, labels)
```

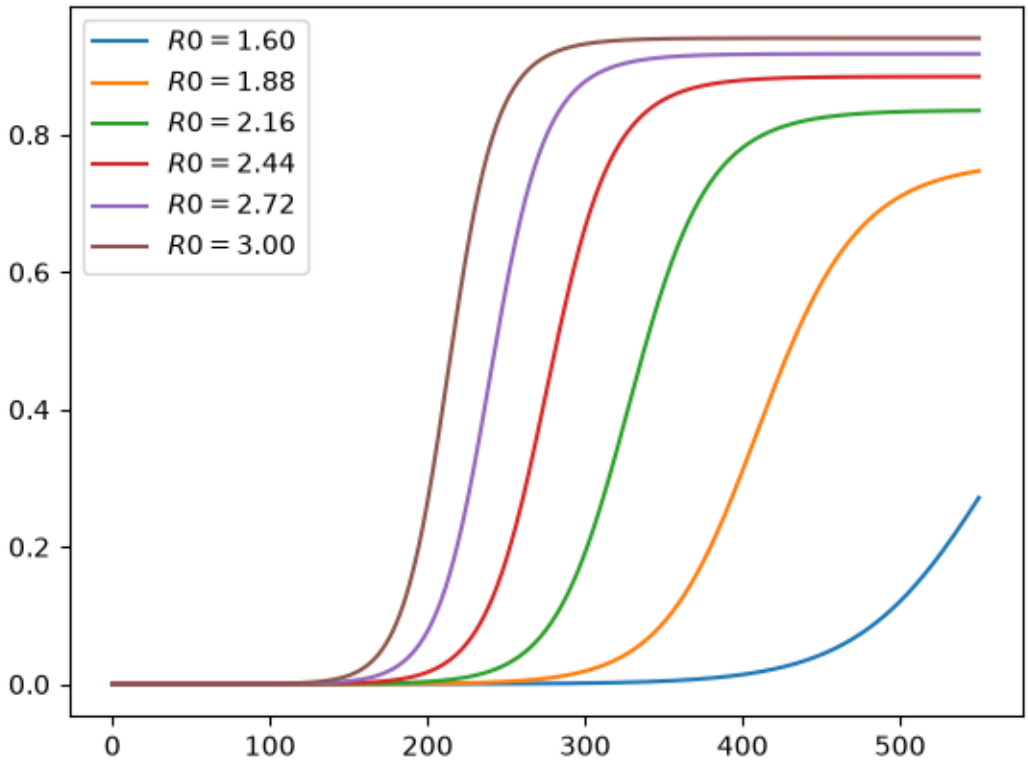


As expected, lower effective transmission rates defer the peak of infections.

They also lead to a lower peak in current cases.

Here are cumulative cases, as a fraction of population:

```
plot_paths(c_paths, labels)
```



1.4.2 Experiment 2: Changing Mitigation

Let's look at a scenario where mitigation (e.g., social distancing) is successively imposed.

Here's a specification for `R0` as a function of time.

```
def R0_mitigating(t, r0=3, η=1, r_bar=1.6):
    R0 = r0 * exp(- η * t) + (1 - exp(- η * t)) * r_bar
    return R0
```

The idea is that `R0` starts off at 3 and falls to 1.6.

This is due to progressive adoption of stricter mitigation measures.

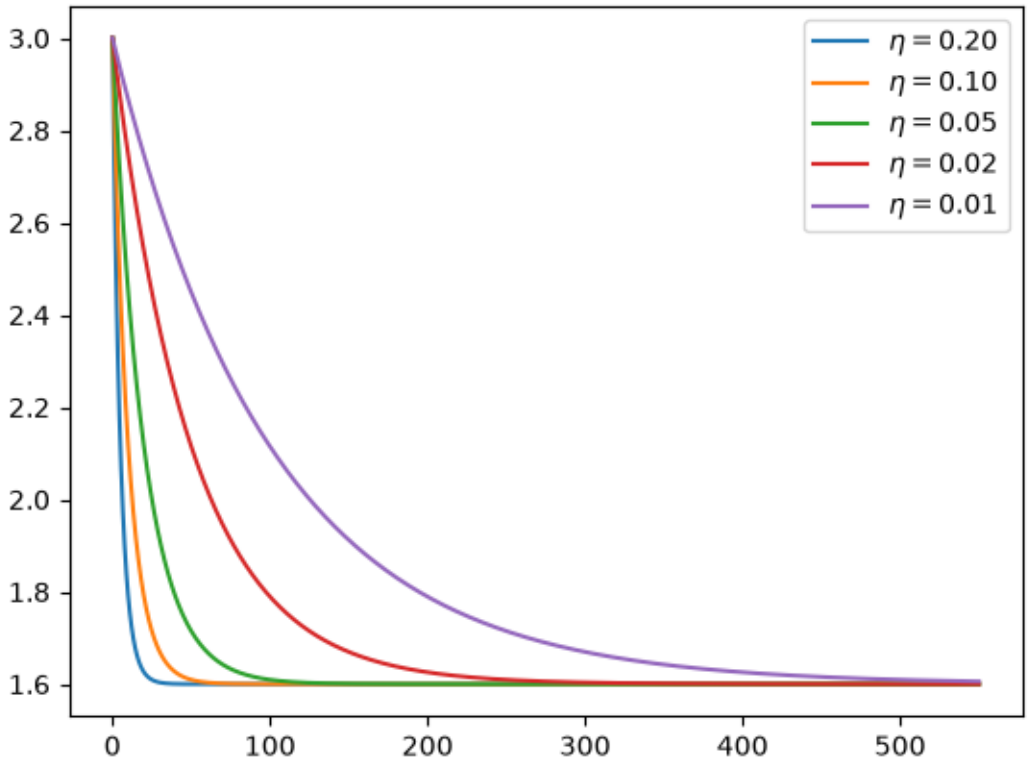
The parameter `η` controls the rate, or the speed at which restrictions are imposed.

We consider several different rates:

```
η_vals = 1/5, 1/10, 1/20, 1/50, 1/100
labels = [fr'$\eta = {η:.2f}$' for η in η_vals]
```

This is what the time path of `R0` looks like at these alternative rates:

```
fig, ax = plt.subplots()

for η, label in zip(η_vals, labels):
    ax.plot(t_vec, R0_mitigating(t_vec, η=η), label=label)

ax.legend()
plt.show()
```

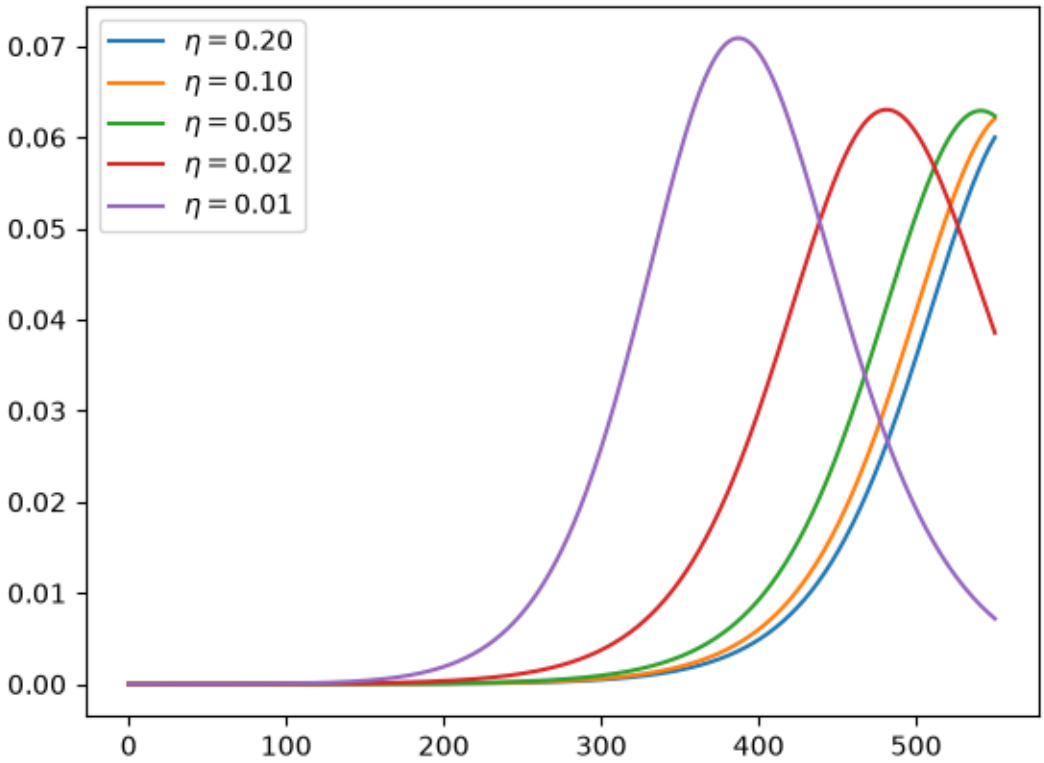


Let's calculate the time path of infected people:

```
i_paths, c_paths = [], []

for η in η_vals:
    R0 = lambda t: R0_mitigating(t, η=η)
    i_path, c_path = solve_path(R0, t_vec)
    i_paths.append(i_path)
    c_paths.append(c_path)
```

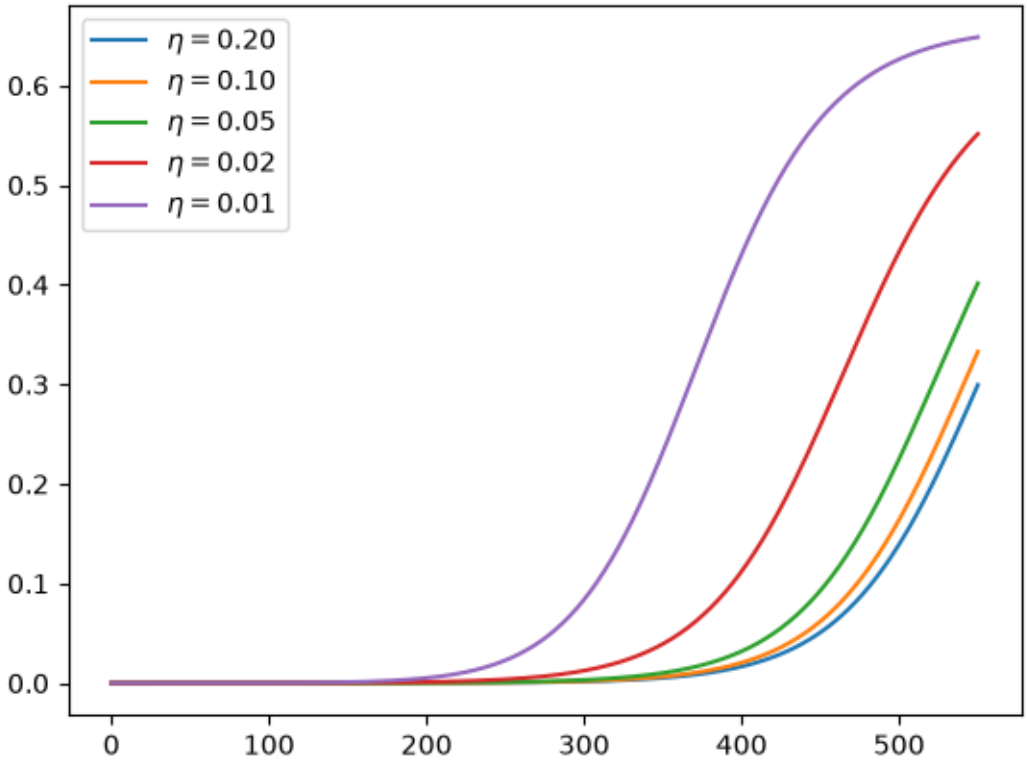
These are current cases under the different scenarios:

```
plot_paths(i_paths, labels)
```



Here are cumulative cases, as a fraction of population:

```
plot_paths(c_paths, labels)
```



1.5 Ending Lockdown

The following replicates additional results by Andrew Atkeson on the timing of lifting lockdown.

Consider these two mitigation scenarios:

1. $R_t = 0.5$ for 30 days and then $R_t = 2$ for the remaining 17 months. This corresponds to lifting lockdown in 30 days.
2. $R_t = 0.5$ for 120 days and then $R_t = 2$ for the remaining 14 months. This corresponds to lifting lockdown in 4 months.

The parameters considered here start the model with 25,000 active infections and 75,000 agents already exposed to the virus and thus soon to be contagious.

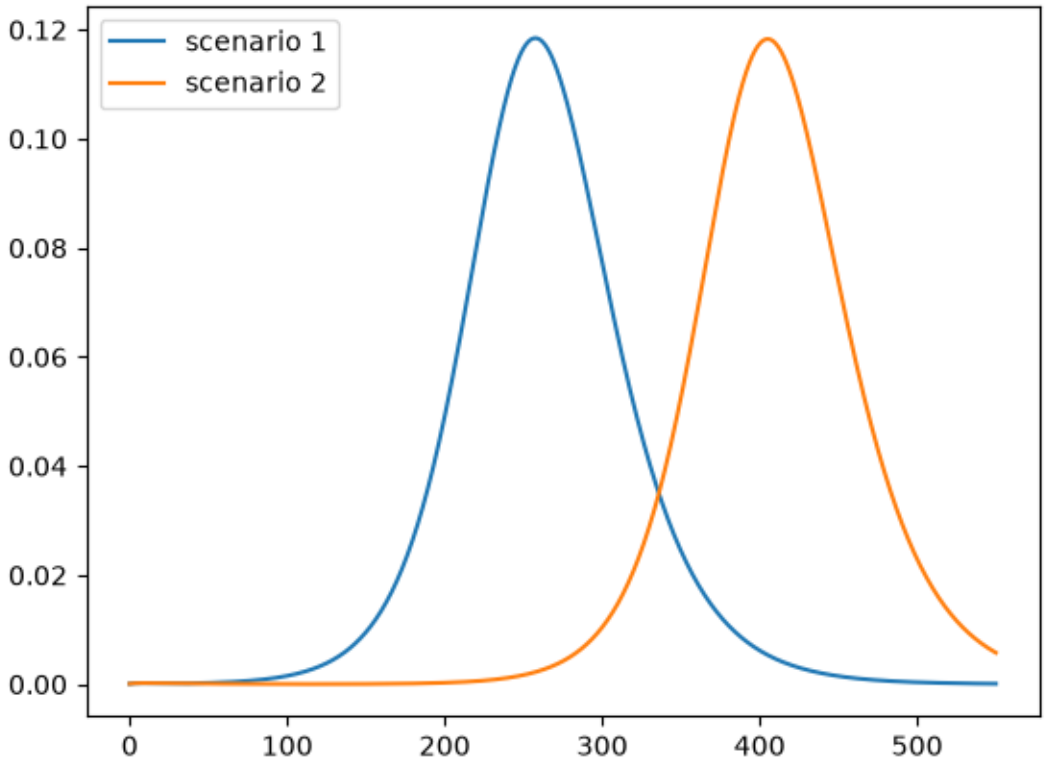
```
# initial conditions
i_0 = 25_000 / pop_size
e_0 = 75_000 / pop_size
s_0 = 1 - i_0 - e_0
x_0 = s_0, e_0, i_0
```

Let's calculate the paths:

```
R0_paths = (lambda t: 0.5 if t < 30 else 2,
            lambda t: 0.5 if t < 120 else 2)

labels = [f'scenario {i}' for i in (1, 2)]

i_paths, c_paths = [], []

for R0 in R0_paths:
    i_path, c_path = solve_path(R0, t_vec, x_init=x_0)
    i_paths.append(i_path)
    c_paths.append(c_path)
```

Here is the number of active infections:

```
plot_paths(i_paths, labels)
```



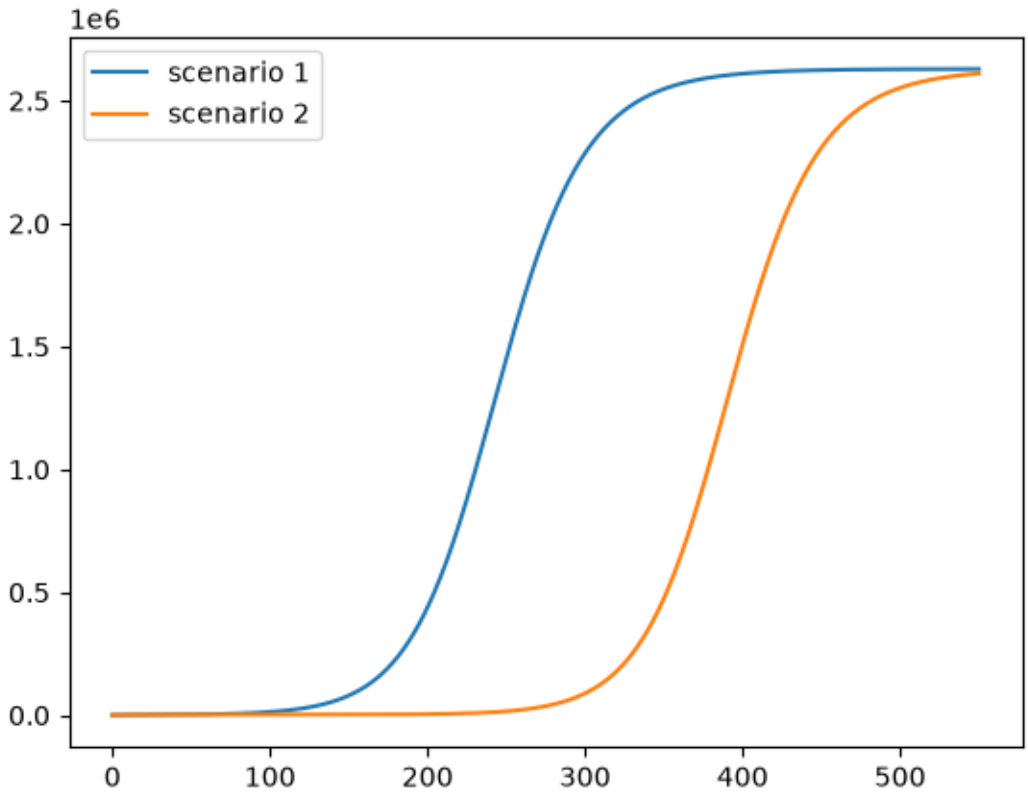
What kind of mortality can we expect under these scenarios?

Suppose that 1% of cases result in death

```
ν = 0.01
```

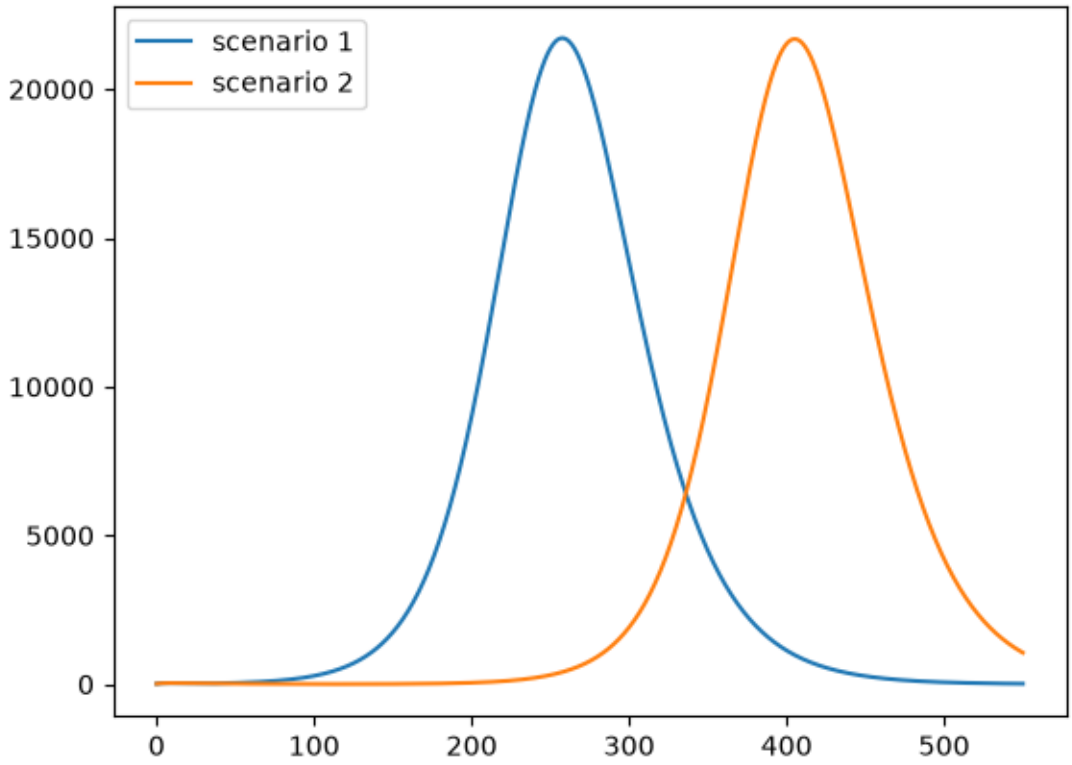
This is the cumulative number of deaths:

```
paths = [path * ν * pop_size for path in c_paths]
plot_paths(paths, labels)
```



This is the daily death rate:

```
paths = [path * ν * γ * pop_size for path in i_paths]
plot_paths(paths, labels)
```



Pushing the peak of curve further into the future may reduce cumulative deaths if a vaccine is found.

CHAPTER

TWO

LINEAR ALGEBRA

Contents

- *Linear Algebra*
  - *Overview*
  - *Vectors*
  - *Matrices*
  - *Solving Systems of Equations*
  - *Eigenvalues and Eigenvectors*
  - *Further Topics*
  - *Exercises*

2.1 Overview

Linear algebra is one of the most useful branches of applied mathematics for economists to invest in.

For example, many applied problems in economics and finance require the solution of a linear system of equations, such as

$$\begin{aligned} y_1 &= ax_1 + bx_2 \\ y_2 &= cx_1 + dx_2 \end{aligned}$$

or, more generally,

$$\begin{aligned} y_1 &= a_{11}x_1 + a_{12}x_2 + \cdots + a_{1k}x_k \\ &\vdots \\ y_n &= a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nk}x_k \end{aligned} \tag{2.1}$$

The objective here is to solve for the "unknowns" $x_1, \ldots, x_k$ given $a_{11}, \ldots, a_{nk}$ and $y_1, \ldots, y_n$.

When considering such problems, it is essential that we first consider at least some of the following questions

- Does a solution actually exist?
- Are there in fact many solutions, and if so how should we interpret them?
- If no solution exists, is there a best "approximate" solution?

- If a solution exists, how should we compute it?

These are the kinds of topics addressed by linear algebra.

In this lecture we will cover the basics of linear and matrix algebra, treating both theory and computation.

We admit some overlap with this lecture, where operations on NumPy arrays were first explained.

Note that this lecture is more theoretical than most, and contains background material that will be used in applications as we go along.

Let's start with some imports:

```
import matplotlib.pyplot as plt
import numpy as np
from matplotlib import cm
from mpl_toolkits.mplot3d import Axes3D
from scipy.linalg import inv, solve, det, eig
```

2.2 Vectors

A **vector** of length n is just a sequence (or array, or tuple) of n numbers, which we write as $x = (x_1, \ldots, x_n)$ or $x = [x_1, \ldots, x_n]$.

We will write these sequences either horizontally or vertically as we please.

(Later, when we wish to perform certain matrix operations, it will become necessary to distinguish between the two)

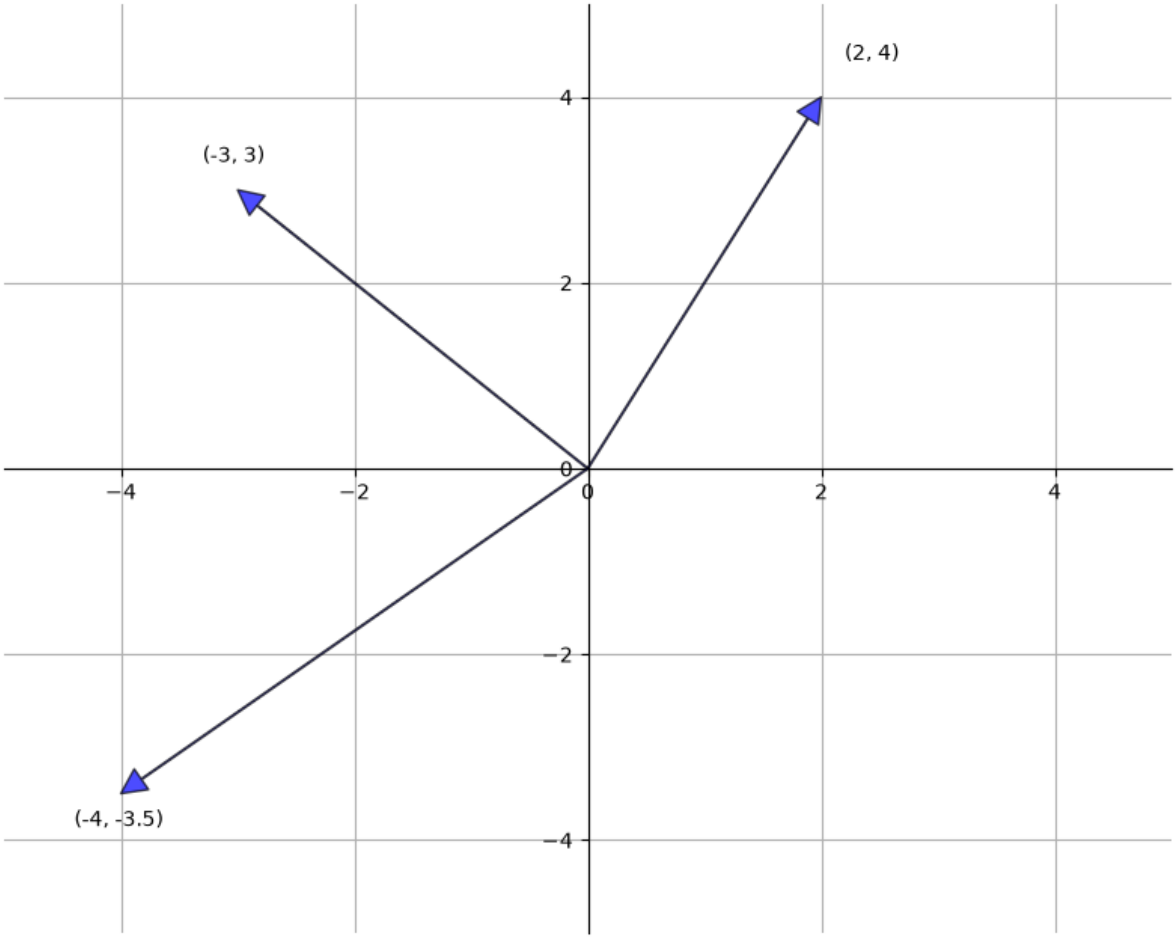
The set of all n-vectors is denoted by $\mathbb{R}^n$.

For example, $\mathbb{R}^2$ is the plane, and a vector in $\mathbb{R}^2$ is just a point in the plane.

Traditionally, vectors are represented visually as arrows from the origin to the point.

The following figure represents three vectors in this manner

```
fig, ax = plt.subplots(figsize=(10, 8))
# Set the axes through the origin
for spine in ['left', 'bottom']:
    ax.spines[spine].set_position('zero')
for spine in ['right', 'top']:
    ax.spines[spine].set_color('none')

ax.set(xlim=(-5, 5), ylim=(-5, 5))
ax.grid()
vecs = ((2, 4), (-3, 3), (-4, -3.5))
for v in vecs:
    ax.annotate('', xy=v, xytext=(0, 0),
                arrowprops=dict(facecolor='blue',
                shrink=0,
                alpha=0.7,
                width=0.5))
    ax.text(1.1 * v[0], 1.1 * v[1], str(v))
plt.show()
```



2.2.1 Vector Operations

The two most common operators for vectors are addition and scalar multiplication, which we now describe.

As a matter of definition, when we add two vectors, we add them element-by-element

$$x + y = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} := \begin{bmatrix} x_1 + y_1 \\ x_2 + y_2 \\ \vdots \\ x_n + y_n \end{bmatrix}$$

Scalar multiplication is an operation that takes a number γ and a vector x and produces

$$\gamma x := \begin{bmatrix} \gamma x_1 \\ \gamma x_2 \\ \vdots \\ \gamma x_n \end{bmatrix}$$

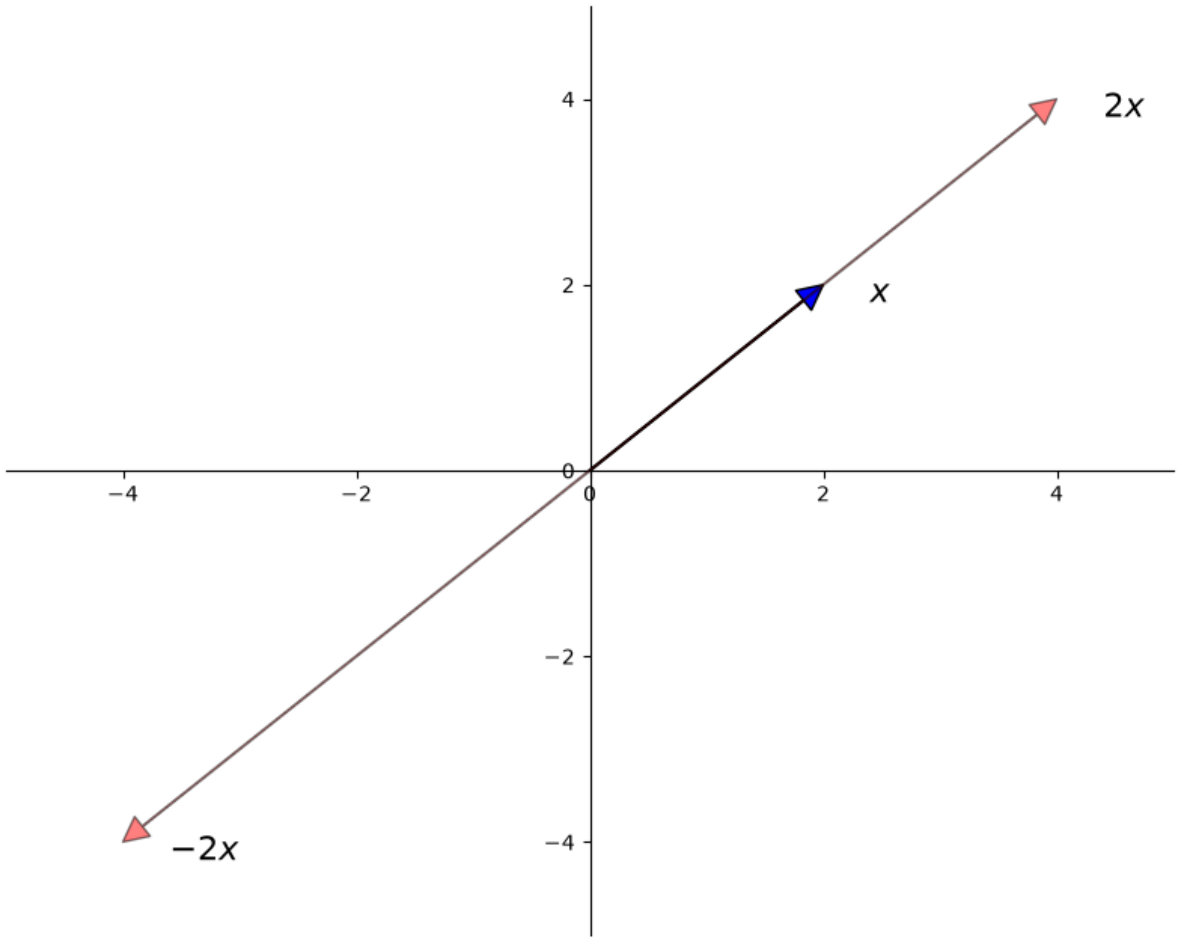
Scalar multiplication is illustrated in the next figure

```
fig, ax = plt.subplots(figsize=(10, 8))
# Set the axes through the origin
for spine in ['left', 'bottom']:
```

(continues on next page)

(continued from previous page)

```
    ax.spines[spine].set_position('zero')
for spine in ['right', 'top']:
    ax.spines[spine].set_color('none')

ax.set(xlim=(-5, 5), ylim=(-5, 5))
x = (2, 2)
ax.annotate('', xy=x, xytext=(0, 0),
            arrowprops=dict(facecolor='blue',
            shrink=0,
            alpha=1,
            width=0.5))
ax.text(x[0] + 0.4, x[1] - 0.2, '$x$', fontsize='16')


scalars = (-2, 2)
x = np.array(x)

for s in scalars:
    v = s * x
    ax.annotate('', xy=v, xytext=(0, 0),
                arrowprops=dict(facecolor='red',
                shrink=0,
                alpha=0.5,
                width=0.5))
    ax.text(v[0] + 0.4, v[1] - 0.2, f'${s} x$', fontsize='16')
plt.show()
```



In Python, a vector can be represented as a list or tuple, such as `x = (2, 4, 6)`, but is more commonly represented as a NumPy array.

One advantage of NumPy arrays is that scalar multiplication and addition have very natural syntax

```
x = np.ones(3)             # Vector of three ones
y = np.array((2, 4, 6))    # Converts tuple (2, 4, 6) into array
x + y
```

```
array([3., 5., 7.])
```

```
4 * x
```

```
array([4., 4., 4.])
```

2.2.2 Inner Product and Norm

The **inner product** of vectors $x, y \in \mathbb{R}^n$ is defined as

$$x'y := \sum_{i=1}^{n} x_i y_i$$

Two vectors are called **orthogonal** if their inner product is zero.

The **norm** of a vector x represents its "length" (i.e., its distance from the zero vector) and is defined as

$$\|x\| := \sqrt{x'x} := \left(\sum_{i=1}^{n} x_i^2 \right)^{1/2}$$

The expression $\|x - y\|$ is thought of as the distance between x and y.

Continuing on from the previous example, the inner product and norm can be computed as follows

```
np.sum(x * y)           # Inner product of x and y, method 1
```

```
np.float64(12.0)
```

```
x @ y                   # Inner product of x and y, method 2 (preferred)
```

```
np.float64(12.0)
```

The `@` operator is preferred because it uses optimized BLAS libraries that implement fused multiply-add operations, providing better performance and numerical accuracy compared to the separate multiply and sum operations.

```
np.sqrt(np.sum(x**2))   # Norm of x, take one
```

```
np.float64(1.7320508075688772)
```

```
np.sqrt(x @ x)          # Norm of x, take two (preferred)
```

```
np.float64(1.7320508075688772)
```

```
np.linalg.norm(x)       # Norm of x, take three
```

```
np.float64(1.7320508075688772)
```

2.2.3 Span

Given a set of vectors $A := \{a_1, \ldots, a_k\}$ in $\mathbb{R}^n$, it's natural to think about the new vectors we can create by performing linear operations.

New vectors created in this manner are called **linear combinations** of A.

In particular, $y \in \mathbb{R}^n$ is a linear combination of $A := \{a_1, \ldots, a_k\}$ if

$$y = \beta_1 a_1 + \cdots + \beta_k a_k \text{ for some scalars } \beta_1, \ldots, \beta_k$$

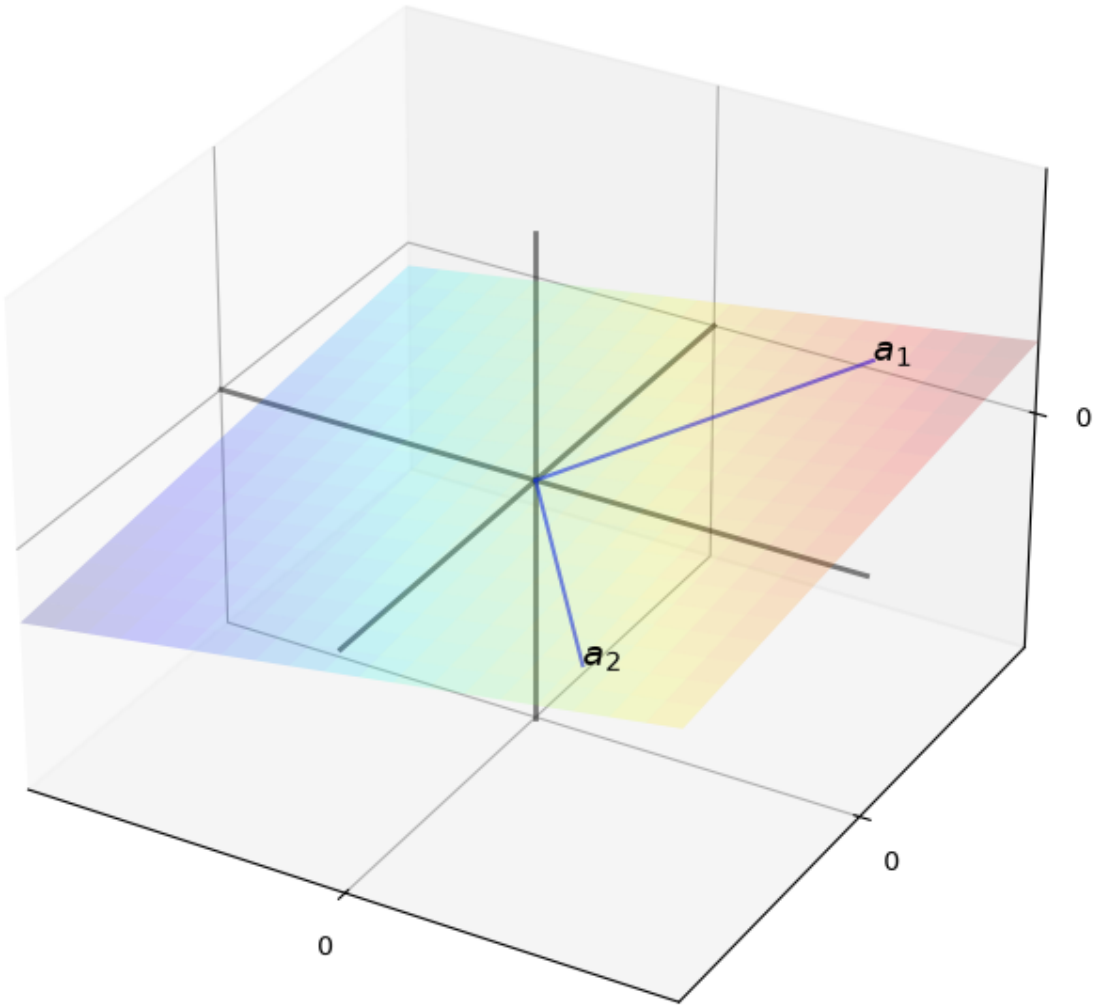
In this context, the values $\beta_1, \ldots, \beta_k$ are called the **coefficients** of the linear combination.

The set of linear combinations of A is called the **span** of A.

The next figure shows the span of $A = \{a_1, a_2\}$ in $\mathbb{R}^3$.

The span is a two-dimensional plane passing through these two points and the origin.

```
ax = plt.figure(figsize=(10, 8)).add_subplot(projection='3d')

x_min, x_max = -5, 5
y_min, y_max = -5, 5

α, β = 0.2, 0.1

ax.set(xlim=(x_min, x_max), ylim=(x_min, x_max), zlim=(x_min, x_max),
       xticks=(0,), yticks=(0,), zticks=(0,))

gs = 3
z = np.linspace(x_min, x_max, gs)
x = np.zeros(gs)
y = np.zeros(gs)
ax.plot(x, y, z, 'k-', lw=2, alpha=0.5)
ax.plot(z, x, y, 'k-', lw=2, alpha=0.5)
ax.plot(y, z, x, 'k-', lw=2, alpha=0.5)


# Fixed linear function, to generate a plane
def f(x, y):
    return α * x + β * y

# Vector locations, by coordinate
x_coords = np.array((3, 3))
y_coords = np.array((4, -4))
z = f(x_coords, y_coords)
for i in (0, 1):
    ax.text(x_coords[i], y_coords[i], z[i], f'$a_{i+1}$', fontsize=14)

# Lines to vectors
for i in (0, 1):
    x = (0, x_coords[i])
    y = (0, y_coords[i])
    z = (0, f(x_coords[i], y_coords[i]))
    ax.plot(x, y, z, 'b-', lw=1.5, alpha=0.6)


# Draw the plane
grid_size = 20
xr2 = np.linspace(x_min, x_max, grid_size)
yr2 = np.linspace(y_min, y_max, grid_size)
x2, y2 = np.meshgrid(xr2, yr2)
z2 = f(x2, y2)
ax.plot_surface(x2, y2, z2, rstride=1, cstride=1, cmap=cm.jet,
                linewidth=0, antialiased=True, alpha=0.2)
plt.show()
```



Examples

If A contains only one vector $a_1 \in \mathbb{R}^2$, then its span is just the scalar multiples of a_1, which is the unique line passing through both a_1 and the origin.

If $A = \{e_1, e_2, e_3\}$ consists of the **canonical basis vectors** of $\mathbb{R}^3$, that is

$$e_1 := \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \quad e_2 := \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \quad e_3 := \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

then the span of A is all of $\mathbb{R}^3$, because, for any $x = (x_1, x_2, x_3) \in \mathbb{R}^3$, we can write

$$x = x_1 e_1 + x_2 e_2 + x_3 e_3$$

Now consider $A_0 = \{e_1, e_2, e_1 + e_2\}$.

If $y = (y_1, y_2, y_3)$ is any linear combination of these vectors, then $y_3 = 0$ (check it).

Hence A_0 fails to span all of $\mathbb{R}^3$.

2.2.4 Linear Independence

As we'll see, it's often desirable to find families of vectors with relatively large span, so that many vectors can be described by linear operators on a few vectors.

The condition we need for a set of vectors to have a large span is what's called linear independence.

In particular, a collection of vectors $A := \{a_1, \ldots, a_k\}$ in $\mathbb{R}^n$ is said to be

- **linearly dependent** if some strict subset of A has the same span as A.
- **linearly independent** if it is not linearly dependent.

Put differently, a set of vectors is linearly independent if no vector is redundant to the span and linearly dependent otherwise.

To illustrate the idea, recall *the figure* that showed the span of vectors $\{a_1, a_2\}$ in $\mathbb{R}^3$ as a plane through the origin.

If we take a third vector a_3 and form the set $\{a_1, a_2, a_3\}$, this set will be

- linearly dependent if a_3 lies in the plane
- linearly independent otherwise

As another illustration of the concept, since $\mathbb{R}^n$ can be spanned by n vectors (see the discussion of canonical basis vectors above), any collection of $m > n$ vectors in $\mathbb{R}^n$ must be linearly dependent.

The following statements are equivalent to linear independence of $A := \{a_1, \ldots, a_k\} \subset \mathbb{R}^n$

1. No vector in A can be formed as a linear combination of the other elements.
2. If $\beta_1 a_1 + \cdots \beta_k a_k = 0$ for scalars $\beta_1, \ldots, \beta_k$, then $\beta_1 = \cdots = \beta_k = 0$.

(The zero in the first expression is the origin of $\mathbb{R}^n$)

2.2.5 Unique Representations

Another nice thing about sets of linearly independent vectors is that each element in the span has a unique representation as a linear combination of these vectors.

In other words, if $A := \{a_1, \ldots, a_k\} \subset \mathbb{R}^n$ is linearly independent and

$$y = \beta_1 a_1 + \cdots \beta_k a_k$$

then no other coefficient sequence $\gamma_1, \ldots, \gamma_k$ will produce the same vector y.

Indeed, if we also have $y = \gamma_1 a_1 + \cdots \gamma_k a_k$, then

$$(\beta_1 - \gamma_1) a_1 + \cdots + (\beta_k - \gamma_k) a_k = 0$$

Linear independence now implies $\gamma_i = \beta_i$ for all i.

2.3 Matrices

Matrices are a neat way of organizing data for use in linear operations.

An $n \times k$ matrix is a rectangular array A of numbers with n rows and k columns:

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1k} \\ a_{21} & a_{22} & \cdots & a_{2k} \\ \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nk} \end{bmatrix}$$

Often, the numbers in the matrix represent coefficients in a system of linear equations, as discussed at the start of this lecture.

For obvious reasons, the matrix A is also called a vector if either $n = 1$ or $k = 1$.

In the former case, A is called a **row vector**, while in the latter it is called a **column vector**.

If $n = k$, then A is called **square**.

The matrix formed by replacing a_{ij} by a_{ji} for every i and j is called the **transpose** of A and denoted A' or $A^\top$.

If $A = A'$, then A is called **symmetric**.

For a square matrix A, the i elements of the form a_{ii} for $i = 1, \ldots, n$ are called the **principal diagonal**.

A is called **diagonal** if the only nonzero entries are on the principal diagonal.

If, in addition to being diagonal, each element along the principal diagonal is equal to 1, then A is called the **identity matrix** and denoted by I.

2.3.1 Matrix Operations

Just as was the case for vectors, a number of algebraic operations are defined for matrices.

Scalar multiplication and addition are immediate generalizations of the vector case:

$$\gamma A = \gamma \begin{bmatrix} a_{11} & \cdots & a_{1k} \\ \vdots & \vdots & \vdots \\ a_{n1} & \cdots & a_{nk} \end{bmatrix} := \begin{bmatrix} \gamma a_{11} & \cdots & \gamma a_{1k} \\ \vdots & \vdots & \vdots \\ \gamma a_{n1} & \cdots & \gamma a_{nk} \end{bmatrix}$$

and

$$A + B = \begin{bmatrix} a_{11} & \cdots & a_{1k} \\ \vdots & \vdots & \vdots \\ a_{n1} & \cdots & a_{nk} \end{bmatrix} + \begin{bmatrix} b_{11} & \cdots & b_{1k} \\ \vdots & \vdots & \vdots \\ b_{n1} & \cdots & b_{nk} \end{bmatrix} := \begin{bmatrix} a_{11} + b_{11} & \cdots & a_{1k} + b_{1k} \\ \vdots & \vdots & \vdots \\ a_{n1} + b_{n1} & \cdots & a_{nk} + b_{nk} \end{bmatrix}$$

In the latter case, the matrices must have the same shape in order for the definition to make sense.

We also have a convention for *multiplying* two matrices.

The rule for matrix multiplication generalizes the idea of inner products discussed above and is designed to make multiplication play well with basic linear operations.

If A and B are two matrices, then their product AB is formed by taking as its i, j-th element the inner product of the i-th row of A and the j-th column of B.

There are many tutorials to help you visualize this operation, such as this one, or the discussion on the Wikipedia page.

If A is $n \times k$ and B is $j \times m$, then to multiply A and B we require $k = j$, and the resulting matrix AB is $n \times m$.

As perhaps the most important special case, consider multiplying $n \times k$ matrix A and $k \times 1$ column vector x.

According to the preceding rule, this gives us an $n \times 1$ column vector

$$Ax = \begin{bmatrix} a_{11} & \cdots & a_{1k} \\ \vdots & \vdots & \vdots \\ a_{n1} & \cdots & a_{nk} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_k \end{bmatrix} := \begin{bmatrix} a_{11}x_1 + \cdots + a_{1k}x_k \\ \vdots \\ a_{n1}x_1 + \cdots + a_{nk}x_k \end{bmatrix} \tag{2.2}$$

Note

AB and BA are not generally the same thing.

Another important special case is the identity matrix.

You should check that if A is $n \times k$ and I is the $k \times k$ identity matrix, then $AI = A$.

If I is the $n \times n$ identity matrix, then $IA = A$.

2.3.2 Matrices in NumPy

NumPy arrays are also used as matrices, and have fast, efficient functions and methods for all the standard matrix operations[1].

You can create them manually from tuples of tuples (or lists of lists) as follows

```
A = ((1, 2),
     (3, 4))

type(A)
```

```
tuple
```

```
A = np.array(A)

type(A)
```

```
numpy.ndarray
```

```
A.shape
```

```
(2, 2)
```

The `shape` attribute is a tuple giving the number of rows and columns — see here for more discussion.

To get the transpose of `A`, use `A.transpose()` or, more simply, `A.T`.

There are many convenient functions for creating common matrices (matrices of zeros, ones, etc.) — see here.

Since operations are performed elementwise by default, scalar multiplication and addition have very natural syntax

```
A = np.identity(3)
B = np.ones((3, 3))
2 * A
```

[1] Although there is a specialized matrix data type defined in NumPy, it's more standard to work with ordinary NumPy arrays. See this discussion.

```
array([[2., 0., 0.],
       [0., 2., 0.],
       [0., 0., 2.]])
```

```
A + B
```

```
array([[2., 1., 1.],
       [1., 2., 1.],
       [1., 1., 2.]])
```

To multiply matrices we use the `@` symbol.

In particular, `A @ B` is matrix multiplication, whereas `A * B` is element-by-element multiplication.

See here for more discussion.

2.3.3 Matrices as Maps

Each $n \times k$ matrix A can be identified with a function $f(x) = Ax$ that maps $x \in \mathbb{R}^k$ into $y = Ax \in \mathbb{R}^n$.

These kinds of functions have a special property: they are **linear**.

A function $f \colon \mathbb{R}^k \to \mathbb{R}^n$ is called **linear** if, for all $x, y \in \mathbb{R}^k$ and all scalars α, β, we have

$$f(\alpha x + \beta y) = \alpha f(x) + \beta f(y)$$

You can check that this holds for the function $f(x) = Ax + b$ when b is the zero vector and fails when b is nonzero.

In fact, it's known that f is linear if and *only if* there exists a matrix A such that $f(x) = Ax$ for all x.

2.4 Solving Systems of Equations

Recall again the system of equations (2.1).

If we compare (2.1) and (2.2), we see that (2.1) can now be written more conveniently as

$$y = Ax \tag{2.3}$$

The problem we face is to determine a vector $x \in \mathbb{R}^k$ that solves (2.3), taking y and A as given.

This is a special case of a more general problem: Find an x such that $y = f(x)$.

Given an arbitrary function f and a y, is there always an x such that $y = f(x)$?

If so, is it always unique?

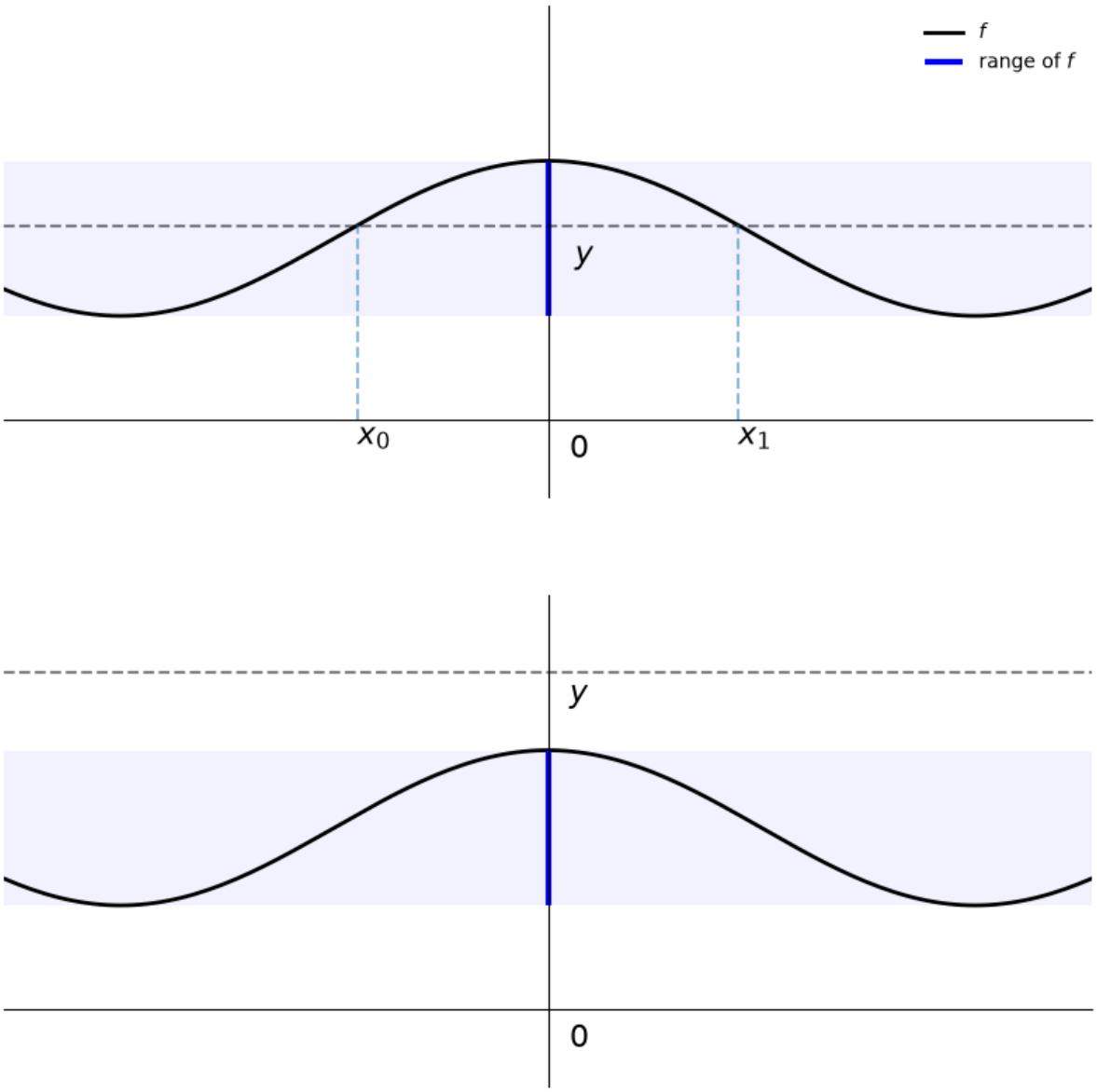
The answer to both these questions is negative, as the next figure shows

```
def f(x):
    return 0.6 * np.cos(4 * x) + 1.4


xmin, xmax = -1, 1
x = np.linspace(xmin, xmax, 160)
y = f(x)
ya, yb = np.min(y), np.max(y)
```

(continues on next page)

(continued from previous page)

```
fig, axes = plt.subplots(2, 1, figsize=(10, 10))

for ax in axes:
    # Set the axes through the origin
    for spine in ['left', 'bottom']:
        ax.spines[spine].set_position('zero')
    for spine in ['right', 'top']:
        ax.spines[spine].set_color('none')

    ax.set(ylim=(-0.6, 3.2), xlim=(xmin, xmax),
           yticks=(), xticks=())

    ax.plot(x, y, 'k-', lw=2, label='$f$')
    ax.fill_between(x, ya, yb, facecolor='blue', alpha=0.05)
    ax.vlines([0], ya, yb, lw=3, color='blue', label='range of $f$')
    ax.text(0.04, -0.3, '$0$', fontsize=16)

ax = axes[0]

ax.legend(loc='upper right', frameon=False)
ybar = 1.5
ax.plot(x, x * 0 + ybar, 'k--', alpha=0.5)
ax.text(0.05, 0.8 * ybar, '$y$', fontsize=16)
for i, z in enumerate((-0.35, 0.35)):
    ax.vlines(z, 0, f(z), linestyle='--', alpha=0.5)
    ax.text(z, -0.2, f'$x_{i}$', fontsize=16)

ax = axes[1]

ybar = 2.6
ax.plot(x, x * 0 + ybar, 'k--', alpha=0.5)
ax.text(0.04, 0.91 * ybar, '$y$', fontsize=16)

plt.show()
```



In the first plot, there are multiple solutions, as the function is not one-to-one, while in the second there are no solutions, since y lies outside the range of f.

Can we impose conditions on A in (2.3) that rule out these problems?

In this context, the most important thing to recognize about the expression Ax is that it corresponds to a linear combination of the columns of A.

In particular, if $a_1, \ldots, a_k$ are the columns of A, then

$$Ax = x_1 a_1 + \cdots + x_k a_k$$

Hence the range of $f(x) = Ax$ is exactly the span of the columns of A.

We want the range to be large so that it contains arbitrary y.

As you might recall, the condition that we want for the span to be large is *linear independence*.

A happy fact is that linear independence of the columns of A also gives us uniqueness.

Indeed, it follows from our *earlier discussion* that if $\{a_1, \ldots, a_k\}$ are linearly independent and $y = Ax = x_1 a_1 + \cdots + x_k a_k$, then no $z \neq x$ satisfies $y = Az$.

2.4.1 The Square Matrix Case

Let's discuss some more details, starting with the case where A is $n \times n$.

This is the familiar case where the number of unknowns equals the number of equations.

For arbitrary $y \in \mathbb{R}^n$, we hope to find a unique $x \in \mathbb{R}^n$ such that $y = Ax$.

In view of the observations immediately above, if the columns of A are linearly independent, then their span, and hence the range of $f(x) = Ax$, is all of $\mathbb{R}^n$.

Hence there always exists an x such that $y = Ax$.

Moreover, the solution is unique.

In particular, the following are equivalent

1. The columns of A are linearly independent.
2. For any $y \in \mathbb{R}^n$, the equation $y = Ax$ has a unique solution.

The property of having linearly independent columns is sometimes expressed as having **full column rank**.

Inverse Matrices

Can we give some sort of expression for the solution?

If y and A are scalar with $A \neq 0$, then the solution is $x = A^{-1} y$.

A similar expression is available in the matrix case.

In particular, if square matrix A has full column rank, then it possesses a multiplicative **inverse matrix** A^{-1}, with the property that $AA^{-1} = A^{-1}A = I$.

As a consequence, if we pre-multiply both sides of $y = Ax$ by A^{-1}, we get $x = A^{-1} y$.

This is the solution that we're looking for.

Determinants

Another quick comment about square matrices is that to every such matrix we assign a unique number called the **determinant** of the matrix — you can find the expression for it here.

If the determinant of A is not zero, then we say that A is **nonsingular**.

Perhaps the most important fact about determinants is that A is nonsingular if and only if A is of full column rank.

This gives us a useful one-number summary of whether or not a square matrix can be inverted.

2.4.2 More Rows than Columns

This is the $n \times k$ case with $n > k$.

This case is very important in many settings, not least in the setting of linear regression (where n is the number of observations, and k is the number of explanatory variables).

Given arbitrary $y \in \mathbb{R}^n$, we seek an $x \in \mathbb{R}^k$ such that $y = Ax$.

In this setting, the existence of a solution is highly unlikely.

Without much loss of generality, let's go over the intuition focusing on the case where the columns of A are linearly independent.

It follows that the span of the columns of A is a k-dimensional subspace of $\mathbb{R}^n$.

This span is very "unlikely" to contain arbitrary $y \in \mathbb{R}^n$.

To see why, recall the *figure above*, where $k = 2$ and $n = 3$.

Imagine an arbitrarily chosen $y \in \mathbb{R}^3$, located somewhere in that three-dimensional space.

What's the likelihood that y lies in the span of $\{a_1, a_2\}$ (i.e., the two dimensional plane through these points)?

In a sense, it must be very small, since this plane has zero "thickness".

As a result, in the $n > k$ case we usually give up on existence.

However, we can still seek the best approximation, for example, an x that makes the distance $\|y - Ax\|$ as small as possible.

To solve this problem, one can use either calculus or the theory of orthogonal projections.

The solution is known to be $\hat{x} = (A'A)^{-1}A'y$ — see for example chapter 3 of these notes.

2.4.3 More Columns than Rows

This is the $n \times k$ case with $n < k$, so there are fewer equations than unknowns.

In this case there are either no solutions or infinitely many — in other words, uniqueness never holds.

For example, consider the case where $k = 3$ and $n = 2$.

Thus, the columns of A consists of 3 vectors in $\mathbb{R}^2$.

This set can never be linearly independent, since it is possible to find two vectors that span $\mathbb{R}^2$.

(For example, use the canonical basis vectors)

It follows that one column is a linear combination of the other two.

For example, let's say that $a_1 = \alpha a_2 + \beta a_3$.

Then if $y = Ax = x_1 a_1 + x_2 a_2 + x_3 a_3$, we can also write

$$y = x_1(\alpha a_2 + \beta a_3) + x_2 a_2 + x_3 a_3 = (x_1\alpha + x_2)a_2 + (x_1\beta + x_3)a_3$$

In other words, uniqueness fails.

2.4.4 Linear Equations with SciPy

Here's an illustration of how to solve linear equations with SciPy's `linalg` submodule.

All of these routines are Python front ends to time-tested and highly optimized FORTRAN code

```
A = ((1, 2), (3, 4))
A = np.array(A)
y = np.ones((2, 1))  # Column vector
det(A)  # Check that A is nonsingular, and hence invertible
```

```
np.float64(-2.0)
```

```
A_inv = inv(A)  # Compute the inverse
A_inv
```

```
array([[-2. ,  1. ],
       [ 1.5, -0.5]])
```

```
x = A_inv @ y  # Solution
A @ x          # Should equal y
```

```
array([[1.],
       [1.]])
```

```
solve(A, y)  # Produces the same solution
```

```
array([[-1.],
       [ 1.]])
```

Observe how we can solve for $x = A^{-1}y$ by either via `inv(A) @ y`, or using `solve(A, y)`.

The latter method uses a different algorithm (LU decomposition) that is numerically more stable, and hence should almost always be preferred.

To obtain the least-squares solution $\hat{x} = (A'A)^{-1}A'y$, use `scipy.linalg.lstsq(A, y)`.

2.5 Eigenvalues and Eigenvectors

Let A be an $n \times n$ square matrix.

If λ is scalar and v is a non-zero vector in $\mathbb{R}^n$ such that

$$Av = \lambda v$$

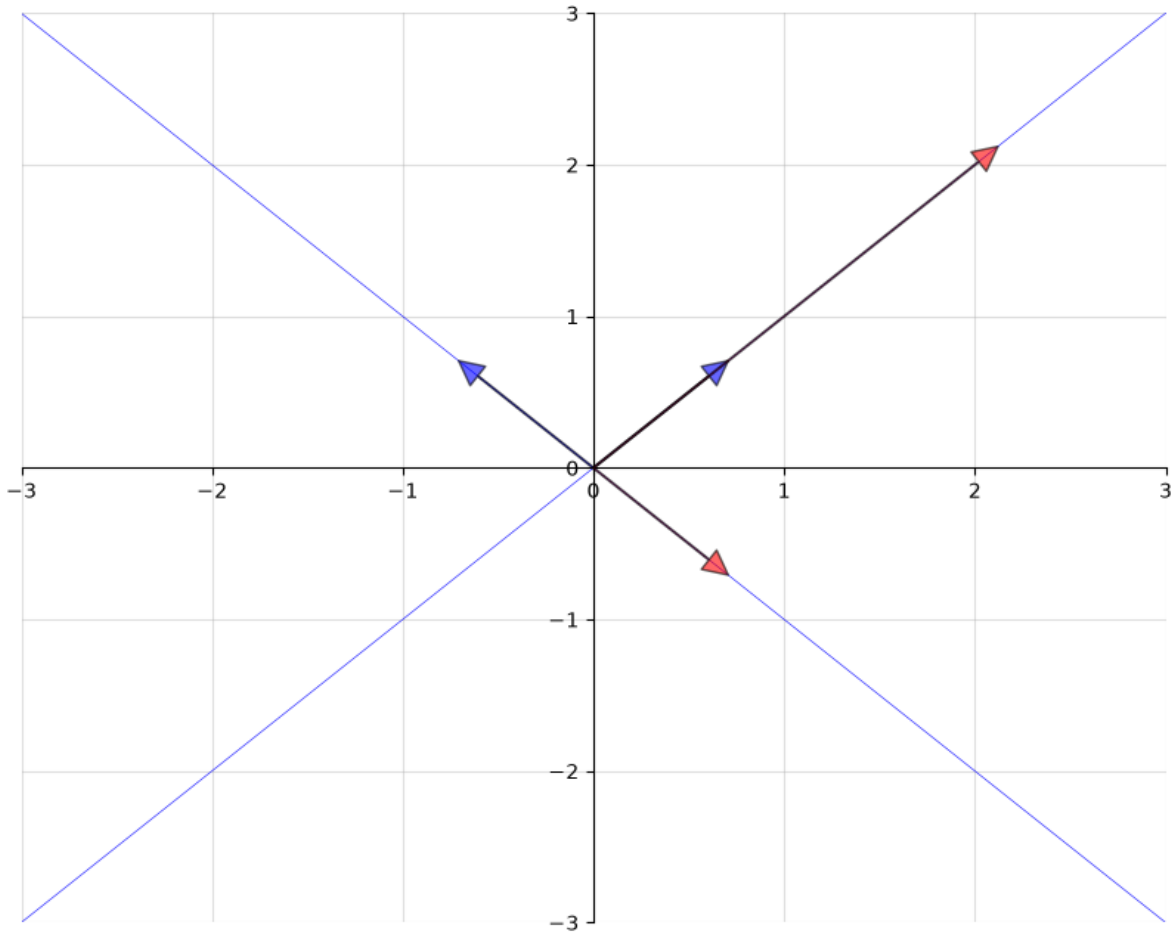
then we say that λ is an **eigenvalue** of A, and v is an **eigenvector**.

Thus, an eigenvector of A is a vector such that when the map $f(x) = Ax$ is applied, v is merely scaled.

The next figure shows two eigenvectors (blue arrows) and their images under A (red arrows).

As expected, the image Av of each v is just a scaled version of the original

```
A = ((1, 2),
     (2, 1))
A = np.array(A)
evals, evecs = eig(A)
evecs = evecs[:, 0], evecs[:, 1]

fig, ax = plt.subplots(figsize=(10, 8))
# Set the axes through the origin
for spine in ['left', 'bottom']:
    ax.spines[spine].set_position('zero')
for spine in ['right', 'top']:
    ax.spines[spine].set_color('none')
ax.grid(alpha=0.4)

xmin, xmax = -3, 3
ymin, ymax = -3, 3
ax.set(xlim=(xmin, xmax), ylim=(ymin, ymax))

# Plot each eigenvector
for v in evecs:
    ax.annotate('', xy=v, xytext=(0, 0),
                arrowprops=dict(facecolor='blue',
                shrink=0,
                alpha=0.6,
                width=0.5))

# Plot the image of each eigenvector
for v in evecs:
    v = A @ v
    ax.annotate('', xy=v, xytext=(0, 0),
                arrowprops=dict(facecolor='red',
                shrink=0,
                alpha=0.6,
                width=0.5))

# Plot the lines they run through
x = np.linspace(xmin, xmax, 3)
for v in evecs:
    a = v[1] / v[0]
    ax.plot(x, a * x, 'b-', lw=0.4)

plt.show()
```



The eigenvalue equation is equivalent to $(A - \lambda I)v = 0$, and this has a nonzero solution v only when the columns of $A - \lambda I$ are linearly dependent.

This in turn is equivalent to stating that the determinant is zero.

Hence to find all eigenvalues, we can look for λ such that the determinant of $A - \lambda I$ is zero.

This problem can be expressed as one of solving for the roots of a polynomial in λ of degree n.

This in turn implies the existence of n solutions in the complex plane, although some might be repeated.

Some nice facts about the eigenvalues of a square matrix A are as follows

1. The determinant of A equals the product of the eigenvalues.
2. The trace of A (the sum of the elements on the principal diagonal) equals the sum of the eigenvalues.
3. If A is symmetric, then all of its eigenvalues are real.
4. If A is invertible and $\lambda_1, \ldots, \lambda_n$ are its eigenvalues, then the eigenvalues of A^{-1} are $1/\lambda_1, \ldots, 1/\lambda_n$.

A corollary of the first statement is that a matrix is invertible if and only if all its eigenvalues are nonzero.

Using SciPy, we can solve for the eigenvalues and eigenvectors of a matrix as follows

```
A = ((1, 2),
     (2, 1))
```

(continues on next page)

(continued from previous page)

```
A = np.array(A)
evals, evecs = eig(A)
evals
```

```
array([ 3.+0.j, -1.+0.j])
```

```
evecs
```

```
array([[ 0.70710678, -0.70710678],
       [ 0.70710678,  0.70710678]])
```

Note that the *columns* of `evecs` are the eigenvectors.

Since any scalar multiple of an eigenvector is an eigenvector with the same eigenvalue (check it), the eig routine normalizes the length of each eigenvector to one.

2.5.1 Generalized Eigenvalues

It is sometimes useful to consider the **generalized eigenvalue problem**, which, for given matrices A and B, seeks generalized eigenvalues λ and eigenvectors v such that

$$Av = \lambda Bv$$

This can be solved in SciPy via `scipy.linalg.eig(A, B)`.

Of course, if B is square and invertible, then we can treat the generalized eigenvalue problem as an ordinary eigenvalue problem $B^{-1}Av = \lambda v$, but this is not always the case.

2.6 Further Topics

We round out our discussion by briefly mentioning several other important topics.

2.6.1 Series Expansions

Recall the usual summation formula for a geometric progression, which states that if $|a| < 1$, then $\sum_{k=0}^{\infty} a^k = (1-a)^{-1}$.

A generalization of this idea exists in the matrix setting.

Matrix Norms

Let A be a square matrix, and let

$$\|A\| := \max_{\|x\|=1} \|Ax\|$$

The norms on the right-hand side are ordinary vector norms, while the norm on the left-hand side is a **matrix norm** — in this case, the so-called **spectral norm**.

For example, for a square matrix S, the condition $\|S\| < 1$ means that S is **contractive**, in the sense that it pulls all vectors towards the origin[2].

[2] Suppose that $\|S\| < 1$. Take any nonzero vector x, and let $r := \|x\|$. We have $\|Sx\| = r\|S(x/r)\| \leq r\|S\| < r = \|x\|$. Hence every point is pulled towards the origin.

Neumann's Theorem

Let A be a square matrix and let $A^k := AA^{k-1}$ with $A^1 := A$.

In other words, A^k is the k-th power of A.

Neumann's theorem states the following: If $\|A^k\| < 1$ for some $k \in \mathbb{N}$, then $I - A$ is invertible, and

$$(I - A)^{-1} = \sum_{k=0}^{\infty} A^k \tag{2.4}$$

Spectral Radius

A result known as Gelfand's formula tells us that, for any square matrix A,

$$\rho(A) = \lim_{k \to \infty} \|A^k\|^{1/k}$$

Here $\rho(A)$ is the **spectral radius**, defined as $\max_i |\lambda_i|$, where $\{\lambda_i\}_i$ is the set of eigenvalues of A.

As a consequence of Gelfand's formula, if all eigenvalues are strictly less than one in modulus, there exists a k with $\|A^k\| < 1$.

In which case (2.4) is valid.

2.6.2 Positive Definite Matrices

Let A be a symmetric $n \times n$ matrix.

We say that A is

1. **positive definite** if $x'Ax > 0$ for every $x \in \mathbb{R}^n \setminus \{0\}$
2. **positive semi-definite** or **nonnegative definite** if $x'Ax \geq 0$ for every $x \in \mathbb{R}^n$

Analogous definitions exist for negative definite and negative semi-definite matrices.

It is notable that if A is positive definite, then all of its eigenvalues are strictly positive, and hence A is invertible (with positive definite inverse).

2.6.3 Differentiating Linear and Quadratic Forms

The following formulas are useful in many economic contexts. Let

- z, x and a all be $n \times 1$ vectors
- A be an $n \times n$ matrix
- B be an $m \times n$ matrix and y be an $m \times 1$ vector

Then

1. $\frac{\partial a'x}{\partial x} = a$
2. $\frac{\partial Ax}{\partial x} = A'$
3. $\frac{\partial x'Ax}{\partial x} = (A + A')x$
4. $\frac{\partial y'Bz}{\partial y} = Bz$

5. $\frac{\partial y' B z}{\partial B} = y z'$

Exercise 2.7.1 below asks you to apply these formulas.

2.6.4 Further Reading

The documentation of the `scipy.linalg` submodule can be found here.

Chapters 2 and 3 of the Econometric Theory contains a discussion of linear algebra along the same lines as above, with solved exercises.

If you don't mind a slightly abstract approach, a nice intermediate-level text on linear algebra is [Jänich, 1994].

2.7 Exercises

Exercise 2.7.1

Let x be a given $n \times 1$ vector and consider the problem

$$v(x) = \max_{y,u} \{-y'Py - u'Qu\}$$

subject to the linear constraint

$$y = Ax + Bu$$

Here

- P is an $n \times n$ matrix and Q is an $m \times m$ matrix
- A is an $n \times n$ matrix and B is an $n \times m$ matrix
- both P and Q are symmetric and positive semidefinite

(What must the dimensions of y and u be to make this a well-posed problem?)

One way to solve the problem is to form the Lagrangian

$$\mathcal{L} = -y'Py - u'Qu + \lambda' \left[Ax + Bu - y\right]$$

where λ is an $n \times 1$ vector of Lagrange multipliers.

Try applying the formulas given above for differentiating quadratic and linear forms to obtain the first-order conditions for maximizing $\mathcal{L}$ with respect to y, u and minimizing it with respect to λ.

Show that these conditions imply that

1. $\lambda = -2Py$.
2. The optimizing choice of u satisfies $u = -(Q + B'PB)^{-1}B'PAx$.
3. The function v satisfies $v(x) = -x'\tilde{P}x$ where $\tilde{P} = A'PA - A'PB(Q + B'PB)^{-1}B'PA$.

As we will see, in economic contexts Lagrange multipliers often are shadow prices.

Note

If we don't care about the Lagrange multipliers, we can substitute the constraint into the objective function, and then just maximize $-(Ax + Bu)'P(Ax + Bu) - u'Qu$ with respect to u. You can verify that this leads to the same maximizer.

Solution

We have an optimization problem:

$$v(x) = \max_{y,u} \{-y'Py - u'Qu\}$$

s.t.

$$y = Ax + Bu$$

with primitives

- P be a symmetric and positive semidefinite $n \times n$ matrix
- Q be a symmetric and positive semidefinite $m \times m$ matrix

- A an $n \times n$ matrix
- B an $n \times m$ matrix

The associated Lagrangian is:

$$L = -y'Py - u'Qu + \lambda'[Ax + Bu - y]$$

Step 1.

Differentiating Lagrangian equation w.r.t y and setting its derivative equal to zero yields

$$\frac{\partial L}{\partial y} = -(P + P')y - \lambda = -2Py - \lambda = 0 \,,$$

since P is symmetric.

Accordingly, the first-order condition for maximizing L w.r.t. y implies

$$\lambda = -2Py$$

Step 2.

Differentiating Lagrangian equation w.r.t. u and setting its derivative equal to zero yields

$$\frac{\partial L}{\partial u} = -(Q + Q')u - B'\lambda = -2Qu + B'\lambda = 0$$

Substituting $\lambda = -2Py$ gives

$$Qu + B'Py = 0$$

Substituting the linear constraint $y = Ax + Bu$ into above equation gives

$$Qu + B'P(Ax + Bu) = 0$$

$$(Q + B'PB)u + B'PAx = 0$$

which is the first-order condition for maximizing L w.r.t. u.

Thus, the optimal choice of u must satisfy

$$u = -(Q + B'PB)^{-1}B'PAx \,,$$

which follows from the definition of the first-order conditions for Lagrangian equation.

Step 3.

Rewriting our problem by substituting the constraint into the objective function, we get

$$v(x) = \max_u \{-(Ax + Bu)'P(Ax + Bu) - u'Qu\}$$

Since we know the optimal choice of u satisfies $u = -(Q + B'PB)^{-1}B'PAx$, then

$$v(x) = -(Ax + Bu)'P(Ax + Bu) - u'Qu \quad with \quad u = -(Q + B'PB)^{-1}B'PAx$$

To evaluate the function

$$\begin{aligned} v(x) &= -(Ax + Bu)'P(Ax + Bu) - u'Qu \\ &= -(x'A' + u'B')P(Ax + Bu) - u'Qu \\ &= -x'A'PAx - u'B'PAx - x'A'PBu - u'B'PBu - u'Qu \\ &= -x'A'PAx - 2u'B'PAx - u'(Q + B'PB)u \end{aligned}$$

For simplicity, denote by $S := (Q + B'PB)^{-1}B'PA$, then $u = -Sx$.

Regarding the second term $-2u'B'PAx$,

$$\begin{aligned} -2u'B'PAx &= -2x'S'B'PAx \\ &= 2x'A'PB(Q + B'PB)^{-1}B'PAx \end{aligned}$$

Notice that the term $(Q + B'PB)^{-1}$ is symmetric as both P and Q are symmetric.

Regarding the third term $-u'(Q + B'PB)u$,

$$\begin{aligned} -u'(Q + B'PB)u &= -x'S'(Q + B'PB)Sx \\ &= -x'A'PB(Q + B'PB)^{-1}B'PAx \end{aligned}$$

Hence, the summation of second and third terms is $x'A'PB(Q + B'PB)^{-1}B'PAx$.

This implies that

$$\begin{aligned} v(x) &= -x'A'PAx - 2u'B'PAx - u'(Q + B'PB)u \\ &= -x'A'PAx + x'A'PB(Q + B'PB)^{-1}B'PAx \\ &= -x'[A'PA - A'PB(Q + B'PB)^{-1}B'PA]x \end{aligned}$$

Therefore, the solution to the optimization problem $v(x) = -x'\tilde{P}x$ follows the above result by denoting $\tilde{P} := A'PA - A'PB(Q + B'PB)^{-1}B'PA$

CHAPTER

THREE

QR DECOMPOSITION

3.1 Overview

This lecture describes the QR decomposition and how it relates to

- Orthogonal projection and least squares
- A Gram-Schmidt process
- Eigenvalues and eigenvectors

We'll write some Python code to help consolidate our understandings.

3.2 Matrix Factorization

The QR decomposition (also called the QR factorization) of a matrix is a decomposition of a matrix into the product of an orthogonal matrix and a triangular matrix.

A QR decomposition of a real matrix A takes the form

$$A = QR$$

where

- Q is an orthogonal matrix (so that $Q^T Q = I$)
- R is an upper triangular matrix

We'll use a **Gram-Schmidt process** to compute a QR decomposition

Because doing so is so educational, we'll write our own Python code to do the job

3.3 Gram-Schmidt process

We'll start with a **square** matrix A.

If a square matrix A is nonsingular, then a QR factorization is unique.

We'll deal with a rectangular matrix A later.

Actually, our algorithm will work with a rectangular A that is not square.

3.3.1 Gram-Schmidt process for square A

Here we apply a Gram-Schmidt process to the **columns** of matrix A.

In particular, let

$$A = \begin{bmatrix} a_1 \mid a_2 \mid \cdots \mid a_n \end{bmatrix}$$

Let $||\cdot||$ denote the L2 norm.

The Gram-Schmidt algorithm repeatedly combines the following two steps in a particular order

- **normalize** a vector to have unit norm
- **orthogonalize** the next vector

To begin, we set $u_1 = a_1$ and then **normalize**:

$$u_1 = a_1, \quad e_1 = \frac{u_1}{||u_1||}$$

We **orthogonalize** first to compute u_2 and then **normalize** to create e_2:

$$u_2 = a_2 - (a_2 \cdot e_1)e_1, \quad e_2 = \frac{u_2}{||u_2||}$$

We invite the reader to verify that e_1 is orthogonal to e_2 by checking that $e_1 \cdot e_2 = 0$.

The Gram-Schmidt procedure continues iterating.

Thus, for $k = 2, \ldots, n-1$ we construct

$$u_{k+1} = a_{k+1} - (a_{k+1} \cdot e_1)e_1 - \cdots - (a_{k+1} \cdot e_k)e_k, \quad e_{k+1} = \frac{u_{k+1}}{||u_{k+1}||}$$

Here $(a_j \cdot e_i)$ can be interpreted as the linear least squares **regression coefficient** of a_j on e_i

- it is the inner product of a_j and e_i divided by the inner product of e_i where $e_i \cdot e_i = 1$, as *normalization* has assured us.
- this regression coefficient has an interpretation as being a **covariance** divided by a **variance**

It can be verified that

$$A = \begin{bmatrix} a_1 \mid a_2 \mid \cdots \mid a_n \end{bmatrix} = \begin{bmatrix} e_1 \mid e_2 \mid \cdots \mid e_n \end{bmatrix} \begin{bmatrix} a_1 \cdot e_1 & a_2 \cdot e_1 & \cdots & a_n \cdot e_1 \\ 0 & a_2 \cdot e_2 & \cdots & a_n \cdot e_2 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_n \cdot e_n \end{bmatrix}$$

Thus, we have constructed the decomposision

$$A = QR$$

where

$$Q = \begin{bmatrix} a_1 \mid a_2 \mid \cdots \mid a_n \end{bmatrix} = \begin{bmatrix} e_1 \mid e_2 \mid \cdots \mid e_n \end{bmatrix}$$

and

$$R = \begin{bmatrix} a_1 \cdot e_1 & a_2 \cdot e_1 & \cdots & a_n \cdot e_1 \\ 0 & a_2 \cdot e_2 & \cdots & a_n \cdot e_2 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_n \cdot e_n \end{bmatrix}$$

3.3.2 A not square

Now suppose that A is an $n \times m$ matrix where $m > n$.

Then a QR decomposition is

$$A = \begin{bmatrix} a_1 & a_2 & \cdots & a_m \end{bmatrix} = \begin{bmatrix} e_1 & e_2 & \cdots & e_n \end{bmatrix} \begin{bmatrix} a_1 \cdot e_1 & a_2 \cdot e_1 & \cdots & a_n \cdot e_1 & a_{n+1} \cdot e_1 & \cdots & a_m \cdot e_1 \\ 0 & a_2 \cdot e_2 & \cdots & a_n \cdot e_2 & a_{n+1} \cdot e_2 & \cdots & a_m \cdot e_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_n \cdot e_n & a_{n+1} \cdot e_n & \cdots & a_m \cdot e_n \end{bmatrix}$$

which implies that

$$\begin{aligned} a_1 &= (a_1 \cdot e_1)e_1 \\ a_2 &= (a_2 \cdot e_1)e_1 + (a_2 \cdot e_2)e_2 \\ &\vdots \quad \vdots \\ a_n &= (a_n \cdot e_1)e_1 + (a_n \cdot e_2)e_2 + \cdots + (a_n \cdot e_n)e_n \\ a_{n+1} &= (a_{n+1} \cdot e_1)e_1 + (a_{n+1} \cdot e_2)e_2 + \cdots + (a_{n+1} \cdot e_n)e_n \\ &\vdots \quad \vdots \\ a_m &= (a_m \cdot e_1)e_1 + (a_m \cdot e_2)e_2 + \cdots + (a_m \cdot e_n)e_n \end{aligned}$$

3.4 Some Code

Now let's write some homemade Python code to implement a QR decomposition by deploying the Gram-Schmidt process described above.

```
import numpy as np
from scipy.linalg import qr
```

```
def QR_Decomposition(A):
    n, m = A.shape # get the shape of A

    Q = np.empty((n, n)) # initialize matrix Q
    u = np.empty((n, n)) # initialize matrix u

    u[:, 0] = A[:, 0]
    Q[:, 0] = u[:, 0] / np.linalg.norm(u[:, 0])

    for i in range(1, n):

        u[:, i] = A[:, i]
        for j in range(i):
            u[:, i] -= (A[:, i] @ Q[:, j]) * Q[:, j] # get each u vector

        Q[:, i] = u[:, i] / np.linalg.norm(u[:, i]) # compute each e vetor

    R = np.zeros((n, m))
    for i in range(n):
        for j in range(i, m):
            R[i, j] = A[:, j] @ Q[:, i]

    return Q, R
```

The preceding code is fine but can benefit from some further housekeeping.

We want to do this because later in this notebook we want to compare results from using our homemade code above with the code for a QR that the Python `scipy` package delivers.

There can be be sign differences between the Q and R matrices produced by different numerical algorithms.

All of these are valid QR decompositions because of how the sign differences cancel out when we compute QR.

However, to make the results from our homemade function and the QR module in `scipy` comparable, let's require that Q have positive diagonal entries.

We do this by adjusting the signs of the columns in Q and the rows in R appropriately.

To accomplish this we'll define a pair of functions.

```
def diag_sign(A):
    "Compute the signs of the diagonal of matrix A"

    D = np.diag(np.sign(np.diag(A)))

    return D

def adjust_sign(Q, R):
    """
    Adjust the signs of the columns in Q and rows in R to
    impose positive diagonal of Q
    """

    D = diag_sign(Q)

    Q[:, :] = Q @ D
    R[:, :] = D @ R

    return Q, R
```

3.5 Example

Now let's do an example.

```
A = np.array([[1.0, 1.0, 0.0], [1.0, 0.0, 1.0], [0.0, 1.0, 1.0]])
# A = np.array([[1.0, 0.5, 0.2], [0.5, 0.5, 1.0], [0.0, 1.0, 1.0]])
# A = np.array([[1.0, 0.5, 0.2], [0.5, 0.5, 1.0]])

A
```

```
array([[1., 1., 0.],
       [1., 0., 1.],
       [0., 1., 1.]])
```

```
Q, R = adjust_sign(*QR_Decomposition(A))
```

```
Q
```

```
array([[ 0.70710678, -0.40824829, -0.57735027],
       [ 0.70710678,  0.40824829,  0.57735027],
       [ 0.        , -0.81649658,  0.57735027]])
```

```
R
```

```
array([[ 1.41421356,  0.70710678,  0.70710678],
       [ 0.        , -1.22474487, -0.40824829],
       [ 0.        ,  0.        ,  1.15470054]])
```

Let's compare outcomes with what the `scipy` package produces

```
Q_scipy, R_scipy = adjust_sign(*qr(A))
```

```
print('Our Q: \n', Q)
print('\n')
print('Scipy Q: \n', Q_scipy)
```

```
Our Q:
 [[ 0.70710678 -0.40824829 -0.57735027]
 [ 0.70710678  0.40824829  0.57735027]
 [ 0.         -0.81649658  0.57735027]]


Scipy Q:
 [[ 0.70710678 -0.40824829 -0.57735027]
 [ 0.70710678  0.40824829  0.57735027]
 [ 0.         -0.81649658  0.57735027]]
```

```
print('Our R: \n', R)
print('\n')
print('Scipy R: \n', R_scipy)
```

```
Our R:
 [[ 1.41421356  0.70710678  0.70710678]
 [ 0.         -1.22474487 -0.40824829]
 [ 0.          0.          1.15470054]]


Scipy R:
 [[ 1.41421356  0.70710678  0.70710678]
 [ 0.         -1.22474487 -0.40824829]
 [ 0.          0.          1.15470054]]
```

The above outcomes give us the good news that our homemade function agrees with what scipy produces.

Now let's do a QR decomposition for a rectangular matrix A that is $n \times m$ with $m > n$.

```
A = np.array([[1, 3, 4], [2, 0, 9]])
```

```
Q, R = adjust_sign(*QR_Decomposition(A))
Q, R
```

```
(array([[ 0.4472136 , -0.89442719],
        [ 0.89442719,  0.4472136 ]]),
```

(continues on next page)

(continued from previous page)

```
array([[ 2.23606798,  1.34164079,  9.8386991 ],
       [ 0.        , -2.68328157,  0.4472136 ]]))
```

```
Q_scipy, R_scipy = adjust_sign(*qr(A))
Q_scipy, R_scipy
```

```
(array([[ 0.4472136 , -0.89442719],
        [ 0.89442719,  0.4472136 ]]),
 array([[ 2.23606798,  1.34164079,  9.8386991 ],
        [ 0.        , -2.68328157,  0.4472136 ]]))
```

3.6 Using QR Decomposition to Compute Eigenvalues

Now for a useful fact about the QR algorithm.

The following iterations on the QR decomposition can be used to compute **eigenvalues** of a **square** matrix A.

Here is the algorithm:

1. Set $A_0 = A$ and form $A_0 = Q_0 R_0$
2. Form $A_1 = R_0 Q_0$. Note that A_1 is similar to A_0 (easy to verify) and so has the same eigenvalues.
3. Form $A_1 = Q_1 R_1$ (i.e., form the QR decomposition of A_1).
4. Form $A_2 = R_1 Q_1$ and then $A_2 = Q_2 R_2$.
5. Iterate to convergence.
6. Compute eigenvalues of A and compare them to the diagonal values of the limiting A_n found from this process.

Remark: this algorithm is close to one of the most efficient ways of computing eigenvalues!

Let's write some Python code to try out the algorithm

```
def QR_eigvals(A, tol=1e-12, maxiter=1000):
    "Find the eigenvalues of A using QR decomposition."

    A_old = np.copy(A)
    A_new = np.copy(A)

    diff = np.inf
    i = 0
    while (diff > tol) and (i < maxiter):
        A_old[:, :] = A_new
        Q, R = QR_Decomposition(A_old)

        A_new[:, :] = R @ Q

        diff = np.abs(A_new - A_old).max()
        i += 1

    eigvals = np.diag(A_new)

    return eigvals
```

Now let's try the code and compare the results with what `scipy.linalg.eigvals` gives us

Here goes

```
# experiment this with one random A matrix
A = np.random.random((3, 3))
```

```
sorted(QR_eigvals(A))
```

```
[np.float64(-0.45620377666658674),
 np.float64(0.0939850155333833),
 np.float64(1.7877040592425257)]
```

Compare with the `scipy` package.

```
sorted(np.linalg.eigvals(A))
```

```
[np.float64(-0.45620377666658746),
 np.float64(0.0939850155333826),
 np.float64(1.787704059242525)]
```

3.7 QR and PCA

There are interesting connections between the QR decomposition and principal components analysis (PCA).

Here are some.

1. Let X' be a $k \times n$ random matrix where the jth column is a random draw from $\mathcal{N}(\mu, \Sigma)$ where μ is $k \times 1$ vector of means and Σ is a $k \times k$ covariance matrix. We want $n >> k$ – this is an "econometrics example".
2. Form $X' = QR$ where Q is $k \times k$ and R is $k \times n$.
3. Form the eigenvalues of RR', i.e., we'll compute $RR' = \tilde{P}\Lambda\tilde{P}'$.
4. Form $X'X = Q\tilde{P}\Lambda\tilde{P}'Q'$ and compare it with the eigen decomposition $X'X = P\hat{\Lambda}P'$.
5. It will turn out that that $\Lambda = \hat{\Lambda}$ and that $P = Q\tilde{P}$.

Let's verify conjecture 5 with some Python code.

Start by simulating a random (n, k) matrix X.

```
k = 5
n = 1000

# generate some random moments
μ = np.random.random(size=k)
C = np.random.random((k, k))
Σ = C.T @ C
```

```
# X is random matrix where each column follows multivariate normal dist.
X = np.random.multivariate_normal(μ, Σ, size=n)
```

```
X.shape
```

```
(1000, 5)
```

Let's apply the QR decomposition to X'.

```
Q, R = adjust_sign(*QR_Decomposition(X.T))
```

Check the shapes of Q and R.

```
Q.shape, R.shape
```

```
((5, 5), (5, 1000))
```

Now we can construct $RR' = \tilde{P}\Lambda\tilde{P}'$ and form an eigen decomposition.

```
RR = R @ R.T

λ, P_tilde = np.linalg.eigh(RR)
Λ = np.diag(λ)
```

We can also apply the decomposition to $X'X = P\hat{\Lambda}P'$.

```
XX = X.T @ X

λ_hat, P = np.linalg.eigh(XX)
Λ_hat = np.diag(λ_hat)
```

Compare the eigenvalues that are on the diagonals of Λ and $\hat{\Lambda}$.

```
λ, λ_hat
```

```
(array([ 191.64050295,  296.4083486 ,  527.97765654,  762.0537361 ,
        7030.16761592]),
 array([ 191.64050295,  296.4083486 ,  527.97765654,  762.0537361 ,
        7030.16761592]))
```

Let's compare P and $Q\tilde{P}$.

Again we need to be careful about sign differences between the columns of P and $Q\tilde{P}$.

```
QP_tilde = Q @ P_tilde

np.abs(P @ diag_sign(P) - QP_tilde @ diag_sign(QP_tilde)).max()
```

```
np.float64(3.6637359812630166e-15)
```

Let's verify that $X'X$ can be decomposed as $Q\tilde{P}\Lambda\tilde{P}'Q'$.

```
QPΛPQ = Q @ P_tilde @ Λ @ P_tilde.T @ Q.T
```

```
np.abs(QPΛPQ - XX).max()
```

```
np.float64(2.7284841053187847e-12)
```

CHAPTER

FOUR

CIRCULANT MATRICES

4.1 Overview

This lecture describes circulant matrices and some of their properties.

Circulant matrices are useful because multiplying by them is closely connected to convolution, and their eigenvectors can be constructed using the Discrete Fourier Transform.

We use circulant matrices to connect several useful concepts, including

- convolution
- Fourier transforms
- permutation matrices

For background on eigenvalues and eigenvectors, see *Linear Algebra*; for another use of Fourier transforms and convolution, see *Fault Tree Uncertainties*.

Circulant matrices are also widely used in machine learning, for example, in image processing.

We begin by importing some Python packages

```
import numpy as np
import matplotlib.pyplot as plt
```

```
np.set_printoptions(precision=3, suppress=True)
```

4.2 Constructing a Circulant Matrix

To construct an $N \times N$ circulant matrix, we need only the first row, say,

$$\begin{bmatrix} c_0 & c_1 & c_2 & c_3 & c_4 & \cdots & c_{N-1} \end{bmatrix}.$$

After setting entries in the first row, the remaining rows of a circulant matrix are determined as follows:

$$C = \begin{bmatrix} c_0 & c_1 & c_2 & c_3 & c_4 & \cdots & c_{N-1} \\ c_{N-1} & c_0 & c_1 & c_2 & c_3 & \cdots & c_{N-2} \\ c_{N-2} & c_{N-1} & c_0 & c_1 & c_2 & \cdots & c_{N-3} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ c_3 & c_4 & c_5 & c_6 & c_7 & \cdots & c_2 \\ c_2 & c_3 & c_4 & c_5 & c_6 & \cdots & c_1 \\ c_1 & c_2 & c_3 & c_4 & c_5 & \cdots & c_0 \end{bmatrix} \tag{4.1}$$

This pattern can be formalized as follows.

Definition 4.2.1 (Circulant matrix)

An $N \times N$ matrix C is **circulant** if there are numbers $c_0, \dots, c_{N-1}$ such that

$$C_{ij} = c_{(j-i) \bmod N}, \qquad 0 \leq i, j \leq N-1.$$

Equivalently, each row is obtained from the previous row by shifting entries one step to the right.

It is also possible to construct a circulant matrix by creating the transpose of the above matrix, in which case only the first column needs to be specified.

Let's write some Python code to generate a circulant matrix.

```
def construct_circulant(row):

    N = row.size

    C = np.empty((N, N))

    for i in range(N):

        C[i, i:] = row[:N-i]
        C[i, :i] = row[N-i:]

    return C
```

```
# a simple case when N = 3
construct_circulant(np.array([1., 2., 3.]))
```

```
array([[1., 2., 3.],
       [3., 1., 2.],
       [2., 3., 1.]])
```

4.2.1 Some Properties of Circulant Matrices

Here are some useful properties:

Suppose that A and B are both circulant matrices of the same order and constructed using the same cyclic shift convention.

Then it can be verified that

- The transpose of a circulant matrix is a circulant matrix.
- $A + B$ is a circulant matrix
- AB is a circulant matrix
- $AB = BA$

Now consider a circulant matrix with first row

$$c = \begin{bmatrix} c_0 & c_1 & \cdots & c_{N-1} \end{bmatrix}$$

and consider a vector

$$a = \begin{bmatrix} a_0 & a_1 & \cdots & a_{N-1} \end{bmatrix}$$

The **convolution** of vectors c and a is defined as the vector $b = c * a$ with components

$$b_k = \sum_{i=0}^{N-1} c_{k-i} a_i \tag{4.2}$$

Here and below, indices such as $k - i$ are interpreted modulo N.

We use $*$ to denote **convolution** via the calculation described in equation (4.2).

It can be verified that the vector b satisfies

$$b = C^\top a$$

where $C^\top$ is the transpose of the circulant matrix defined in equation (4.1).

4.3 Connection to Permutation Matrix

A good way to construct a circulant matrix is to use a **permutation matrix**.

Before defining a permutation **matrix**, we'll define a **permutation**.

A **permutation** of a set of the set of non-negative integers $\{0, 1, 2, ...\}$ is a one-to-one mapping of the set into itself.

A permutation of a set $\{1, 2, ..., n\}$ rearranges the n integers in the set.

A permutation matrix is obtained by permuting the rows of an $n \times n$ identity matrix according to a permutation of the numbers 1 to n.

Thus, every row and every column contain precisely a single 1 with 0 everywhere else.

Every permutation corresponds to a unique permutation matrix.

For example, the $N \times N$ matrix

$$P = \begin{bmatrix} 0 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 1 \\ 1 & 0 & 0 & 0 & \cdots & 0 \end{bmatrix} \tag{4.3}$$

serves as a **cyclic shift** operator that, when applied to an $N \times 1$ vector h, shifts entries in rows 2 through N up one row and shifts the entry in row 1 to row N.

Eigenvalues of the cyclic shift permutation matrix P defined in equation (4.3) can be computed by constructing

$$P - \lambda I = \begin{bmatrix} -\lambda & 1 & 0 & 0 & \cdots & 0 \\ 0 & -\lambda & 1 & 0 & \cdots & 0 \\ 0 & 0 & -\lambda & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 1 \\ 1 & 0 & 0 & 0 & \cdots & -\lambda \end{bmatrix}$$

and solving

$$\det(P - \lambda I) = (-1)^N (\lambda^N - 1) = 0$$

Eigenvalues λ_i can be complex.

Magnitudes $\mid \lambda_i \mid$ of these eigenvalues λ_i all equal 1.

Thus, **singular values** of the permutation matrix P defined in equation (4.3) all equal 1.

It can be verified that permutation matrices are orthogonal matrices:

$$PP^\top = I$$

4.4 Examples with Python

Let's write some Python code to illustrate these ideas.

```
def construct_cyclic_shift_matrix(N):

    P = np.zeros((N, N))

    for i in range(N-1):
        P[i, i+1] = 1
    P[-1, 0] = 1

    return P
```

```
P4 = construct_cyclic_shift_matrix(4)
P4
```

```
array([[0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.],
       [1., 0., 0., 0.]])
```

```
# compute the eigenvalues and eigenvectors
λ, Q = np.linalg.eig(P4)
```

```
for i in range(4):
    print(f'λ{i} = {λ[i]:.1f} \nvec{i} = {Q[:, i]}\n')
```

```
λ0 = -1.0+0.0j
vec0 = [-0.5+0.j  0.5+0.j -0.5+0.j  0.5+0.j]

λ1 = 0.0+1.0j
vec1 = [-0. -0.5j  0.5+0.j   0. +0.5j -0.5-0.j ]

λ2 = 0.0-1.0j
vec2 = [-0. +0.5j  0.5-0.j   0. -0.5j -0.5+0.j ]

λ3 = 1.0+0.0j
vec3 = [-0.5+0.j -0.5+0.j -0.5+0.j -0.5+0.j]
```

In graphs below, we shall portray eigenvalues of a shift permutation matrix in the complex plane.

These eigenvalues are uniformly distributed along the unit circle.

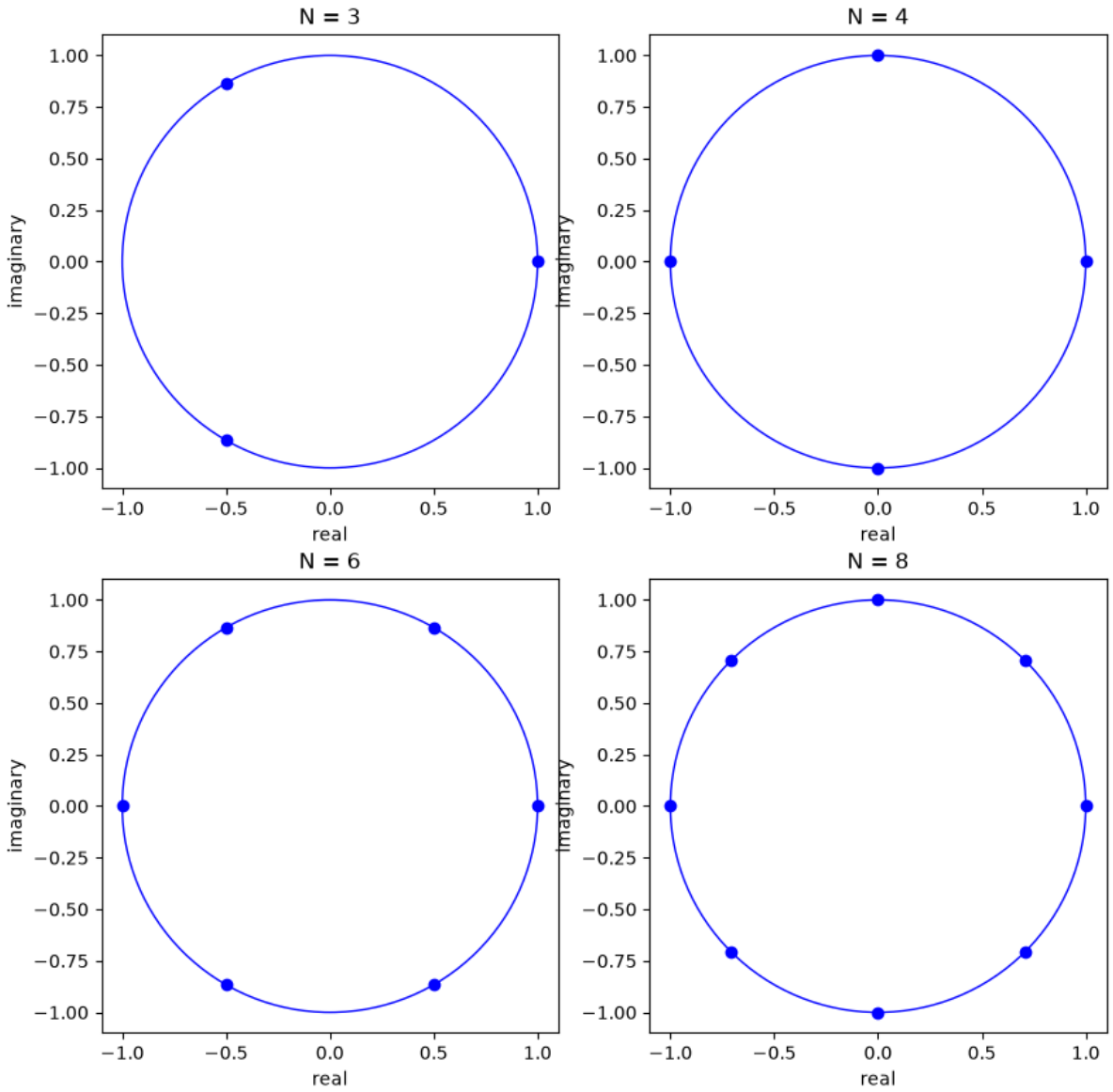
They are the n **roots of unity**, meaning they are the n numbers z that solve $z^n = 1$, where z is a complex number.

In particular, the n roots of unity are

$$z = \exp\left(\frac{2\pi jk}{N}\right), \quad k = 0, \ldots, N-1$$

where j denotes the purely imaginary unit number.

```
fig, ax = plt.subplots(2, 2, figsize=(10, 10))

for i, N in enumerate([3, 4, 6, 8]):

    row_i = i // 2
    col_i = i % 2

    P = construct_cyclic_shift_matrix(N)
    λ, Q = np.linalg.eig(P)

    circ = plt.Circle((0, 0), radius=1, edgecolor='b', facecolor='None')
    ax[row_i, col_i].add_patch(circ)

    for j in range(N):
        ax[row_i, col_i].scatter(λ[j].real, λ[j].imag, c='b')

    ax[row_i, col_i].set_title(f'N = {N}')
    ax[row_i, col_i].set_xlabel('real')
    ax[row_i, col_i].set_ylabel('imaginary')

plt.show()
```



For a vector of coefficients $\{c_i\}_{i=0}^{n-1}$, eigenvectors of P are also eigenvectors of

$$C = c_0 I + c_1 P + c_2 P^2 + \cdots + c_{N-1} P^{N-1}.$$

Consider an example in which $N = 8$ and let $w = e^{-2\pi j/N}$.

It can be verified that the matrix F_8 of eigenvectors of P_8 is

$$F_8 = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & w & w^2 & \cdots & w^7 \\ 1 & w^2 & w^4 & \cdots & w^{14} \\ 1 & w^3 & w^6 & \cdots & w^{21} \\ 1 & w^4 & w^8 & \cdots & w^{28} \\ 1 & w^5 & w^{10} & \cdots & w^{35} \\ 1 & w^6 & w^{12} & \cdots & w^{42} \\ 1 & w^7 & w^{14} & \cdots & w^{49} \end{bmatrix}$$

The matrix F_8 defines a Discrete Fourier Transform.

To convert it into an orthogonal eigenvector matrix, we can simply normalize it by dividing every entry by $\sqrt{8}$.

- stare at the first column of F_8 above to convince yourself of this fact

The eigenvalues corresponding to each eigenvector are $\{w^j\}_{j=0}^7$ in order.

```
def construct_F(N):

    w = np.e ** (-complex(0, 2*np.pi/N))

    F = np.ones((N, N), dtype=complex)
    for i in range(1, N):
        F[i, 1:] = w ** (i * np.arange(1, N))

    return F, w
```

```
F8, w = construct_F(8)
```

```
w
```

```
(0.7071067811865476-0.7071067811865475j)
```

```
F8
```

```
array([[ 1.   +0.j   ,  1.   +0.j   ,  1.   +0.j   ,  1.   +0.j   ,
         1.   +0.j   ,  1.   +0.j   ,  1.   +0.j   ,  1.   +0.j   ],
       [ 1.   +0.j   ,  0.707-0.707j,  0.   -1.j   , -0.707-0.707j,
        -1.   -0.j   , -0.707+0.707j, -0.   +1.j   ,  0.707+0.707j],
       [ 1.   +0.j   ,  0.   -1.j   , -1.   -0.j   , -0.   +1.j   ,
         1.   +0.j   ,  0.   -1.j   , -1.   -0.j   , -0.   +1.j   ],
       [ 1.   +0.j   , -0.707-0.707j, -0.   +1.j   ,  0.707-0.707j,
        -1.   -0.j   ,  0.707+0.707j,  0.   -1.j   , -0.707+0.707j],
       [ 1.   +0.j   , -1.   -0.j   ,  1.   +0.j   , -1.   -0.j   ,
         1.   +0.j   , -1.   -0.j   ,  1.   +0.j   , -1.   -0.j   ],
       [ 1.   +0.j   , -0.707+0.707j,  0.   -1.j   ,  0.707+0.707j,
        -1.   -0.j   ,  0.707-0.707j, -0.   +1.j   , -0.707-0.707j],
       [ 1.   +0.j   , -0.   +1.j   , -1.   -0.j   ,  0.   -1.j   ,
         1.   +0.j   , -0.   +1.j   , -1.   -0.j   ,  0.   -1.j   ],
       [ 1.   +0.j   ,  0.707+0.707j, -0.   +1.j   , -0.707+0.707j,
        -1.   -0.j   , -0.707-0.707j,  0.   -1.j   ,  0.707-0.707j]])
```

```
# normalize
Q8 = F8 / np.sqrt(8)
```

```
# verify the orthogonality (unitarity)
Q8 @ np.conjugate(Q8)
```

```
array([[ 1.+0.j, -0.+0.j, -0.+0.j, -0.+0.j, -0.+0.j,  0.+0.j,  0.+0.j,
         0.+0.j],
       [-0.-0.j,  1.+0.j, -0.+0.j, -0.+0.j, -0.+0.j, -0.+0.j,  0.+0.j,
         0.+0.j],
       [-0.-0.j, -0.-0.j,  1.+0.j, -0.+0.j, -0.+0.j, -0.+0.j,  0.+0.j,
         0.+0.j],
       [-0.-0.j, -0.-0.j, -0.-0.j,  1.+0.j, -0.+0.j, -0.+0.j, -0.+0.j,
        -0.+0.j],
       [-0.-0.j, -0.-0.j, -0.-0.j, -0.-0.j,  1.+0.j, -0.+0.j, -0.+0.j,
```

(continues on next page)

(continued from previous page)

```
        -0.+0.j],
       [ 0.-0.j, -0.-0.j, -0.-0.j, -0.-0.j, -0.-0.j,  1.+0.j, -0.+0.j,
        -0.+0.j],
       [ 0.-0.j,  0.-0.j,  0.-0.j, -0.-0.j, -0.-0.j, -0.-0.j,  1.+0.j,
        -0.+0.j],
       [ 0.-0.j,  0.-0.j,  0.-0.j, -0.-0.j, -0.-0.j, -0.-0.j, -0.-0.j,
         1.+0.j]])
```

Let's verify that kth column of Q_8 is an eigenvector of P_8 with an eigenvalue w^k.

```
P8 = construct_cyclic_shift_matrix(8)
```

```
diff_arr = np.empty(8, dtype=complex)
for j in range(8):
    diff = P8 @ Q8[:, j] - w ** j * Q8[:, j]
    diff_arr[j] = diff @ diff.T
```

```
diff_arr
```

```
array([ 0.+0.j, -0.+0.j, -0.+0.j, -0.+0.j, -0.+0.j, -0.+0.j, -0.+0.j,
       -0.+0.j])
```

4.5 Associated Permutation Matrix

Next, we execute calculations to verify that the circulant matrix C defined in equation (4.1) can be written as

$$C = c_0 I + c_1 P + \cdots + c_{N-1} P^{N-1}$$

and that every eigenvector of P is also an eigenvector of C.

We illustrate this for $N = 8$ case.

```
c = np.random.random(8)
```

```
c
```

```
array([0.207, 0.299, 0.682, 0.468, 0.391, 0.854, 0.753, 0.964])
```

```
C8 = construct_circulant(c)
```

Compute $c_0 I + c_1 P + \cdots + c_{N-1} P^{N-1}$.

```
N = 8

C = np.zeros((N, N))
P = np.eye(N)

for i in range(N):
    C += c[i] * P
    P = P8 @ P
```

```
C
```

```
array([[0.207, 0.299, 0.682, 0.468, 0.391, 0.854, 0.753, 0.964],
       [0.964, 0.207, 0.299, 0.682, 0.468, 0.391, 0.854, 0.753],
       [0.753, 0.964, 0.207, 0.299, 0.682, 0.468, 0.391, 0.854],
       [0.854, 0.753, 0.964, 0.207, 0.299, 0.682, 0.468, 0.391],
       [0.391, 0.854, 0.753, 0.964, 0.207, 0.299, 0.682, 0.468],
       [0.468, 0.391, 0.854, 0.753, 0.964, 0.207, 0.299, 0.682],
       [0.682, 0.468, 0.391, 0.854, 0.753, 0.964, 0.207, 0.299],
       [0.299, 0.682, 0.468, 0.391, 0.854, 0.753, 0.964, 0.207]])
```

```
C8
```

```
array([[0.207, 0.299, 0.682, 0.468, 0.391, 0.854, 0.753, 0.964],
       [0.964, 0.207, 0.299, 0.682, 0.468, 0.391, 0.854, 0.753],
       [0.753, 0.964, 0.207, 0.299, 0.682, 0.468, 0.391, 0.854],
       [0.854, 0.753, 0.964, 0.207, 0.299, 0.682, 0.468, 0.391],
       [0.391, 0.854, 0.753, 0.964, 0.207, 0.299, 0.682, 0.468],
       [0.468, 0.391, 0.854, 0.753, 0.964, 0.207, 0.299, 0.682],
       [0.682, 0.468, 0.391, 0.854, 0.753, 0.964, 0.207, 0.299],
       [0.299, 0.682, 0.468, 0.391, 0.854, 0.753, 0.964, 0.207]])
```

Now let's compute the difference between two circulant matrices that we have constructed in two different ways.

```
np.abs(C - C8).max()
```

```
np.float64(0.0)
```

The jth column of Q_8 is an eigenvector of C_8 associated with eigenvalue $\sum_{k=0}^{7} c_k w^{jk}$.

```
λ_C8 = np.zeros(8, dtype=complex)

for j in range(8):
    for k in range(8):
        λ_C8[j] += c[k] * w ** (j * k)
```

```
λ_C8
```

```
array([ 4.619+0.j   , -0.226+0.813j, -0.837+0.28j , -0.143+0.673j,
       -0.553-0.j   , -0.143-0.673j, -0.837-0.28j , -0.226-0.813j])
```

We can verify this by comparing `C8 @ Q8[:, j]` with `λ_C8[j] * Q8[:, j]`.

```
# verify
for j in range(8):
    diff = C8 @ Q8[:, j] - λ_C8[j] * Q8[:, j]
    print(diff)
```

```
[0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
[ 0.+0.j  0.-0.j -0.-0.j -0.-0.j -0.-0.j -0.+0.j -0.+0.j  0.+0.j]
[ 0.-0.j -0.-0.j -0.-0.j -0.+0.j  0.+0.j  0.-0.j -0.-0.j -0.+0.j]
[-0.+0.j  0.-0.j -0.-0.j  0.-0.j -0.-0.j -0.+0.j  0.-0.j -0.+0.j]
[0.+0.j 0.-0.j 0.+0.j 0.-0.j 0.+0.j 0.-0.j 0.+0.j 0.-0.j]
[ 0.+0.j -0.-0.j  0.-0.j -0.-0.j  0.-0.j -0.+0.j -0.-0.j  0.+0.j]
```

(continues on next page)

(continued from previous page)

```
[-0.+0.j -0.-0.j  0.-0.j  0.+0.j -0.+0.j -0.-0.j  0.-0.j  0.+0.j]
[-0.+0.j  0.-0.j  0.-0.j  0.-0.j  0.-0.j  0.+0.j  0.+0.j -0.+0.j]
```

4.6 Discrete Fourier Transform

The **Discrete Fourier Transform** (DFT) allows us to represent a discrete time sequence as a weighted sum of complex sinusoids.

Consider a sequence of N real numbers $\{x_j\}_{j=0}^{N-1}$.

The **Discrete Fourier Transform** maps $\{x_j\}_{j=0}^{N-1}$ into a sequence of complex numbers $\{X_k\}_{k=0}^{N-1}$

where

$$X_k = \sum_{n=0}^{N-1} x_n e^{-2\pi \frac{kn}{N} i}$$

```
def DFT(x):
    "The discrete Fourier transform."

    N = len(x)
    w = np.e ** (-complex(0, 2*np.pi/N))

    X = np.zeros(N, dtype=complex)
    for k in range(N):
        for n in range(N):
            X[k] += x[n] * w ** (k * n)

    return X
```

Consider the following example.

$$x_n = \begin{cases} 1/2 & n = 0, 1 \\ 0 & \text{otherwise} \end{cases}$$

```
x = np.zeros(10)
x[0:2] = 1/2
```

```
x
```

```
array([0.5, 0.5, 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ])
```

Apply a discrete Fourier transform.

```
X = DFT(x)
```

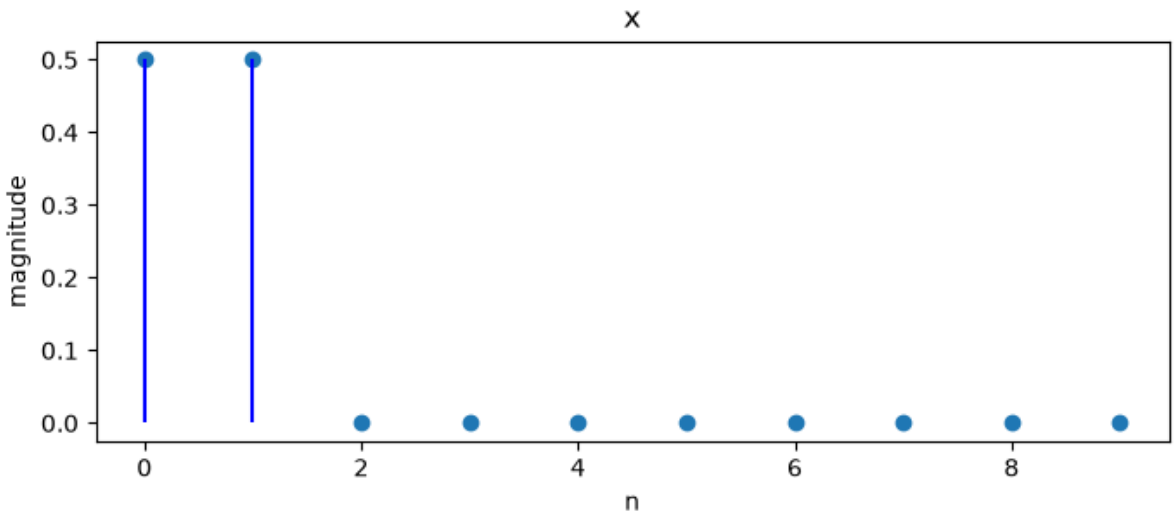
```
X
```

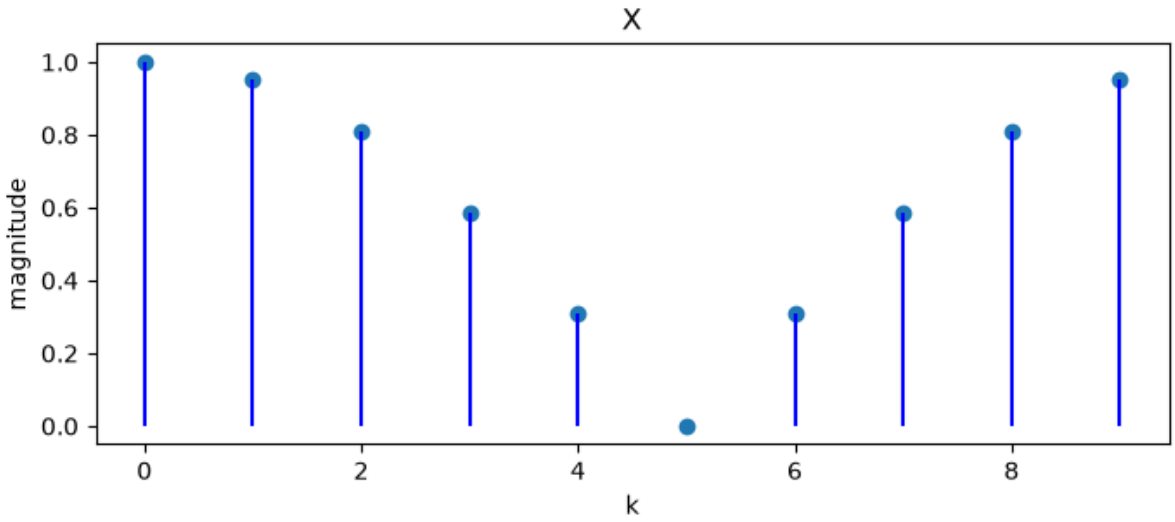
```
array([ 1.   +0.j   ,  0.905-0.294j,  0.655-0.476j,  0.345-0.476j,
        0.095-0.294j, -0.   +0.j   ,  0.095+0.294j,  0.345+0.476j,
        0.655+0.476j,  0.905+0.294j])
```

We can plot magnitudes of a sequence of numbers and the associated discrete Fourier transform.

```
def plot_magnitude(x=None, X=None):

    data = []
    names = []
    xs = []
    if (x is not None):
        data.append(x)
        names.append('x')
        xs.append('n')
    if (X is not None):
        data.append(X)
        names.append('X')
        xs.append('k')

    num = len(data)
    for i in range(num):
        n = data[i].size
        plt.figure(figsize=(8, 3))
        plt.scatter(range(n), np.abs(data[i]))
        plt.vlines(range(n), 0, np.abs(data[i]), color='b')

        plt.xlabel(xs[i])
        plt.ylabel('magnitude')
        plt.title(names[i])
        plt.show()
```

```
plot_magnitude(x=x, X=X)
```





The **inverse Fourier transform** transforms a Fourier transform X of x back to x.

The inverse Fourier transform is defined as

$$x_n = \sum_{k=0}^{N-1} \frac{1}{N} X_k e^{2\pi(\frac{kn}{N})i}, \quad n = 0, 1, \dots, N-1$$

```
def inverse_DFT(X):

    N = len(X)
    w = np.e ** (complex(0, 2*np.pi/N))

    x = np.zeros(N, dtype=complex)
    for n in range(N):
        for k in range(N):
            x[n] += X[k] * w ** (k * n) / N

    return x
```

```
inverse_DFT(X)
```

```
array([ 0.5+0.j,  0.5-0.j, -0. -0.j, -0. -0.j, -0. -0.j, -0. -0.j, -0. -0.j,
       -0. +0.j, -0. +0.j, -0. +0.j, -0. +0.j])
```

Another example is

$$x_n = 2\cos\left(2\pi \frac{11}{40} n\right), \; n = 0, 1, 2, \cdots 19$$

Since $N = 20$, we cannot use an integer multiple of $\frac{1}{20}$ to represent a frequency $\frac{11}{40}$.

To handle this, we shall end up using all N of the available frequencies in the DFT.

Since $\frac{11}{40}$ is in between $\frac{10}{40}$ and $\frac{12}{40}$ (each of which is an integer multiple of $\frac{1}{20}$), the complex coefficients in the DFT have their largest magnitudes at $k = 5, 6, 15, 16$, not just at a single frequency.

```
N = 20
x = np.empty(N)
```

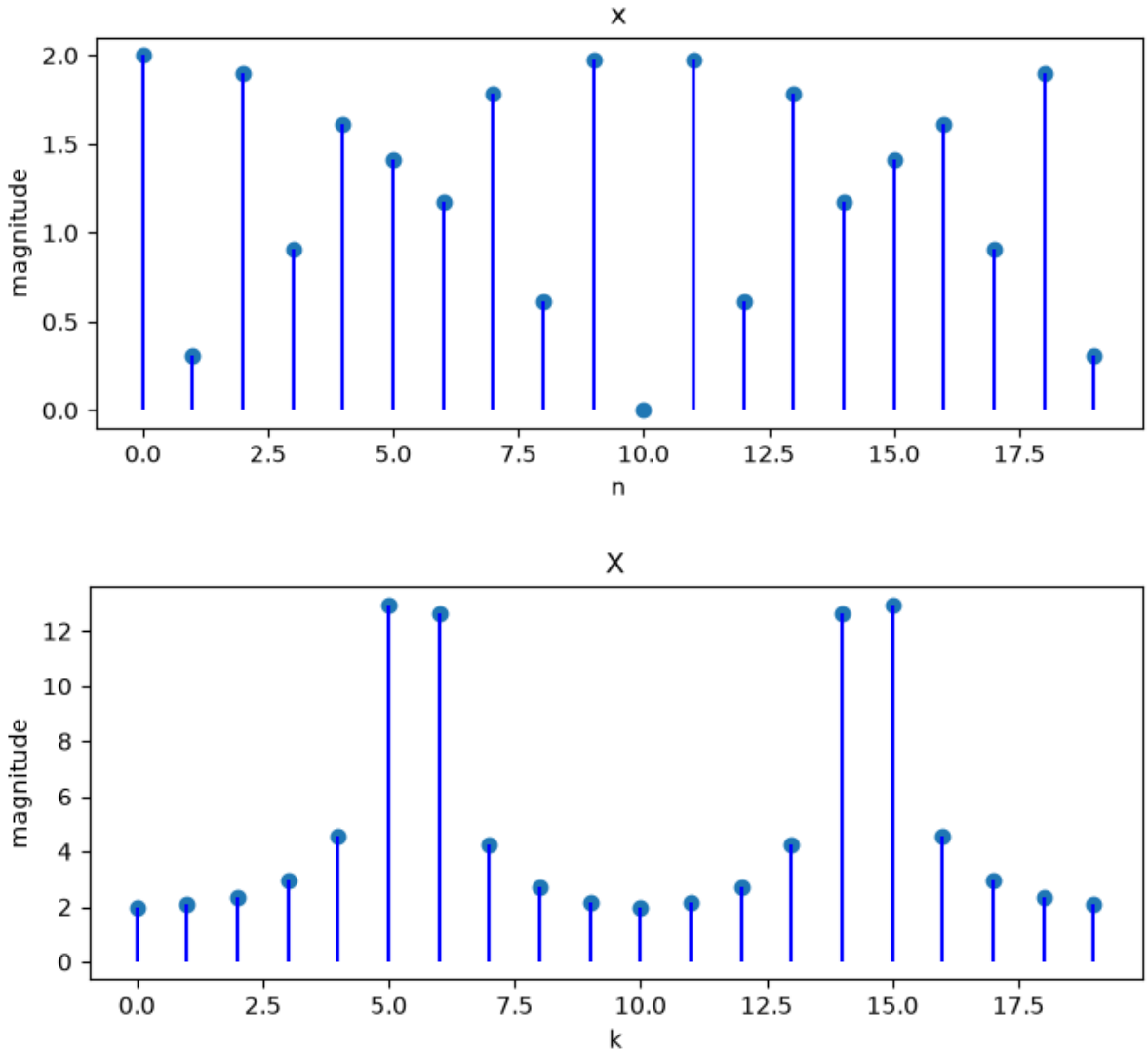
(continues on next page)

(continued from previous page)

```
for j in range(N):
    x[j] = 2 * np.cos(2 * np.pi * 11 * j / 40)
```

```
X = DFT(x)
```

```
plot_magnitude(x=x, X=X)
```



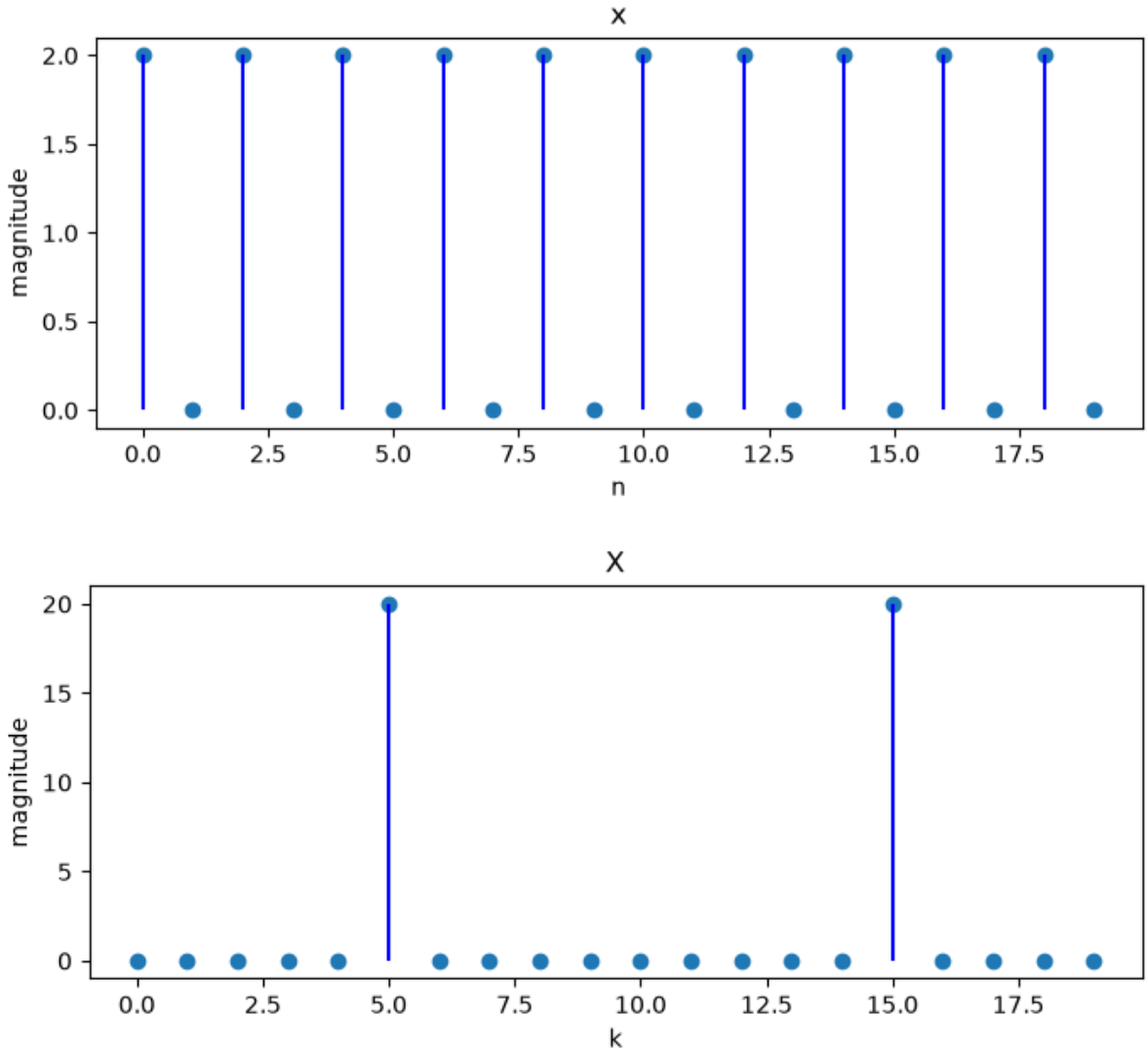
What happens if we change the last example to $x_n = 2\cos\left(2\pi\frac{10}{40}n\right)$?

Note that $\frac{10}{40}$ is an integer multiple of $\frac{1}{20}$.

```
N = 20
x = np.empty(N)

for j in range(N):
    x[j] = 2 * np.cos(2 * np.pi * 10 * j / 40)
```

```
X = DFT(x)
```

```
plot_magnitude(x=x, X=X)
```



If we represent the discrete Fourier transform as a matrix, we discover that it equals the matrix F_N of eigenvectors of the permutation matrix P_N.

We can use the example where $x_n = 2\cos\left(2\pi\frac{11}{40}n\right),\ n = 0, 1, 2, \cdots 19$ to illustrate this.

```
N = 20
x = np.empty(N)

for j in range(N):
    x[j] = 2 * np.cos(2 * np.pi * 11 * j / 40)
```

```
x
```

```
array([ 2.   , -0.313, -1.902,  0.908,  1.618, -1.414, -1.176,  1.782,
        0.618, -1.975, -0.   ,  1.975, -0.618, -1.782,  1.176,  1.414,
       -1.618, -0.908,  1.902,  0.313])
```

First use the summation formula to transform x to X.

```
X = DFT(x)
X
```

```
array([2. +0.j   , 2. +0.558j, 2. +1.218j, 2. +2.174j, 2. +4.087j,
       2.+12.785j, 2.-12.466j, 2. -3.751j, 2. -1.801j, 2. -0.778j,
       2. -0.j   , 2. +0.778j, 2. +1.801j, 2. +3.751j, 2.+12.466j,
       2.-12.785j, 2. -4.087j, 2. -2.174j, 2. -1.218j, 2. -0.558j])
```

Now let's evaluate the outcome of postmultiplying the eigenvector matrix F_{20} by the vector x, a product that we claim should equal the Fourier transform of the sequence $\{x_n\}_{n=0}^{N-1}$.

```
F20, _ = construct_F(20)
```

```
F20 @ x
```

```
array([2. +0.j   , 2. +0.558j, 2. +1.218j, 2. +2.174j, 2. +4.087j,
       2.+12.785j, 2.-12.466j, 2. -3.751j, 2. -1.801j, 2. -0.778j,
       2. -0.j   , 2. +0.778j, 2. +1.801j, 2. +3.751j, 2.+12.466j,
       2.-12.785j, 2. -4.087j, 2. -2.174j, 2. -1.218j, 2. -0.558j])
```

Similarly, the inverse DFT can be expressed as an inverse DFT matrix F_{20}^{-1}.

```
F20_inv = np.linalg.inv(F20)
F20_inv @ X
```

```
array([ 2.   -0.j, -0.313-0.j, -1.902+0.j,  0.908-0.j,  1.618-0.j,
       -1.414+0.j, -1.176+0.j,  1.782+0.j,  0.618-0.j, -1.975-0.j,
       -0.   +0.j,  1.975-0.j, -0.618-0.j, -1.782+0.j,  1.176+0.j,
        1.414-0.j, -1.618-0.j, -0.908+0.j,  1.902+0.j,  0.313-0.j])
```

CHAPTER

FIVE

SINGULAR VALUE DECOMPOSITION (SVD)

5.1 Overview

The **singular value decomposition** (SVD) is a work-horse in applications of least squares projection that form foundations for many statistical and machine learning methods.

After defining the SVD, we'll describe how it connects to

- **four fundamental spaces** of linear algebra
- under-determined and over-determined **least squares regressions**
- **principal components analysis** (PCA)

Like principal components analysis (PCA), DMD can be thought of as a data-reduction procedure that represents salient patterns by projecting data onto a limited set of factors.

In a sequel to this lecture about *Dynamic Mode Decompositions*, we'll describe how SVD's provide ways rapidly to compute reduced-order approximations to first-order Vector Autoregressions (VARs).

5.2 The Setting

Let X be an $m \times n$ matrix of rank p.

Necessarily, $p \leq \min(m, n)$.

In much of this lecture, we'll think of X as a matrix of data in which

- each column is an **individual** – a time period or person, depending on the application
- each row is a **random variable** describing an attribute of a time period or a person, depending on the application

We'll be interested in two situations

- A **short and fat** case in which $m << n$, so that there are many more columns (individuals) than rows (attributes).
- A **tall and skinny** case in which $m >> n$, so that there are many more rows (attributes) than columns (individuals).

We'll apply a **singular value decomposition** of X in both situations.

In the $m << n$ case in which there are many more individuals n than attributes m, we can calculate sample moments of a joint distribution by taking averages across observations of functions of the observations.

In this $m << n$ case, we'll look for patterns by using a singular value decomposition to do a principal components analysis (PCA).

In the $m >> n$ case in which there are many more attributes m than individuals n and when we are in a time-series setting in which n equals the number of time periods covered in the data set X, we'll proceed in a different way.

We'll again use a singular value decomposition, but now to construct a **dynamic mode decomposition** (DMD)

5.3 Singular Value Decomposition

A **singular value decomposition** of an $m \times n$ matrix X of rank $p \leq \min(m, n)$ is

$$X = U\Sigma V^\top \tag{5.1}$$

where

$$\begin{aligned} UU^\top &= I \qquad U^\top U = I \\ VV^\top &= I \qquad V^\top V = I \end{aligned}$$

and

- U is an $m \times m$ orthogonal matrix of **left singular vectors** of X
- Columns of U are eigenvectors of $XX^\top$
- V is an $n \times n$ orthogonal matrix of **right singular vectors** of X
- Columns of V are eigenvectors of $X^\top X$
- Σ is an $m \times n$ matrix in which the first p places on its main diagonal are positive numbers $\sigma_1, \sigma_2, \ldots, \sigma_p$ called **singular values**; remaining entries of Σ are all zero
- The p singular values are positive square roots of the eigenvalues of the $m \times m$ matrix $XX^\top$ and also of the $n \times n$ matrix $X^\top X$
- We adopt a convention that when U is a complex valued matrix, $U^\top$ denotes the **conjugate-transpose** or **Hermitian-transpose** of U, meaning that $U^\top_{ij}$ is the complex conjugate of U_{ji}.
- Similarly, when V is a complex valued matrix, $V^\top$ denotes the **conjugate-transpose** or **Hermitian-transpose** of V

The matrices U, Σ, V entail linear transformations that reshape in vectors in the following ways:

- multiplying vectors by the unitary matrices U and V **rotates** them, but leaves **angles between vectors** and **lengths of vectors** unchanged.
- multiplying vectors by the diagonal matrix Σ leaves **angles between vectors** unchanged but **rescales** vectors.

Thus, representation (5.1) asserts that multiplying an $n \times 1$ vector y by the $m \times n$ matrix X amounts to performing the following three multiplications of y sequentially:

- **rotating** y by computing $V^\top y$
- **rescaling** $V^\top y$ by multiplying it by Σ
- **rotating** $\Sigma V^\top y$ by multiplying it by U

This structure of the $m \times n$ matrix X opens the door to constructing systems of data **encoders** and **decoders**.

Thus,

- $V^\top y$ is an encoder
- Σ is an operator to be applied to the encoded data
- U is a decoder to be applied to the output from applying operator Σ to the encoded data

We'll apply this circle of ideas later in this lecture when we study Dynamic Mode Decomposition.

Road Ahead

What we have described above is called a **full** SVD.

In a **full** SVD, the shapes of U, Σ, and V are (m, m), (m, n), (n, n), respectively.

Later we'll also describe an **economy** or **reduced** SVD.

Before we study a **reduced** SVD we'll say a little more about properties of a **full** SVD.

5.4 Four Fundamental Subspaces

Let $\mathcal{C}$ denote a column space, $\mathcal{N}$ denote a null space, and $\mathcal{R}$ denote a row space.

Let's start by recalling the four fundamental subspaces of an $m \times n$ matrix X of rank p.

- The **column space** of X, denoted $\mathcal{C}(X)$, is the span of the columns of X, i.e., all vectors y that can be written as linear combinations of columns of X. Its dimension is p.
- The **null space** of X, denoted $\mathcal{N}(X)$ consists of all vectors y that satisfy $Xy = 0$. Its dimension is $n - p$.
- The **row space** of X, denoted $\mathcal{R}(X)$ is the column space of $X^\top$. It consists of all vectors z that can be written as linear combinations of rows of X. Its dimension is p.
- The **left null space** of X, denoted $\mathcal{N}(X^\top)$, consist of all vectors z such that $X^\top z = 0$. Its dimension is $m - p$.

For a full SVD of a matrix X, the matrix U of left singular vectors and the matrix V of right singular vectors contain orthogonal bases for all four subspaces.

They form two pairs of orthogonal subspaces that we'll describe now.

Let $u_i, i = 1, \ldots, m$ be the m column vectors of U and let $v_i, i = 1, \ldots, n$ be the n column vectors of V.

Let's write the full SVD of X as

$$X = \begin{bmatrix} U_L & U_R \end{bmatrix} \begin{bmatrix} \Sigma_p & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} V_L & V_R \end{bmatrix}^\top \tag{5.2}$$

where Σ_p is a $p \times p$ diagonal matrix with the p singular values on the diagonal and

$$\begin{aligned} U_L &= \begin{bmatrix} u_1 & \cdots & u_p \end{bmatrix}, \quad U_R = \begin{bmatrix} u_{p+1} & \cdots u_m \end{bmatrix} \\ V_L &= \begin{bmatrix} v_1 & \cdots & v_p \end{bmatrix}, \quad U_R = \begin{bmatrix} v_{p+1} & \cdots u_n \end{bmatrix} \end{aligned}$$

Representation (5.2) implies that

$$X \begin{bmatrix} V_L & V_R \end{bmatrix} = \begin{bmatrix} U_L & U_R \end{bmatrix} \begin{bmatrix} \Sigma_p & 0 \\ 0 & 0 \end{bmatrix}$$

or

$$\begin{aligned} XV_L &= U_L \Sigma_p \\ XV_R &= 0 \end{aligned} \tag{5.3}$$

or

$$\begin{aligned} Xv_i &= \sigma_i u_i, \quad i = 1, \ldots, p \\ Xv_i &= 0, \quad i = p+1, \ldots, n \end{aligned} \tag{5.4}$$

Equations (5.4) tell how the transformation X maps a pair of orthonormal vectors v_i, v_j for i and j both less than or equal to the rank p of X into a pair of orthonormal vectors u_i, u_j.

Equations (5.3) assert that

$$\begin{aligned}\mathcal{C}(X) &= \mathcal{C}(U_L)\\ \mathcal{N}(X) &= \mathcal{C}(V_R)\end{aligned}$$

Taking transposes on both sides of representation (5.2) implies

$$X^\top \begin{bmatrix} U_L & U_R \end{bmatrix} = \begin{bmatrix} V_L & V_R \end{bmatrix} \begin{bmatrix} \Sigma_p & 0 \\ 0 & 0 \end{bmatrix}$$

or

$$\begin{aligned} X^\top U_L &= V_L \Sigma_p \\ X^\top U_R &= 0 \end{aligned} \tag{5.5}$$

or

$$\begin{aligned} X^\top u_i &= \sigma_i v_i, \quad i = 1, \ldots, p \\ X^\top u_i &= 0 \quad i = p+1, \ldots, m \end{aligned} \tag{5.6}$$

Notice how equations (5.6) assert that the transformation $X^\top$ maps a pair of distinct orthonormal vectors u_i, u_j for i and j both less than or equal to the rank p of X into a pair of distinct orthonormal vectors v_i, v_j .

Equations (5.5) assert that

$$\begin{aligned}\mathcal{R}(X) \equiv \mathcal{C}(X^\top) &= \mathcal{C}(V_L)\\ \mathcal{N}(X^\top) &= \mathcal{C}(U_R)\end{aligned}$$

Thus, taken together, the systems of equations (5.3) and (5.5) describe the four fundamental subspaces of X in the following ways:

$$\begin{aligned} \mathcal{C}(X) &= \mathcal{C}(U_L) \\ \mathcal{N}(X^\top) &= \mathcal{C}(U_R) \\ \mathcal{R}(X) \equiv \mathcal{C}(X^\top) &= \mathcal{C}(V_L) \\ \mathcal{N}(X) &= \mathcal{C}(V_R) \end{aligned} \tag{5.7}$$

Since U and V are both orthonormal matrices, collection (5.7) asserts that

- U_L is an orthonormal basis for the column space of X
- U_R is an orthonormal basis for the null space of $X^\top$
- V_L is an orthonormal basis for the row space of X
- V_R is an orthonormal basis for the null space of X

We have verified the four claims in (5.7) simply by performing the multiplications called for by the right side of (5.2) and reading them.

The claims in (5.7) and the fact that U and V are both unitary (i.e, orthonormal) matrices imply that

- the column space of X is orthogonal to the null space of $X^\top$
- the null space of X is orthogonal to the row space of X

Sometimes these properties are described with the following two pairs of orthogonal complement subspaces:

- $\mathcal{C}(X)$ is the orthogonal complement of $\mathcal{N}(X^\top)$
- $\mathcal{R}(X)$ is the orthogonal complement $\mathcal{N}(X)$

Let's do an example.

```
import numpy as np
import numpy.linalg as LA
import matplotlib.pyplot as plt
```

Having imported these modules, let's do the example.

```
np.set_printoptions(precision=2)

# Define the matrix
A = np.array([[1, 2, 3, 4, 5],
              [2, 3, 4, 5, 6],
              [3, 4, 5, 6, 7],
              [4, 5, 6, 7, 8],
              [5, 6, 7, 8, 9]])

# Compute the SVD of the matrix
U, S, V = np.linalg.svd(A,full_matrices=True)

# Compute the rank of the matrix
rank = np.linalg.matrix_rank(A)

# Print the rank of the matrix
print("Rank of matrix:\n", rank)
print("S: \n", S)

# Compute the four fundamental subspaces
row_space = U[:, :rank]
col_space = V[:, :rank]
null_space = V[:, rank:]
left_null_space = U[:, rank:]


print("U:\n", U)
print("Column space:\n", col_space)
print("Left null space:\n", left_null_space)
print("V.T:\n", V.T)
print("Row space:\n", row_space.T)
print("Right null space:\n", null_space.T)
```

```
Rank of matrix:
 2
S:
 [2.69e+01 1.86e+00 1.20e-15 2.24e-16 5.82e-17]
U:
 [[-0.27 -0.73  0.63 -0.06  0.06]
 [-0.35 -0.42 -0.69 -0.45  0.12]
 [-0.43 -0.11 -0.24  0.85  0.12]
 [-0.51  0.19  0.06 -0.1  -0.83]
 [-0.59  0.5   0.25 -0.24  0.53]]
Column space:
 [[-0.27 -0.35]
 [ 0.73  0.42]
 [ 0.32 -0.65]
 [ 0.54 -0.39]
 [-0.06 -0.35]]
Left null space:
 [[ 0.63 -0.06  0.06]
```

(continues on next page)

(continued from previous page)

```
 [-0.69 -0.45  0.12]
 [-0.24  0.85  0.12]
 [ 0.06 -0.1  -0.83]
 [ 0.25 -0.24  0.53]]
V.T:
 [[-0.27  0.73  0.32  0.54 -0.06]
 [-0.35  0.42 -0.65 -0.39 -0.35]
 [-0.43  0.11  0.02 -0.29  0.85]
 [-0.51 -0.19  0.61 -0.41 -0.4 ]
 [-0.59 -0.5  -0.31  0.55 -0.04]]
Row space:
 [[-0.27 -0.35 -0.43 -0.51 -0.59]
 [-0.73 -0.42 -0.11  0.19  0.5 ]]
Right null space:
 [[-0.43  0.11  0.02 -0.29  0.85]
 [-0.51 -0.19  0.61 -0.41 -0.4 ]
 [-0.59 -0.5  -0.31  0.55 -0.04]]
```

5.5 Eckart-Young Theorem

Suppose that we want to construct the best rank r approximation of an $m \times n$ matrix X.

By best, we mean a matrix X_r of rank $r < p$ that, among all rank r matrices, minimizes

$$||X - X_r||$$

where $||\cdot||$ denotes a norm of a matrix X and where X_r belongs to the space of all rank r matrices of dimension $m \times n$.

Three popular **matrix norms** of an $m \times n$ matrix X can be expressed in terms of the singular values of X

- the **spectral** or l^2 norm $||X||_2 = \max_{||y|| \neq 0} \frac{||Xy||}{||y||} = \sigma_1$
- the **Frobenius** norm $||X||_F = \sqrt{\sigma_1^2 + \cdots + \sigma_p^2}$
- the **nuclear** norm $||X||_N = \sigma_1 + \cdots + \sigma_p$

The Eckart-Young theorem states that for each of these three norms, same rank r matrix is best and that it equals

$$\hat{X}_r = \sigma_1 U_1 V_1^\top + \sigma_2 U_2 V_2^\top + \cdots + \sigma_r U_r V_r^\top \tag{5.8}$$

This is a very powerful theorem that says that we can take our $m \times n$ matrix X that in not full rank, and we can best approximate it by a full rank $p \times p$ matrix through the SVD.

Moreover, if some of these p singular values carry more information than others, and if we want to have the most amount of information with the least amount of data, we can take r leading singular values ordered by magnitude.

We'll say more about this later when we present Principal Component Analysis.

You can read about the Eckart-Young theorem and some of its uses here.

We'll make use of this theorem when we discuss principal components analysis (PCA) and also dynamic mode decomposition (DMD).

5.6 Full and Reduced SVD's

Up to now we have described properties of a **full** SVD in which shapes of U, Σ, and V are (m, m), (m, n), (n, n), respectively.

There is an alternative bookkeeping convention called an **economy** or **reduced** SVD in which the shapes of U, Σ and V are different from what they are in a full SVD.

Thus, note that because we assume that X has rank p, there are only p nonzero singular values, where $p = \text{rank}(X) \leq \min(m, n)$.

A **reduced** SVD uses this fact to express U, Σ, and V as matrices with shapes (m, p), (p, p), (n, p).

You can read about reduced and full SVD here https://numpy.org/doc/stable/reference/generated/numpy.linalg.svd.html

For a full SVD,

$$\begin{aligned} UU^\top = I \qquad & U^\top U = I \\ VV^\top = I \qquad & V^\top V = I \end{aligned}$$

But not all these properties hold for a **reduced** SVD.

Which properties hold depend on whether we are in a **tall-skinny** case or a **short-fat** case.

- In a **tall-skinny** case in which $m >> n$, for a **reduced** SVD

$$\begin{aligned} UU^\top \neq I \qquad & U^\top U = I \\ VV^\top = I \qquad & V^\top V = I \end{aligned}$$

- In a **short-fat** case in which $m << n$, for a **reduced** SVD

$$\begin{aligned} UU^\top = I \qquad & U^\top U = I \\ VV^\top = I \qquad & V^\top V \neq I \end{aligned}$$

When we study Dynamic Mode Decomposition below, we shall want to remember these properties when we use a reduced SVD to compute some DMD representations.

Let's do an exercise to compare **full** and **reduced** SVD's.

To review,

- in a **full** SVD
  - U is $m \times m$
  - Σ is $m \times n$
  - V is $n \times n$
- in a **reduced** SVD
  - U is $m \times p$
  - Σ is $p \times p$
  - V is $n \times p$

First, let's study a case in which $m = 5 > n = 2$.

(This is a small example of the **tall-skinny** case that will concern us when we study **Dynamic Mode Decompositions** below.)

```
import numpy as np
X = np.random.rand(5,2)
U, S, V = np.linalg.svd(X,full_matrices=True)  # full SVD
Uhat, Shat, Vhat = np.linalg.svd(X,full_matrices=False) # economy SVD
print('U, S, V =')
U, S, V
```

```
U, S, V =
```

```
(array([[-0.29,  0.04, -0.42, -0.48, -0.71],
        [-0.44, -0.37, -0.31, -0.43,  0.63],
        [-0.49, -0.14,  0.82, -0.21, -0.16],
        [-0.58, -0.22, -0.22,  0.74, -0.14],
        [-0.39,  0.89, -0.04,  0.  ,  0.24]]),
 array([1.6 , 0.27]),
 array([[-0.93, -0.37],
        [ 0.37, -0.93]]))
```

```
print('Uhat, Shat, Vhat = ')
Uhat, Shat, Vhat
```

```
Uhat, Shat, Vhat =
```

```
(array([[-0.29,  0.04],
        [-0.44, -0.37],
        [-0.49, -0.14],
        [-0.58, -0.22],
        [-0.39,  0.89]]),
 array([1.6 , 0.27]),
 array([[-0.93, -0.37],
        [ 0.37, -0.93]]))
```

```
rr = np.linalg.matrix_rank(X)
print(f'rank of X = {rr}')
```

```
rank of X = 2
```

Properties:

- Where U is constructed via a full SVD, $U^\top U = I_{m\times m}$ and $UU^\top = I_{m\times m}$
- Where $\hat{U}$ is constructed via a reduced SVD, although $\hat{U}^\top\hat{U} = I_{p\times p}$, it happens that $\hat{U}\hat{U}^\top \neq I_{m\times m}$

We illustrate these properties for our example with the following code cells.

```
UTU = U.T@U
UUT = U@U.T
print('UUT, UTU = ')
UUT, UTU
```

```
UUT, UTU =
```

```
(array([[ 1.00e+00, -1.89e-16, -9.37e-17, -1.11e-16, -1.01e-16],
        [-1.89e-16,  1.00e+00, -3.09e-17, -3.82e-17, -1.24e-16],
        [-9.37e-17, -3.09e-17,  1.00e+00, -1.82e-17, -9.05e-17],
```

(continues on next page)

(continued from previous page)

```
        [-1.11e-16, -3.82e-17, -1.82e-17,  1.00e+00, -1.59e-17],
        [-1.01e-16, -1.24e-16, -9.05e-17, -1.59e-17,  1.00e+00]]),
 array([[ 1.00e+00,  9.58e-17, -2.58e-17, -1.67e-16, -1.73e-16],
        [ 9.58e-17,  1.00e+00,  4.31e-17,  7.67e-17, -1.50e-16],
        [-2.58e-17,  4.31e-17,  1.00e+00, -1.09e-17, -1.05e-16],
        [-1.67e-16,  7.67e-17, -1.09e-17,  1.00e+00, -9.48e-17],
        [-1.73e-16, -1.50e-16, -1.05e-16, -9.48e-17,  1.00e+00]]))
```

```
UhatUhatT = Uhat@Uhat.T
UhatTUhat = Uhat.T@Uhat
print('UhatUhatT, UhatTUhat= ')
UhatUhatT, UhatTUhat
```

```
UhatUhatT, UhatTUhat=
```

```
(array([[ 0.09,  0.11,  0.14,  0.16,  0.15],
        [ 0.11,  0.33,  0.26,  0.34, -0.16],
        [ 0.14,  0.26,  0.26,  0.31,  0.07],
        [ 0.16,  0.34,  0.31,  0.38,  0.02],
        [ 0.15, -0.16,  0.07,  0.02,  0.94]]),
 array([[1.00e+00, 9.58e-17],
        [9.58e-17, 1.00e+00]]))
```

Remarks:

The cells above illustrate the application of the `full_matrices=True` and `full_matrices=False` options. Using `full_matrices=False` returns a reduced singular value decomposition.

The **full** and **reduced** SVD's both accurately decompose an $m \times n$ matrix X

When we study Dynamic Mode Decompositions below, it will be important for us to remember the preceding properties of full and reduced SVD's in such tall-skinny cases.

Now let's turn to a short-fat case.

To illustrate this case, we'll set $m = 2 < 5 = n$ and compute both full and reduced SVD's.

```
import numpy as np
X = np.random.rand(2,5)
U, S, V = np.linalg.svd(X,full_matrices=True)  # full SVD
Uhat, Shat, Vhat = np.linalg.svd(X,full_matrices=False) # economy SVD
print('U, S, V = ')
U, S, V
```

```
U, S, V =
```

```
(array([[ 0.8, -0.6],
        [ 0.6,  0.8]]),
 array([1.72, 0.85]),
 array([[ 0.54,  0.18,  0.64,  0.35,  0.39],
        [-0.33,  0.49,  0.12,  0.6 , -0.52],
        [-0.52, -0.38,  0.72, -0.21, -0.1 ],
        [-0.06, -0.7 , -0.19,  0.68,  0.12],
        [-0.57,  0.31, -0.12,  0.1 ,  0.75]]))
```

```
print('Uhat, Shat, Vhat = ')
Uhat, Shat, Vhat
```

```
Uhat, Shat, Vhat =
```

```
(array([[ 0.8, -0.6],
        [ 0.6,  0.8]]),
 array([1.72, 0.85]),
 array([[ 0.54,  0.18,  0.64,  0.35,  0.39],
        [-0.33,  0.49,  0.12,  0.6 , -0.52]]))
```

Let's verify that our reduced SVD accurately represents X

```
SShat=np.diag(Shat)
np.allclose(X, Uhat@SShat@Vhat)
```

```
True
```

5.7 Polar Decomposition

A **reduced** singular value decomposition (SVD) of X is related to a **polar decomposition** of X

$$X = SQ$$

where

$$S = U\Sigma U^\top$$
$$Q = UV^\top$$

Here

- S is an $m \times m$ **symmetric** matrix
- Q is an $m \times n$ **orthogonal** matrix

and in our reduced SVD

- U is an $m \times p$ orthonormal matrix
- Σ is a $p \times p$ diagonal matrix
- V is an $n \times p$ orthonormal

5.8 Application: Principal Components Analysis (PCA)

Let's begin with a case in which $n >> m$, so that we have many more individuals n than attributes m.

The matrix X is **short and fat** in an $n >> m$ case as opposed to a **tall and skinny** case with $m >> n$ to be discussed later.

We regard X as an $m \times n$ matrix of **data**:

$$X = [X_1 \mid X_2 \mid \cdots \mid X_n]$$

where for $j = 1, \ldots, n$ the column vector $X_j = \begin{bmatrix} x_{1j} \\ x_{2j} \\ \vdots \\ x_{mj} \end{bmatrix}$ is a vector of observations on variables $\begin{bmatrix} X_1 \\ X_2 \\ \vdots \\ X_m \end{bmatrix}$.

In a **time series** setting, we would think of columns j as indexing different **times** at which random variables are observed, while rows index different random variables.

In a **cross-section** setting, we would think of columns j as indexing different **individuals** for which random variables are observed, while rows index different **attributes**.

As we have seen before, the SVD is a way to decompose a matrix into useful components, just like polar decomposition, eigendecomposition, and many others.

PCA, on the other hand, is a method that builds on the SVD to analyze data. The goal is to apply certain steps, to help better visualize patterns in data, using statistical tools to capture the most important patterns in data.

Step 1: Standardize the data:

Because our data matrix may hold variables of different units and scales, we first need to standardize the data.

First by computing the average of each row of X.

$$\bar{X_i} = \frac{1}{n} \sum_{j=1}^{n} x_{ij}$$

We then create an average matrix out of these means:

$$\bar{X} = \begin{bmatrix} \bar{X_1} \\ \bar{X_2} \\ \cdots \\ \bar{X_m} \end{bmatrix} \begin{bmatrix} 1 \mid 1 \mid \cdots \mid 1 \end{bmatrix}$$

And subtract out of the original matrix to create a mean centered matrix:

$$B = X - \bar{X}$$

Step 2: Compute the covariance matrix:

Then because we want to extract the relationships between variables rather than just their magnitude, in other words, we want to know how they can explain each other, we compute the covariance matrix of B.

$$C = \frac{1}{n} B B^\top$$

Step 3: Decompose the covariance matrix and arrange the singular values:

Since the matrix C is positive definite, we can eigendecompose it, find its eigenvalues, and rearrange the eigenvalue and eigenvector matrices in a decreasing order.

The eigendecomposition of C can be found by decomposing B instead. Since B is not a square matrix, we obtain an SVD of B:

$$\begin{aligned} BB^\top &= U\Sigma V^\top (U\Sigma V^\top)^\top \\ &= U\Sigma V^\top V \Sigma^\top U^\top \\ &= U\Sigma\Sigma^\top U^\top \end{aligned}$$

$$C = \frac{1}{n} U\Sigma\Sigma^\top U^\top$$

We can then rearrange the columns in the matrices U and Σ so that the singular values are in decreasing order.

Step 4: Select singular values, (optional) truncate the rest:

We can now decide how many singular values to pick, based on how much variance you want to retain. (e.g., retaining 95% of the total variance).

We can obtain the percentage by calculating the variance contained in the leading r factors divided by the variance in total:

$$\frac{\sum_{i=1}^{r} \sigma_i^2}{\sum_{i=1}^{p} \sigma_i^2}$$

Step 5: Create the Score Matrix:

$$\begin{aligned} T &= BV \\ &= U\Sigma V^\top V \\ &= U\Sigma \end{aligned}$$

5.9 Relationship of PCA to SVD

To relate an SVD to a PCA of data set X, first construct the SVD of the data matrix X:

Let's assume that sample means of all variables are zero, so we don't need to standardize our matrix.

$$X = U\Sigma V^\top = \sigma_1 U_1 V_1^\top + \sigma_2 U_2 V_2^\top + \cdots + \sigma_p U_p V_p^\top \tag{5.9}$$

where

$$U = [U_1 | U_2 | \ldots | U_m]$$

$$V^\top = \begin{bmatrix} V_1^\top \\ V_2^\top \\ \ldots \\ V_n^\top \end{bmatrix}$$

In equation (5.9), each of the $m \times n$ matrices $U_j V_j^\top$ is evidently of rank 1.

Thus, we have

$$X = \sigma_1 \begin{bmatrix} U_{11} V_1^\top \\ U_{21} V_1^\top \\ \ldots \\ U_{m1} V_1^\top \end{bmatrix} + \sigma_2 \begin{bmatrix} U_{12} V_2^\top \\ U_{22} V_2^\top \\ \ldots \\ U_{m2} V_2^\top \end{bmatrix} + \ldots + \sigma_p \begin{bmatrix} U_{1p} V_p^\top \\ U_{2p} V_p^\top \\ \ldots \\ U_{mp} V_p^\top \end{bmatrix} \tag{5.10}$$

Here is how we would interpret the objects in the matrix equation (5.10) in a time series context:

- for each $k = 1, \ldots, n$, the object $\{V_{kj}\}_{j=1}^n$ is a time series for the kth **principal component**
- $U_j = \begin{bmatrix} U_{1k} \\ U_{2k} \\ \ldots \\ U_{mk} \end{bmatrix}$ $k = 1, \ldots, m$ is a vector of **loadings** of variables X_i on the kth principal component, $i = 1, \ldots, m$
- σ_k for each $k = 1, \ldots, p$ is the strength of kth **principal component**, where strength means contribution to the overall covariance of X.

5.10 PCA with Eigenvalues and Eigenvectors

We now use an eigen decomposition of a sample covariance matrix to do PCA.

Let $X_{m\times n}$ be our $m \times n$ data matrix.

Let's assume that sample means of all variables are zero.

We can assure this by **pre-processing** the data by subtracting sample means.

Define a sample covariance matrix Ω as

$$\Omega = XX^\top$$

Then use an eigen decomposition to represent Ω as follows:

$$\Omega = P\Lambda P^\top$$

Here

- P is $m \times m$ matrix of eigenvectors of Ω
- Λ is a diagonal matrix of eigenvalues of Ω

We can then represent X as

$$X = P\epsilon$$

where

$$\epsilon = P^{-1}X$$

and

$$\epsilon\epsilon^\top = \Lambda.$$

We can verify that

$$XX^\top = P\Lambda P^\top. \tag{5.11}$$

It follows that we can represent the data matrix X as

$$X = [X_1|X_2|...|X_m] = [P_1|P_2|...|P_m]\begin{bmatrix}\epsilon_1\\ \epsilon_2\\ ...\\ \epsilon_m\end{bmatrix} = P_1\epsilon_1 + P_2\epsilon_2 + ... + P_m\epsilon_m$$

To reconcile the preceding representation with the PCA that we had obtained earlier through the SVD, we first note that $\epsilon_j^2 = \lambda_j \equiv \sigma_j^2$.

Now define $\tilde{\epsilon_j} = \frac{\epsilon_j}{\sqrt{\lambda_j}}$, which implies that $\tilde{\epsilon}_j\tilde{\epsilon}_j^\top = 1$.

Therefore

$$\begin{aligned} X &= \sqrt{\lambda_1}P_1\tilde{\epsilon_1} + \sqrt{\lambda_2}P_2\tilde{\epsilon_2} + ... + \sqrt{\lambda_m}P_m\tilde{\epsilon_m} \\ &= \sigma_1 P_1\tilde{\epsilon_2} + \sigma_2 P_2\tilde{\epsilon_2} + ... + \sigma_m P_m\tilde{\epsilon_m}, \end{aligned}$$

which agrees with

$$X = \sigma_1 U_1 {V_1}^T + \sigma_2 U_2 {V_2}^T + ... + \sigma_r U_r {V_r}^T$$

provided that we set

- $U_j = P_j$ (a vector of loadings of variables on principal component j)
- ${V_k}^T = \tilde{\epsilon_k}$ (the kth principal component)

Because there are alternative algorithms for computing P and U for given a data matrix X, depending on algorithms used, we might have sign differences or different orders of eigenvectors.

We can resolve such ambiguities about U and P by

1. sorting eigenvalues and singular values in descending order
2. imposing positive diagonals on P and U and adjusting signs in $V^\top$ accordingly

5.11 Connections

To pull things together, it is useful to assemble and compare some formulas presented above.

First, consider an SVD of an $m \times n$ matrix:

$$X = U\Sigma V^\top$$

Compute:

$$\begin{aligned} XX^\top &= U\Sigma V^\top V\Sigma^\top U^\top \\ &\equiv U\Sigma\Sigma^\top U^\top \\ &\equiv U\Lambda U^\top \end{aligned} \tag{5.12}$$

Compare representation (5.12) with equation (5.11) above.

Evidently, U in the SVD is the matrix P of eigenvectors of $XX^\top$ and $\Sigma\Sigma^\top$ is the matrix Λ of eigenvalues.

Second, let's compute

$$\begin{aligned} X^\top X &= V\Sigma^\top U^\top U\Sigma V^\top \\ &= V\Sigma^\top\Sigma V^\top \end{aligned}$$

Thus, the matrix V in the SVD is the matrix of eigenvectors of $X^\top X$

Summarizing and fitting things together, we have the eigen decomposition of the sample covariance matrix

$$XX^\top = P\Lambda P^\top$$

where P is an orthogonal matrix.

Further, from the SVD of X, we know that

$$XX^\top = U\Sigma\Sigma^\top U^\top$$

where U is an orthogonal matrix.

Thus, $P = U$ and we have the representation of X

$$X = P\epsilon = U\Sigma V^\top$$

It follows that

$$U^\top X = \Sigma V^\top = \epsilon$$

Note that the preceding implies that

$$\epsilon\epsilon^\top = \Sigma V^\top V \Sigma^\top = \Sigma\Sigma^\top = \Lambda,$$

so that everything fits together.

Below we define a class `DecomAnalysis` that wraps PCA and SVD for a given a data matrix `X`.

```
class DecomAnalysis:
    """
    A class for conducting PCA and SVD.
    X: data matrix
    r_component: chosen rank for best approximation
    """

    def __init__(self, X, r_component=None):

        self.X = X

        self.Ω = (X @ X.T)

        self.m, self.n = X.shape
        self.r = LA.matrix_rank(X)

        if r_component:
            self.r_component = r_component
        else:
            self.r_component = self.m

    def pca(self):

        λ, P = LA.eigh(self.Ω)     # columns of P are eigenvectors

        ind = sorted(range(λ.size), key=lambda x: λ[x], reverse=True)

        # sort by eigenvalues
        self.λ = λ[ind]
        P = P[:, ind]
        self.P = P @ diag_sign(P)

        self.Λ = np.diag(self.λ)

        self.explained_ratio_pca = np.cumsum(self.λ) / self.λ.sum()

        # compute the N by T matrix of principal components
        self.ϵ = self.P.T @ self.X

        P = self.P[:, :self.r_component]
        ϵ = self.ϵ[:self.r_component, :]

        # transform data
        self.X_pca = P @ ϵ

    def svd(self):

        U, σ, VT = LA.svd(self.X)

        ind = sorted(range(σ.size), key=lambda x: σ[x], reverse=True)
```

(continues on next page)

(continued from previous page)

```
        # sort by eigenvalues
        d = min(self.m, self.n)

        self.σ = σ[ind]
        U = U[:, ind]
        D = diag_sign(U)
        self.U = U @ D
        VT[:d, :] = D @ VT[ind, :]
        self.VT = VT

        self.Σ = np.zeros((self.m, self.n))
        self.Σ[:d, :d] = np.diag(self.σ)

        σ_sq = self.σ ** 2
        self.explained_ratio_svd = np.cumsum(σ_sq) / σ_sq.sum()

        # slicing matrices by the number of components to use
        U = self.U[:, :self.r_component]
        Σ = self.Σ[:self.r_component, :self.r_component]
        VT = self.VT[:self.r_component, :]

        # transform data
        self.X_svd = U @ Σ @ VT

    def fit(self, r_component):

        # pca
        P = self.P[:, :r_component]
        ϵ = self.ϵ[:r_component, :]

        # transform data
        self.X_pca = P @ ϵ

        # svd
        U = self.U[:, :r_component]
        Σ = self.Σ[:r_component, :r_component]
        VT = self.VT[:r_component, :]

        # transform data
        self.X_svd = U @ Σ @ VT

def diag_sign(A):
    "Compute the signs of the diagonal of matrix A"

    D = np.diag(np.sign(np.diag(A)))

    return D
```

We also define a function that prints out information so that we can compare decompositions obtained by different algorithms.

```
def compare_pca_svd(da):
    """
    Compare the outcomes of PCA and SVD.
    """

    da.pca()
```

(continues on next page)

(continued from previous page)

```
da.svd()

print('Eigenvalues and Singular values\n')
print(f'λ = {da.λ}\n')
print(f'σ^2 = {da.σ**2}\n')
print('\n')

# loading matrices
fig, axs = plt.subplots(1, 2, figsize=(14, 5))
plt.suptitle('loadings')
axs[0].plot(da.P.T)
axs[0].set_title('P')
axs[0].set_xlabel('m')
axs[1].plot(da.U.T)
axs[1].set_title('U')
axs[1].set_xlabel('m')
plt.show()

# principal components
fig, axs = plt.subplots(1, 2, figsize=(14, 5))
plt.suptitle('principal components')
axs[0].plot(da.ε.T)
axs[0].set_title('ε')
axs[0].set_xlabel('n')
axs[1].plot(da.VT[:da.r, :].T * np.sqrt(da.λ))
axs[1].set_title(r'$V^\top *\sqrt{\lambda}$')
axs[1].set_xlabel('n')
plt.show()
```

5.12 Exercises

Exercise 5.12.1

In Ordinary Least Squares (OLS), we learn to compute $\hat{\beta} = (X^\top X)^{-1} X^\top y$, but there are cases such as when we have colinearity or an underdetermined system: **short fat** matrix.

In these cases, the $(X^\top X)$ matrix is not not invertible (its determinant is zero) or ill-conditioned (its determinant is very close to zero).

What we can do instead is to create what is called a pseudoinverse, a full rank approximation of the inverted matrix so we can compute $\hat{\beta}$ with it.

Thinking in terms of the Eckart-Young theorem, build the pseudoinverse matrix X^+ and use it to compute $\hat{\beta}$.

Solution

We can use SVD to compute the pseudoinverse:

$$X = U \Sigma V^\top$$

inverting X, we have:

$$X^+ = V\Sigma^+U^\top$$

where:

$$\Sigma^+ = \begin{bmatrix} \frac{1}{\sigma_1} & 0 & \cdots & 0 & 0 \\ 0 & \frac{1}{\sigma_2} & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & \frac{1}{\sigma_p} & 0 \\ 0 & 0 & \cdots & 0 & 0 \end{bmatrix}$$

and finally:

$$\hat{\beta} = X^+y = V\Sigma^+U^\top y$$

For an example PCA applied to analyzing the structure of intelligence tests see this lecture *Multivariable Normal Distribution*.

Look at parts of that lecture that describe and illustrate the classic factor analysis model.

As mentioned earlier, in a sequel to this lecture about *Dynamic Mode Decompositions*, we'll describe how SVD's provide ways rapidly to compute reduced-order approximations to first-order Vector Autoregressions (VARs).

CHAPTER

SIX

VARS AND DMDS

This lecture applies computational methods that we learned about in this lecture *Singular Value Decomposition* to

- first-order vector autoregressions (VARs)
- dynamic mode decompositions (DMDs)
- connections between DMDs and first-order VARs

6.1 First-Order Vector Autoregressions

We want to fit a **first-order vector autoregression**

$$X_{t+1} = AX_t + C\epsilon_{t+1}, \quad \epsilon_{t+1} \perp X_t \tag{6.1}$$

where ϵ_{t+1} is the time $t+1$ component of a sequence of i.i.d. $m \times 1$ random vectors with mean vector zero and identity covariance matrix and where the $m \times 1$ vector X_t is

$$X_t = \begin{bmatrix} X_{1,t} & X_{2,t} & \cdots & X_{m,t} \end{bmatrix}^\top \tag{6.2}$$

and where $\cdot^\top$ again denotes complex transposition and $X_{i,t}$ is variable i at time t.

We want to fit equation (6.1).

Our data are organized in an $m \times (n+1)$ matrix $\tilde{X}$

$$\tilde{X} = \begin{bmatrix} X_1 \mid X_2 \mid \cdots \mid X_n \mid X_{n+1} \end{bmatrix}$$

where for $t = 1, \ldots, n+1$, the $m \times 1$ vector X_t is given by (6.2).

Thus, we want to estimate a system (6.1) that consists of m least squares regressions of **everything** on one lagged value of **everything**.

The i'th equation of (6.1) is a regression of $X_{i,t+1}$ on the vector X_t.

We proceed as follows.

From $\tilde{X}$, we form two $m \times n$ matrices

$$X = \begin{bmatrix} X_1 \mid X_2 \mid \cdots \mid X_n \end{bmatrix}$$

and

$$X' = \begin{bmatrix} X_2 \mid X_3 \mid \cdots \mid X_{n+1} \end{bmatrix}$$

Here $'$ is part of the name of the matrix X' and does not indicate matrix transposition.

We use $\cdot^\top$ to denote matrix transposition or its extension to complex matrices.

In forming X and X', we have in each case dropped a column from $\tilde{X}$, the last column in the case of X, and the first column in the case of X'.

Evidently, X and X' are both $m \times n$ matrices.

We denote the rank of X as $p \leq \min(m, n)$.

Two cases that interest us are

- $n >> m$, so that we have many more time series observations n than variables m
- $m >> n$, so that we have many more variables m than time series observations n

At a general level that includes both of these special cases, a common formula describes the least squares estimator $\hat{A}$ of A.

But important details differ.

The common formula is

$$\hat{A} = X' X^+ \tag{6.3}$$

where X^+ is the pseudo-inverse of X.

To read about the **Moore-Penrose pseudo-inverse** please see Moore-Penrose pseudo-inverse

Applicable formulas for the pseudo-inverse differ for our two cases.

Short-Fat Case:

When $n >> m$, so that we have many more time series observations n than variables m and when X has linearly independent **rows**, $XX^\top$ has an inverse and the pseudo-inverse X^+ is

$$X^+ = X^\top (XX^\top)^{-1}$$

Here X^+ is a **right-inverse** that verifies $XX^+ = I_{m \times m}$.

In this case, our formula (6.3) for the least-squares estimator of the population matrix of regression coefficients A becomes

$$\hat{A} = X' X^\top (XX^\top)^{-1} \tag{6.4}$$

This formula for least-squares regression coefficients is widely used in econometrics.

It is used to estimate vector autorgressions.

The right side of formula (6.4) is proportional to the empirical cross second moment matrix of X_{t+1} and X_t times the inverse of the second moment matrix of X_t.

Tall-Skinny Case:

When $m >> n$, so that we have many more attributes m than time series observations n and when X has linearly independent **columns**, $X^\top X$ has an inverse and the pseudo-inverse X^+ is

$$X^+ = (X^\top X)^{-1} X^\top$$

Here X^+ is a **left-inverse** that verifies $X^+ X = I_{n \times n}$.

In this case, our formula (6.3) for a least-squares estimator of A becomes

$$\hat{A} = X' (X^\top X)^{-1} X^\top \tag{6.5}$$

Please compare formulas (6.4) and (6.5) for $\hat{A}$.

Here we are especially interested in formula (6.5).

The ith row of $\hat{A}$ is an $m \times 1$ vector of regression coefficients of $X_{i,t+1}$ on $X_{j,t}, j = 1, \ldots, m$.

If we use formula (6.5) to calculate $\hat{A}X$ we find that

$$\hat{A}X = X'$$

so that the regression equation **fits perfectly**.

This is a typical outcome in an **underdetermined least-squares** model.

To reiterate, in the **tall-skinny** case (described in *Singular Value Decomposition*) in which we have a number n of observations that is small relative to the number m of attributes that appear in the vector X_t, we want to fit equation (6.1).

We confront the facts that the least squares estimator is underdetermined and that the regression equation fits perfectly.

To proceed, we'll want efficiently to calculate the pseudo-inverse X^+.

The pseudo-inverse X^+ will be a component of our estimator of A.

As our estimator $\hat{A}$ of A we want to form an $m \times m$ matrix that solves the least-squares best-fit problem

$$\hat{A} = \text{argmin}_{\check{A}} ||X' - \check{A}X||_F \tag{6.6}$$

where $||\cdot||_F$ denotes the Frobenius (or Euclidean) norm of a matrix.

The Frobenius norm is defined as

$$||A||_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{m} |A_{ij}|^2}$$

The minimizer of the right side of equation (6.6) is

$$\hat{A} = X'X^+ \tag{6.7}$$

where the (possibly huge) $n \times m$ matrix $X^+ = (X^\top X)^{-1}X^\top$ is again a pseudo-inverse of X.

For some situations that we are interested in, $X^\top X$ can be close to singular, a situation that makes some numerical algorithms be inaccurate.

To acknowledge that possibility, we'll use efficient algorithms to constructing a **reduced-rank approximation** of $\hat{A}$ in formula (6.5).

Such an approximation to our vector autoregression will no longer fit perfectly.

The ith row of $\hat{A}$ is an $m \times 1$ vector of regression coefficients of $X_{i,t+1}$ on $X_{j,t}, j = 1, \ldots, m$.

An efficient way to compute the pseudo-inverse X^+ is to start with a singular value decomposition

$$X = U\Sigma V^\top \tag{6.8}$$

where we remind ourselves that for a **reduced** SVD, X is an $m \times n$ matrix of data, U is an $m \times p$ matrix, Σ is a $p \times p$ matrix, and V is an $n \times p$ matrix.

We can efficiently construct the pertinent pseudo-inverse X^+ by recognizing the following string of equalities.

$$\begin{aligned} X^+ &= (X^\top X)^{-1}X^\top \\ &= (V\Sigma U^\top U\Sigma V^\top)^{-1}V\Sigma U^\top \\ &= (V\Sigma\Sigma V^\top)^{-1}V\Sigma U^\top \\ &= V\Sigma^{-1}\Sigma^{-1}V^\top V\Sigma U^\top \\ &= V\Sigma^{-1}U^\top \end{aligned} \tag{6.9}$$

(Since we are in the $m >> n$ case in which $V^\top V = I_{p \times p}$ in a reduced SVD, we can use the preceding string of equalities for a reduced SVD as well as for a full SVD.)

Thus, we shall construct a pseudo-inverse X^+ of X by using a singular value decomposition of X in equation (6.8) to compute

$$X^+ = V \Sigma^{-1} U^\top \tag{6.10}$$

where the matrix Σ^{-1} is constructed by replacing each non-zero element of Σ with σ_j^{-1}.

We can use formula (6.10) together with formula (6.7) to compute the matrix $\hat{A}$ of regression coefficients.

Thus, our estimator $\hat{A} = X' X^+$ of the $m \times m$ matrix of coefficients A is

$$\hat{A} = X' V \Sigma^{-1} U^\top \tag{6.11}$$

6.2 Dynamic Mode Decomposition (DMD)

We turn to the $m >> n$ **tall and skinny** case associated with **Dynamic Mode Decomposition**.

Here an $m \times n+1$ data matrix $\tilde{X}$ contains many more attributes (or variables) m than time periods $n+1$.

Dynamic mode decomposition was introduced by [Schmid, 2010],

You can read about Dynamic Mode Decomposition [Kutz *et al.*, 2016] and [Brunton and Kutz, 2019] (section 7.2).

Dynamic Mode Decomposition (DMD) computes a rank $r < p$ approximation to the least squares regression coefficients $\hat{A}$ described by formula (6.11).

We'll build up gradually to a formulation that is useful in applications.

We'll do this by describing three alternative representations of our first-order linear dynamic system, i.e., our vector autoregression.

Guide to three representations: In practice, we'll mainly be interested in Representation 3.

We use the first two representations to present some useful intermediate steps that help us to appreciate what is under the hood of Representation 3.

In applications, we'll use only a small subset of **DMD modes** to approximate dynamics.

We use such a small subset of DMD modes to construct a reduced-rank approximation to A.

To do that, we'll want to use the **reduced** SVD's affiliated with representation 3, not the **full** SVD's affiliated with representations 1 and 2.

Guide to impatient reader: In our applications, we'll be using Representation 3.

You might want to skip the stage-setting representations 1 and 2 on first reading.

6.3 Representation 1

In this representation, we shall use a **full** SVD of X.

We use the m **columns** of U, and thus the m **rows** of $U^\top$, to define a $m \times 1$ vector $\tilde{b}_t$ as

$$\tilde{b}_t = U^\top X_t. \tag{6.12}$$

The original data X_t can be represented as

$$X_t = U\tilde{b}_t \tag{6.13}$$

(Here we use b to remind ourselves that we are creating a **basis** vector.)

Since we are now using a **full** SVD, $UU^\top = I_{m\times m}$.

So it follows from equation (6.12) that we can reconstruct X_t from $\tilde{b}_t$.

In particular,

- Equation (6.12) serves as an **encoder** that **rotates** the $m \times 1$ vector X_t to become an $m \times 1$ vector $\tilde{b}_t$
- Equation (6.13) serves as a **decoder** that **reconstructs** the $m \times 1$ vector X_t by rotating the $m \times 1$ vector $\tilde{b}_t$

Define a transition matrix for an $m \times 1$ basis vector $\tilde{b}_t$ by

$$\tilde{A} = U^\top \hat{A} U \tag{6.14}$$

We can recover $\hat{A}$ from

$$\hat{A} = U\tilde{A}U^\top$$

Dynamics of the $m \times 1$ basis vector $\tilde{b}_t$ are governed by

$$\tilde{b}_{t+1} = \tilde{A}\tilde{b}_t$$

To construct forecasts $\overline{X}_t$ of future values of X_t conditional on X_1, we can apply decoders (i.e., rotators) to both sides of this equation and deduce

$$\overline{X}_{t+1} = U\tilde{A}^t U^\top X_1$$

where we use $\overline{X}_{t+1}, t \geq 1$ to denote a forecast.

6.4 Representation 2

This representation is related to one originally proposed by [Schmid, 2010].

It can be regarded as an intermediate step on the way to obtaining a related representation 3 to be presented later

As with Representation 1, we continue to

- use a **full** SVD and **not** a reduced SVD

As we observed and illustrated in a lecture about the *Singular Value Decomposition*

- (a) for a full SVD $UU^\top = I_{m\times m}$ and $U^\top U = I_{p\times p}$ are both identity matrices
- (b) for a reduced SVD of X, $U^\top U$ is not an identity matrix.

As we shall see later, a full SVD is too confining for what we ultimately want to do, namely, cope with situations in which $U^\top U$ is **not** an identity matrix because we use a reduced SVD of X.

But for now, let's proceed under the assumption that we are using a full SVD so that requirements (a) and (b) are both satisfied.

Form an eigendecomposition of the $m \times m$ matrix $\tilde{A} = U^\top \hat{A} U$ defined in equation (6.14):

$$\tilde{A} = W\Lambda W^{-1} \tag{6.15}$$

where Λ is a diagonal matrix of eigenvalues and W is an $m \times m$ matrix whose columns are eigenvectors corresponding to rows (eigenvalues) in Λ.

When $UU^\top = I_{m\times m}$, as is true with a full SVD of X, it follows that

$$\hat{A} = U\tilde{A}U^\top = UW\Lambda W^{-1}U^\top \tag{6.16}$$

According to equation (6.16), the diagonal matrix Λ contains eigenvalues of $\hat{A}$ and corresponding eigenvectors of $\hat{A}$ are columns of the matrix UW.

It follows that the systematic (i.e., not random) parts of the X_t dynamics captured by our first-order vector autoregressions are described by

$$X_{t+1} = UW\Lambda W^{-1}U^\top X_t$$

Multiplying both sides of the above equation by $W^{-1}U^\top$ gives

$$W^{-1}U^\top X_{t+1} = \Lambda W^{-1}U^\top X_t$$

or

$$\hat{b}_{t+1} = \Lambda \hat{b}_t$$

where our **encoder** is

$$\hat{b}_t = W^{-1}U^\top X_t$$

and our **decoder** is

$$X_t = UW\hat{b}_t$$

We can use this representation to construct a predictor $\overline{X}_{t+1}$ of X_{t+1} conditional on X_1 via:

$$\overline{X}_{t+1} = UW\Lambda^t W^{-1}U^\top X_1 \tag{6.17}$$

In effect, [Schmid, 2010] defined an $m \times m$ matrix Φ_s as

$$\Phi_s = UW \tag{6.18}$$

and a generalized inverse

$$\Phi_s^+ = W^{-1}U^\top \tag{6.19}$$

[Schmid, 2010] then represented equation (6.17) as

$$\overline{X}_{t+1} = \Phi_s \Lambda^t \Phi_s^+ X_1 \tag{6.20}$$

Components of the basis vector $\hat{b}_t = W^{-1}U^\top X_t \equiv \Phi_s^+ X_t$ are
DMD **projected modes**.

To understand why they are called **projected modes**, notice that

$$\Phi_s^+ = (\Phi_s^\top \Phi_s)^{-1}\Phi_s^\top$$

so that the $m \times p$ matrix

$$\hat{b} = \Phi_s^+ X$$

is a matrix of regression coefficients of the $m \times n$ matrix X on the $m \times p$ matrix Φ_s.

We'll say more about this interpretation in a related context when we discuss representation 3, which was suggested by Tu et al. [Tu *et al.*, 2014].

It is more appropriate to use representation 3 when, as is often the case in practice, we want to use a reduced SVD.

6.5 Representation 3

Departing from the procedures used to construct Representations 1 and 2, each of which deployed a **full** SVD, we now use a **reduced** SVD.

Again, we let $p \leq \min(m, n)$ be the rank of X.

Construct a **reduced** SVD

$$X = \tilde{U}\tilde{\Sigma}\tilde{V}^\top,$$

where now $\tilde{U}$ is $m \times p$, $\tilde{\Sigma}$ is $p \times p$, and $\tilde{V}^\top$ is $p \times n$.

Our minimum-norm least-squares approximator of A now has representation

$$\hat{A} = X'\tilde{V}\tilde{\Sigma}^{-1}\tilde{U}^\top \tag{6.21}$$

Computing Dominant Eigenvectors of $\hat{A}$

We begin by paralleling a step used to construct Representation 1, define a transition matrix for a rotated $p \times 1$ state $\tilde{b}_t$ by

$$\tilde{A} = \tilde{U}^\top \hat{A} \tilde{U} \tag{6.22}$$

Interpretation as projection coefficients

[Brunton and Kutz, 2022] remark that $\tilde{A}$ can be interpreted in terms of a projection of $\hat{A}$ onto the p modes in $\tilde{U}$.

To verify this, first note that, because $\tilde{U}^\top\tilde{U} = I$, it follows that

$$\tilde{A} = \tilde{U}^\top \hat{A} \tilde{U} = \tilde{U}^\top X'\tilde{V}\tilde{\Sigma}^{-1}\tilde{U}^\top\tilde{U} = \tilde{U}^\top X'\tilde{V}\tilde{\Sigma}^{-1}\tilde{U}^\top \tag{6.23}$$

Next, we'll just compute the regression coefficients in a projection of $\hat{A}$ on $\tilde{U}$ using a standard least-squares formula

$$(\tilde{U}^\top\tilde{U})^{-1}\tilde{U}^\top\hat{A} = (\tilde{U}^\top\tilde{U})^{-1}\tilde{U}^\top X'\tilde{V}\tilde{\Sigma}^{-1}\tilde{U}^\top = \tilde{U}^\top X'\tilde{V}\tilde{\Sigma}^{-1}\tilde{U}^\top = \tilde{A}.$$

Thus, we have verified that $\tilde{A}$ is a least-squares projection of $\hat{A}$ onto $\tilde{U}$.

An Inverse Challenge

Because we are using a reduced SVD, $\tilde{U}\tilde{U}^\top \neq I$.

Consequently,

$$\hat{A} \neq \tilde{U}\tilde{A}\tilde{U}^\top,$$

so we can't simply recover $\hat{A}$ from $\tilde{A}$ and $\tilde{U}$.

A Blind Alley

We can start by hoping for the best and proceeding to construct an eigendecomposition of the $p \times p$ matrix $\tilde{A}$:

$$\tilde{A} = \tilde{W}\Lambda\tilde{W}^{-1} \tag{6.24}$$

where Λ is a diagonal matrix of p eigenvalues and the columns of $\tilde{W}$ are corresponding eigenvectors.

Mimicking our procedure in Representation 2, we cross our fingers and compute an $m \times p$ matrix

$$\tilde{\Phi}_s = \tilde{U}\tilde{W} \tag{6.25}$$

that corresponds to (6.18) for a full SVD.

At this point, where $\hat{A}$ is given by formula (6.21) it is interesting to compute $\hat{A}\tilde{\Phi}_s$:

$$\begin{aligned}\hat{A}\tilde{\Phi}_s &= (X'\tilde{V}\tilde{\Sigma}^{-1}\tilde{U}^\top)(\tilde{U}\tilde{W}) \\ &= X'\tilde{V}\tilde{\Sigma}^{-1}\tilde{W} \\ &\neq (\tilde{U}\tilde{W})\Lambda \\ &= \tilde{\Phi}_s\Lambda\end{aligned}$$

That $\hat{A}\tilde{\Phi}_s \neq \tilde{\Phi}_s\Lambda$ means that, unlike the corresponding situation in Representation 2, columns of $\tilde{\Phi}_s = \tilde{U}\tilde{W}$ are **not** eigenvectors of $\hat{A}$ corresponding to eigenvalues on the diagonal of matix Λ.

An Approach That Works

Continuing our quest for eigenvectors of $\hat{A}$ that we **can** compute with a reduced SVD, let's define an $m \times p$ matrix Φ as

$$\Phi \equiv \hat{A}\tilde{\Phi}_s = X'\tilde{V}\tilde{\Sigma}^{-1}\tilde{W} \tag{6.26}$$

It turns out that columns of Φ **are** eigenvectors of $\hat{A}$.

This is a consequence of a result established by Tu et al. [Tu *et al.*, 2014] that we now present.

Proposition 6.5.1

The p columns of Φ are eigenvectors of $\hat{A}$.

Proof. From formula (6.26) we have

$$\begin{aligned}\hat{A}\Phi &= (X'\tilde{V}\tilde{\Sigma}^{-1}\tilde{U}^\top)(X'\tilde{V}\Sigma^{-1}\tilde{W}) \\ &= X'\tilde{V}\tilde{\Sigma}^{-1}\tilde{A}\tilde{W} \\ &= X'\tilde{V}\tilde{\Sigma}^{-1}\tilde{W}\Lambda \\ &= \Phi\Lambda\end{aligned}$$

so that

$$\hat{A}\Phi = \Phi\Lambda. \tag{6.27}$$

Let ϕ_i be the ith column of Φ and λ_i be the corresponding i eigenvalue of $\tilde{A}$ from decomposition (6.24).

Equating the $m \times 1$ vectors that appear on the two sides of equation (6.27) gives

$$\hat{A}\phi_i = \lambda_i\phi_i.$$

This equation confirms that ϕ_i is an eigenvector of $\hat{A}$ that corresponds to eigenvalue λ_i of both $\tilde{A}$ and $\hat{A}$.

Also see [Brunton and Kutz, 2022] (p. 238)

6.5.1 Decoder of $\check{b}$ as a linear projection

From eigendecomposition (6.27) we can represent $\hat{A}$ as

$$\hat{A} = \Phi \Lambda \Phi^{+}. \tag{6.28}$$

From formula (6.28) we can deduce dynamics of the $p \times 1$ vector $\check{b}_t$:

$$\check{b}_{t+1} = \Lambda \check{b}_t$$

where

$$\check{b}_t = \Phi^{+} X_t \tag{6.29}$$

Since the $m \times p$ matrix Φ has p linearly independent columns, the generalized inverse of Φ is

$$\Phi^{+} = (\Phi^{\top}\Phi)^{-1}\Phi^{\top}$$

and so

$$\check{b} = (\Phi^{\top}\Phi)^{-1}\Phi^{\top} X \tag{6.30}$$

The $p \times n$ matrix $\check{b}$ is recognizable as a matrix of least squares regression coefficients of the $m \times n$ matrix X on the $m \times p$ matrix Φ and consequently

$$\check{X} = \Phi \check{b} \tag{6.31}$$

is an $m \times n$ matrix of least squares projections of X on Φ.

Variance Decomposition of X

By virtue of the least-squares projection theory discussed in this quantecon lecture https://python-advanced.quantecon.org/orth_proj.html, we can represent X as the sum of the projection $\check{X}$ of X on Φ plus a matrix of errors.

To verify this, note that the least squares projection $\check{X}$ is related to X by

$$X = \check{X} + \epsilon$$

or

$$X = \Phi \check{b} + \epsilon \tag{6.32}$$

where ϵ is an $m \times n$ matrix of least squares errors satisfying the least squares orthogonality conditions $\epsilon^{\top}\Phi = 0$ or

$$(X - \Phi \check{b})^{\top}\Phi = 0_{m \times p} \tag{6.33}$$

Rearranging the orthogonality conditions (6.33) gives $X^{\top}\Phi = \check{b}\Phi^{\top}\Phi$, which implies formula (6.30).

6.5.2 An Approximation

We now describe a way to approximate the $p \times 1$ vector $\check{b}_t$ instead of using formula (6.29).

In particular, the following argument adapted from [Brunton and Kutz, 2022] (page 240) provides a computationally efficient way to approximate $\check{b}_t$.

For convenience, we'll apply the method at time $t = 1$.

For $t = 1$, from equation (6.32) we have

$$\check{X}_1 = \Phi \check{b}_1 \tag{6.34}$$

where $\check{b}_1$ is a $p \times 1$ vector.

Recall from representation 1 above that $X_1 = U\tilde{b}_1$, where $\tilde{b}_1$ is a time 1 basis vector for representation 1 and U is from the full SVD $X = U\Sigma V^\top$.

It then follows from equation (6.32) that

$$U\tilde{b}_1 = X'\tilde{V}\tilde{\Sigma}^{-1}\tilde{W}\check{b}_1 + \epsilon_1$$

where ϵ_1 is a least-squares error vector from equation (6.32).

It follows that

$$\tilde{b}_1 = U^\top X'V\tilde{\Sigma}^{-1}\tilde{W}\check{b}_1 + U^\top\epsilon_1$$

Replacing the error term $U^\top\epsilon_1$ by zero, and replacing U from a **full** SVD of X with $\tilde{U}$ from a **reduced** SVD, we obtain an approximation $\hat{b}_1$ to $\tilde{b}_1$:

$$\hat{b}_1 = \tilde{U}^\top X'\tilde{V}\tilde{\Sigma}^{-1}\tilde{W}\check{b}_1$$

Recall that from equation (6.23), $\tilde{A} = \tilde{U}^\top X'\tilde{V}\tilde{\Sigma}^{-1}$.

It then follows that

$$\hat{b}_1 = \tilde{A}\tilde{W}\check{b}_1$$

and therefore, by the eigendecomposition (6.24) of $\tilde{A}$, we have

$$\hat{b}_1 = \tilde{W}\Lambda\check{b}_1$$

Consequently,

$$\hat{b}_1 = (\tilde{W}\Lambda)^{-1}\tilde{b}_1$$

or

$$\hat{b}_1 = (\tilde{W}\Lambda)^{-1}\tilde{U}^\top X_1, \tag{6.35}$$

which is a computationally efficient approximation to the following instance of equation (6.29) for the initial vector $\check{b}_1$:

$$\check{b}_1 = \Phi^+ X_1 \tag{6.36}$$

(To highlight that (6.35) is an approximation, users of DMD sometimes call components of basis vector $\check{b}_t = \Phi^+ X_t$ the **exact** DMD modes and components of $\hat{b}_t = (\tilde{W}\Lambda)^{-1}\tilde{U}^\top X_t$ the **approximate** modes.)

Conditional on X_t, we can compute a decoded $\check{X}_{t+j}, j = 1, 2, \ldots$ from the exact modes via

$$\check{X}_{t+j} = \Phi\Lambda^j\Phi^+ X_t \tag{6.37}$$

or use compute a decoded $\hat{X}_{t+j}$ from approximate modes via

$$\hat{X}_{t+j} = \Phi\Lambda^j(\tilde{W}\Lambda)^{-1}\tilde{U}^\top X_t. \tag{6.38}$$

We can then use a decoded $\check{X}_{t+j}$ or $\hat{X}_{t+j}$ to forecast X_{t+j}.

6.5.3 Using Fewer Modes

In applications, we'll actually use only a few modes, often three or less.

Some of the preceding formulas assume that we have retained all p modes associated with singular values of X.

We can adjust our formulas to describe a situation in which we instead retain only the $r < p$ largest singular values.

In that case, we simply replace $\tilde{\Sigma}$ with the appropriate $r \times r$ matrix of singular values, $\tilde{U}$ with the $m \times r$ matrix whose columns correspond to the r largest singular values, and $\tilde{V}$ with the $n \times r$ matrix whose columns correspond to the r largest singular values.

Counterparts of all of the salient formulas above then apply.

6.6 Source for Some Python Code

You can find a Python implementation of DMD here:

https://mathlab.sissa.it/pydmd

CHAPTER

SEVEN

USING NEWTON'S METHOD TO SOLVE ECONOMIC MODELS

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

Contents

- *Using Newton's Method to Solve Economic Models*
  - *Overview*
  - *Fixed point computation using Newton's method*
  - *Root-Finding in one dimension*
  - *Multivariate Newton's method*
  - *Exercises*

7.1 Overview

Many economic problems involve finding fixed points or zeros (also called "roots") of functions.

For example, in a simple supply and demand model, an equilibrium price is one that makes excess demand zero.

In other words, an equilibrium is a zero of the excess demand function.

There are various computational techniques for solving for fixed points and zeros.

In this lecture we study an important gradient-based technique called Newton's method.

Newton's method does not always work but, in situations where it does, convergence is often fast when compared to other methods.

The lecture will apply Newton's method in one-dimensional and multidimensional settings to solve fixed-point and zero-finding problems.

- When finding the fixed point of a function f, Newton's method updates an existing guess of the fixed point by solving for the fixed point of a linear approximation to the function f.
- When finding the zero of a function f, Newton's method updates an existing guess by solving for the zero of a linear approximation to the function f.

To build intuition, we first consider an easy, one-dimensional fixed point problem where we know the solution and solve it using both successive approximation and Newton's method.

Then we apply Newton's method to multidimensional settings to solve for market equilibria with multiple goods.

At the end of the lecture, we leverage the power of automatic differentiation in `jax` to solve a very high-dimensional equilibrium problem.

We use the following imports in this lecture

```
import matplotlib.pyplot as plt
from typing import NamedTuple
from scipy.optimize import root
import jax.numpy as jnp
import jax

# Enable 64-bit precision
jax.config.update("jax_enable_x64", True)
```

7.2 Fixed point computation using Newton's method

In this section we solve the fixed point of the law of motion for capital in the setting of the Solow growth model.

We will inspect the fixed point visually, solve it by successive approximation, and then apply Newton's method to achieve faster convergence.

7.2.1 The Solow model

In the Solow growth model, assuming Cobb-Douglas production technology and zero population growth, the law of motion for capital is

$$k_{t+1} = g(k_t) \quad \text{where} \quad g(k) := sAk^\alpha + (1-\delta)k \tag{7.1}$$

Here

- k_t is capital stock per worker,
- $A, \alpha > 0$ are production parameters with $\alpha < 1$
- $s > 0$ is a savings rate, and
- $\delta \in (0, 1)$ is a rate of depreciation

In this example, we wish to calculate the unique strictly positive fixed point of g, the law of motion for capital.

In other words, we seek a $k^* > 0$ such that $g(k^*) = k^*$.

- Such a k^* is called a steady state, since $k_t = k^*$ implies $k_{t+1} = k^*$.

Using pencil and paper to solve $g(k) = k$, you will be able to confirm that

$$k^* = \left(\frac{sA}{\delta}\right)^{1/(1-\alpha)}$$

7.2.2 Implementation

Let's store our parameters in `NamedTuple` to help us keep our code clean and concise.

```
class SolowParameters(NamedTuple):
    A: float
    s: float
    α: float
    δ: float
```

This function creates a suitable `SolowParameters` with default parameter values.

```
def create_solow_params(A=2.0, s=0.3, α=0.3, δ=0.4):
    """Creates a Solow model parameterization with default values."""
    return SolowParameters(A=A, s=s, α=α, δ=δ)
```

The next two functions implement the law of motion *(7.2.1)* and store the true fixed point k^*.

```
def g(k, params):
    A, s, α, δ = params
    return A * s * k**α + (1 - δ) * k


def exact_fixed_point(params):
    A, s, α, δ = params
    return ((s * A) / δ) ** (1 / (1 - α))
```

Here is a function to provide a 45 degree plot of the dynamics.

```
def plot_45(params, ax, fontsize=14):

    k_min, k_max = 0.0, 3.0
    k_grid = jnp.linspace(k_min, k_max, 1200)

    # Plot the functions
    lb = r"$g(k) = sAk^{\alpha} + (1 - \delta)k$"
    ax.plot(k_grid, g(k_grid, params), lw=2, alpha=0.6, label=lb)
    ax.plot(k_grid, k_grid, "k--", lw=1, alpha=0.7, label="45")

    # Show and annotate the fixed point
    kstar = exact_fixed_point(params)
    fps = (kstar,)
    ax.plot(fps, fps, "go", ms=10, alpha=0.6)
    ax.annotate(
        r"$k^* = (sA / \delta)^{\frac{1}{1-\alpha}}$",
        xy=(kstar, kstar),
        xycoords="data",
        xytext=(20, -20),
        textcoords="offset points",
        fontsize=fontsize,
    )

    ax.legend(loc="upper left", frameon=False, fontsize=fontsize)

    ax.set_yticks((0, 1, 2, 3))
    ax.set_yticklabels((0.0, 1.0, 2.0, 3.0), fontsize=fontsize)
    ax.set_ylim(0, 3)
```

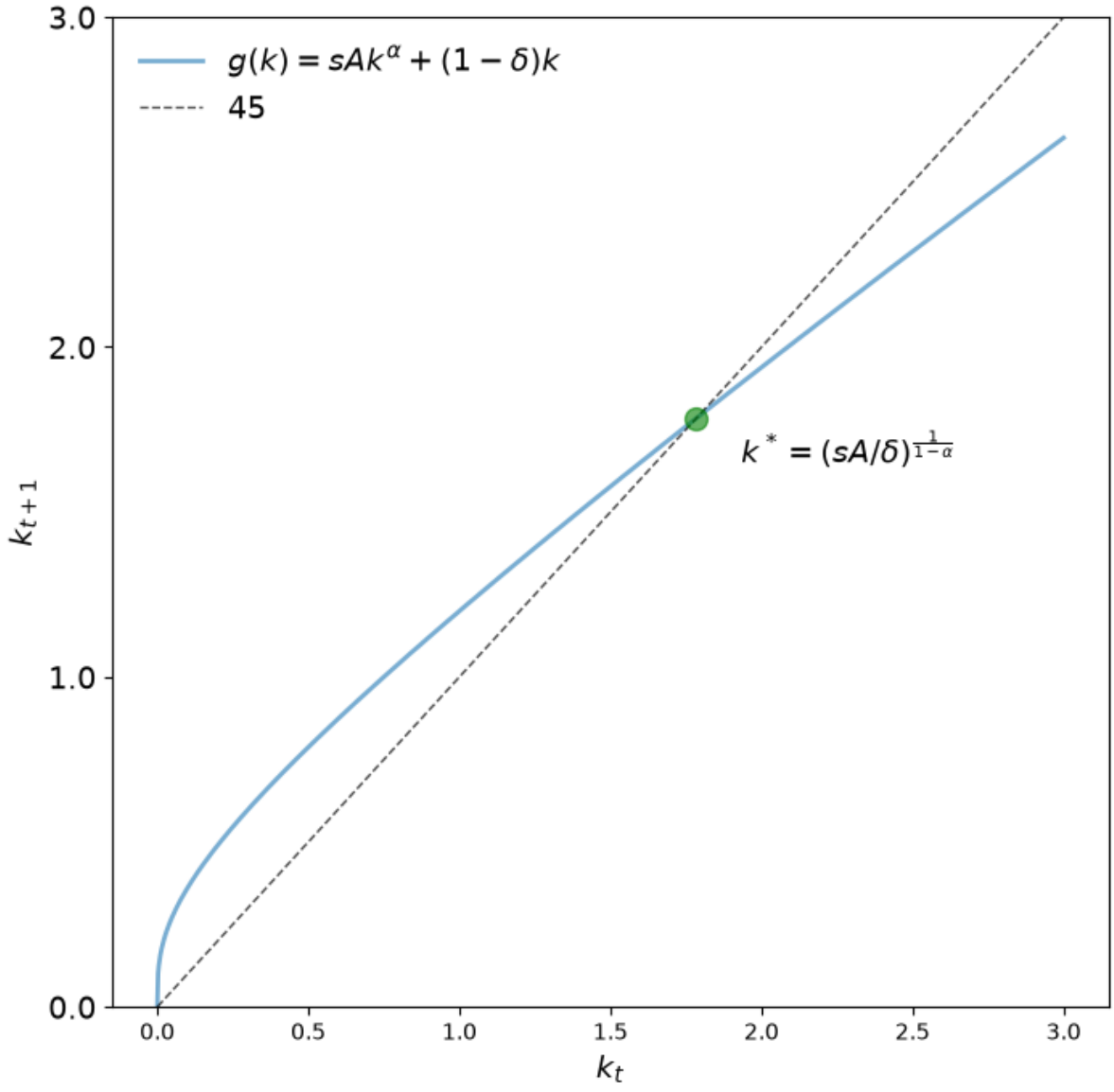
(continues on next page)

(continued from previous page)

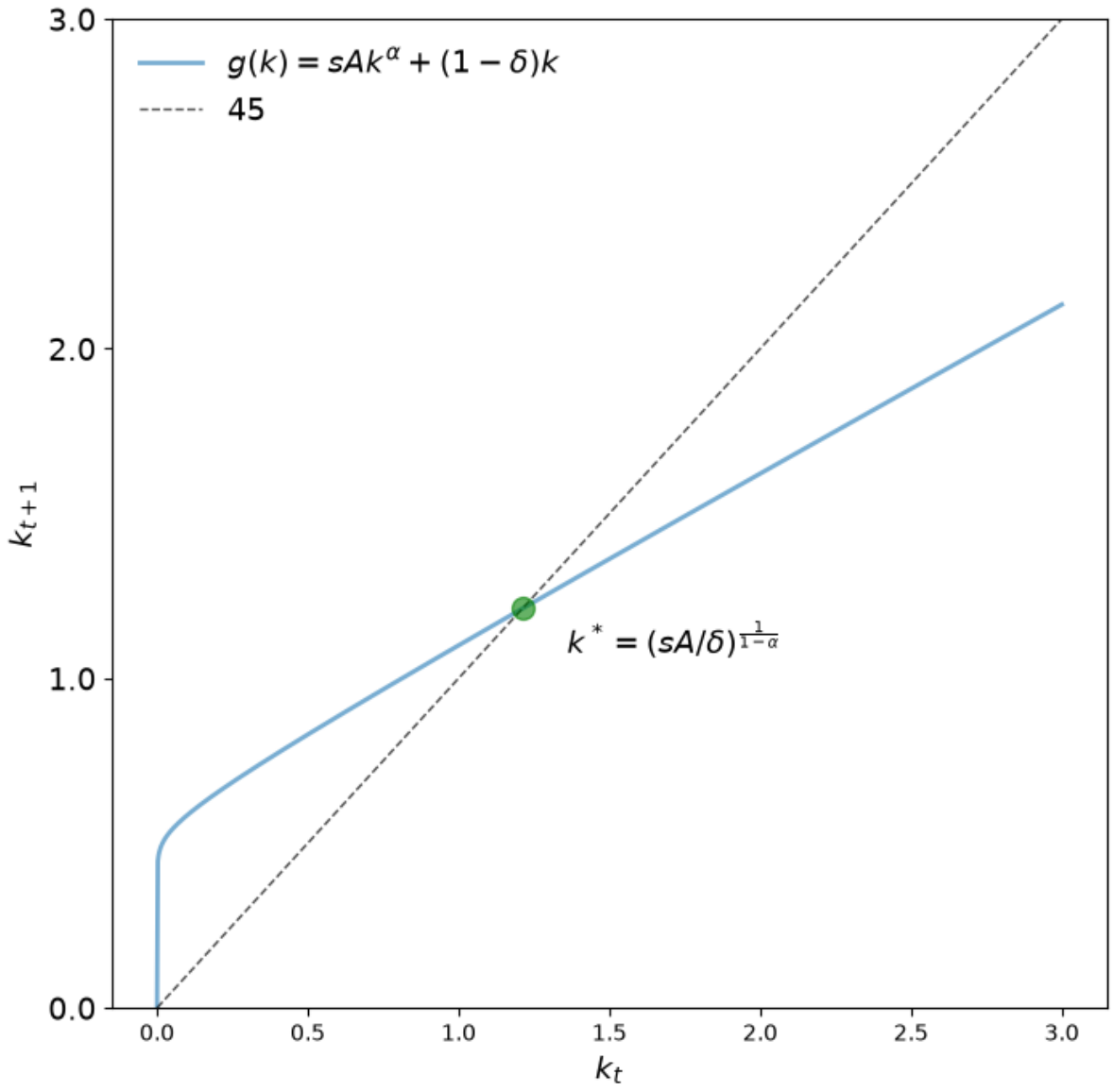
```
    ax.set_xlabel("$k_t$", fontsize=fontsize)
    ax.set_ylabel("$k_{t+1}$", fontsize=fontsize)
```

Let's look at the 45 degree diagram for two parameterizations.

```
params = create_solow_params()
fig, ax = plt.subplots(figsize=(8, 8))
plot_45(params, ax)
plt.show()
```




```
params = create_solow_params(α=0.05, δ=0.5)
fig, ax = plt.subplots(figsize=(8, 8))
plot_45(params, ax)
plt.show()
```



We see that k^* is indeed the unique positive fixed point.

Successive approximation

First let's compute the fixed point using successive approximation.

In this case, successive approximation means repeatedly updating capital from some initial state k_0 using the law of motion.

Here's a time series from a particular choice of k_0.

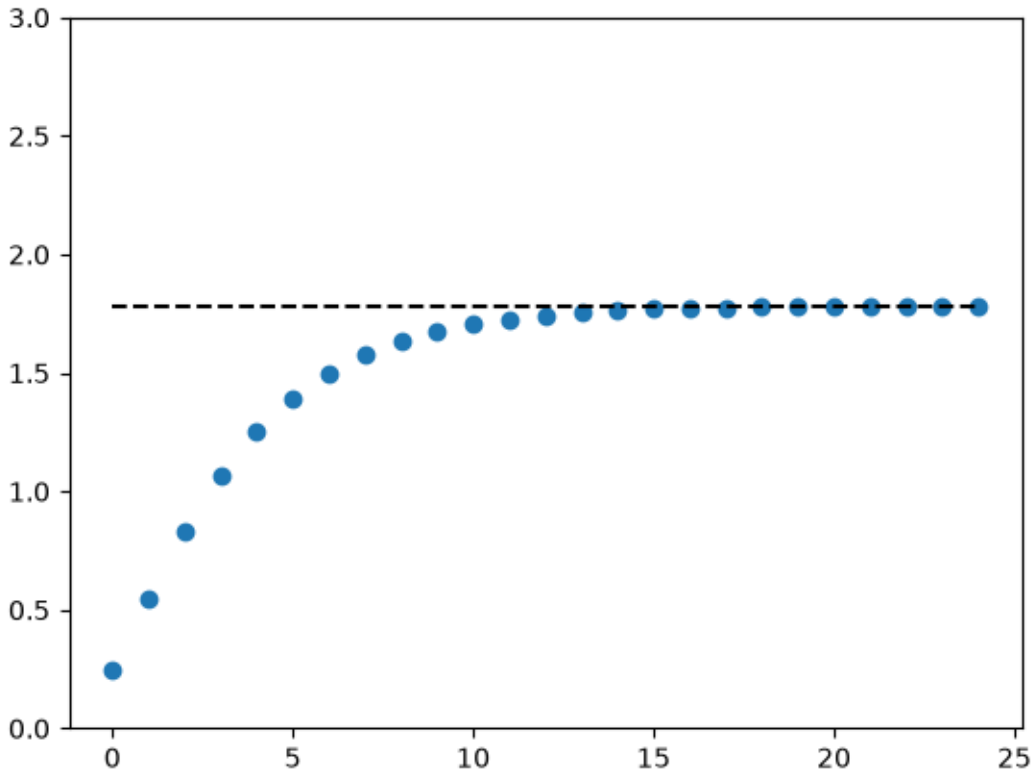
```
def compute_iterates(k_0, f, params, n=25):
    """Compute time series of length n generated by function f."""
    k = k_0
    k_iterates = []
    for t in range(n):
        k_iterates.append(k)
```

(continues on next page)

(continued from previous page)

```
        k = f(k, params)
    return k_iterates
```

```
params = create_solow_params()
k_0 = 0.25
k_series = compute_iterates(k_0, g, params)
k_star = exact_fixed_point(params)

fig, ax = plt.subplots()
ax.plot(k_series, "o")
ax.plot([k_star] * len(k_series), "k--")
ax.set_ylim(0, 3)
plt.show()
```



Let's see the output for a long time series.

```
k_series = compute_iterates(k_0, g, params, n=10_000)
k_star_approx = k_series[-1]
k_star_approx
```

```
1.7846741842265788
```

This is close to the true value.

```
k_star
```

```
1.7846741842265788
```

Newton's method

In general, when applying Newton's fixed point method to some function g, we start with a guess x_0 of the fixed point and then update by solving for the fixed point of a tangent line at x_0.

To begin with, we recall that the first-order approximation of g at x_0 (i.e., the first order Taylor approximation of g at x_0) is the function

$$\hat{g}(x) \approx g(x_0) + g'(x_0)(x - x_0) \tag{7.2}$$

We solve for the fixed point of $\hat{g}$ by calculating the x_1 that solves

$$x_1 = \frac{g(x_0) - g'(x_0)x_0}{1 - g'(x_0)}$$

Generalising the process above, Newton's fixed point method iterates on

$$x_{t+1} = \frac{g(x_t) - g'(x_t)x_t}{1 - g'(x_t)}, \quad x_0 \text{ given} \tag{7.3}$$

To implement Newton's method we observe that the derivative of the law of motion for capital *(7.2.1)* is

$$g'(k) = \alpha s A k^{\alpha - 1} + (1 - \delta) \tag{7.4}$$

Let's define this:

```
def Dg(k, params):
    A, s, α, δ = params
    return α * A * s * k ** (α - 1) + (1 - δ)
```

Here's a function q representing *(7.2.3)*.

```
def q(k, params):
    return (g(k, params) - Dg(k, params) * k) / (1 - Dg(k, params))
```

Now let's plot some trajectories.

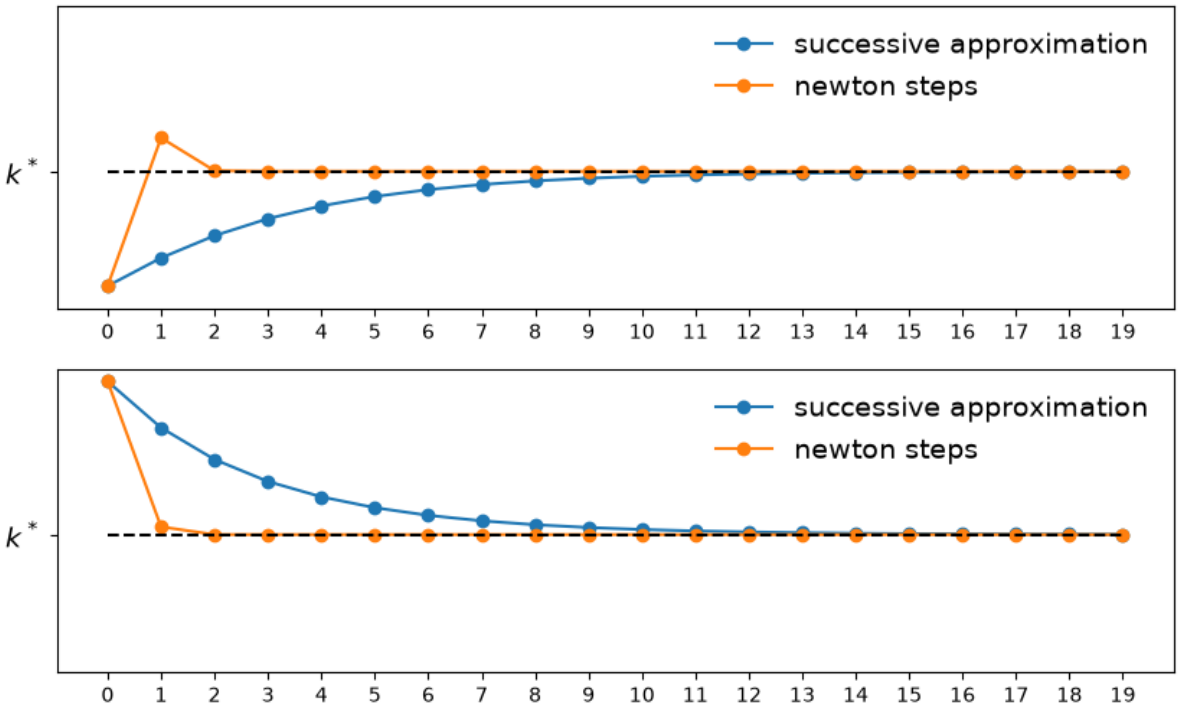
```
def plot_trajectories(
    params,
    k0_a=0.8,  # first initial condition
    k0_b=3.1,  # second initial condition
    n=20,      # length of time series
    fs=14,     # fontsize
):

    fig, axes = plt.subplots(2, 1, figsize=(10, 6))
    ax1, ax2 = axes

    ks1 = compute_iterates(k0_a, g, params, n)
    ax1.plot(ks1, "-o", label="successive approximation")

    ks2 = compute_iterates(k0_b, g, params, n)
    ax2.plot(ks2, "-o", label="successive approximation")

    ks3 = compute_iterates(k0_a, q, params, n)
    ax1.plot(ks3, "-o", label="newton steps")

    ks4 = compute_iterates(k0_b, q, params, n)
```

(continues on next page)

(continued from previous page)

```
    ax2.plot(ks4, "-o", label="newton steps")

    for ax in axes:
        ax.plot(k_star * jnp.ones(n), "k--")
        ax.legend(fontsize=fs, frameon=False)
        ax.set_ylim(0.6, 3.2)
        ax.set_yticks((k_star,))
        ax.set_yticklabels(("$k^*$",), fontsize=fs)
        ax.set_xticks(jnp.linspace(0, 19, 20))

    plt.show()
```

```
params = create_solow_params()
plot_trajectories(params)
```



We can see that Newton's method converges faster than successive approximation.

7.3 Root-Finding in one dimension

In the previous section we computed fixed points.

In fact Newton's method is more commonly associated with the problem of finding zeros of functions.

Let's discuss this "root-finding" problem and then show how it is connected to the problem of finding fixed points.

7.3.1 Newton's method for zeros

Let's suppose we want to find an x such that $f(x) = 0$ for some smooth function f mapping real numbers to real numbers.

Suppose we have a guess x_0 and we want to update it to a new point x_1.

As a first step, we take the first-order approximation of f around x_0:

$$\hat{f}(x) \approx f(x_0) + f'(x_0)(x - x_0)$$

Now we solve for the zero of $\hat{f}$.

In particular, we set $\hat{f}(x_1) = 0$ and solve for x_1 to get

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}, \quad x_0 \text{ given}$$

Generalizing the formula above, for one-dimensional zero-finding problems, Newton's method iterates on

$$x_{t+1} = x_t - \frac{f(x_t)}{f'(x_t)}, \quad x_0 \text{ given} \tag{7.5}$$

The following code implements the iteration *(7.3.1)*

```
def newton(f, x_0, tol=1e-7, max_iter=100_000):
    x = x_0
    Df = jax.grad(f)

    # Implement the zero-finding formula
    @jax.jit
    def q(x):
        return x - f(x) / Df(x)

    error = tol + 1
    n = 0
    while error > tol:
        n += 1
        if n > max_iter:
            raise Exception("Max iteration reached without convergence")
        y = q(x)
        error = jnp.abs(x - y)
        x = y
        print(f"iteration {n}, error = {error:.5f}")
    return x.item()
```

Numerous libraries implement Newton's method in one dimension, including SciPy, so the code is just for illustrative purposes.

(That said, when we want to apply Newton's method using techniques such as automatic differentiation or GPU acceleration, it will be helpful to know how to implement Newton's method ourselves.)

7.3.2 Application to finding fixed points

Now consider again the Solow fixed-point calculation, where we solve for k satisfying $g(k) = k$.

We can convert to this to a zero-finding problem by setting $f(x) := g(x) - x$.

Any zero of f is clearly a fixed point of g.

Let's apply this idea to the Solow problem

```
params = create_solow_params()
k_star_approx_newton = newton(f = lambda x: g(x, params) - x, x_0=0.8)
```

```
iteration 1, error = 1.27209
iteration 2, error = 0.28180
iteration 3, error = 0.00561
iteration 4, error = 0.00000
iteration 5, error = 0.00000
```

```
k_star_approx_newton
```

```
1.7846741842265788
```

The result confirms convergence we saw in the graphs above: a very accurate result is reached with only 5 iterations.

7.4 Multivariate Newton's method

In this section, we introduce a two-good problem, present a visualization of the problem, and solve for the equilibrium of the two-good market using both a zero finder in `SciPy` and Newton's method.

We then expand the idea to a larger market with 5,000 goods and compare the performance of the two methods again.

We will see a significant performance gain when using Newton's method.

7.4.1 A two-goods market equilibrium

Let's start by computing the market equilibrium of a two-good problem.

We consider a market for two related products, good 0 and good 1, with price vector $p = (p_0, p_1)$

Supply of good i at price p is

$$q_i^s(p) = b_i \sqrt{p_i}$$

Demand of good i at price p is

$$q_i^d(p) = \exp(-(a_{i0} p_0 + a_{i1} p_1)) + c_i$$

Here c_i, b_i and a_{ij} are parameters.

For example, the two goods might be computer components that are typically used together, in which case they are complements. Hence demand depends on the price of both components.

The excess demand function is

$$e_i(p) = q_i^d(p) - q_i^s(p), \quad i = 0, 1$$

An equilibrium price vector p^* satisfies $e_i(p^*) = 0$.

We set

$$A = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix}, \qquad b = \begin{bmatrix} b_0 \\ b_1 \end{bmatrix} \qquad \text{and} \qquad c = \begin{bmatrix} c_0 \\ c_1 \end{bmatrix}$$

for this particular question.

A graphical exploration

Since our problem is only two-dimensional, we can use graphical analysis to visualize and help understand the problem.

Our first step is to define the excess demand function

$$e(p) = \begin{bmatrix} e_0(p) \\ e_1(p) \end{bmatrix}$$

The function below calculates the excess demand for given parameters

```
@jax.jit
def e(p, A, b, c):
    return jnp.exp(-A @ p) + c - b * jnp.sqrt(p)
```

Our default parameter values will be

$$A = \begin{bmatrix} 0.5 & 0.4 \\ 0.8 & 0.2 \end{bmatrix}, \qquad b = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \qquad \text{and} \qquad c = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

```
A = jnp.array([[0.5, 0.4], [0.8, 0.2]])
b = jnp.ones(2)
c = jnp.ones(2)
```

At a price level of $p = (1, 0.5)$, the excess demand is

```
p = jnp.array([1, 0.5])
ex_demand = e(p, A, b, c)

print(
    f"The excess demand for good 0 is {ex_demand[0]:.3f} \n"
    f"The excess demand for good 1 is {ex_demand[1]:.3f}"
)
```

```
The excess demand for good 0 is 0.497
The excess demand for good 1 is 0.699
```

To increase the efficiency of computation, we will use the power of vectorization using `jax.vmap`. This is much faster than the python loops.

```
# Create vectorization on the first axis of p.
e_vectorized_p_1 = jax.vmap(e, in_axes=(0, None, None, None))

# Create vectorization on the second axis of p.
e_vectorized = jax.vmap(e_vectorized_p_1, in_axes=(0, None, None, None))
```

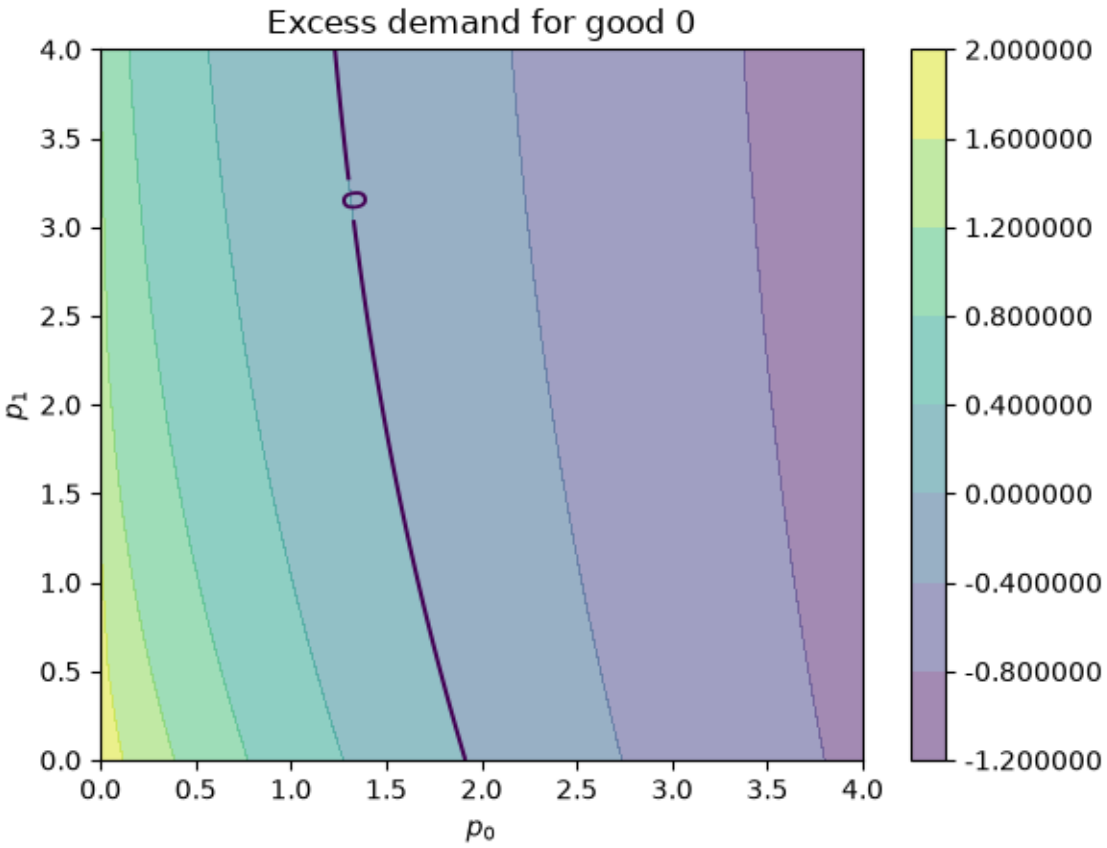
Next we plot the two functions e_0 and e_1 on a grid of (p_0, p_1) values, using contour surfaces and lines.

We will use the following function to build the contour plots

```
def plot_excess_demand(ax, good=0, grid_size=100, grid_max=4, surface=True):
    p_grid = jnp.linspace(0, grid_max, grid_size)

    # Create meshgrid for all combinations of p_1 and p_2
    P1, P2 = jnp.meshgrid(p_grid, p_grid, indexing="ij")

    # Stack to create array of shape (grid_size, grid_size, 2)
    P = jnp.stack([P1, P2], axis=-1)

    # Compute all values at once using vectorized function
    z_full = e_vectorized(P, A, b, c)
    z = z_full[:, :, good]

    if surface:
        cs1 = ax.contourf(p_grid, p_grid, z.T, alpha=0.5)
        plt.colorbar(cs1, ax=ax, format="%.6f")

    ctr1 = ax.contour(p_grid, p_grid, z.T, levels=[0.0])
    ax.set_xlabel("$p_0$")
    ax.set_ylabel("$p_1$")
    ax.set_title(f"Excess demand for good {good}")
    plt.clabel(ctr1, inline=1, fontsize=13)
```

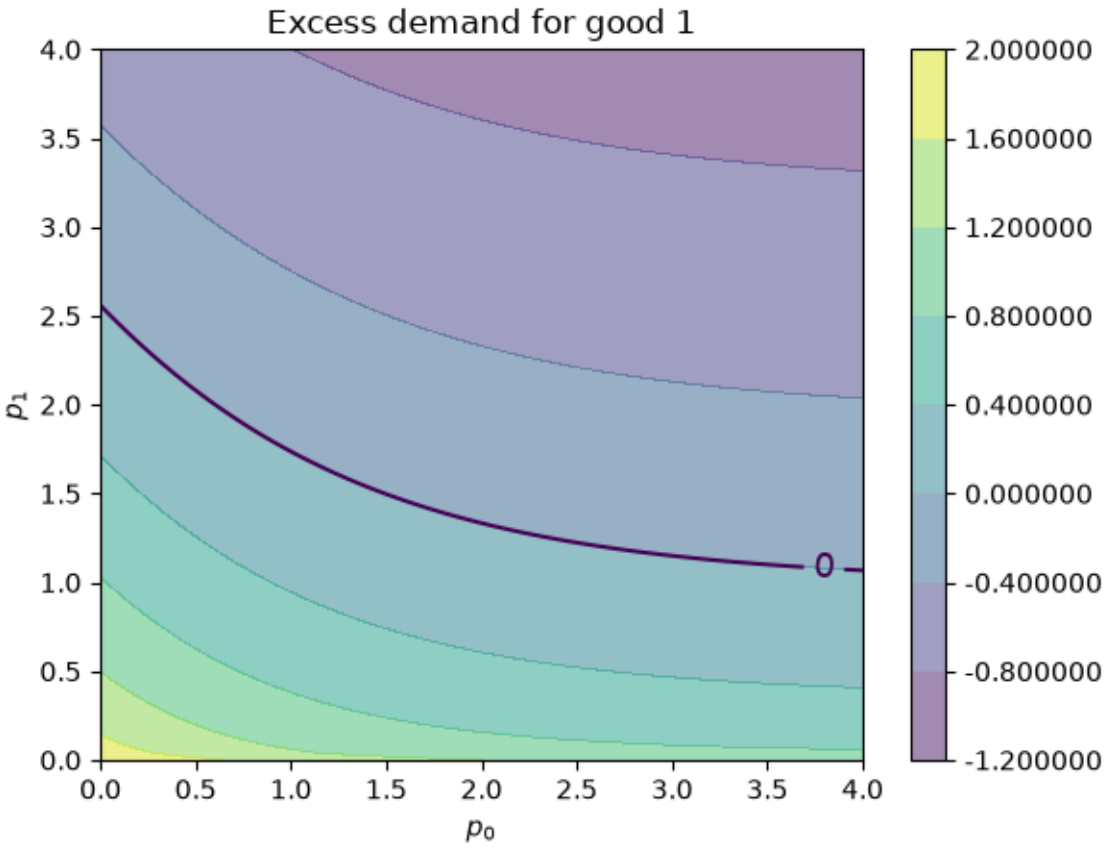
Here's our plot of e_0:

```
fig, ax = plt.subplots()
plot_excess_demand(ax, good=0)
plt.show()
```



Here's our plot of e_1:

```
fig, ax = plt.subplots()
plot_excess_demand(ax, good=1)
plt.show()
```

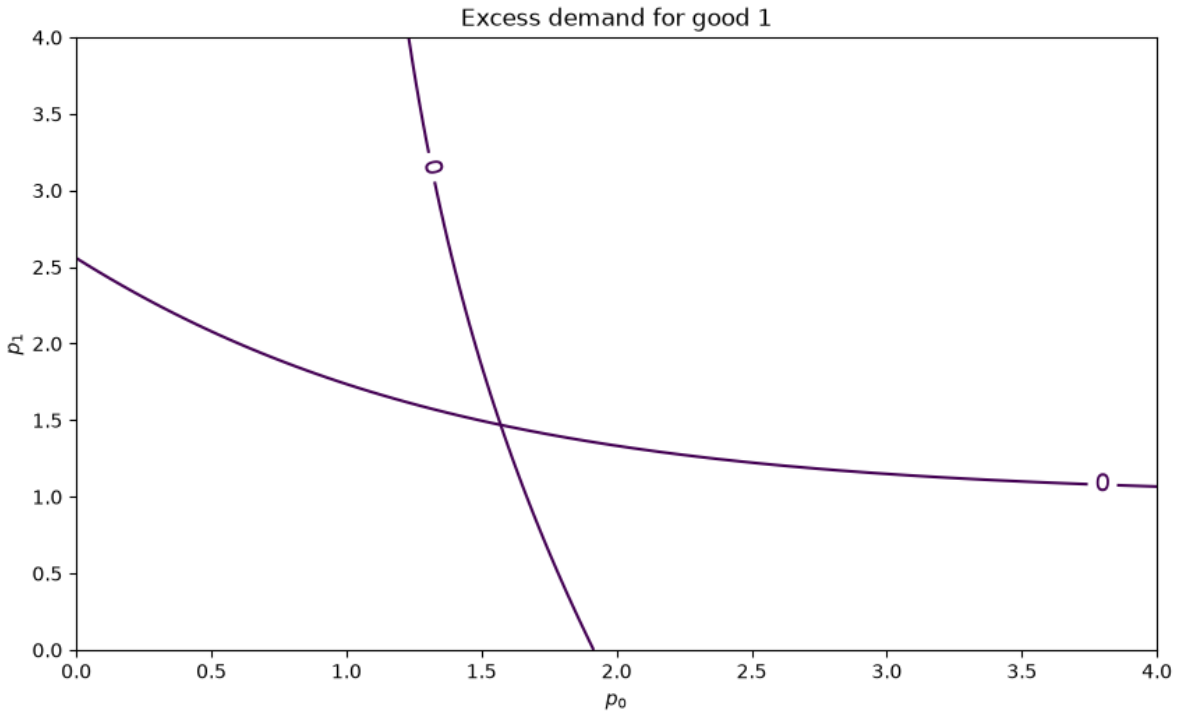


We see the black contour line of zero, which tells us when $e_i(p) = 0$.

For a price vector p such that $e_i(p) = 0$ we know that good i is in equilibrium (demand equals supply).

If these two contour lines cross at some price vector p^*, then p^* is an equilibrium price vector.

```
fig, ax = plt.subplots(figsize=(10, 5.7))
for good in (0, 1):
    plot_excess_demand(ax, good=good, surface=False)
plt.show()
```



It seems there is an equilibrium close to $p = (1.6, 1.5)$.

Using a multidimensional root finder

To solve for p^* more precisely, we use a zero-finding algorithm from `scipy.optimize`.

We supply $p = (1, 1)$ as our initial guess.

```
init_p = jnp.ones(2)
```

This uses the modified Powell method to find the zero

```
%%time
solution = root(lambda p: e(p, A, b, c), init_p, method="hybr")
```

```
CPU times: user 6.13 ms, sys: 1.42 ms, total: 7.55 ms
Wall time: 4.87 ms
```

Here's the resulting value:

```
p = solution.x
p
```

```
array([1.57080182, 1.46928838])
```

This looks close to our guess from observing the figure. We can plug it back into e to test that $e(p) \approx 0$:

```
e_p = jnp.max(jnp.abs(e(p, A, b, c)))
e_p.item()
```

```
2.0383694732117874e-13
```

This is indeed a very small error.

Adding gradient information

In many cases, for zero-finding algorithms applied to smooth functions, supplying the Jacobian of the function leads to better convergence properties.

Here, we manually calculate the elements of the Jacobian

$$J(p) = \begin{bmatrix} \frac{\partial e_0}{\partial p_0}(p) & \frac{\partial e_0}{\partial p_1}(p) \\ \frac{\partial e_1}{\partial p_0}(p) & \frac{\partial e_1}{\partial p_1}(p) \end{bmatrix}$$

```
def jacobian_e(p, A, b, c):
    p_0, p_1 = p
    a_00, a_01 = A[0, :]
    a_10, a_11 = A[1, :]
    j_00 = -a_00 * jnp.exp(-a_00 * p_0) - (b[0] / 2) * p_0 ** (-1 / 2)
    j_01 = -a_01 * jnp.exp(-a_01 * p_1)
    j_10 = -a_10 * jnp.exp(-a_10 * p_0)
    j_11 = -a_11 * jnp.exp(-a_11 * p_1) - (b[1] / 2) * p_1 ** (-1 / 2)
    J = [[j_00, j_01], [j_10, j_11]]
    return jnp.array(J)
```

```
%%time
solution = root(
    lambda p: e(p, A, b, c),
    init_p,
    jac = lambda p: jacobian_e(p, A, b, c),
    method="hybr",
)
```

```
CPU times: user 252 ms, sys: 13 ms, total: 265 ms
Wall time: 373 ms
```

Now the solution is even more accurate (although, in this low-dimensional problem, the difference is quite small):

```
p = solution.x
e_p = jnp.max(jnp.abs(e(p, A, b, c)))
e_p.item()
```

```
1.3322676295501878e-15
```

Using Newton's method

Now let's use Newton's method to compute the equilibrium price using the multivariate version of Newton's method

$$p_{n+1} = p_n - J_e(p_n)^{-1} e(p_n) \tag{7.6}$$

This is a multivariate version of *(7.3.1)*

(Here $J_e(p_n)$ is the Jacobian of e evaluated at p_n.)

The iteration starts from some initial guess of the price vector p_0.

Here, instead of coding Jacobian by hand, we use the `jacobian()` function in the `jax` library to auto-differentiate and calculate the Jacobian.

With only slight modification, we can generalize *our previous attempt* to multidimensional problems

```
def newton(f, x_0, tol=1e-5, max_iter=10):
    x = x_0
    f_jac = jax.jacobian(f)

    @jax.jit
    def q(x):
        return x - jnp.linalg.solve(f_jac(x), f(x))

    error = tol + 1
    n = 0
    while error > tol:
        n += 1
        if n > max_iter:
            raise Exception("Max iteration reached without convergence")
        y = q(x)
        if any(jnp.isnan(y)):
            raise Exception("Solution not found with NaN generated")
        error = jnp.linalg.norm(x - y)
        x = y
        print(f"iteration {n}, error = {error:.5f}")
    print("\n" + f"Result = {x} \n")
    return x
```

We find the algorithm terminates in 4 steps

```
%%time
p = newton(lambda p: e(p, A, b, c), init_p)
```

```
iteration 1, error = 0.62515
iteration 2, error = 0.11152
iteration 3, error = 0.00258
iteration 4, error = 0.00000

Result = [1.57080182 1.46928838]

CPU times: user 303 ms, sys: 18.6 ms, total: 321 ms
Wall time: 353 ms
```

```
e_p = jnp.max(jnp.abs(e(p, A, b, c)))
e_p.item()
```

```
1.4632739464559563e-13
```

The result is very accurate.

With the larger overhead, the speed is not better than the optimized `scipy` function.

7.4.2 A high-dimensional problem

Our next step is to investigate a large market with 3,000 goods.

The excess demand function is essentially the same, but now the matrix A is 3000×3000 and the parameter vectors b and c are 3000×1.

```
dim = 3000

# Create JAX random key
key = jax.random.PRNGKey(0)

# Create a random matrix A and normalize the columns to sum to one
A = jax.random.uniform(key, (dim, dim))
s = jnp.sum(A, axis=0)
A = A / s

# Set up b and c
b = jnp.ones(dim)
c = jnp.ones(dim)
```

Here's our initial condition

```
init_p = jnp.ones(dim)
```

```
%%time
p = newton(lambda p: e(p, A, b, c), init_p)
```

```
iteration 1, error = 23.22262
iteration 2, error = 3.94537
iteration 3, error = 0.08500
iteration 4, error = 0.00004
iteration 5, error = 0.00000

Result = [1.50723773 1.51041603 1.50134795 ... 1.49941629 1.49033692 1.49666807]

CPU times: user 5.68 s, sys: 1.65 s, total: 7.33 s
Wall time: 7.15 s
```

```
e_p = jnp.max(jnp.abs(e(p, A, b, c)))
e_p.item()
```

```
4.440892098500626e-16
```

With the same tolerance, we compare the runtime and accuracy of Newton's method to SciPy's `root` function

```
%%time
solution = root(
    lambda p: e(p, A, b, c),
    init_p,
    jac = lambda p: jax.jacobian(e)(p, A, b, c),
    method="hybr",
    tol=1e-5,
)
```

```
CPU times: user 34.6 s, sys: 68.8 ms, total: 34.7 s
Wall time: 35.1 s
```

```
p = solution.x
e_p = jnp.max(jnp.abs(e(p, A, b, c)))
e_p.item()
```

```
9.209231102147442e-07
```

7.5 Exercises

Exercise 7.5.1

Consider a three-dimensional extension of the Solow fixed point problem with

$$A = \begin{bmatrix} 2 & 3 & 3 \\ 2 & 4 & 2 \\ 1 & 5 & 1 \end{bmatrix}, \quad s = 0.2, \quad \alpha = 0.5, \quad \delta = 0.8$$

As before the law of motion is

$$k_{t+1} = g(k_t) \quad \text{where} \quad g(k) := sAk^\alpha + (1 - \delta)k$$

However, k_t is now a 3×1 vector.

Solve for the fixed point using Newton's method with the following initial values:

$$k1_0 = (1, 1, 1)$$
$$k2_0 = (3, 5, 5)$$
$$k3_0 = (50, 50, 50)$$

Hint

- The computation of the fixed point is equivalent to computing k^* such that $g(k^*) - k^* = 0$.
- If you are unsure about your solution, you can start with the solved example:

$$A = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{bmatrix}$$

with $s = 0.3$, $\alpha = 0.3$, and $\delta = 0.4$ and starting value:

$$k_0 = (1, 1, 1)$$

The result should converge to the *analytical solution*.

Solution

Let's first define the parameters for this problem

```
A = jnp.array([[2.0, 3.0, 3.0], [2.0, 4.0, 2.0], [1.0, 5.0, 1.0]])

s = 0.2
α = 0.5
δ = 0.8

initLs = [jnp.ones(3), jnp.array([3.0, 5.0, 5.0]), jnp.repeat(50.0, 3)]
```

Then define the multivariate version of the formula for the *(7.2.1)*

```
@jax.jit
def multivariate_solow(k, A=A, s=s, α=α, δ=δ):
    return s * jnp.dot(A, k**α) + (1 - δ) * k
```

Let's run through each starting value and see the output

```
attempt = 1
for init in initLs:
    print(f'Attempt {attempt}: Starting value is {init} \n')
    %time k = newton(lambda k: multivariate_solow(k) - k, \
                     init)
    print('-'*64)
    attempt += 1
```

```
Attempt 1: Starting value is [1. 1. 1.]
iteration 1, error = 50.49630
iteration 2, error = 41.10937
iteration 3, error = 4.29413
iteration 4, error = 0.38543
iteration 5, error = 0.00544
iteration 6, error = 0.00000

Result = [3.84058108 3.87071771 3.41091933]

CPU times: user 285 ms, sys: 14 ms, total: 299 ms
Wall time: 316 ms
----------------------------------------------------------------
Attempt 2: Starting value is [3. 5. 5.]

iteration 1, error = 2.07011
iteration 2, error = 0.12642
iteration 3, error = 0.00060
iteration 4, error = 0.00000

Result = [3.84058108 3.87071771 3.41091933]

CPU times: user 152 ms, sys: 9.06 ms, total: 161 ms
Wall time: 120 ms
----------------------------------------------------------------
Attempt 3: Starting value is [50. 50. 50.]
iteration 1, error = 73.00943
iteration 2, error = 6.49379
iteration 3, error = 0.68070
iteration 4, error = 0.01620
iteration 5, error = 0.00001
iteration 6, error = 0.00000

Result = [3.84058108 3.87071771 3.41091933]

CPU times: user 341 ms, sys: 25.7 ms, total: 366 ms
Wall time: 292 ms
```

```
------------------------------------------------------------------
```

We find that the results are invariant to the starting values given the well-defined property of this question.

But the number of iterations it takes to converge is dependent on the starting values.

Let's substitute the output back into the formula to check our last result

```
multivariate_solow(k) - k
```

```
Array([0., 0., 0.], dtype=float64)
```

Note the error is very small.

We can also test our results on the known solution

```
A = jnp.array([[2.0, 0.0, 0.0], [0.0, 2.0, 0.0], [0.0, 0.0, 2.0]])

s = 0.3
α = 0.3
δ = 0.4

init = jnp.repeat(1.0, 3)
%%time

k = newton(lambda k: multivariate_solow(k, A=A, s=s, α=α, δ=δ) - k, init)
```

```
iteration 1, error = 1.57459
iteration 2, error = 0.21345
iteration 3, error = 0.00205
iteration 4, error = 0.00000

Result = [1.78467418 1.78467418 1.78467418]

CPU times: user 320 ms, sys: 17.4 ms, total: 337 ms
Wall time: 253 ms
```

The result is very close to the ground truth but still slightly different.

```
%%time

k = newton(
    lambda k: multivariate_solow(k, A=A, s=s, α=α, δ=δ) - k, init, tol=1e-7
)
```

```
iteration 1, error = 1.57459
iteration 2, error = 0.21345
iteration 3, error = 0.00205
iteration 4, error = 0.00000
iteration 5, error = 0.00000

Result = [1.78467418 1.78467418 1.78467418]

CPU times: user 313 ms, sys: 13.6 ms, total: 326 ms
Wall time: 242 ms
```

We can see it steps towards a more accurate solution.

Exercise 7.5.2

In this exercise, let's try different initial values and check how Newton's method responds to different starting points.

Let's define a three-good problem with the following default values:

$$A = \begin{bmatrix} 0.2 & 0.1 & 0.7 \\ 0.3 & 0.2 & 0.5 \\ 0.1 & 0.8 & 0.1 \end{bmatrix}, \qquad b = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \qquad \text{and} \qquad c = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

For this exercise, use the following extreme price vectors as initial values:

$$p1_0 = (5, 5, 5)$$
$$p2_0 = (1, 1, 1)$$
$$p3_0 = (4.5, 0.1, 4)$$

Set the tolerance to $1e - 15$ for more accurate output.

Solution

Define parameters and initial values

```
A = jnp.array([[0.2, 0.1, 0.7], [0.3, 0.2, 0.5], [0.1, 0.8, 0.1]])
b = jnp.array([1.0, 1.0, 1.0])
c = jnp.array([1.0, 1.0, 1.0])

initLs = [jnp.repeat(5.0, 3), jnp.ones(3), jnp.array([4.5, 0.1, 4.0])]
```

Let's run through each initial guess and check the output

```
for attempt, init in enumerate(initLs, start=1):
    print(f"Attempt {attempt}: Starting value is {init} \n")
    %time p = newton(lambda p: e(p, A, b, c), init, tol=1e-15, max_iter=15)
    print("-" * 64)
```

```
Attempt 1: Starting value is [5. 5. 5.]
iteration 1, error = 9.24381
CPU times: user 295 ms, sys: 7.97 ms, total: 303 ms
Wall time: 232 ms
```

```
---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
Cell In[51], line 3
      1 for attempt, init in enumerate(initLs, start=1):
      2     print(f"Attempt {attempt}: Starting value is {init} \n")
----> 3     get_ipython().run_line_magic('time', 'p = newton(lambda p: e(p, A,␣
 ↪b, c), init, tol=1e-15, max_iter=15)')
      4     print("-" * 64)

File <timed exec>:1
----> 1 'Could not get source, probably due dynamically evaluated source code.'

Cell In[34], line 17, in newton(f, x_0, tol, max_iter)
     13         if n > max_iter:
     14             raise Exception("Max iteration reached without convergence")
     15         y = q(x)
     16         if any(jnp.isnan(y)):
---> 17             raise Exception("Solution not found with NaN generated")
     18         error = jnp.linalg.norm(x - y)
     19         x = y
     20         print(f"iteration {n}, error = {error:.5f}")

Exception: Solution not found with NaN generated
```

We can see that Newton's method may fail for some starting values.

Sometimes it may take a few initial guesses to achieve convergence.

Substitute the result back to the formula to check our result using the second initial guess which converges

```
p_solution = newton(lambda p: e(p, A, b, c), initLs[1], tol=1e-15, max_iter=15)
e(p_solution, A, b, c)
```

```
iteration 1, error = 0.73419
iteration 2, error = 0.12472
iteration 3, error = 0.00269
iteration 4, error = 0.00000
iteration 5, error = 0.00000
iteration 6, error = 0.00000

Result = [1.49744442 1.49744442 1.49744442]
```

```
Array([0.00000000e+00, 0.00000000e+00, 2.22044605e-16], dtype=float64)
```

We can see the result is very accurate.

Part II

Elementary Statistics

CHAPTER

EIGHT

ELEMENTARY PROBABILITY WITH MATRICES

This lecture uses matrix algebra to illustrate some basic ideas about probability theory.

After introducing underlying objects, we'll use matrices and vectors to describe probability distributions.

Among concepts that we'll be studying include

- a joint probability distribution
- marginal distributions associated with a given joint distribution
- conditional probability distributions
- statistical independence of two random variables
- joint distributions associated with a prescribed set of marginal distributions
  - couplings
  - copulas
- the probability distribution of a sum of two independent random variables
  - convolution of marginal distributions
- parameters that define a probability distribution
- sufficient statistics as data summaries

We'll use a matrix to represent a bivariate or multivariate probability distribution and a vector to represent a univariate probability distribution

This *companion lecture* describes some popular probability distributions and describes how to use Python to sample from them.

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install prettytable
```

As usual, we'll start with some imports

```
import numpy as np
import matplotlib.pyplot as plt
import prettytable as pt
from scipy import stats
from scipy.special import comb
from mpl_toolkits.mplot3d import Axes3D
from matplotlib_inline.backend_inline import set_matplotlib_formats
set_matplotlib_formats('retina')

rng = np.random.default_rng(0)
```

8.1 Sketch of basic concepts

We'll briefly define what we mean by a **probability space**, a **probability measure**, and a **random variable**.

For most of this lecture, we sweep these objects into the background

> **Note**
>
> Nevertheless, they'll be lurking beneath **induced distributions** of random variables that we'll focus on here.
>
> These deeper objects are essential for defining and analysing the concepts of stationarity and ergodicity that underly laws of large numbers.
>
> For a relatively nontechnical presentation of some of these results see this chapter from Lars Peter Hansen and Thomas J. Sargent's online monograph titled *Risk, Uncertainty, and Values*.

Let Ω be a set of possible underlying outcomes and let $\omega \in \Omega$ be a particular underlying outcome.

Let $\mathcal{F}$ be a collection of subsets of Ω that we call **events**.

(Technically, $\mathcal{F}$ is a σ-algebra.)

A **probability measure** μ maps each event $\mathcal{G} \in \mathcal{F}$ into a scalar number $\mu(\mathcal{G})$ between 0 and 1, with $\mu(\Omega) = 1$.

The triple $\Omega, \mathcal{F}, \mu$ forms our **probability space**.

A **random variable** $X(\omega)$ is a function of the underlying outcome $\omega \in \Omega$ that assigns a value in some set of possible values.

If A is a set of possible values of X, then the event that X lies in A is

$$\mathcal{G} = \{\omega \in \Omega : X(\omega) \in A\}.$$

The random variable $X(\omega)$ has a **probability distribution** induced by the probability measure μ:

$$\text{Prob}(X \in A) = \mu(\mathcal{G}).$$

If μ has a density $p(\omega)$, then we can also write

$$\text{Prob}(X \in A) = \int_{\mathcal{G}} p(\omega)\, d\omega \tag{8.1}$$

We call this the induced probability distribution of random variable X.

Instead of working explicitly with an underlying probability space $\Omega, \mathcal{F}$ and probability measure μ, applied statisticians often proceed simply by specifying a form for an induced distribution for a random variable X.

That is how we'll proceed in this lecture and in many subsequent lectures.

8.2 What does probability mean?

Before diving in, we'll say a few words about what probability theory means and how it connects to statistics.

We also touch on these topics in *Two Meanings of Probability* and *Bayesian versus Frequentist Decision Rules*.

For much of this lecture we'll be discussing fixed "population" probabilities.

These are purely mathematical objects.

To appreciate how statisticians connect probabilities to data, the key is to understand the following concepts:

- A single draw from a probability distribution
- Repeated independently and identically distributed (i.i.d.) draws of "samples" or "realizations" from the same probability distribution
- A **statistic** defined as a function of a sequence of samples
- An **empirical distribution** or **histogram** (a binned empirical distribution) that records observed **relative frequencies**
- The idea that a population probability distribution is what we anticipate **relative frequencies** will be in a long sequence of i.i.d. draws. Here the following mathematical machinery makes precise what is meant by **anticipated relative frequencies**
  - **Law of Large Numbers (LLN)**
  - **Central Limit Theorem (CLT)**

8.2.1 A discrete random variable example

Scalar example

Let X be a scalar random variable that takes on the I possible values $0, 1, 2, \ldots, I-1$ with probabilities

$$\text{Prob}(X = i) = f_i,$$

where

$$f_i \geqslant 0, \quad \sum_i f_i = 1.$$

We sometimes write

$$X \sim \{f_i\}_{i=0}^{I-1}$$

as a short-hand way of saying that the random variable X is described by the probability distribution $\{f_i\}_{i=0}^{I-1}$.

Consider drawing a sample $x_0, x_1, \ldots, x_{N-1}$ of N independent and identically distributed draws of X.

What do "identical" and "independent" mean in IID or iid ("identically and independently distributed")?

- "identical" means that each draw is from the same distribution.
- "independent" means that the joint distribution equals the product of marginal distributions, i.e.,

$$\begin{aligned}\text{Prob}\{x_0 = i_0, x_1 = i_1, \ldots, x_{N-1} = i_{N-1}\} &= \text{Prob}\{x_0 = i_0\} \cdot \cdots \cdot \text{Prob}\{x_{N-1} = i_{N-1}\} \\ &= f_{i_0} f_{i_1} \cdot \cdots \cdot f_{i_{N-1}}\end{aligned}$$

We define an **empirical distribution** as follows.

For each $i = 0, \ldots, I-1$, let

$$\begin{aligned}N_i &= \text{number of times } X = i, \\ N &= \sum_{i=0}^{I-1} N_i \quad \text{total number of draws}, \\ \tilde{f}_i &= \frac{N_i}{N} \sim \text{ frequency of draws for which } X = i\end{aligned}$$

Key concepts that connect probability theory with statistics are laws of large numbers and central limit theorems.

A Law of Large Numbers (LLN) states that $\tilde{f}_i \to f_i$ as $N \to \infty$.

A Central Limit Theorem (CLT) describes a **rate** at which $\tilde{f}_i \to f_i$.

See *LLN and CLT* for a detailed treatment of both results.

8.2.2 Understanding probability: frequentist vs. Bayesian

For "frequentist" statisticians, **anticipated relative frequency** is **all** that a probability distribution means.

But for a Bayesian it means something else – something partly subjective and purely personal.

We say "partly" because a Bayesian also pays attention to relative frequencies.

8.3 Representing probability distributions

A probability distribution $\text{Prob}(X \in A)$ can be described by its **cumulative distribution function (CDF)**

$$F_X(x) = \text{Prob}\{X \leq x\}.$$

Sometimes, but not always, a random variable can also be described by a **density function** $f(x)$ that is related to its CDF by

$$\text{Prob}\{X \in B\} = \int_{t \in B} f(t)dt$$

$$F(x) = \int_{-\infty}^{x} f(t)dt$$

Here B is a set of possible X's whose probability of occurring we want to compute.

When a probability density exists, a probability distribution can be characterized either by its CDF or by its density.

For a **discrete-valued** random variable

- the number of possible values of X is finite or countably infinite
- we replace a **density** with a **probability mass function**, a non-negative sequence that sums to one
- when a density exists, we replace integration with summation in formulas like (8.1)

In this lecture, we mostly discuss discrete random variables.

Doing this enables us to confine our tool set basically to linear algebra.

Later we'll briefly discuss how to approximate a continuous random variable with a discrete random variable.

8.4 Univariate probability distributions

We'll devote most of this lecture to discrete-valued random variables, but we'll say a few things about continuous-valued random variables.

8.4.1 Discrete random variable

Let X be a discrete random variable that takes possible values: $i = 0, 1, \ldots, I-1 = \bar{X}$.

Here, we choose the maximum index $I-1$ because of how this aligns nicely with Python's index convention.

Define $f_i \equiv \text{Prob}\{X = i\}$ and assemble the non-negative vector

$$f = \begin{bmatrix} f_0 \\ f_1 \\ \vdots \\ f_{I-1} \end{bmatrix} \tag{8.2}$$

for which $f_i \in [0,1]$ for each i and $\sum_{i=0}^{I-1} f_i = 1$.

This vector defines a **probability mass function**.

The distribution (8.2) has **parameters** $\{f_i\}_{i=0,1,\cdots,I-2}$ since $f_{I-1} = 1 - \sum_{i=0}^{I-2} f_i$.

These parameters pin down the shape of the distribution.

(Sometimes $I = \infty$.)

Such a "non-parametric" distribution has as many "parameters" as there are possible values of the random variable.

We often work with special distributions that are characterized by a small number parameters.

In these special parametric distributions,

$$f_i = g(i;\theta)$$

where θ is a vector of parameters that is of much smaller dimension than I.

A **statistical model** is a joint probability distribution characterized by a list of **parameters**.

The concept of **parameter** is intimately related to the notion of **sufficient statistic**.

A **statistic** is a nonlinear function of a data set.

Sufficient statistics summarize all **information** that a data set contains about parameters of a statistical model.

Note that a sufficient statistic corresponds to a particular statistical model.

Sufficient statistics are key tools that AI uses to summarize or compress a **big data** set.

R. A. Fisher provided a rigorous definition of **information** – see Fisher information.

An example of a parametric probability distribution is a **geometric distribution**.

It is described by

$$f_i = \text{Prob}\{X = i\} = (1-\lambda)\lambda^i, \quad \lambda \in [0,1), \quad i = 0,1,2,...$$

Evidently, $\sum_{i=0}^{\infty} f_i = 1$.

Let θ be a vector of parameters of the distribution described by f, then

$$f_i(\theta) \geq 0, \sum_{i=0}^{\infty} f_i(\theta) = 1$$

8.4.2 Continuous random variable

Let X be a continuous random variable that takes values in a set $\tilde{X} \subseteq \mathbb{R}$ and whose distribution has parameters θ.

$$\text{Prob}\{X \in A\} = \int_{x \in A} f(x;\theta)\, dx; \quad f(x;\theta) \geq 0$$

where A is a subset of $\tilde{X}$ and

$$\text{Prob}\{X \in \tilde{X}\} = 1$$

8.5 Bivariate probability distributions

We'll now discuss a bivariate **joint distribution**.

To begin, we restrict ourselves to two discrete random variables.

Let X, Y be two discrete random variables that take values:

$$X \in \{0, \ldots, I-1\}$$

$$Y \in \{0, \ldots, J-1\}$$

Then their **joint distribution** is described by a matrix

$$F_{I \times J} = [f_{ij}]_{i \in \{0,\ldots,I-1\}, j \in \{0,\ldots,J-1\}}$$

whose elements are

$$f_{ij} = \text{Prob}\{X = i, Y = j\} \geq 0$$

where

$$\sum_i \sum_j f_{ij} = 1$$

8.6 Marginal probability distributions

The joint distribution induces marginal distributions

$$\text{Prob}\{X = i\} = \sum_{j=0}^{J-1} f_{ij} = \mu_i, \quad i = 0, \ldots, I-1$$

$$\text{Prob}\{Y = j\} = \sum_{i=0}^{I-1} f_{ij} = \nu_j, \quad j = 0, \ldots, J-1$$

For example, let a joint distribution over (X, Y) be

$$F = \begin{bmatrix} .25 & .1 \\ .15 & .5 \end{bmatrix} \tag{8.3}$$

The implied marginal distributions are:

$$\begin{aligned} \text{Prob}\{X = 0\} &= .25 + .1 = .35 \\ \text{Prob}\{X = 1\} &= .15 + .5 = .65 \\ \text{Prob}\{Y = 0\} &= .25 + .15 = .4 \\ \text{Prob}\{Y = 1\} &= .1 + .5 = .6 \end{aligned}$$

As a digression, if two random variables X, Y are continuous and have joint density $f(x, y)$, then marginal distributions can be computed by

$$f(x) = \int_{\mathbb{R}} f(x, y) dy$$

$$f(y) = \int_{\mathbb{R}} f(x, y) dx$$

8.7 Conditional probability distributions

Conditional probabilities are defined according to

$$\text{Prob}\{A \mid B\} = \frac{\text{Prob}\{A \cap B\}}{\text{Prob}\{B\}}$$

where A, B are two events.

For a pair of discrete random variables, we have the **conditional distribution**

$$\text{Prob}\{X = i | Y = j\} = \frac{f_{ij}}{\sum_i f_{ij}} = \frac{\text{Prob}\{X = i, Y = j\}}{\text{Prob}\{Y = j\}}$$

where $i = 0, \ldots, I-1, \quad j = 0, \ldots, J-1.$

Note that

$$\sum_i \text{Prob}\{X = i | Y = j\} = \frac{\sum_i f_{ij}}{\sum_i f_{ij}} = 1$$

The mathematics of conditional probability implies:

$$\text{Prob}\{X = i | Y = j\} = \frac{\text{Prob}\{X = i, Y = j\}}{\text{Prob}\{Y = j\}} = \frac{\text{Prob}\{Y = j | X = i\}\text{Prob}\{X = i\}}{\text{Prob}\{Y = j\}} \tag{8.4}$$

Note

Formula (8.4) is also what a Bayesian calls **Bayes' Law**.

A Bayesian statistician regards marginal probability distribution $\text{Prob}(X = i), i = 0, \ldots, I-1$ as a **prior** distribution that describes his personal subjective beliefs about X.

He then interprets formula (8.4) as a procedure for constructing a **posterior** distribution that describes how he would revise his subjective beliefs after observing that Y equals j.

For the joint distribution (8.3)

$$\text{Prob}\{X = 0 | Y = 1\} = \frac{.1}{.1 + .5} = \frac{.1}{.6}$$

8.8 Transition probability matrix

Consider the following joint probability distribution of two random variables.

Let X, Y be discrete random variables with joint distribution

$$\text{Prob}\{X = i, Y = j\} = \rho_{ij}$$

where $i = 0, \ldots, I-1; j = 0, \ldots, J-1$ and

$$\sum_i \sum_j \rho_{ij} = 1, \quad \rho_{ij} \geqslant 0.$$

An associated conditional distribution is

$$\text{Prob}\{Y = j | X = i\} = \frac{\rho_{ij}}{\sum_j \rho_{ij}} = \frac{\text{Prob}\{Y = j, X = i\}}{\text{Prob}\{X = i\}}$$

We can define a transition probability matrix P with i, j component

$$p_{ij} = \text{Prob}\{Y = j | X = i\} = \frac{\rho_{ij}}{\sum_j \rho_{ij}}$$

where

$$\begin{bmatrix} p_{00} & p_{01} \\ p_{10} & p_{11} \end{bmatrix}$$

The first row is the probability that $Y = j, j = 0, 1$ conditional on $X = 0$.

The second row is the probability that $Y = j, j = 0, 1$ conditional on $X = 1$.

Note that

- $\sum_j p_{ij} = \frac{\sum_j \rho_{ij}}{\sum_j \rho_{ij}} = 1$, so each row of the transition matrix P is a probability distribution (not so for each column).

8.9 Application: forecasting a time series

Suppose that there are two time periods.

- $t = 0$ "today"
- $t = 1$ "tomorrow"

Let $X(0)$ be a random variable to be realized at $t = 0$, $X(1)$ be a random variable to be realized at $t = 1$.

Suppose that

$$\text{Prob}\{X(0) = i, X(1) = j\} = f_{ij} \geq 0, \quad i = 0, \cdots, I-1, \quad j = 0, \cdots, J-1$$
$$\sum_i \sum_j f_{ij} = 1$$

f_{ij} is a joint distribution over $[X(0), X(1)]$.

A conditional distribution is

$$\text{Prob}\{X(1) = j | X(0) = i\} = \frac{f_{ij}}{\sum_j f_{ij}}$$

This formula is a workhorse for applied economic forecasters.

8.10 Statistical independence

Random variables X and Y are statistically **independent** if

$$\text{Prob}\{X = i, Y = j\} = f_i g_j$$

where

$$\text{Prob}\{X = i\} = f_i \geq 0, \quad \sum_i f_i = 1$$
$$\text{Prob}\{Y = j\} = g_j \geq 0, \quad \sum_j g_j = 1$$

Conditional distributions are

$$\text{Prob}\{X = i|Y = j\} = \frac{f_i g_j}{\sum_i f_i g_j} = \frac{f_i g_j}{g_j} = f_i$$
$$\text{Prob}\{Y = j|X = i\} = \frac{f_i g_j}{\sum_j f_i g_j} = \frac{f_i g_j}{f_i} = g_j$$

8.11 Means and variances

The mean and variance of a discrete random variable X are

$$\mu_X \equiv \mathbb{E}[X] = \sum_k k\text{Prob}\{X = k\}$$
$$\sigma_X^2 \equiv \mathbb{D}[X] = \sum_k (k - \mathbb{E}[X])^2 \text{Prob}\{X = k\}$$

A continuous random variable having density $f_X(x)$ has mean and variance

$$\mu_X \equiv \mathbb{E}[X] = \int_{-\infty}^{\infty} x f_X(x) dx$$
$$\sigma_X^2 \equiv \mathbb{D}[X] = \text{E}\left[(X - \mu_X)^2\right] = \int_{-\infty}^{\infty} (x - \mu_X)^2 f_X(x) dx$$

8.12 Matrix representations of some bivariate distributions

Let's use matrices to represent a joint distribution, conditional distribution, marginal distribution, and the mean and variance of a bivariate random variable.

The table below illustrates a probability distribution for a bivariate random variable.

$$F = [f_{ij}] = \begin{bmatrix} 0.3 & 0.2 \\ 0.1 & 0.4 \end{bmatrix}$$

Marginal distributions are

$$\text{Prob}(X = i) = \sum_j f_{ij} = u_i$$

$$\text{Prob}(Y = j) = \sum_i f_{ij} = v_j$$

Let's write some Python code that lets us draw some long samples and compute relative frequencies.

The code lets us check whether the "sampling" distribution agrees with the "population" distribution – confirming that the population distribution correctly tells us the relative frequencies that we should expect in a large sample.

```
# specify parameters
xs = np.array([0, 1])
ys = np.array([10, 20])
f = np.array([[0.3, 0.2], [0.1, 0.4]])
f_cum = np.cumsum(f)

# draw random numbers
```

(continues on next page)

(continued from previous page)

```
p = rng.random(1_000_000)
x = np.vstack([xs[1]*np.ones(p.shape), ys[1]*np.ones(p.shape)])
# map to the bivariate distribution

x[0, p < f_cum[2]] = xs[1]
x[1, p < f_cum[2]] = ys[0]

x[0, p < f_cum[1]] = xs[0]
x[1, p < f_cum[1]] = ys[1]

x[0, p < f_cum[0]] = xs[0]
x[1, p < f_cum[0]] = ys[0]
print(x)
```

```
[[ 1.  0.  0. ...  0.  0.  0.]
 [20. 10. 10. ... 10. 10. 20.]]
```

Note

To generate random draws from the joint distribution F, we use the inverse CDF technique described in *this companion lecture*.

```
# marginal distribution
xp = np.sum(x[0, :] == xs[0])/1_000_000
yp = np.sum(x[1, :] == ys[0])/1_000_000

# print output
print("marginal distribution for x")
xmtb = pt.PrettyTable()
xmtb.field_names = ['x_value', 'x_prob']
xmtb.add_row([xs[0], xp])
xmtb.add_row([xs[1], 1-xp])
print(xmtb)

print("\nmarginal distribution for y")
ymtb = pt.PrettyTable()
ymtb.field_names = ['y_value', 'y_prob']
ymtb.add_row([ys[0], yp])
ymtb.add_row([ys[1], 1-yp])
print(ymtb)
```

```
marginal distribution for x
+---------+----------+
| x_value |  x_prob  |
+---------+----------+
|    0    | 0.500194 |
|    1    | 0.499806 |
+---------+----------+

marginal distribution for y
+---------+----------+
| y_value |  y_prob  |
+---------+----------+
|    10   | 0.399741 |
```

(continues on next page)

(continued from previous page)

```
|    20   | 0.600259 |
+---------+----------+
```

```
# conditional distributions
xc1 = x[0, x[1, :] == ys[0]]
xc2 = x[0, x[1, :] == ys[1]]
yc1 = x[1, x[0, :] == xs[0]]
yc2 = x[1, x[0, :] == xs[1]]

xc1p = np.sum(xc1 == xs[0])/len(xc1)
xc2p = np.sum(xc2 == xs[0])/len(xc2)
yc1p = np.sum(yc1 == ys[0])/len(yc1)
yc2p = np.sum(yc2 == ys[0])/len(yc2)

# print output
print("conditional distribution for x")
xctb = pt.PrettyTable()
xctb.field_names = ['y_value', 'prob(x=0)', 'prob(x=1)']
xctb.add_row([ys[0], xc1p, 1-xc1p])
xctb.add_row([ys[1], xc2p, 1-xc2p])
print(xctb)

print("\nconditional distribution for y")
yctb = pt.PrettyTable()
yctb.field_names = ['x_value',  'prob(y=10)', 'prob(y=20)']
yctb.add_row([xs[0], yc1p, 1-yc1p])
yctb.add_row([xs[1], yc2p, 1-yc2p])
print(yctb)
```

```
conditional distribution for x
+---------+--------------------+---------------------+
| y_value |     prob(x=0)      |      prob(x=1)      |
+---------+--------------------+---------------------+
|    10   | 0.7504634250677313 | 0.24953657493226866 |
|    20   | 0.333527693878809  |  0.666472306121191  |
+---------+--------------------+---------------------+

conditional distribution for y
+---------+--------------------+---------------------+
| x_value |      prob(y=10)    |      prob(y=20)     |
+---------+--------------------+---------------------+
|    0    |  0.5997492972726582 | 0.40025070272734176 |
|    1    | 0.19957743604518552 |  0.8004225639548145 |
+---------+--------------------+---------------------+
```

Let's calculate population marginal and conditional probabilities using matrix algebra.

$$
\begin{bmatrix}
 & \vdots & y_1 & y_2 & \vdots & x \\
\cdots & \vdots & \cdots & \cdots & \vdots & \cdots \\
x_1 & \vdots & 0.3 & 0.2 & \vdots & 0.5 \\
x_2 & \vdots & 0.1 & 0.4 & \vdots & 0.5 \\
\cdots & \vdots & \cdots & \cdots & \vdots & \cdots \\
y & \vdots & 0.4 & 0.6 & \vdots & 1
\end{bmatrix}
$$

$\Longrightarrow$

(1) Marginal distribution:

$$\begin{bmatrix} var & \vdots & var_1 & var_2 \\ \cdots & \vdots & \cdots & \cdots \\ x & \vdots & 0.5 & 0.5 \\ \cdots & \vdots & \cdots & \cdots \\ y & \vdots & 0.4 & 0.6 \end{bmatrix}$$

(2) Conditional distribution:

$$\begin{bmatrix} x & \vdots & x_1 & x_2 \\ \cdots\cdots & \vdots & \cdots\cdots & \cdots\cdots \\ y = y_1 & \vdots & \frac{0.3}{0.4} = 0.75 & \frac{0.1}{0.4} = 0.25 \\ \cdots\cdots & \vdots & \cdots\cdots & \cdots\cdots \\ y = y_2 & \vdots & \frac{0.2}{0.6} \approx 0.33 & \frac{0.4}{0.6} \approx 0.67 \end{bmatrix}$$

$$\begin{bmatrix} y & \vdots & y_1 & y_2 \\ \cdots\cdots & \vdots & \cdots\cdots & \cdots\cdots \\ x = x_1 & \vdots & \frac{0.3}{0.5} = 0.6 & \frac{0.2}{0.5} = 0.4 \\ \cdots\cdots & \vdots & \cdots\cdots & \cdots\cdots \\ x = x_2 & \vdots & \frac{0.1}{0.5} = 0.2 & \frac{0.4}{0.5} = 0.8 \end{bmatrix}$$

These population objects closely resemble the sample counterparts computed above.

Let's wrap some of the functions we have used in a Python class that will let us generate and sample from a discrete bivariate joint distribution.

```
class discrete_bijoint:

    def __init__(self, f, xs, ys):
        '''initialization
        -----------------
        parameters:
        f: the bivariate joint probability matrix
        xs: values of x vector
        ys: values of y vector
        '''
        self.f, self.xs, self.ys = f, xs, ys

    def joint_tb(self):
        '''print the joint distribution table'''
        xs = self.xs
        ys = self.ys
        f = self.f
        jtb = pt.PrettyTable()
        jtb.field_names = ['x_value/y_value', *ys, 'marginal sum for x']
        for i in range(len(xs)):
            jtb.add_row([xs[i], *f[i, :], np.sum(f[i, :])])
        jtb.add_row(['marginal_sum for y', *np.sum(f, 0), np.sum(f)])
        print("\nThe joint probability distribution for x and y\n", jtb)
        self.jtb = jtb

    def draw(self, n):
        '''draw random numbers
        ----------------------
        parameters:
        n: number of random numbers to draw
        '''
```

(continues on next page)

(continued from previous page)

```
        xs = self.xs
        ys = self.ys
        f_cum = np.cumsum(self.f)
        p = rng.random(n)
        x = np.empty([2, p.shape[0]])
        lf = len(f_cum)
        lx = len(xs)-1
        ly = len(ys)-1
        for i in range(lf):
            x[0, p < f_cum[lf-1-i]] = xs[lx]
            x[1, p < f_cum[lf-1-i]] = ys[ly]
            if ly == 0:
                lx -= 1
                ly = len(ys)-1
            else:
                ly -= 1
        self.x = x
        self.n = n

    def marg_dist(self):
        '''marginal distribution'''
        x = self.x
        xs = self.xs
        ys = self.ys
        n = self.n
        xmp = [np.sum(x[0, :] == xs[i])/n for i in range(len(xs))]
        ymp = [np.sum(x[1, :] == ys[i])/n for i in range(len(ys))]

        # print output
        xmtb = pt.PrettyTable()
        ymtb = pt.PrettyTable()
        xmtb.field_names = ['x_value', 'x_prob']
        ymtb.field_names = ['y_value', 'y_prob']
        for i in range(max(len(xs), len(ys))):
            if i < len(xs):
                xmtb.add_row([xs[i], xmp[i]])
            if i < len(ys):
                ymtb.add_row([ys[i], ymp[i]])
        xmtb.add_row(['sum', np.sum(xmp)])
        ymtb.add_row(['sum', np.sum(ymp)])
        print("\nmarginal distribution for x\n", xmtb)
        print("\nmarginal distribution for y\n", ymtb)

        self.xmp = xmp
        self.ymp = ymp

    def cond_dist(self):
        '''conditional distribution'''
        x = self.x
        xs = self.xs
        ys = self.ys
        n = self.n
        xcp = np.empty([len(ys), len(xs)])
        ycp = np.empty([len(xs), len(ys)])
        for i in range(max(len(ys), len(xs))):
            if i < len(ys):
                xi = x[0, x[1, :] == ys[i]]
```

(continues on next page)

(continued from previous page)

```
                idx = xi.reshape(len(xi), 1) == xs.reshape(1, len(xs))
                xcp[i, :] = np.sum(idx, 0)/len(xi)
            if i < len(xs):
                yi = x[1, x[0, :] == xs[i]]
                idy = yi.reshape(len(yi), 1) == ys.reshape(1, len(ys))
                ycp[i, :] = np.sum(idy, 0)/len(yi)

        # print output
        xctb = pt.PrettyTable()
        yctb = pt.PrettyTable()
        xctb.field_names = ['x_value', *xs, 'sum']
        yctb.field_names = ['y_value', *ys, 'sum']
        for i in range(max(len(xs), len(ys))):
            if i < len(ys):
                xctb.add_row([ys[i], *xcp[i], np.sum(xcp[i])])
            if i < len(xs):
                yctb.add_row([xs[i], *ycp[i], np.sum(ycp[i])])
        print("\nconditional distribution for x\n", xctb)
        print("\nconditional distribution for y\n", yctb)

        self.xcp = xcp
        self.xyp = ycp
```

Let's apply our code to some examples.

8.12.1 Numerical examples

Example 1

```
# joint
d = discrete_bijoint(f, xs, ys)
d.joint_tb()
```

```
The joint probability distribution for x and y
 +--------------------+-----+--------------------+--------------------+
|  x_value/y_value   |  10 |         20         | marginal sum for x |
+--------------------+-----+--------------------+--------------------+
|         0          | 0.3 |        0.2         |        0.5         |
|         1          | 0.1 |        0.4         |        0.5         |
| marginal_sum for y | 0.4 | 0.6000000000000001 |        1.0         |
+--------------------+-----+--------------------+--------------------+
```

```
# sample marginal
d.draw(1_000_000)
d.marg_dist()
```

```
marginal distribution for x
 +---------+----------+
| x_value |  x_prob  |
+---------+----------+
|    0    | 0.499395 |
|    1    | 0.500605 |
|   sum   |   1.0    |
```

(continues on next page)

(continued from previous page)

```
+---------+----------+

marginal distribution for y
 +---------+----------+
| y_value |  y_prob  |
+---------+----------+
|    10   | 0.400501 |
|    20   | 0.599499 |
|   sum   |   1.0    |
+---------+----------+
```

```
# sample conditional
d.cond_dist()
```

```
conditional distribution for x
 +---------+--------------------+---------------------+-----+
| x_value |         0          |          1          | sum |
+---------+--------------------+---------------------+-----+
|    10   | 0.7491766562380618 | 0.25082334376193816 | 1.0 |
|    20   | 0.3325259925371018 |  0.6674740074628982 | 1.0 |
+---------+--------------------+---------------------+-----+

conditional distribution for y
 +---------+---------------------+--------------------+-----+
| y_value |          10         |         20         | sum |
+---------+---------------------+--------------------+-----+
|    0    |  0.6008189909790846 | 0.3991810090209153 | 1.0 |
|    1    | 0.20066719269683683 | 0.7993328073031631 | 1.0 |
+---------+---------------------+--------------------+-----+
```

Example 2

```
xs_new = np.array([10, 20, 30])
ys_new = np.array([1, 2])
f_new = np.array([[0.2, 0.1], [0.1, 0.3], [0.15, 0.15]])
d_new = discrete_bijoint(f_new, xs_new, ys_new)
d_new.joint_tb()
```

```
The joint probability distribution for x and y
 +--------------------+---------------------+------+---------------------+
|  x_value/y_value   |          1          |  2   |  marginal sum for x |
+--------------------+---------------------+------+---------------------+
|         10         |         0.2         | 0.1  | 0.30000000000000004 |
|         20         |         0.1         | 0.3  |         0.4         |
|         30         |         0.15        | 0.15 |         0.3         |
| marginal_sum for y | 0.45000000000000007 | 0.55 |         1.0         |
+--------------------+---------------------+------+---------------------+
```

```
d_new.draw(1_000_000)
d_new.marg_dist()
```

```
marginal distribution for x
 +---------+----------+
```

(continues on next page)

(continued from previous page)

```
| x_value |  x_prob  |
+---------+----------+
|    10   | 0.299848 |
|    20   | 0.401015 |
|    30   | 0.299137 |
|   sum   |   1.0    |
+---------+----------+

marginal distribution for y
 +---------+----------+
| y_value |  y_prob  |
+---------+----------+
|    1    | 0.449195 |
|    2    | 0.550805 |
|   sum   |   1.0    |
+---------+----------+
```

```
d_new.cond_dist()
```

```
conditional distribution for x
 +---------+---------------------+---------------------+--------------------+-----+
| x_value |          10         |          20         |         30         | sum |
+---------+---------------------+---------------------+--------------------+-----+
|    1    |  0.4445886530348735 | 0.22256258417836353 | 0.332848762786763  | 1.0 |
|    2    | 0.18180844400468404 |  0.5465473261862184 | 0.2716442298090976 | 1.0 |
+---------+---------------------+---------------------+--------------------+-----+

conditional distribution for y
 +---------+---------------------+--------------------+-----+
| y_value |          1          |         2          | sum |
+---------+---------------------+--------------------+-----+
|    10   |  0.6660274539099811 | 0.3339725460900189 | 1.0 |
|    20   | 0.24930239517225042 | 0.7506976048277496 | 1.0 |
|    30   |  0.4998178092312218 | 0.5001821907687782 | 1.0 |
+---------+---------------------+--------------------+-----+
```

8.13 A continuous bivariate random vector

A two-dimensional Gaussian distribution has joint density

$$f(x,y) = (2\pi\sigma_1\sigma_2\sqrt{1-\rho^2})^{-1} \exp\left[-\frac{1}{2(1-\rho^2)}\left(\frac{(x-\mu_1)^2}{\sigma_1^2} - \frac{2\rho(x-\mu_1)(y-\mu_2)}{\sigma_1\sigma_2} + \frac{(y-\mu_2)^2}{\sigma_2^2}\right)\right]$$

We start with a bivariate normal distribution pinned down by

$$\mu = \begin{bmatrix} 0 \\ 5 \end{bmatrix}, \quad \Sigma = \begin{bmatrix} 5 & .2 \\ .2 & 1 \end{bmatrix}$$

```
# define the joint probability density function
def func(x, y, μ1=0, μ2=5, σ1=np.sqrt(5), σ2=np.sqrt(1), ρ=.2/np.sqrt(5*1)):
    A = (2 * np.pi * σ1 * σ2 * np.sqrt(1 - ρ**2))**(-1)
    B = -1 / 2 / (1 - ρ**2)
    C1 = (x - μ1)**2 / σ1**2
```

(continues on next page)

(continued from previous page)

```
    C2 = 2 * ρ * (x - μ1) * (y - μ2) / σ1 / σ2
    C3 = (y - μ2)**2 / σ2**2
    return A * np.exp(B * (C1 - C2 + C3))
```

```
μ1 = 0
μ2 = 5
σ1 = np.sqrt(5)
σ2 = np.sqrt(1)
ρ = .2 / np.sqrt(5 * 1)
```

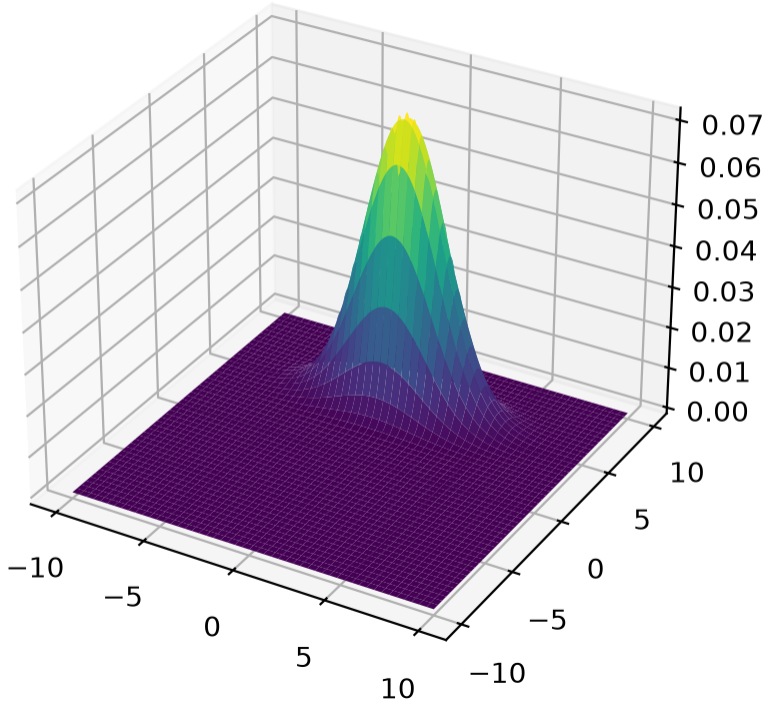
```
x = np.linspace(-10, 10, 1_000)
y = np.linspace(-10, 10, 1_000)
x_mesh, y_mesh = np.meshgrid(x, y, indexing="ij")
```

8.13.1 Joint, marginal, and conditional distributions

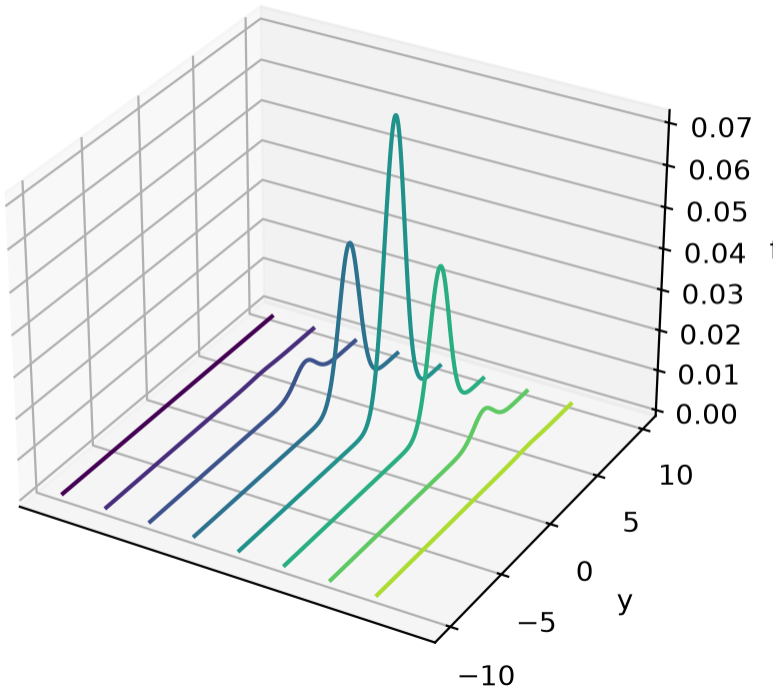
Joint distribution

Let's plot the **population** joint density.

```
# %matplotlib notebook

fig = plt.figure()
ax = plt.axes(projection='3d')

surf = ax.plot_surface(x_mesh, y_mesh, func(x_mesh, y_mesh), cmap='viridis')
plt.show()
```




```
# %matplotlib notebook

fig = plt.figure()
ax = plt.axes(projection='3d')

curve = ax.contour(x_mesh, y_mesh, func(x_mesh, y_mesh), zdir='x')
plt.ylabel('y')
ax.set_zlabel('f')
ax.set_xticks([])
plt.show()
```



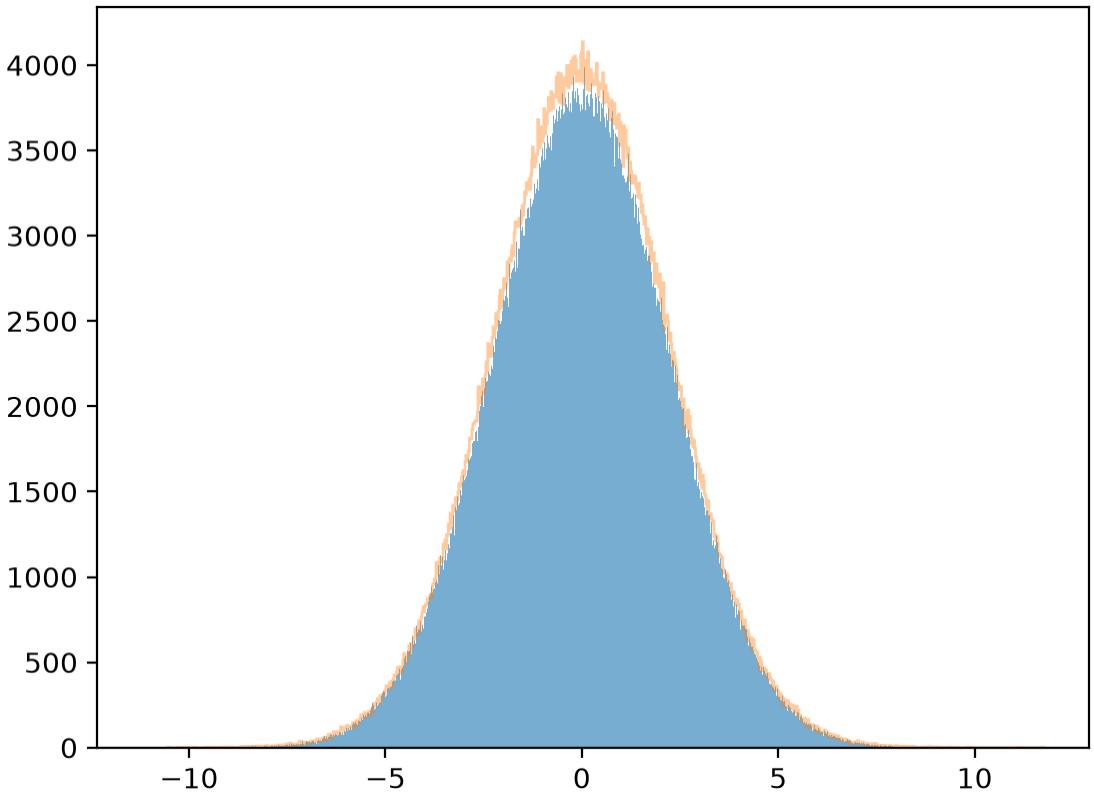
Next we can use a built-in `numpy` function to draw random samples, then calculate a **sample** marginal distribution from the sample mean and variance.

```
μ= np.array([0, 5])
σ= np.array([[5, .2], [.2, 1]])
n = 1_000_000
data = rng.multivariate_normal(μ, σ, n)
x = data[:, 0]
y = data[:, 1]
```

Marginal distribution

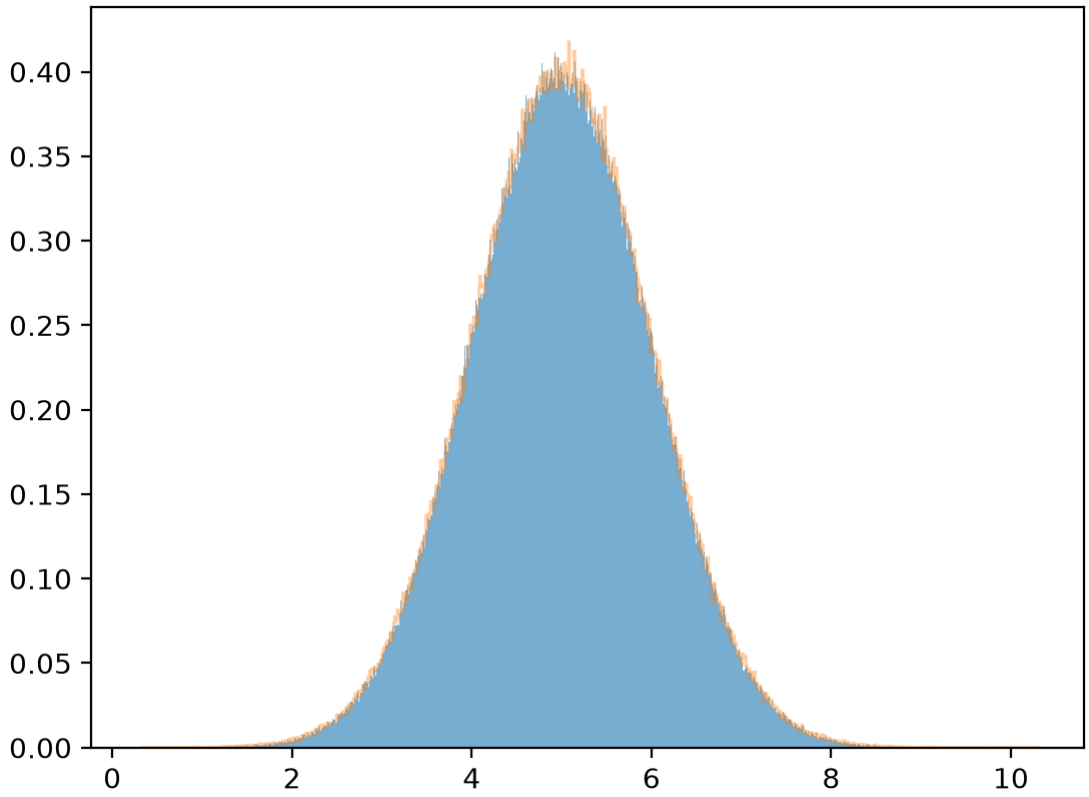
```
plt.hist(x, bins=1_000, alpha=0.6)
μx_hat, σx_hat = np.mean(x), np.std(x)
print(μx_hat, σx_hat)
x_sim = rng.normal(μx_hat, σx_hat, 1_000_000)
plt.hist(x_sim, bins=1_000, alpha=0.4, histtype="step")
plt.show()
```

```
-0.002835018476282638 2.237281153907461
```




```
plt.hist(y, bins=1_000, density=True, alpha=0.6)
μy_hat, σy_hat = np.mean(y), np.std(y)
print(μy_hat, σy_hat)
y_sim = rng.normal(μy_hat, σy_hat, 1_000_000)
plt.hist(y_sim, bins=1_000, density=True, alpha=0.4, histtype="step")
plt.show()
```

```
4.998242637632656 0.9993565018250069
```



Conditional distribution

For a bivariate normal population distribution, the conditional distributions are also normal:

$$X \mid Y = y \sim \mathbb{N}\left[\mu_X + \rho\sigma_X \frac{y - \mu_Y}{\sigma_Y}, \sigma_X^2(1-\rho^2)\right]$$
$$Y \mid X = x \sim \mathbb{N}\left[\mu_Y + \rho\sigma_Y \frac{x - \mu_X}{\sigma_X}, \sigma_Y^2(1-\rho^2)\right]$$

Note

Please see this *quantecon lecture* for more details.

Let's approximate the joint density by discretizing and mapping the approximating joint density into a matrix.

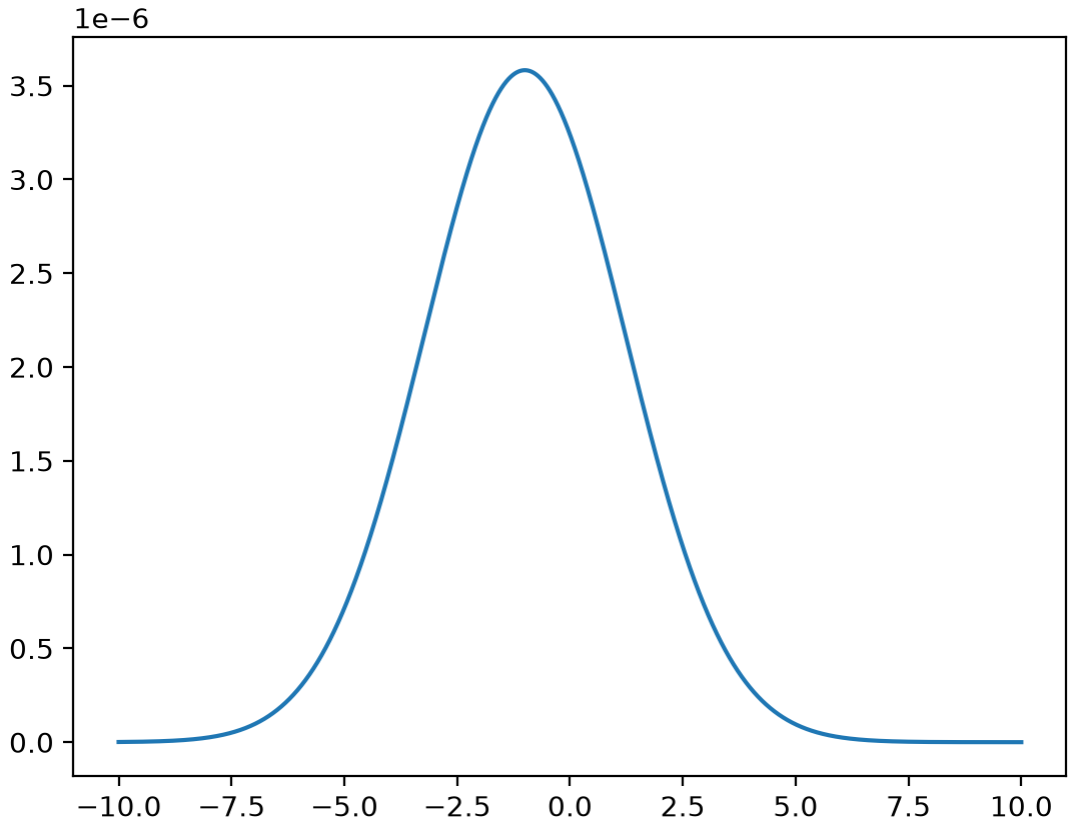
On an evenly spaced grid, we can approximate the conditional distribution by assigning probability weights proportional to a slice of the joint density.

For fixed y, this means that

$$z_i \equiv \frac{f(x_i, y)}{\sum_k f(x_k, y)}$$

Fix $y = 0$.

```
# discretized conditional distribution of X given Y = 0
x = np.linspace(-10, 10, 1_000_000)
z = func(x, y=0) / np.sum(func(x, y=0))
plt.plot(x, z)
plt.show()
```



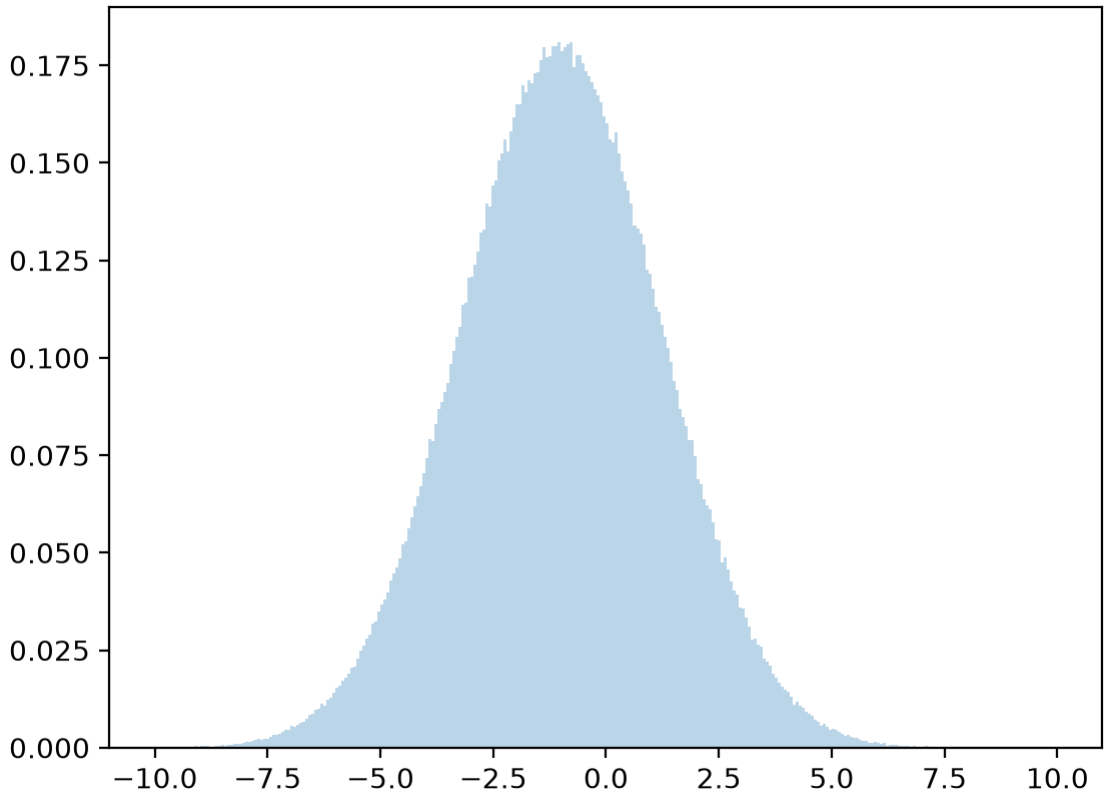
The conditional mean and variance are then approximated by

$$\mathbb{E}\left[X|Y=y\right] \approx \sum_i x_i z_i$$

$$\mathbb{D}\left[X|Y=y\right] \approx \sum_i \left(x_i - \mu_{X|Y=y}\right)^2 z_i$$

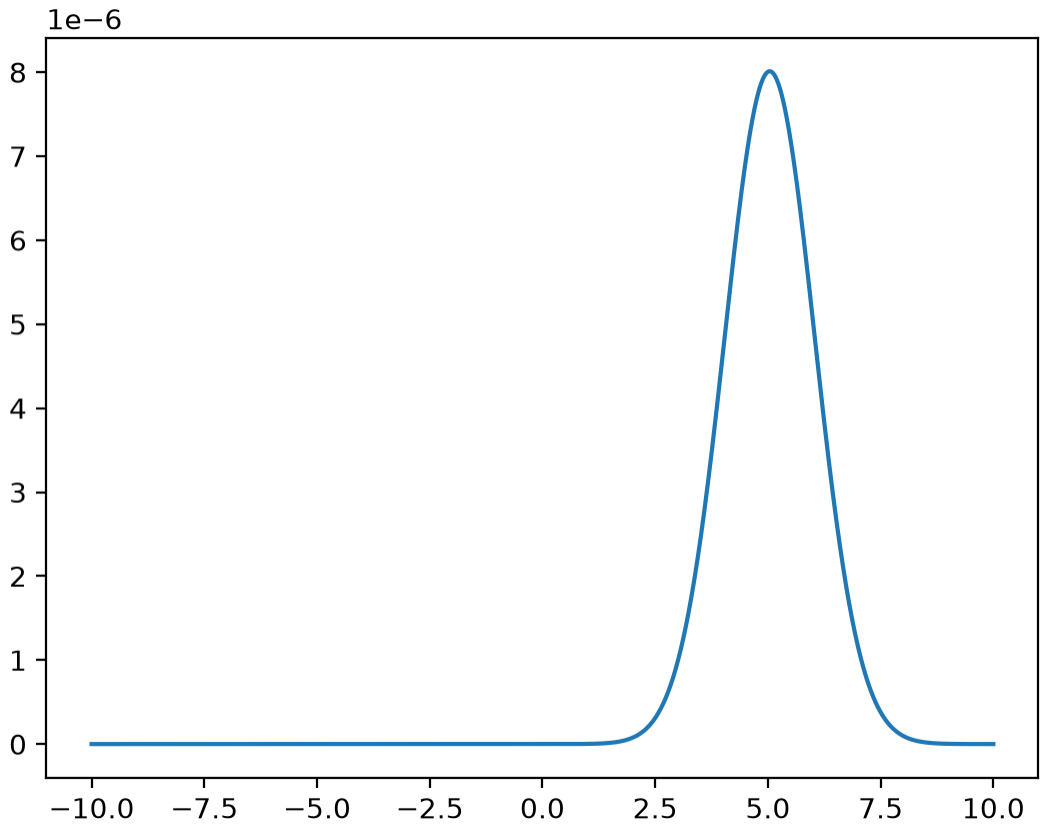
Let's draw from a normal distribution with above mean and variance and check how accurate our approximation is.

```
# discretized mean
μx = np.dot(x, z)

# discretized standard deviation
σx = np.sqrt(np.dot((x - μx)**2, z))

# sample
zz = rng.normal(μx, σx, 1_000_000)
plt.hist(zz, bins=300, density=True, alpha=0.3, range=[-10, 10])
plt.show()
```

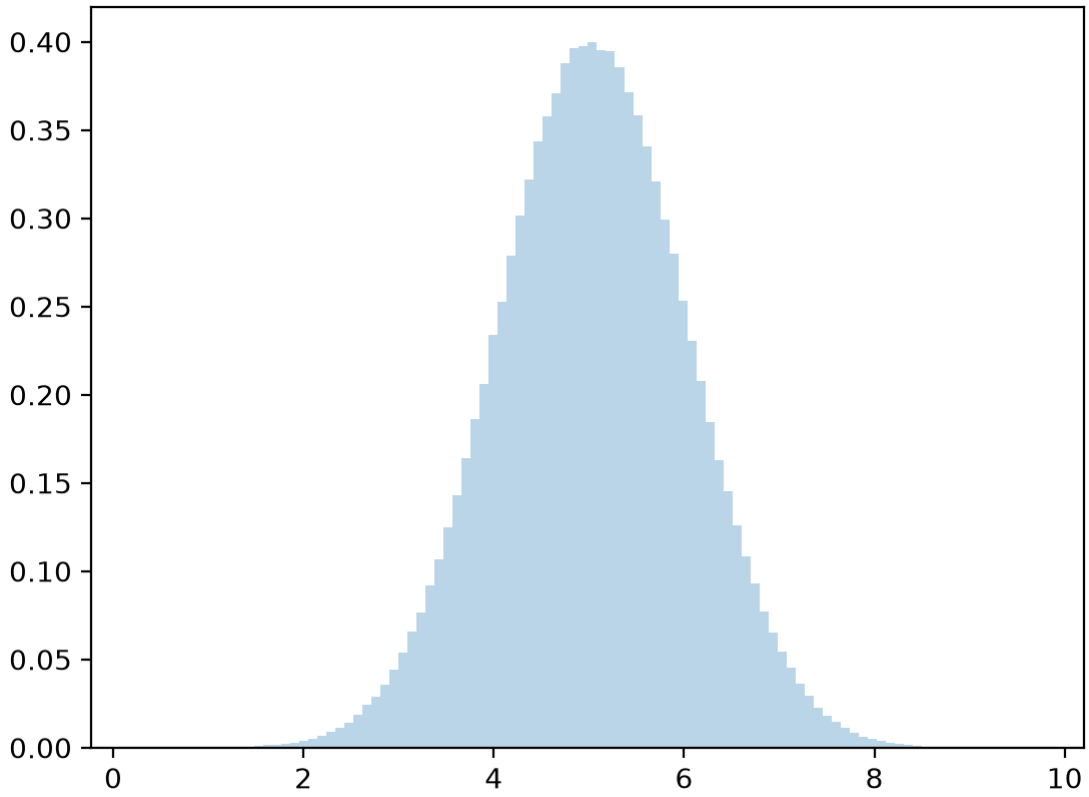


Fix $x = 1$.

```
y = np.linspace(-10, 10, 1_000_000)
z = func(x=1, y=y) / np.sum(func(x=1, y=y))
plt.plot(y,z)
plt.show()
```




```
# discretized conditional mean and standard deviation
μy = np.dot(y,z)
σy = np.sqrt(np.dot((y - μy)**2, z))

# sample
zz = rng.normal(μy, σy, 1_000_000)
plt.hist(zz, bins=100, density=True, alpha=0.3)
plt.show()
```



We compare with the analytically computed parameters and note that they are close.

```
print(μx, σx)
print(μ1 + ρ * σ1 * (0 - μ2) / σ2, np.sqrt(σ1**2 * (1 - ρ**2)))

print(μy, σy)
print(μ2 + ρ * σ2 * (1 - μ1) / σ1, np.sqrt(σ2**2 * (1 - ρ**2)))
```

```
-0.9997518414498444 2.2265841331697698
-1.0 2.227105745132009
5.039998363587647 0.9959878931873033
5.04 0.9959919678390986
```

8.14 Sum of two independently distributed random variables

Let X, Y be two independent discrete random variables that take values in $\bar{X}, \bar{Y}$, respectively.

Define a new random variable $Z = X + Y$.

Evidently, Z takes values from $\bar{Z}$ defined as follows:

$$\bar{X} = \{0, 1, \ldots, I-1\}; \qquad f_i = \text{Prob}\{X = i\}$$
$$\bar{Y} = \{0, 1, \ldots, J-1\}; \qquad g_j = \text{Prob}\{Y = j\}$$
$$\bar{Z} = \{0, 1, \ldots, I+J-2\}; \qquad h_k = \text{Prob}\{X + Y = k\}$$

Independence of X and Y implies that

$$h_k = \text{Prob}\{X = 0, Y = k\} + \text{Prob}\{X = 1, Y = k-1\} + ... + \text{Prob}\{X = k, Y = 0\}$$
$$h_k = f_0 g_k + f_1 g_{k-1} + ... + f_{k-1} g_1 + f_k g_0 \qquad \text{for} \quad k = 0, 1, ..., I + J - 2$$

Thus, we have:

$$h_k = \sum_{i=0}^{k} f_i g_{k-i} \equiv f * g$$

where $f * g$ denotes the **convolution** of the f and g sequences.

Similarly, for two random variables X, Y with densities f_X, g_Y, the density of $Z = X + Y$ is

$$f_Z(z) = \int_{-\infty}^{\infty} f_X(x) f_Y(z - x) dx \equiv f_X * g_Y$$

where $f_X * g_Y$ denotes the **convolution** of the f_X and g_Y functions.

8.15 Coupling

Start with a joint distribution

$$f_{ij} = \text{Prob}\{X = i, Y = j\}$$
$$i = 0, \cdots, I - 1$$
$$j = 0, \cdots, J - 1$$
$$\text{stacked to an } I \times J \text{ matrix}$$
$$e.g. \quad I = 2, J = 2$$

where

$$\begin{bmatrix} f_{00} & f_{01} \\ f_{10} & f_{11} \end{bmatrix}$$

From the joint distribution, we have shown above that we obtain **unique** marginal distributions.

Now we'll try to go in a reverse direction.

We'll find that from two marginal distributions we can usually construct more than one joint distribution that satisfies these marginals.

Each of these joint distributions is called a **coupling** of the two marginal distributions.

Let's start with marginal distributions

$$\text{Prob}\{X = i\} = \sum_j f_{ij} = \mu_i, i = 0, \cdots, I - 1$$
$$\text{Prob}\{Y = j\} = \sum_i f_{ij} = \nu_j, j = 0, \cdots, J - 1$$

Given two marginal distributions, μ for X and ν for Y, a joint distribution f_{ij} is said to be a **coupling** of μ and ν.

Consider the following bivariate example.

$$\text{Prob}\{X = 0\} = 1 - q = \mu_0$$
$$\text{Prob}\{X = 1\} = q = \mu_1$$
$$\text{Prob}\{Y = 0\} = 1 - r = \nu_0$$
$$\text{Prob}\{Y = 1\} = r = \nu_1$$
$$\text{where } 0 \leq q \leq r \leq 1$$

We construct two couplings.

The first coupling of our two marginal distributions is the joint distribution

$$f_{ij} = \begin{bmatrix} (1-q)(1-r) & (1-q)r \\ q(1-r) & qr \end{bmatrix}$$

To verify that it is a coupling, we check that

$$\begin{aligned} (1-q)(1-r) + (1-q)r + q(1-r) + qr &= 1 \\ \mu_0 = (1-q)(1-r) + (1-q)r &= 1-q \\ \mu_1 = q(1-r) + qr &= q \\ \nu_0 = (1-q)(1-r) + (1-r)q &= 1-r \\ \nu_1 = r(1-q) + qr &= r \end{aligned}$$

A second coupling of our two marginal distributions is the joint distribution

$$f_{ij} = \begin{bmatrix} (1-r) & r-q \\ 0 & q \end{bmatrix}$$

To verify that this is a coupling, note that

$$\begin{aligned} 1 - r + r - q + q &= 1 \\ \mu_0 &= 1-q \\ \mu_1 &= q \\ \nu_0 &= 1-r \\ \nu_1 &= r \end{aligned}$$

Thus, our two proposed joint distributions have the same marginal distributions.

But the joint distributions differ.

Thus, multiple joint distributions $[f_{ij}]$ can have the same marginals.

Couplings are important in optimal transport problems and in Markov processes. Please see this *lecture about optimal transport*.

8.16 Copula functions

Suppose that $X_1, X_2, \ldots, X_N$ are N random variables and that

- their marginal distributions are $F_1(x_1), F_2(x_2), \ldots, F_N(x_N)$, and
- their joint distribution is $H(x_1, x_2, \ldots, x_N)$

Then there exists a **copula function** $C(\cdot)$ that verifies

$$H(x_1, x_2, \ldots, x_N) = C(F_1(x_1), F_2(x_2), \ldots, F_N(x_N)).$$

If the marginal distributions are continuous, then the copula is unique.

In that case, we can recover it from the marginal inverses:

$$C(u_1, u_2, \ldots, u_N) = H(F_1^{-1}(u_1), F_2^{-1}(u_2), \ldots, F_N^{-1}(u_N))$$

When marginal distributions are not continuous, one uses generalized inverses, and the copula is uniquely determined only on $\text{Ran}(F_1) \times \cdots \times \text{Ran}(F_N)$.

In a reverse direction of logic, given univariate **marginal distributions** $F_1(x_1), F_2(x_2), \ldots, F_N(x_N)$ and a copula function $C(\cdot)$, the function $H(x_1, x_2, \ldots, x_N) = C(F_1(x_1), F_2(x_2), \ldots, F_N(x_N))$ is a **coupling** of $F_1(x_1), F_2(x_2), \ldots, F_N(x_N)$.

Thus, for given marginal distributions, we can use a copula function to determine a joint distribution when the associated univariate random variables are not independent.

Copula functions are often used to characterize **dependence** of random variables.

8.16.1 Bivariate examples with discrete and continuous distributions

Discrete marginal distribution

As mentioned above, for two given marginal distributions there can be more than one coupling.

For example, consider two random variables X, Y with distributions

$$\begin{aligned}
\text{Prob}(X = 0) &= 0.6, \\
\text{Prob}(X = 1) &= 0.4, \\
\text{Prob}(Y = 0) &= 0.3, \\
\text{Prob}(Y = 1) &= 0.7,
\end{aligned}$$

For these two random variables there can be more than one coupling.

Let's first generate X and Y.

```
μ = np.array([0.6, 0.4])
ν = np.array([0.3, 0.7])

# number of draws
draws = 1_000_000

# generate independent draws from uniform distribution for X and Y
p_x = rng.random(draws)
p_y = rng.random(draws)

# generate draws of X and Y via independent uniform draws
x = np.ones(draws)
y = np.ones(draws)
x[p_x <= μ[0]] = 0
x[p_x > μ[0]] = 1
y[p_y <= ν[0]] = 0
y[p_y > ν[0]] = 1
```

```
# calculate parameters from draws
q_hat = sum(x[x == 1])/draws
r_hat = sum(y[y == 1])/draws

# print output
print("distribution for x")
xmtb = pt.PrettyTable()
xmtb.field_names = ['x_value', 'x_prob']
xmtb.add_row([0, 1-q_hat])
xmtb.add_row([1, q_hat])
print(xmtb)
```

(continues on next page)

(continued from previous page)

```
print("distribution for y")
ymtb = pt.PrettyTable()
ymtb.field_names = ['y_value', 'y_prob']
ymtb.add_row([0, 1-r_hat])
ymtb.add_row([1, r_hat])
print(ymtb)
```

```
distribution for x
+---------+----------+
| x_value |  x_prob  |
+---------+----------+
|    0    | 0.599645 |
|    1    | 0.400355 |
+---------+----------+
distribution for y
+---------+---------------------+
| y_value |        y_prob       |
+---------+---------------------+
|    0    | 0.30042599999999997 |
|    1    |       0.699574      |
+---------+---------------------+
```

Let's now take our two marginal distributions, one for X, the other for Y, and construct two distinct couplings.

For the first joint distribution:

$$\text{Prob}(X = i, Y = j) = f_{ij}$$

where

$$[f_{ij}] = \begin{bmatrix} 0.18 & 0.42 \\ 0.12 & 0.28 \end{bmatrix}$$

Let's use Python to construct this joint distribution and then verify that its marginal distributions are what we want.

```
# define parameters
f1 = np.array([[0.18, 0.42], [0.12, 0.28]])
f1_cum = np.cumsum(f1)

# number of draws
draws1 = 1_000_000

# generate draws from uniform distribution
p = rng.random(draws1)

# generate draws of first coupling via uniform distribution
c1 = np.vstack([np.ones(draws1), np.ones(draws1)])
# X=0, Y=0
c1[0, p <= f1_cum[0]] = 0
c1[1, p <= f1_cum[0]] = 0
# X=0, Y=1
c1[0, (p > f1_cum[0])*(p <= f1_cum[1])] = 0
c1[1, (p > f1_cum[0])*(p <= f1_cum[1])] = 1
# X=1, Y=0
c1[0, (p > f1_cum[1])*(p <= f1_cum[2])] = 1
c1[1, (p > f1_cum[1])*(p <= f1_cum[2])] = 0
# X=1, Y=1
```

(continues on next page)

(continued from previous page)

```
c1[0, (p > f1_cum[2])*(p <= f1_cum[3])] = 1
c1[1, (p > f1_cum[2])*(p <= f1_cum[3])] = 1
```

```
# calculate parameters from draws
f1_00 = sum((c1[0, :] == 0)*(c1[1, :] == 0))/draws1
f1_01 = sum((c1[0, :] == 0)*(c1[1, :] == 1))/draws1
f1_10 = sum((c1[0, :] == 1)*(c1[1, :] == 0))/draws1
f1_11 = sum((c1[0, :] == 1)*(c1[1, :] == 1))/draws1

# print output of first joint distribution
print("first joint distribution for c1")
c1_mtb = pt.PrettyTable()
c1_mtb.field_names = ['c1_x_value', 'c1_y_value', 'c1_prob']
c1_mtb.add_row([0, 0, f1_00])
c1_mtb.add_row([0, 1, f1_01])
c1_mtb.add_row([1, 0, f1_10])
c1_mtb.add_row([1, 1, f1_11])
print(c1_mtb)
```

```
first joint distribution for c1
+------------+------------+----------+
| c1_x_value | c1_y_value | c1_prob  |
+------------+------------+----------+
|     0      |     0      | 0.18002  |
|     0      |     1      | 0.419939 |
|     1      |     0      | 0.120069 |
|     1      |     1      | 0.279972 |
+------------+------------+----------+
```

```
# calculate parameters from draws
c1_q_hat = sum(c1[0, :] == 1)/draws1
c1_r_hat = sum(c1[1, :] == 1)/draws1

# print output
print("marginal distribution for x")
c1_x_mtb = pt.PrettyTable()
c1_x_mtb.field_names = ['c1_x_value', 'c1_x_prob']
c1_x_mtb.add_row([0, 1-c1_q_hat])
c1_x_mtb.add_row([1, c1_q_hat])
print(c1_x_mtb)

print("marginal distribution for y")
c1_ymtb = pt.PrettyTable()
c1_ymtb.field_names = ['c1_y_value', 'c1_y_prob']
c1_ymtb.add_row([0, 1-c1_r_hat])
c1_ymtb.add_row([1, c1_r_hat])
print(c1_ymtb)
```

```
marginal distribution for x
+------------+-----------+
| c1_x_value | c1_x_prob |
+------------+-----------+
|     0      |  0.599959 |
|     1      |  0.400041 |
+------------+-----------+
```

(continues on next page)

(continued from previous page)

```
marginal distribution for y
+------------+---------------------+
| c1_y_value |      c1_y_prob      |
+------------+---------------------+
|     0      | 0.30008900000000005 |
|     1      |       0.699911      |
+------------+---------------------+
```

Now, let's construct another joint distribution that is also a coupling of X and Y

$$[f_{ij}] = \begin{bmatrix} 0.3 & 0.3 \\ 0 & 0.4 \end{bmatrix}$$

```
# define parameters
f2 = np.array([[0.3, 0.3], [0, 0.4]])
f2_cum = np.cumsum(f2)

# number of draws
draws2 = 1_000_000

# generate draws from uniform distribution
p = rng.random(draws2)

# generate draws of second coupling via uniform distribution
c2 = np.vstack([np.ones(draws2), np.ones(draws2)])
# X=0, Y=0
c2[0, p <= f2_cum[0]] = 0
c2[1, p <= f2_cum[0]] = 0
# X=0, Y=1
c2[0, (p > f2_cum[0])*(p <= f2_cum[1])] = 0
c2[1, (p > f2_cum[0])*(p <= f2_cum[1])] = 1
# X=1, Y=0
c2[0, (p > f2_cum[1])*(p <= f2_cum[2])] = 1
c2[1, (p > f2_cum[1])*(p <= f2_cum[2])] = 0
# X=1, Y=1
c2[0, (p > f2_cum[2])*(p <= f2_cum[3])] = 1
c2[1, (p > f2_cum[2])*(p <= f2_cum[3])] = 1
```

```
# calculate parameters from draws
f2_00 = sum((c2[0, :] == 0)*(c2[1, :] == 0))/draws2
f2_01 = sum((c2[0, :] == 0)*(c2[1, :] == 1))/draws2
f2_10 = sum((c2[0, :] == 1)*(c2[1, :] == 0))/draws2
f2_11 = sum((c2[0, :] == 1)*(c2[1, :] == 1))/draws2

# print output of second joint distribution
print("second joint distribution for c2")
c2_mtb = pt.PrettyTable()
c2_mtb.field_names = ['c2_x_value', 'c2_y_value', 'c2_prob']
c2_mtb.add_row([0, 0, f2_00])
c2_mtb.add_row([0, 1, f2_01])
c2_mtb.add_row([1, 0, f2_10])
c2_mtb.add_row([1, 1, f2_11])
print(c2_mtb)
```

```
second joint distribution for c2
+------------+------------+----------+
```

(continues on next page)

(continued from previous page)

```
| c2_x_value | c2_y_value | c2_prob  |
+------------+------------+----------+
|     0      |     0      | 0.300568 |
|     0      |     1      | 0.299249 |
|     1      |     0      |   0.0    |
|     1      |     1      | 0.400183 |
+------------+------------+----------+
```

```
# calculate parameters from draws
c2_q_hat = sum(c2[0, :] == 1)/draws2
c2_r_hat = sum(c2[1, :] == 1)/draws2

# print output
print("marginal distribution for x")
c2_x_mtb = pt.PrettyTable()
c2_x_mtb.field_names = ['c2_x_value', 'c2_x_prob']
c2_x_mtb.add_row([0, 1-c2_q_hat])
c2_x_mtb.add_row([1, c2_q_hat])
print(c2_x_mtb)

print("marginal distribution for y")
c2_ymtb = pt.PrettyTable()
c2_ymtb.field_names = ['c2_y_value', 'c2_y_prob']
c2_ymtb.add_row([0, 1-c2_r_hat])
c2_ymtb.add_row([1, c2_r_hat])
print(c2_ymtb)
```

```
marginal distribution for x
+------------+-----------+
| c2_x_value | c2_x_prob |
+------------+-----------+
|     0      |  0.599817 |
|     1      |  0.400183 |
+------------+-----------+
marginal distribution for y
+------------+---------------------+
| c2_y_value |      c2_y_prob      |
+------------+---------------------+
|     0      | 0.30056799999999995 |
|     1      |       0.699432      |
+------------+---------------------+
```

We have verified that both joint distributions, c_1 and c_2, have identical marginal distributions of X and Y, respectively.

So they are both couplings of X and Y.

8.16.2 Gaussian copula example

A **Gaussian copula** uses the bivariate normal distribution to induce dependence between arbitrary marginal distributions.

The construction has three steps:

1. Draw (Z_1, Z_2) from a bivariate standard normal with correlation ρ.
2. Apply the standard normal CDF: $U_k = \Phi(Z_k)$.
    - The pair (U_1, U_2) has uniform marginals but retains the dependence structure of (Z_1, Z_2) — this is the copula.
3. Apply the inverse CDF of any desired marginal: $X_k = F_k^{-1}(U_k)$.

The following code illustrates this with exponential marginals.

```
# Gaussian copula parameters
ρ_cop = 0.8
n_cop = 100_000

# Draw from bivariate standard normal with correlation ρ_cop
z = rng.multivariate_normal(
    [0, 0], [[1, ρ_cop], [ρ_cop, 1]], n_cop
)

# Apply normal CDF -> uniform marginals (the copula itself)
u1 = stats.norm.cdf(z[:, 0])
u2 = stats.norm.cdf(z[:, 1])

# Apply inverse CDFs of desired marginals (here: Exponential)
x1 = stats.expon.ppf(u1, scale=1.0)    # Exp with mean 1
x2 = stats.expon.ppf(u2, scale=0.5)    # Exp with mean 0.5

fig, axes = plt.subplots(1, 2, figsize=(10, 4))
axes[0].scatter(u1[:3000], u2[:3000], alpha=0.2, s=2)
axes[0].set_xlabel('$u_1$')
axes[0].set_ylabel('$u_2$')
axes[1].scatter(x1[:3000], x2[:3000], alpha=0.2, s=2)
axes[1].set_xlabel('$x_1$ (Exp, mean=1)')
axes[1].set_ylabel('$x_2$ (Exp, mean=0.5)')
plt.show()

print(f"Sample correlation of (x1, x2): {np.corrcoef(x1, x2)[0, 1]:.3f}")
print(f"Sample correlation of (u1, u2): {np.corrcoef(u1, u2)[0, 1]:.3f}")
```

```
Sample correlation of (x1, x2): 0.768
Sample correlation of (u1, u2): 0.786
```

The left panel shows the copula itself – the dependence structure in uniform coordinates, drawn from a bivariate normal with correlation $\rho = 0.8$.

The right panel shows the same dependence translated to exponential marginals.

Changing ρ controls the strength of dependence while the marginals remain unchanged.

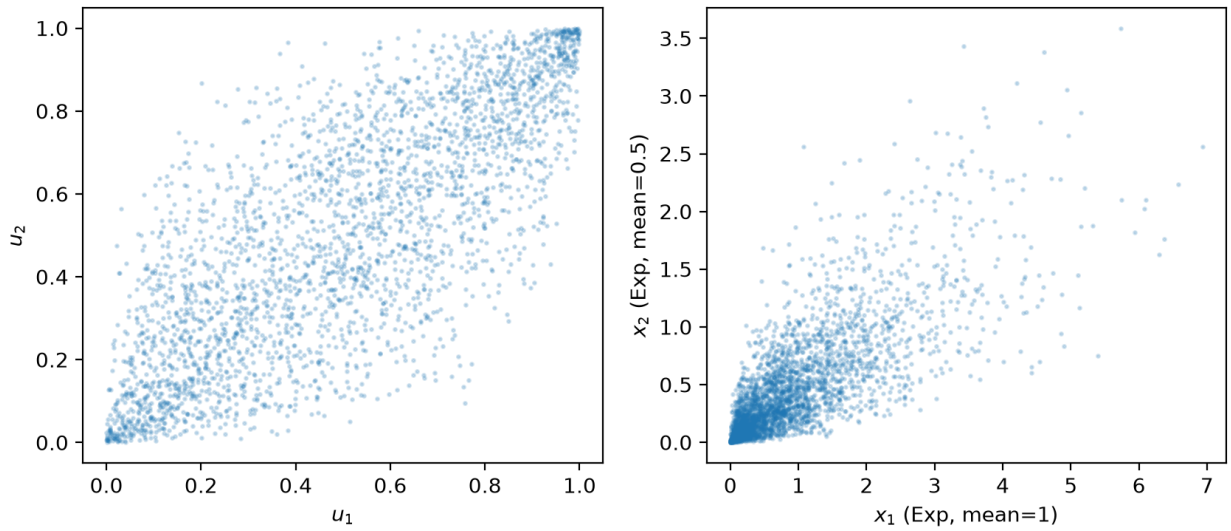


Fig. 8.1: gaussian copula with exponential marginals

8.17 Exercises

Exercise 8.17.1

Independence Test

Consider the joint distribution

$$F = \begin{bmatrix} 0.3 & 0.2 \\ 0.1 & 0.4 \end{bmatrix}$$

where $X \in \{0, 1\}$ and $Y \in \{10, 20\}$.

1. Compute the marginal distributions $\mu_i = \text{Prob}\{X = i\}$ and $\nu_j = \text{Prob}\{Y = j\}$.
2. Form the independence matrix $f_{ij}^{\perp} = \mu_i \nu_j$ (the outer product of the two marginal vectors).
3. Compare F with $f^{\perp}$ and determine whether X and Y are independent.
4. Verify your conclusion by computing $\text{Prob}\{X = 0 | Y = 10\}$ and checking whether it equals $\text{Prob}\{X = 0\}$.

Solution

Here is one solution:

```
F = np.array([[0.3, 0.2],
              [0.1, 0.4]])

μ = F.sum(axis=1)
ν = F.sum(axis=0)
print("μ (marginal of X):", μ)
print("ν (marginal of Y):", ν)

F_indep = np.outer(μ, ν)
```

```
print("\nIndependence matrix (outer product):\n", F_indep)
print("\nActual joint F:\n", F)

print("\nIndependent (F == μ times ν)?", np.allclose(F, F_indep))

prob_X0_given_Y10 = F[0, 0] / ν[0]
print(f"\nProb(X=0 | Y=10) = {prob_X0_given_Y10:.4f}")
print(f"Prob(X=0)          = {μ[0]:.4f}")
```

```
μ (marginal of X): [0.5 0.5]
ν (marginal of Y): [0.4 0.6]

Independence matrix (outer product):
 [[0.2 0.3]
 [0.2 0.3]]

Actual joint F:
 [[0.3 0.2]
 [0.1 0.4]]

Independent (F == μ times ν)? False

Prob(X=0 | Y=10) = 0.7500
Prob(X=0)          = 0.5000
```

Exercise 8.17.2

Covariance and Correlation

Using the same joint distribution F and values $X \in \{0, 1\}$, $Y \in \{10, 20\}$ as in Exercise 1:

1. Compute $\mathbb{E}[X]$, $\mathbb{E}[Y]$, and $\mathbb{E}[XY] = \sum_i \sum_j x_i y_j f_{ij}$.
2. Compute $\text{Cov}(X, Y) = \mathbb{E}[XY] - \mathbb{E}[X]\mathbb{E}[Y]$.
3. Compute $\text{Cor}(X, Y) = \text{Cov}(X, Y)/(\sigma_X \sigma_Y)$.
4. Show analytically that $X \perp Y$ implies $\text{Cov}(X, Y) = 0$.

Solution

Here is one solution:

```
xs = np.array([0, 1])
ys = np.array([10, 20])
F  = np.array([[0.3, 0.2],
               [0.1, 0.4]])

μ = F.sum(axis=1)
ν = F.sum(axis=0)

E_X  = xs @ μ
E_Y  = ys @ ν
E_XY = sum(xs[i] * ys[j] * F[i, j] for i in range(2) for j in range(2))
print(f"E[X] = {E_X}, E[Y] = {E_Y}, E[XY] = {E_XY}")

cov_XY = E_XY - E_X * E_Y
```

```
print(f"Cov(X,Y) = {cov_XY:.4f}")

var_X  = ((xs - E_X)**2) @ μ
var_Y  = ((ys - E_Y)**2) @ ν
cor_XY = cov_XY / np.sqrt(var_X * var_Y)
print(f"Cor(X,Y) = {cor_XY:.4f}")
```

```
E[X] = 0.5, E[Y] = 16.0, E[XY] = 9.0
Cov(X,Y) = 1.0000
Cor(X,Y) = 0.4082
```

For part 4: if $X \perp Y$ then $f_{ij} = \mu_i \nu_j$, so

$$\mathbb{E}[XY] = \sum_i \sum_j x_i y_j \mu_i \nu_j = \left(\sum_i x_i \mu_i \right) \left(\sum_j y_j \nu_j \right) = \mathbb{E}[X]\,\mathbb{E}[Y]$$

and therefore $\text{Cov}(X, Y) = \mathbb{E}[XY] - \mathbb{E}[X]\mathbb{E}[Y] = 0$.

Exercise 8.17.3

Sum of Two Dice

Let X and Y be **independent** random variables, each uniformly distributed on $\{1, 2, 3, 4, 5, 6\}$, and let $Z = X + Y$.

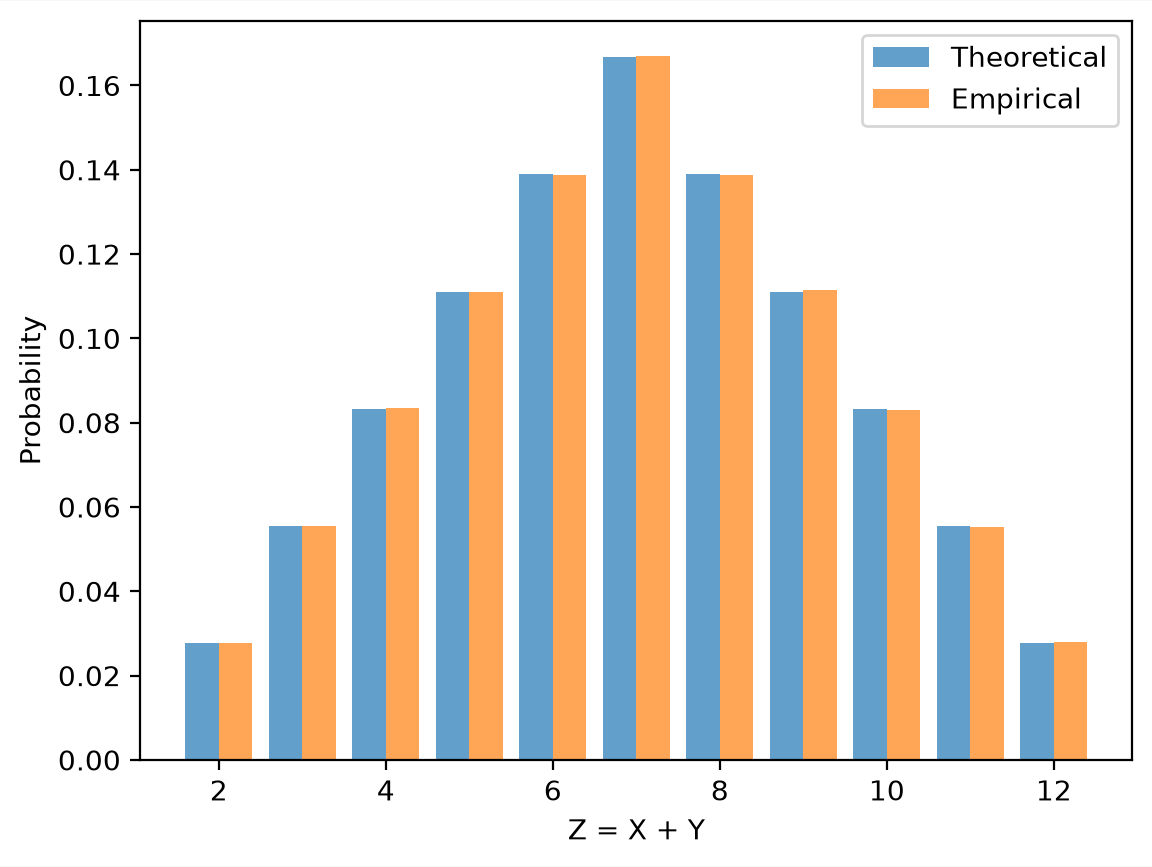
1. Use the convolution formula $h_k = \sum_i f_i g_{k-i}$ to compute the distribution of Z.
2. Plot the result generated by the formula.
3. Simulate 10^6 rolls and overlay the empirical histogram on the plot.
4. Compute $\mathbb{E}[Z]$ and $\text{Var}(Z)$ from the two calculations

Solution

Here is one solution:

```
f = np.ones(6) / 6
g = np.ones(6) / 6
h = [
    sum(f[i]*g[k-i] for i in range(
        max(0, k-len(g)+1), # f_i exists
        min(len(f), k+1))   # g_{k-i} exists
        )
        for k in range(len(f) + len(g) - 1)]
z_vals = np.arange(2, 13)

n = 1_000_000
z_sim = rng.integers(1, 7, n) + rng.integers(1, 7, n)
counts = np.bincount(z_sim, minlength=13)[2:]

fig, ax = plt.subplots()
ax.bar(z_vals - 0.2, h,          0.4, alpha=0.7, label='Theoretical')
ax.bar(z_vals + 0.2, counts / n, 0.4, alpha=0.7, label='Empirical')
ax.set_xlabel('Z = X + Y')
ax.set_ylabel('Probability')
ax.legend()
```

```
plt.show()

E_Z   = z_vals @ h
Var_Z = ((z_vals - E_Z)**2) @ h
print(f"Theory:     E[Z] = {E_Z:.2f}, Var(Z) = {Var_Z:.4f}")
print(f"Simulation: E[Z] = {np.mean(z_sim):.2f}, Var(Z) = {np.var(z_sim):.4f}")
```




```
Theory:     E[Z] = 7.00, Var(Z) = 5.8333
Simulation: E[Z] = 7.00, Var(Z) = 5.8271
```

Exercise 8.17.4

Multi-Step Transition Probabilities

Consider a two-state Markov chain with transition matrix

$$P = \begin{bmatrix} 0.9 & 0.1 \\ 0.2 & 0.8 \end{bmatrix}$$

where $p_{ij} = \text{Prob}\{X(t+1) = j \mid X(t) = i\}$.

1. Starting from $\psi_0 = [1, 0]$, compute $\psi_n = \psi_0 P^n$ for $n = 1, 5, 20, 100$.
2. Find the stationary distribution ψ^* satisfying $\psi^* P = \psi^*$ and $\sum_i \psi_i^* = 1$.
3. Verify numerically that $\psi_n \to \psi^*$ as n grows.

Solution

Here is one solution:

```
P = np.array([[0.9, 0.1],
              [0.2, 0.8]])
ψ0 = np.array([1.0, 0.0])

for n in [1, 5, 20, 100]:
    print(f"ψ_{n:3d} = {ψ0 @ np.linalg.matrix_power(P, n)}")

A = np.vstack([P.T - np.eye(2), np.ones(2)])
b = np.array([0.0, 0.0, 1.0])
ψ_star, *_ = np.linalg.lstsq(A, b, rcond=None)
print(f"\nStationary distribution: {ψ_star}")

ψ_100 = ψ0 @ np.linalg.matrix_power(P, 100)
print(f"ψ_100 close to stationary? {np.allclose(ψ_100, ψ_star, atol=1e-6)}")
```

```
ψ_  1 = [0.9 0.1]
ψ_  5 = [0.72269 0.27731]
ψ_ 20 = [0.66693264 0.33306736]
ψ_100 = [0.66666667 0.33333333]

Stationary distribution: [0.66666667 0.33333333]
ψ_100 close to stationary? True
```

Exercise 8.17.5

Bayes' Law with a Discrete Prior

A coin has unknown bias $\theta \in \{0.2,\ 0.5,\ 0.8\}$ with prior $\pi = [0.25,\ 0.50,\ 0.25]$.

Assume that, conditional on θ, the coin flips are i.i.d. Bernoulli(θ).

1. After observing $k = 7$ heads in $n = 10$ flips, compute the likelihood

$$\mathcal{L}(\theta \mid \text{data}) = \binom{10}{7} \theta^7 (1-\theta)^3$$

   for each θ.

2. Apply equation (8.4) to compute the posterior $\pi(\theta \mid \text{data})$.
3. Plot the prior and posterior side by side.
4. Repeat for $k = 3$ heads and describe how the posterior shifts.

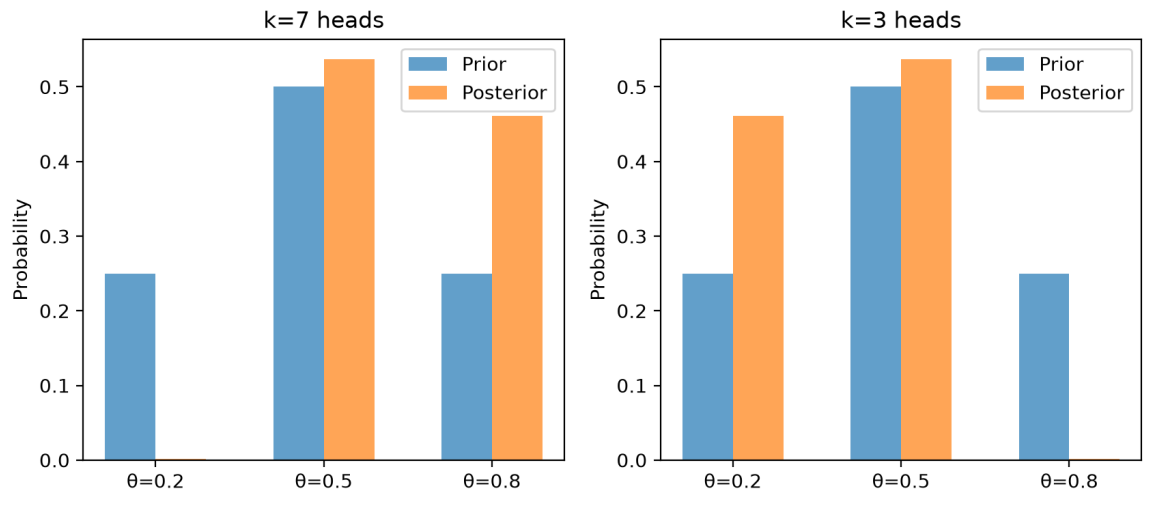
Solution

Here is one solution:

```
θ_vals = np.array([0.2, 0.5, 0.8])
π = np.array([0.25, 0.50, 0.25])

def compute_posterior(k, n, θ_vals, π):
    likelihood = comb(n, k) * θ_vals**k * (1 - θ_vals)**(n - k)
```

```
    unnorm = likelihood * π
    return unnorm / unnorm.sum(), likelihood

post7, lik7 = compute_posterior(7, 10, θ_vals, π)
post3, lik3 = compute_posterior(3, 10, θ_vals, π)

print("k=7:  likelihood =", lik7.round(4),
      " posterior =", post7.round(4))
print("k=3:  likelihood =", lik3.round(4),
      " posterior =", post3.round(4))

x = np.arange(len(θ_vals))
w = 0.3
fig, axes = plt.subplots(1, 2, figsize=(10, 4))
for ax, post, title in zip(
    axes, [post7, post3], ['k=7 heads', 'k=3 heads']):
    ax.bar(x - w/2, π, w, label='Prior',     alpha=0.7)
    ax.bar(x + w/2, post,  w, label='Posterior', alpha=0.7)
    ax.set_xticks(x)
    ax.set_xticklabels([f'θ={t}' for t in θ_vals])
    ax.set_ylabel('Probability')
    ax.set_title(title)
    ax.legend()
plt.show()
```

```
k=7:  likelihood = [0.0008 0.1172 0.2013]  posterior = [0.0018 0.537  0.4612]
k=3:  likelihood = [0.2013 0.1172 0.0008]  posterior = [0.4612 0.537  0.0018]
```



CHAPTER

NINE

SOME PROBABILITY DISTRIBUTIONS

This lecture is a supplement to *this lecture on statistics with matrices*.

It describes some popular distributions and uses Python to sample from them.

It also describes a way to sample from an arbitrary probability distribution that you make up by transforming a sample from a uniform probability distribution.

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install prettytable
```

As usual, we'll start with some imports

```
import numpy as np
import matplotlib.pyplot as plt
import prettytable as pt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib_inline.backend_inline import set_matplotlib_formats
set_matplotlib_formats('retina')
```

9.1 Some Discrete Probability Distributions

Let's write some Python code to compute means and variances of some univariate random variables.

We'll use our code to

- compute population means and variances from the probability distribution
- generate a sample of N independently and identically distributed draws and compute sample means and variances
- compare population and sample means and variances

9.2 Geometric distribution

A discrete geometric distribution has probability mass function

$$\text{Prob}(X = k) = (1-p)^{k-1}p, k = 1, 2, \ldots, \quad p \in (0, 1)$$

where $k = 1, 2, \ldots$ is the number of trials before the first success.

The mean and variance of this one-parameter probability distribution are

$$\mathbb{E}(X) = \frac{1}{p}$$
$$\mathbb{V}\text{ar}(X) = \frac{1-p}{p^2}$$

Let's use Python draw observations from the distribution and compare the sample mean and variance with the theoretical results.

```
rng = np.random.default_rng()

# specify parameters
p, n = 0.3, 1_000_000

# draw observations from the distribution
x = rng.geometric(p, n)

# compute sample mean and variance
μ_hat = np.mean(x)
σ2_hat = np.var(x)

print("The sample mean is: ", μ_hat, "\nThe sample variance is: ", σ2_hat)

# compare with theoretical results
print("\nThe population mean is: ", 1/p)
print("The population variance is: ", (1-p)/(p**2))
```

```
The sample mean is:  3.337025
The sample variance is:  7.778195149374999

The population mean is:  3.3333333333333335
The population variance is:  7.777777777777778
```

9.3 Pascal (negative binomial) distribution

Consider a sequence of independent Bernoulli trials.

Let p be the probability of success.

Let X be a random variable that represents the number of failures before we get r successes.

Its distribution is

$$X \sim NB(r, p)$$
$$\text{Prob}(X = k; r, p) = \begin{bmatrix} k+r-1 \\ r-1 \end{bmatrix} p^r (1-p)^k$$

Here, we choose from among $k + r - 1$ possible outcomes because the last draw is by definition a success.

We compute the mean and variance to be

$$\mathbb{E}(X) = \frac{k(1-p)}{p}$$
$$\mathbb{V}(X) = \frac{k(1-p)}{p^2}$$

```
# specify parameters
r, p, n = 10, 0.3, 1_000_000

# draw observations from the distribution
x = rng.negative_binomial(r, p, n)

# compute sample mean and variance
μ_hat = np.mean(x)
σ2_hat = np.var(x)

print("The sample mean is: ", μ_hat, "\nThe sample variance is: ", σ2_hat)
print("\nThe population mean is: ", r*(1-p)/p)
print("The population variance is: ", r*(1-p)/p**2)
```

```
The sample mean is:  23.318165
The sample variance is:  77.640348032775

The population mean is:  23.333333333333336
The population variance is:  77.77777777777779
```

9.4 Newcomb–Benford distribution

The **Newcomb–Benford law** fits many data sets, e.g., reports of incomes to tax authorities, in which the leading digit is more likely to be small than large.

See https://en.wikipedia.org/wiki/Benford's_law

A Benford probability distribution is

$$\text{Prob}\{X = d\} = \log_{10}(d+1) - \log_{10}(d) = \log_{10}\left(1 + \frac{1}{d}\right)$$

where $d \in \{1, 2, \cdots, 9\}$ can be thought of as a **first digit** in a sequence of digits.

This is a well defined discrete distribution since we can verify that probabilities are nonnegative and sum to 1.

$$\log_{10}\left(1 + \frac{1}{d}\right) \geq 0, \quad \sum_{d=1}^{9} \log_{10}\left(1 + \frac{1}{d}\right) = 1$$

The mean and variance of a Benford distribution are

$$\mathbb{E}[X] = \sum_{d=1}^{9} d \log_{10}\left(1 + \frac{1}{d}\right) \simeq 3.4402$$

$$\mathbb{V}[X] = \sum_{d=1}^{9} (d - \mathbb{E}[X])^2 \log_{10}\left(1 + \frac{1}{d}\right) \simeq 6.0565$$

We verify the above and compute the mean and variance using `numpy`.

```
Benford_pmf = np.array([np.log10(1+1/d) for d in range(1,10)])
k = np.arange(1, 10)

# mean
mean = k @ Benford_pmf
```

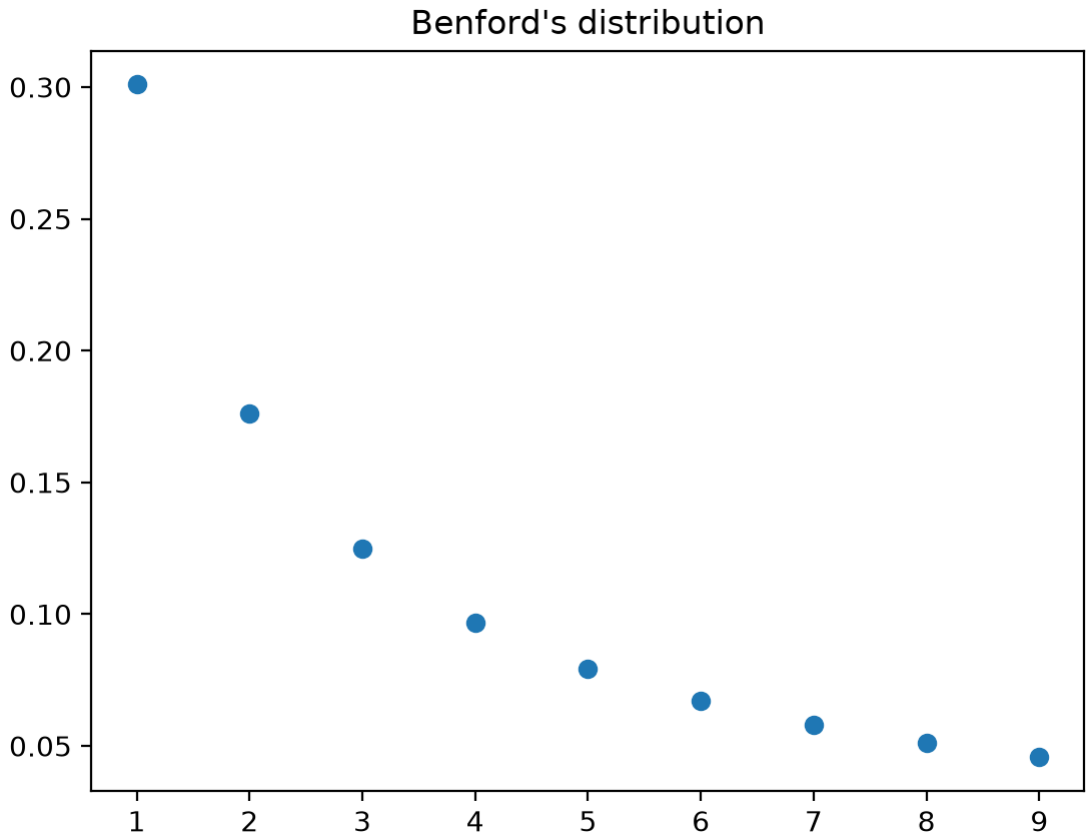
(continues on next page)

(continued from previous page)

```
# variance
var = ((k - mean) ** 2) @ Benford_pmf

# verify sum to 1
print(np.sum(Benford_pmf))
print(mean)
print(var)
```

```
0.9999999999999999
3.4402369671232065
6.056512631375667
```

```
# plot distribution
plt.plot(range(1,10), Benford_pmf, 'o')
plt.title('Benford\'s distribution')
plt.show()
```



Now let's turn to some continuous random variables.

9.5 Univariate Gaussian distribution

We write

$$X \sim N(\mu, \sigma^2)$$

to indicate the probability distribution

$$f(x|u, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{[-\frac{1}{2\sigma^2}(x-u)^2]}$$

In the below example, we set $\mu = 0, \sigma = 0.1$.

```
# specify parameters
μ, σ = 0, 0.1

# specify number of draws
n = 1_000_000

# draw observations from the distribution
x = rng.normal(μ, σ, n)

# compute sample mean and variance
μ_hat = np.mean(x)
σ_hat = np.std(x)

print("The sample mean is: ", μ_hat)
print("The sample standard deviation is: ", σ_hat)
```

```
The sample mean is:  0.00011071660878087941
The sample standard deviation is:  0.09996537547977614
```

```
# compare
print(μ-μ_hat < 1e-3)
print(σ-σ_hat < 1e-3)
```

```
True
True
```

9.6 Uniform Distribution

$$X \sim U[a, b]$$

$$f(x) = \begin{cases} \frac{1}{b-a}, & a \leq x \leq b \\ 0, & \text{otherwise} \end{cases}$$

The population mean and variance are

$$\mathbb{E}(X) = \frac{a+b}{2}$$

$$\mathbb{V}(X) = \frac{(b-a)^2}{12}$$

```
# specify parameters
a, b = 10, 20

# specify number of draws
n = 1_000_000

# draw observations from the distribution
x = a + (b-a)*rng.random(n)

# compute sample mean and variance
μ_hat = np.mean(x)
σ2_hat = np.var(x)

print("The sample mean is: ", μ_hat, "\nThe sample variance is: ", σ2_hat)
print("\nThe population mean is: ", (a+b)/2)
print("The population variance is: ", (b-a)**2/12)
```

```
The sample mean is:  15.004362744970496
The sample variance is:  8.338127502889677

The population mean is:  15.0
The population variance is:  8.333333333333334
```

9.7 A Mixed Discrete-Continuous Distribution

We'll motivate this example with a little story.

Suppose that to apply for a job you take an interview and either pass or fail it.

You have 5% chance to pass an interview and you know your salary will uniformly distributed in the interval 300~400 a day only if you pass.

We can describe your daily salary as a discrete-continuous variable with the following probabilities:

$$P(X = 0) = 0.95$$

$$P(300 \leq X \leq 400) = \int_{300}^{400} f(x)\, dx = 0.05$$

$$f(x) = 0.0005$$

Let's start by generating a random sample and computing sample moments.

```
x = rng.random(1_000_000)
# x[x > 0.95] = 100*x[x > 0.95]+300
x[x > 0.95] = 100*rng.random(len(x[x > 0.95]))+300
x[x <= 0.95] = 0

μ_hat = np.mean(x)
σ2_hat = np.var(x)

print("The sample mean is: ", μ_hat, "\nThe sample variance is: ", σ2_hat)
```

```
The sample mean is:  17.476369433321434
The sample variance is:  5852.208435389325
```

The analytical mean and variance can be computed:

$$\begin{aligned}\mu &= \int_{300}^{400} xf(x)dx \\ &= 0.0005 \int_{300}^{400} xdx \\ &= 0.0005 \times \frac{1}{2}x^2\Big|_{300}^{400}\end{aligned}$$

$$\begin{aligned}\sigma^2 &= 0.95 \times (0-17.5)^2 + \int_{300}^{400} (x-17.5)^2 f(x)dx \\ &= 0.95 \times 17.5^2 + 0.0005 \int_{300}^{400} (x-17.5)^2 dx \\ &= 0.95 \times 17.5^2 + 0.0005 \times \frac{1}{3}(x-17.5)^3\Big|_{300}^{400}\end{aligned}$$

```
mean = 0.0005*0.5*(400**2 - 300**2)
var = 0.95*17.5**2+0.0005/3*((400-17.5)**3-(300-17.5)**3)
print("mean: ", mean)
print("variance: ", var)
```

```
mean:  17.5
variance:  5860.416666666666
```

9.8 Drawing a Random Number from a Particular Distribution

Suppose we have at our disposal a pseudo random number that draws a uniform random variable, i.e., one with probability distribution

$$\text{Prob}\{\tilde{X} = i\} = \frac{1}{I}, \quad i = 0, \dots, I-1$$

How can we transform $\tilde{X}$ to get a random variable X for which $\text{Prob}\{X = i\} = f_i, \quad i = 0, \dots, I-1$, where f_i is an arbitary discrete probability distribution on $i = 0, 1, \dots, I-1$?

The key tool is the inverse of a cumulative distribution function (CDF).

Observe that the CDF of a distribution is monotone and non-decreasing, taking values between 0 and 1.

We can draw a sample of a random variable X with a known CDF as follows:

- draw a random variable u from a uniform distribution on $[0, 1]$
- pass the sample value of u into the **"inverse"** target CDF for X
- X has the target CDF

Thus, knowing the **"inverse"** CDF of a distribution is enough to simulate from this distribution.

Note

The "inverse" CDF needs to exist for this method to work.

The inverse CDF is

$$F^{-1}(u) \equiv \inf\{x \in \mathbb{R} : F(x) \geq u\} \quad (0 < u < 1)$$

Here we use infimum because a CDF is a non-decreasing and right-continuous function.

Thus, suppose that

- U is a uniform random variable $U \in [0, 1]$
- We want to sample a random variable X whose CDF is F.

It turns out that if we use draw uniform random numbers U and then compute X from

$$X = F^{-1}(U),$$

then X is a random variable with CDF $F_X(x) = F(x) = \text{Prob}\{X \leq x\}$.

We'll verify this in the special case in which F is continuous and bijective so that its inverse function exists and can be denoted by F^{-1}.

Note that

$$\begin{aligned} F_X(x) &= \text{Prob}\{X \leq x\} \\ &= \text{Prob}\left\{F^{-1}(U) \leq x\right\} \\ &= \text{Prob}\{U \leq F(x)\} \\ &= F(x) \end{aligned}$$

where the last equality occurs because U is distributed uniformly on $[0, 1]$ while $F(x)$ is a constant given x that also lies on $[0, 1]$.

Let's use `numpy` to compute some examples.

Example: A continuous geometric (exponential) distribution

Let X follow a geometric distribution, with parameter $\lambda > 0$.

Its density function is

$$f(x) = \lambda e^{-\lambda x}$$

Its CDF is

$$F(x) = \int_0^\infty \lambda e^{-\lambda x} = 1 - e^{-\lambda x}$$

Let U follow a uniform distribution on $[0, 1]$.

X is a random variable such that $U = F(X)$.

The distribution X can be deduced from

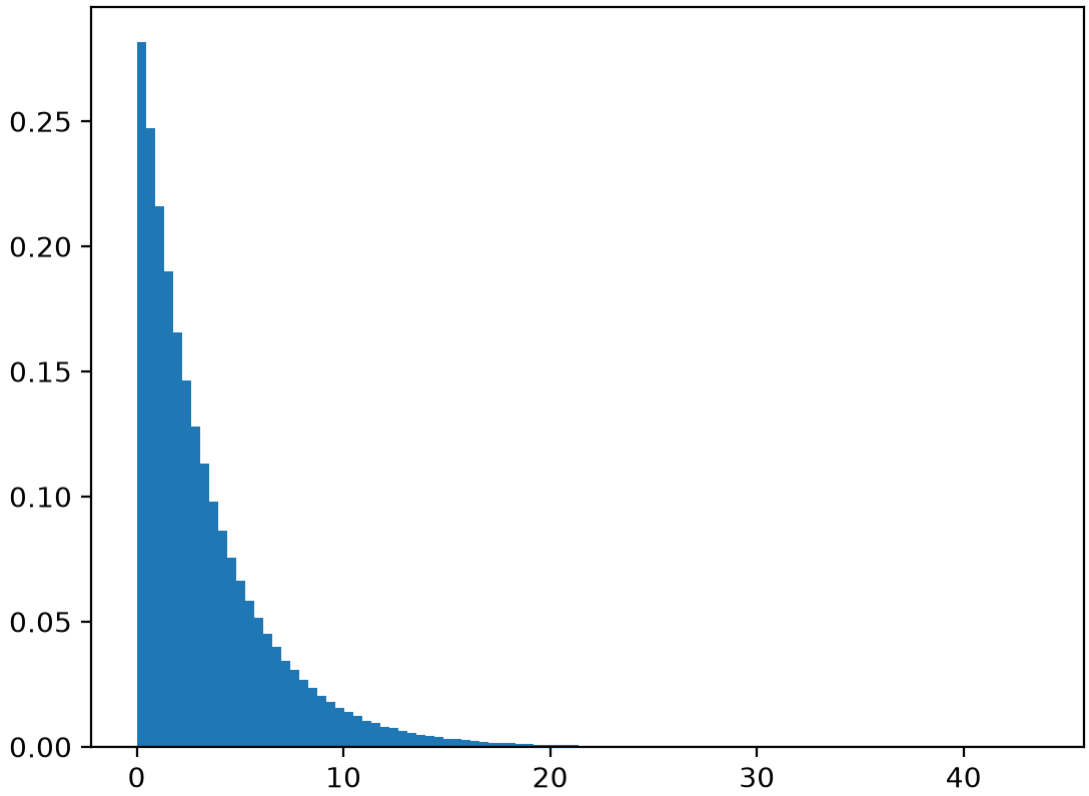
$$\begin{aligned} U = F(X) &= 1 - e^{-\lambda X} \\ \implies \quad -U &= e^{-\lambda X} \\ \implies \quad \log(1-U) &= -\lambda X \\ \implies \quad X &= \frac{(1-U)}{-\lambda} \end{aligned}$$

Let's draw u from $U[0, 1]$ and calculate $x = \frac{log(1-U)}{-\lambda}$.

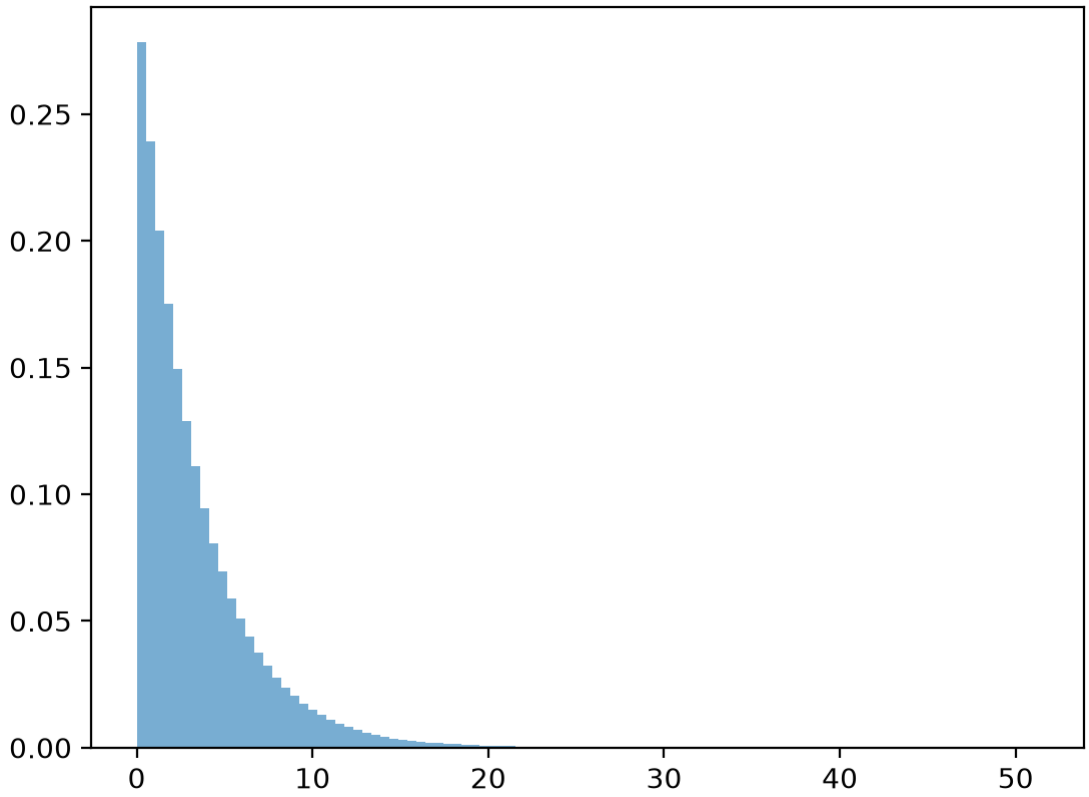
We'll check whether X seems to follow a **continuous geometric** (exponential) distribution.

Let's check with `numpy`.

```
n, λ = 1_000_000, 0.3

# draw uniform numbers
u = rng.random(n)

# transform
x = -np.log(1-u)/λ

# draw geometric distributions
x_g = rng.exponential(1 / λ, n)

# plot and compare
plt.hist(x, bins=100, density=True)
plt.show()
```




```
plt.hist(x_g, bins=100, density=True, alpha=0.6)
plt.show()
```



Geometric distribution

Let X distributed geometrically, that is

$$\text{Prob}(X = i) = (1 - \lambda)\lambda^i, \quad \lambda \in (0, 1), \quad i = 0, 1, ...$$

$$\sum_{i=0}^{\infty} \text{Prob}(X = i) = 1 \longleftrightarrow (1 - \lambda) \sum_{i=0}^{\infty} \lambda^i = \frac{1 - \lambda}{1 - \lambda} = 1$$

Its CDF is given by

$$\begin{aligned} \text{Prob}(X \leq i) &= (1 - \lambda) \sum_{j=0}^{i} \lambda^i \\ &= (1 - \lambda)[\frac{1 - \lambda^{i+1}}{1 - \lambda}] \\ &= 1 - \lambda^{i+1} \\ &= F(X) = F_i \end{aligned}$$

Again, let $\tilde{U}$ follow a uniform distribution and we want to find X such that $F(X) = \tilde{U}$.

Let's deduce the distribution of X from

$$\begin{aligned} \tilde{U} = F(X) &= 1 - \lambda^{x+1} \\ 1 - \tilde{U} &= \lambda^{x+1} \\ \log(1 - \tilde{U}) &= (x + 1) \log \lambda \\ \frac{\log(1 - \tilde{U})}{\log \lambda} &= x + 1 \\ \frac{\log(1 - \tilde{U})}{\log \lambda} - 1 &= x \end{aligned}$$

However, $\tilde{U} = F^{-1}(X)$ may not be an integer for any $x \geq 0$.

So let

$$x = \lceil \frac{\log(1 - \tilde{U})}{\log \lambda} - 1 \rceil$$

where $\lceil . \rceil$ is the ceiling function.

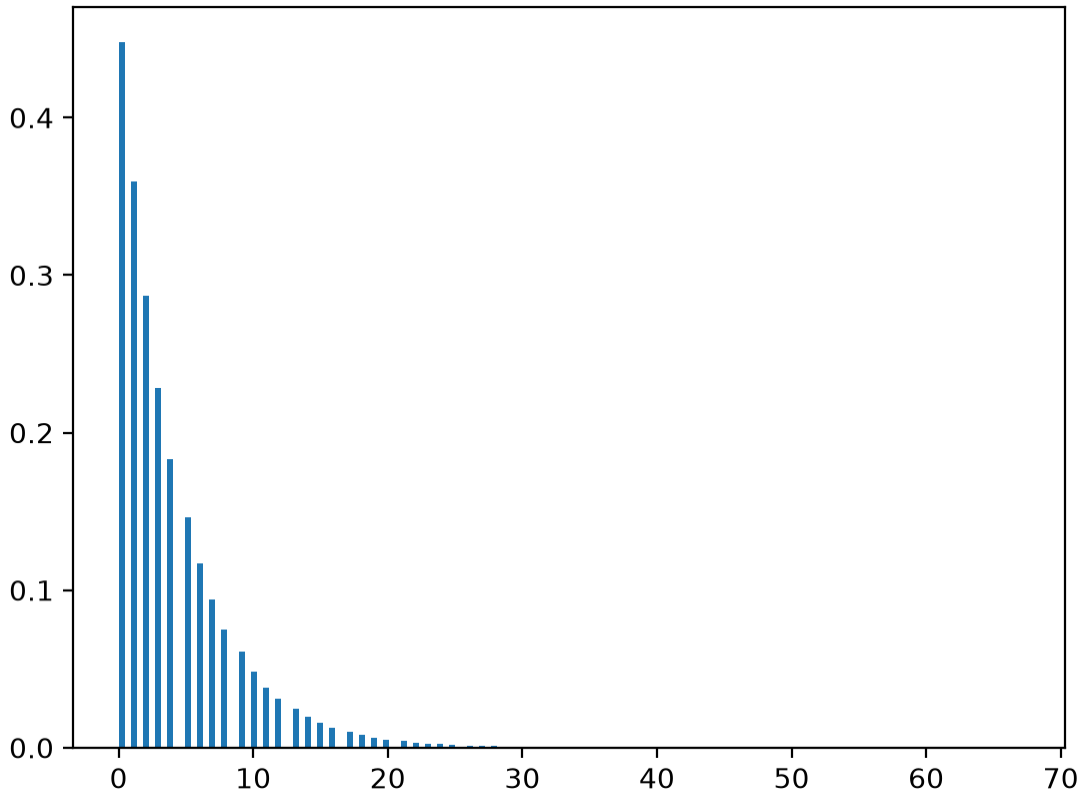
Thus x is the smallest integer such that the discrete geometric CDF is greater than or equal to $\tilde{U}$.

We can verify that x is indeed geometrically distributed by the following `numpy` program.

Note

The exponential distribution is the continuous analog of geometric distribution.

```
n, λ = 1_000_000, 0.8

# draw uniform numbers
u = rng.random(n)

# transform
x = np.ceil(np.log(1-u)/np.log(λ) - 1)

# draw geometric distributions
x_g = rng.geometric(1-λ, n)

# plot and compare
plt.hist(x, bins=150, density=True)
plt.show()
```



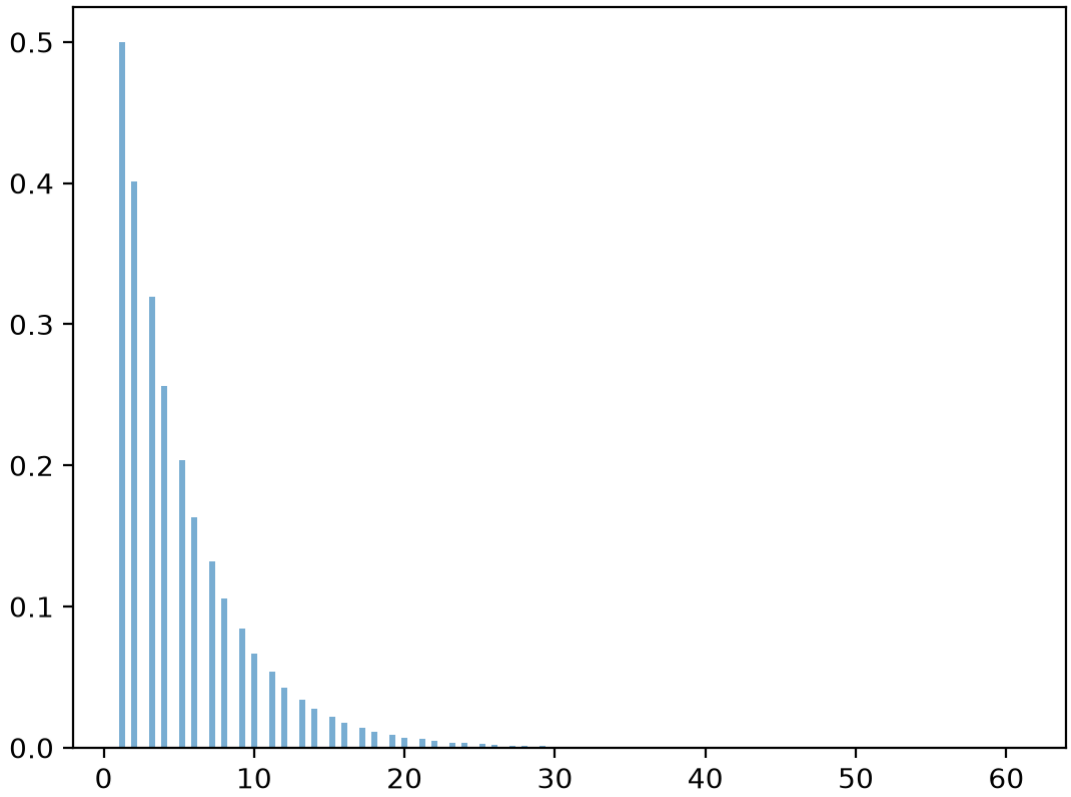

```
rng.geometric(1-λ, n).max()
```

```
np.int64(67)
```

```
np.log(0.4)/np.log(0.3)
```

```
np.float64(0.7610560044063083)
```

```
plt.hist(x_g, bins=150, density=True, alpha=0.6)
plt.show()
```


0.5
0.4
0.3
0.2
0.1
0.0
0
10
20
30
40
50
60

CHAPTER

TEN

LLN AND CLT

Contents

- *LLN and CLT*
  - *Overview*
  - *Relationships*
  - *LLN*
  - *CLT*
  - *Exercises*

10.1 Overview

This lecture illustrates two of the most important theorems of probability and statistics: The law of large numbers (LLN) and the central limit theorem (CLT).

These beautiful theorems lie behind many of the most fundamental results in econometrics and quantitative economic modeling.

The lecture is based around simulations that show the LLN and CLT in action.

We also demonstrate how the LLN and CLT break down when the assumptions they are based on do not hold.

In addition, we examine several useful extensions of the classical theorems, such as

- The delta method, for smooth functions of random variables, and
- the multivariate case.

Some of these extensions are presented as exercises.

We'll need the following imports:

```
import matplotlib.pyplot as plt
import random
import numpy as np
from scipy.stats import t, beta, lognorm, expon, gamma, uniform
from scipy.stats import gaussian_kde, poisson, binom, norm, chi2
from mpl_toolkits.mplot3d import Axes3D
```

(continues on next page)

(continued from previous page)

```
from matplotlib.collections import PolyCollection
from scipy.linalg import inv, sqrtm
```

10.2 Relationships

The CLT refines the LLN.

The LLN gives conditions under which sample moments converge to population moments as sample size increases.

The CLT provides information about the rate at which sample moments converge to population moments as sample size increases.

10.3 LLN

We begin with the law of large numbers, which tells us when sample averages will converge to their population means.

10.3.1 The Classical LLN

The classical law of large numbers concerns independent and identically distributed (IID) random variables.

Here is the strongest version of the classical LLN, known as **Kolmogorov's strong law**.

Let $X_1, \ldots, X_n$ be independent and identically distributed scalar random variables, with common distribution F.

When it exists, let μ denote the common mean of this sample:

$$\mu := \mathbb{E}X = \int x F(dx)$$

In addition, let

$$\bar{X}_n := \frac{1}{n} \sum_{i=1}^{n} X_i$$

Kolmogorov's strong law states that, if $\mathbb{E}|X|$ is finite, then

$$\mathbb{P}\left\{\bar{X}_n \to \mu \text{ as } n \to \infty\right\} = 1 \tag{10.1}$$

What does this last expression mean?

Let's think about it from a simulation perspective, imagining for a moment that our computer can generate perfect random samples (which of course it can't).

Let's also imagine that we can generate infinite sequences so that the statement $\bar{X}_n \to \mu$ can be evaluated.

In this setting, (10.1) should be interpreted as meaning that the probability of the computer producing a sequence where $\bar{X}_n \to \mu$ fails to occur is zero.

10.3.2 Proof

The proof of Kolmogorov's strong law is nontrivial – see, for example, theorem 8.3.5 of [Dudley, 2002].

On the other hand, we can prove a weaker version of the LLN very easily and still get most of the intuition.

The version we prove is as follows: If $X_1, \ldots, X_n$ is IID with $\mathbb{E}X_i^2 < \infty$, then, for any $\epsilon > 0$, we have

$$\mathbb{P}\left\{|\bar{X}_n - \mu| \geq \epsilon\right\} \to 0 \quad \text{as} \quad n \to \infty \tag{10.2}$$

(This version is weaker because we claim only convergence in probability rather than almost sure convergence, and assume a finite second moment)

To see that this is so, fix $\epsilon > 0$, and let σ^2 be the variance of each X_i.

Recall the Chebyshev inequality, which tells us that

$$\mathbb{P}\left\{|\bar{X}_n - \mu| \geq \epsilon\right\} \leq \frac{\mathbb{E}[(\bar{X}_n - \mu)^2]}{\epsilon^2} \tag{10.3}$$

Now observe that

$$\begin{aligned}
\mathbb{E}[(\bar{X}_n - \mu)^2] &= \mathbb{E}\left\{\left[\frac{1}{n}\sum_{i=1}^{n}(X_i - \mu)\right]^2\right\} \\
&= \frac{1}{n^2}\sum_{i=1}^{n}\sum_{j=1}^{n}\mathbb{E}(X_i - \mu)(X_j - \mu) \\
&= \frac{1}{n^2}\sum_{i=1}^{n}\mathbb{E}(X_i - \mu)^2 \\
&= \frac{\sigma^2}{n}
\end{aligned}$$

Here the crucial step is at the third equality, which follows from independence.

Independence means that if $i \neq j$, then the covariance term $\mathbb{E}(X_i - \mu)(X_j - \mu)$ drops out.

As a result, $n^2 - n$ terms vanish, leading us to a final expression that goes to zero in n.

Combining our last result with (10.3), we come to the estimate

$$\mathbb{P}\left\{|\bar{X}_n - \mu| \geq \epsilon\right\} \leq \frac{\sigma^2}{n\epsilon^2} \tag{10.4}$$

The claim in (10.2) is now clear.

Of course, if the sequence $X_1, \ldots, X_n$ is correlated, then the cross-product terms $\mathbb{E}(X_i - \mu)(X_j - \mu)$ are not necessarily zero.

While this doesn't mean that the same line of argument is impossible, it does mean that if we want a similar result then the covariances should be "almost zero" for "most" of these terms.

In a long sequence, this would be true if, for example, $\mathbb{E}(X_i - \mu)(X_j - \mu)$ approached zero when the difference between i and j became large.

In other words, the LLN can still work if the sequence $X_1, \ldots, X_n$ has a kind of "asymptotic independence", in the sense that correlation falls to zero as variables become further apart in the sequence.

This idea is very important in time series analysis, and we'll come across it again soon enough.

10.3.3 Illustration

Let's now illustrate the classical IID law of large numbers using simulation.

In particular, we aim to generate some sequences of IID random variables and plot the evolution of $\bar{X}_n$ as n increases.

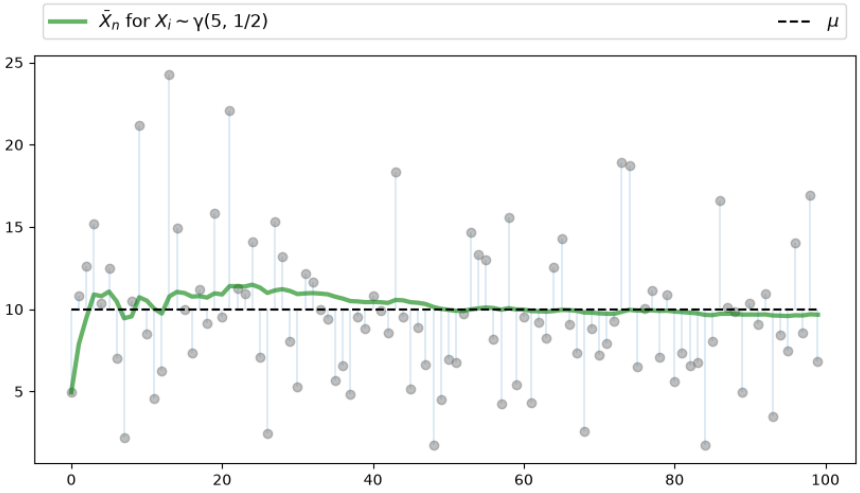
Below is a figure that does just this (as usual, you can click on it to expand it).

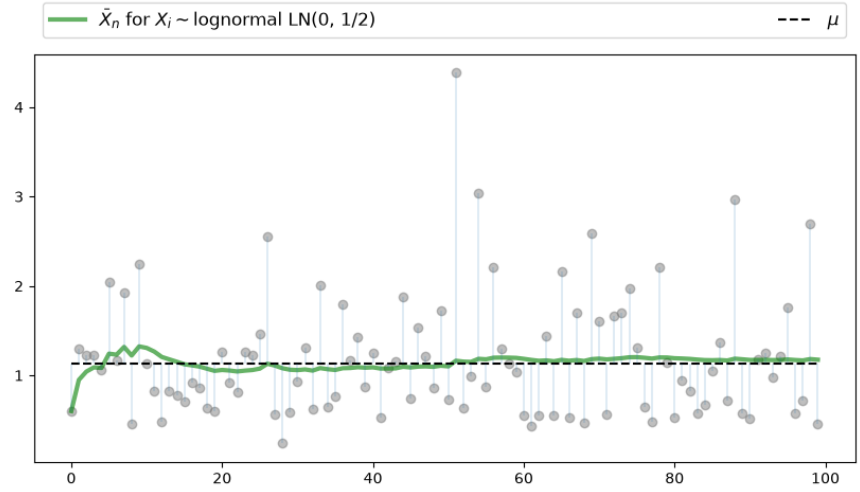
It shows IID observations from three different distributions and plots $\bar{X}_n$ against n in each case.

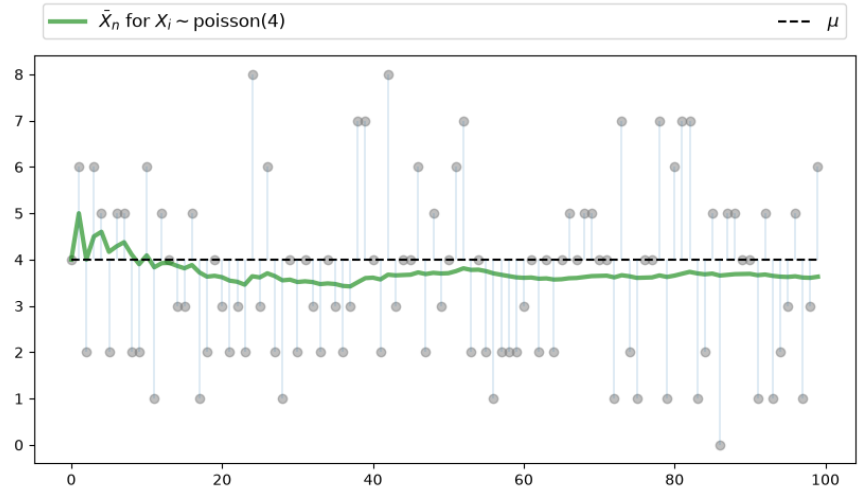
The dots represent the underlying observations X_i for $i = 1, \ldots, 100$.

In each of the three cases, convergence of $\bar{X}_n$ to μ occurs as predicted

```
n = 100

# Arbitrary collection of distributions
distributions = {"student's t with 10 degrees of freedom": t(10),
                 "β(2, 2)": beta(2, 2),
                 "lognormal LN(0, 1/2)": lognorm(0.5),
                 "γ(5, 1/2)": gamma(5, scale=2),
                 "poisson(4)": poisson(4),
                 "exponential with λ = 1": expon(1)}

# Create a figure and some axes
num_plots = 3
fig, axes = plt.subplots(num_plots, 1, figsize=(10, 20))

# Set some plotting parameters to improve layout
bbox = (0., 1.02, 1., .102)
legend_args = {'ncol': 2,
               'bbox_to_anchor': bbox,
               'loc': 3,
               'mode': 'expand'}
plt.subplots_adjust(hspace=0.5)

for ax in axes:
    # Choose a randomly selected distribution
    name = random.choice(list(distributions.keys()))
    distribution = distributions.pop(name)

    # Generate n draws from the distribution
    data = distribution.rvs(n)

    # Compute sample mean at each n
    sample_mean = np.empty(n)
    for i in range(n):
        sample_mean[i] = np.mean(data[:i+1])

    # Plot
    ax.plot(list(range(n)), data, 'o', color='grey', alpha=0.5)
    axlabel = r'$\bar{X}_n$ for $X_i \sim$' + name
    ax.plot(list(range(n)), sample_mean, 'g-', lw=3, alpha=0.6, label=axlabel)
    m = distribution.mean()
    ax.plot(list(range(n)), [m] * n, 'k--', lw=1.5, label=r'$\mu$')
    ax.vlines(list(range(n)), m, data, lw=0.2)
    ax.legend(**legend_args, fontsize=12)

plt.show()
```


$\bar{X}_n$ for $X_i \sim \gamma(5, 1/2)$
μ
25
20
15
10
5
0
20
40
60
80
100


$\bar{X}_n$ for $X_i \sim$ lognormal LN(0, 1/2)
μ
4
3
2
1
0
20
40
60
80
100


$\bar{X}_n$ for $X_i \sim$ poisson(4)
μ
8
7
6
5
4
3
2
1
0
20
40
60
80
100

The three distributions are chosen at random from a selection stored in the dictionary `distributions`.

10.4 CLT

Next, we turn to the central limit theorem, which tells us about the distribution of the deviation between sample averages and population means.

10.4.1 Statement of the Theorem

The central limit theorem is one of the most remarkable results in all of mathematics.

In the classical IID setting, it tells us the following:

If the sequence $X_1, \ldots, X_n$ is IID, with common mean μ and common variance $\sigma^2 \in (0, \infty)$, then

$$\sqrt{n}(\bar{X}_n - \mu) \xrightarrow{d} N(0, \sigma^2) \quad \text{as} \quad n \to \infty \tag{10.5}$$

Here $\xrightarrow{d} N(0, \sigma^2)$ indicates convergence in distribution to a centered (i.e, zero mean) normal with standard deviation σ.

10.4.2 Intuition

The striking implication of the CLT is that for **any** distribution with finite second moment, the simple operation of adding independent copies **always** leads to a Gaussian curve.

A relatively simple proof of the central limit theorem can be obtained by working with characteristic functions (see, e.g., theorem 9.5.6 of [Dudley, 2002]).

The proof is elegant but almost anticlimactic, and it provides surprisingly little intuition.

In fact, all of the proofs of the CLT that we know are similar in this respect.

Why does adding independent copies produce a bell-shaped distribution?

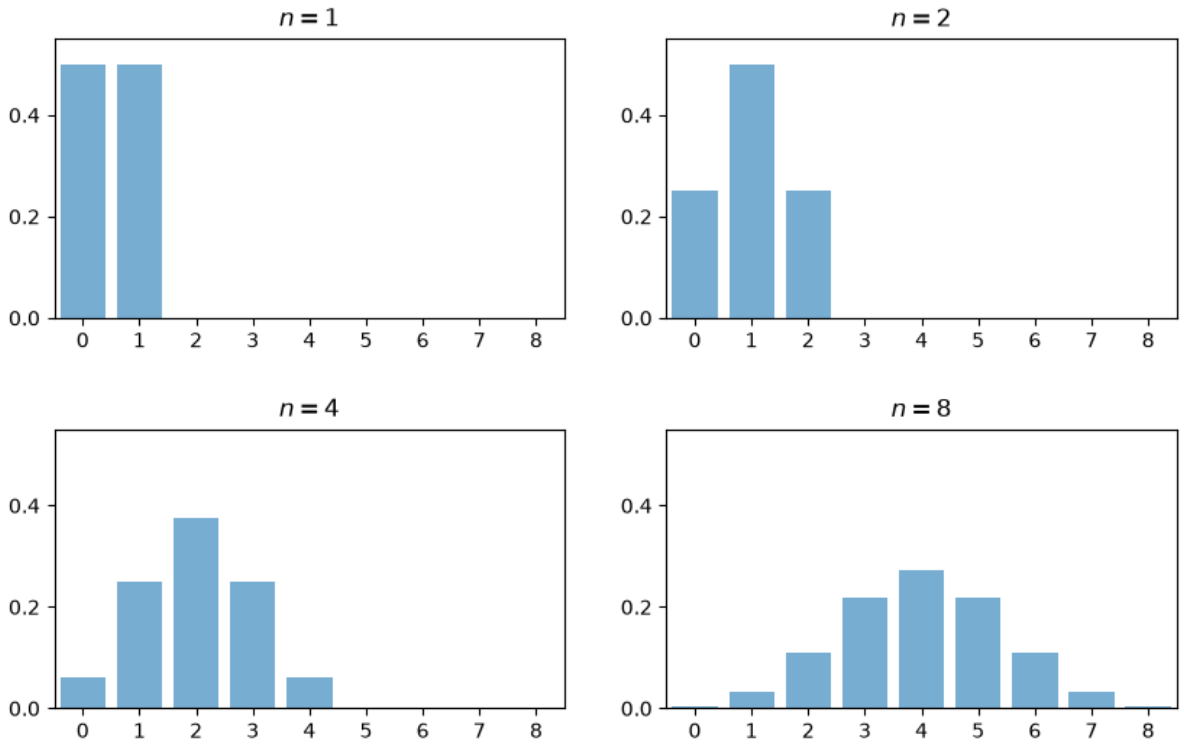
Part of the answer can be obtained by investigating the addition of independent Bernoulli random variables.

In particular, let X_i be binary, with $\mathbb{P}\{X_i = 0\} = \mathbb{P}\{X_i = 1\} = 0.5$, and let $X_1, \ldots, X_n$ be independent.

Think of $X_i = 1$ as a "success", so that $Y_n = \sum_{i=1}^n X_i$ is the number of successes in n trials.

The next figure plots the probability mass function of Y_n for $n = 1, 2, 4, 8$

```
fig, axes = plt.subplots(2, 2, figsize=(10, 6))
plt.subplots_adjust(hspace=0.4)
axes = axes.flatten()
ns = [1, 2, 4, 8]
dom = list(range(9))

for ax, n in zip(axes, ns):
    b = binom(n, 0.5)
    ax.bar(dom, b.pmf(dom), alpha=0.6, align='center')
    ax.set(xlim=(-0.5, 8.5), ylim=(0, 0.55),
           xticks=list(range(9)), yticks=(0, 0.2, 0.4),
           title=f'$n = {n}$')

plt.show()
```



When $n = 1$, the distribution is flat — one success or no successes have the same probability.

When $n = 2$ we can either have 0, 1 or 2 successes.

Notice the peak in probability mass at the mid-point $k = 1$.

The reason is that there are more ways to get 1 success ("fail then succeed" or "succeed then fail") than to get zero or two successes.

Moreover, the two trials are independent, so the outcomes "fail then succeed" and "succeed then fail" are just as likely as the outcomes "fail then fail" and "succeed then succeed".

(If there was positive correlation, say, then "succeed then fail" would be less likely than "succeed then succeed")

Here, already we have the essence of the CLT: addition under independence leads probability mass to pile up in the middle and thin out at the tails.

For $n = 4$ and $n = 8$ we again get a peak at the "middle" value (halfway between the minimum and the maximum possible value).

The intuition is the same — there are simply more ways to get these middle outcomes.

If we continue, the bell-shaped curve becomes even more pronounced.

We are witnessing the binomial approximation of the normal distribution.

10.4.3 Simulation 1

Since the CLT seems almost magical, running simulations that verify its implications is one good way to build intuition.

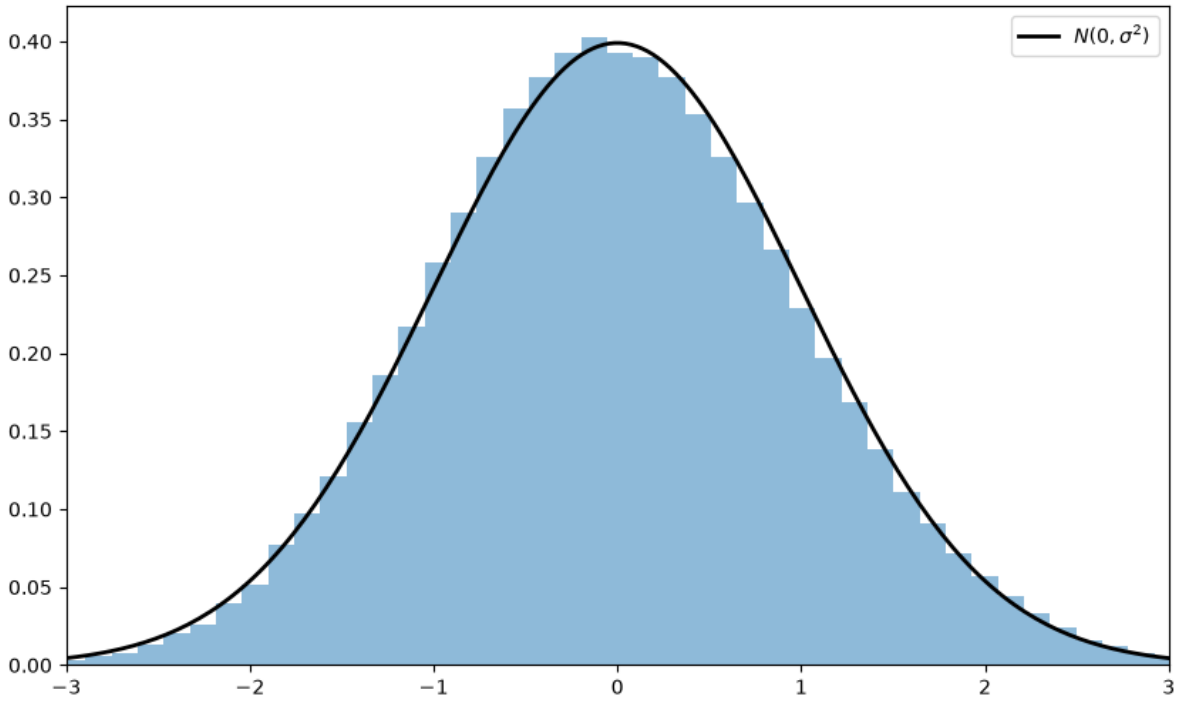
To this end, we now perform the following simulation

1. Choose an arbitrary distribution F for the underlying observations X_i.
2. Generate independent draws of $Y_n := \sqrt{n}(\bar{X}_n - \mu)$.
3. Use these draws to compute some measure of their distribution — such as a histogram.
4. Compare the latter to $N(0, \sigma^2)$.

Here's some code that does exactly this for the exponential distribution $F(x) = 1 - e^{-\lambda x}$.

(Please experiment with other choices of F, but remember that, to conform with the conditions of the CLT, the distribution must have a finite second moment.)

```
# Set parameters
n = 250                  # Choice of n
k = 100000               # Number of draws of Y_n
distribution = expon(2)  # Exponential distribution, λ = 1/2
μ, s = distribution.mean(), distribution.std()

# Draw underlying RVs. Each row contains a draw of X_1,..,X_n
data = distribution.rvs((k, n))
# Compute mean of each row, producing k draws of \bar X_n
sample_means = data.mean(axis=1)
# Generate observations of Y_n
Y = np.sqrt(n) * (sample_means - μ)

# Plot
fig, ax = plt.subplots(figsize=(10, 6))
xmin, xmax = -3 * s, 3 * s
ax.set_xlim(xmin, xmax)
ax.hist(Y, bins=60, alpha=0.5, density=True)
xgrid = np.linspace(xmin, xmax, 200)
ax.plot(xgrid, norm.pdf(xgrid, scale=s), 'k-', lw=2, label=r'$N(0, \sigma^2)$')
ax.legend()

plt.show()
```



Notice the absence of for loops — every operation is vectorized, meaning that the major calculations are all shifted to highly optimized C code.

The fit to the normal density is already tight and can be further improved by increasing `n`.

You can also experiment with other specifications of F.

10.4.4 Simulation 2

Our next simulation is somewhat like the first, except that we aim to track the distribution of $Y_n := \sqrt{n}(\bar{X}_n - \mu)$ as n increases.

In the simulation, we'll be working with random variables having $\mu = 0$.

Thus, when $n = 1$, we have $Y_1 = X_1$, so the first distribution is just the distribution of the underlying random variable.

For $n = 2$, the distribution of Y_2 is that of $(X_1 + X_2)/\sqrt{2}$, and so on.

What we expect is that, regardless of the distribution of the underlying random variable, the distribution of Y_n will smooth out into a bell-shaped curve.

The next figure shows this process for $X_i \sim f$, where f was specified as the convex combination of three different beta densities.

(Taking a convex combination is an easy way to produce an irregular shape for f.)

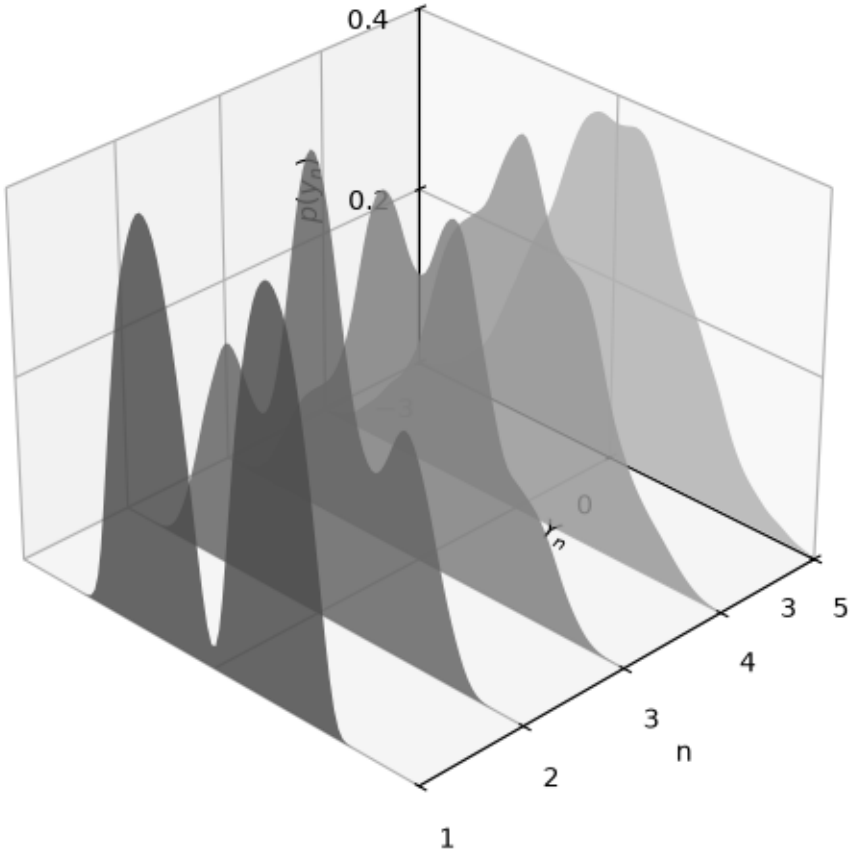
In the figure, the closest density is that of Y_1, while the furthest is that of Y_5

```
beta_dist = beta(2, 2)

def gen_x_draws(k, rng):
    """
    Returns a flat array containing k independent draws from the
    distribution of X, the underlying random variable.  This distribution
```

(continues on next page)

(continued from previous page)

```
    is itself a convex combination of three beta distributions.
    """
    bdraws = beta_dist.rvs((3, k))
    # Transform rows, so each represents a different distribution
    bdraws[0, :] -= 0.5
    bdraws[1, :] += 0.6
    bdraws[2, :] -= 1.1
    # Set X[i] = bdraws[j, i], where j is a random draw from {0, 1, 2}
    js = rng.integers(0, 2, size=k)
    X = bdraws[js, np.arange(k)]
    # Rescale, so that the random variable is zero mean
    m, sigma = X.mean(), X.std()
    return (X - m) / sigma

nmax = 5
reps = 100000
ns = list(range(1, nmax + 1))
rng = np.random.default_rng()

# Form a matrix Z such that each column is reps independent draws of X
Z = np.empty((reps, nmax))
for i in range(nmax):
    Z[:, i] = gen_x_draws(reps, rng)
# Take cumulative sum across columns
S = Z.cumsum(axis=1)
# Multiply j-th column by sqrt j
Y = (1 / np.sqrt(ns)) * S

# Plot
ax = plt.figure(figsize = (10, 6)).add_subplot(projection='3d')

a, b = -3, 3
gs = 100
xs = np.linspace(a, b, gs)

# Build verts
greys = np.linspace(0.3, 0.7, nmax)
verts = []
for n in ns:
    density = gaussian_kde(Y[:, n-1])
    ys = density(xs)
    verts.append(list(zip(xs, ys)))

poly = PolyCollection(verts, facecolors=[str(g) for g in greys])
poly.set_alpha(0.85)
ax.add_collection3d(poly, zs=ns, zdir='x')

ax.set(xlim3d=(1, nmax), xticks=(ns), ylabel='$Y_n$', zlabel='$p(y_n)$',
       xlabel=("n"), yticks=((-3, 0, 3)), ylim3d=(a, b),
       zlim3d=(0, 0.4), zticks=((0.2, 0.4)))
ax.invert_xaxis()
# Rotates the plot 30 deg on z axis and 45 deg on x axis
ax.view_init(30, 45)
plt.show()
```



As expected, the distribution smooths out into a bell curve as n increases.

We leave you to investigate its contents if you wish to know more.

If you run the file from the ordinary IPython shell, the figure should pop up in a window that you can rotate with your mouse, giving different views on the density sequence.

10.4.5 The Multivariate Case

The law of large numbers and central limit theorem work just as nicely in multidimensional settings.

To state the results, let's recall some elementary facts about random vectors.

A random vector $\mathbf{X}$ is just a sequence of k random variables $(X_1, \ldots, X_k)$.

Each realization of $\mathbf{X}$ is an element of $\mathbb{R}^k$.

A collection of random vectors $\mathbf{X}_1, \ldots, \mathbf{X}_n$ is called independent if, given any n vectors $\mathbf{x}_1, \ldots, \mathbf{x}_n$ in $\mathbb{R}^k$, we have

$$\mathbb{P}\{\mathbf{X}_1 \leq \mathbf{x}_1, \ldots, \mathbf{X}_n \leq \mathbf{x}_n\} = \mathbb{P}\{\mathbf{X}_1 \leq \mathbf{x}_1\} \times \cdots \times \mathbb{P}\{\mathbf{X}_n \leq \mathbf{x}_n\}$$

(The vector inequality $\mathbf{X} \leq \mathbf{x}$ means that $X_j \leq x_j$ for $j = 1, \ldots, k$)

Let $\mu_j := \mathbb{E}[X_j]$ for all $j = 1, \ldots, k$.

The expectation $\mathbb{E}[\mathbf{X}]$ of $\mathbf{X}$ is defined to be the vector of expectations:

$$\mathbb{E}[\mathbf{X}] := \begin{pmatrix} \mathbb{E}[X_1] \\ \mathbb{E}[X_2] \\ \vdots \\ \mathbb{E}[X_k] \end{pmatrix} = \begin{pmatrix} \mu_1 \\ \mu_2 \\ \vdots \\ \mu_k \end{pmatrix} =: \mu$$

The **variance-covariance matrix** of random vector $\mathbf{X}$ is defined as

$$\text{Var}[\mathbf{X}] := \mathbb{E}[(\mathbf{X} - \mu)(\mathbf{X} - \mu)']$$

Expanding this out, we get

$$\text{Var}[\mathbf{X}] = \begin{pmatrix} \mathbb{E}[(X_1 - \mu_1)(X_1 - \mu_1)] & \cdots & \mathbb{E}[(X_1 - \mu_1)(X_k - \mu_k)] \\ \mathbb{E}[(X_2 - \mu_2)(X_1 - \mu_1)] & \cdots & \mathbb{E}[(X_2 - \mu_2)(X_k - \mu_k)] \\ \vdots & \vdots & \vdots \\ \mathbb{E}[(X_k - \mu_k)(X_1 - \mu_1)] & \cdots & \mathbb{E}[(X_k - \mu_k)(X_k - \mu_k)] \end{pmatrix}$$

The j, k-th term is the scalar covariance between X_j and X_k.

With this notation, we can proceed to the multivariate LLN and CLT.

Let $\mathbf{X}_1, \ldots, \mathbf{X}_n$ be a sequence of independent and identically distributed random vectors, each one taking values in $\mathbb{R}^k$.

Let μ be the vector $\mathbb{E}[\mathbf{X}_i]$, and let Σ be the variance-covariance matrix of $\mathbf{X}_i$.

Interpreting vector addition and scalar multiplication in the usual way (i.e., pointwise), let

$$\bar{\mathbf{X}}_n := \frac{1}{n} \sum_{i=1}^{n} \mathbf{X}_i$$

In this setting, the LLN tells us that

$$\mathbb{P}\left\{\bar{\mathbf{X}}_n \to \mu \text{ as } n \to \infty\right\} = 1 \tag{10.6}$$

Here $\bar{\mathbf{X}}_n \to \mu$ means that $\|\bar{\mathbf{X}}_n - \mu\| \to 0$, where $\|\cdot\|$ is the standard Euclidean norm.

The CLT tells us that, provided Σ is finite,

$$\sqrt{n}(\bar{\mathbf{X}}_n - \mu) \xrightarrow{d} N(\mathbf{0}, \Sigma) \quad \text{as} \quad n \to \infty \tag{10.7}$$

10.5 Exercises

Exercise 10.5.1

One very useful consequence of the central limit theorem is as follows.

Assume the conditions of the CLT as *stated above*.

If $g\colon \mathbb{R} \to \mathbb{R}$ is differentiable at μ and $g'(\mu) \neq 0$, then

$$\sqrt{n}\{g(\bar{X}_n) - g(\mu)\} \xrightarrow{d} N(0, g'(\mu)^2 \sigma^2) \quad \text{as} \quad n \to \infty \tag{10.8}$$

This theorem is used frequently in statistics to obtain the asymptotic distribution of estimators — many of which can be expressed as functions of sample means.

(These kinds of results are often said to use the "delta method".)

The proof is based on a Taylor expansion of g around the point μ.

Taking the result as given, let the distribution F of each X_i be uniform on $[0, \pi/2]$ and let $g(x) = \sin(x)$.

Derive the asymptotic distribution of $\sqrt{n}\{g(\bar{X}_n) - g(\mu)\}$ and illustrate convergence in the same spirit as the program discussed *above*.

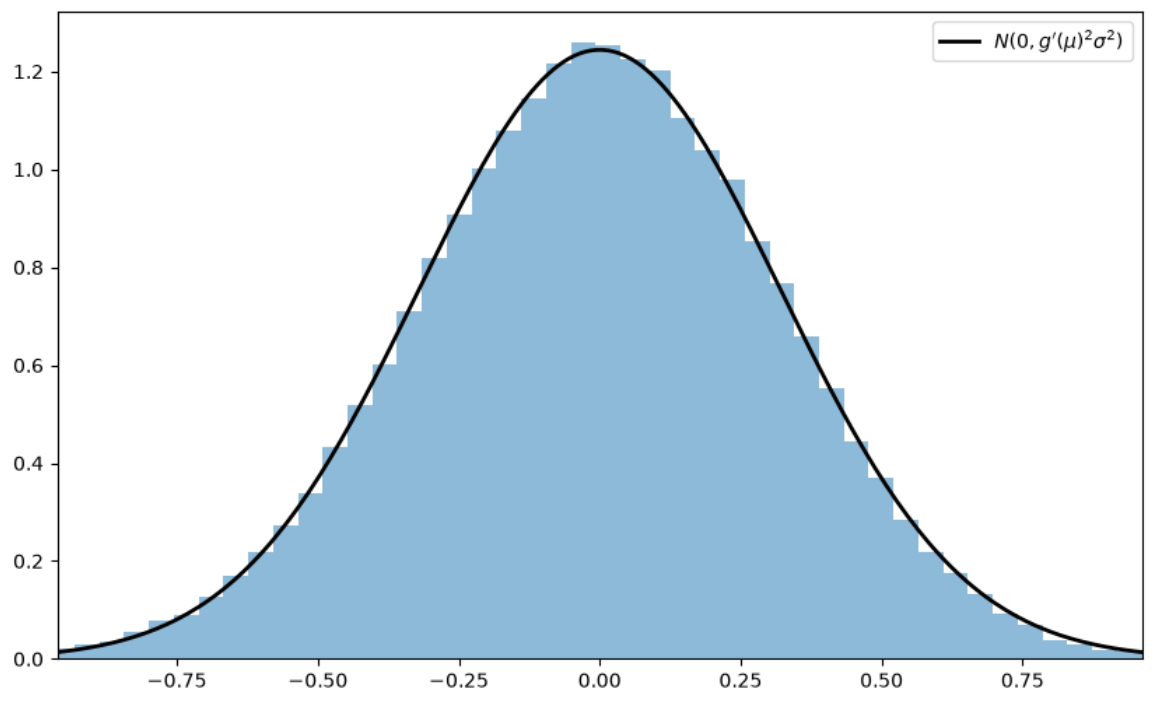
What happens when you replace $[0, \pi/2]$ with $[0, \pi]$?

What is the source of the problem?

Solution

Here is one solution

```
"""
Illustrates the delta method, a consequence of the central limit theorem.
"""

# Set parameters
n = 250
replications = 100000
distribution = uniform(loc=0, scale=(np.pi / 2))
μ, s = distribution.mean(), distribution.std()

g = np.sin
g_prime = np.cos

# Generate obs of sqrt{n} (g(X_n) - g(μ))
data = distribution.rvs((replications, n))
sample_means = data.mean(axis=1)  # Compute mean of each row
error_obs = np.sqrt(n) * (g(sample_means) - g(μ))

# Plot
asymptotic_sd = g_prime(μ) * s
fig, ax = plt.subplots(figsize=(10, 6))
xmin = -3 * g_prime(μ) * s
xmax = -xmin
ax.set_xlim(xmin, xmax)
ax.hist(error_obs, bins=60, alpha=0.5, density=True)
xgrid = np.linspace(xmin, xmax, 200)
lb = r"$N(0, g'(\mu)^2  \sigma^2)$"
ax.plot(xgrid, norm.pdf(xgrid, scale=asymptotic_sd), 'k-', lw=2, label=lb)
ax.legend()
plt.show()
```



What happens when you replace $[0, \pi/2]$ with $[0, \pi]$?

In this case, the mean μ of this distribution is $\pi/2$, and since $g' = \cos$, we have $g'(\mu) = 0$.

Hence the conditions of the delta theorem are not satisfied.

Exercise 10.5.2

Here's a result that's often used in developing statistical tests, and is connected to the multivariate central limit theorem.

If you study econometric theory, you will see this result used again and again.

Assume the setting of the multivariate CLT *discussed above*, so that

1. $\mathbf{X}_1, \ldots, \mathbf{X}_n$ is a sequence of IID random vectors, each taking values in $\mathbb{R}^k$.
2. $\mu := \mathbb{E}[\mathbf{X}_i]$, and Σ is the variance-covariance matrix of $\mathbf{X}_i$.
3. The convergence

$$\sqrt{n}(\bar{\mathbf{X}}_n - \mu) \xrightarrow{d} N(\mathbf{0}, \Sigma) \tag{10.9}$$

is valid.

In a statistical setting, one often wants the right-hand side to be **standard** normal so that confidence intervals are easily computed.

This normalization can be achieved on the basis of three observations.

First, if $\mathbf{X}$ is a random vector in $\mathbb{R}^k$ and $\mathbf{A}$ is constant and $k \times k$, then

$$\operatorname{Var}[\mathbf{AX}] = \mathbf{A} \operatorname{Var}[\mathbf{X}] \mathbf{A}'$$

Second, by the continuous mapping theorem, if $\mathbf{Z}_n \xrightarrow{d} \mathbf{Z}$ in $\mathbb{R}^k$ and $\mathbf{A}$ is constant and $k \times k$, then

$$\mathbf{A}\mathbf{Z}_n \xrightarrow{d} \mathbf{A}\mathbf{Z}$$

Third, if $\mathbf{S}$ is a $k \times k$ symmetric positive definite matrix, then there exists a symmetric positive definite matrix $\mathbf{Q}$, called the inverse square root of $\mathbf{S}$, such that

$$\mathbf{Q}\mathbf{S}\mathbf{Q}' = \mathbf{I}$$

Here $\mathbf{I}$ is the $k \times k$ identity matrix.

Putting these things together, your first exercise is to show that if $\mathbf{Q}$ is the inverse square root of [?], then

$$\mathbf{Z}_n := \sqrt{n}\mathbf{Q}(\bar{\mathbf{X}}_n - \mu) \xrightarrow{d} \mathbf{Z} \sim N(\mathbf{0}, \mathbf{I})$$

Applying the continuous mapping theorem one more time tells us that

$$\|\mathbf{Z}_n\|^2 \xrightarrow{d} \|\mathbf{Z}\|^2$$

Given the distribution of $\mathbf{Z}$, we conclude that

$$n\|\mathbf{Q}(\bar{\mathbf{X}}_n - \mu)\|^2 \xrightarrow{d} \chi^2(k) \tag{10.10}$$

where $\chi^2(k)$ is the chi-squared distribution with k degrees of freedom.

(Recall that k is the dimension of $\mathbf{X}_i$, the underlying random vectors.)

Your second exercise is to illustrate the convergence in (10.10) with a simulation.

In doing so, let

$$\mathbf{X}_i := \begin{pmatrix} W_i \\ U_i + W_i \end{pmatrix}$$

where

- each W_i is an IID draw from the uniform distribution on $[-1, 1]$.
- each U_i is an IID draw from the uniform distribution on $[-2, 2]$.
- U_i and W_i are independent of each other.

Hint

1. `scipy.linalg.sqrtm(A)` computes the square root of `A`. You still need to invert it.
2. You should be able to work out Σ from the preceding information.

Solution

First we want to verify the claim that

$$\sqrt{n}\mathbf{Q}(\bar{\mathbf{X}}_n - \mu) \xrightarrow{d} N(\mathbf{0}, \mathbf{I})$$

This is straightforward given the facts presented in the exercise.

Let

$$\mathbf{Y}_n := \sqrt{n}(\bar{\mathbf{X}}_n - \mu) \quad \text{and} \quad \mathbf{Y} \sim N(\mathbf{0}, \Sigma)$$

By the multivariate CLT and the continuous mapping theorem, we have

$$\mathbf{QY}_n \xrightarrow{d} \mathbf{QY}$$

Since linear combinations of normal random variables are normal, the vector $\mathbf{QY}$ is also normal.

Its mean is clearly $\mathbf{0}$, and its variance-covariance matrix is

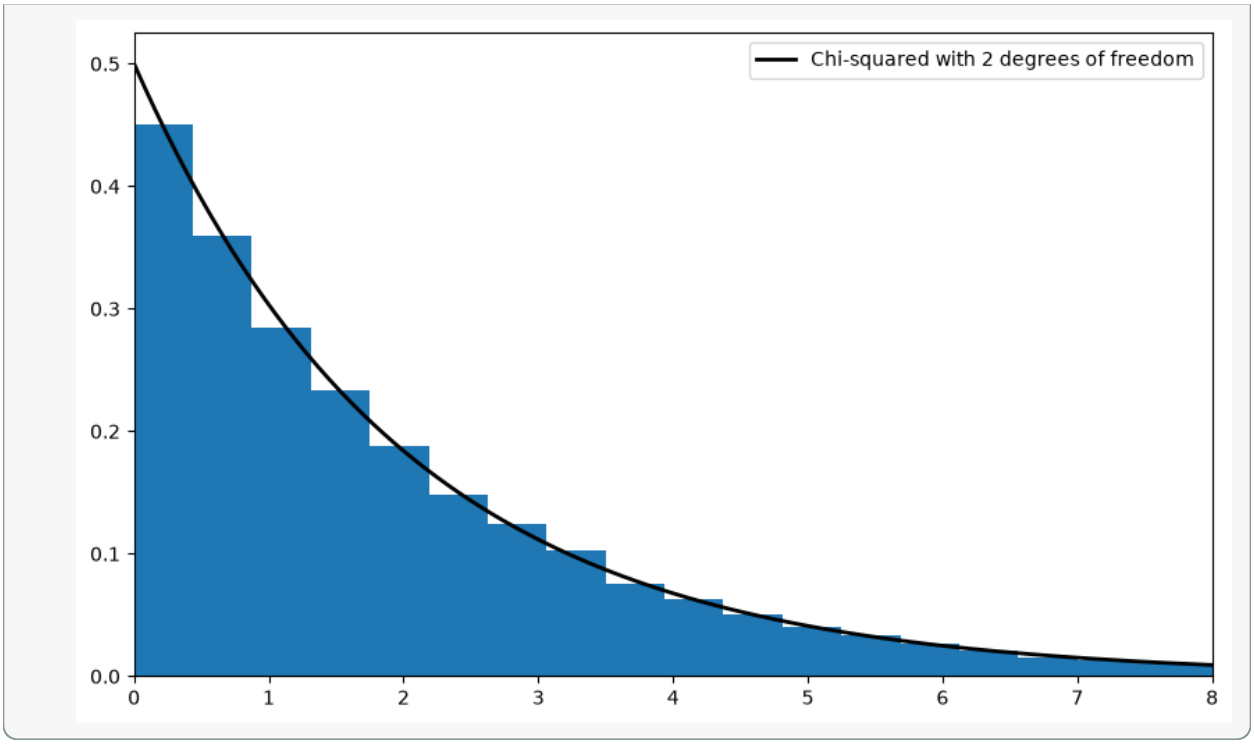
$$\text{Var}[\mathbf{QY}] = \mathbf{Q}\text{Var}[\mathbf{Y}]\mathbf{Q}' = \mathbf{Q}\Sigma\mathbf{Q}' = \mathbf{I}$$

In conclusion, $\mathbf{QY}_n \xrightarrow{d} \mathbf{QY} \sim N(\mathbf{0}, \mathbf{I})$, which is what we aimed to show.

Now we turn to the simulation exercise.

Our solution is as follows

```
# Set parameters
n = 250
replications = 50000
dw = uniform(loc=-1, scale=2)  # Uniform(-1, 1)
du = uniform(loc=-2, scale=4)  # Uniform(-2, 2)
sw, su = dw.std(), du.std()
vw, vu = sw**2, su**2
Σ = ((vw, vw), (vw, vw + vu))
Σ = np.array(Σ)

# Compute Σ^{-1/2}
Q = inv(sqrtm(Σ))

# Generate observations of the normalized sample mean
error_obs = np.empty((2, replications))
for i in range(replications):
    # Generate one sequence of bivariate shocks
    X = np.empty((2, n))
    W = dw.rvs(n)
    U = du.rvs(n)
    # Construct the n observations of the random vector
    X[0, :] = W
    X[1, :] = W + U
    # Construct the i-th observation of Y_n
    error_obs[:, i] = np.sqrt(n) * X.mean(axis=1)

# Premultiply by Q and then take the squared norm
temp = Q @ error_obs
chisq_obs = np.sum(temp**2, axis=0)

# Plot
fig, ax = plt.subplots(figsize=(10, 6))
xmax = 8
ax.set_xlim(0, xmax)
xgrid = np.linspace(0, xmax, 200)
lb = "Chi-squared with 2 degrees of freedom"
ax.plot(xgrid, chi2.pdf(xgrid, 2), 'k-', lw=2, label=lb)
ax.legend()
ax.hist(chisq_obs, bins=50, density=True)
plt.show()
```


Chi-squared with 2 degrees of freedom
0.5
0.4
0.3
0.2
0.1
0.0
0
1
2
3
4
5
6
7
8

CHAPTER

ELEVEN

TWO MEANINGS OF PROBABILITY

11.1 Overview

This lecture illustrates two distinct interpretations of a **probability distribution**

- A frequentist interpretation as **relative frequencies** anticipated to occur in a large IID sample
- A Bayesian interpretation as a **personal opinion** (about a parameter or list of parameters) after seeing a collection of observations

We recommend watching the following two videos before proceeding:

- Hypothesis testing within the frequentist approach
- The Bayesian approach to constructing coverage intervals

After you are familiar with the material in these videos, this lecture uses the Socratic method to help consolidate your understanding of these two meanings of probability.

Along the way we construct a **Bayesian coverage interval** and contrast the question it answers with the frequentist relative-frequency reasoning that opens the lecture.

We do this by inviting you to write some Python code.

It would be especially useful if you tried doing this after each question that we pose for you, before proceeding to read the rest of the lecture.

We provide our own answers as the lecture unfolds, but you'll learn more if you try writing your own code before reading and running ours.

11.1.1 Code for answering questions

To answer our coding questions, we'll start with some imports

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy.stats import binom
import scipy.stats as st
```

Empowered with these Python tools, we'll now explore the two meanings described above.

11.2 Frequentist Interpretation

Consider the following classic example.

The random variable X takes on possible values $k = 0, 1, 2, \ldots, n$ with probabilities

$$p(k \mid \theta) := \mathbb{P}\{X = k \mid \theta\} = \binom{n}{k} \theta^k (1-\theta)^{n-k}$$

where the fixed parameter $\theta \in (0, 1)$.

This is called the binomial distribution.

Here

- θ is the probability that one toss of a coin will be a head, an outcome that we encode as $Y = 1$.
- $1 - \theta$ is the probability that one toss of the coin will be a tail, an outcome that we denote $Y = 0$.
- X is the total number of heads that came up after flipping the coin n times.

Consider the following experiment:

Take I **independent** sequences of n **independent** flips of the coin

Notice the repeated use of the adjective **independent**:

- we use it once to describe that we are drawing n independent times from a **Bernoulli** distribution with parameter θ to arrive at one draw from a **Binomial** distribution with parameters θ, n.
- we use it again to describe that we are then drawing I sequences of n coin draws.

Let $y_h^i \in \{0, 1\}$ be the realized value of Y on the hth flip during the ith sequence of flips.

Let $\sum_{h=1}^n y_h^i$ denote the total number of times heads come up during the ith sequence of n independent coin flips.

Let f_k record the fraction of samples of length n for which $\sum_{h=1}^n y_h^i = k$:

$$f_k^I = \frac{1}{I} \sum_{i=1}^{I} \mathbb{1}\left\{ \sum_{h=1}^{n} y_h^i = k \right\}$$

The probability $p(k \mid \theta)$ answers the following question:

- As I becomes large, in what fraction of I independent draws of n coin flips should we anticipate k heads to occur?

As usual, a law of large numbers justifies this answer.

Exercise 11.2.1

1. Write Python code to compute f_k^I
2. Use your code to compute $f_k^I, k = 0, \ldots, n$ and compare them to $p(k \mid \theta)$ for various values of θ, n and I
3. With the Law of Large Numbers in mind, use your code to describe the relationship between f_k^I and $p(k \mid \theta)$ as I grows

Solution

Here is one solution.

We simulate the coin flips with one function and assemble the comparison table with another.

```
def simulate_head_counts(θ, n, I, seed=1234):
    "Simulate I sequences of n coin flips; return the heads count of each sequence.
 ↪"
    rng = np.random.default_rng(seed)
    Y = (rng.random((I, n)) <= θ).astype(int)
    return Y.sum(axis=1)
def compare_frequencies(θ, n, I, seed=1234):
    "Tabulate theoretical binomial probabilities against simulated frequencies."
    head_counts = simulate_head_counts(θ, n, I, seed)
    rows = [
        (k, binom.pmf(k, n, θ), np.mean(head_counts == k))
        for k in range(n + 1)
    ]
    return pd.DataFrame(
        rows, columns=['k', 'Theoretical', 'Frequentist']
    ).set_index('k')
θ, n, k, I = 0.7, 20, 10, 1_000_000

compare_frequencies(θ, n, I)
```

k	Theoretical	Frequentist
0	3.486784e-11	0.000000
1	1.627166e-09	0.000000
2	3.606885e-08	0.000000
3	5.049639e-07	0.000000
4	5.007558e-06	0.000002
5	3.738977e-05	0.000047
6	2.181070e-04	0.000247
7	1.017833e-03	0.001029
8	3.859282e-03	0.003721
9	1.200665e-02	0.011986
10	3.081708e-02	0.030698
11	6.536957e-02	0.065110
12	1.143967e-01	0.114892
13	1.642620e-01	0.164639
14	1.916390e-01	0.191724
15	1.788631e-01	0.178623
16	1.304210e-01	0.129870
17	7.160367e-02	0.072004
18	2.784587e-02	0.027818
19	6.839337e-03	0.006836
20	7.979227e-04	0.000754

From the table above, can you see the law of large numbers at work?

Let's do some more calculations.

11.2.1 Comparison with different θ

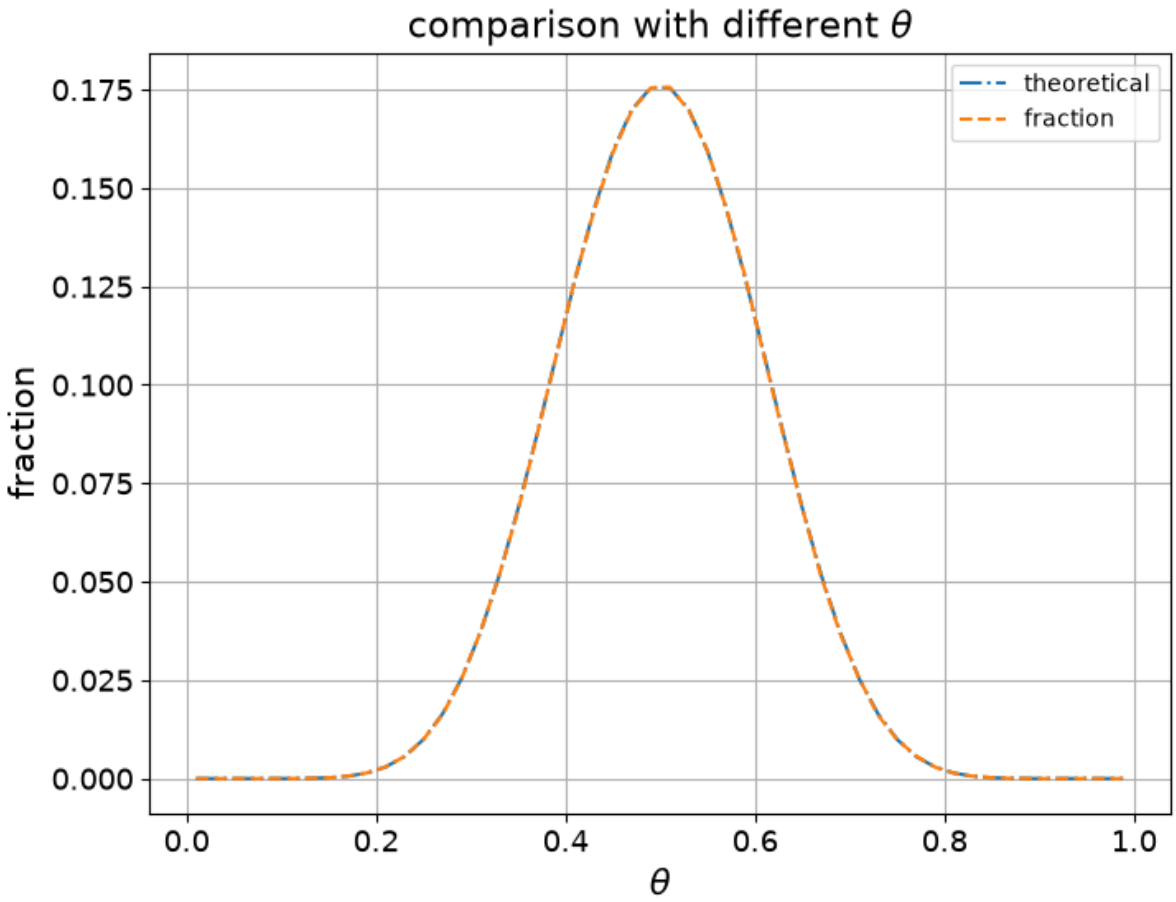
Now we fix

$$n = 20, k = 10, I = 1,000,000$$

We'll vary θ from 0.01 to 0.99 and plot outcomes against θ.

```
θ_low, θ_high, n_thetas = 0.01, 0.99, 50
thetas = np.linspace(θ_low, θ_high, n_thetas)
P = []
f_kI = []
for i in range(n_thetas):
    P.append(binom.pmf(k, n, thetas[i]))
    head_counts = simulate_head_counts(thetas[i], n, I, seed=i)
    f_kI.append(np.mean(head_counts == k))
```

```
fig, ax = plt.subplots(figsize=(8, 6))
ax.grid()
ax.plot(thetas, P, '-.', label='theoretical')
ax.plot(thetas, f_kI, '--', label='fraction')
ax.set_title(r'comparison with different $\theta$',
             fontsize=16)
ax.set_xlabel(r'$\theta$', fontsize=15)
ax.set_ylabel('fraction', fontsize=15)
ax.tick_params(labelsize=13)
ax.legend()
plt.show()
```



11.2.2 Comparison with different n

Now we fix $\theta = 0.7, k = 10, I = 1,000,000$ and vary n from 1 to 100.

Then we'll plot outcomes.

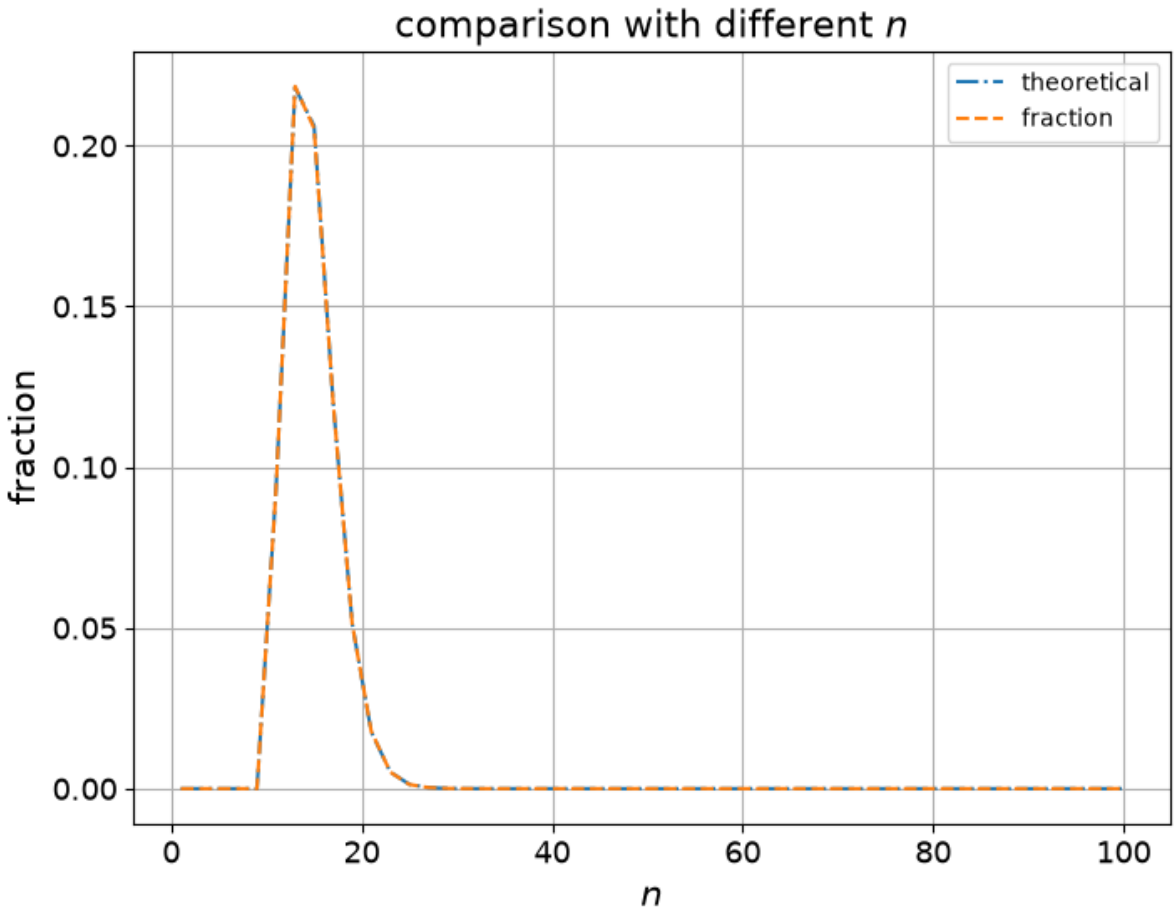
```
n_low, n_high, n_ns = 1, 100, 50
ns = np.linspace(n_low, n_high, n_ns, dtype='int')
P = []
f_kI = []
for i in range(n_ns):
    P.append(binom.pmf(k, ns[i], θ))
    head_counts = simulate_head_counts(θ, ns[i], I, seed=i)
    f_kI.append(np.mean(head_counts == k))
```

```
fig, ax = plt.subplots(figsize=(8, 6))
ax.grid()
ax.plot(ns, P, '-.', label='theoretical')
ax.plot(ns, f_kI, '--', label='fraction')
ax.set_title(r'comparison with different $n$',
             fontsize=16)
ax.set_xlabel(r'$n$', fontsize=15)
ax.set_ylabel('fraction', fontsize=15)
ax.tick_params(labelsize=13)
```

(continues on next page)

(continued from previous page)

```
ax.legend()
plt.show()
```



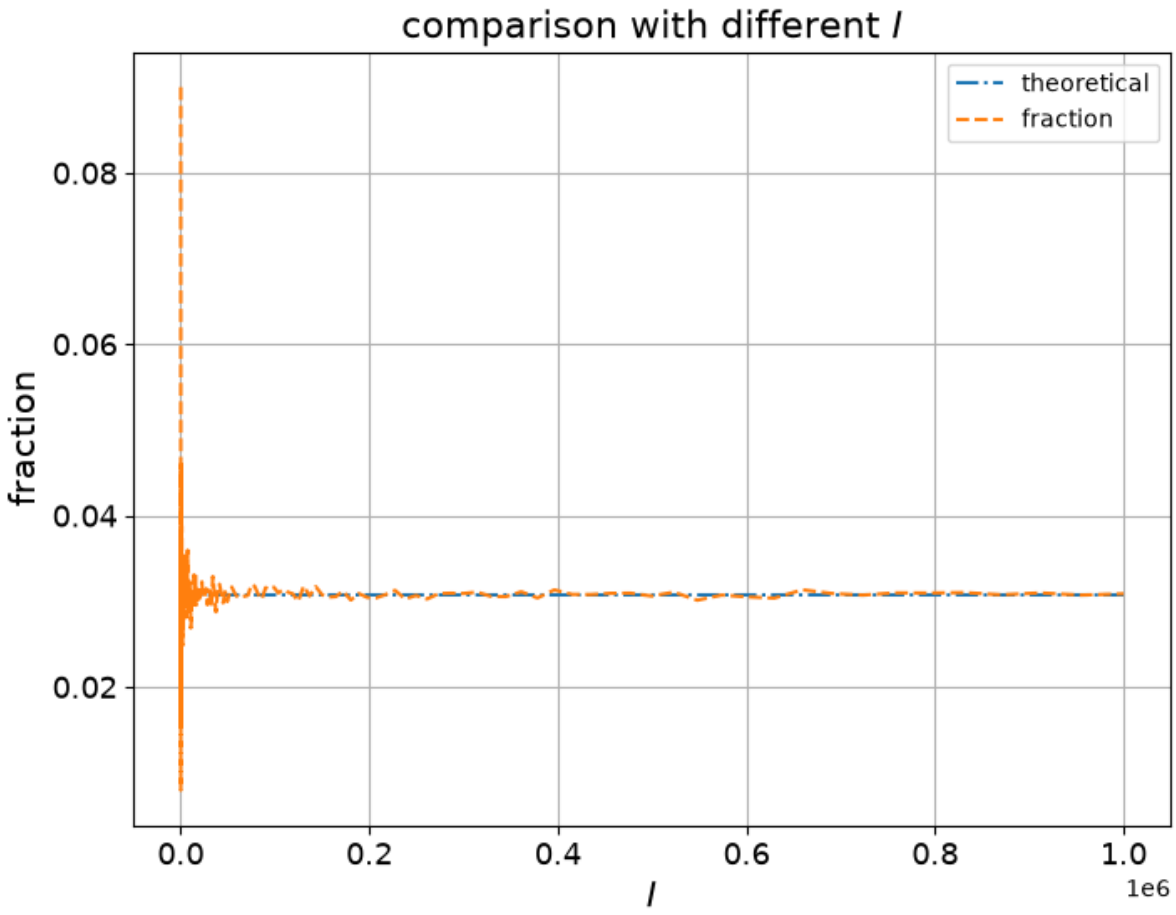
Because $k = 10$ is held fixed, $p(k \mid \theta)$ is zero whenever $n < 10$ — we cannot obtain 10 heads in fewer than 10 flips — and it is largest near $n \approx 14$, where the expected number of heads $n\theta$ is closest to k. The simulated fraction tracks this shape.

11.2.3 Comparison with different I

Now we fix $\theta = 0.7, n = 20, k = 10$ and vary $\log(I)$ from 2 to 6.

```
I_log_low, I_log_high, n_Is = 2, 6, 200
log_Is = np.linspace(I_log_low, I_log_high, n_Is)
Is = np.power(10, log_Is).astype(int)
P = []
f_kI = []
for i in range(n_Is):
    P.append(binom.pmf(k, n, θ))
    head_counts = simulate_head_counts(θ, n, Is[i], seed=i)
    f_kI.append(np.mean(head_counts == k))
```

```
fig, ax = plt.subplots(figsize=(8, 6))
ax.grid()
ax.plot(Is, P, '-.', label='theoretical')
ax.plot(Is, f_kI, '--', label='fraction')
ax.set_title(r'comparison with different $I$',
             fontsize=16)
ax.set_xlabel(r'$I$', fontsize=15)
ax.set_ylabel('fraction', fontsize=15)
ax.tick_params(labelsize=13)
ax.legend()
plt.show()
```



From the above graphs, we can see that I**, the number of independent sequences,** plays an important role.

When I becomes larger, the difference between theoretical probability and frequentist estimate becomes smaller.

Also, as long as I is large enough, changing θ or n does not substantially change the accuracy of the observed fraction as an approximation of $p(k \mid \theta)$.

The Law of Large Numbers is at work here.

For each independent sequence i, define the indicator $\rho_{k,i} = \mathbb{1}\{X_i = k\}$ — that is, $\rho_{k,i}$ equals 1 if the i-th sequence produces exactly k heads and 0 otherwise.

The $\rho_{k,i}$ are IID across i, each with mean $p(k \mid \theta)$ and variance

$$p(k \mid \theta) \cdot (1 - p(k \mid \theta)).$$

So, by the LLN, the average of $\rho_{k,i}$ converges to:

$$\mathbb{E}[\rho_{k,i}] = p(k \mid \theta) = \binom{n}{k} \theta^k (1-\theta)^{n-k}$$

as I goes to infinity.

11.3 Bayesian Interpretation

The likelihood remains binomial, but now we treat θ as a **random variable** rather than a fixed parameter.

So θ is described by a probability distribution.

But now this probability distribution means something different than a relative frequency that we can anticipate to occur in a large IID sample.

Instead, the probability distribution of θ is now a summary of our views about likely values of θ either

- **before** we have seen **any** data at all, or
- **before** we have seen **more** data, after we have seen **some** data

11.3.1 The beta prior and its posterior

Thus, suppose that, before seeing any data, you have a personal prior probability distribution with density

$$p(\theta) = \frac{\theta^{\alpha-1}(1-\theta)^{\beta-1}}{B(\alpha, \beta)}$$

where $B(\alpha, \beta)$ is a **beta function** , so that $p(\theta)$ is the density of a **beta distribution** with parameters α, β.

We can update this prior after observing data using Bayes' Law (see *Probability with Matrices* for an introduction).

For a sample of n coin flips that yields k heads, the **likelihood function** is the binomial PMF $p(k \mid \theta)$ introduced above.

Applying Bayes' Law with our beta prior, the **posterior density** is

$$p(\theta \mid k) = \frac{p(k \mid \theta) \cdot p(\theta)}{\int_0^1 p(k \mid \theta) \cdot p(\theta)\, d\theta}$$

Because the beta prior is conjugate to the binomial likelihood, this integral evaluates to (the kernel of) another beta density, so that

$$\theta \mid k \sim \text{Beta}(\alpha + k,\ \beta + n - k) \tag{11.1}$$

The first exercise below asks you to derive this closed form.

Exercise 11.3.1

a) Write down the **likelihood function** for a single coin flip with outcome $Y \in \{0, 1\}$.

b) Write down the **posterior** distribution for θ after observing that single flip.

c) Derive the closed-form posterior (11.1) for a sample of n flips that yields k heads.

Solution

a) The **likelihood function** for a single coin flip with outcome $Y \in \{0, 1\}$ is

$$p(Y \mid \theta) = \theta^Y (1-\theta)^{1-Y}$$

b) By Bayes' Law, the posterior density for θ after observing a single flip Y is

$$p(\theta \mid Y) = \frac{p(Y \mid \theta) \cdot p(\theta)}{\int_0^1 p(Y \mid \theta) \cdot p(\theta)\, d\theta}$$

Substituting the likelihood from (a) and the beta prior density, this becomes

$$p(\theta \mid Y) = \frac{\theta^Y (1-\theta)^{1-Y} \cdot \theta^{\alpha-1}(1-\theta)^{\beta-1}/B(\alpha,\beta)}{\int_0^1 \theta^Y (1-\theta)^{1-Y} \cdot \theta^{\alpha-1}(1-\theta)^{\beta-1}/B(\alpha,\beta)\, d\theta}$$

Collecting powers of θ and $(1-\theta)$, we recognize the kernel of a beta density:

$$p(\theta \mid Y) = \frac{\theta^{Y+\alpha-1}(1-\theta)^{1-Y+\beta-1}}{\int_0^1 \theta^{Y+\alpha-1}(1-\theta)^{1-Y+\beta-1}\, d\theta}$$

which means that

$$\theta \mid Y \sim \text{Beta}(\alpha + Y,\ \beta + (1-Y))$$

c) The same calculation, with the binomial likelihood in place of the Bernoulli likelihood, generalizes the result to a sample of n flips that yields k heads.

The beta prior contributes the factor $\theta^{\alpha-1}(1-\theta)^{\beta-1}$, and the binomial likelihood contributes $\theta^k(1-\theta)^{n-k}$, so the posterior is proportional to

$$\theta^{\alpha+k-1}(1-\theta)^{\beta+n-k-1},$$

which is the kernel of a beta density. Hence

$$\theta \mid k \sim \text{Beta}(\alpha + k,\ \beta + n - k),$$

as stated in (11.1).

11.3.2 Exploring the posterior numerically

The next exercise puts this posterior to work.

Exercise 11.3.2

a) Now pretend that the true value of $\theta = 0.4$ and that someone who doesn't know this has a beta prior distribution with parameters $\beta = \alpha = 0.5$. Write Python code to simulate this person's personal posterior distribution for θ for a *single* sequence of n draws.

b) Plot the posterior distribution for θ as a function of θ as n grows as $1, 2, \ldots$.

c) For various n's, describe and compute a Bayesian coverage interval for the interval $[0.45, 0.55]$.

d) Tell what question a Bayesian coverage interval answers.

e) Compute the posterior probability that $\theta \in [0.45, 0.55]$ for various values of sample size n.

f) Use your Python code to study what happens to the posterior distribution as $n \to +\infty$, again assuming that the true value of $\theta = 0.4$, though it is unknown to the person doing the updating via Bayes' Law.

Solution

a)

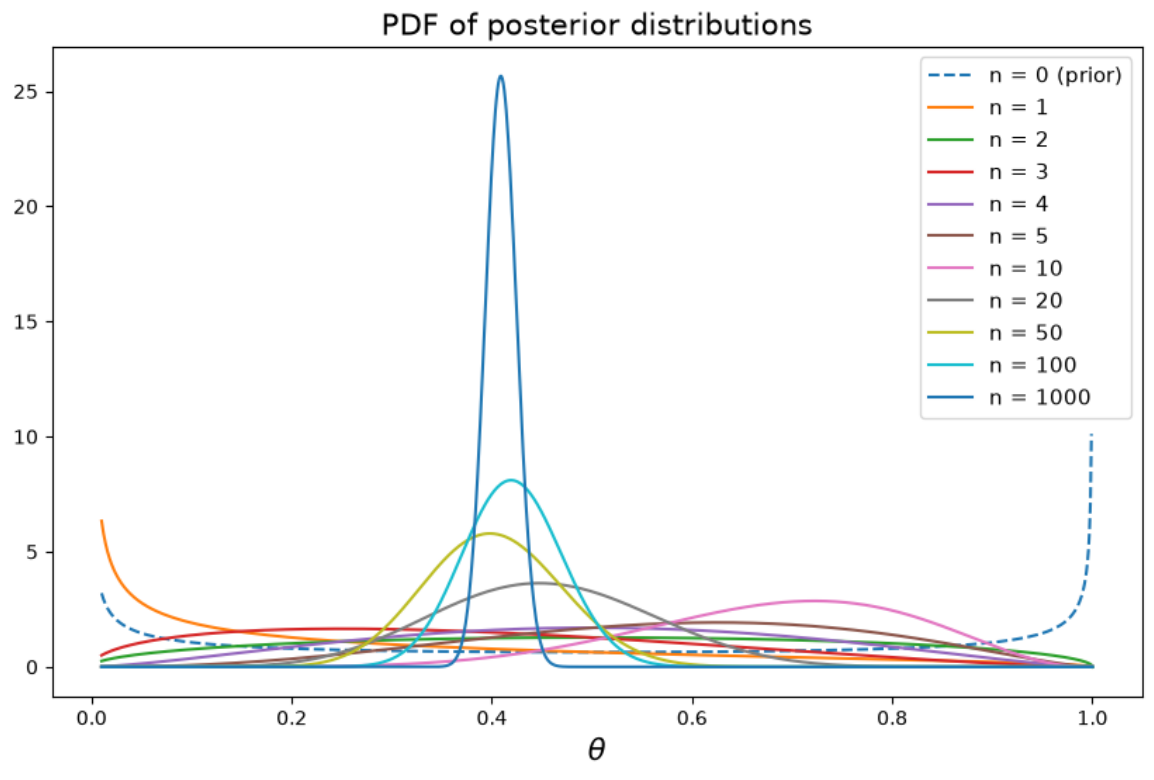
We use one function to simulate a sequence of coin flips and another to form the Beta posterior from the first `n_obs` of those flips.

```
def simulate_flips(θ=0.4, n=1_000_000, seed=1234):
    "Simulate n coin flips, each landing heads (1) with probability θ."
    rng = np.random.default_rng(seed)
    return (rng.random(n) < θ).astype(int)
```

```
def form_posterior(draws, n_obs, α=0.5, β=0.5):
    "Beta posterior for θ from the first n_obs flips, given a Beta(α, β) prior."
    heads = draws[:n_obs].sum()
    return st.beta(α + heads, β + n_obs - heads)
```

b)

```
draws = simulate_flips()

n_obs_list = [1, 2, 3, 4, 5, 10, 20, 50,
              100, 1000,
              5000, 10_000, 50_000, 100_000,
              200_000, 300_000]

posterior_list = [form_posterior(draws, n_obs) for n_obs in n_obs_list]

θ_values = np.linspace(0.01, 1, 1000)

fig, ax = plt.subplots(figsize=(10, 6))

prior = st.beta(0.5, 0.5)
ax.plot(θ_values, prior.pdf(θ_values),
        label='n = 0 (prior)', linestyle='--')

for i, n_obs in enumerate(n_obs_list[:10]):
    posterior = posterior_list[i]
    ax.plot(θ_values, posterior.pdf(θ_values),
            label=f'n = {n_obs}')

ax.set_title('PDF of posterior distributions',
             fontsize=15)
ax.set_xlabel(r"$\theta$", fontsize=15)

ax.legend(fontsize=11)
plt.show()
```



c)

```
lower_bound = [post.ppf(0.05) for post in posterior_list[:10]]
upper_bound = [post.ppf(0.95) for post in posterior_list[:10]]

interval_df = pd.DataFrame()
interval_df['upper'] = upper_bound
interval_df['lower'] = lower_bound
interval_df.index = n_obs_list[:10]
interval_df = interval_df.T
interval_df
```

```
            1         2         3        4         5         10        20    \
upper  0.771480  0.902692  0.764466  0.83472  0.872224  0.882671  0.629953
lower  0.001543  0.097308  0.062413  0.16528  0.260634  0.441873  0.280091

            50        100       1000
upper  0.516104  0.502138  0.434744
lower  0.292234  0.341239  0.383644
```

As n increases, we can see that Bayesian coverage intervals narrow and move toward 0.4.

d) The Bayesian coverage interval tells the range of θ that corresponds to the $[q_1, q_2]$ quantiles of the cumulative distribution function (CDF) of the posterior distribution.

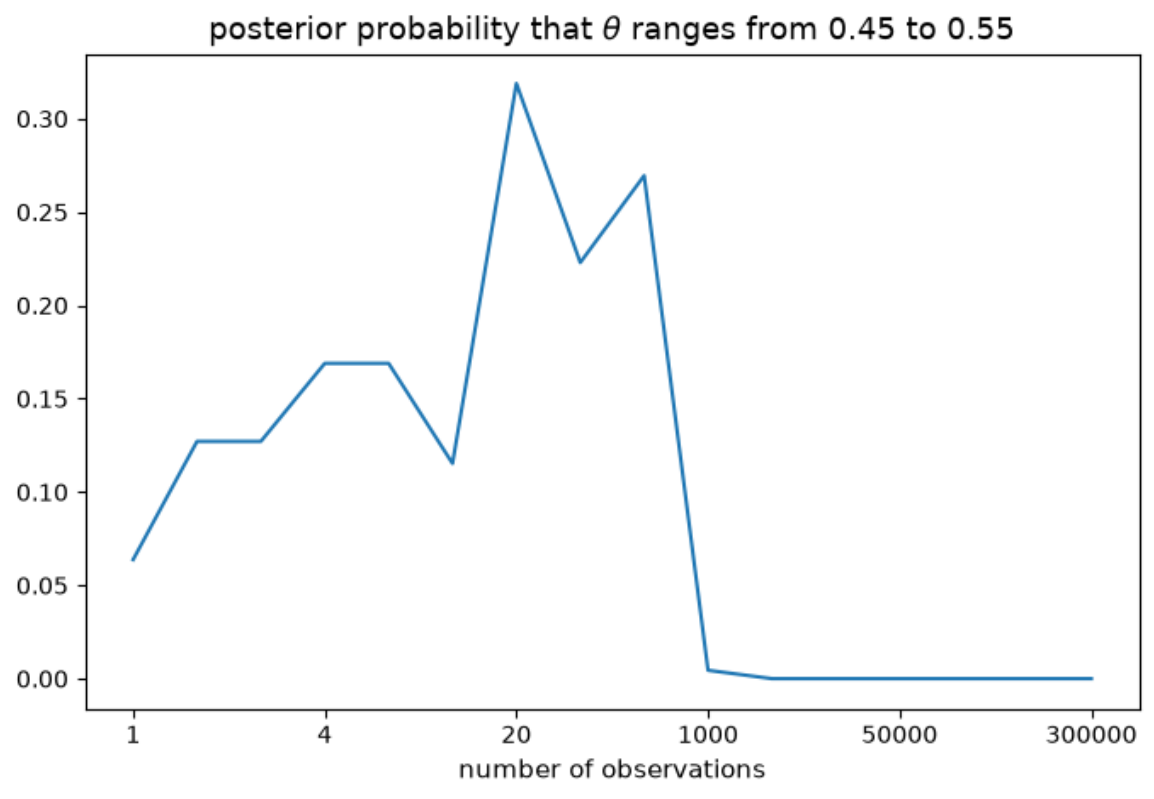
To construct the coverage interval we first compute a posterior distribution of the unknown parameter θ.

If the CDF is $F(\theta)$, then the Bayesian coverage interval $[a, b]$ for the interval $[q_1, q_2]$ is described by

$$F(a) = q_1, F(b) = q_2$$

e)

```
left_value, right_value = 0.45, 0.55

posterior_prob_list = [
    post.cdf(right_value) - post.cdf(left_value)
    for post in posterior_list
]

fig, ax = plt.subplots(figsize=(8, 5))
ax.plot(posterior_prob_list)
ax.set_title(
    r'posterior probability that $\theta$'
    f' ranges from {left_value:.2f}'
    f' to {right_value:.2f}',
    fontsize=13)
ax.set_xticks(np.arange(0, len(posterior_prob_list), 3))
ax.set_xticklabels(n_obs_list[::3])
ax.set_xlabel('number of observations', fontsize=11)

plt.show()
```



Notice that in the graph above the posterior probability that $\theta \in [0.45, 0.55]$ exhibits a hump shape as n increases.

Two opposing forces are at work.

The first force is that the individual adjusts his belief as he observes new outcomes, so his posterior probability distribution becomes more and more realistic, which explains the rise of the posterior probability.

However, $[0.45, 0.55]$ actually excludes the true $\theta = 0.4$ that generates the data.

As a result, the posterior probability drops as larger and larger samples refine his posterior probability distribution of θ.

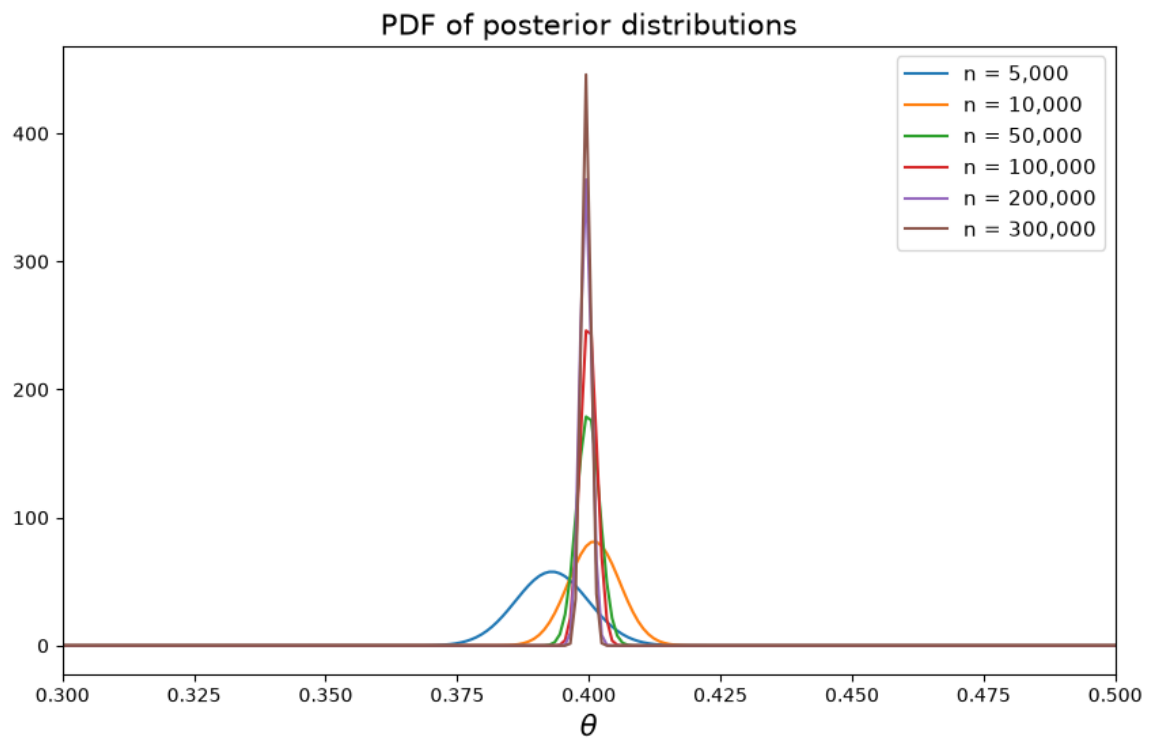
The descent seems precipitous only because of the scale of the graph that has the number of observations increasing disproportionately.

When the number of observations becomes large enough, our Bayesian becomes so confident about θ that he considers $\theta \in [0.45, 0.55]$ very unlikely.

That is why we see a nearly horizontal line when the number of observations exceeds 1000.

f) Using the functions we wrote above, we can see the evolution of posterior distributions as n approaches infinity.

```
fig, ax = plt.subplots(figsize=(10, 6))

for i, n_obs in enumerate(n_obs_list[10:]):
    posterior = posterior_list[i + 10]
    ax.plot(θ_values, posterior.pdf(θ_values),
            label=f'n = {n_obs:,}')

ax.set_title('PDF of posterior distributions', fontsize=15)
ax.set_xlabel(r"$\theta$", fontsize=15)
ax.set_xlim(0.3, 0.5)

ax.legend(fontsize=11)
plt.show()
```



As n increases, we can see that the probability density functions *concentrate* on 0.4, the true value of θ.

The next section explains *why* this concentration occurs and how fast it happens.

11.3.3 Why the posterior concentrates

In the solution to Exercise 11.3.2 we watched the posterior distribution concentrate ever more tightly around the true value $\theta = 0.4$ as the sample grew. Why does this happen?

The answer is encoded in the posterior we derived.

Recall that after observing k heads in n flips, the posterior is $\text{Beta}(\alpha + k,\ \beta + n - k)$.

A beta distribution with parameters a and b has

- mean $\dfrac{a}{a+b}$,
- variance $\dfrac{a\,b}{(a+b)^2\,(a+b+1)}$.

Substituting the *posterior* parameters $a = \alpha + k$ and $b = \beta + n - k$, so that $a + b = \alpha + \beta + n$, gives

$$\mathbb{E}[\theta \mid k] = \frac{\alpha + k}{\alpha + \beta + n}, \qquad \text{Var}[\theta \mid k] = \frac{(\alpha + k)(\beta + n - k)}{(\alpha + \beta + n)^2\,(\alpha + \beta + n + 1)}.$$

As n grows, the fixed prior counts α and β become negligible beside the data.

Since the data are generated with $\theta = 0.4$, the Law of Large Numbers gives $k/n \to 0.4$ (see Exercise 11.2.1), so the posterior mean

$$\frac{\alpha + k}{\alpha + \beta + n} \;\approx\; \frac{k}{n} \;\to\; 0.4.$$

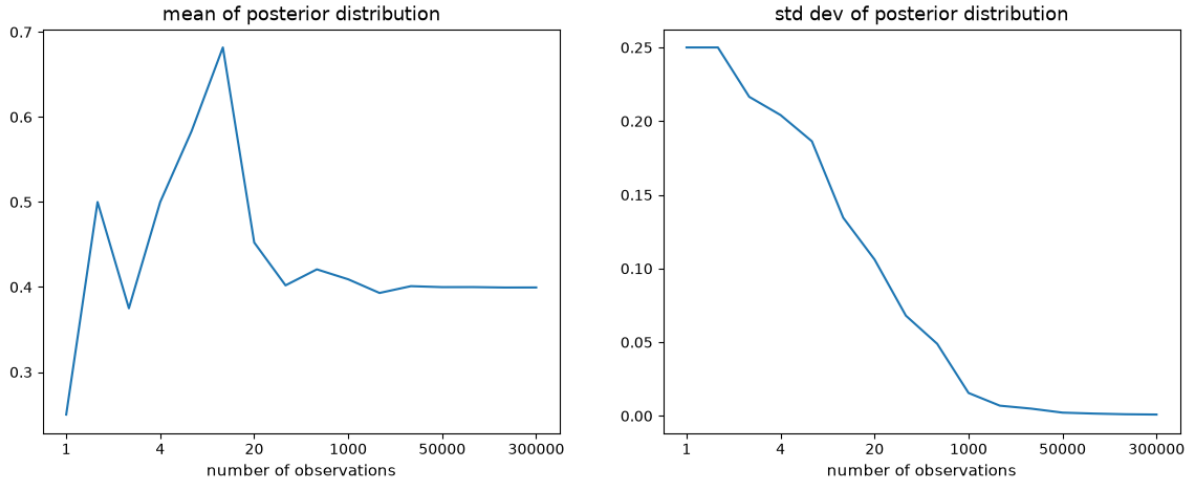
In the variance, the numerator grows like n^2 while the denominator grows like n^3, so

$$\text{Var}[\theta \mid k] \;\approx\; \frac{\theta(1-\theta)}{n} \;\longrightarrow\; 0.$$

The posterior mean therefore homes in on the truth while its spread vanishes at rate $1/n$.

The next figure confirms both claims: the posterior mean settles on 0.4 and the standard deviation decays toward zero.

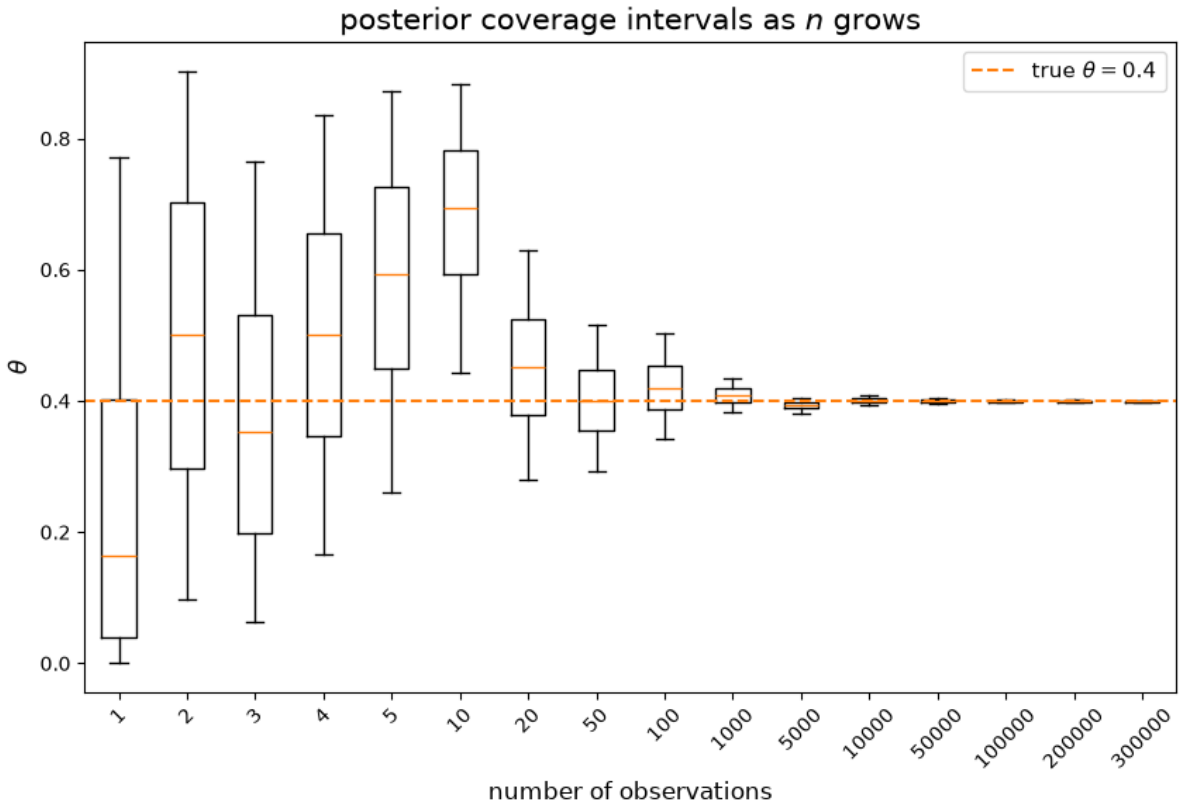
```
mean_list = [post.mean() for post in posterior_list]
std_list = [post.std() for post in posterior_list]

fig, ax = plt.subplots(1, 2, figsize=(14, 5))

ax[0].plot(mean_list)
ax[0].set_title('mean of posterior distribution',
                fontsize=13)
ax[0].set_xticks(np.arange(0, len(mean_list), 3))
ax[0].set_xticklabels(n_obs_list[::3])
ax[0].set_xlabel('number of observations', fontsize=11)

ax[1].plot(std_list)
ax[1].set_title('std dev of posterior distribution',
                fontsize=13)
ax[1].set_xticks(np.arange(0, len(std_list), 3))
ax[1].set_xticklabels(n_obs_list[::3])
ax[1].set_xlabel('number of observations', fontsize=11)

plt.show()
```



We can also display the Bayesian coverage intervals directly.

The box-and-whisker plot below summarizes each posterior by its median (central line), interquartile range (box), and 5th–95th percentile range (whiskers), with the true value $\theta = 0.4$ marked.

```
quantiles = [0.05, 0.25, 0.5, 0.75, 0.95]
box_stats = []
for post in posterior_list:
    lo, q1, med, q3, hi = post.ppf(quantiles)
    box_stats.append({'med': med, 'q1': q1, 'q3': q3,
                      'whislo': lo, 'whishi': hi, 'fliers': []})

fig, ax = plt.subplots(figsize=(10, 6))
ax.bxp(box_stats, positions=np.arange(len(box_stats)), showfliers=False)
ax.axhline(0.4, color='C1', linestyle='--', label=r'true $\theta = 0.4$')
ax.set_xticks(np.arange(len(box_stats)))
ax.set_xticklabels(n_obs_list, rotation=45)
ax.set_xlabel('number of observations', fontsize=12)
ax.set_ylabel(r'$\theta$', fontsize=12)
ax.set_title('posterior coverage intervals as $n$ grows', fontsize=15)
ax.legend(fontsize=11)
plt.show()
```



After observing a large number of outcomes, the posterior distribution collapses around 0.4.

Thus, the Bayesian statistician comes to believe that θ is near 0.4.

As shown in the figure above, as the number of observations grows, the Bayesian coverage intervals (BCIs) become narrower and narrower around 0.4.

However, if you take a closer look, you will find that the centers of the BCIs are not exactly 0.4, due to the persistent influence of the prior distribution and the randomness of the simulation path.

11.4 Comparing the two interpretations

We can now place the two meanings of probability side by side.

In the frequentist calculations, θ was a *fixed* number and a probability was a *relative frequency*: $p(k \mid \theta)$ is the fraction of a large number of independent length-n sequences in which exactly k heads occur, and we confirmed by simulation that $f_k^I \to p(k \mid \theta)$ as $I \to \infty$.

In the Bayesian calculations, θ was itself a *random variable* describing our beliefs, and a probability was a *degree of belief*. The Bayesian coverage interval summarizes the posterior $p(\theta \mid k)$: it reports the range of θ lying between two posterior quantiles, and so answers the question "given the data I have seen and my prior, where do I now believe θ lies?"

These are genuinely different questions, even though both rest on the same binomial likelihood. The frequentist statement describes the data-generating mechanism for a fixed θ; the Bayesian statement describes our uncertainty about θ given the particular data we happened to observe.

11.5 Role of a Conjugate Prior

We have made assumptions that link functional forms of our likelihood function and our prior in a way that has eased our calculations considerably.

In particular, our assumptions that the likelihood function is **binomial** and that the prior distribution is a **beta distribution** have the consequence that the posterior distribution implied by Bayes' Law is also a **beta distribution**.

So posterior and prior are both beta distributions, albeit ones with different parameters.

When a likelihood function and prior fit together like hand and glove in this way, we can say that the prior and posterior are **conjugate distributions**.

In this situation, we also sometimes say that we have **conjugate prior** for the likelihood function $p(k \mid \theta)$.

Typically, the functional form of the likelihood function determines the functional form of a **conjugate prior**.

A natural question to ask is why should a person's personal prior about a parameter θ be restricted to be described by a conjugate prior?

Why not some other functional form that more sincerely describes the person's beliefs?

To be argumentative, one could ask, why should the form of the likelihood function have *anything* to say about my personal beliefs about θ?

A dignified response to that question is, well, it shouldn't, but if you want to compute a posterior easily you'll just be happier if your prior is conjugate to your likelihood.

Otherwise, your posterior won't have a convenient analytical form and you'll be in the situation of wanting to apply the Markov chain Monte Carlo techniques deployed in *Non-Conjugate Priors*.

We also apply these powerful methods to approximating Bayesian posteriors for non-conjugate priors in *Posterior Distributions for AR(1) Parameters* and *Forecasting an AR(1) Process*.

CHAPTER

TWELVE

MULTIVARIATE HYPERGEOMETRIC DISTRIBUTION

Contents

- *Multivariate Hypergeometric Distribution*
  - *Overview*
  - *The Administrator's Problem*
  - *Usage*

12.1 Overview

This lecture describes how an administrator deployed a **multivariate hypergeometric distribution** in order to access the fairness of a procedure for awarding research grants.

In the lecture we'll learn about

- properties of the multivariate hypergeometric distribution
- first and second moments of a multivariate hypergeometric distribution
- using a Monte Carlo simulation of a multivariate normal distribution to evaluate the quality of a normal approximation
- the administrator's problem and why the multivariate hypergeometric distribution is the right tool

12.2 The Administrator's Problem

An administrator in charge of allocating research grants is in the following situation.

To help us forget details that are none of our business here and to protect the anonymity of the administrator and the subjects, we call research proposals **balls** and continents of residence of authors of a proposal a **color**.

There are K_i balls (proposals) of color i.

There are c distinct colors (continents of residence).

Thus, $i = 1, 2, \ldots, c$

So there is a total of $N = \sum_{i=1}^{c} K_i$ balls.

All N of these balls are placed in an urn.

Then n balls are drawn randomly.

The selection procedure is supposed to be **color blind** meaning that **ball quality**, a random variable that is supposed to be independent of **ball color**, governs whether a ball is drawn.

Thus, the selection procedure is supposed randomly to draw n balls from the urn.

The n balls drawn represent successful proposals and are awarded research funds.

The remaining $N - n$ balls receive no research funds.

12.2.1 Details of the Awards Procedure Under Study

Let k_i be the number of balls of color i that are drawn.

Things have to add up so $\sum_{i=1}^{c} k_i = n$.

Under the hypothesis that the selection process judges proposals on their quality and that quality is independent of continent of the author's continent of residence, the administrator views the outcome of the selection procedure as a random vector

$$X = \begin{bmatrix} k_1 \\ k_2 \\ \vdots \\ k_c \end{bmatrix}.$$

To evaluate whether the selection procedure is **color blind** the administrator wants to study whether the particular realization of X drawn can plausibly be said to be a random draw from the probability distribution that is implied by the **color blind** hypothesis.

The appropriate probability distribution is the one described here.

Let's now instantiate the administrator's problem, while continuing to use the colored balls metaphor.

The administrator has an urn with $N = 238$ balls.

157 balls are blue, 11 balls are green, 46 balls are yellow, and 24 balls are black.

So $(K_1, K_2, K_3, K_4) = (157, 11, 46, 24)$ and $c = 4$.

15 balls are drawn without replacement.

So $n = 15$.

The administrator wants to know the probability distribution of outcomes

$$X = \begin{bmatrix} k_1 \\ k_2 \\ \vdots \\ k_4 \end{bmatrix}.$$

In particular, he wants to know whether a particular outcome - in the form of a 4×1 vector of integers recording the numbers of blue, green, yellow, and black balls, respectively, - contains evidence against the hypothesis that the selection process is *fair*, which here means *color blind* and truly are random draws without replacement from the population of N balls.

The right tool for the administrator's job is the **multivariate hypergeometric distribution**.

12.2.2 Multivariate Hypergeometric Distribution

Let's start with some imports.

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.special import comb
from scipy.stats import normaltest
from numba import jit, prange
```

To recapitulate, we assume there are in total c types of objects in an urn.

If there are K_i type i object in the urn and we take n draws at random without replacement, then the numbers of type i objects in the sample $(k_1, k_2, \ldots, k_c)$ has the multivariate hypergeometric distribution.

Note again that $N = \sum_{i=1}^{c} K_i$ is the total number of objects in the urn and $n = \sum_{i=1}^{c} k_i$.

Notation

We use the following notation for **binomial coefficients**: $\binom{m}{q} = \frac{m!}{(m-q)!}$.

The multivariate hypergeometric distribution has the following properties:

Probability mass function:

$$\Pr\{X_i = k_i \ \forall i\} = \frac{\prod_{i=1}^{c} \binom{K_i}{k_i}}{\binom{N}{n}}$$

Mean:

$$\mathrm{E}(X_i) = n\frac{K_i}{N}$$

Variances and covariances:

$$\mathrm{Var}(X_i) = n\frac{N-n}{N-1}\ \frac{K_i}{N}\left(1 - \frac{K_i}{N}\right)$$

$$\mathrm{Cov}(X_i, X_j) = -n\frac{N-n}{N-1}\ \frac{K_i}{N}\frac{K_j}{N}$$

To do our work for us, we'll write an `Urn` class.

```
class Urn:

    def __init__(self, K_arr):
        """
        Initialization given the number of each type i object in the urn.

        Parameters
        ----------
        K_arr: ndarray(int)
            number of each type i object.
        """

        self.K_arr = np.array(K_arr)
        self.N = np.sum(K_arr)
        self.c = len(K_arr)

    def pmf(self, k_arr):
        """
```

(continues on next page)

(continued from previous page)

```
        Probability mass function.

        Parameters
        ----------
        k_arr: ndarray(int)
            number of observed successes of each object.
        """

        K_arr, N = self.K_arr, self.N

        k_arr = np.atleast_2d(k_arr)
        n = np.sum(k_arr, 1)

        num = np.prod(comb(K_arr, k_arr), 1)
        denom = comb(N, n)

        pr = num / denom

        return pr

    def moments(self, n):
        """
        Compute the mean and variance-covariance matrix for
        multivariate hypergeometric distribution.

        Parameters
        ----------
        n: int
            number of draws.
        """

        K_arr, N, c = self.K_arr, self.N, self.c

        # mean
        μ = n * K_arr / N

        # variance-covariance matrix
        Σ = np.full((c, c), n * (N - n) / (N - 1) / N ** 2)
        for i in range(c-1):
            Σ[i, i] *= K_arr[i] * (N - K_arr[i])
            for j in range(i+1, c):
                Σ[i, j] *= - K_arr[i] * K_arr[j]
                Σ[j, i] = Σ[i, j]

        Σ[-1, -1] *= K_arr[-1] * (N - K_arr[-1])

        return μ, Σ

    def simulate(self, n, size=1, seed=None):
        """
        Simulate a sample from multivariate hypergeometric
        distribution where at each draw we take n objects
        from the urn without replacement.

        Parameters
        ----------
        n: int
```

(continues on next page)

(continued from previous page)

```
            number of objects for each draw.
        size: int(optional)
            sample size.
        seed: int(optional)
            random seed.
        """

        K_arr = self.K_arr

        gen = np.random.Generator(np.random.PCG64(seed))
        sample = gen.multivariate_hypergeometric(K_arr, n, size=size)

        return sample
```

12.3 Usage

12.3.1 First example

Apply this to an example from wiki:

Suppose there are 5 black, 10 white, and 15 red marbles in an urn. If six marbles are chosen without replacement, the probability that exactly two of each color are chosen is

$$P(\text{2 black, 2 white, 2 red}) = \frac{\binom{5}{2}\binom{10}{2}\binom{15}{2}}{\binom{30}{6}} = 0.079575596816976$$

```
# construct the urn
K_arr = [5, 10, 15]
urn = Urn(K_arr)
```

Now use the Urn Class method `pmf` to compute the probability of the outcome $X = \begin{bmatrix} 2 & 2 & 2 \end{bmatrix}$

```
k_arr = [2, 2, 2] # array of number of observed successes
urn.pmf(k_arr)
```

```
array([0.0795756])
```

We can use the code to compute probabilities of a list of possible outcomes by constructing a 2-dimensional array `k_arr` and `pmf` will return an array of probabilities for observing each case.

```
k_arr = [[2, 2, 2], [1, 3, 2]]
urn.pmf(k_arr)
```

```
array([0.0795756, 0.1061008])
```

Now let's compute the mean vector and variance-covariance matrix.

```
n = 6
μ, Σ = urn.moments(n)
```

```
μ
```

```
array([1., 2., 3.])
```

```
Σ
```

```
array([[ 0.68965517, -0.27586207, -0.4137931 ],
       [-0.27586207,  1.10344828, -0.82758621],
       [-0.4137931 , -0.82758621,  1.24137931]])
```

12.3.2 Back to The Administrator's Problem

Now let's turn to the grant administrator's problem.

Here the array of numbers of i objects in the urn is $(157, 11, 46, 24)$.

```
K_arr = [157, 11, 46, 24]
urn = Urn(K_arr)
```

Let's compute the probability of the outcome $(10, 1, 4, 0)$.

```
k_arr = [10, 1, 4, 0]
urn.pmf(k_arr)
```

```
array([0.01547738])
```

We can compute probabilities of three possible outcomes by constructing a 3-dimensional arrays `k_arr` and utilizing the method `pmf` of the `Urn` class.

```
k_arr = [[5, 5, 4 ,1], [10, 1, 2, 2], [13, 0, 2, 0]]
urn.pmf(k_arr)
```

```
array([6.21412534e-06, 2.70935969e-02, 1.61839976e-02])
```

Now let's compute the mean and variance-covariance matrix of X when $n = 6$.

```
n = 6 # number of draws
μ, Σ = urn.moments(n)
```

```
# mean
μ
```

```
array([3.95798319, 0.27731092, 1.15966387, 0.60504202])
```

```
# variance-covariance matrix
Σ
```

```
array([[ 1.31862604, -0.17907267, -0.74884935, -0.39070401],
       [-0.17907267,  0.25891399, -0.05246715, -0.02737417],
       [-0.74884935, -0.05246715,  0.91579029, -0.11447379],
       [-0.39070401, -0.02737417, -0.11447379,  0.53255196]])
```

We can simulate a large sample and verify that sample means and covariances closely approximate the population means and covariances.

```
size = 10_000_000
sample = urn.simulate(n, size=size)
```

```
# mean
np.mean(sample, 0)
```

```
array([3.9579971, 0.2774066, 1.1595536, 0.6050427])
```

```
# variance covariance matrix
np.cov(sample.T)
```

```
array([[ 1.31702119, -0.17873534, -0.74848336, -0.38980249],
       [-0.17873534,  0.2589064 , -0.05256713, -0.02760394],
       [-0.74848336, -0.05256713,  0.91573514, -0.11468465],
       [-0.38980249, -0.02760394, -0.11468465,  0.53209108]])
```

Evidently, the sample means and covariances approximate their population counterparts well.

12.3.3 Quality of Normal Approximation

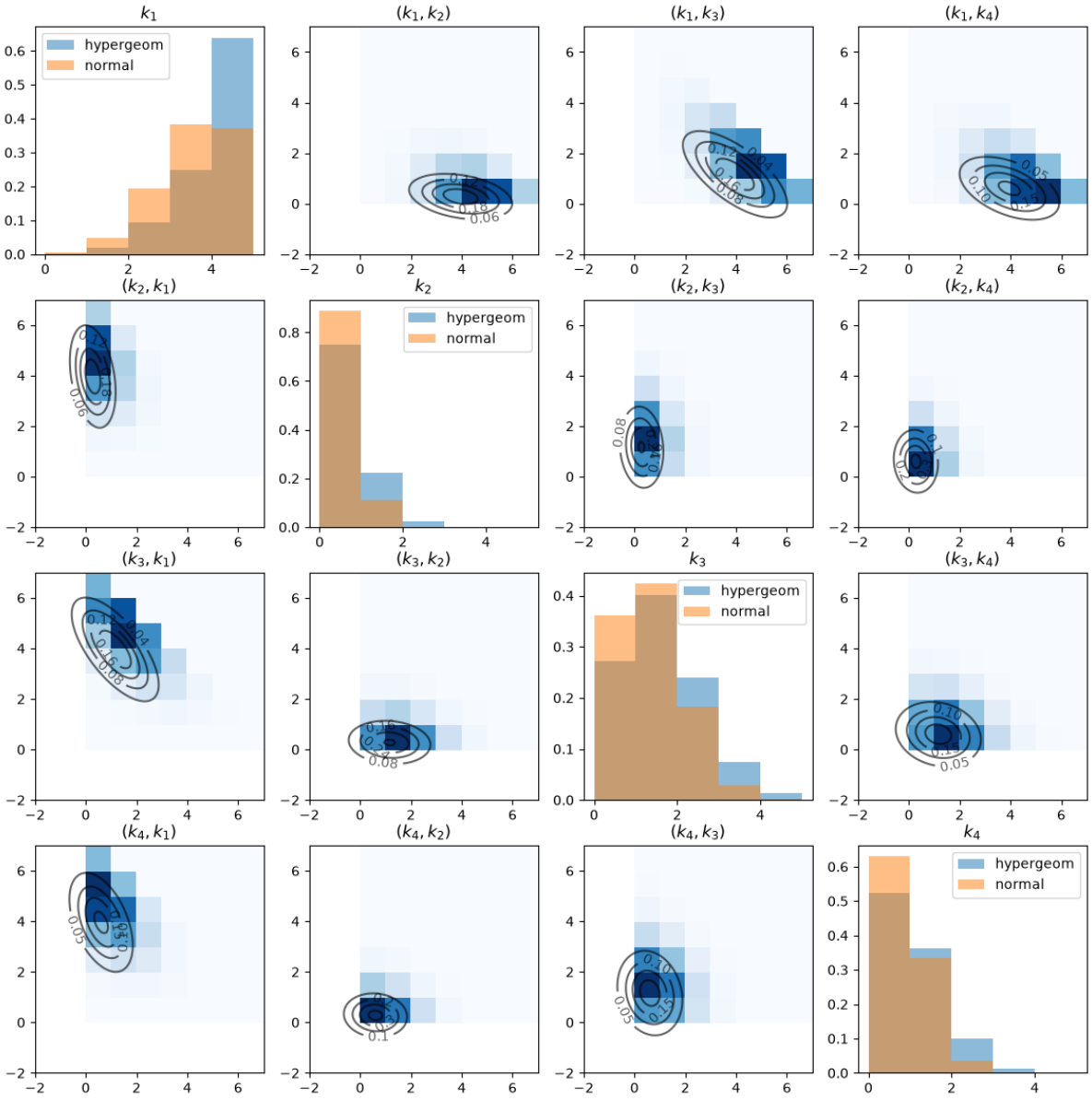
To judge the quality of a multivariate normal approximation to the multivariate hypergeometric distribution, we draw a large sample from a multivariate normal distribution with the mean vector and covariance matrix for the corresponding multivariate hypergeometric distribution and compare the simulated distribution with the population multivariate hypergeometric distribution.

```
sample_normal = np.random.multivariate_normal(μ, Σ, size=size)
```

```
def bivariate_normal(x, y, μ, Σ, i, j):

    μ_x, μ_y = μ[i], μ[j]
    σ_x, σ_y = np.sqrt(Σ[i, i]), np.sqrt(Σ[j, j])
    σ_xy = Σ[i, j]

    x_μ = x - μ_x
    y_μ = y - μ_y

    ρ = σ_xy / (σ_x * σ_y)
    z = x_μ**2 / σ_x**2 + y_μ**2 / σ_y**2 - 2 * ρ * x_μ * y_μ / (σ_x * σ_y)
    denom = 2 * np.pi * σ_x * σ_y * np.sqrt(1 - ρ**2)

    return np.exp(-z / (2 * (1 - ρ**2))) / denom
```

```
@jit
def count(vec1, vec2, n):
    size = sample.shape[0]

    count_mat = np.zeros((n+1, n+1))
    for i in prange(size):
        count_mat[vec1[i], vec2[i]] += 1

    return count_mat
```

```
c = urn.c
fig, axs = plt.subplots(c, c, figsize=(14, 14))

# grids for ploting the bivariate Gaussian
x_grid = np.linspace(-2, n+1, 100)
y_grid = np.linspace(-2, n+1, 100)
X, Y = np.meshgrid(x_grid, y_grid)

for i in range(c):
    axs[i, i].hist(sample[:, i], bins=np.arange(0, n, 1), alpha=0.5, density=True,␣
↪label='hypergeom')
    axs[i, i].hist(sample_normal[:, i], bins=np.arange(0, n, 1), alpha=0.5,␣
↪density=True, label='normal')
    axs[i, i].legend()
    axs[i, i].set_title('$k_{' +str(i+1) +'}$')
    for j in range(c):
        if i == j:
            continue

        # bivariate Gaussian density function
        Z = bivariate_normal(X, Y, μ, Σ, i, j)
        cs = axs[i, j].contour(X, Y, Z, 4, colors="black", alpha=0.6)
        axs[i, j].clabel(cs, inline=1, fontsize=10)

        # empirical multivariate hypergeometric distrbution
        count_mat = count(sample[:, i], sample[:, j], n)
        axs[i, j].pcolor(count_mat.T/size, cmap='Blues')
        axs[i, j].set_title('$(k_{' +str(i+1) +'}, k_{' + str(j+1) + '})$')

plt.show()
```



The diagonal graphs plot the marginal distributions of k_i for each i using histograms.

Note the substantial differences between hypergeometric distribution and the approximating normal distribution.

The off-diagonal graphs plot the empirical joint distribution of k_i and k_j for each pair (i, j).

The darker the blue, the more data points are contained in the corresponding cell. (Note that k_i is on the x-axis and k_j is on the y-axis).

The contour maps plot the bivariate Gaussian density function of $\left(k_i, k_j\right)$ with the population mean and covariance given by slices of μ and Σ that we computed above.

Let's also test the normality for each k_i using `scipy.stats.normaltest` that implements D'Agostino and Pearson's test that combines skew and kurtosis to form an omnibus test of normality.

The null hypothesis is that the sample follows normal distribution.

`normaltest` returns an array of p-values associated with tests for each k_i sample.

```
test_multihyper = normaltest(sample)
test_multihyper.pvalue
```

```
array([0., 0., 0., 0.])
```

As we can see, all the p-values are almost 0 and the null hypothesis is soundly rejected.

By contrast, the sample from normal distribution does not reject the null hypothesis.

```
test_normal = normaltest(sample_normal)
test_normal.pvalue
```

```
array([0.3240851 , 0.04062622, 0.60824685, 0.32149153])
```

The lesson to take away from this is that the normal approximation is imperfect.

CHAPTER

THIRTEEN

MULTIVARIATE NORMAL DISTRIBUTION

Contents

- *Multivariate Normal Distribution*
  - *Overview*
  - *The multivariate normal distribution*
  - *Bivariate example*
  - *Trivariate example*
  - *One dimensional intelligence (IQ)*
  - *Information as surprise*
  - *Cholesky factor magic*
  - *Math and verbal intelligence*
  - *Univariate time series analysis*
  - *Stochastic difference equation*
  - *Application to stock price model*
  - *Filtering foundations*
  - *Classic factor analysis model*
  - *PCA and factor analysis*
  - *Exercises*

13.1 Overview

This lecture describes a workhorse in probability theory, statistics, and economics, namely, the **multivariate normal distribution**.

In this lecture, you will learn formulas for

- the joint distribution of a random vector x of length N
- marginal distributions for all subvectors of x
- conditional distributions for subvectors of x conditional on other subvectors of x

We will use the multivariate normal distribution to formulate some useful models:

- a factor analytic model of an intelligence quotient, i.e., IQ
- a factor analytic model of two independent inherent abilities, say, mathematical and verbal.
- a more general factor analytic model
- Principal Components Analysis (PCA) as an approximation to a factor analytic model
- time series generated by linear stochastic difference equations
- optimal linear filtering theory

13.2 The multivariate normal distribution

This lecture defines a Python class `MultivariateNormal` to be used to generate **marginal** and **conditional** distributions associated with a multivariate normal distribution.

For a multivariate normal distribution it is very convenient that

- conditional expectations equal linear least squares projections
- conditional distributions are characterized by multivariate linear regressions

We apply our Python class to some examples.

We use the following imports:

```
import matplotlib.pyplot as plt
import numpy as np
from numba import jit
import statsmodels.api as sm

rng = np.random.default_rng(0)
```

Assume that an $N \times 1$ random vector z has a multivariate normal probability density.

This means that the probability density takes the form

$$f\left(z;\mu,\Sigma\right)=\left(2\pi\right)^{-\left(\frac{N}{2}\right)}\det\left(\Sigma\right)^{-\frac{1}{2}}\exp\left(-.5\left(z-\mu\right)^{\top}\Sigma^{-1}\left(z-\mu\right)\right)$$

where $\mu=Ez$ is the mean of the random vector z and $\Sigma=E\left(z-\mu\right)\left(z-\mu\right)^{\top}$ is the covariance matrix of z.

The covariance matrix Σ is symmetric and positive definite.

```
@jit
def f(z, μ, Σ):
    """
    The density function of multivariate normal distribution.

    Parameters
    ---------------
    z: ndarray(float, dim=2)
        random vector, N by 1
    μ: ndarray(float, dim=1 or 2)
        the mean of z, N by 1
    Σ: ndarray(float, dim=2)
        the covariance matrix of z, N by N
    """
```

(continues on next page)

(continued from previous page)

```
    z = np.atleast_2d(z)
    μ = np.atleast_2d(μ)
    Σ = np.atleast_2d(Σ)

    N = z.size

    temp1 = np.linalg.det(Σ) ** (-1/2)
    temp2 = np.exp(-.5 * (z - μ).T @ np.linalg.inv(Σ) @ (z - μ))

    return (2 * np.pi) ** (-N/2) * temp1 * temp2
```

For some integer $k \in \{1, \ldots, N-1\}$, partition z as

$$z = \begin{bmatrix} z_1 \\ z_2 \end{bmatrix},$$

where z_1 is an $(N-k) \times 1$ vector and z_2 is a $k \times 1$ vector.

Let

$$\mu = \begin{bmatrix} \mu_1 \\ \mu_2 \end{bmatrix}, \quad \Sigma = \begin{bmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{bmatrix}$$

be corresponding partitions of μ and Σ.

The **marginal** distribution of z_1 is

- multivariate normal with mean μ_1 and covariance matrix Σ_{11}.

The **marginal** distribution of z_2 is

- multivariate normal with mean μ_2 and covariance matrix Σ_{22}.

The distribution of z_1 **conditional** on z_2 is

- multivariate normal with mean

$$\hat{\mu}_1 = \mu_1 + \beta (z_2 - \mu_2)$$

and covariance matrix

$$\hat{\Sigma}_{11} = \Sigma_{11} - \Sigma_{12}\Sigma_{22}^{-1}\Sigma_{21} = \Sigma_{11} - \beta\Sigma_{22}\beta^\top$$

where

$$\beta = \Sigma_{12}\Sigma_{22}^{-1}$$

is an $(N-k) \times k$ matrix of **population regression coefficients** of the $(N-k) \times 1$ random vector $z_1 - \mu_1$ on the $k \times 1$ random vector $z_2 - \mu_2$.

The following class constructs a multivariate normal distribution instance with two methods.

- a method `partition` computes β, taking k as an input
- a method `cond_dist` computes either the distribution of z_1 conditional on z_2 or the distribution of z_2 conditional on z_1

```
class MultivariateNormal:
    """
    Class of multivariate normal distribution.

    Parameters
    ----------
    μ: ndarray(float, dim=1)
        the mean of z, N by 1
    Σ: ndarray(float, dim=2)
        the covariance matrix of z, N by N

    Arguments
    ---------
    μ, Σ:
        see parameters
    μs: list(ndarray(float, dim=1))
        list of mean vectors μ1 and μ2 in order
    Σs: list(list(ndarray(float, dim=2)))
        2 dimensional list of covariance matrices
        Σ11, Σ12, Σ21, Σ22 in order
    βs: list(ndarray(float, dim=1))
        list of regression coefficients β1 and β2 in order
    """

    def __init__(self, μ, Σ):
        "initialization"
        self.μ = np.array(μ)
        self.Σ = np.atleast_2d(Σ)

    def partition(self, k):
        """
        Given k, partition the random vector z into a size k vector z1
        and a size N-k vector z2. Partition the mean vector μ into
        μ1 and μ2, and the covariance matrix Σ into Σ11, Σ12, Σ21, Σ22
        correspondingly. Compute the regression coefficients β1 and β2
        using the partitioned arrays.
        """
        μ = self.μ
        Σ = self.Σ

        self.μs = [μ[:k], μ[k:]]
        self.Σs = [[Σ[:k, :k], Σ[:k, k:]],
                   [Σ[k:, :k], Σ[k:, k:]]]

        self.βs = [self.Σs[0][1] @ np.linalg.inv(self.Σs[1][1]),
                   self.Σs[1][0] @ np.linalg.inv(self.Σs[0][0])]

    def cond_dist(self, ind, z):
        """
        Compute the conditional distribution of z1 given z2, or reversely.
        Argument ind determines whether we compute the conditional
        distribution of z1 (ind=0) or z2 (ind=1).

        Returns
        ---------
        μ_hat: ndarray(float, ndim=1)
            The conditional mean of z1 or z2.
```

(continues on next page)

(continued from previous page)

```
        Σ_hat: ndarray(float, ndim=2)
            The conditional covariance matrix of z1 or z2.
        """
        β = self.βs[ind]
        μs = self.μs
        Σs = self.Σs

        μ_hat = μs[ind] + β @ (z - μs[1-ind])
        Σ_hat = Σs[ind][ind] - β @ Σs[1-ind][1-ind] @ β.T

        return μ_hat, Σ_hat
```

Let's put this code to work on a suite of examples.

We begin with a simple bivariate example; after that we'll turn to a trivariate example.

We'll compute population moments of some conditional distributions using our `MultivariateNormal` class.

For fun we'll also compute sample analogs of the associated population regressions by generating simulations and then computing linear least squares regressions.

We'll compare those linear least squares regressions for the simulated data to their population counterparts.

13.3 Bivariate example

We start with a bivariate normal distribution pinned down by

$$\mu = \begin{bmatrix} .5 \\ 1.0 \end{bmatrix}, \quad \Sigma = \begin{bmatrix} 1 & .5 \\ .5 & 1 \end{bmatrix}$$

```
μ = np.array([.5, 1.])
Σ = np.array([[1., .5], [.5 ,1.]])

# construction of the multivariate normal instance
multi_normal = MultivariateNormal(μ, Σ)
```

```
k = 1 # choose partition

# partition and compute regression coefficients
multi_normal.partition(k)
multi_normal.βs[0],multi_normal.βs[1]
```

```
(array([[0.5]]), array([[0.5]]))
```

Let's illustrate the fact that you *can regress anything on anything else*.

We have computed everything we need to compute two regression lines, one of z_2 on z_1, the other of z_1 on z_2.

We'll represent these regressions as

$$z_1 = a_1 + b_1 z_2 + \epsilon_1$$

and

$$z_2 = a_2 + b_2 z_1 + \epsilon_2$$

where we have the population least squares orthogonality conditions

$$E\epsilon_1 z_2 = 0$$

and

$$E\epsilon_2 z_1 = 0$$

Let's compute a_1, a_2, b_1, b_2.

```
β = multi_normal.βs

a1 = μ[0] - β[0]*μ[1]
b1 = β[0]

a2 = μ[1] - β[1]*μ[0]
b2 = β[1]
```

Let's print out the intercepts and slopes.

For the regression of z_1 on z_2 we have

```
print ("a1 = ", a1)
print ("b1 = ", b1)
```

```
a1 =  [[0.]]
b1 =  [[0.5]]
```

For the regression of z_2 on z_1 we have

```
print ("a2 = ", a2)
print ("b2 = ", b2)
```

```
a2 =  [[0.75]]
b2 =  [[0.5]]
```

Now let's plot the two regression lines and stare at them.

```
z2 = np.linspace(-4,4,100)


a1 = np.squeeze(a1)
b1 = np.squeeze(b1)

a2 = np.squeeze(a2)
b2 = np.squeeze(b2)

z1  = b1*z2 + a1


z1h = z2/b2 - a2/b2


fig = plt.figure(figsize=(12,12))
ax = fig.add_subplot(1, 1, 1)
ax.set(xlim=(-4, 4), ylim=(-4, 4))
ax.spines['left'].set_position('center')
```

(continues on next page)

(continued from previous page)

```
ax.spines['bottom'].set_position('zero')
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.xaxis.set_ticks_position('bottom')
ax.yaxis.set_ticks_position('left')
plt.ylabel('$z_1$', loc = 'top')
plt.xlabel('$z_2$,', loc = 'right')
plt.plot(z2,z1, 'r', label = "$z_1$ on $z_2$")
plt.plot(z2,z1h, 'b', label = "$z_2$ on $z_1$")
plt.legend()
plt.show()
```

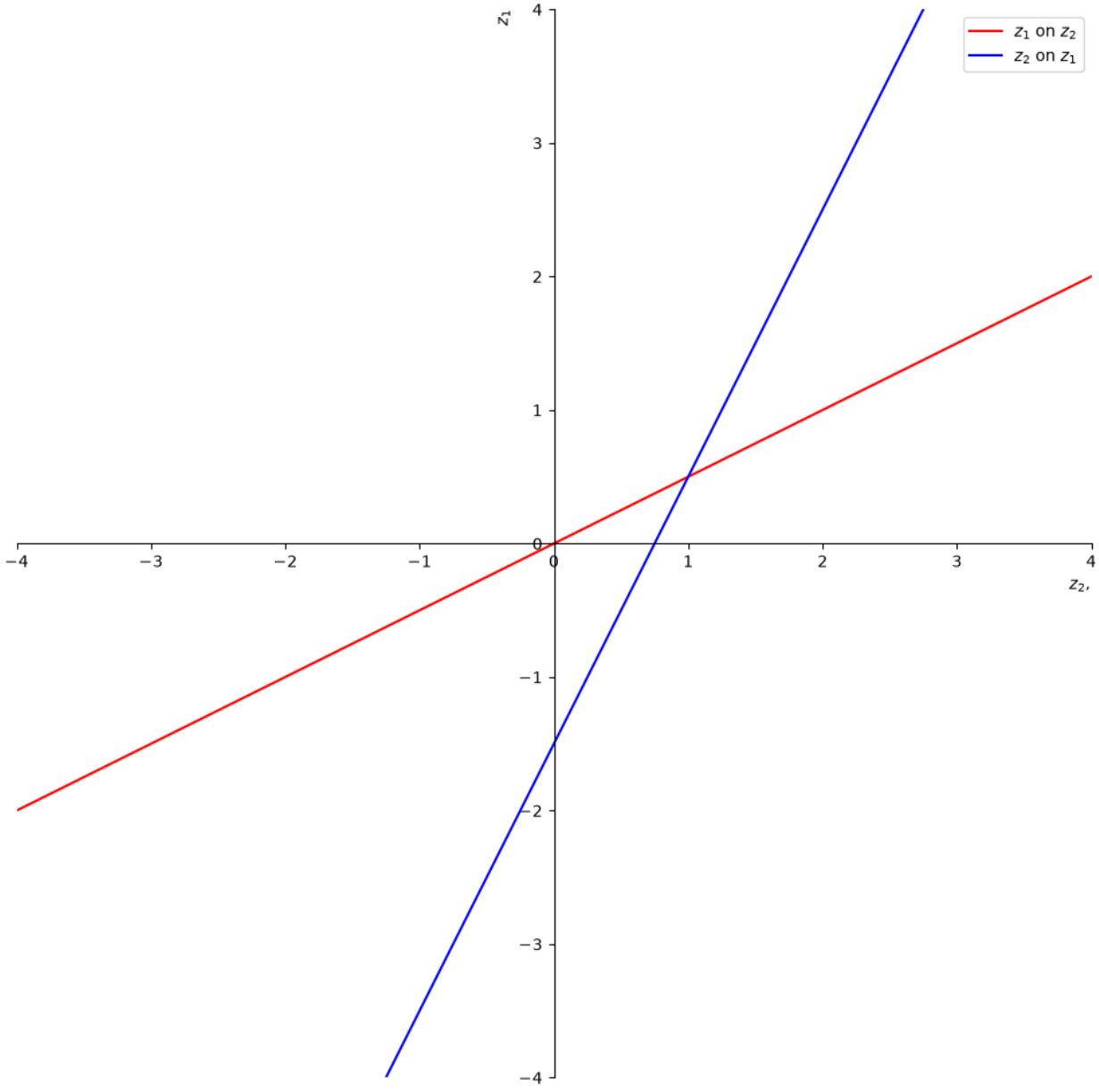


Fig. 13.1: two regressions

The red line is the expectation of z_1 conditional on z_2.

The intercept and slope of the red line are

```
print("a1 = ", a1)
print("b1 = ", b1)
```

```
a1 =  0.0
b1 =  0.5
```

The blue line is the expectation of z_2 conditional on z_1.

The intercept and slope of the blue line are

```
print("-a2/b2 = ", - a2/b2)
print("1/b2 = ", 1/b2)
```

```
-a2/b2 =  -1.5
1/b2 =  2.0
```

We can use these regression lines or our code to compute conditional expectations.

Let's compute the mean and variance of the distribution of z_2 conditional on $z_1 = 5$.

After that we'll reverse what are on the left and right sides of the regression.

```
# compute the cond. dist. of z1
ind = 1
z1 = np.array([5.]) # given z1

μ2_hat, Σ2_hat = multi_normal.cond_dist(ind, z1)
print('μ2_hat, Σ2_hat = ', μ2_hat, Σ2_hat)
```

```
μ2_hat, Σ2_hat =  [3.25] [[0.75]]
```

Now let's compute the mean and variance of the distribution of z_1 conditional on $z_2 = 5$.

```
# compute the cond. dist. of z1
ind = 0
z2 = np.array([5.]) # given z2

μ1_hat, Σ1_hat = multi_normal.cond_dist(ind, z2)
print('μ1_hat, Σ1_hat = ', μ1_hat, Σ1_hat)
```

```
μ1_hat, Σ1_hat =  [2.5] [[0.75]]
```

Let's compare the preceding population mean and variance with outcomes from drawing a large sample and then regressing $z_1 - \mu_1$ on $z_2 - \mu_2$.

We know that

$$Ez_1|z_2 = (\mu_1 - \beta\mu_2) + \beta z_2$$

which can be arranged to

$$z_1 - \mu_1 = \beta\left(z_2 - \mu_2\right) + \epsilon,$$

We anticipate that for larger and larger sample sizes, estimated OLS coefficients will converge to β and the estimated variance of ϵ will converge to $\hat{\Sigma}_1$.

```
n = 1_000_000 # sample size

# simulate multivariate normal random vectors
data = rng.multivariate_normal(μ, Σ, size=n)
z1_data = data[:, 0]
z2_data = data[:, 1]

# OLS regression
μ1, μ2 = multi_normal.μs
results = sm.OLS(z1_data - μ1, z2_data - μ2).fit()
```

Let's compare the preceding population β with the OLS sample estimate on $z_2 - \mu_2$

```
multi_normal.βs[0], results.params
```

```
(array([[0.5]]), array([0.49918333]))
```

Let's compare our population $\hat{\Sigma}_1$ with the degrees-of-freedom adjusted estimate of the variance of ϵ

```
Σ1_hat, results.resid @ results.resid.T / (n - 1)
```

```
(array([[0.75]]), np.float64(0.7489203195072748))
```

Lastly, let's compute the estimate of $\hat{Ez_1|z_2}$ and compare it with $\hat{\mu}_1$

```
μ1_hat, results.predict(z2 - μ2) + μ1
```

```
(array([2.5]), array([2.49673331]))
```

Thus, in each case, for our very large sample size, the sample analogues closely approximate their population counterparts.

A Law of Large Numbers explains why sample analogues approximate population objects.

13.4 Trivariate example

Let's apply our code to a trivariate example.

We'll specify the mean vector and the covariance matrix as follows.

```
μ = rng.random(3)
C = rng.random((3, 3))
Σ = C @ C.T # positive semi-definite

multi_normal = MultivariateNormal(μ, Σ)
```

```
μ, Σ
```

```
(array([0.4201905 , 0.04279279, 0.32149359]),
 array([[0.95070417, 0.48194854, 0.38025012],
        [0.48194854, 1.34029754, 0.43476327],
        [0.38025012, 0.43476327, 0.20864566]]))
```

```
k = 1
multi_normal.partition(k)
```

Let's compute the distribution of z_1 conditional on $z_2 = \begin{bmatrix} 2 \\ 5 \end{bmatrix}$.

```
ind = 0
z2 = np.array([2., 5.])

μ1_hat, Σ1_hat = multi_normal.cond_dist(ind, z2)
```

```
n = 1_000_000
data = rng.multivariate_normal(μ, Σ, size=n)
z1_data = data[:, :k]
z2_data = data[:, k:]
```

```
μ1, μ2 = multi_normal.μs
results = sm.OLS(z1_data - μ1, z2_data - μ2).fit()
```

As above, we compare population and sample regression coefficients, the conditional covariance matrix, and the conditional mean vector in that order.

```
multi_normal.βs[0], results.params
```

```
(array([[-0.71459379,  3.311496  ]]), array([-0.71383888,  3.31026328]))
```

```
Σ1_hat, results.resid @ results.resid.T / (n - 1)
```

```
(array([[0.03590485]]), np.float64(0.035947932906627136))
```

```
μ1_hat, results.predict(z2 - μ2) + μ1
```

```
(array([14.5144376]), array([14.51014783]))
```

Once again, sample analogues do a good job of approximating their populations counterparts.

13.5 One dimensional intelligence (IQ)

Let's move closer to a real-life example, namely, inferring a one-dimensional measure of intelligence called IQ from a list of test scores.

The ith test score y_i equals the sum of an unknown scalar IQ θ and a random variable w_i.

$$y_i = \theta + \sigma_y w_i, \quad i = 1, \ldots, n$$

The distribution of IQ's for a cross-section of people is a normal random variable described by

$$\theta = \mu_\theta + \sigma_\theta w_{n+1}.$$

We assume that the noises $\{w_i\}_{i=1}^N$ in the test scores are IID and not correlated with IQ.

We also assume that $\{w_i\}_{i=1}^{n+1}$ are i.i.d. standard normal:

$$w = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \\ w_{n+1} \end{bmatrix} \sim N\left(0, I_{n+1}\right)$$

The following system describes the $(n+1) \times 1$ random vector X that interests us:

$$X = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \\ \theta \end{bmatrix} = \begin{bmatrix} \mu_\theta \\ \mu_\theta \\ \vdots \\ \mu_\theta \\ \mu_\theta \end{bmatrix} + \begin{bmatrix} \sigma_y & 0 & \cdots & 0 & \sigma_\theta \\ 0 & \sigma_y & \cdots & 0 & \sigma_\theta \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & \sigma_y & \sigma_\theta \\ 0 & 0 & \cdots & 0 & \sigma_\theta \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \\ w_{n+1} \end{bmatrix},$$

or equivalently,

$$X = \mu_\theta 1_{n+1} + Dw$$

where $X = \begin{bmatrix} y \\ \theta \end{bmatrix}$, 1_{n+1} is a vector of 1s of size $n+1$, and D is an $n+1$ by $n+1$ matrix.

Let's define a Python function that constructs the mean μ and covariance matrix Σ of the random vector X that we know is governed by a multivariate normal distribution.

As arguments, the function takes the number of tests n, the mean μ_θ and the standard deviation σ_θ of the IQ distribution, and the standard deviation of the randomness in test scores σ_y.

```
def construct_moments_IQ(n, μθ, σθ, σy):

    μ_IQ = np.full(n+1, μθ)

    D_IQ = np.zeros((n+1, n+1))
    D_IQ[range(n), range(n)] = σy
    D_IQ[:, n] = σθ

    Σ_IQ = D_IQ @ D_IQ.T

    return μ_IQ, Σ_IQ, D_IQ
```

Now let's consider a specific instance of this model.

Assume we have recorded 50 test scores and we know that $\mu_\theta = 100$, $\sigma_\theta = 10$, and $\sigma_y = 10$.

We can compute the mean vector and covariance matrix of X easily with our `construct_moments_IQ` function as follows.

```
n = 50
μθ, σθ, σy = 100., 10., 10.

μ_IQ, Σ_IQ, D_IQ = construct_moments_IQ(n, μθ, σθ, σy)
μ_IQ, Σ_IQ, D_IQ
```

```
(array([100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
        100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
        100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
        100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
```

(continues on next page)

(continued from previous page)

```
       100., 100., 100., 100., 100., 100., 100.]),
array([[200., 100., 100., ..., 100., 100., 100.],
       [100., 200., 100., ..., 100., 100., 100.],
       [100., 100., 200., ..., 100., 100., 100.],
       ...,
       [100., 100., 100., ..., 200., 100., 100.],
       [100., 100., 100., ..., 100., 200., 100.],
       [100., 100., 100., ..., 100., 100., 100.]], shape=(51, 51)),
array([[10.,  0.,  0., ...,  0.,  0., 10.],
       [ 0., 10.,  0., ...,  0.,  0., 10.],
       [ 0.,  0., 10., ...,  0.,  0., 10.],
       ...,
       [ 0.,  0.,  0., ..., 10.,  0., 10.],
       [ 0.,  0.,  0., ...,  0., 10., 10.],
       [ 0.,  0.,  0., ...,  0.,  0., 10.]], shape=(51, 51)))
```

We can now use our `MultivariateNormal` class to construct an instance, then partition the mean vector and covariance matrix as we wish.

We want to regress IQ, the random variable θ (*what we don't know*), on the vector y of test scores (*what we do know*).

We choose `k=n` so that $z_1 = y$ and $z_2 = \theta$.

```
multi_normal_IQ = MultivariateNormal(μ_IQ, Σ_IQ)

k = n
multi_normal_IQ.partition(k)
```

Using the generator `multivariate_normal`, we can make one draw of the random vector from our distribution and then compute the distribution of θ conditional on our test scores.

Let's do that and then print out some pertinent quantities.

```
x = rng.multivariate_normal(μ_IQ, Σ_IQ)
y = x[:-1] # test scores
θ = x[-1]  # IQ
```

```
# the true value
θ
```

```
np.float64(98.2495327983163)
```

The method `cond_dist` takes test scores y as input and returns the conditional normal distribution of the IQ θ.

In the following code, `ind` sets the variables on the right side of the regression.

Given the way we have defined the vector X, we want to set `ind=1` in order to make θ the left side variable in the population regression.

```
ind = 1
multi_normal_IQ.cond_dist(ind, y)
```

```
(array([97.20269774]), array([[1.96078431]]))
```

The first number is the conditional mean $\hat{\mu}_\theta$ and the second is the conditional variance $\hat{\Sigma}_\theta$.

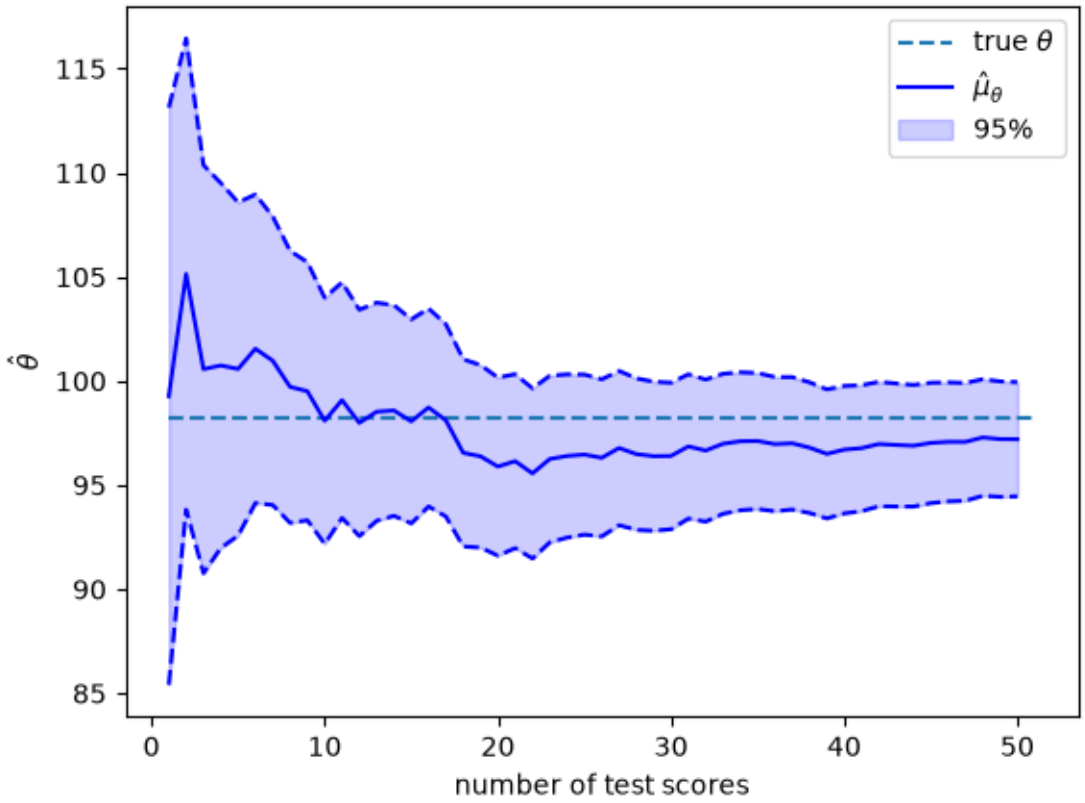
How do additional test scores affect our inferences?

To shed light on this, we compute a sequence of conditional distributions of θ by varying the number of test scores in the conditioning set from 1 to n.

We'll make a pretty graph showing how our judgment of the person's IQ change as more test results come in.

```
# array for containing moments
μθ_hat_arr = np.empty(n)
Σθ_hat_arr = np.empty(n)

# loop over number of test scores
for i in range(1, n+1):
    # construction of multivariate normal distribution instance
    μ_IQ_i, Σ_IQ_i, D_IQ_i = construct_moments_IQ(i, μθ, σθ, σy)
    multi_normal_IQ_i = MultivariateNormal(μ_IQ_i, Σ_IQ_i)

    # partition and compute conditional distribution
    multi_normal_IQ_i.partition(i)
    scores_i = y[:i]
    μθ_hat_i, Σθ_hat_i = multi_normal_IQ_i.cond_dist(1, scores_i)

    # store the results
    μθ_hat_arr[i-1] = μθ_hat_i[0]
    Σθ_hat_arr[i-1] = Σθ_hat_i[0, 0]

# transform variance to standard deviation
σθ_hat_arr = np.sqrt(Σθ_hat_arr)
```

```
μθ_hat_lower = μθ_hat_arr - 1.96 * σθ_hat_arr
μθ_hat_higher = μθ_hat_arr + 1.96 * σθ_hat_arr

plt.hlines(θ, 1, n+1, ls='--', label='true $θ$')
plt.plot(range(1, n+1), μθ_hat_arr, color='b', label=r'$\hat{μ}_{θ}$')
plt.plot(range(1, n+1), μθ_hat_lower, color='b', ls='--')
plt.plot(range(1, n+1), μθ_hat_higher, color='b', ls='--')
plt.fill_between(range(1, n+1), μθ_hat_lower, μθ_hat_higher,
                 color='b', alpha=0.2, label='95%')

plt.xlabel('number of test scores')
plt.ylabel(r'$\hat{θ}$')
plt.legend()

plt.show()
```



The solid blue line in the plot above shows $\hat{\mu}_\theta$ as a function of the number of test scores that we have recorded and conditioned on.

The blue area shows the span that comes from adding or subtracting $1.96\hat{\sigma}_\theta$ from $\hat{\mu}_\theta$.

Therefore, 95% of the probability mass of the conditional distribution falls in this range.

The value of the random θ that we drew is shown by the black dotted line.

As more and more test scores come in, our estimate of the person's θ become more and more reliable.

By staring at the changes in the conditional distributions, we see that adding more test scores makes $\hat{\theta}$ settle down and approach θ.

Thus, each y_i adds information about θ.

If we were to drive the number of tests $n \to +\infty$, the conditional standard deviation $\hat{\sigma}_\theta$ would converge to 0 at rate $\frac{1}{n^{.5}}$.

13.6 Information as surprise

By using a different representation, let's look at things from a different perspective.

We can represent the random vector X defined above as

$$X = \mu_\theta 1_{n+1} + C\epsilon, \quad \epsilon \sim N(0, I)$$

where C is a lower triangular **Cholesky factor** of Σ so that

$$\Sigma \equiv DD^\top = CC^\top$$

and

$$E\epsilon\epsilon^\top = I.$$

It follows that

$$\epsilon \sim N(0, I).$$

Let $G = C^{-1}$

G is also lower triangular.

We can compute ϵ from the formula

$$\epsilon = G\left(X - \mu_\theta 1_{n+1}\right)$$

This formula confirms that the orthonormal vector ϵ contains the same information as the non-orthogonal vector $(X - \mu_\theta 1_{n+1})$.

We can say that ϵ is an orthogonal basis for $(X - \mu_\theta 1_{n+1})$.

Let c_i be the ith element in the last row of C.

Then we can write

$$\theta = \mu_\theta + c_1\epsilon_1 + c_2\epsilon_2 + \cdots + c_n\epsilon_n + c_{n+1}\epsilon_{n+1} \tag{13.1}$$

The mutual orthogonality of the ϵ_i's provides us with an informative way to interpret them in light of equation (13.1).

Thus, relative to what is known from tests $i = 1, \ldots, n-1$, $c_i\epsilon_i$ is the amount of **new information** about θ brought by the test number i.

Here **new information** means **surprise** or what could not be predicted from earlier information.

Formula (13.1) also provides us with an enlightening way to express conditional means and conditional variances that we computed earlier.

In particular,

$$E\left[\theta \mid y_1, \ldots, y_k\right] = \mu_\theta + c_1\epsilon_1 + \cdots + c_k\epsilon_k$$

and

$$Var\left(\theta \mid y_1, \ldots, y_k\right) = c_{k+1}^2 + c_{k+2}^2 + \cdots + c_{n+1}^2.$$

```
C = np.linalg.cholesky(Σ_IQ)
G = np.linalg.inv(C)

ε = G @ (x - μθ)
```

```
cε = C[n, :] * ε

# compute the sequence of μθ and Σθ conditional on y1, y2, ..., yk
μθ_hat_arr_C = np.array([np.sum(cε[:k+1]) for k in range(n)]) + μθ
Σθ_hat_arr_C = np.array([C[n, i+1:n+1] @ C[n, i+1:n+1] for i in range(n)])
```

To confirm that these formulas give the same answers that we computed earlier, we can compare the means and variances of θ conditional on $\{y_i\}_{i=1}^k$ with what we obtained above using the formulas implemented in the class `MultivariateNormal` built on our original representation of conditional distributions for multivariate normal distributions.

```
# conditional mean
np.max(np.abs(μθ_hat_arr - μθ_hat_arr_C)) < 1e-10
```

```
np.True_
```

```
# conditional variance
np.max(np.abs(Σθ_hat_arr - Σθ_hat_arr_C)) < 1e-10
```

```
np.True_
```

13.7 Cholesky factor magic

Evidently, the Cholesky factorizations automatically computes the population **regression coefficients** and associated statistics that are produced by our `MultivariateNormal` class.

The Cholesky factorization computes these things **recursively**.

Indeed, in formula (13.1),

- the random variable $c_i \epsilon_i$ is information about θ that is not contained by the information in $\epsilon_1, \epsilon_2, \ldots, \epsilon_{i-1}$
- the coefficient c_i is the simple population regression coefficient of $\theta - \mu_\theta$ on ϵ_i

13.8 Math and verbal intelligence

We can alter the preceding example to be more realistic.

There is ample evidence that IQ is not a scalar.

Some people are good in math skills but poor in language skills.

Other people are good in language skills but poor in math skills.

So now we shall assume that there are two dimensions of IQ, θ and η.

These determine average performances in math and language tests, respectively.

We observe math scores $\{y_i\}_{i=1}^n$ and language scores $\{y_i\}_{i=n+1}^{2n}$.

When $n = 2$, we assume that outcomes are draws from a multivariate normal distribution with representation

$$X = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ \theta \\ \eta \end{bmatrix} = \begin{bmatrix} \mu_\theta \\ \mu_\theta \\ \mu_\eta \\ \mu_\eta \\ \mu_\theta \\ \mu_\eta \end{bmatrix} + \begin{bmatrix} \sigma_y & 0 & 0 & 0 & \sigma_\theta & 0 \\ 0 & \sigma_y & 0 & 0 & \sigma_\theta & 0 \\ 0 & 0 & \sigma_y & 0 & 0 & \sigma_\eta \\ 0 & 0 & 0 & \sigma_y & 0 & \sigma_\eta \\ 0 & 0 & 0 & 0 & \sigma_\theta & 0 \\ 0 & 0 & 0 & 0 & 0 & \sigma_\eta \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \\ w_5 \\ w_6 \end{bmatrix}$$

where $w = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_6 \end{bmatrix}$ is a standard normal random vector.

We construct a Python function `construct_moments_IQ2d` to construct the mean vector and covariance matrix of the joint normal distribution.

```
def construct_moments_IQ2d(n, μθ, σθ, μη, ση, σy):

    μ_IQ2d = np.empty(2*(n+1))
    μ_IQ2d[:n] = μθ
    μ_IQ2d[2*n] = μθ
    μ_IQ2d[n:2*n] = μη
    μ_IQ2d[2*n+1] = μη


    D_IQ2d = np.zeros((2*(n+1), 2*(n+1)))
    D_IQ2d[range(2*n), range(2*n)] = σy
    D_IQ2d[:n, 2*n] = σθ
    D_IQ2d[2*n, 2*n] = σθ
    D_IQ2d[n:2*n, 2*n+1] = ση
    D_IQ2d[2*n+1, 2*n+1] = ση

    Σ_IQ2d = D_IQ2d @ D_IQ2d.T

    return μ_IQ2d, Σ_IQ2d, D_IQ2d
```

Let's put the function to work.

```
n = 2
# mean and variance of θ, η, and y
μθ, σθ, μη, ση, σy = 100., 10., 100., 10, 10

μ_IQ2d, Σ_IQ2d, D_IQ2d = construct_moments_IQ2d(n, μθ, σθ, μη, ση, σy)
μ_IQ2d, Σ_IQ2d, D_IQ2d
```

```
(array([100., 100., 100., 100., 100., 100.]),
 array([[200., 100.,   0.,   0., 100.,   0.],
        [100., 200.,   0.,   0., 100.,   0.],
        [  0.,   0., 200., 100.,   0., 100.],
        [  0.,   0., 100., 200.,   0., 100.],
        [100., 100.,   0.,   0., 100.,   0.],
        [  0.,   0., 100., 100.,   0., 100.]]),
 array([[10.,  0.,  0.,  0., 10.,  0.],
        [ 0., 10.,  0.,  0., 10.,  0.],
        [ 0.,  0., 10.,  0.,  0., 10.],
        [ 0.,  0.,  0., 10.,  0., 10.],
        [ 0.,  0.,  0.,  0., 10.,  0.],
        [ 0.,  0.,  0.,  0.,  0., 10.]]))
```

```
# take one draw
x = rng.multivariate_normal(μ_IQ2d, Σ_IQ2d)
y1 = x[:n]
y2 = x[n:2*n]
θ = x[2*n]
η = x[2*n+1]

# the true values
θ, η
```

```
(np.float64(86.84580129140488), np.float64(98.5057398320253))
```

We first compute the joint normal distribution of (θ, η).

```
multi_normal_IQ2d = MultivariateNormal(μ_IQ2d, Σ_IQ2d)

k = 2*n # the length of data vector
multi_normal_IQ2d.partition(k)

multi_normal_IQ2d.cond_dist(1, [*y1, *y2])
```

```
(array([ 87.15090791, 104.56448792]),
 array([[33.33333333,  0.        ],
        [ 0.        , 33.33333333]]))
```

Now let's compute distributions of θ and η separately conditional on various subsets of test scores.

It will be fun to compare outcomes with the help of an auxiliary function `cond_dist_IQ2d` that we now construct.

```
def cond_dist_IQ2d(μ, Σ, data):

    n = len(μ)

    multi_normal = MultivariateNormal(μ, Σ)
    multi_normal.partition(n-1)
    μ_hat, Σ_hat = multi_normal.cond_dist(1, data)

    return μ_hat, Σ_hat
```

Let's see how things work for an example.

```
for indices, IQ, conditions in [([*range(2*n), 2*n], 'θ', 'y1, y2, y3, y4'),
                                ([*range(n), 2*n], 'θ', 'y1, y2'),
                                ([*range(n, 2*n), 2*n], 'θ', 'y3, y4'),
                                ([*range(2*n), 2*n+1], 'η', 'y1, y2, y3, y4'),
                                ([*range(n), 2*n+1], 'η', 'y1, y2'),
                                ([*range(n, 2*n), 2*n+1], 'η', 'y3, y4')]:

    μ_hat, Σ_hat = cond_dist_IQ2d(μ_IQ2d[indices], Σ_IQ2d[indices][:, indices],␣
↪x[indices[:-1]])
    print(f'The mean and variance of {IQ} conditional on {conditions: <15} are ' +
          f'{μ_hat[0]:1.2f} and {Σ_hat[0, 0]:1.2f} respectively')
```

```
The mean and variance of θ conditional on y1, y2, y3, y4  are 87.15 and 33.33␣
↪respectively
The mean and variance of θ conditional on y1, y2          are 87.15 and 33.33␣
↪respectively
The mean and variance of θ conditional on y3, y4          are 100.00 and 100.00␣
↪respectively
The mean and variance of η conditional on y1, y2, y3, y4  are 104.56 and 33.33␣
↪respectively
The mean and variance of η conditional on y1, y2          are 100.00 and 100.00␣
↪respectively
The mean and variance of η conditional on y3, y4          are 104.56 and 33.33␣
↪respectively
```

Evidently, math tests provide no information about η and language tests provide no information about θ.

13.9 Univariate time series analysis

We can use the multivariate normal distribution and a little matrix algebra to present foundations of univariate linear time series analysis.

Let x_t, y_t, v_t, w_{t+1} each be scalars for $t \geq 0$.

Consider the following model:

$$
\begin{aligned}
x_0 &\sim N\left(0, \sigma_0^2\right) \\
x_{t+1} &= ax_t + bw_{t+1}, \quad w_{t+1} \sim N\left(0,1\right), t \geq 0 \\
y_t &= cx_t + dv_t, \quad v_t \sim N\left(0,1\right), t \geq 0
\end{aligned}
$$

We can compute the moments of x_t

1. $Ex_{t+1}^2 = a^2 Ex_t^2 + b^2, t \geq 0$, where $Ex_0^2 = \sigma_0^2$
2. $Ex_{t+j}x_t = a^j Ex_t^2, \forall t \; \forall j$

Given some T, we can formulate the sequence $\{x_t\}_{t=0}^T$ as a random vector

$$
X = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_T \end{bmatrix}
$$

and the covariance matrix Σ_x can be constructed using the moments we have computed above.

Similarly, we can define

$$
Y = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_T \end{bmatrix}, \quad v = \begin{bmatrix} v_0 \\ v_1 \\ \vdots \\ v_T \end{bmatrix}
$$

and therefore

$$
Y = CX + DV
$$

where C and D are both diagonal matrices with constant c and d as diagonal respectively.

Consequently, the covariance matrix of Y is

$$
\Sigma_y = EYY^\top = C\Sigma_x C^\top + DD^\top
$$

By stacking X and Y, we can write

$$
Z = \begin{bmatrix} X \\ Y \end{bmatrix}
$$

and

$$
\Sigma_z = EZZ^\top = \begin{bmatrix} \Sigma_x & \Sigma_x C^\top \\ C\Sigma_x & \Sigma_y \end{bmatrix}
$$

Thus, the stacked sequences $\{x_t\}_{t=0}^T$ and $\{y_t\}_{t=0}^T$ jointly follow the multivariate normal distribution $N\left(0, \Sigma_z\right)$.

```
# as an example, consider the case where T = 3
T = 3
```

```
# variance of the initial distribution x_0
σ0 = 1.

# parameters of the equation system
a = .9
b = 1.
c = 1.0
d = .05
```

```
# construct the covariance matrix of X
Σx = np.empty((T+1, T+1))

Σx[0, 0] = σ0 ** 2
for i in range(T):
    Σx[i, i+1:] = Σx[i, i] * a ** np.arange(1, T+1-i)
    Σx[i+1:, i] = Σx[i, i+1:]

    Σx[i+1, i+1] = a ** 2 * Σx[i, i] + b ** 2
```

```
Σx
```

```
array([[1.      , 0.9     , 0.81    , 0.729   ],
       [0.9     , 1.81    , 1.629   , 1.4661  ],
       [0.81    , 1.629   , 2.4661  , 2.21949 ],
       [0.729   , 1.4661  , 2.21949 , 2.997541]])
```

```
# construct the covariance matrix of Y
C = np.eye(T+1) * c
D = np.eye(T+1) * d

Σy = C @ Σx @ C.T + D @ D.T
```

```
# construct the covariance matrix of Z
Σz = np.empty((2*(T+1), 2*(T+1)))

Σz[:T+1, :T+1] = Σx
Σz[:T+1, T+1:] = Σx @ C.T
Σz[T+1:, :T+1] = C @ Σx
Σz[T+1:, T+1:] = Σy
```

```
Σz
```

```
array([[1.      , 0.9     , 0.81    , 0.729   , 1.      , 0.9     ,
        0.81    , 0.729   ],
       [0.9     , 1.81    , 1.629   , 1.4661  , 0.9     , 1.81    ,
        1.629   , 1.4661  ],
       [0.81    , 1.629   , 2.4661  , 2.21949 , 0.81    , 1.629   ,
        2.4661  , 2.21949 ],
       [0.729   , 1.4661  , 2.21949 , 2.997541, 0.729   , 1.4661  ,
        2.21949 , 2.997541],
       [1.      , 0.9     , 0.81    , 0.729   , 1.0025  , 0.9     ,
        0.81    , 0.729   ],
       [0.9     , 1.81    , 1.629   , 1.4661  , 0.9     , 1.8125  ,
        1.629   , 1.4661  ],
       [0.81    , 1.629   , 2.4661  , 2.21949 , 0.81    , 1.629   ,
```

(continues on next page)

(continued from previous page)

```
        2.4686  , 2.21949 ],
       [0.729   , 1.4661  , 2.21949 , 2.997541, 0.729   , 1.4661  ,
        2.21949 , 3.000041]])
```

```
# construct the mean vector of Z
μz = np.zeros(2*(T+1))
```

The following Python code lets us sample random vectors X and Y.

This is going to be very useful for doing the conditioning to be used in the fun exercises below.

```
z = rng.multivariate_normal(μz, Σz)

x = z[:T+1]
y = z[T+1:]
```

13.9.1 Smoothing example

This is an instance of a classic `smoothing` calculation whose purpose is to compute $EX \mid Y$.

An interpretation of this example is

- X is a random sequence of hidden Markov state variables x_t
- Y is a sequence of observed signals y_t bearing information about the hidden state

```
# construct a MultivariateNormal instance
multi_normal_ex1 = MultivariateNormal(μz, Σz)
x = z[:T+1]
y = z[T+1:]
```

```
# partition Z into X and Y
multi_normal_ex1.partition(T+1)
```

```
# compute the conditional mean and covariance matrix of X given Y=y

print("X = ", x)
print("Y = ", y)
print(" E [ X | Y] = ", )

multi_normal_ex1.cond_dist(0, y)
```

```
X =  [ 0.241748   -0.48167052 -0.2751467   0.62696378]
Y =  [ 0.2465054  -0.55890629 -0.28918582  0.57124943]
 E [ X | Y] =
```

```
(array([ 0.24414853, -0.55648653, -0.28785472,  0.56917881]),
 array([[2.48875094e-03, 5.57449314e-06, 1.24861727e-08, 2.80242496e-11],
        [5.57449314e-06, 2.48876343e-03, 5.57452116e-06, 1.25113948e-08],
        [1.24861727e-08, 5.57452116e-06, 2.48876346e-03, 5.58575339e-06],
        [2.80242496e-11, 1.25113948e-08, 5.58575339e-06, 2.49377812e-03]]))
```

13.9.2 Filtering exercise

Compute $E\left[x_t \mid y_{t-1}, y_{t-2}, \dots, y_0\right]$.

To do so, we need to first construct the mean vector and the covariance matrix of the subvector $\left[x_t, y_0, \dots, y_{t-2}, y_{t-1}\right]$.

For example, let's say that we want the conditional distribution of x_3.

```
t = 3
```

```
# mean of the subvector
sub_μz = np.zeros(t+1)

# covariance matrix of the subvector
sub_Σz = np.empty((t+1, t+1))

sub_Σz[0, 0] = Σz[t, t] # x_t
sub_Σz[0, 1:] = Σz[t, T+1:T+t+1]
sub_Σz[1:, 0] = Σz[T+1:T+t+1, t]
sub_Σz[1:, 1:] = Σz[T+1:T+t+1, T+1:T+t+1]
```

```
sub_Σz
```

```
array([[2.997541, 0.729   , 1.4661  , 2.21949 ],
       [0.729   , 1.0025  , 0.9     , 0.81    ],
       [1.4661  , 0.9     , 1.8125  , 1.629   ],
       [2.21949 , 0.81    , 1.629   , 2.4686  ]])
```

```
multi_normal_ex2 = MultivariateNormal(sub_μz, sub_Σz)
multi_normal_ex2.partition(1)
```

```
sub_y = y[:t]

multi_normal_ex2.cond_dist(0, sub_y)
```

```
(array([-0.26074227]), array([[1.00201996]]))
```

13.9.3 Prediction exercise

Compute $E\left[y_t \mid y_{t-j}, \dots, y_0\right]$.

As what we did in exercise 2, we will construct the mean vector and covariance matrix of the subvector $\left[y_t, y_0, \dots, y_{t-j-1}, y_{t-j}\right]$.

For example, we take a case in which $t = 3$ and $j = 2$.

```
t = 3
j = 2
```

```
sub_μz = np.zeros(t-j+2)
sub_Σz = np.empty((t-j+2, t-j+2))

sub_Σz[0, 0] = Σz[T+t+1, T+t+1]
sub_Σz[0, 1:] = Σz[T+t+1, T+1:T+t-j+2]
```

(continues on next page)

(continued from previous page)

```
sub_Σz[1:, 0] = Σz[T+1:T+t-j+2, T+t+1]
sub_Σz[1:, 1:] = Σz[T+1:T+t-j+2, T+1:T+t-j+2]
```

```
sub_Σz
```

```
array([[3.000041, 0.729   , 1.4661  ],
       [0.729   , 1.0025  , 0.9     ],
       [1.4661  , 0.9     , 1.8125  ]])
```

```
multi_normal_ex3 = MultivariateNormal(sub_μz, sub_Σz)
multi_normal_ex3.partition(1)
```

```
sub_y = y[:t-j+1]

multi_normal_ex3.cond_dist(0, sub_y)
```

```
(array([-0.45114129]), array([[1.81413617]]))
```

13.9.4 Constructing a Wold representation

Now we'll apply Cholesky decomposition to decompose $\Sigma_y = HH^\top$ and form

$$\epsilon = H^{-1}Y.$$

Then we can represent y_t as

$$y_t = h_{t,t}\epsilon_t + h_{t,t-1}\epsilon_{t-1} + \cdots + h_{t,0}\epsilon_0.$$

```
H = np.linalg.cholesky(Σy)

H
```

```
array([[1.00124922, 0.        , 0.        , 0.        ],
       [0.8988771 , 1.00225743, 0.        , 0.        ],
       [0.80898939, 0.89978675, 1.00225743, 0.        ],
       [0.72809046, 0.80980808, 0.89978676, 1.00225743]])
```

```
ε = np.linalg.inv(H) @ y

ε
```

```
array([ 0.24619784, -0.7784506 ,  0.2116046 ,  0.83011777])
```

```
y
```

```
array([ 0.2465054 , -0.55890629, -0.28918582,  0.57124943])
```

This example is an instance of what is known as a **Wold representation** in time series analysis.

13.10 Stochastic difference equation

Consider the stochastic second-order linear difference equation

$$y_t = \alpha_0 + \alpha_1 y_{t-1} + \alpha_2 y_{t-2} + u_t$$

where $u_t \sim N\left(0, \sigma_u^2\right)$ and

$$\begin{bmatrix} y_{-1} \\ y_0 \end{bmatrix} \sim N\left(\mu_{\tilde{y}}, \Sigma_{\tilde{y}}\right)$$

It can be written as a stacked system

$$\underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ -\alpha_1 & 1 & 0 & 0 & \cdots & 0 & 0 & 0 \\ -\alpha_2 & -\alpha_1 & 1 & 0 & \cdots & 0 & 0 & 0 \\ 0 & -\alpha_2 & -\alpha_1 & 1 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \cdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & -\alpha_2 & -\alpha_1 & 1 \end{bmatrix}}_{\equiv A} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ \vdots \\ y_T \end{bmatrix} = \underbrace{\begin{bmatrix} \alpha_0 + \alpha_1 y_0 + \alpha_2 y_{-1} \\ \alpha_0 + \alpha_2 y_0 \\ \alpha_0 \\ \alpha_0 \\ \vdots \\ \alpha_0 \end{bmatrix}}_{\equiv b} + \underbrace{\begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ \vdots \\ u_T \end{bmatrix}}_{\equiv u}$$

We can compute y by solving the system

$$y = A^{-1}\left(b + u\right)$$

We have

$$\begin{aligned} \mu_y &= A^{-1}\mu_b \\ \Sigma_y &= A^{-1}E\left[\left(b - \mu_b + u\right)\left(b - \mu_b + u\right)^\top\right]\left(A^{-1}\right)^\top \\ &= A^{-1}\left(\Sigma_b + \Sigma_u\right)\left(A^{-1}\right)^\top \end{aligned}$$

where

$$\mu_b = \begin{bmatrix} \alpha_0 + \alpha_1 \mu_{y_0} + \alpha_2 \mu_{y_{-1}} \\ \alpha_0 + \alpha_2 \mu_{y_0} \\ \alpha_0 \\ \vdots \\ \alpha_0 \end{bmatrix}$$

$$\Sigma_b = \begin{bmatrix} C\Sigma_{\tilde{y}} C^\top & 0_{N-2\times N-2} \\ 0_{N-2\times 2} & 0_{N-2\times N-2} \end{bmatrix}, \quad C = \begin{bmatrix} \alpha_2 & \alpha_1 \\ 0 & \alpha_2 \end{bmatrix}$$

$$\Sigma_u = \begin{bmatrix} \sigma_u^2 & 0 & \cdots & 0 \\ 0 & \sigma_u^2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & \sigma_u^2 \end{bmatrix}$$

```
# set parameters
T = 160
# coefficients of the second order difference equation
a0 = 10
a1 = 1.53
a2 = -.9

# variance of u
σu = 10.

# distribution of y_{-1} and y_{0}
μy_tilde = np.array([1., 0.5])
Σy_tilde = np.array([[2., 1.], [1., 0.5]])
```

```
# construct A and A^\top
A = np.zeros((T, T))

for i in range(T):
    A[i, i] = 1

    if i-1 >= 0:
        A[i, i-1] = -α1

    if i-2 >= 0:
        A[i, i-2] = -α2

A_inv = np.linalg.inv(A)
```

```
# compute the mean vectors of b and y
μb = np.full(T, α0)
μb[0] += α1 * μy_tilde[1] + α2 * μy_tilde[0]
μb[1] += α2 * μy_tilde[1]

μy = A_inv @ μb
```

```
# compute the covariance matrices of b and y
Σu = np.eye(T) * σu ** 2

Σb = np.zeros((T, T))

C = np.array([[α2, α1], [0, α2]])
Σb[:2, :2] = C @ Σy_tilde @ C.T

Σy = A_inv @ (Σb + Σu) @ A_inv.T
```

13.11 Application to stock price model

Let

$$p_t = \sum_{j=0}^{T-t} \beta^j y_{t+j}$$

Form

$$\underbrace{\begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ \vdots \\ p_T \end{bmatrix}}_{\equiv p} = \underbrace{\begin{bmatrix} 1 & \beta & \beta^2 & \cdots & \beta^{T-1} \\ 0 & 1 & \beta & \cdots & \beta^{T-2} \\ 0 & 0 & 1 & \cdots & \beta^{T-3} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix}}_{\equiv B} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_T \end{bmatrix}$$

we have

$$\mu_p = B\mu_y$$
$$\Sigma_p = B\Sigma_y B^\top$$

```
β = .96
```

```
# construct B
B = np.zeros((T, T))

for i in range(T):
    B[i, i:] = β ** np.arange(0, T-i)
```

Denote

$$z = \begin{bmatrix} y \\ p \end{bmatrix} = \underbrace{\begin{bmatrix} I \\ B \end{bmatrix}}_{\equiv D} y$$

Thus, $\{y_t\}_{t=1}^T$ and $\{p_t\}_{t=1}^T$ jointly follow the multivariate normal distribution $N(\mu_z, \Sigma_z)$, where

$$\mu_z = D\mu_y$$

$$\Sigma_z = D\Sigma_y D^\top$$

```
D = np.vstack([np.eye(T), B])
```

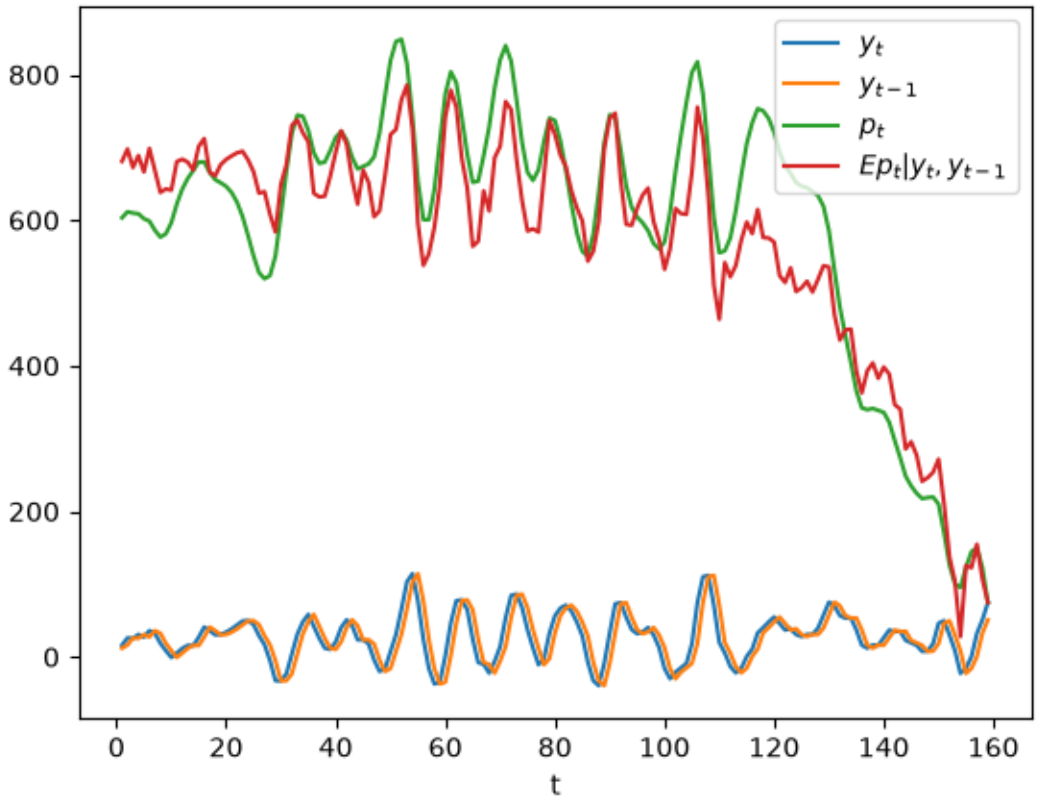
```
μz = D @ μy
Σz = D @ Σy @ D.T
```

We can simulate paths of y_t and p_t and compute the conditional mean $E[p_t \mid y_{t-1}, y_t]$ using the `MultivariateNormal` class.

```
z = rng.multivariate_normal(μz, Σz)
y, p = z[:T], z[T:]
```

```
cond_Ep = np.empty(T-1)

sub_μ = np.empty(3)
sub_Σ = np.empty((3, 3))
for t in range(2, T+1):
    sub_μ[:] = μz[[t-2, t-1, T-1+t]]
    sub_Σ[:, :] = Σz[[t-2, t-1, T-1+t], :][:, [t-2, t-1, T-1+t]]

    multi_normal = MultivariateNormal(sub_μ, sub_Σ)
    multi_normal.partition(2)

    cond_Ep[t-2] = multi_normal.cond_dist(1, y[t-2:t])[0][0]
```

```
plt.plot(range(1, T), y[1:], label='$y_{t}$')
plt.plot(range(1, T), y[:-1], label='$y_{t-1}$')
plt.plot(range(1, T), p[1:], label='$p_{t}$')
plt.plot(range(1, T), cond_Ep, label='$Ep_{t}|y_{t}, y_{t-1}$')

plt.xlabel('t')
plt.legend(loc=1)
plt.show()
```



In the above graph, the green line is what the price of the stock would be if people had perfect foresight about the path of dividends while the red line is the conditional expectation $Ep_t|y_t, y_{t-1}$, which is what the price would be if people did not have perfect foresight but were optimally predicting future dividends on the basis of the information y_t, y_{t-1} at time t.

13.12 Filtering foundations

Assume that x_0 is an $n \times 1$ random vector and that y_0 is a $p \times 1$ random vector determined by the *observation equation*

$$y_0 = Gx_0 + v_0, \quad x_0 \sim \mathcal{N}(\hat{x}_0, \Sigma_0), \quad v_0 \sim \mathcal{N}(0, R)$$

where v_0 is orthogonal to x_0, G is a $p \times n$ matrix, and R is a $p \times p$ positive definite matrix.

We consider the problem of someone who

- *observes* y_0
- does not observe x_0,
- knows $\hat{x}_0, \Sigma_0, G, R$ and therefore the joint probability distribution of the vector $\begin{bmatrix} x_0 \\ y_0 \end{bmatrix}$
- wants to infer x_0 from y_0 in light of what he knows about that joint probability distribution.

Therefore, the person wants to construct the probability distribution of x_0 conditional on the random vector y_0.

The joint distribution of $\begin{bmatrix} x_0 \\ y_0 \end{bmatrix}$ is multivariate normal $\mathcal{N}(\mu, \Sigma)$ with

$$\mu = \begin{bmatrix} \hat{x}_0 \\ G\hat{x}_0 \end{bmatrix}, \quad \Sigma = \begin{bmatrix} \Sigma_0 & \Sigma_0 G' \\ G\Sigma_0 & G\Sigma_0 G' + R \end{bmatrix}$$

By applying an appropriate instance of the above formulas for the mean vector $\hat{\mu}_1$ and covariance matrix $\hat{\Sigma}_{11}$ of z_1 conditional on z_2, we find that the probability distribution of x_0 conditional on y_0 is $\mathcal{N}(\tilde{x}_0, \tilde{\Sigma}_0)$ where

$$\begin{aligned}\beta_0 &= \Sigma_0 G'(G\Sigma_0 G' + R)^{-1} \\ \tilde{x}_0 &= \hat{x}_0 + \beta_0(y_0 - G\hat{x}_0) \\ \tilde{\Sigma}_0 &= \Sigma_0 - \Sigma_0 G'(G\Sigma_0 G' + R)^{-1} G\Sigma_0\end{aligned}$$

We can express our finding that the probability distribution of x_0 conditional on y_0 is $\mathcal{N}(\tilde{x}_0, \tilde{\Sigma}_0)$ by representing x_0 as

$$x_0 = \tilde{x}_0 + \zeta_0 \tag{13.2}$$

where ζ_0 is a Gaussian random vector that is orthogonal to $\tilde{x}_0$ and y_0 and that has mean vector 0 and conditional covariance matrix $E[\zeta_0 \zeta_0' | y_0] = \tilde{\Sigma}_0$.

13.12.1 Step toward dynamics

Now suppose that we are in a time series setting and that we have the one-step state transition equation

$$x_1 = Ax_0 + Cw_1, \quad w_1 \sim \mathcal{N}(0, I)$$

where A is an $n \times n$ matrix and C is an $n \times m$ matrix.

Using equation (13.2), we can also represent x_1 as

$$x_1 = A(\tilde{x}_0 + \zeta_0) + Cw_1$$

It follows that

$$Ex_1 | y_0 = A\tilde{x}_0$$

and that the corresponding conditional covariance matrix $E(x_1 - Ex_1|y_0)(x_1 - Ex_1|y_0)' \equiv \Sigma_1$ is

$$\Sigma_1 = A\tilde{\Sigma}_0 A' + CC'$$

or

$$\Sigma_1 = A\Sigma_0 A' - A\Sigma_0 G'(G\Sigma_0 G' + R)^{-1} G\Sigma_0 A'$$

We can write the mean of x_1 conditional on y_0 as

$$\hat{x}_1 = A\hat{x}_0 + A\Sigma_0 G'(G\Sigma_0 G' + R)^{-1}(y_0 - G\hat{x}_0)$$

or

$$\hat{x}_1 = A\hat{x}_0 + K_0(y_0 - G\hat{x}_0)$$

where

$$K_0 = A\Sigma_0 G'(G\Sigma_0 G' + R)^{-1}$$

13.12.2 Dynamic version

Suppose now that for $t \geq 0$, $\{x_{t+1}, y_t\}_{t=0}^{\infty}$ are governed by the equations

$$
\begin{aligned}
x_{t+1} &= Ax_t + Cw_{t+1} \\
y_t &= Gx_t + v_t
\end{aligned}
$$

where as before $x_0 \sim \mathcal{N}(\hat{x}_0, \Sigma_0)$, w_{t+1} is the $t+1$th component of an i.i.d. stochastic process distributed as $w_{t+1} \sim \mathcal{N}(0, I)$, and v_t is the tth component of an i.i.d. process distributed as $v_t \sim \mathcal{N}(0, R)$ and the $\{w_{t+1}\}_{t=0}^{\infty}$ and $\{v_t\}_{t=0}^{\infty}$ processes are orthogonal at all pairs of dates.

The logic and formulas that we applied above imply that the probability distribution of x_t conditional on $y_0, y_1, \ldots, y_{t-1} = y^{t-1}$ is

$$
x_t | y^{t-1} \sim \mathcal{N}(A\tilde{x}_{t-1}, A\tilde{\Sigma}_{t-1}A' + CC')
$$

where $\{\tilde{x}_t, \tilde{\Sigma}_t\}_{t=1}^{\infty}$ can be computed by iterating on the following equations starting from $t = 1$ and initial conditions for $\tilde{x}_0, \tilde{\Sigma}_0$ computed as we have above:

$$
\begin{aligned}
\Sigma_t &= A\tilde{\Sigma}_{t-1}A' + CC' \\
\hat{x}_t &= A\tilde{x}_{t-1} \\
\beta_t &= \Sigma_t G'(G\Sigma_t G' + R)^{-1} \\
\tilde{x}_t &= \hat{x}_t + \beta_t(y_t - G\hat{x}_t) \\
\tilde{\Sigma}_t &= \Sigma_t - \Sigma_t G'(G\Sigma_t G' + R)^{-1} G\Sigma_t
\end{aligned}
$$

If we shift the first equation forward one period and then substitute the expression for $\tilde{\Sigma}_t$ on the right side of the fifth equation into it we obtain

$$
\Sigma_{t+1} = CC' + A\Sigma_t A' - A\Sigma_t G'(G\Sigma_t G' + R)^{-1} G\Sigma_t A'.
$$

This is a matrix Riccati difference equation that is closely related to another matrix Riccati difference equation that appears in a quantecon lecture on the basics of linear quadratic control theory.

That equation has the form

$$
P_{t-1} = R + A'P_t A - A'P_t B(B'P_t B + Q)^{-1} B'P_t A.
$$

Stare at the two preceding equations for a moment or two, the first being a matrix difference equation for a conditional covariance matrix, the second being a matrix difference equation in the matrix appearing in a quadratic form for an intertemporal cost of value function.

Although the two equations are not identical, they display striking family resemblances.

- the first equation tells dynamics that work **forward** in time
- the second equation tells dynamics that work **backward** in time
- while many of the terms are similar, one equation seems to apply matrix transformations to some matrices that play similar roles in the other equation

The family resemblences of these two equations reflects a transcendent **duality** that prevails between control theory and filtering theory.

13.12.3 An example

We can use the Python class *MultivariateNormal* to construct examples.

Here is an example for a single period problem at time 0

```
G = np.array([[1., 3.]])
R = np.array([[1.]])

x0_hat = np.array([0., 1.])
Σ0 = np.array([[1., .5], [.5, 2.]])

μ = np.hstack([x0_hat, G @ x0_hat])
Σ = np.block([[Σ0, Σ0 @ G.T], [G @ Σ0, G @ Σ0 @ G.T + R]])
```

```
# construction of the multivariate normal instance
multi_normal = MultivariateNormal(μ, Σ)
```

```
multi_normal.partition(2)
```

```
# the observation of y
y0 = 2.3

# conditional distribution of x0
μ1_hat, Σ11 = multi_normal.cond_dist(0, y0)
μ1_hat, Σ11
```

```
(array([-0.07608696,  0.80217391]),
 array([[ 0.72826087, -0.20652174],
        [-0.20652174,  0.16304348]]))
```

```
A = np.array([[0.5, 0.2], [-0.1, 0.3]])
C = np.array([[2.], [1.]])

# conditional distribution of x1
x1_cond = A @ μ1_hat
Σ1_cond = C @ C.T + A @ Σ11 @ A.T
x1_cond, Σ1_cond
```

```
(array([0.1223913 , 0.24826087]),
 array([[4.14728261, 1.94652174],
        [1.94652174, 1.03434783]]))
```

13.12.4 Code for iterating

Here is code for solving a dynamic filtering problem by iterating on our equations, followed by an example.

```
def iterate(x0_hat, Σ0, A, C, G, R, y_seq):

    p, n = G.shape

    T = len(y_seq)
    x_hat_seq = np.empty((T+1, n))
    Σ_hat_seq = np.empty((T+1, n, n))
```

(continues on next page)

(continued from previous page)

```
    x_hat_seq[0] = x0_hat
    Σ_hat_seq[0] = Σ0

    for t in range(T):
        xt_hat = x_hat_seq[t]
        Σt = Σ_hat_seq[t]
        μ = np.hstack([xt_hat, G @ xt_hat])
        Σ = np.block([[Σt, Σt @ G.T], [G @ Σt, G @ Σt @ G.T + R]])

        # filtering
        multi_normal = MultivariateNormal(μ, Σ)
        multi_normal.partition(n)
        x_tilde, Σ_tilde = multi_normal.cond_dist(0, y_seq[t])

        # forecasting
        x_hat_seq[t+1] = A @ x_tilde
        Σ_hat_seq[t+1] = C @ C.T + A @ Σ_tilde @ A.T

    return x_hat_seq, Σ_hat_seq
```

```
iterate(x0_hat, Σ0, A, C, G, R, [2.3, 1.2, 3.2])
```

```
(array([[0.        , 1.        ],
        [0.1223913 , 0.24826087],
        [0.18729801, 0.06881254],
        [0.75600369, 0.05547207]]),
 array([[[1.        , 0.5       ],
         [0.5       , 2.        ]],

        [[4.14728261, 1.94652174],
         [1.94652174, 1.03434783]],

        [[4.08853181, 1.98917687],
         [1.98917687, 1.00757802]],

        [[4.06650435, 1.99952946],
         [1.99952946, 1.00271293]]]))
```

The iterative algorithm just described is a version of the celebrated **Kalman filter**.

We describe the Kalman filter and some applications of it in *A First Look at the Kalman Filter*

13.13 Classic factor analysis model

The factor analysis model can be represented as

$$Y = \Lambda f + U$$

where

1. Y is $n \times 1$ random vector, $EUU^\top = D$ is a diagonal matrix,
2. Λ is $n \times k$ coefficient matrix,
3. f is $k \times 1$ random vector, $Eff^\top = I$,

4. U is $n \times 1$ random vector, and $U \perp f$ (i.e., $EUf^\top = 0$)
5. It is presumed that k is small relative to n; often k is only 1 or 2, as in our IQ examples.

This implies that

$$\begin{aligned} \Sigma_y = EYY^\top &= \Lambda\Lambda^\top + D \\ EYf^\top &= \Lambda \\ EfY^\top &= \Lambda^\top \end{aligned}$$

Thus, the covariance matrix Σ_Y is the sum of a diagonal matrix D and a positive semi-definite matrix $\Lambda\Lambda^\top$ of rank k.

This means that all covariances among the n components of the Y vector are intermediated by their common dependencies on the k factors.

Form

$$Z = \begin{pmatrix} f \\ Y \end{pmatrix}$$

the covariance matrix of the expanded random vector Z can be computed as

$$\Sigma_z = EZZ^\top = \begin{pmatrix} I & \Lambda^\top \\ \Lambda & \Lambda\Lambda^\top + D \end{pmatrix}$$

In the following, we first construct the mean vector and the covariance matrix for the case where $N = 10$ and $k = 2$.

```
N = 10
k = 2
```

We set the coefficient matrix Λ and the covariance matrix of U to be

$$\Lambda = \begin{pmatrix} 1 & 0 \\ \vdots & \vdots \\ 1 & 0 \\ 0 & 1 \\ \vdots & \vdots \\ 0 & 1 \end{pmatrix}, \quad D = \begin{pmatrix} \sigma_u^2 & 0 & \cdots & 0 \\ 0 & \sigma_u^2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & \sigma_u^2 \end{pmatrix}$$

where the first half of the first column of Λ is filled with 1s and 0s for the rest half, and symmetrically for the second column.

D is a diagonal matrix with parameter σ_u^2 on the diagonal.

```
Λ = np.zeros((N, k))
Λ[:N//2, 0] = 1
Λ[N//2:, 1] = 1

σu = .5
D = np.eye(N) * σu ** 2
```

```
# compute Σy
Σy = Λ @ Λ.T + D
```

We can now construct the mean vector and the covariance matrix for Z.

```
μz = np.zeros(k+N)

Σz = np.empty((k+N, k+N))
```

(continues on next page)

(continued from previous page)

```
Σz[:k, :k] = np.eye(k)
Σz[:k, k:] = Λ.T
Σz[k:, :k] = Λ
Σz[k:, k:] = Σy
```

```
z = rng.multivariate_normal(μz, Σz)

f = z[:k]
y = z[k:]
```

```
multi_normal_factor = MultivariateNormal(μz, Σz)
multi_normal_factor.partition(k)
```

Let's compute the conditional distribution of the hidden factor f on the observations Y, namely, $f \mid Y = y$.

```
multi_normal_factor.cond_dist(0, y)
```

```
(array([0.14356426, 1.84615504]),
 array([[0.04761905, 0.        ],
        [0.        , 0.04761905]]))
```

We can verify that the conditional mean $E[f \mid Y = y] = BY$ where $B = \Lambda^\top \Sigma_y^{-1}$.

```
B = Λ.T @ np.linalg.inv(Σy)

B @ y
```

```
array([0.14356426, 1.84615504])
```

Similarly, we can compute the conditional distribution $Y \mid f$.

```
multi_normal_factor.cond_dist(1, f)
```

```
(array([0.41858317, 0.41858317, 0.41858317, 0.41858317, 0.41858317,
        2.04367341, 2.04367341, 2.04367341, 2.04367341, 2.04367341]),
 array([[0.25, 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ],
        [0.  , 0.25, 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ],
        [0.  , 0.  , 0.25, 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ],
        [0.  , 0.  , 0.  , 0.25, 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ],
        [0.  , 0.  , 0.  , 0.  , 0.25, 0.  , 0.  , 0.  , 0.  , 0.  ],
        [0.  , 0.  , 0.  , 0.  , 0.  , 0.25, 0.  , 0.  , 0.  , 0.  ],
        [0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.25, 0.  , 0.  , 0.  ],
        [0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.25, 0.  , 0.  ],
        [0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.25, 0.  ],
        [0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.25]]))
```

It can be verified that the mean is $\Lambda I^{-1} f = \Lambda f$.

```
Λ @ f
```

```
array([0.41858317, 0.41858317, 0.41858317, 0.41858317, 0.41858317,
       2.04367341, 2.04367341, 2.04367341, 2.04367341, 2.04367341])
```

13.14 PCA and factor analysis

To learn about Principal Components Analysis (PCA), please see this lecture *Singular Value Decompositions*.

For fun, let's apply a PCA decomposition to a covariance matrix Σ_y that in fact is governed by our factor-analytic model.

Technically, this means that the PCA model is misspecified.

(Can you explain why?)

Nevertheless, this exercise will let us study how well the first two principal components from a PCA can approximate the conditional expectations $Ef_i|Y$ for our two factors $f_i, i = 1, 2$ for the factor analytic model that we have assumed truly governs the data on Y we have generated.

So we compute the PCA decomposition

$$\Sigma_y = P\tilde{\Lambda}P^\top$$

where $\tilde{\Lambda}$ is a diagonal matrix.

We have

$$Y = P\epsilon$$

and

$$\epsilon = P^\top Y$$

Note that we will arrange the eigenvectors in P in the *descending* order of eigenvalues.

```
λ_tilde, P = np.linalg.eigh(Σy)

# arrange the eigenvectors by eigenvalues
ind = sorted(range(N), key=lambda x: λ_tilde[x], reverse=True)

P = P[:, ind]
λ_tilde = λ_tilde[ind]
Λ_tilde = np.diag(λ_tilde)

print('λ_tilde =', λ_tilde)
```

```
λ_tilde = [5.25 5.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25]
```

```
# verify the orthogonality of eigenvectors
np.abs(P @ P.T - np.eye(N)).max()
```

```
np.float64(4.440892098500626e-16)
```

```
# verify the eigenvalue decomposition is correct
P @ Λ_tilde @ P.T
```

```
array([[1.25, 1.  , 1.  , 1.  , 1.  , 0.  , 0.  , 0.  , 0.  , 0.  ],
       [1.  , 1.25, 1.  , 1.  , 1.  , 0.  , 0.  , 0.  , 0.  , 0.  ],
       [1.  , 1.  , 1.25, 1.  , 1.  , 0.  , 0.  , 0.  , 0.  , 0.  ],
       [1.  , 1.  , 1.  , 1.25, 1.  , 0.  , 0.  , 0.  , 0.  , 0.  ],
       [1.  , 1.  , 1.  , 1.  , 1.25, 0.  , 0.  , 0.  , 0.  , 0.  ],
```

(continues on next page)

(continued from previous page)

```
       [0.  , 0.  , 0.  , 0.  , 0.  , 1.25, 1.  , 1.  , 1.  , 1.  ],
       [0.  , 0.  , 0.  , 0.  , 0.  , 1.  , 1.25, 1.  , 1.  , 1.  ],
       [0.  , 0.  , 0.  , 0.  , 0.  , 1.  , 1.  , 1.25, 1.  , 1.  ],
       [0.  , 0.  , 0.  , 0.  , 0.  , 1.  , 1.  , 1.  , 1.25, 1.  ],
       [0.  , 0.  , 0.  , 0.  , 0.  , 1.  , 1.  , 1.  , 1.  , 1.25]])
```

```
ε = P.T @ y

print("ε = ", ε)
```

```
ε =  [ 0.33707041  4.33453458  0.26386722 -0.39706296 -0.30020843  0.45473871
  0.19640695  0.07004059  0.35348891  0.81330557]
```

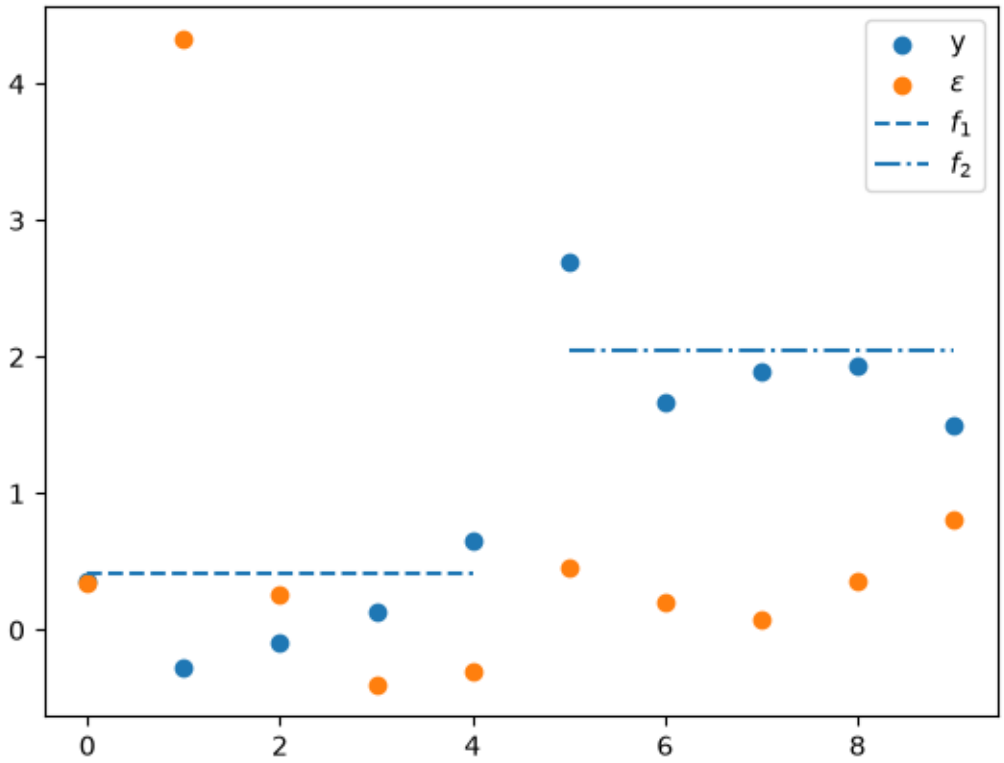
```
# print the values of the two factors

print('f = ', f)
```

```
f =  [0.41858317 2.04367341]
```

Below we'll plot several things

- the N values of y
- the N values of the principal components ϵ
- the value of the first factor f_1 plotted only for the first $N/2$ observations of y for which it receives a non-zero loading in Λ
- the value of the second factor f_2 plotted only for the final $N/2$ observations for which it receives a non-zero loading in Λ

```
plt.scatter(range(N), y, label='y')
plt.scatter(range(N), ε, label=r'$\epsilon$')
plt.hlines(f[0], 0, N//2-1, ls='--', label='$f_{1}$')
plt.hlines(f[1], N//2, N-1, ls='-.', label='$f_{2}$')
plt.legend()

plt.show()
```



Consequently, the first two ϵ_j correspond to the largest two eigenvalues.

Let's look at them, after which we'll look at $Ef|y = By$

```
ε[:2]
```

```
array([0.33707041, 4.33453458])
```

```
# compare with Ef|y
B @ y
```

```
array([0.14356426, 1.84615504])
```

Note

The two largest eigenvalues are both 5.25 in this example.

When an eigenvalue is repeated, the associated principal components are not individually pinned down: any orthonormal basis for the same two-dimensional eigenspace is valid.

For that reason, it is not meaningful to compare ϵ_1 and ϵ_2 component-by-component with $E[f \mid Y]$.

The PC scores live in a PCA coordinate system, while $E[f \mid Y]$ lives in factor space.

Even within the common two-dimensional subspace, the PCA basis can be rotated or sign-flipped, and its coordinates need not use the same scaling as the factor coordinates.

What is uniquely determined is the two-dimensional subspace spanned by the first two columns of P.

In this symmetric example, that subspace is exactly the column space of Λ.

The fraction of variance in y_t explained by the first two principal components is

```
λ_tilde[:2].sum() / λ_tilde.sum()
```

```
np.float64(0.84)
```

To compare PCA with the factor model in observation space, compute

$$\hat{Y} = P_j \epsilon_j + P_k \epsilon_k$$

where P_j and P_k are the eigenvectors associated with the two largest eigenvalues.

```
y_hat = P[:, :2] @ ε[:2]
```

$\hat{Y}$ is the rank-2 PCA approximation to Y in observation space, so it is a 10-vector rather than a 2-vector.

The natural observation-space counterpart from the factor model is $\Lambda E[f \mid Y]$, which is also a 10-vector.

In this symmetric example, both vectors lie in the same two-dimensional subspace, namely the column space of Λ.

They are therefore close, but not identical.

The PCA reconstruction uses the block means directly, while $\Lambda E[f \mid Y]$ shrinks those block means toward zero by the factor $5/(5 + \sigma_u^2) \approx 0.952$.

The next plot makes this comparison concrete.

The two scatter plots, $E[Y \mid f] = \Lambda f$ and $\hat{Y}$, are both 10-vectors in observation space, so they can be compared directly.

The horizontal lines show the factor values f_1 and f_2, together with their posterior means $E[f_i \mid Y]$.

These are 2-dimensional factor-space quantities, drawn over the relevant half of the index set to match the block structure of Λ.

This uses the same idea as the earlier formula $E[Y \mid f] = \Lambda f$: the matrix Λ maps a 2-vector in factor space into a 10-vector in observation space.

In our example,

$$\Lambda a = \begin{bmatrix} a_1 \\ \vdots \\ a_1 \\ a_2 \\ \vdots \\ a_2 \end{bmatrix} \quad \text{for any } a = \begin{bmatrix} a_1 \\ a_2 \end{bmatrix},$$

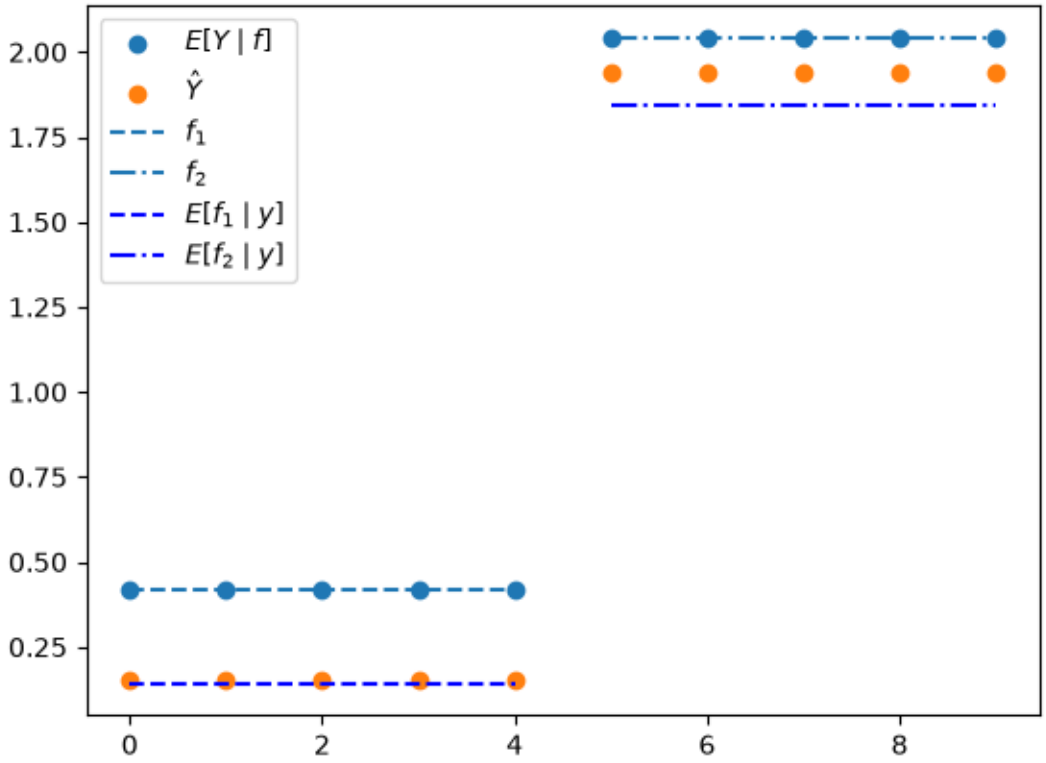
because the first five rows of Λ are $(1, 0)$ and the last five rows are $(0, 1)$.

Therefore, once we observe $Y = y$, the posterior mean $E[f \mid Y = y] = \begin{bmatrix} E[f_1 \mid y] \\ E[f_2 \mid y] \end{bmatrix}$ is converted into the observation-space vector

$$\Lambda E[f \mid Y = y] = \begin{bmatrix} E[f_1 \mid y] \\ \vdots \\ E[f_1 \mid y] \\ E[f_2 \mid y] \\ \vdots \\ E[f_2 \mid y] \end{bmatrix}.$$

So the horizontal line at height $E[f_1 \mid y]$ over the first five indices, together with the horizontal line at height $E[f_2 \mid y]$ over the last five indices, is exactly a picture of $\Lambda E[f \mid Y = y]$.

```
plt.scatter(range(N), Λ @ f, label=r'$E[Y \mid f]$')
plt.scatter(range(N), y_hat, label=r'$\hat{Y}$')
plt.hlines(f[0], 0, N//2-1, ls='--', label='$f_{1}$')
plt.hlines(f[1], N//2, N-1, ls='-.', label='$f_{2}$')

Efy = B @ y
plt.hlines(Efy[0], 0, N//2-1, ls='--', color='b', label=r'$E[f_1 \mid y]$')
plt.hlines(Efy[1], N//2, N-1, ls='-.', color='b', label=r'$E[f_2 \mid y]$')
plt.legend()

plt.show()
```



To compute the covariance matrix of $\hat{Y}$, first form the covariance matrix of ϵ and then extract the upper-left block corresponding to ϵ_1 and ϵ_2.

```
Σεjk = (P.T @ Σy @ P)[:2, :2]

Pjk = P[:, :2]

Σy_hat = Pjk @ Σεjk @ Pjk.T
print('Σy_hat = \n', Σy_hat)
```

```
Σy_hat =
 [[1.05 1.05 1.05 1.05 1.05 0.   0.   0.   0.   0.  ]
 [1.05 1.05 1.05 1.05 1.05 0.   0.   0.   0.   0.  ]
 [1.05 1.05 1.05 1.05 1.05 0.   0.   0.   0.   0.  ]
 [1.05 1.05 1.05 1.05 1.05 0.   0.   0.   0.   0.  ]
 [1.05 1.05 1.05 1.05 1.05 0.   0.   0.   0.   0.  ]
 [0.   0.   0.   0.   0.   1.05 1.05 1.05 1.05 1.05]
```

(continues on next page)

(continued from previous page)

```
[0.   0.   0.   0.   0.   1.05 1.05 1.05 1.05 1.05]
[0.   0.   0.   0.   0.   1.05 1.05 1.05 1.05 1.05]
[0.   0.   0.   0.   0.   1.05 1.05 1.05 1.05 1.05]
[0.   0.   0.   0.   0.   1.05 1.05 1.05 1.05 1.05]]
```

13.15 Exercises

Exercise 13.15.1

Verify conditional mean and variance by simulation

For the bivariate normal with

$$\mu = \begin{bmatrix} 0.5 \\ 1.0 \end{bmatrix}, \quad \Sigma = \begin{bmatrix} 1 & 0.5 \\ 0.5 & 1 \end{bmatrix}$$

fix $z_2 = 2$.

1. Use `MultivariateNormal` to compute the analytical conditional mean $\hat{\mu}_1$ and variance $\hat{\Sigma}_{11}$ of $z_1 \mid z_2 = 2$.
2. Draw 10^6 samples from the joint distribution.

   Retain only those for which $|z_2 - 2| < 0.05$.

   Compute the sample mean and variance of the retained z_1 values.
3. Confirm that the sample estimates are close to the analytical values.

Solution

Here is one solution:

```
μ = np.array([.5, 1.])
Σ = np.array([[1., .5], [.5, 1.]])

mn = MultivariateNormal(μ, Σ)
mn.partition(1)
μ1_hat, Σ11_hat = mn.cond_dist(0, np.array([2.]))
print(f"Analytical  μ1_hat = {μ1_hat[0]:.4f},  Σ11_hat = {Σ11_hat[0,0]:.4f}")

n = 1_000_000
data = rng.multivariate_normal(μ, Σ, size=n)
z1_all, z2_all = data[:, 0], data[:, 1]

mask = np.abs(z2_all - 2.) < 0.05
z1_cond = z1_all[mask]
print(f"Sample size in band: {mask.sum()}")
print(f"Sample      μ1_hat = {np.mean(z1_cond):.4f},  Σ11_hat = {np.var(z1_cond,␣
↪ddof=1):.4f}")
```

```
Analytical  μ1_hat = 1.0000,  Σ11_hat = 0.7500
Sample size in band: 24314
Sample      μ1_hat = 1.0050,  Σ11_hat = 0.7458
```

Exercise 13.15.2

Product of regression slopes equals squared correlation

For a bivariate normal with standard deviations $\sigma_1 = \sigma_2 = 1$ and correlation ρ, show analytically that $b_1 b_2 = \rho^2$, where b_1 is the slope of z_1 on z_2 and b_2 is the slope of z_2 on z_1.

Then verify numerically for $\rho \in \{0.2, 0.5, 0.9\}$ that `βs[0] * βs[1]` $= \rho^2$ by constructing the appropriate `MultivariateNormal` instances.

Solution

The regression slopes are

$$b_1 = \frac{\Sigma_{12}}{\Sigma_{22}} = \frac{\rho \sigma_1 \sigma_2}{\sigma_2^2} = \rho \frac{\sigma_1}{\sigma_2}, \qquad b_2 = \frac{\Sigma_{21}}{\Sigma_{11}} = \rho \frac{\sigma_2}{\sigma_1}$$

so $b_1 b_2 = \rho^2$.

```
for ρ in [0.2, 0.5, 0.9]:
    Σ = np.array([[1., ρ], [ρ, 1.]])
    mn = MultivariateNormal(np.zeros(2), Σ)
    mn.partition(1)
    product = mn.βs[0].item() * mn.βs[1].item()
    print(f"ρ={ρ:.1f}:  b1*b2 = {product:.4f}")
    print(f"ρ^2 = {ρ**2:.4f},  match: {np.isclose(product, ρ**2)}")
```

```
ρ=0.2:  b1*b2 = 0.0400
ρ^2 = 0.0400,  match: True
ρ=0.5:  b1*b2 = 0.2500
ρ^2 = 0.2500,  match: True
ρ=0.9:  b1*b2 = 0.8100
ρ^2 = 0.8100,  match: True
```

Exercise 13.15.3

IQ inference: effect of the signal-to-noise ratio

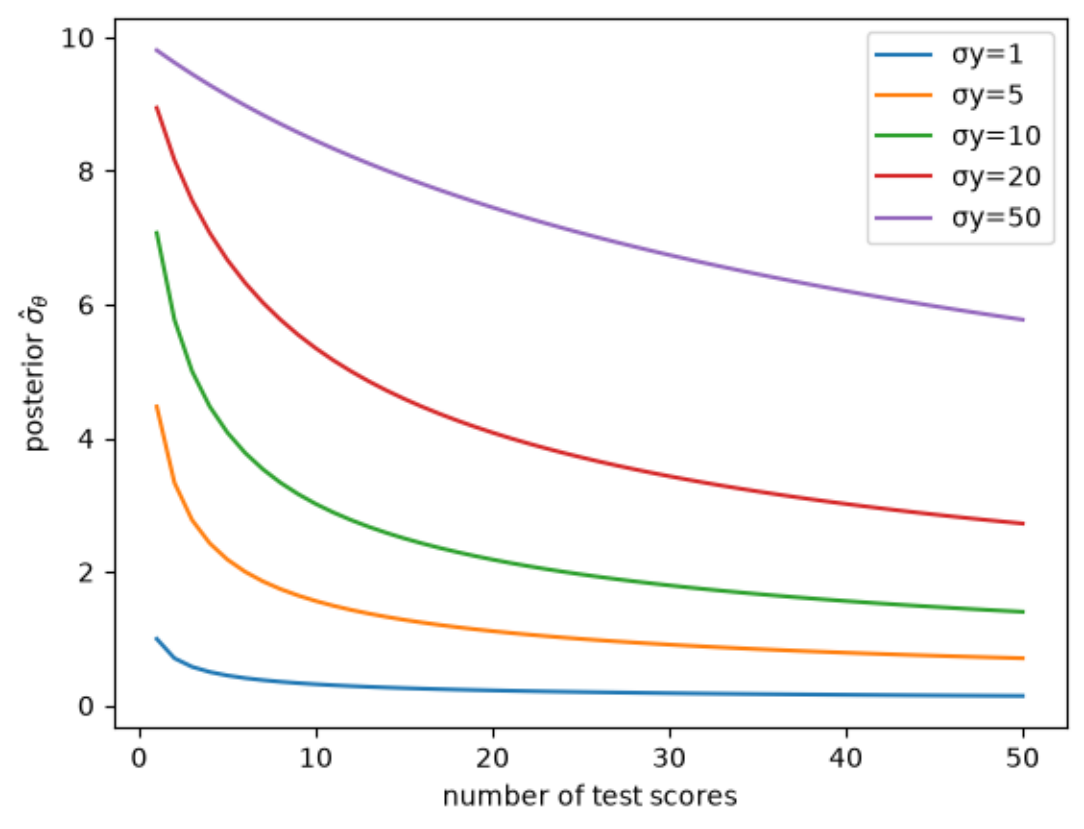
Using the one-dimensional IQ model with $n = 50$ test scores and $\mu_\theta = 100$, $\sigma_\theta = 10$:

1. Vary the test-score noise $\sigma_y \in \{1, 5, 10, 20, 50\}$.
   - For each value, plot the posterior standard deviation $\hat{\sigma}_\theta$ as a function of the number of test scores included (from 1 to 50), with all curves on the same axes.
2. Explain intuitively why a larger σ_y leads to a slower decline of posterior uncertainty.

Solution

Here is one solution:

```
n_max = 50
μθ_val, σθ_val = 100., 10.

fig, ax = plt.subplots()
for σy_val in [1., 5., 10., 20., 50.]:
    σθ_hat_arr = np.empty(n_max)
    for i in range(1, n_max + 1):
        μ_i, Σ_i, _ = construct_moments_IQ(i, μθ_val, σθ_val, σy_val)
        mn_i = MultivariateNormal(μ_i, Σ_i)
        mn_i.partition(i)
        _, Σθ_i = mn_i.cond_dist(1, np.zeros(i))
        σθ_hat_arr[i - 1] = np.sqrt(Σθ_i[0, 0])
    ax.plot(range(1, n_max + 1), σθ_hat_arr, label=f'σy={σy_val:.0f}')

ax.set_xlabel('number of test scores')
ax.set_ylabel(r'posterior $\hat{\sigma}_\theta$')
ax.legend()
plt.show()
```



When σ_y is large each test score is a noisy signal about θ, so many more observations are required before the posterior variance falls appreciably.

In the limit $\sigma_y \to 0$ a single observation pins down θ exactly.

Exercise 13.15.4

Prior vs. likelihood in IQ inference

Using the one-dimensional IQ model with $n = 20$ test scores and $\mu_\theta = 100$, $\sigma_y = 10$:

1. Fix $\sigma_y = 10$ and vary the prior spread $\sigma_\theta \in \{1, 5, 10, 50, 500\}$.
   - For each value compute the posterior mean $\hat{\mu}_\theta$ given the same set of $n = 20$ test scores and plot $\hat{\mu}_\theta$ against σ_θ.
2. Show analytically (or verify numerically) that
   - as $\sigma_\theta \to \infty$ the posterior mean converges to the sample mean $\bar{y}$ (the data dominate the prior), and
   - as $\sigma_\theta \to 0$ the posterior mean converges to the prior mean μ_θ (the prior dominates the data).

Hint

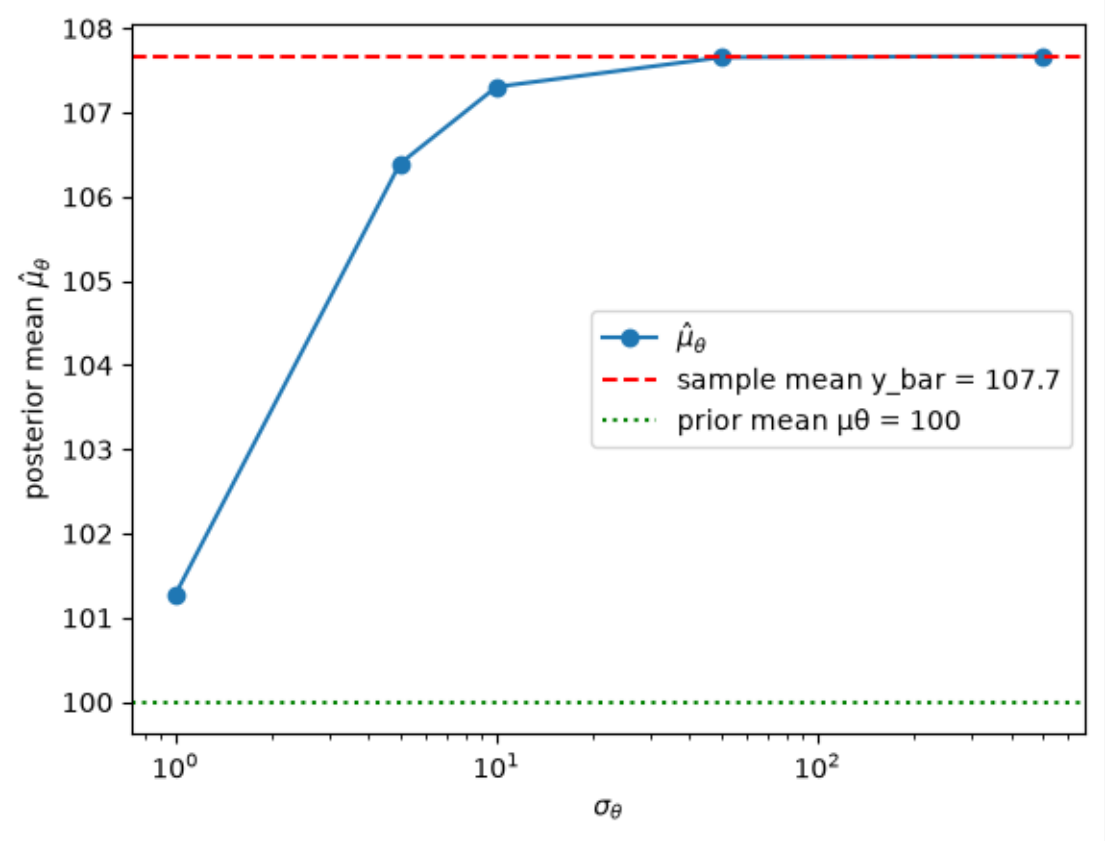
The posterior mean formula is $\hat{\mu}_\theta = (\mu_\theta/\sigma_\theta^2 + n\bar{y}/\sigma_y^2)\big/(1/\sigma_\theta^2 + n/\sigma_y^2)$.

Examine each limit by letting σ_θ go to ∞ or 0.

Solution

Here is one solution:

```
n_scores = 20
μθ_val, σy_val = 100., 10.

rng = np.random.default_rng(42)
true_θ = 108.
y_obs = true_θ + σy_val * rng.standard_normal(n_scores)
y_bar = np.mean(y_obs)

σθ_vals = [1., 5., 10., 50., 500.]
μθ_hat_vals = []

for σθ_val in σθ_vals:
    μ_i, Σ_i, _ = construct_moments_IQ(n_scores, μθ_val, σθ_val, σy_val)
    mn_i = MultivariateNormal(μ_i, Σ_i)
    mn_i.partition(n_scores)
    μθ_hat, _ = mn_i.cond_dist(1, y_obs)
    μθ_hat_vals.append(μθ_hat.item())

def posterior_mean(σθ_val):
    μ_i, Σ_i, _ = construct_moments_IQ(n_scores, μθ_val, σθ_val, σy_val)
    mn_i = MultivariateNormal(μ_i, Σ_i)
    mn_i.partition(n_scores)
    μθ_hat, _ = mn_i.cond_dist(1, y_obs)
    return μθ_hat.item()

fig, ax = plt.subplots()
ax.semilogx(σθ_vals, μθ_hat_vals, 'o-',
            label=r'$\hat{\mu}_\theta$')
ax.axhline(y_bar,  ls='--', color='r',
            label=f'sample mean y_bar = {y_bar:.1f}')
ax.axhline(μθ_val, ls=':',  color='g',
            label=f'prior mean μθ = {μθ_val:.0f}')
ax.set_xlabel(r'$\sigma_\theta$')
ax.set_ylabel(r'posterior mean $\hat{\mu}_\theta$')
ax.legend()
plt.show()

σθ_small = 1e-2
σθ_large = 1e4

print(f"y_bar = {y_bar:.4f}")
print(f"Posterior mean with σθ={σθ_large:.0e}: {posterior_mean(σθ_large):.4f}")
print(f"Posterior mean with σθ={σθ_small:.0e}: {posterior_mean(σθ_small):.4f}")
print(f"Prior mean μθ = {μθ_val:.4f}")
```




```
y_bar = 107.6707
Posterior mean with σθ=1e+04: 107.6707
Posterior mean with σθ=1e-02: 100.0002
Prior mean μθ = 100.0000
```

Exercise 13.15.5

Kalman filter convergence

Using the `iterate` function from the Filtering Foundations section with

$$A = \begin{bmatrix} 0.9 & 0 \\ 0 & 0.5 \end{bmatrix}, \quad C = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \quad G = \begin{bmatrix} 1 & 0 \end{bmatrix}, \quad R = [1]$$

and initial conditions $\hat{x}_0 = [0, 0]'$, $\Sigma_0 = I_2$:

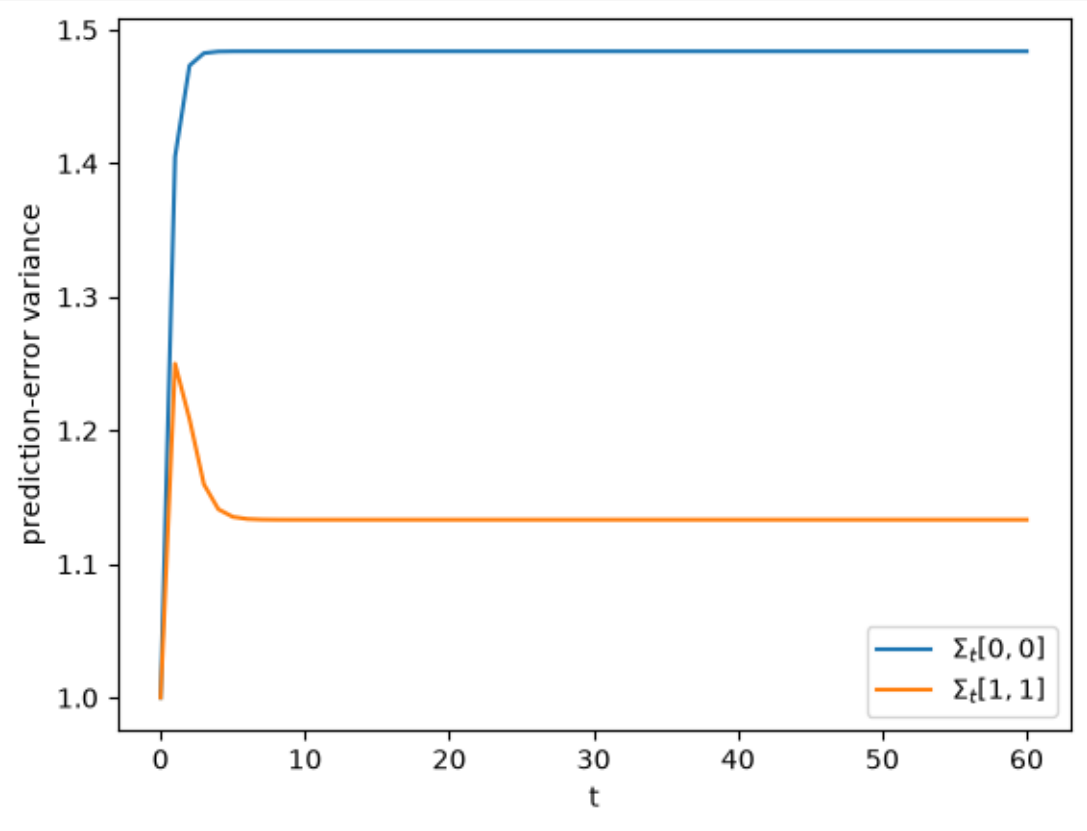
1. Simulate $T = 60$ periods of $\{x_t, y_t\}$ and run the filter.
2. Plot the sequences of conditional variances $\Sigma_t[0, 0]$ and $\Sigma_t[1, 1]$ over time.

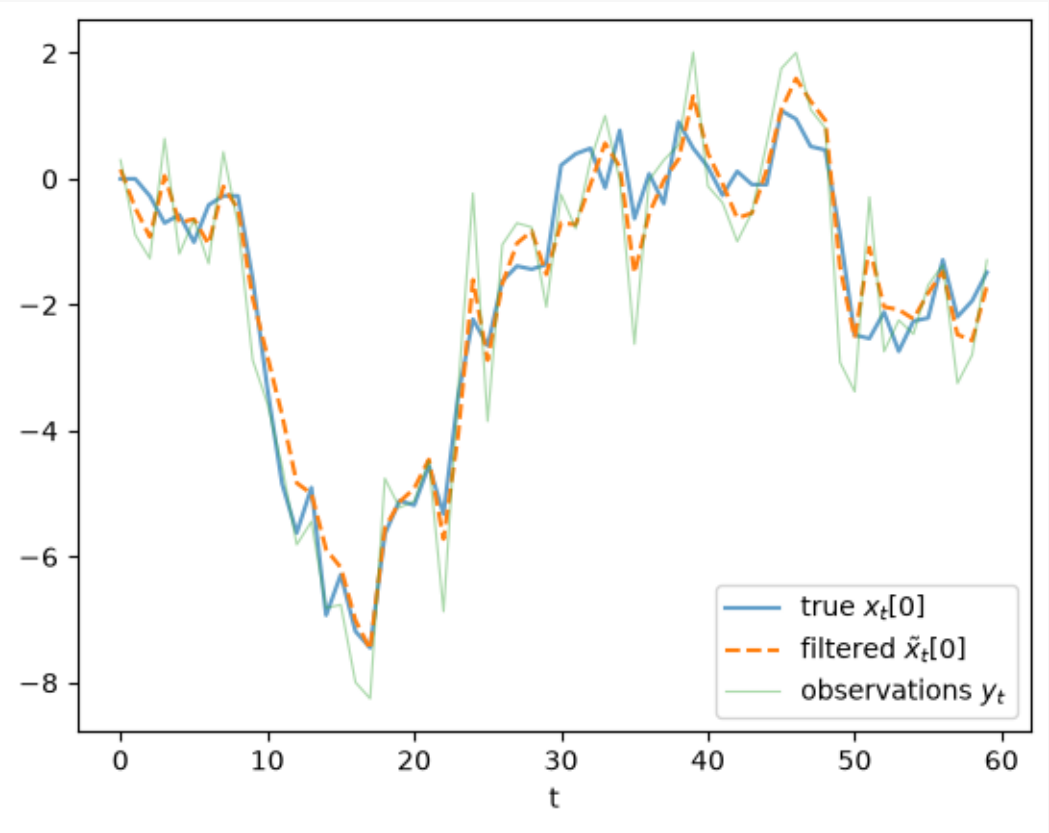
   Verify that they converge to a steady state.
3. Plot the filtered state estimates $\tilde{x}_t[0]$ together with the true $x_t[0]$ and the raw observations y_t on a single figure.

Solution

Here is one solution:

```
A_ex = np.array([[0.9, 0.], [0., 0.5]])
C_ex = np.array([[1.], [1.]])
G_ex = np.array([[1., 0.]])
R_ex = np.array([[1.]])

T_ex = 60
x0_hat_ex = np.zeros(2)
Σ0_ex = np.eye(2)

rng = np.random.default_rng(7)
x_true = np.zeros((T_ex + 1, 2))
y_seq_ex = np.zeros(T_ex)
for t in range(T_ex):
    x_true[t + 1] = A_ex @ x_true[t] + C_ex[:, 0] * rng.standard_normal()
    y_seq_ex[t] = (G_ex @ x_true[t]).item() + rng.standard_normal()

x_hat_seq, Σ_hat_seq = iterate(
    x0_hat_ex, Σ0_ex, A_ex, C_ex, G_ex, R_ex, y_seq_ex)

# x_hat_seq[t] = E[x_t | y^{t-1}] (one-step-ahead prediction)
# Σ_hat_seq[t] = corresponding prediction-error covariance
fig, ax = plt.subplots()
ax.plot(Σ_hat_seq[:, 0, 0], label=r'$\Sigma_t[0,0]$')
ax.plot(Σ_hat_seq[:, 1, 1], label=r'$\Sigma_t[1,1]$')
ax.set_xlabel('t')
ax.set_ylabel('prediction-error variance')
ax.legend()
plt.show()

# The `iterate` function stores one-step-ahead predictions.
# We recover the filtered estimates E[x_t | y^t] by re-applying
# the measurement-update step at each t.
n_state = 2
x_filt_seq = np.empty((T_ex, n_state))
for t in range(T_ex):
    xt_hat = x_hat_seq[t]
    Σt     = Σ_hat_seq[t]
    μ_k = np.hstack([xt_hat, G_ex @ xt_hat])
    Σ_k = np.block([[Σt,          Σt  @ G_ex.T          ],
                    [G_ex @ Σt,   G_ex @ Σt @ G_ex.T + R_ex]])
    mn_k = MultivariateNormal(μ_k, Σ_k)
    mn_k.partition(n_state)
    x_filt_seq[t], _ = mn_k.cond_dist(0, y_seq_ex[t:t+1])

fig, ax = plt.subplots()
ax.plot(x_true[:-1, 0], label='true $x_t[0]$', alpha=0.7)
ax.plot(x_filt_seq[:, 0], label=r'filtered $\tilde{x}_t[0]$', ls='--')
ax.plot(y_seq_ex, label='observations $y_t$', alpha=0.4, lw=0.8)
ax.set_xlabel('t')
ax.legend()
plt.show()
```


1.5
1.4
1.3
1.2
1.1
1.0
prediction-error variance
0
10
20
30
40
50
60
t
$\Sigma_t[0,0]$
$\Sigma_t[1,1]$


2
0
−2
−4
−6
−8
0
10
20
30
40
50
60
t
true $x_t[0]$
filtered $\tilde{x}_t[0]$
observations y_t

Exercise 13.15.6

PCA vs. factor analysis

In the classic factor analysis model at the end of the lecture the true covariance is $\Sigma_y = \Lambda\Lambda' + D$.

1. Set $\sigma_u = 2$ (instead of 0.5).
   - Recompute the fraction of variance explained by the first two principal components and compare it with the $\sigma_u = 0.5$ result.
   - Explain the change.
2. Show that the observation-space factor-analytic posterior $\Lambda E[f \mid Y] = \Lambda BY$ (an N-vector) is **not** equal to the two-component PCA reconstruction $\hat{Y} = P_{:,1:2}\,\epsilon_{1:2}$ (also an N-vector).
   - Plot both on the same axes.

   Note: $E[f \mid Y] = BY$ is a k-vector and $\hat{Y}$ is an N-vector, so they cannot be compared directly; the comparison must be made in observation space via $\Lambda E[f \mid Y]$.
3. In one or two sentences, explain why PCA is misspecified for factor-analytic data.

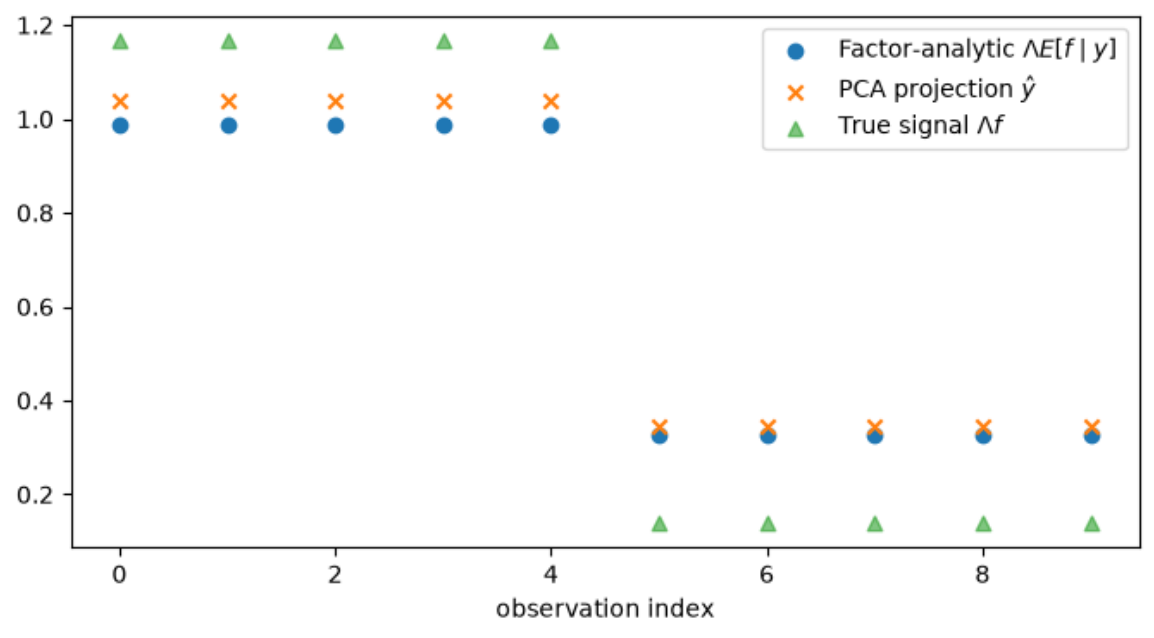
Solution

Here is one solution:

```
rng = np.random.default_rng(42)

N_fa = 10
k_fa = 2

Λ_fa = np.zeros((N_fa, k_fa))
Λ_fa[:N_fa//2, 0] = 1
Λ_fa[N_fa//2:, 1] = 1

for σu_val in [0.5, 2.0]:
    D_fa = np.eye(N_fa) * σu_val ** 2
    Σy_fa = Λ_fa @ Λ_fa.T + D_fa

    λ_fa, P_fa = np.linalg.eigh(Σy_fa)
    ind_fa = sorted(range(N_fa), key=lambda x: λ_fa[x], reverse=True)
    P_fa   = P_fa[:, ind_fa]
    λ_fa   = λ_fa[ind_fa]

    frac = λ_fa[:2].sum() / λ_fa.sum()
    print(f"σu={σu_val}: fraction explained by first 2 PCs = {frac:.4f}")

σu_b = 0.5
D_b = np.eye(N_fa) * σu_b ** 2
Σy_b = Λ_fa @ Λ_fa.T + D_b

μz_b = np.zeros(k_fa + N_fa)
Σz_b = np.block([[np.eye(k_fa), Λ_fa.T], [Λ_fa, Σy_b]])
z_b = rng.multivariate_normal(μz_b, Σz_b)
f_b = z_b[:k_fa]
y_b = z_b[k_fa:]

B_b = Λ_fa.T @ np.linalg.inv(Σy_b)
```

```
Efy_b  = B_b @ y_b

λ_b, P_b = np.linalg.eigh(Σy_b)
ind_b = sorted(range(N_fa), key=lambda x: λ_b[x], reverse=True)
P_b = P_b[:, ind_b]
ε_b = P_b.T @ y_b
y_hat_b = P_b[:, :2] @ ε_b[:2]

fig, ax = plt.subplots(figsize=(8, 4))
ax.scatter(range(N_fa),
        Λ_fa @ Efy_b, label=r'Factor-analytic $\Lambda E[f\mid y]$')
ax.scatter(range(N_fa),
        y_hat_b, marker='x', label=r'PCA projection $\hat{y}$')
ax.scatter(range(N_fa),
        Λ_fa @ f_b, marker='^', alpha=0.6, label=r'True signal $\Lambda f$')
ax.set_xlabel('observation index')
ax.legend()
plt.show()
```

```
σu=0.5: fraction explained by first 2 PCs = 0.8400
σu=2.0: fraction explained by first 2 PCs = 0.3600
```



In this symmetric example, PCA does recover the same two-dimensional observation-space subspace as the factor model, namely the column space of Λ. But PCA is still misspecified for factor-analytic data, because it treats the covariance matrix as an arbitrary matrix to be approximated and does not use the special decomposition $\Sigma_y = \Lambda\Lambda^\top + D$ into a common part and an idiosyncratic noise part.

So the two methods are solving different problems. PCA forms $\hat{Y}$ as the best rank-2 approximation to the observed data vector Y, which in this example amounts to using the block means. The factor model instead computes $\Lambda E[f \mid Y]$, the conditional mean of the latent common component Λf given the data, and because it accounts for noise it shrinks those block means toward zero.

CHAPTER

FOURTEEN

FAULT TREE UNCERTAINTIES

14.1 Overview

This lecture puts elementary tools to work to approximate probability distributions of the annual failure rates of a system consisting of a number of critical parts.

We'll use log normal distributions to approximate probability distributions of critical component parts.

To approximate the probability distribution of the **sum** of n log normal probability distributions that describes the failure rate of the entire system, we'll compute the convolution of those n log normal probability distributions.

We'll use the following concepts and tools:

- log normal distributions
- the convolution theorem that describes the probability distribution of the sum independent random variables
- fault tree analysis for approximating a failure rate of a multi-component system
- a hierarchical probability model for describing uncertain probabilities
- Fourier transforms and inverse Fourier tranforms as efficient ways of computing convolutions of sequences

For more about Fourier transforms see this quantecon lecture Circulant Matrices as well as these lecture Covariance Stationary Processes and Estimation of Spectra.

El-Shanawany, Ardron, and Walker [El-Shanawany *et al.*, 2018] and Greenfield and Sargent [Greenfield and Sargent, 1993] used some of the methods described here to approximate probabilities of failures of safety systems in nuclear facilities.

These methods respond to some of the recommendations made by Apostolakis [Apostolakis, 1990] for constructing procedures for quantifying uncertainty about the reliability of a safety system.

We'll start by bringing in some Python machinery.

```
!pip install tabulate
```

```
Requirement already satisfied: tabulate in /home/runner/miniconda3/envs/quantecon/
↪lib/python3.13/site-packages (0.10.0)
```

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import fftconvolve
from tabulate import tabulate
import time
```

```
np.set_printoptions(precision=3, suppress=True)
```

14.2 Log normal distribution

If a random variable x follows a normal distribution with mean μ and variance σ^2, then the natural logarithm of x, say $y = \log(x)$, follows a **log normal distribution** with parameters μ, σ^2.

Notice that we said **parameters** and not **mean and variance** μ, σ^2.

- μ and σ^2 are the mean and variance of $x = \exp(y)$
- they are **not** the mean and variance of y
- instead, the mean of y is $e^{\mu + \frac{1}{2}\sigma^2}$ and the variance of y is $(e^{\sigma^2} - 1)e^{2\mu + \sigma^2}$

A log normal random variable y is nonnegative.

The density for a log normal random variate y is

$$f(y) = \frac{1}{y\sigma\sqrt{2\pi}} \exp\left(\frac{-(\log y - \mu)^2}{2\sigma^2}\right)$$

for $y \geq 0$.

Important features of a log normal random variable are

mean:	$e^{\mu + \frac{1}{2}\sigma^2}$
variance:	$(e^{\sigma^2} - 1)e^{2\mu + \sigma^2}$
median:	e^{μ}
mode:	$e^{\mu - \sigma^2}$
.95 quantile:	$e^{\mu + 1.645\sigma}$
.95-.05 quantile ratio:	$e^{1.645\sigma}$

Recall the following *stability* property of two independent normally distributed random variables:

If x_1 is normal with mean μ_1 and variance σ_1^2 and x_2 is independent of x_1 and normal with mean μ_2 and variance σ_2^2, then $x_1 + x_2$ is normally distributed with mean $\mu_1 + \mu_2$ and variance $\sigma_1^2 + \sigma_2^2$.

Independent log normal distributions have a different *stability* property.

The **product** of independent log normal random variables is also log normal.

In particular, if y_1 is log normal with parameters (μ_1, σ_1^2) and y_2 is log normal with parameters (μ_2, σ_2^2), then the product $y_1 y_2$ is log normal with parameters $(\mu_1 + \mu_2, \sigma_1^2 + \sigma_2^2)$.

Note

While the product of two log normal distributions is log normal, the **sum** of two log normal distributions is **not** log normal.

This observation sets the stage for challenge that confronts us in this lecture, namely, to approximate probability distributions of **sums** of independent log normal random variables.

To compute the probability distribution of the sum of two log normal distributions, we can use the following convolution property of a probability distribution that is a sum of independent random variables.

14.3 The Convolution Property

Let x be a random variable with probability density $f(x)$, where $x \in \mathbf{R}$.

Let y be a random variable with probability density $g(y)$, where $y \in \mathbf{R}$.

Let x and y be independent random variables and let $z = x + y \in \mathbf{R}$.

Then the probability distribution of z is

$$h(z) = (f * g)(z) \equiv \int_{-\infty}^{\infty} f(z)g(z-\tau)d\tau$$

where $(f * g)$ denotes the **convolution** of the two functions f and g.

If the random variables are both nonnegative, then the above formula specializes to

$$h(z) = (f * g)(z) \equiv \int_{0}^{\infty} f(z)g(z-\tau)d\tau$$

Below, we'll use a discretized version of the preceding formula.

In particular, we'll replace both f and g with discretized counterparts, normalized to sum to 1 so that they are probability distributions.

- by **discretized** we mean an equally spaced sampled version

Then we'll use the following version of the above formula

$$h_n = (f * g)_n = \sum_{m=0}^{\infty} f_m g_{n-m}, n \geq 0$$

to compute a discretized version of the probability distribution of the sum of two random variables, one with probability mass function f, the other with probability mass function g.

Before applying the convolution property to sums of log normal distributions, let's practice on some simple discrete distributions.

To take one example, let's consider the following two probability distributions

$$f_j = \text{Prob}(X = j), j = 0, 1$$

and

$$g_j = \text{Prob}(Y = j), j = 0, 1, 2, 3$$

and

$$h_j = \text{Prob}(Z \equiv X + Y = j), j = 0, 1, 2, 3, 4$$

The convolution property tells us that

$$h = f * g = g * f$$

Let's compute an example using the `numpy.convolve` and `scipy.signal.fftconvolve`.

```
f = [.75, .25]
g = [0., .6,  0., .4]
h = np.convolve(f,g)
hf = fftconvolve(f,g)

print("f = ", f,  ", np.sum(f) = ", np.sum(f))
print("g = ", g, ", np.sum(g) = ", np.sum(g))
print("h = ", h, ", np.sum(h) = ", np.sum(h))
print("hf = ", hf, ",np.sum(hf) = ", np.sum(hf))
```

```
f =  [0.75, 0.25] , np.sum(f) =  1.0
g =  [0.0, 0.6, 0.0, 0.4] , np.sum(g) =  1.0
h =  [0.   0.45 0.15 0.3  0.1 ] , np.sum(h) =  1.0
hf =  [0.   0.45 0.15 0.3  0.1 ] ,np.sum(hf) =  1.0000000000000002
```

A little later we'll explain some advantages that come from using `scipy.signal.ftconvolve` rather than `numpy.convolve`.numpy program convolve.

They provide the same answers but `scipy.signal.ftconvolve` is much faster.

That's why we rely on it later in this lecture.

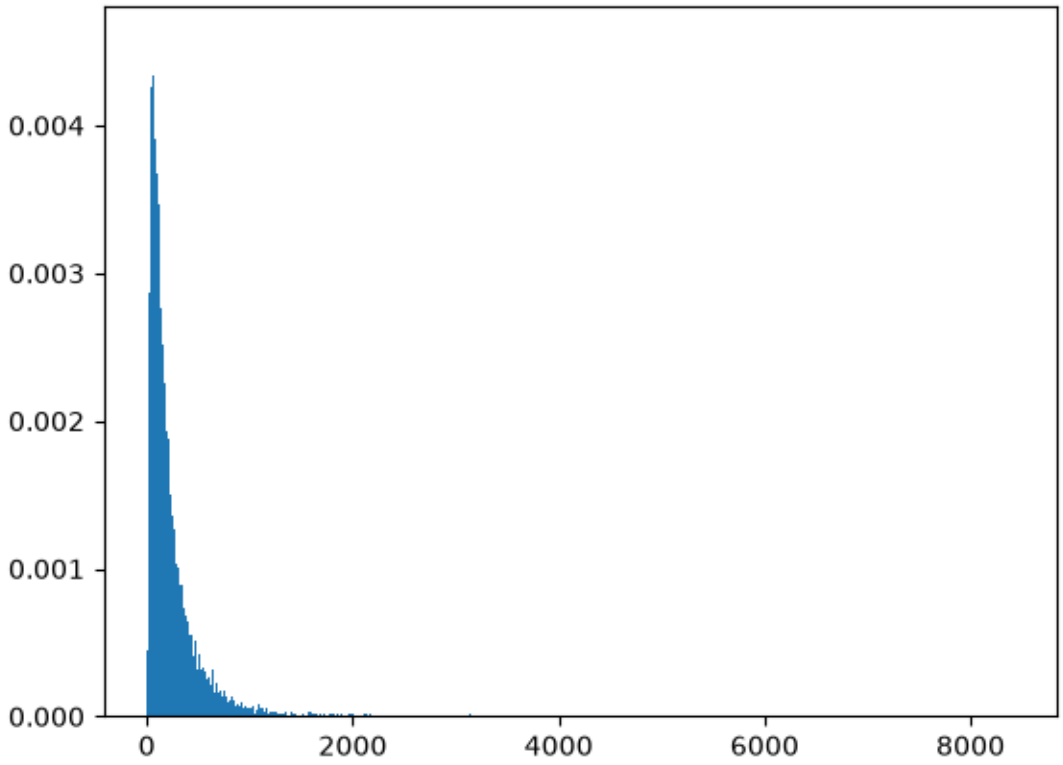
14.4 Approximating Distributions

We'll construct an example to verify that discretized distributions can do a good job of approximating samples drawn from underlying continuous distributions.

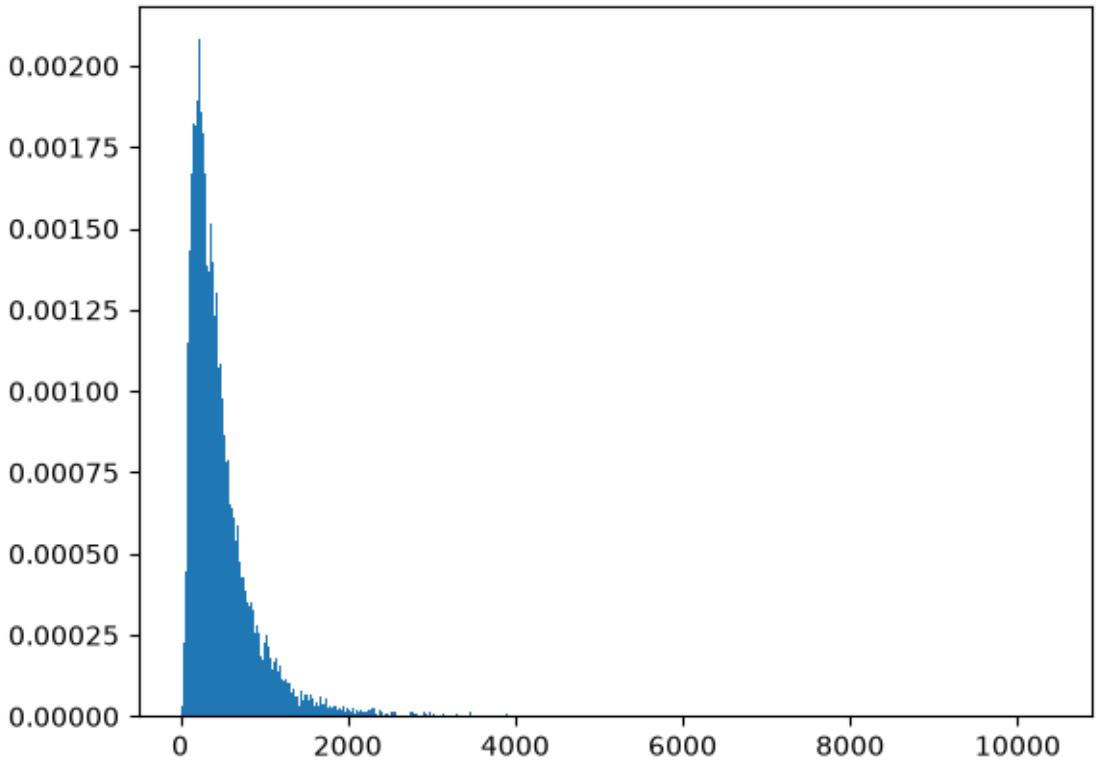
We'll start by generating samples of size 25000 of three independent log normal random variates as well as pairwise and triple-wise sums.

Then we'll plot histograms and compare them with convolutions of appropriate discretized log normal distributions.

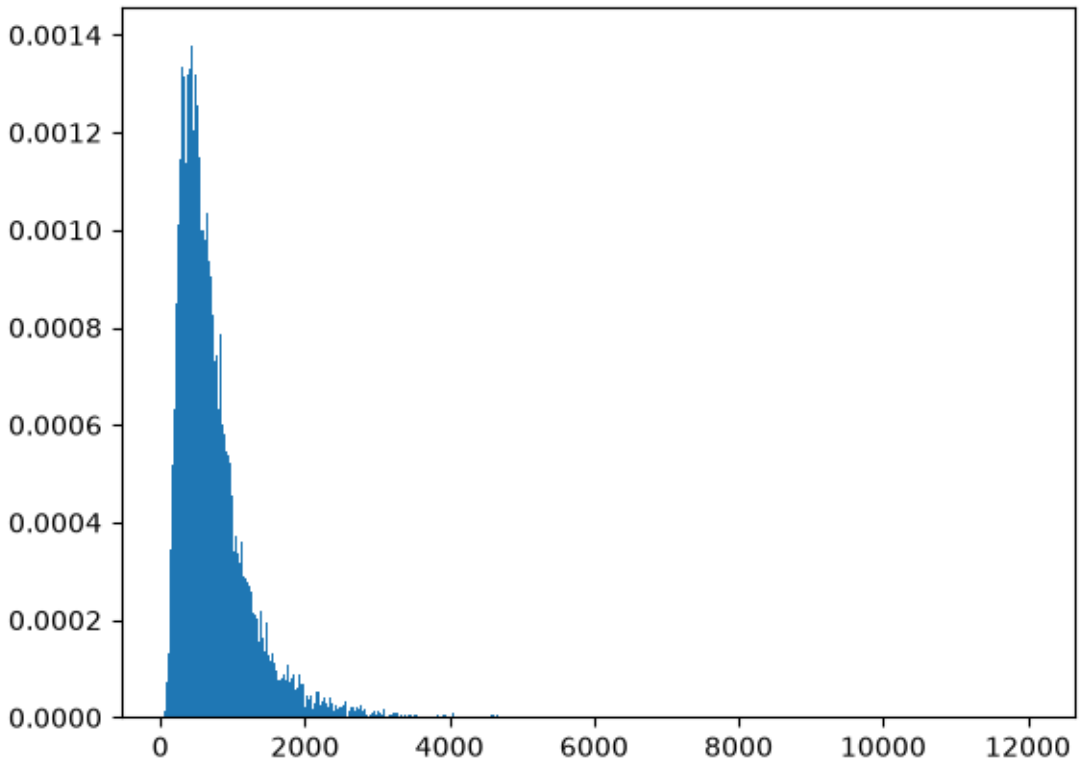
```
## create sums of two and three log normal random variates ssum2 = s1 + s2 and ssum3
 ↪= s1 + s2 + s3


mu1, sigma1 = 5., 1. # mean and standard deviation
s1 = np.random.lognormal(mu1, sigma1, 25000)

mu2, sigma2 = 5., 1. # mean and standard deviation
s2 = np.random.lognormal(mu2, sigma2, 25000)

mu3, sigma3 = 5., 1. # mean and standard deviation
s3 = np.random.lognormal(mu3, sigma3, 25000)

ssum2 = s1 + s2

ssum3 = s1 + s2 + s3

count, bins, ignored = plt.hist(s1, 1000, density=True, align='mid')
```




```
count, bins, ignored = plt.hist(ssum2, 1000, density=True, align='mid')
```



```
count, bins, ignored = plt.hist(ssum3, 1000, density=True, align='mid')
```




```
samp_mean2 = np.mean(s2)
pop_mean2 = np.exp(mu2+ (sigma2**2)/2)

pop_mean2, samp_mean2, mu2, sigma2
```

```
(np.float64(244.69193226422038), np.float64(247.18597394640673), 5.0, 1.0)
```

Here are helper functions that create a discretized version of a log normal probability density function.

```
def p_log_normal(x,μ,σ):
    p = 1 / (σ*x*np.sqrt(2*np.pi)) * np.exp(-1/2*((np.log(x) - μ)/σ)**2)
    return p

def pdf_seq(μ,σ,I,m):
    x = np.arange(1e-7,I,m)
    p_array = p_log_normal(x,μ,σ)
    p_array_norm = p_array/np.sum(p_array)
    return p_array,p_array_norm,x
```

Now we shall set a grid length I and a grid increment size $m = 1$ for our discretizations.

Note

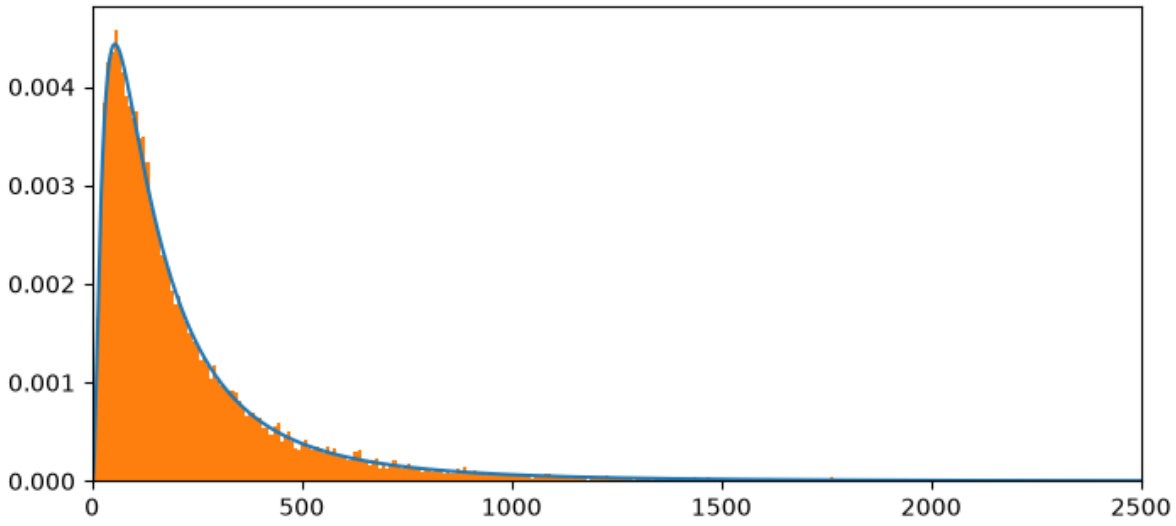
We set I equal to a power of two because we want to be free to use a Fast Fourier Transform to compute a convolution of two sequences (discrete distributions).

We recommend experimenting with different values of the power p of 2.

Setting it to 15 rather than 12, for example, improves how well the discretized probability mass function approximates the original continuous probability density function being studied.

```
p=15
I = 2**p # Truncation value
m = .1 # increment size
```

```
## Cell to check -- note what happens when don't normalize!
## things match up without adjustment. Compare with above

p1,p1_norm,x = pdf_seq(mu1,sigma1,I,m)
## compute number of points to evaluate the probability mass function
NT = x.size

plt.figure(figsize = (8,8))
plt.subplot(2,1,1)
plt.plot(x[:int(NT)],p1[:int(NT)],label = '')
plt.xlim(0,2500)
count, bins, ignored = plt.hist(s1, 1000, density=True, align='mid')

plt.show()
```




```
# Compute mean from discretized pdf and compare with the theoretical value

mean= np.sum(np.multiply(x[:NT],p1_norm[:NT]))
meantheory = np.exp(mu1+.5*sigma1**2)
mean, meantheory
```

```
(np.float64(244.69059898302908), np.float64(244.69193226422038))
```

14.5 Convolving Probability Mass Functions

Now let's use the convolution theorem to compute the probability distribution of a sum of the two log normal random variables we have parameterized above.

We'll also compute the probability of a sum of three log normal distributions constructed above.

Before we do these things, we shall explain our choice of Python algorithm to compute a convolution of two sequences.

Because the sequences that we convolve are long, we use the `scipy.signal.fftconvolve` function rather than the numpy.convove function.

These two functions give virtually equivalent answers but for long sequences `scipy.signal.fftconvolve` is much faster.

The program `scipy.signal.fftconvolve` uses fast Fourier transforms and their inverses to calculate convolutions.

Let's define the Fourier transform and the inverse Fourier transform.

The **Fourier transform** of a sequence $\{x_t\}_{t=0}^{T-1}$ is a sequence of complex numbers $\{x(\omega_j)\}_{j=0}^{T-1}$ given by

$$x(\omega_j) = \sum_{t=0}^{T-1} x_t \exp(-i\omega_j t) \tag{14.1}$$

where $\omega_j = \frac{2\pi j}{T}$ for $j = 0, 1, \ldots, T-1$.

The **inverse Fourier transform** of the sequence $\{x(\omega_j)\}_{j=0}^{T-1}$ is

$$x_t = T^{-1} \sum_{j=0}^{T-1} x(\omega_j) \exp(i\omega_j t) \tag{14.2}$$

The sequences $\{x_t\}_{t=0}^{T-1}$ and $\{x(\omega_j)\}_{j=0}^{T-1}$ contain the same information.

The pair of equations (14.1) and (14.2) tell how to recover one series from its Fourier partner.

The program `scipy.signal.fftconvolve` deploys the theorem that a convolution of two sequences $\{f_k\}, \{g_k\}$ can be computed in the following way:

- Compute Fourier transforms $F(\omega), G(\omega)$ of the $\{f_k\}$ and $\{g_k\}$ sequences, respectively
- Form the product $H(\omega) = F(\omega)G(\omega)$
- The convolution of $f * g$ is the inverse Fourier transform of $H(\omega)$

The **fast Fourier transform** and the associated **inverse fast Fourier transform** execute these calculations very quickly.

This is the algorithm that `scipy.signal.fftconvolve` uses.

Let's do a warmup calculation that compares the times taken by `numpy.convove` and `scipy.signal.fftconvolve`.

```
p1,p1_norm,x = pdf_seq(mu1,sigma1,I,m)
p2,p2_norm,x = pdf_seq(mu2,sigma2,I,m)
p3,p3_norm,x = pdf_seq(mu3,sigma3,I,m)

tic = time.perf_counter()

c1 = np.convolve(p1_norm,p2_norm)
c2 = np.convolve(c1,p3_norm)
```

(continues on next page)

(continued from previous page)

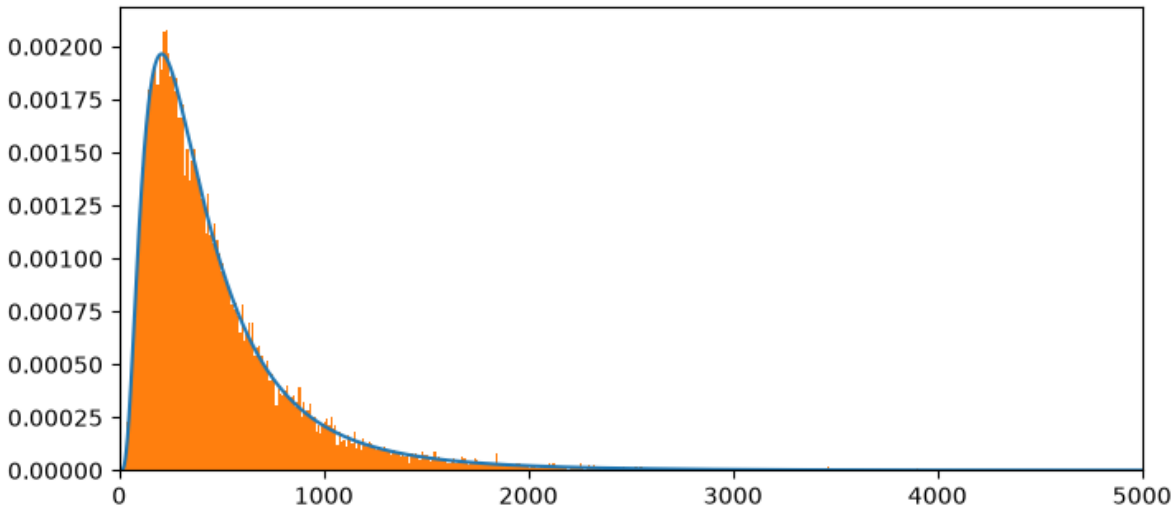
```
toc = time.perf_counter()

tdiff1 = toc - tic

tic = time.perf_counter()

c1f = fftconvolve(p1_norm,p2_norm)
c2f = fftconvolve(c1f,p3_norm)
toc = time.perf_counter()

toc = time.perf_counter()

tdiff2 = toc - tic

print("time with np.convolve = ", tdiff1,  "; time with fftconvolve = ",  tdiff2)
```

```
time with np.convolve =  30.823216856000045 ; time with fftconvolve =  0.
↪14449454999999034
```

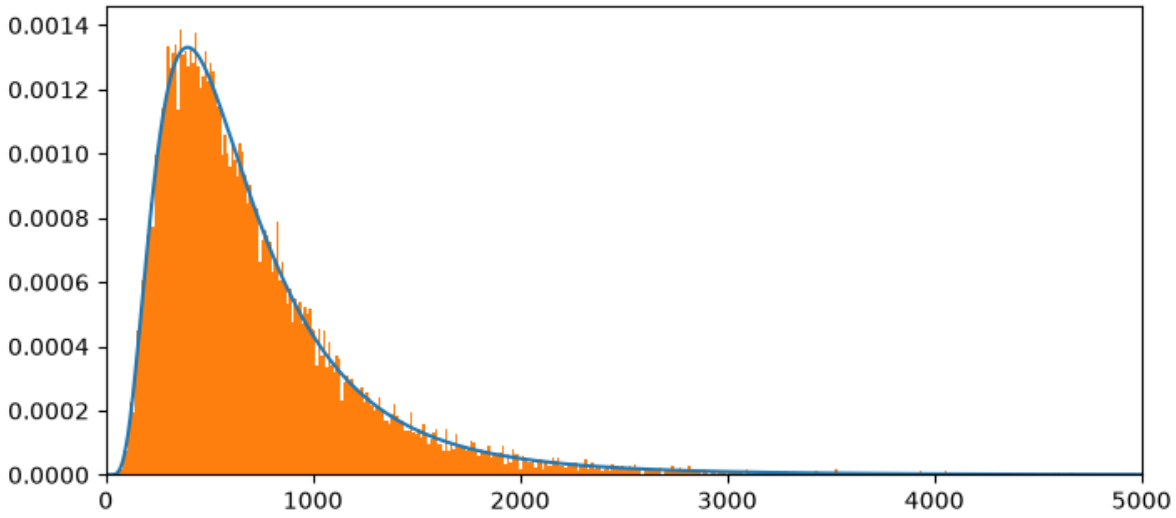
The fast Fourier transform is two orders of magnitude faster than `numpy.convolve`

Now let's plot our computed probability mass function approximation for the sum of two log normal random variables against the histogram of the sample that we formed above.

```
NT= np.size(x)

plt.figure(figsize = (8,8))
plt.subplot(2,1,1)
plt.plot(x[:int(NT)],c1f[:int(NT)]/m,label = '')
plt.xlim(0,5000)

count, bins, ignored = plt.hist(ssum2, 1000, density=True, align='mid')
# plt.plot(P2P3[:10000],label = 'FFT method',linestyle = '--')

plt.show()
```



```
NT= np.size(x)
plt.figure(figsize = (8,8))
plt.subplot(2,1,1)
plt.plot(x[:int(NT)],c2f[:int(NT)]/m,label = '')
plt.xlim(0,5000)

count, bins, ignored = plt.hist(ssum3, 1000, density=True, align='mid')
# plt.plot(P2P3[:10000],label = 'FFT method',linestyle = '--')

plt.show()
```




```
## Let's compute the mean of the discretized pdf
mean= np.sum(np.multiply(x[:NT],c1f[:NT]))
# meantheory = np.exp(mu1+.5*sigma1**2)
mean, 2*meantheory
```

```
(np.float64(489.3810974093834), np.float64(489.38386452844077))
```

```
## Let's compute the mean of the discretized pdf
mean= np.sum(np.multiply(x[:NT],c2f[:NT]))
# meantheory = np.exp(mu1+.5*sigma1**2)
mean, 3*meantheory
```

```
(np.float64(734.0714863312236), np.float64(734.0757967926611))
```

14.6 Failure Tree Analysis

We shall soon apply the convolution theorem to compute the probability of a **top event** in a failure tree analysis.

Before applying the convolution theorem, we first describe the model that connects constituent events to the **top** end whose failure rate we seek to quantify.

The model is an example of the widely used **failure tree analysis** described by El-Shanawany, Ardron, and Walker [El-Shanawany *et al.*, 2018].

To construct the statistical model, we repeatedly use what is called the **rare event approximation**.

We want to compute the probabilty of an event $A \cup B$.

- the union $A \cup B$ is the event that A OR B occurs

A law of probability tells us that A OR B occurs with probability

$$P(A \cup B) = P(A) + P(B) - P(A \cap B)$$

where the intersection $A \cap B$ is the event that A **AND** B both occur and the union $A \cup B$ is the event that A **OR** B occurs.

If A and B are independent, then

$$P(A \cap B) = P(A)P(B)$$

If $P(A)$ and $P(B)$ are both small, then $P(A)P(B)$ is even smaller.

The **rare event approximation** is

$$P(A \cup B) \approx P(A) + P(B)$$

This approximation is widely used in evaluating system failures.

14.7 Application

A system has been designed with the feature a system failure occurs when **any** of n critical components fails.

The failure probability $P(A_i)$ of each event A_i is small.

We assume that failures of the components are statistically independent random variables.

We repeatedly apply a **rare event approximation** to obtain the following formula for the problem of a system failure:

$$P(F) \approx P(A_1) + P(A_2) + \cdots + P(A_n)$$

or

$$P(F) \approx \sum_{i=1}^{n} P(A_i) \tag{14.3}$$

Probabilities for each event are recorded as failure rates per year.

14.8 Failure Rates Unknown

Now we come to the problem that really interests us, following [El-Shanawany *et al.*, 2018] and Greenfield and Sargent [Greenfield and Sargent, 1993] in the spirit of Apostolakis [Apostolakis, 1990].

The constituent probabilities or failure rates $P(A_i)$ are not known a priori and have to be estimated.

We address this problem by specifying **probabilities of probabilities** that capture one notion of not knowing the constituent probabilities that are inputs into a failure tree analysis.

Thus, we assume that a system analyst is uncertain about the failure rates $P(A_i), i = 1, \ldots, n$ for components of a system.

The analyst copes with this situation by regarding the systems failure probability $P(F)$ and each of the component probabilities $P(A_i)$ as random variables.

- dispersions of the probability distribution of $P(A_i)$ characterizes the analyst's uncertainty about the failure probability $P(A_i)$
- the dispersion of the implied probability distribution of $P(F)$ characterizes his uncertainty about the probability of a system's failure.

This leads to what is sometimes called a **hierarchical** model in which the analyst has probabilities about the probabilities $P(A_i)$.

The analyst formalizes his uncertainty by assuming that

- the failure probability $P(A_i)$ is itself a log normal random variable with parameters (μ_i, σ_i).
- failure rates $P(A_i)$ and $P(A_j)$ are statistically independent for all pairs with $i \neq j$.

The analyst calibrates the parameters (μ_i, σ_i) for the failure events $i = 1, \ldots, n$ by reading reliability studies in engineering papers that have studied historical failure rates of components that are as similar as possible to the components being used in the system under study.

The analyst assumes that such information about the observed dispersion of annual failure rates, or times to failure, can inform him of what to expect about parts' performances in his system.

The analyst assumes that the random variables $P(A_i)$ are statistically mutually independent.

The analyst wants to approximate a probability mass function and cumulative distribution function of the systems failure probability $P(F)$.

- We say probability mass function because of how we discretize each random variable, as described earlier.

The analyst calculates the probability mass function for the **top event** F, i.e., a **system failure**, by repeatedly applying the convolution theorem to compute the probability distribution of a sum of independent log normal random variables, as described in equation (14.3).

14.9 Waste Hoist Failure Rate

We'll take close to a real world example by assuming that $n = 14$.

The example estimates the annual failure rate of a critical hoist at a nuclear waste facility.

A regulatory agency wants the sytem to be designed in a way that makes the failure rate of the top event small with high probability.

This example is Design Option B-2 (Case I) described in Table 10 on page 27 of [Greenfield and Sargent, 1993].

The table describes parameters μ_i, σ_i for fourteen log normal random variables that consist of **seven pairs** of random variables that are identically and independently distributed.

- Within a pair, parameters μ_i, σ_i are the same
- As described in table 10 of [Greenfield and Sargent, 1993] p. 27, parameters of log normal distributions for the seven unique probabilities $P(A_i)$ have been calibrated to be the values in the following Python code:

```
mu1, sigma1 = 4.28, 1.1947
mu2, sigma2 = 3.39, 1.1947
mu3, sigma3 = 2.795, 1.1947
mu4, sigma4 = 2.717, 1.1947
mu5, sigma5 = 2.717, 1.1947
mu6, sigma6 = 1.444, 1.4632
mu7, sigma7 = -.040, 1.4632
```

Note

Because the failure rates are all very small, log normal distributions with the above parameter values actually describe $P(A_i)$ times 10^{-09}.

So the probabilities that we'll put on the x axis of the probability mass function and associated cumulative distribution function should be multiplied by 10^{-09}

To extract a table that summarizes computed quantiles, we'll use a helper function

```
def find_nearest(array, value):
    array = np.asarray(array)
    idx = (np.abs(array - value)).argmin()
    return idx
```

We compute the required thirteen convolutions in the following code.

(Please feel free to try different values of the power parameter p that we use to set the number of points in our grid for constructing the probability mass functions that discretize the continuous log normal distributions.)

We'll plot a counterpart to the cumulative distribution function (CDF) in figure 5 on page 29 of [Greenfield and Sargent, 1993] and we'll also present a counterpart to their Table 11 on page 28.

```
p=15
I = 2**p # Truncation value
m =  .05 # increment size



p1,p1_norm,x = pdf_seq(mu1,sigma1,I,m)
p2,p2_norm,x = pdf_seq(mu2,sigma2,I,m)
p3,p3_norm,x = pdf_seq(mu3,sigma3,I,m)
p4,p4_norm,x = pdf_seq(mu4,sigma4,I,m)
p5,p5_norm,x = pdf_seq(mu5,sigma5,I,m)
p6,p6_norm,x = pdf_seq(mu6,sigma6,I,m)
p7,p7_norm,x = pdf_seq(mu7,sigma7,I,m)
p8,p8_norm,x = pdf_seq(mu7,sigma7,I,m)
p9,p9_norm,x = pdf_seq(mu7,sigma7,I,m)
p10,p10_norm,x = pdf_seq(mu7,sigma7,I,m)
p11,p11_norm,x = pdf_seq(mu7,sigma7,I,m)
p12,p12_norm,x = pdf_seq(mu7,sigma7,I,m)
p13,p13_norm,x = pdf_seq(mu7,sigma7,I,m)
p14,p14_norm,x = pdf_seq(mu7,sigma7,I,m)

tic = time.perf_counter()

c1 = fftconvolve(p1_norm,p2_norm)
c2 = fftconvolve(c1,p3_norm)
c3 = fftconvolve(c2,p4_norm)
c4 = fftconvolve(c3,p5_norm)
c5 = fftconvolve(c4,p6_norm)
c6 = fftconvolve(c5,p7_norm)
c7 = fftconvolve(c6,p8_norm)
c8 = fftconvolve(c7,p9_norm)
c9 = fftconvolve(c8,p10_norm)
c10 = fftconvolve(c9,p11_norm)
```

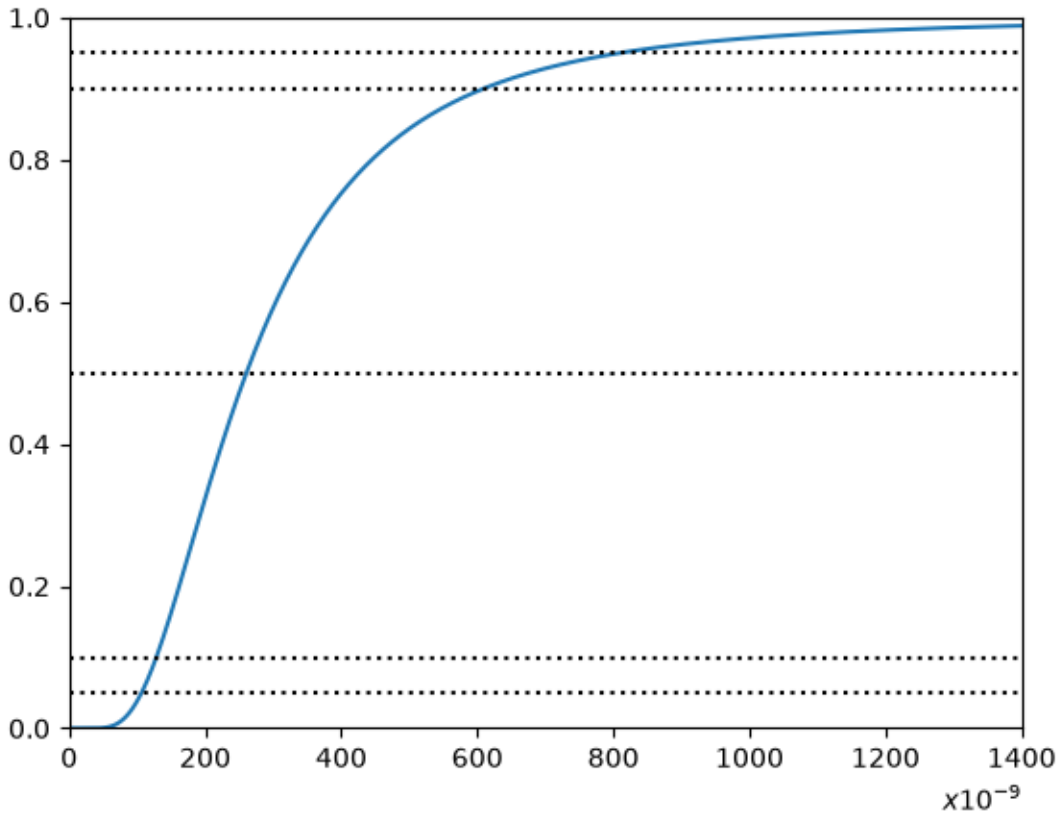
(continues on next page)

(continued from previous page)

```
c11 = fftconvolve(c10,p12_norm)
c12 = fftconvolve(c11,p13_norm)
c13 = fftconvolve(c12,p14_norm)

toc = time.perf_counter()

tdiff13 = toc - tic

print("time for 13 convolutions = ", tdiff13)
```

```
time for 13 convolutions =  7.489028703000031
```

```
d13 = np.cumsum(c13)
Nx=int(1400)
plt.figure()
plt.plot(x[0:int(Nx/m)],d13[0:int(Nx/m)])  # show Yad this -- I multiplied by m --␣
↪step size
plt.hlines(0.5,min(x),Nx,linestyles='dotted',colors = {'black'})
plt.hlines(0.9,min(x),Nx,linestyles='dotted',colors = {'black'})
plt.hlines(0.95,min(x),Nx,linestyles='dotted',colors = {'black'})
plt.hlines(0.1,min(x),Nx,linestyles='dotted',colors = {'black'})
plt.hlines(0.05,min(x),Nx,linestyles='dotted',colors = {'black'})
plt.ylim(0,1)
plt.xlim(0,Nx)
plt.xlabel("$x10^{-9}$",loc = "right")
plt.show()

x_1 = x[find_nearest(d13,0.01)]
x_5 = x[find_nearest(d13,0.05)]
x_10 = x[find_nearest(d13,0.1)]
x_50 = x[find_nearest(d13,0.50)]
x_66 = x[find_nearest(d13,0.665)]
x_85 = x[find_nearest(d13,0.85)]
x_90 = x[find_nearest(d13,0.90)]
x_95 = x[find_nearest(d13,0.95)]
x_99 = x[find_nearest(d13,0.99)]
x_9978 = x[find_nearest(d13,0.9978)]

print(tabulate([
    ['1%',f"{x_1}"],
    ['5%',f"{x_5}"],
    ['10%',f"{x_10}"],
    ['50%',f"{x_50}"],
    ['66.5%',f"{x_66}"],
    ['85%',f"{x_85}"],
    ['90%',f"{x_90}"],
    ['95%',f"{x_95}"],
    ['99%',f"{x_99}"],
    ['99.78%',f"{x_9978}"]],
    headers = ['Percentile', 'x * 1e-9']))
```



Percentile	x * 1e-9
1%	76.15
5%	106.5
10%	128.2
50%	260.55
66.5%	338.55
85%	509.4
90%	608.8
95%	807.6
99%	1470.2
99.78%	2474.85

The above table agrees closely with column 2 of Table 11 on p. 28 of of [Greenfield and Sargent, 1993].

Discrepancies are probably due to slight differences in the number of digits retained in inputting $\mu_i, \sigma_i, i = 1, \ldots, 14$ and in the number of points deployed in the discretizations.

CHAPTER

FIFTEEN

INTRODUCTION TO ARTIFICIAL NEURAL NETWORKS

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

In addition to what's included in base Anaconda, we need to install the following packages

```
!pip install -U kaleido plotly
!conda install -y -c plotly plotly-orca

# kaleido needs chrome to build images
import kaleido
kaleido.get_chrome_sync()
```

Note

If you are running this on Google Colab the above cell will present an error. This is because Google Colab doesn't use Anaconda to manage the Python packages. However this lecture will still execute as Google Colab has `plotly` installed.

We also need to install JAX to run this lecture

```
!pip install --upgrade jax
```

```
import jax
print(f"JAX backend: {jax.devices()[0].platform}") # to check that gpu is activated␣
↪in environment
```

```
JAX backend: gpu
```

15.1 Overview

Substantial parts of **machine learning** and **artificial intelligence** are about

- approximating an unknown function with a known function
- estimating the known function from a set of data on the left- and right-hand variables

This lecture describes the structure of a plain vanilla **artificial neural network** (ANN) of a type that is widely used to approximate a function f that maps x in a space X into y in a space Y.

To introduce elementary concepts, we study an example in which x and y are scalars.

We'll describe the following concepts that are brick and mortar for neural networks:

- a neuron
- an activation function
- a network of neurons
- A neural network as a composition of functions
- back-propagation and its relationship to the chain rule of differential calculus

15.2 A Deep (but not Wide) Artificial Neural Network

We describe a "deep" neural network of "width" one.

Deep means that the network composes a large number of functions organized into nodes of a graph.

Width refers to the number of right hand side variables on the right hand side of the function being approximated.

Setting "width" to one means that the network composes just univariate functions.

Let $x \in \mathbb{R}$ be a scalar and $y \in \mathbb{R}$ be another scalar.

We assume that y is a nonlinear function of x:

$$y = f(x)$$

We want to approximate $f(x)$ with another function that we define recursively.

For a network of depth $N \geq 1$, each **layer** $i = 1, \ldots N$ consists of

- an input x_i
- an **affine function** $w_i x_i + bI$, where w_i is a scalar **weight** placed on the input x_i and b_i is a scalar **bias**
- an **activation function** h_i that takes $(w_i x_i + b_i)$ as an argument and produces an output x_{i+1}

An example of an activation function h is the **sigmoid** function

$$h(z) = \frac{1}{1 + e^{-z}}$$

Another popular activation function is the **rectified linear unit** (ReLU) function

$$h(z) = \max(0, z)$$

Yet another activation function is the identity function

$$h(z) = z$$

As activation functions below, we'll use the sigmoid function for layers 1 to $N-1$ and the identity function for layer N.

To approximate a function $f(x)$ we construct $\hat{f}(x)$ by proceeding as follows.

Let

$$l_i(x) = w_i x + b_i.$$

We construct $\hat{f}$ by iterating on compositions of functions $h_i \circ l_i$:

$$f(x) \approx \hat{f}(x) = h_N \circ l_N \circ h_{N-1} \circ l_1 \circ \cdots \circ h_1 \circ l_1(x)$$

If $N > 1$, we call the right side a "deep" neural net.

The larger is the integer N, the "deeper" is the neural net.

Evidently, if we know the parameters $\{w_i, b_i\}_{i=1}^N$, then we can compute $\hat{f}(x)$ for a given $x = \tilde{x}$ by iterating on the recursion

$$x_{i+1} = h_i \circ l_i(x_i), \quad , i = 1, \ldots N \tag{15.1}$$

starting from $x_1 = \tilde{x}$.

The value of x_{N+1} that emerges from this iterative scheme equals $\hat{f}(\tilde{x})$.

15.3 Calibrating Parameters

We now consider a neural network like the one describe above with width 1, depth N, and activation functions h_i for $1 \leqslant i \leqslant N$ that map $\mathbb{R}$ into itself.

Let $\{(w_i, b_i)\}_{i=1}^N$ denote a sequence of weights and biases.

As mentioned above, for a given input x_1, our approximating function $\hat{f}$ evaluated at x_1 equals the "output" x_{N+1} from our network that can be computed by iterating on $x_{i+1} = h_i(w_i x_i + b_i)$.

For a given **prediction** $\hat{y}(x)$ and **target** $y = f(x)$, consider the loss function

$$\mathcal{L}(\hat{y}, y)(x) = \frac{1}{2}(\hat{y} - y)^2(x).$$

This criterion is a function of the parameters $\{(w_i, b_i)\}_{i=1}^N$ and the point x.

We're interested in solving the following problem:

$$\min_{\{(w_i, b_i)\}_{i=1}^N} \int \mathcal{L}(x_{N+1}, y)(x) d\mu(x)$$

where $\mu(x)$ is some measure of points $x \in \mathbb{R}$ over which we want a good approximation $\hat{f}(x)$ to $f(x)$.

Stack weights and biases into a vector of parameters p:

$$p = \begin{bmatrix} w_1 \\ b_1 \\ w_2 \\ b_2 \\ \vdots \\ w_N \\ b_N \end{bmatrix}$$

Applying a "poor man's version" of a **stochastic gradient descent** algorithm for finding a zero of a function leads to the following update rule for parameters:

$$p_{k+1} = p_k - \alpha \frac{d\mathcal{L}}{dx_{N+1}} \frac{dx_{N+1}}{dp_k} \tag{15.2}$$

where $\frac{d\mathcal{L}}{dx_{N+1}} = -\left(x_{N+1} - y\right)$ and $\alpha > 0$ is a step size.

(See this and this to gather insights about how stochastic gradient descent relates to Newton's method.)

To implement one step of this parameter update rule, we want the vector of derivatives $\frac{dx_{N+1}}{dp_k}$.

In the neural network literature, this step is accomplished by what is known as **back propagation**.

15.4 Back Propagation and the Chain Rule

Thanks to properties of

- the chain and product rules for differentiation from differential calculus, and
- lower triangular matrices

back propagation can actually be accomplished in one step by

- inverting a lower triangular matrix, and
- matrix multiplication

(This idea is from the last 7 minutes of this great youtube video by MIT's Alan Edelman)

https://youtu.be/rZS2LGiurKY

Here goes.

Define the derivative of $h(z)$ with respect to z evaluated at $z = z_i$ as δ_i:

$$\delta_i = \frac{d}{dz} h(z)|_{z=z_i}$$

or

$$\delta_i = h'\left(w_i x_i + b_i\right).$$

Repeated application of the chain rule and product rule to our recursion (15.1) allows us to obtain:

$$dx_{i+1} = \delta_i \left(dw_i x_i + w_i dx_i + b_i\right)$$

After imposing $dx_1 = 0$, we get the following system of equations:

$$\begin{pmatrix} dx_2 \\ \vdots \\ dx_{N+1} \end{pmatrix} = \underbrace{\begin{pmatrix} \delta_1 w_1 & \delta_1 & 0 & 0 & 0 \\ 0 & 0 & \ddots & 0 & 0 \\ 0 & 0 & 0 & \delta_N w_N & \delta_N \end{pmatrix}}_{D} \begin{pmatrix} dw_1 \\ db_1 \\ \vdots \\ dw_N \\ db_N \end{pmatrix} + \underbrace{\begin{pmatrix} 0 & 0 & 0 & 0 \\ w_2 & 0 & 0 & 0 \\ 0 & \ddots & 0 & 0 \\ 0 & 0 & w_N & 0 \end{pmatrix}}_{L} \begin{pmatrix} dx_2 \\ \vdots \\ dx_{N+1} \end{pmatrix}$$

or

$$dx = Ddp + Ldx$$

which implies that

$$dx = (I - L)^{-1} D dp$$

which in turn implies

$$\begin{pmatrix} dx_{N+1}/dw_1 \\ dx_{N+1}/db_1 \\ \vdots \\ dx_{N+1}/dw_N \\ dx_{N+1}/db_N \end{pmatrix} = e_N (I - L)^{-1} D.$$

We can then solve the above problem by applying our update for p multiple times for a collection of input-output pairs $\{(x_1^i, y^i)\}_{i=1}^M$ that we'll call our "training set".

15.5 Training Set

Choosing a training set amounts to a choice of measure μ in the above formulation of our function approximation problem as a minimization problem.

In this spirit, we shall use a uniform grid of, say, 50 or 200 points.

There are many possible approaches to the minimization problem posed above:

- batch gradient descent in which you use an average gradient over the training set
- stochastic gradient descent in which you sample points randomly and use individual gradients
- something in-between (so-called "mini-batch gradient descent")

The update rule (15.2) described above amounts to a stochastic gradient descent algorithm.

```
from IPython.display import Image
import jax.numpy as jnp
from jax import grad, jit, jacfwd, vmap
from jax import random
import jax
import plotly.graph_objects as go
```

```
# A helper function to randomly initialize weights and biases
# for a dense neural network layer
def random_layer_params(m, n, key, scale=1.):
    w_key, b_key = random.split(key)
    return scale * random.normal(w_key, (n, m)), scale * random.normal(b_key, (n,))

# Initialize all layers for a fully-connected neural network with sizes "sizes"
def init_network_params(sizes, key):
    keys = random.split(key, len(sizes))
    return [random_layer_params(m, n, k) for m, n, k in zip(sizes[:-1], sizes[1:],␣
 ↪keys)]
```

```
def compute_xδw_seq(params, x):
    # Initialize arrays
    δ = jnp.zeros(len(params))
    xs = jnp.zeros(len(params) + 1)
    ws = jnp.zeros(len(params))
```

(continues on next page)

(continued from previous page)

```
    bs = jnp.zeros(len(params))

    h = jax.nn.sigmoid

    xs = xs.at[0].set(x)
    for i, (w, b) in enumerate(params[:-1]):
        output = w * xs[i] + b
        activation = h(output[0, 0])

        # Store elements
        δ = δ.at[i].set(grad(h)(output[0, 0]))
        ws = ws.at[i].set(w[0, 0])
        bs = bs.at[i].set(b[0])
        xs = xs.at[i+1].set(activation)

    final_w, final_b = params[-1]
    preds = final_w * xs[-2] + final_b

    # Store elements
    δ = δ.at[-1].set(1.)
    ws = ws.at[-1].set(final_w[0, 0])
    bs = bs.at[-1].set(final_b[0])
    xs = xs.at[-1].set(preds[0, 0])

    return xs, δ, ws, bs


def loss(params, x, y):
    xs, δ, ws, bs = compute_xδw_seq(params, x)
    preds = xs[-1]

    return 1 / 2 * (y - preds) ** 2
```

```
# Parameters
N = 3  # Number of layers
layer_sizes = [1, ] * (N + 1)
param_scale = 0.1
step_size = 0.01
params = init_network_params(layer_sizes, random.PRNGKey(1))
```

```
x = 5
y = 3
xs, δ, ws, bs = compute_xδw_seq(params, x)
```

```
dxs_ad = jacfwd(lambda params, x: compute_xδw_seq(params, x)[0], argnums=0)(params, x)
dxs_ad_mat = jnp.block([dx.reshape((-1, 1)) for dx_tuple in dxs_ad for dx in dx_tuple␣
↪])[1:]
```

```
jnp.block([[δ * xs[:-1]], [δ]])
```

```
Array([[1.0165801 , 0.06087969, 0.09382247],
       [0.20331602, 0.08501981, 1.        ]], dtype=float32)
```

```
L = jnp.diag(δ * ws, k=-1)
```

(continues on next page)

(continued from previous page)

```
L = L[1:, 1:]

D = jax.scipy.linalg.block_diag(*[row.reshape((1, 2)) for row in jnp.block([[δ * xs[:-
↪1]], [δ]]).T])

dxs_la = jax.scipy.linalg.solve_triangular(jnp.eye(N) - L, D, lower=True)
```

```
# Check that the `dx` generated by the linear algebra method
# are the same as the ones generated using automatic differentiation
jnp.max(jnp.abs(dxs_ad_mat - dxs_la))
```

```
Array(0., dtype=float32)
```

```
grad_loss_ad = jnp.block([dx.reshape((-1, 1)) for dx_tuple in grad(loss)(params, x,␣
↪y) for dx in dx_tuple ])
```

```
# Check that the gradient of the loss is the same for both approaches
jnp.max(jnp.abs(-(y - xs[-1]) * dxs_la[-1] - grad_loss_ad))
```

```
Array(5.9604645e-08, dtype=float32)
```

```
@jit
def update_ad(params, x, y):
    grads = grad(loss)(params, x, y)
    return [(w - step_size * dw, b - step_size * db)
          for (w, b), (dw, db) in zip(params, grads)]

@jit
def update_la(params, x, y):
    xs, δ, ws, bs = compute_xδw_seq(params, x)
    N = len(params)
    L = jnp.diag(δ * ws, k=-1)
    L = L[1:, 1:]

    D = jax.scipy.linalg.block_diag(*[row.reshape((1, 2)) for row in jnp.block([[δ *␣
↪xs[:-1]], [δ]]).T])

    dxs_la = jax.scipy.linalg.solve_triangular(jnp.eye(N) - L, D, lower=True)

    grads = -(y - xs[-1]) * dxs_la[-1]

    return [(w - step_size * dw, b - step_size * db)
            for (w, b), (dw, db) in zip(params, grads.reshape((-1, 2)))]
```

```
# Check that both updates are the same
update_la(params, x, y)
```

```
[(Array([[-0.00826643]], dtype=float32), Array([0.94700736], dtype=float32)),
 (Array([[-2.0638916]], dtype=float32), Array([-0.7872697], dtype=float32)),
 (Array([[1.6248171]], dtype=float32), Array([1.5765371], dtype=float32))]
```

```
update_ad(params, x, y)
```

```
[(Array([[-0.00826644]], dtype=float32), Array([0.94700736], dtype=float32)),
 (Array([[-2.0638916]], dtype=float32), Array([-0.7872697], dtype=float32)),
 (Array([[1.6248171]], dtype=float32), Array([1.5765371], dtype=float32))]
```

15.6 Example 1

Consider the function

$$f(x) = -3x + 2$$

on $[0.5, 3]$.

We use a uniform grid of 200 points and update the parameters for each point on the grid 300 times.

h_i is the sigmoid activation function for all layers except the final one for which we use the identity function and $N = 3$.

Weights are initialized randomly.

```
def f(x):
    return -3 * x + 2

M = 200
grid = jnp.linspace(0.5, 3, num=M)
f_val = f(grid)
```

```
indices = jnp.arange(M)
key = random.PRNGKey(0)

def train(params, grid, f_val, key, num_epochs=300):
    for epoch in range(num_epochs):
        key, _ = random.split(key)
        random_permutation = random.permutation(random.PRNGKey(1), indices)
        for x, y in zip(grid[random_permutation], f_val[random_permutation]):
            params = update_la(params, x, y)

    return params
```

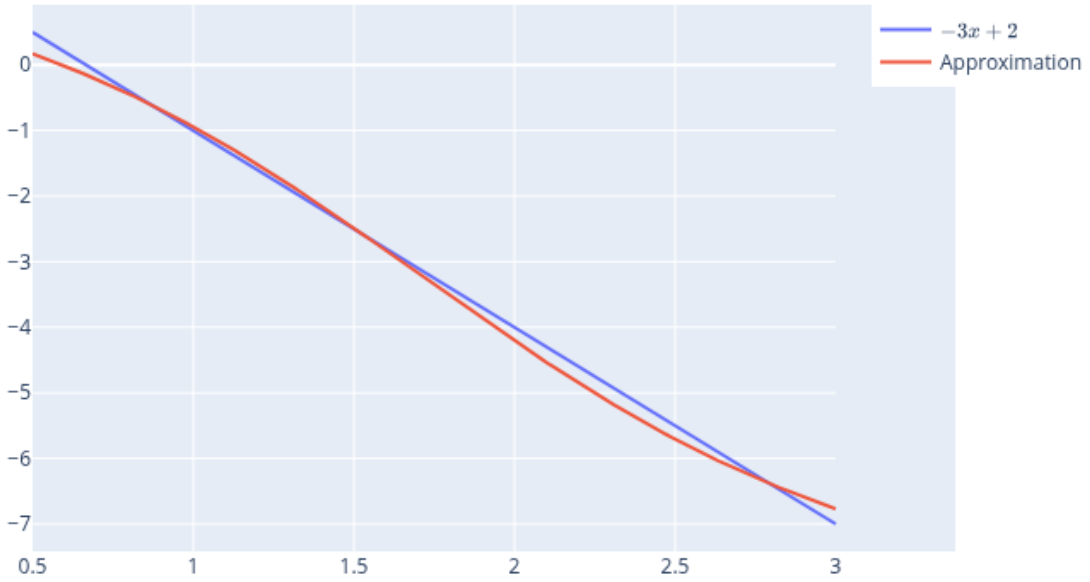
```
# Parameters
N = 3  # Number of layers
layer_sizes = [1, ] * (N + 1)
params_ex1 = init_network_params(layer_sizes, key)
```

```
%%time
params_ex1 = train(params_ex1, grid, f_val, key, num_epochs=500)
```

```
CPU times: user 19.9 s, sys: 4.04 s, total: 23.9 s
Wall time: 17.4 s
```

```
predictions = vmap(compute_xδw_seq, in_axes=(None, 0))(params_ex1, grid)[0][:, -1]
```

```
fig = go.Figure()
fig.add_trace(go.Scatter(x=grid, y=f_val, name=r'$-3x+2$'))
fig.add_trace(go.Scatter(x=grid, y=predictions, name='Approximation'))

# Export to PNG file
Image(fig.to_image(format="png"))
# fig.show() will provide interactive plot when running
# notebook locally
```



15.7 How Deep?

It is fun to think about how deepening the neural net for the above example affects the quality of approximation

- If the network is too deep, you'll run into the vanishing gradient problem
- Other parameters such as the step size and the number of epochs can be as important or more important than the number of layers in the situation considered in this lecture.
- Indeed, since f is a linear function of x, a one-layer network with the identity map as an activation would probably work best.

15.8 Example 2

We use the same setup as for the previous example with

$$f(x) = \log(x)$$

```
def f(x):
    return jnp.log(x)

grid = jnp.linspace(0.5, 3, num=M)
f_val = f(grid)
```

```
# Parameters
N = 1  # Number of layers
layer_sizes = [1, ] * (N + 1)
params_ex2_1 = init_network_params(layer_sizes, key)
```

```
# Parameters
N = 2  # Number of layers
layer_sizes = [1, ] * (N + 1)
params_ex2_2 = init_network_params(layer_sizes, key)
```

```
# Parameters
N = 3  # Number of layers
layer_sizes = [1, ] * (N + 1)
params_ex2_3 = init_network_params(layer_sizes, key)
```

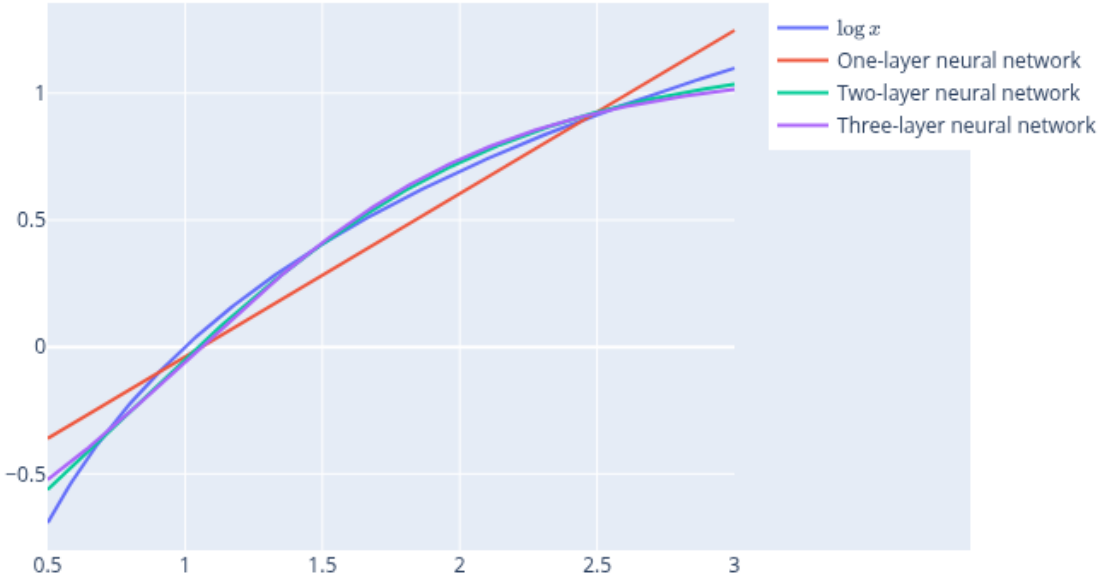
```
params_ex2_1 = train(params_ex2_1, grid, f_val, key, num_epochs=300)
```

```
params_ex2_2 = train(params_ex2_2, grid, f_val, key, num_epochs=300)
```

```
params_ex2_3 = train(params_ex2_3, grid, f_val, key, num_epochs=300)
```

```
predictions_1 = vmap(compute_xδw_seq, in_axes=(None, 0))(params_ex2_1, grid)[0][:, -1]
predictions_2 = vmap(compute_xδw_seq, in_axes=(None, 0))(params_ex2_2, grid)[0][:, -1]
predictions_3 = vmap(compute_xδw_seq, in_axes=(None, 0))(params_ex2_3, grid)[0][:, -1]
```

```
fig = go.Figure()
fig.add_trace(go.Scatter(x=grid, y=f_val, name=r'$\log{x}$'))
fig.add_trace(go.Scatter(x=grid, y=predictions_1, name='One-layer neural network'))
fig.add_trace(go.Scatter(x=grid, y=predictions_2, name='Two-layer neural network'))
fig.add_trace(go.Scatter(x=grid, y=predictions_3, name='Three-layer neural network'))

# Export to PNG file
Image(fig.to_image(format="png"))
# fig.show() will provide interactive plot when running
# notebook locally
```


log x
One-layer neural network
Two-layer neural network
Three-layer neural network
1
0.5
0
−0.5
0.5
1
1.5
2
2.5
3

CHAPTER

SIXTEEN

RANDOMIZED RESPONSE SURVEYS

16.1 Overview

Social stigmas can inhibit people from confessing potentially embarrassing activities or opinions.

When people are reluctant to participate a sample survey about personally sensitive issues, they might decline to participate, and even if they do participate, they might choose to provide incorrect answers to sensitive questions.

These problems induce **selection** biases that present challenges to interpreting and designing surveys.

To illustrate how social scientists have thought about estimating the prevalence of such embarrassing activities and opinions, this lecture describes a classic approach of S. L. Warner [Warner, 1965].

Warner used elementary probability to construct a way to protect the privacy of **individual** respondents to surveys while still estimating the fraction of a **collection** of individuals who have a socially stigmatized characteristic or who engage in a socially stigmatized activity.

Warner's idea was to add **noise** between the respondent's answer and the **signal** about that answer that the survey maker ultimately receives.

Knowing about the structure of the noise assures the respondent that the survey maker does not observe his answer.

Statistical properties of the noise injection procedure provide the respondent **plausible deniability**.

Related ideas underlie modern **differential privacy** systems.

(See https://en.wikipedia.org/wiki/Differential_privacy)

16.2 Warner's Strategy

As usual, let's bring in the Python modules we'll be using.

```
import numpy as np
import pandas as pd
```

Suppose that every person in population either belongs to Group A or Group B.

We want to estimate the proportion π who belong to Group A while protecting individual respondents' privacy.

Warner [Warner, 1965] proposed and analyzed the following procedure.

- A random sample of n people is drawn with replacement from the population and each person is interviewed.
- Draw n random samples from the population with replacement and interview each person.
- Prepare a **random spinner** that with p probability points to the Letter A and with $(1-p)$ probability points to the Letter B.

- Each subject spins a random spinner and sees an outcome (A or B) that the interviewer does **not observe**.
- The subject states whether he belongs to the group to which the spinner points.
- If the spinner points to the group that the spinner belongs, the subject reports "yes"; otherwise he reports "no".
- The subject answers the question truthfully.

Warner constructed a maximum likelihood estimators of the proportion of the population in set A.

Let

- π : True probability of A in the population
- p : Probability that the spinner points to A
- $X_i = \begin{cases} 1, \text{ if the } i\text{th subject says yes} \\ 0, \text{ if the } i\text{th subject says no} \end{cases}$

Index the sample set so that the first n_1 report "yes", while the second $n - n_1$ report "no".

The likelihood function of a sample set is

$$L = [\pi p + (1-\pi)(1-p)]^{n_1} [(1-\pi)p + \pi(1-p)]^{n-n_1} \tag{16.1}$$

The log of the likelihood function is:

$$\log(L) = n_1 \log [\pi p + (1-\pi)(1-p)] + (n-n_1) \log [(1-\pi)p + \pi(1-p)] \tag{16.2}$$

The first-order necessary condition for maximizing the log likelihood function with respect to π is:

$$\frac{(n-n_1)(2p-1)}{(1-\pi)p + \pi(1-p)} = \frac{n_1(2p-1)}{\pi p + (1-\pi)(1-p)}$$

or

$$\pi p + (1-\pi)(1-p) = \frac{n_1}{n} \tag{16.3}$$

If $p \neq \frac{1}{2}$, then the maximum likelihood estimator (MLE) of π is:

$$\hat{\pi} = \frac{p-1}{2p-1} + \frac{n_1}{(2p-1)n} \tag{16.4}$$

We compute the mean and variance of the MLE estimator $\hat{\pi}$ to be:

$$\begin{aligned} \mathbb{E}(\hat{\pi}) &= \frac{1}{2p-1} \left[p - 1 + \frac{1}{n} \sum_{i=1}^{n} \mathbb{E} X_i \right] \\ &= \frac{1}{2p-1} [p - 1 + \pi p + (1-\pi)(1-p)] \\ &= \pi \end{aligned} \tag{16.5}$$

and

$$\begin{aligned} Var(\hat{\pi}) &= \frac{nVar(X_i)}{(2p-1)^2 n^2} \\ &= \frac{[\pi p + (1-\pi)(1-p)] [(1-\pi)p + \pi(1-p)]}{(2p-1)^2 n^2} \\ &= \frac{\frac{1}{4} + (2p^2 - 2p + \frac{1}{2})(-2\pi^2 + 2\pi - \frac{1}{2})}{(2p-1)^2 n^2} \\ &= \frac{1}{n} \left[\frac{1}{16(p - \frac{1}{2})^2} - (\pi - \frac{1}{2})^2 \right] \end{aligned} \tag{16.6}$$

Equation (16.5) indicates that $\hat{\pi}$ is an **unbiased estimator** of π while equation (16.6) tell us the variance of the estimator.

To compute a confidence interval, first rewrite (16.6) as:

$$Var(\hat{\pi}) = \frac{\frac{1}{4} - (\pi - \frac{1}{2})^2}{n} + \frac{\frac{1}{16(p-\frac{1}{2})^2} - \frac{1}{4}}{n} \tag{16.7}$$

This equation indicates that the variance of $\hat{\pi}$ can be represented as a sum of the variance due to sampling plus the variance due to the random device.

From the expressions above we can find that:

- When p is $\frac{1}{2}$, expression (16.1) degenerates to a constant.
- When p is 1 or 0, the randomized estimate degenerates to an estimator without randomized sampling.

We shall only discuss situations in which $p \in (\frac{1}{2}, 1)$

(a situation in which $p \in (0, \frac{1}{2})$ is symmetric).

From expressions (16.5) and (16.7) we can deduce that:

- The MSE of $\hat{\pi}$ decreases as p increases.

16.3 Comparing Two Survey Designs

Let's compare the preceding randomized-response method with a stylized non-randomized response method.

In our non-randomized response method, we suppose that:

- Members of Group A tells the truth with probability T_a while the members of Group B tells the truth with probability T_b
- Y_i is 1 or 0 according to whether the sample's ith member's report is in Group A or not.

Then we can estimate π as:

$$\hat{\pi} = \frac{\sum_{i=1}^{n} Y_i}{n} \tag{16.8}$$

We calculate the expectation, bias, and variance of the estimator to be:

$$\mathbb{E}(\hat{\pi}) = \pi T_a + [(1 - \pi)(1 - T_b)] \tag{16.9}$$

$$\begin{aligned} Bias(\hat{\pi}) &= \mathbb{E}(\hat{\pi} - \pi) \\ &= \pi[T_a + T_b - 2] + [1 - T_b] \end{aligned} \tag{16.10}$$

$$Var(\hat{\pi}) = \frac{[\pi T_a + (1 - \pi)(1 - T_b)]\,[1 - \pi T_a - (1 - \pi)(1 - T_b)]}{n} \tag{16.11}$$

It is useful to define a

$$\text{MSE Ratio} = \frac{\text{Mean Square Error Randomized}}{\text{Mean Square Error Regular}}$$

We can compute MSE Ratios for different survey designs associated with different parameter values.

The following Python code computes objects we want to stare at in order to make comparisons under different values of π_A and n:

```
class Comparison:
    def __init__(self, A, n):
        self.A = A
        self.n = n
        TaTb = np.array([[0.95,  1], [0.9,   1], [0.7,    1],
                         [0.5,   1], [1,  0.95], [1,    0.9],
                         [1,   0.7], [1,   0.5], [0.95, 0.95],
                         [0.9, 0.9], [0.7, 0.7], [0.5,  0.5]])
        self.p_arr = np.array([0.6, 0.7, 0.8, 0.9])
        self.p_map = dict(zip(self.p_arr, [f"MSE Ratio: p = {x}" for x in self.p_
↪arr]))
        self.template = pd.DataFrame(columns=self.p_arr)
        self.template[['T_a','T_b']] = TaTb
        self.template['Bias'] = None

    def theoretical(self):
        A = self.A
        n = self.n
        df = self.template.copy()
        df['Bias'] = A * (df['T_a'] + df['T_b'] - 2) + (1 - df['T_b'])
        for p in self.p_arr:
            df[p] = (1 / (16 * (p - 1/2)**2) - (A - 1/2)**2) / n / \
                    (df['Bias']**2 + ((A * df['T_a'] + (1 - A) * (1 - df['T_b'])) *␣
↪(1 - A * df['T_a'] - (1 - A) * (1 - df['T_b'])) / n))
            df[p] = df[p].round(2)
        df = df.set_index(["T_a", "T_b", "Bias"]).rename(columns=self.p_map)
        return df

    def MCsimulation(self, size=1000, seed=123456):
        A = self.A
        n = self.n
        df = self.template.copy()
        rng = np.random.default_rng(seed)
        sample = rng.random((size, self.n)) <= A
        random_device = rng.random((size, n))
        mse_rd = {}
        for p in self.p_arr:
            spinner = random_device <= p
            rd_answer = sample * spinner + (1 - sample) * (1 - spinner)
            n1 = rd_answer.sum(axis=1)
            pi_hat = (p - 1) / (2 * p - 1) + n1 / n / (2 * p - 1)
            mse_rd[p] = np.sum((pi_hat - A)**2)
        for inum, irow in df.iterrows():
            truth_a = rng.random((size, self.n)) <= irow.T_a
            truth_b = rng.random((size, self.n)) <= irow.T_b
            trad_answer = sample * truth_a + (1 - sample) * (1 - truth_b)
            pi_trad = trad_answer.sum(axis=1) / n
            df.loc[inum, 'Bias'] = pi_trad.mean() - A
            mse_trad = np.sum((pi_trad - A)**2)
            for p in self.p_arr:
                df.loc[inum, p] = (mse_rd[p] / mse_trad).round(2)
        df = df.set_index(["T_a", "T_b", "Bias"]).rename(columns=self.p_map)
        return df
```

Let's put the code to work for parameter values

- $\pi_A = 0.6$
- $n = 1000$

We can generate MSE Ratios theoretically using the above formulas.

We can also perform Monte Carlo simulations of a MSE Ratio.

```
cp1 = Comparison(0.6, 1000)
df1_theoretical = cp1.theoretical()
df1_theoretical
```

```
                MSE Ratio: p = 0.6  MSE Ratio: p = 0.7  MSE Ratio: p = 0.8  \
T_a  T_b  Bias
0.95 1.00 -0.03               5.45                1.36                0.60
0.90 1.00 -0.06               1.62                0.40                0.18
0.70 1.00 -0.18               0.19                0.05                0.02
0.50 1.00 -0.30               0.07                0.02                0.01
1.00 0.95  0.02               9.82                2.44                1.08
     0.90  0.04               3.41                0.85                0.37
     0.70  0.12               0.43                0.11                0.05
     0.50  0.20               0.16                0.04                0.02
0.95 0.95 -0.01              18.25                4.54                2.00
0.90 0.90 -0.02               9.70                2.41                1.06
0.70 0.70 -0.06               1.62                0.40                0.18
0.50 0.50 -0.10               0.61                0.15                0.07

                MSE Ratio: p = 0.9
T_a  T_b  Bias
0.95 1.00 -0.03               0.33
0.90 1.00 -0.06               0.10
0.70 1.00 -0.18               0.01
0.50 1.00 -0.30               0.00
1.00 0.95  0.02               0.60
     0.90  0.04               0.21
     0.70  0.12               0.03
     0.50  0.20               0.01
0.95 0.95 -0.01               1.11
0.90 0.90 -0.02               0.59
0.70 0.70 -0.06               0.10
0.50 0.50 -0.10               0.04
```

```
df1_mc = cp1.MCsimulation()
df1_mc
```

```
                   MSE Ratio: p = 0.6 MSE Ratio: p = 0.7 MSE Ratio: p = 0.8  \
T_a  T_b  Bias
0.95 1.00 -0.030191              5.37               1.34               0.62
0.90 1.00 -0.060081              1.61                0.4               0.19
0.70 1.00 -0.180543              0.19               0.05               0.02
0.50 1.00 -0.29977               0.07               0.02               0.01
1.00 0.95  0.019297             10.18               2.54               1.18
     0.90  0.03985               3.41               0.85                0.4
     0.70  0.119465              0.43               0.11               0.05
     0.50  0.199398              0.16               0.04               0.02
0.95 0.95 -0.010326             16.91               4.21               1.96
0.90 0.90 -0.019838              9.67               2.41               1.12
0.70 0.70 -0.06076               1.58               0.39               0.18
0.50 0.50 -0.100091              0.61               0.15               0.07

                   MSE Ratio: p = 0.9
```

(continues on next page)

(continued from previous page)

```
T_a  T_b  Bias
0.95 1.00 -0.030191              0.35
0.90 1.00 -0.060081              0.11
0.70 1.00 -0.180543              0.01
0.50 1.00 -0.29977                0.0
1.00 0.95  0.019297              0.67
     0.90  0.03985               0.22
     0.70  0.119465              0.03
     0.50  0.199398              0.01
0.95 0.95 -0.010326               1.1
0.90 0.90 -0.019838              0.63
0.70 0.70 -0.06076                0.1
0.50 0.50 -0.100091              0.04
```

The theoretical calculations do a good job of predicting Monte Carlo results.

We see that in many situations, especially when the bias is not small, the MSE of the randomized-sampling methods is smaller than that of the non-randomized sampling method.

These differences become larger as p increases.

By adjusting parameters π_A and n, we can study outcomes in different situations.

For example, for another situation described in Warner [Warner, 1965]:

- $\pi_A = 0.5$
- $n = 1000$

we can use the code

```
cp2 = Comparison(0.5, 1000)
df2_theoretical = cp2.theoretical()
df2_theoretical
```

```
                  MSE Ratio: p = 0.6  MSE Ratio: p = 0.7  MSE Ratio: p = 0.8  \
T_a  T_b  Bias
0.95 1.00 -0.025                7.15                1.79                0.79
0.90 1.00 -0.050                2.27                0.57                0.25
0.70 1.00 -0.150                0.27                0.07                0.03
0.50 1.00 -0.250                0.10                0.02                0.01
1.00 0.95  0.025                7.15                1.79                0.79
     0.90  0.050                2.27                0.57                0.25
     0.70  0.150                0.27                0.07                0.03
     0.50  0.250                0.10                0.02                0.01
0.95 0.95  0.000               25.00                6.25                2.78
0.90 0.90  0.000               25.00                6.25                2.78
0.70 0.70  0.000               25.00                6.25                2.78
0.50 0.50  0.000               25.00                6.25                2.78

                  MSE Ratio: p = 0.9
T_a  T_b  Bias
0.95 1.00 -0.025                0.45
0.90 1.00 -0.050                0.14
0.70 1.00 -0.150                0.02
0.50 1.00 -0.250                0.01
1.00 0.95  0.025                0.45
     0.90  0.050                0.14
     0.70  0.150                0.02
```

(continues on next page)

(continued from previous page)

```
     0.50  0.250                0.01
0.95 0.95  0.000                1.56
0.90 0.90  0.000                1.56
0.70 0.70  0.000                1.56
0.50 0.50  0.000                1.56
```

```
df2_mc = cp2.MCsimulation()
df2_mc
```

```
                    MSE Ratio: p = 0.6 MSE Ratio: p = 0.7 MSE Ratio: p = 0.8  \
T_a  T_b  Bias
0.95 1.00 -0.025243               7.17               1.78               0.83
0.90 1.00 -0.050271               2.29               0.57               0.27
0.70 1.00 -0.150607               0.28               0.07               0.03
0.50 1.00 -0.250029                0.1               0.03               0.01
1.00 0.95  0.024097               7.63               1.89               0.89
     0.90  0.049794               2.33               0.58               0.27
     0.70  0.149336               0.28               0.07               0.03
     0.50  0.249073                0.1               0.03               0.01
0.95 0.95 -0.000357              23.53               5.84               2.74
0.90 0.90  0.000068              25.04               6.21               2.91
0.70 0.70 -0.00079               26.04               6.46               3.03
0.50 0.50 -0.000077              24.26               6.02               2.82

                    MSE Ratio: p = 0.9
T_a  T_b  Bias
0.95 1.00 -0.025243                0.47
0.90 1.00 -0.050271                0.15
0.70 1.00 -0.150607                0.02
0.50 1.00 -0.250029                0.01
1.00 0.95  0.024097                 0.5
     0.90  0.049794                0.15
     0.70  0.149336                0.02
     0.50  0.249073                0.01
0.95 0.95 -0.000357                1.54
0.90 0.90  0.000068                1.64
0.70 0.70 -0.00079                  1.7
0.50 0.50 -0.000077                1.59
```

We can also revisit a calculation in the concluding section of Warner [Warner, 1965] in which

- $\pi_A = 0.6$
- $n = 2000$

We use the code

```
cp3 = Comparison(0.6, 2000)
df3_theoretical = cp3.theoretical()
df3_theoretical
```

```
                  MSE Ratio: p = 0.6  MSE Ratio: p = 0.7  MSE Ratio: p = 0.8  \
T_a  T_b  Bias
0.95 1.00 -0.03                 3.05                0.76                0.33
0.90 1.00 -0.06                 0.84                0.21                0.09
0.70 1.00 -0.18                 0.10                0.02                0.01
0.50 1.00 -0.30                 0.03                0.01                0.00
```

(continues on next page)

(continued from previous page)

```
1.00 0.95  0.02                6.03                1.50                0.66
     0.90  0.04                1.82                0.45                0.20
     0.70  0.12                0.22                0.05                0.02
     0.50  0.20                0.08                0.02                0.01
0.95 0.95 -0.01               14.12                3.51                1.55
0.90 0.90 -0.02                5.98                1.49                0.66
0.70 0.70 -0.06                0.84                0.21                0.09
0.50 0.50 -0.10                0.31                0.08                0.03

                 MSE Ratio: p = 0.9
T_a  T_b  Bias
0.95 1.00 -0.03                0.19
0.90 1.00 -0.06                0.05
0.70 1.00 -0.18                0.01
0.50 1.00 -0.30                0.00
1.00 0.95  0.02                0.37
     0.90  0.04                0.11
     0.70  0.12                0.01
     0.50  0.20                0.00
0.95 0.95 -0.01                0.86
0.90 0.90 -0.02                0.36
0.70 0.70 -0.06                0.05
0.50 0.50 -0.10                0.02
```

```
df3_mc = cp3.MCsimulation()
df3_mc
```

```
                   MSE Ratio: p = 0.6 MSE Ratio: p = 0.7 MSE Ratio: p = 0.8  \
T_a  T_b  Bias
0.95 1.00 -0.029752               3.04                0.74                0.33
0.90 1.00 -0.059862               0.82                 0.2                0.09
0.70 1.00 -0.179691               0.09                0.02                0.01
0.50 1.00 -0.299452               0.03                0.01                 0.0
1.00 0.95  0.020192               5.85                1.43                0.64
     0.90  0.040388               1.75                0.43                0.19
     0.70  0.120138               0.21                0.05                0.02
     0.50  0.199962               0.08                0.02                0.01
0.95 0.95 -0.009622              14.35                3.51                1.57
0.90 0.90 -0.019695               6.05                1.48                0.66
0.70 0.70 -0.059774               0.83                 0.2                0.09
0.50 0.50 -0.1                     0.3                0.07                0.03

                   MSE Ratio: p = 0.9
T_a  T_b  Bias
0.95 1.00 -0.029752               0.18
0.90 1.00 -0.059862               0.05
0.70 1.00 -0.179691               0.01
0.50 1.00 -0.299452                0.0
1.00 0.95  0.020192               0.35
     0.90  0.040388                0.1
     0.70  0.120138               0.01
     0.50  0.199962                0.0
0.95 0.95 -0.009622               0.85
0.90 0.90 -0.019695               0.36
0.70 0.70 -0.059774               0.05
0.50 0.50 -0.1                    0.02
```

Evidently, as n increases, the randomized response method does better performance in more situations.

16.4 Concluding Remarks

This QuantEcon lecture describes some alternative randomized response surveys.

That lecture presents a utilitarian analysis of those alternatives conducted by Lars Ljungqvist [Ljungqvist, 1993].

```
import matplotlib.pyplot as plt
import numpy as np
```

CHAPTER

SEVENTEEN

EXPECTED UTILITIES OF RANDOM RESPONSES

17.1 Overview

This QuantEcon lecture describes randomized response surveys in the tradition of Warner [Warner, 1965] that are designed to protect respondents' privacy.

Lars Ljungqvist [Ljungqvist, 1993] analyzed how a respondent's decision about whether to answer truthfully depends on **expected utility**.

The lecture tells how Ljungqvist used his framework to shed light on alternative randomized response survey techniques proposed, for example, by [Lanke, 1975], [Lanke, 1976], [Leysieffer and Warner, 1976], [Anderson, 1976], [Fligner *et al.*, 1977], [Greenberg *et al.*, 1977], [Greenberg *et al.*, 1969].

17.2 Privacy Measures

We consider randomized response models with only two possible answers, "yes" and "no."

The design determines probabilities

$$\Pr(\text{yes}|A) = 1 - \Pr(\text{no}|A)$$
$$\Pr(\text{yes}|A^{'}) = 1 - \Pr(\text{no}|A^{'})$$

These design probabilities in turn can be used to compute the conditional probability of belonging to the sensitive group A for a given response, say r:

$$\Pr(A|r) = \frac{\pi_A \Pr(r|A)}{\pi_A \Pr(r|A) + (1 - \pi_A)\Pr(r|A^{'})} \tag{17.1}$$

17.3 Zoo of Concepts

At this point we describe some concepts proposed by various researchers

17.3.1 Leysieffer and Warner(1976)

The response r is regarded as jeopardizing with respect to A or $A^{'}$ if

$$\begin{aligned} &\Pr(A|r) > \pi_A \\ &\text{or} \\ &\Pr(A^{'}|r) > 1 - \pi_A \end{aligned} \tag{17.2}$$

From Bayes's rule:

$$\frac{\Pr(A|r)}{\Pr(A^{'}|r)} \times \frac{(1-\pi_A)}{\pi_A} = \frac{\Pr(r|A)}{\Pr(r|A^{'})} \tag{17.3}$$

If this expression is greater (less) than unity, it follows that r is jeopardizing with respect to $A(A^{'})$. Then, the natural measure of jeopardy will be:

$$\begin{aligned} g(r|A) &= \frac{\Pr(r|A)}{\Pr(r|A^{'})} \\ &\text{and} \\ g(r|A^{'}) &= \frac{\Pr(r|A^{'})}{\Pr(r|A)} \end{aligned} \tag{17.4}$$

Suppose, without loss of generality, that $\Pr(\text{yes}|A) > \Pr(\text{yes}|A^{'})$, then a yes (no) answer is jeopardizing with respect $A(A^{'})$, that is,

$$\begin{aligned} &g(\text{yes}|A) > 1 \\ &\text{and} \\ &g(\text{no}|A^{'}) > 1 \end{aligned}$$

Leysieffer and Warner proved that the variance of the estimate can only be decreased through an increase in one or both of these two measures of jeopardy.

An efficient randomized response model is, therefore, any model that attains the maximum acceptable levels of jeopardy that are consistent with cooperation of the respondents.

As a special example, Leysieffer and Warner considered "a problem in which there is no jeopardy in a no answer"; that is, $g(\text{no}|A^{'})$ can be of unlimited magnitude.

Evidently, an optimal design must have

$$\Pr(\text{yes}|A) = 1$$

which implies that

$$\Pr(A|\text{no}) = 0$$

17.3.2 Lanke(1976)

Lanke (1975) [Lanke, 1975] argued that "it is membership in Group A that people may want to hide, not membership in the complementary Group A'."

For that reason, Lanke (1976) [Lanke, 1976] argued that an appropriate measure of protection is to minimize

$$\max\{\Pr(A|\text{yes}), \Pr(A|\text{no})\} \tag{17.5}$$

Holding this measure constant, he explained under what conditions the smallest variance of the estimate was achieved with the unrelated question model or Warner's (1965) original model.

17.3.3 2.3 Fligner, Policello, and Singh

Fligner, Policello, and Singh reached similar conclusion as Lanke (1976). [Fligner *et al.*, 1977]

They measured "private protection" as

$$\frac{1 - \max\{\Pr(A|\text{yes}), \Pr(A|\text{no})\}}{1 - \pi_A} \tag{17.6}$$

17.3.4 2.4 Greenberg, Kuebler, Abernathy, and Horvitz (1977)

[Greenberg *et al.*, 1977]

Greenberg, Kuebler, Abernathy, and Horvitz (1977) stressed the importance of examining the risk to respondents who do not belong to A as well as the risk to those who do belong to the sensitive group.

They defined the hazard for an individual in A as the probability that he or she is perceived as belonging to A:

$$\Pr(\text{yes}|A) \times \Pr(A|\text{yes}) + \Pr(\text{no}|A) \times \Pr(A|\text{no}) \tag{17.7}$$

Similarly, the hazard for an individual who does not belong to A would be

$$\Pr(\text{yes}|A^{'}) \times \Pr(A|\text{yes}) + \Pr(\text{no}|A^{'}) \times \Pr(A|\text{no}) \tag{17.8}$$

Greenberg et al. (1977) also considered an alternative related measure of hazard that "is likely to be closer to the actual concern felt by a respondent."

The "limited hazard" for an individual in A and $A^{'}$ is

$$\Pr(\text{yes}|A) \times \Pr(A|\text{yes}) \tag{17.9}$$

and

$$\Pr(\text{yes}|A^{'}) \times \Pr(A|\text{yes}) \tag{17.10}$$

This measure is just the first term in (17.7), i.e., the probability that an individual answers "yes" and is perceived to belong to A.

17.4 Respondent's Expected Utility

17.4.1 Truth Border

Key assumptions that underlie a randomized response technique for estimating the fraction of a population that belongs to A are:

- **Assumption 1**: Respondents feel discomfort from being thought of as belonging to A.
- **Assumption 2**: Respondents prefer to answer questions truthfully than to lie, so long as the cost of doing so is not too high. The cost is taken to be the discomfort in 1.

Let r_i denote individual i's response to the randomized question.

r_i can only take values "yes" or "no".

For a given design of a randomized response interview and a given belief about the fraction of the population that belongs to A, the respondent's answer is associated with a conditional probability $\Pr(A|r_i)$ that the individual belongs to A.

Given r_i and complete privacy, the individual's utility is higher if r_i represents a truthful answer rather than a lie.

In terms of a respondent's expected utility as a function of $\Pr(A|r_i)$ and r_i

- The higher is $\Pr(A|r_i)$, the lower isindividual i's expected utility.
- expected utility is higher if r_i represents a truthful answer rather than a lie

Define:

- $\phi_i \in \{\text{truth}, \text{lie}\}$, a dichotomous variable that indicates whether or not r_i is a truthful statement.
- $U_i\left(\Pr(A|r_i), \phi_i\right)$, a utility function that is differentiable in its first argument, summarizes individual i's expected utility.

Then there is an r_i such that

$$\frac{\partial U_i\left(\Pr(A|r_i), \phi_i\right)}{\partial \Pr(A|r_i)} < 0, \text{ for } \phi_i \in \{\text{truth}, \text{lie}\} \tag{17.11}$$

and

$$U_i\left(\Pr(A|r_i), \text{truth}\right) > U_i\left(\Pr(A|r_i), \text{lie}\right), \text{ for } \Pr(A|r_i) \in [0, 1] \tag{17.12}$$

Suppose now that correct answer for individual i is "yes".

Individual i would choose to answer truthfully if

$$U_i\left(\Pr(A|\text{yes}), \text{truth}\right) \geq U_i\left(\Pr(A|\text{no}), \text{lie}\right) \tag{17.13}$$

If the correct answer is "no", individual i would volunteer the correct answer only if

$$U_i\left(\Pr(A|\text{no}), \text{truth}\right) \geq U_i\left(\Pr(A|\text{yes}), \text{lie}\right) \tag{17.14}$$

Assume that

$$\Pr(A|\text{yes}) > \pi_A > \Pr(A|\text{no})$$

so that a "yes" answer increases the odds that an individual belongs to A.

Constraint (17.14) holds for sure.

Consequently, constraint (17.13) becomes the single necessary condition for individual i always to answer truthfully.

At equality, constraint (10.a) determines conditional probabilities that make the individual indifferent between telling the truth and lying when the correct answer is "yes":

$$U_i\left(\Pr(A|\text{yes}), \text{truth}\right) = U_i\left(\Pr(A|\text{no}), \text{lie}\right) \tag{17.15}$$

Equation (17.15) defines a "truth border".

Differentiating (17.15) with respect to the conditional probabilities shows that the truth border has a positive slope in the space of conditional probabilities:

$$\frac{\partial \Pr(A|\text{no})}{\partial \Pr(A|\text{yes})} = \frac{\frac{\partial U_i(\Pr(A|\text{yes}), \text{truth})}{\partial \Pr(A|\text{yes})}}{\frac{\partial U_i(\Pr(A|\text{no}), \text{lie})}{\partial \Pr(A|\text{no})}} > 0 \tag{17.16}$$

The source of the positive relationship is:

- The individual is willing to volunteer a truthful "yes" answer so long as the utility from doing so (i.e., the left side of (17.15)) is at least as high as the utility of lying on the right side of (17.15).
- Suppose now that $\Pr(A|\text{yes})$ increases. That reduces the utility of telling the truth. To preserve indifference between a truthful answer and a lie, $\Pr(A|\text{no})$ must increase to reduce the utility of lying.

17.4.2 Drawing a Truth Border

We can deduce two things about the truth border:

- The truth border divides the space of conditional probabilities into two subsets: "truth telling" and "lying". Thus, sufficient privacy elicits a truthful answer, whereas insufficient privacy results in a lie. The truth border depends on a respondent's utility function.
- Assumptions in (17.11) and (17.11) are sufficient only to guarantee a positive slope of the truth border. The truth border can have either a concave or a convex shape.

We can draw some truth borders with the following Python code:

```
x1 = np.arange(0, 1, 0.001)
y1 = x1 - 0.4
x2 = np.arange(0.4**2, 1, 0.001)
y2 = (pow(x2, 0.5) - 0.4)**2
x3 = np.arange(0.4**0.5, 1, 0.001)
y3 = pow(x3**2 - 0.4, 0.5)
plt.figure(figsize=(12, 10))
plt.plot(x1, y1, 'r-', label=r'Truth Border of: $U_i(Pr(A|r_i),\phi_i)=-Pr(A|r_i)+f(\
 ↪phi_i)$')
plt.fill_between(x1, 0, y1, facecolor='red', alpha=0.05)
plt.plot(x2, y2, 'b-', label=r'Truth Border of: $U_i(Pr(A|r_i),\phi_i)=-Pr(A|r_i)^{2}
 ↪+f(\phi_i)$')
plt.fill_between(x2, 0, y2, facecolor='blue', alpha=0.05)
plt.plot(x3, y3, 'y-', label=r'Truth Border of: $U_i(Pr(A|r_i),\phi_i)=-\sqrt{Pr(A|r_
 ↪i)}+f(\phi_i)$')
plt.fill_between(x3, 0, y3, facecolor='green', alpha=0.05)
plt.plot(x1, x1, ':', linewidth=2)
plt.xlim([0, 1])
plt.ylim([0, 1])

plt.xlabel('Pr(A|yes)')
plt.ylabel('Pr(A|no)')
plt.text(0.42, 0.3, "Truth Telling", fontdict={'size':28, 'style':'italic'})
plt.text(0.8, 0.1, "Lying", fontdict={'size':28, 'style':'italic'})

plt.legend(loc=0, fontsize='large')
plt.title('Figure 1.1')
plt.show()
```

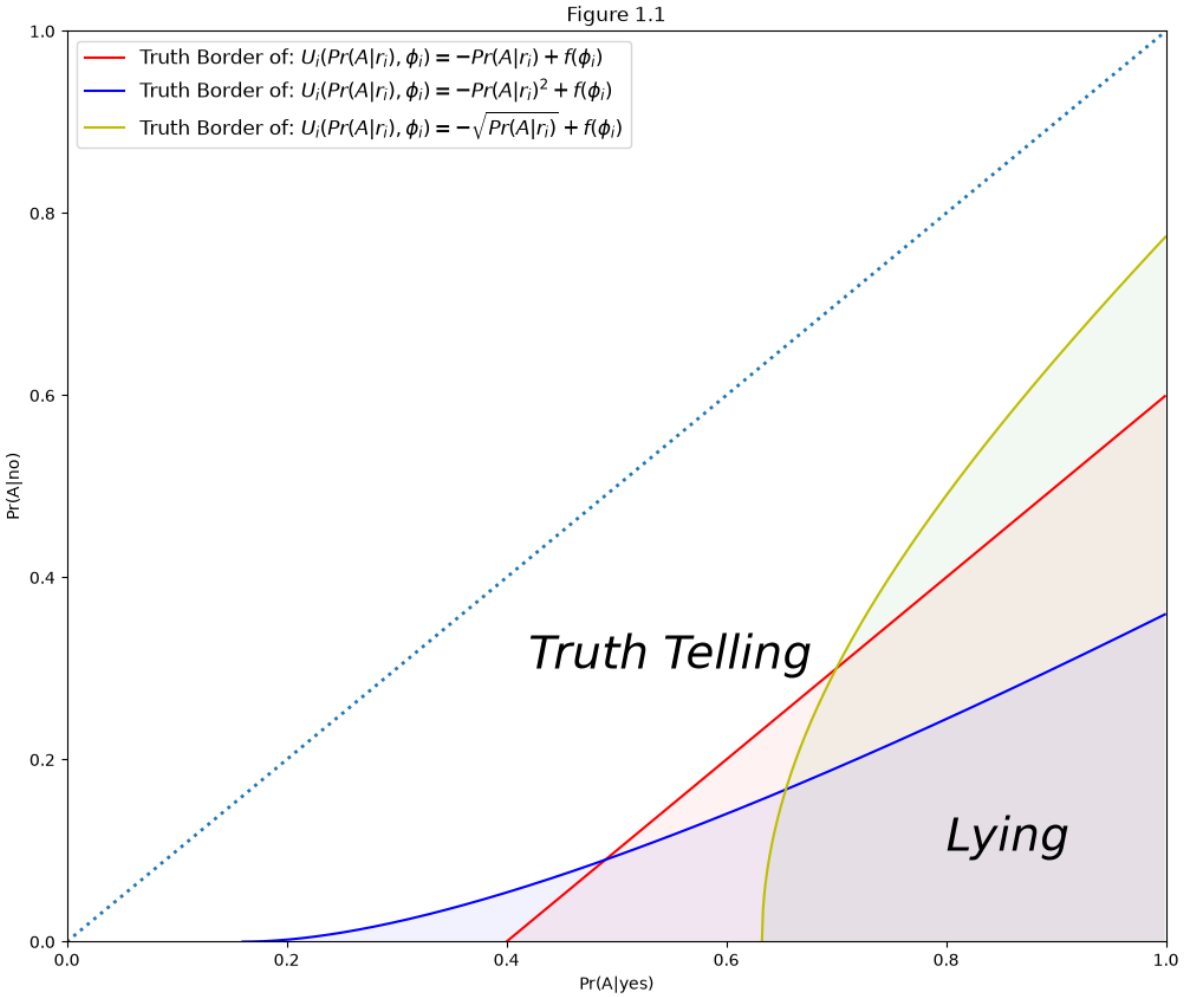


Figure 1.1 three types of truth border.

Without loss of generality, we consider the truth border:

$$U_i(\Pr(A|r_i), \phi_i) = -\Pr(A|r_i) + f(\phi_i)$$

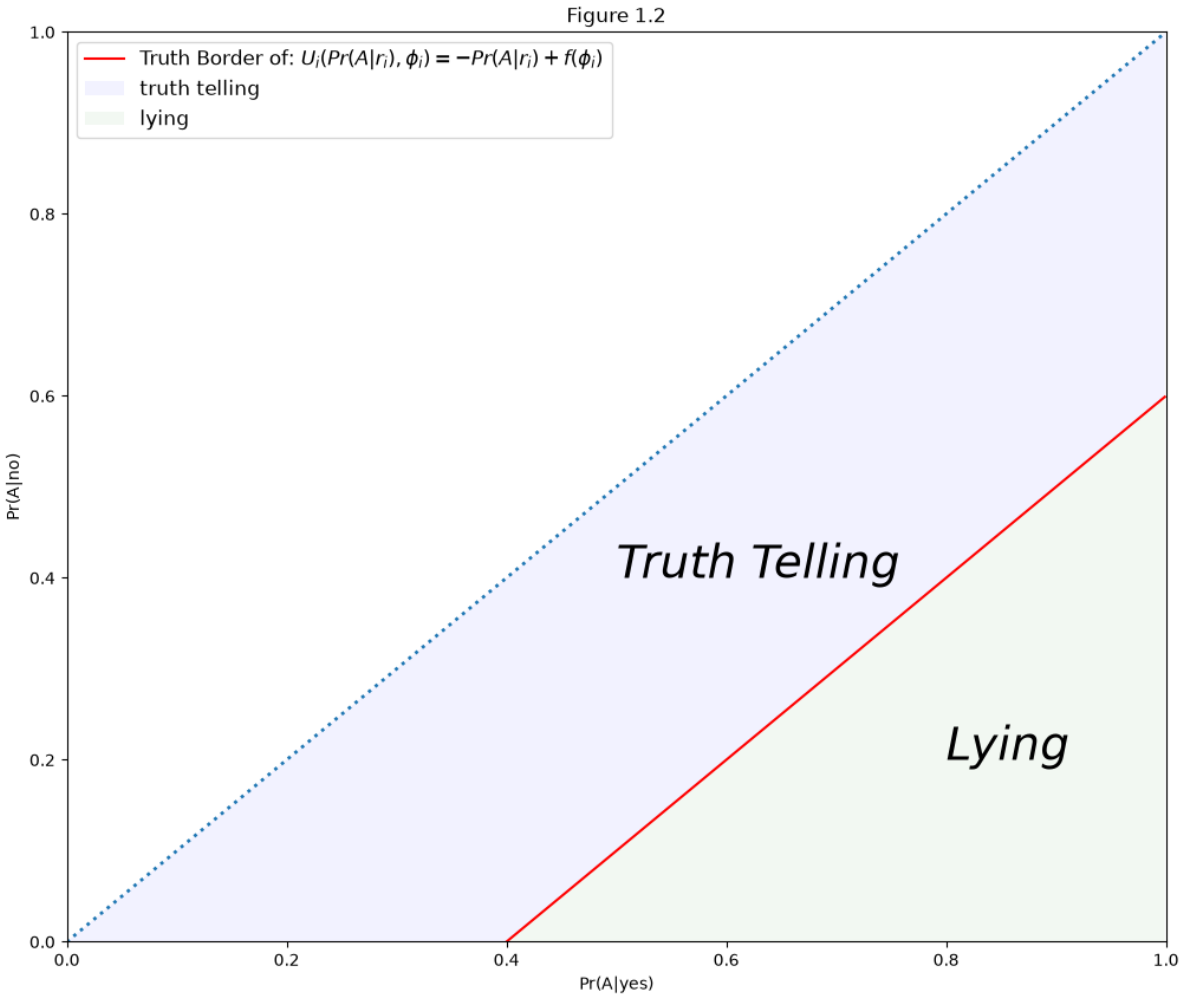
and plot the "truth telling" and "lying area" of individual i in Figure 1.2:

```
x1 = np.arange(0, 1, 0.001)
y1 = x1 - 0.4
z1 = x1
z2 = 0
plt.figure(figsize=(12, 10))
plt.plot(x1, y1,'r-',label=r'Truth Border of: $U_i(Pr(A|r_i),\phi_i)=-Pr(A|r_i)+f(\
↪phi_i)$')
plt.plot(x1, x1, ':', linewidth=2)
plt.fill_between(x1, y1, z1, facecolor='blue', alpha=0.05, label='truth telling')
plt.fill_between(x1, z2, y1, facecolor='green', alpha=0.05, label='lying')
plt.xlim([0, 1])
plt.ylim([0, 1])

plt.xlabel('Pr(A|yes)')
plt.ylabel('Pr(A|no)')
```

(continues on next page)

(continued from previous page)

```
plt.text(0.5, 0.4, "Truth Telling", fontdict={'size':28, 'style':'italic'})
plt.text(0.8, 0.2, "Lying", fontdict={'size':28, 'style':'italic'})

plt.legend(loc=0, fontsize='large')
plt.title('Figure 1.2')
plt.show()
```



17.5 Utilitarian View of Survey Design

17.5.1 Iso-variance Curves

A statistician's objective is

- to find a randomized response survey design that minimizes the bias and the variance of the estimator.

Given a design that ensures truthful answers by all respondents, Anderson(1976, Theorem 1) [Anderson, 1976] showed that the minimum variance estimate in the two-response model has variance

$$V(\Pr(A|\text{yes}), \Pr(A|\text{no})) = \frac{{\pi_A}^2(1-\pi_A)^2}{n} \times \frac{1}{\Pr(A|\text{yes}) - \pi_A} \times \frac{1}{\pi_A - \Pr(A|\text{no})} \tag{17.17}$$

where the random sample with replacement consists of n individuals.

We can use Expression (17.17) to draw iso-variance curves.

The following inequalities restrict the shapes of iso-variance curves:

$$\left.\frac{d\,\Pr(A|\text{no})}{d\,\Pr(A|\text{yes})}\right|_{\text{constant variance}} = \frac{\pi_A - \Pr(A|\text{no})}{\Pr(A|\text{yes}) - \pi_A} > 0 \tag{17.18}$$

$$\left.\frac{d^2\,\Pr(A|\text{no})}{d\,\Pr(A|\text{yes})^2}\right|_{\text{constant variance}} = -\frac{2\left[\pi_A - \Pr(A|\text{no})\right]}{\left[\Pr(A|\text{yes}) - \pi_A\right]^2} < 0 \tag{17.19}$$

From expression (17.17), (17.18) and (17.19) we can see that:

- Variance can be reduced only by increasing the distance of $\Pr(A|\text{yes})$ and/or $\Pr(A|\text{no})$ from r_A.
- Iso-variance curves are always upward-sloping and concave.

17.5.2 Drawing Iso-variance Curves

We use Python code to draw iso-variance curves.

The pairs of conditional probabilities can be attained using Warner's (1965) model.

Note that:

- Any point on the iso-variance curves can be attained with the unrelated question model as long as the statistician can completely control the model design.
- Warner's (1965) original randomized response model is less flexible than the unrelated question model.

```
class Iso_Variance:
    def __init__(self, pi, n):
        self.pi = pi
        self.n = n

    def plotting_iso_variance_curve(self):
        pi = self.pi
        n = self.n

        nv = np.array([0.27, 0.34, 0.49, 0.74, 0.92, 1.1, 1.47, 2.94, 14.7])
        x = np.arange(0, 1, 0.001)
        x0 = np.arange(pi, 1, 0.001)
        x2 = np.arange(0, pi, 0.001)
        y1 = [pi for i in x0]
        y2 = [pi for i in x2]
        y0 = 1 / (1 + (x0 * (1 - pi)**2) / ((1 - x0) * pi**2))

        plt.figure(figsize=(12, 10))
        plt.plot(x0, y0, 'm-', label='Warner')
        plt.plot(x, x, 'c:', linewidth=2)
        plt.plot(x0, y1,'c:', linewidth=2)
        plt.plot(y2, x2, 'c:', linewidth=2)
        for i in range(len(nv)):
            y = pi - (pi**2 * (1 - pi)**2) / (n * (nv[i] / n) * (x0 - pi + 1e-8))
            plt.plot(x0, y, 'k--', alpha=1 - 0.07 * i, label=f'V{i+1}')
        plt.xlim([0, 1])
        plt.ylim([0, 0.5])
        plt.xlabel('Pr(A|yes)')
```

(continues on next page)

(continued from previous page)

```
        plt.ylabel('Pr(A|no)')
        plt.legend(loc=0, fontsize='large')
        plt.text(0.32, 0.28, "High Var", fontdict={'size':15, 'style':'italic'})
        plt.text(0.91, 0.01, "Low Var", fontdict={'size':15, 'style':'italic'})
        plt.title('Figure 2')
        plt.show()
```

Properties of iso-variance curves are:

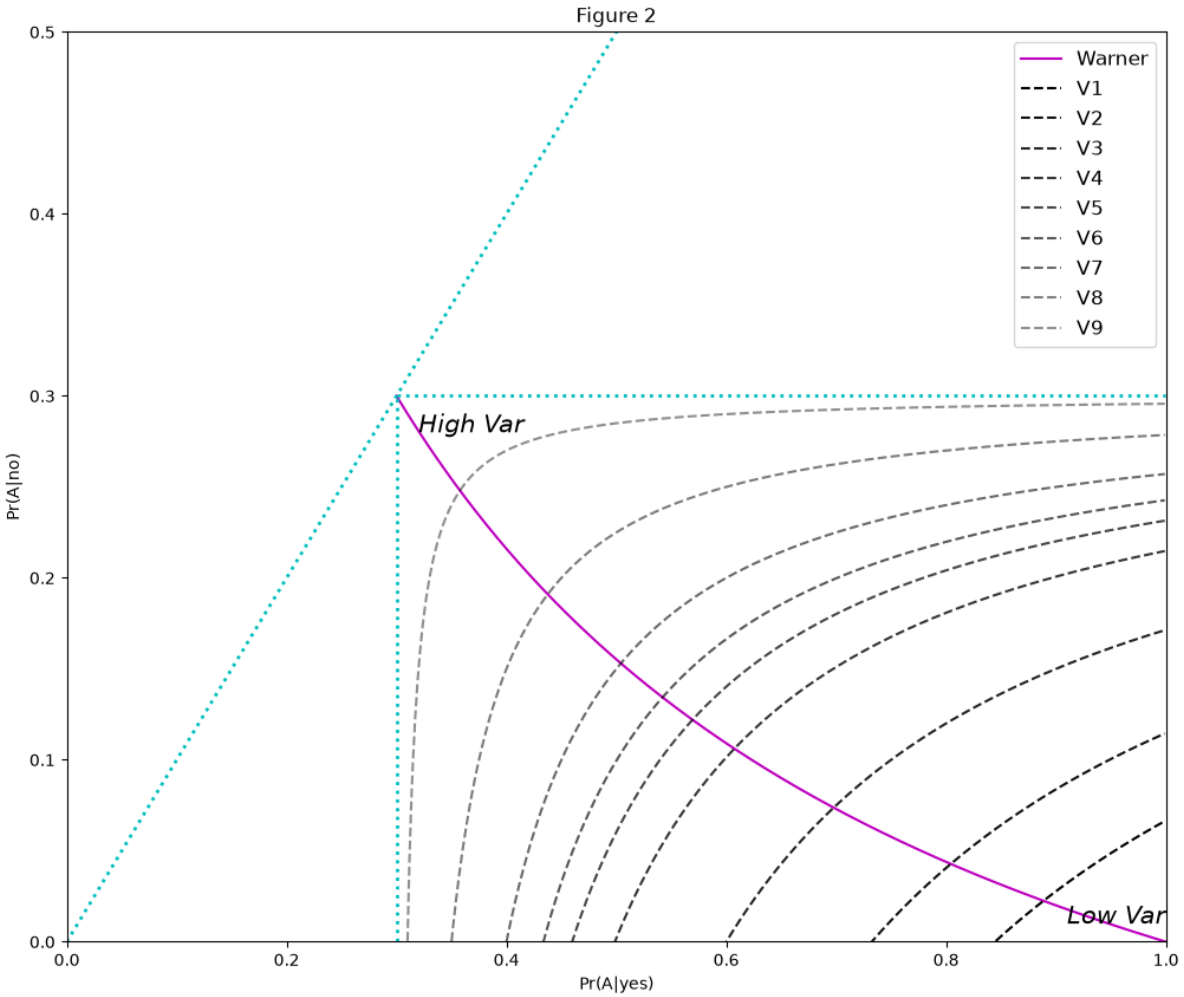
- All points on one iso-variance curve share the same variance
- From V_1 to V_9, the variance of the iso-variance curve increase monotonically, as colors brighten monotonically

Suppose the parameters of the iso-variance model follow those in Ljungqvist [Ljungqvist, 1993], which are:

- $\pi = 0.3$
- $n = 100$

Then we can plot the iso-variance curve in Figure 2:

```
var = Iso_Variance(pi=0.3, n=100)
var.plotting_iso_variance_curve()
```



17.5.3 Optimal Survey

A point on an iso-variance curves can be attained with the unrelated question design.

We now focus on finding an "optimal survey design" that

- Minimizes the variance of the estimator subject to privacy restrictions.

To obtain an optimal design, we first superimpose all individuals' truth borders on the iso-variance mapping.

To construct an optimal design

- The statistician should find the intersection of areas above all truth borders; that is, the set of conditional probabilities ensuring truthful answers from all respondents.
- The point where this set touches the lowest possible iso-variance curve determines an optimal survey design.

Consquently, a minimum variance unbiased estimator is pinned down by an individual who is the least willing to volunteer a truthful answer.

Here are some comments about the model design:

- An individual's decision of whether or not to answer truthfully depends on his or her belief about other respondents' behavior, because this determines the individual's calculation of $\Pr(A|\text{yes})$ and $\Pr(A|\text{no})$.
- An equilibrium of the optimal design model is a Nash equilibrium of a noncooperative game.
- Assumption (17.12) is sufficient to guarantee existence of an optimal model design. By choosing $\Pr(A|\text{yes})$ and $\Pr(A|\text{no})$ sufficiently close to each other, all respondents will find it optimal to answer truthfully. The closer are these probabilities, the higher the variance of the estimator becomes.
- If respondents experience a large enough increase in expected utility from telling the truth, then there is no need to use a randomized response model. The smallest possible variance of the estimate is then obtained at $\Pr(A|\text{yes}) = 1$ and $\Pr(A|\text{no}) = 0$; that is, when respondents answer truthfully to direct questioning.
- A more general design problem would be to minimize some weighted sum of the estimator's variance and bias. It would be optimal to accept some lies from the most "reluctant" respondents.

17.6 Criticisms of Proposed Privacy Measures

We can use a utilitarian approach to analyze some privacy measures.

We'll enlist Python Code to help us.

17.6.1 Analysis of Method of Lanke's (1976)

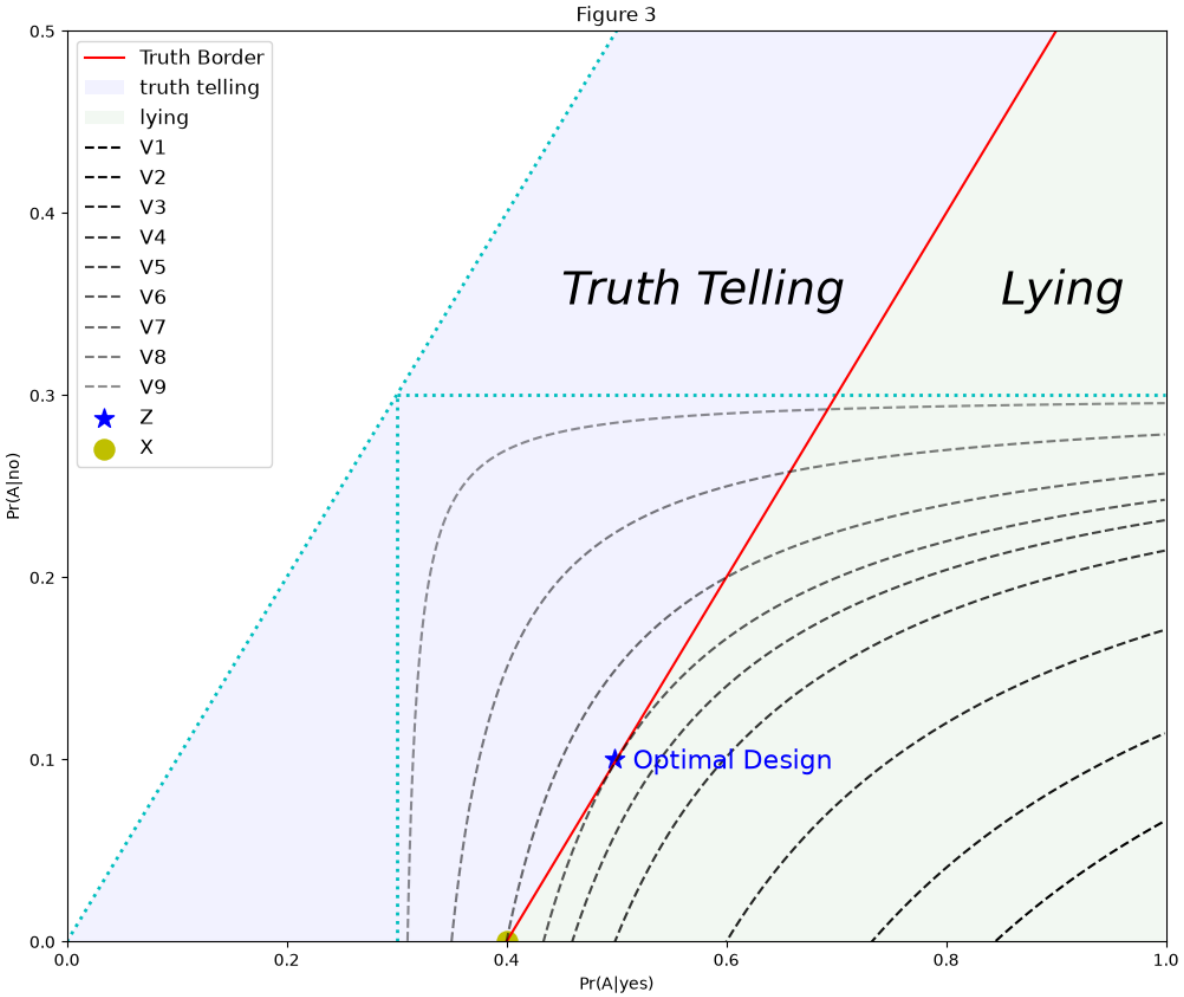
Lanke (1976) recommends a privacy protection criterion that minimizes:

$$\max\{\Pr(A|\text{yes}), \Pr(A|\text{no})\} \tag{17.20}$$

Following Lanke's suggestion, the statistician should find the highest possible $\Pr(A|\text{yes})$ consistent with truth telling while $\Pr(A|\text{no})$ is fixed at 0. The variance is then minimized at point X in Figure 3.

However, we can see that in Figure 3, point Z offers a smaller variance that still allows cooperation of the respondents, and it is achievable following our discussion of the truth border in Part III:

```
pi = 0.3
n = 100
nv = [0.27, 0.34, 0.49, 0.74, 0.92, 1.1, 1.47, 2.94, 14.7]
x = np.arange(0, 1, 0.001)
y = x - 0.4
z = x
x0 = np.arange(pi, 1, 0.001)
x2 = np.arange(0, pi, 0.001)
y1 = [pi for i in x0]
y2 = [pi for i in x2]

plt.figure(figsize=(12, 10))
plt.plot(x, x, 'c:', linewidth=2)
plt.plot(x0, y1, 'c:', linewidth=2)
plt.plot(y2, x2, 'c:', linewidth=2)
plt.plot(x, y, 'r-', label='Truth Border')
plt.fill_between(x, y, z, facecolor='blue', alpha=0.05, label='truth telling')
plt.fill_between(x, 0, y, facecolor='green', alpha=0.05, label='lying')
for i in range(len(nv)):
    y = pi - (pi**2 * (1 - pi)**2) / (n * (nv[i] / n) * (x0 - pi + 1e-8))
    plt.plot(x0, y, 'k--', alpha=1 - 0.07 * i, label=f'V{i+1}')


plt.scatter(0.498, 0.1, c='b', marker='*', label='Z', s=150)
plt.scatter(0.4, 0, c='y', label='X', s=150)
plt.xlim([0, 1])
plt.ylim([0, 0.5])
plt.xlabel('Pr(A|yes)')
plt.ylabel('Pr(A|no)')
plt.text(0.45, 0.35, "Truth Telling", fontdict={'size':28, 'style':'italic'})
plt.text(0.85, 0.35, "Lying",fontdict = {'size':28, 'style':'italic'})
plt.text(0.515, 0.095, "Optimal Design", fontdict={'size':16,'color':'b'})
plt.legend(loc=0, fontsize='large')
plt.title('Figure 3')
plt.show()
```



17.6.2 Method of Leysieffer and Warner (1976)

Leysieffer and Warner (1976) recommend a two-dimensional measure of jeopardy that reduces to a single dimension when there is no jeopardy in a 'no' answer", which means that

$$\Pr(\text{yes}|A) = 1$$

and

$$\Pr(A|\text{no}) = 0$$

This is not an optimal choice under a utilitarian approach.

17.6.3 Analysis on the Method of Chaudhuri and Mukerjee's (1988)

[Chaudhuri and Mukerjee, 1988]

Chaudhuri and Mukerjee (1988) argued that the individual may find that since "yes" may sometimes relate to the sensitive group A, a clever respondent may falsely but safely always be inclined to respond "no". In this situation, the truth border is such that individuals choose to lie whenever the truthful answer is "yes" and

$$\Pr(A|\text{no}) = 0$$

Here the gain from lying is too high for someone to volunteer a "yes" answer.

This means that

$$U_i\left(\Pr(A|\text{yes}), \text{truth}\right) < U_i\left(\Pr(A|\text{no}), \text{lie}\right)$$

in any situation always.

As a result, there is no attainable model design.

However, under a utilitarian approach there should exist other survey designs that are consistent with truthful answers.

In particular, respondents will choose to answer truthfully if the relative advantage from lying is eliminated.

We can use Python to show that the optimal model design corresponds to point Q in Figure 4:

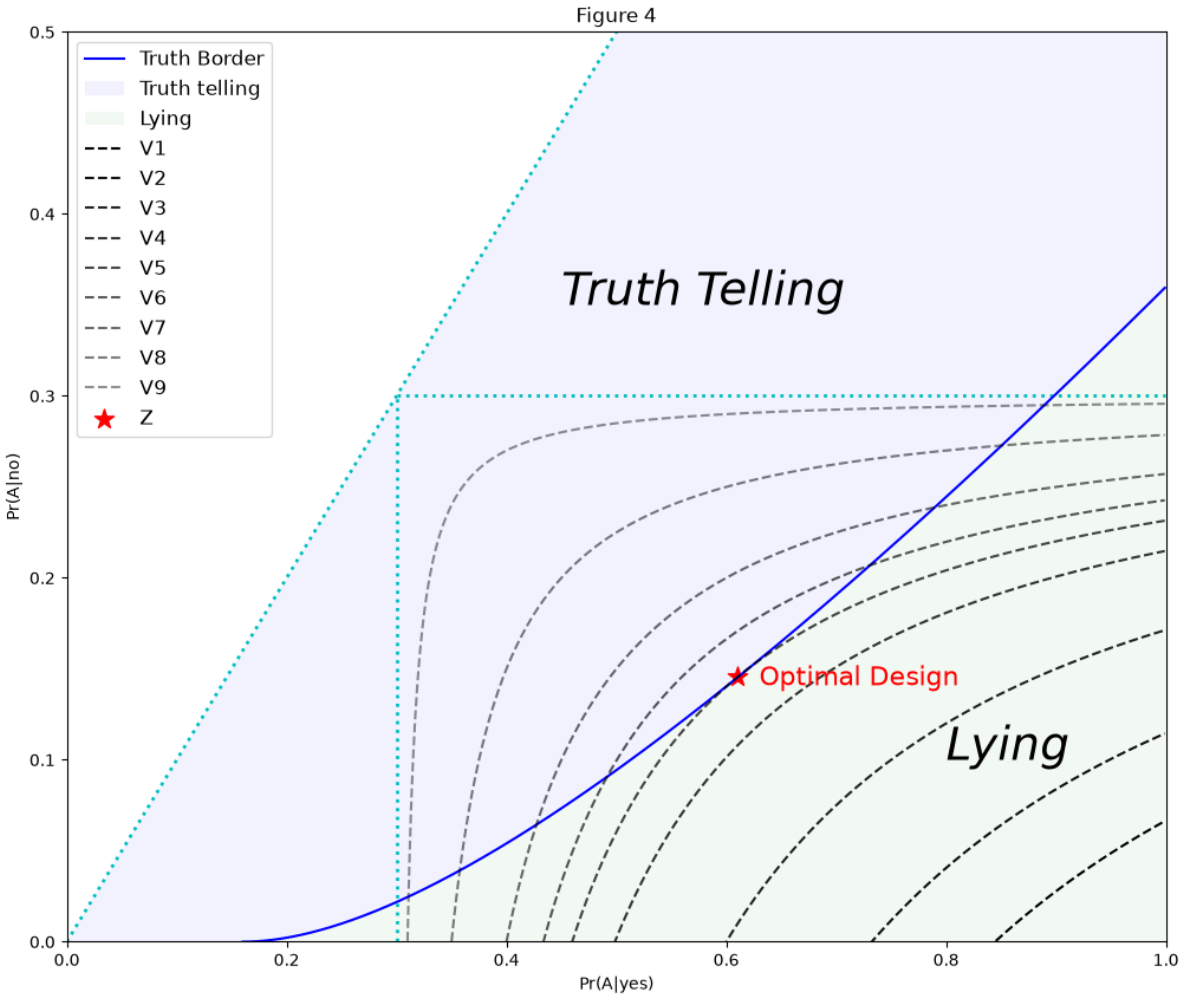
```
def f(x):
    if x < 0.16:
        return 0
    else:
        return (pow(x, 0.5) - 0.4)**2
```

```
pi = 0.3
n = 100
nv = [0.27, 0.34, 0.49, 0.74, 0.92, 1.1, 1.47, 2.94, 14.7]
x = np.arange(0, 1, 0.001)
y = [f(i) for i in x]
z = x
x0 = np.arange(pi, 1, 0.001)
x2 = np.arange(0, pi, 0.001)
y1 = [pi for i in x0]
y2 = [pi for i in x2]
x3 = np.arange(0.16, 1, 0.001)
y3 = (pow(x3, 0.5) - 0.4)**2

plt.figure(figsize=(12, 10))
plt.plot(x, x, 'c:', linewidth=2)
plt.plot(x0, y1,'c:', linewidth=2)
plt.plot(y2, x2,'c:', linewidth=2)
plt.plot(x3, y3,'b-', label='Truth Border')
plt.fill_between(x, y, z, facecolor='blue', alpha=0.05, label='Truth telling')
plt.fill_between(x3, 0, y3,facecolor='green', alpha=0.05, label='Lying')
for i in range(len(nv)):
    y = pi - (pi**2 * (1 - pi)**2) / (n * (nv[i] / n) * (x0 - pi + 1e-8))
    plt.plot(x0, y, 'k--', alpha=1 - 0.07 * i, label=f'V{i+1}')
plt.scatter(0.61, 0.146, c='r', marker='*', label='Z', s=150)
plt.xlim([0, 1])
plt.ylim([0, 0.5])
plt.xlabel('Pr(A|yes)')
```

(continues on next page)

(continued from previous page)

```
plt.ylabel('Pr(A|no)')
plt.text(0.45, 0.35, "Truth Telling", fontdict={'size':28, 'style':'italic'})
plt.text(0.8, 0.1, "Lying", fontdict={'size':28, 'style':'italic'})
plt.text(0.63, 0.141, "Optimal Design", fontdict={'size':16,'color':'r'})
plt.legend(loc=0, fontsize='large')
plt.title('Figure 4')
plt.show()
```



17.6.4 Method of Greenberg et al. (1977)

[Greenberg *et al.*, 1977]

Greenberg et al. (1977) defined the hazard for an individual in A as the probability that he or she is perceived as belonging to A:

$$\Pr(\text{yes}|A) \times \Pr(A|\text{yes}) + \Pr(\text{no}|A) \times \Pr(A|\text{no}) \tag{17.21}$$

The hazard for an individual who does not belong to A is

$$\Pr(\text{yes}|A^{'}) \times \Pr(A|\text{yes}) + \Pr(\text{no}|A^{'}) \times \Pr(A|\text{no}) \tag{17.22}$$

They also considered an alternative related measure of hazard that they said "is likely to be closer to the actual concern felt by a respondent."

Their "limited hazard" for an individual in A and $A^{'}$ is

$$\Pr(\text{yes}|A) \times \Pr(A|\text{yes}) \tag{17.23}$$

and

$$\Pr(\text{yes}|A^{'}) \times \Pr(A|\text{yes}) \tag{17.24}$$

According to Greenberg et al. (1977), a respondent commits himself or herself to answer truthfully on the basis of a probability in (17.21) or (17.23) **before** randomly selecting the question to be answered.

Suppose that the appropriate privacy measure is captured by the notion of "limited hazard" in (17.23) and (17.24).

Consider an unrelated question model where the unrelated question is replaced by the instruction "Say the word 'no'", which implies that

$$\Pr(A|\text{yes}) = 1$$

and it follows that:

- Hazard for an individual in $A^{'}$ is 0.
- Hazard for an individual in A can also be made arbitrarily small by choosing a sufficiently small $\Pr(\text{yes}|A)$.

Even though this hazard can be set arbitrarily close to 0, an individual in A will completely reveal his or her identity whenever truthfully answering the sensitive question.

However, under utilitarian framework, it is obviously contradictory.

If the individuals are willing to volunteer this information, it seems that the randomized response design was not necessary in the first place.

It ignores the fact that respondents retain the option of lying until they have seen the question to be answered.

17.7 Concluding Remarks

The justifications for a randomized response procedure are that

- Respondents are thought to feel discomfort from being perceived as belonging to the sensitive group.
- Respondents prefer to answer questions truthfully than to lie, unless it is too revealing.

If a privacy measure is not completely consistent with the rational behavior of the respondents, all efforts to derive an optimal model design are futile.

A utilitarian approach provides a systematic way to model respondents' behavior under the assumption that they maximize their expected utilities.

In a utilitarian analysis:

- A truth border divides the space of conditional probabilities of being perceived as belonging to the sensitive group, $\Pr(A|\text{yes})$ and $\Pr(A|\text{no})$, into the truth-telling region and the lying region.
- The optimal model design is obtained at the point where the truth border touches the lowest possible iso-variance curve.

A practical implication of the analysis of [Ljungqvist, 1993] is that uncertainty about respondents' demands for privacy can be acknowledged by **choosing $\Pr(A|\text{yes})$ and $\Pr(A|\text{no})$ sufficiently close to each other**.

Part III

Bayes Law

CHAPTER

EIGHTEEN

NON-CONJUGATE PRIORS

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install numpyro jax arviz
```

18.1 Overview

This lecture is a sequel to *Two Meanings of Probability*.

In that lecture we adopted a **beta** prior for the unknown probability θ of a coin landing heads, together with a **binomial** likelihood.

That prior and likelihood form a **conjugate pair**: applying Bayes' law returns a posterior of the *same* family as the prior — again a beta distribution.

Conjugacy is convenient because it delivers a posterior in closed form.

But a person's prior beliefs are their own business, and in general they will not happen to be conjugate to the likelihood.

When the prior and likelihood are **not** conjugate, the posterior usually has no closed form, and we must approximate it numerically.

This lecture introduces two widely used ways to do that, both implemented in the probabilistic programming library NumPyro:

- **Markov chain Monte Carlo (MCMC)** — construct a Markov chain whose stationary distribution is the posterior, then sample from it. We use the **No-U-Turn Sampler (NUTS)**, a state-of-the-art form of Hamiltonian Monte Carlo.
- **Variational inference (VI)** — replace sampling with optimization: search within a tractable family of distributions for the member closest to the posterior.

> **Note**
>
> We treat NUTS as a black box in this lecture.
>
> In brief, it is a form of **Hamiltonian Monte Carlo**, which is itself a version of the **Metropolis–Hastings** algorithm: it proposes candidate draws and accepts or rejects them so that the resulting Markov chain has the posterior as its stationary distribution.
>
> What distinguishes it from a basic Metropolis–Hastings sampler is that its proposals are built from *gradient* (derivative) information about the log-posterior, which lets the chain move efficiently through the parameter space; NUTS additionally tunes the length of each proposed move automatically.
>
> For a more advanced introduction to MCMC and the Metropolis–Hastings algorithm, see this lecture.

Our plan is:

1. Confirm that MCMC reproduces the *conjugate* beta posterior that we can compute analytically — this validates the machinery on a problem whose answer we already know.
2. Replace the beta prior with several **non-conjugate** priors and approximate each posterior with MCMC.
3. Introduce variational inference and compare it with MCMC.

Let us start with some imports.

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.stats as st

import jax.numpy as jnp
from jax import random

import numpyro
import numpyro.distributions as dist
from numpyro.infer import MCMC, NUTS, SVI, Trace_ELBO
from numpyro.infer.autoguide import AutoNormal
from numpyro.optim import Adam

import arviz as az
```

18.2 The coin-flipping model

As in *Two Meanings of Probability*, a coin lands heads ($Y = 1$) with probability θ and tails ($Y = 0$) with probability $1 - \theta$.

If we flip the coin n times, the number of heads k has the **binomial** distribution

$$p(k \mid \theta) = \binom{n}{k} \theta^k (1-\theta)^{n-k}.$$

We treat θ as a random variable with a prior density $p(\theta)$, and we want the posterior

$$p(\theta \mid k) \propto p(k \mid \theta)\, p(\theta).$$

18.2.1 Generating data

We simulate a sequence of coin flips from a coin whose true (but unknown to the analyst) probability of heads is $\theta = 0.4$.

```
def simulate_coin_flips(θ=0.4, n=20, seed=1234):
    "Flip a coin n times; return an array of 0s (tails) and 1s (heads)."
    rng = np.random.default_rng(seed)
    return (rng.random(n) < θ).astype(int)

data = simulate_coin_flips()
k, n = int(data.sum()), len(data)
k, n
```

```
(9, 20)
```

We deliberately use a **small** sample ($n = 20$).

The reason is that the prior matters most when data are scarce.

With a large sample the likelihood dominates and almost any reasonable prior leads to the same posterior — exactly the concentration we saw in *Two Meanings of Probability*.

A modest n keeps the influence of the prior visible, which is what we want to study here.

18.2.2 Specifying the model in NumPyro

For most readers this will be a first encounter with NumPyro, whose style takes some getting used to.

To use it we describe our probability model as a Python function — which, a little confusingly, NumPyro calls a **model**.

Such a function does not *compute* anything when called, and it does not return the posterior.

Instead it is a *declaration* of the generative story for the data: which quantities are random, how they are distributed, and how the data depend on them.

An inference algorithm — such as the NUTS sampler below — then *reads* this declaration and works out the posterior for us.

Inside a model, every random quantity is introduced by a call to `numpyro.sample`, and the keyword `obs` decides its role:

- `numpyro.sample("θ", prior)` introduces a **latent** (unobserved) variable named `"θ"`, drawn from `prior` — a quantity we wish to infer.
- `numpyro.sample("k", dist.Binomial(n, θ), obs=k)` introduces an **observed** variable: the keyword `obs=k` pins it to the data, which is how the likelihood $p(k \mid \theta)$ enters.

The string names (`"θ"` and `"k"`) are the labels NumPyro uses to keep track of the variables; we will use them later to pull the posterior draws back out.

We write a *single* model that takes the prior distribution as an argument, so we can reuse it unchanged for every prior we consider — conjugate or not.

```
def binomial_model(prior, k, n):
    "Binomial likelihood with a caller-supplied prior on θ."
    θ = numpyro.sample("θ", prior)
    numpyro.sample("k", dist.Binomial(n, θ), obs=k)
```

Notice that `binomial_model` returns nothing, and that we never call it ourselves.

Instead we hand it to an inference algorithm, which supplies the arguments and traces the two `sample` statements to assemble the posterior.

We also write a small helper that runs NUTS on a given model and returns the fitted sampler.

We request four chains so that we can check convergence below, and run them with `chain_method="vectorized"`, which evaluates all chains together on a single device — so the same code runs unchanged on a CPU or a GPU.

```
def run_nuts(model, *args, seed=0, num_warmup=1000, num_samples=4000, num_chains=4):
    "Sample a NumPyro model with the NUTS sampler."
    mcmc = MCMC(
        NUTS(model),
        num_warmup=num_warmup,
        num_samples=num_samples,
        num_chains=num_chains,
        chain_method="vectorized",
        progress_bar=False,
    )
    mcmc.run(random.key(seed), *args)
    return mcmc
```

NumPyro is built on JAX, which treats randomness explicitly: rather than relying on a global random state, each run needs its own **PRNG key**, created here with `random.key(seed)`.

(This is why we used NumPy's generator to make the data above but JAX keys here.)

`run_nuts` is deliberately generic: it samples whatever model we pass and forwards the extra arguments (`*args`) on to that model through `mcmc.run`. We always call it as `run_nuts(binomial_model, prior, k, n)`, so `prior`, `k`, and `n` reach `binomial_model` unchanged — there is only ever the one prior.

18.3 MCMC reproduces the conjugate posterior

Before trusting MCMC on hard problems, let us check it on an easy one.

With a Beta(α_0, β_0) prior the posterior is known analytically (see *Two Meanings of Probability*):

$$\theta \mid k \sim \text{Beta}(\alpha_0 + k,\ \beta_0 + n - k).$$

We take $\alpha_0 = \beta_0 = 2$ and sample the posterior with NUTS.

```
α0, β0 = 2.0, 2.0
mcmc = run_nuts(binomial_model, dist.Beta(α0, β0), k, n)
```

Before looking at the posterior we should check that the sampler has done its job.

Unlike the independent draws we are used to, MCMC returns a *dependent* sequence — a Markov chain — whose early draws still remember where the chain started.

We can trust the output only once the chain has "forgotten" its starting point and settled into its stationary distribution, which by construction is the posterior we want.

As a safeguard we ran **four** chains from different random starting points (`num_chains=4` in `run_nuts`) and now check that they agree with one another.

ArviZ is a companion library for examining the output of Bayesian samplers.

The function `az.from_numpyro` repackages our NumPyro results into ArviZ's standard data structure, and `az.summary` prints a table of per-parameter summaries and convergence diagnostics.

```
idata = az.from_numpyro(mcmc)
az.summary(idata, var_names=["θ"])
```

```
      mean    sd eti89_lb eti89_ub  ess_bulk  ess_tail r_hat mcse_mean  mcse_sd
θ  0.461  0.1       0.3      0.62      5861      7337  1.00     0.0013  0.00088
```

Two columns of this table are convergence diagnostics worth understanding.

- **`r_hat`** (the Gelman–Rubin statistic) compares the spread of the draws *within* each chain to the spread *between* chains. If the chains have all converged to the same distribution these two match and `r_hat` is close to 1.0; values above roughly 1.01 warn that the chains disagree and the draws cannot yet be trusted.
- **`ess_bulk`** and **`ess_tail`** report the *effective sample size*. Because consecutive MCMC draws are correlated, a chain of length N carries less information than N independent draws would; the effective sample size estimates how many independent draws it is worth (in the bulk and in the tails of the distribution respectively). Larger is better.

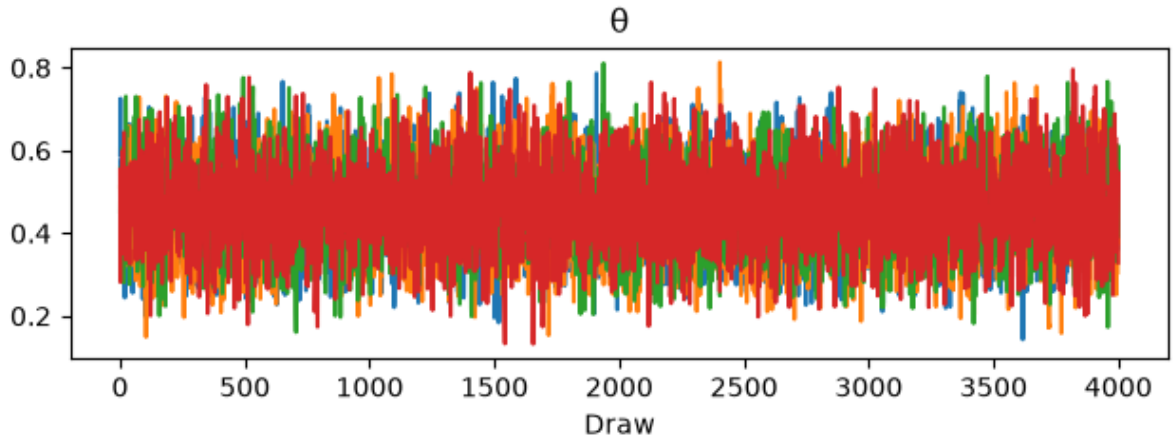
Here `r_hat` is essentially 1.0 and the effective sample sizes run into the thousands, so the chains have mixed well.

A **trace plot** gives a visual check of the same thing.

ArviZ's `plot_trace` draws two panels for each parameter: on the right, the sampled value against the iteration number (one coloured line per chain); on the left, a density estimate of the draws from each chain.

Well-mixed chains look like stationary noise on the right — a fuzzy, flat band, with the chains overlapping rather than drifting or wandering — and their densities on the left lie almost on top of one another.

```
az.plot_trace(idata, var_names=["θ"])
plt.tight_layout()
plt.show()
```



Our chains pass both checks, so we can trust the draws and turn to the posterior itself.

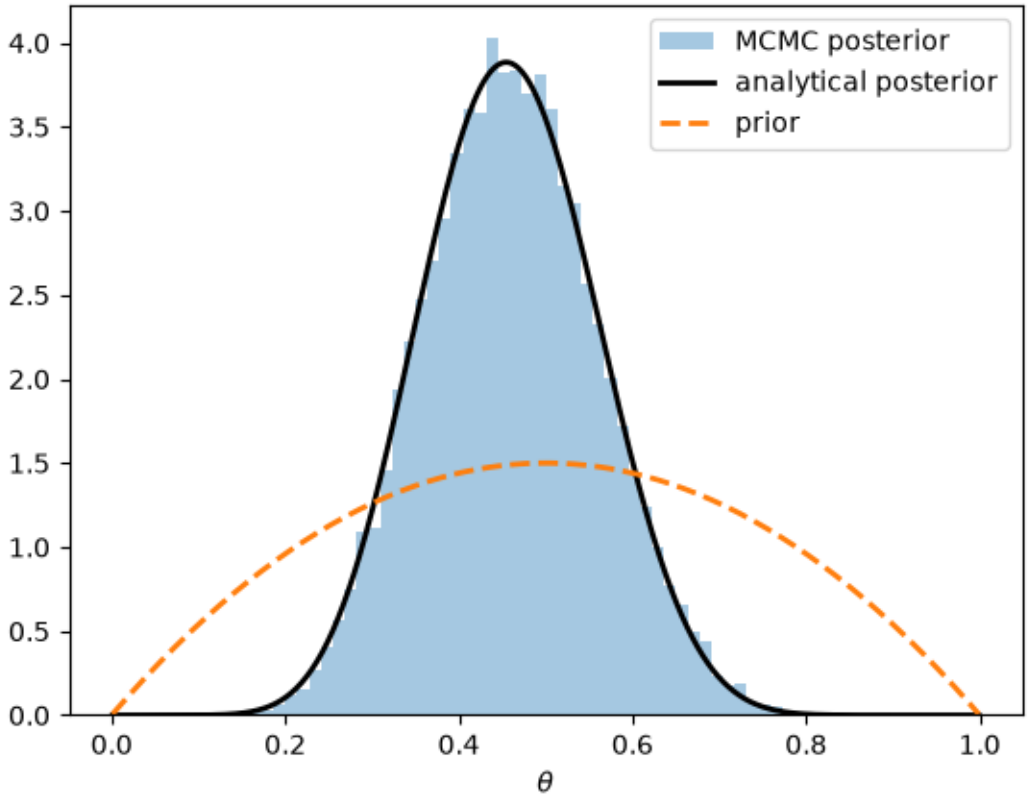
Now we compare the MCMC posterior with the analytical beta posterior.

```
θ_grid = np.linspace(0.001, 0.999, 500)
samples = np.asarray(mcmc.get_samples()["θ"])

fig, ax = plt.subplots()
ax.hist(samples, bins=50, density=True, alpha=0.4,
        label="MCMC posterior")
ax.plot(θ_grid, st.beta(α0 + k, β0 + n - k).pdf(θ_grid),
```

(continues on next page)

(continued from previous page)

```
        'k-', lw=2, label="analytical posterior")
ax.plot(θ_grid, st.beta(α0, β0).pdf(θ_grid),
        'C1--', lw=2, label="prior")
ax.set_xlabel(r"$\theta$")
ax.legend()
plt.show()
```



The histogram of MCMC draws sits right on top of the analytical posterior density.

The sampler works, so we can rely on it for priors that have no closed-form posterior.

18.4 Non-conjugate priors

We now keep the binomial likelihood and the same data, but replace the beta prior with priors that are **not** conjugate to it.

For each prior the recipe is identical:

1. describe the prior and build it as a NumPyro distribution,
2. pass it to `binomial_model` and run NUTS,
3. plot the prior against the resulting posterior.

The following helper draws a prior density and the posterior samples on the same axes.

```
def plot_prior_posterior(prior, samples, title=""):
    "Overlay a prior density and posterior MCMC draws for θ on [0, 1]."
    grid = jnp.linspace(0.001, 0.999, 500)
    # mask the density to the prior's support: dist.Uniform.log_prob
    # returns its constant value even outside [low, high]
    in_support = np.asarray(prior.support(grid))
    prior_pdf = np.where(in_support, np.exp(np.asarray(prior.log_prob(grid))), 0.0)

    fig, ax = plt.subplots()
    ax.hist(np.asarray(samples), bins=50, density=True, alpha=0.4,
            label="posterior (MCMC)")
    ax.plot(np.asarray(grid), prior_pdf, 'C1--', lw=2, label="prior")
    ax.set_xlabel(r"$\theta$")
    ax.set_xlim(0, 1)
    ax.legend()
    if title:
        ax.set_title(title)
    plt.show()
```

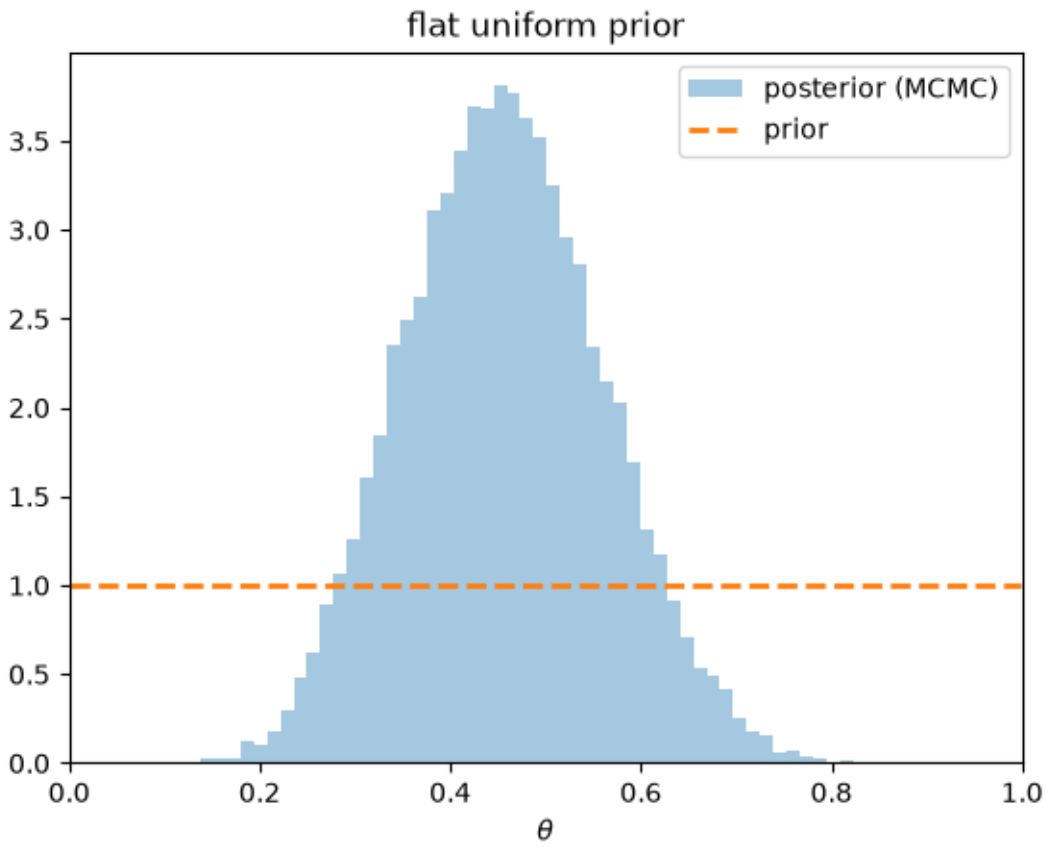
18.4.1 A uniform prior

The simplest non-conjugate prior is **uniform**: the analyst regards every value of θ in some interval as equally likely.

A uniform prior on all of $[0, 1]$ expresses indifference.

Because its density is constant, the posterior is then proportional to the likelihood alone.

```
mcmc_flat = run_nuts(binomial_model, dist.Uniform(0.0, 1.0), k, n)
plot_prior_posterior(dist.Uniform(0.0, 1.0),
                     mcmc_flat.get_samples()["θ"],
                     title="flat uniform prior")
```

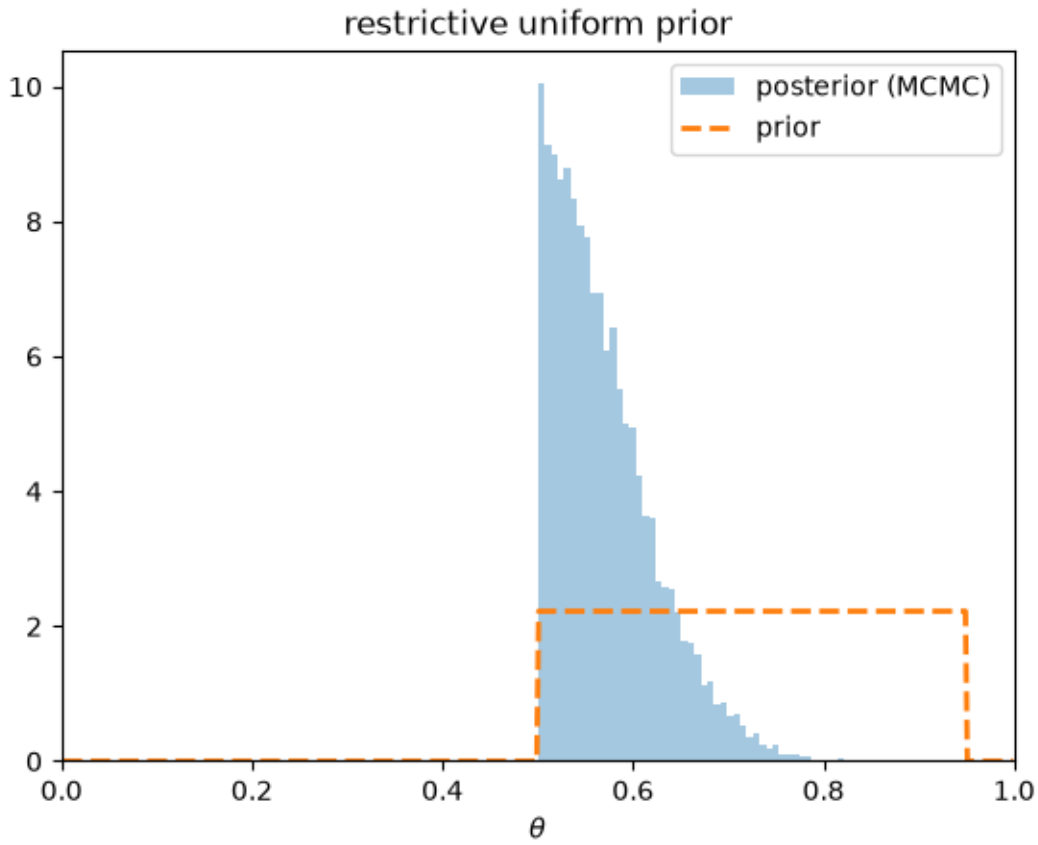


The posterior is centered near the sample frequency k/n, just as the likelihood is.

Now suppose instead that the analyst is convinced the coin favors heads, and places a uniform prior on $[0.5, 0.95]$.

This prior assigns *zero* density to the region around the true value $\theta = 0.4$.

```
mcmc_restr = run_nuts(binomial_model, dist.Uniform(0.5, 0.95), k, n)
plot_prior_posterior(dist.Uniform(0.5, 0.95),
                     mcmc_restr.get_samples()["θ"],
                     title="restrictive uniform prior")
```



The posterior cannot put mass where the prior is zero, so it piles up against the lower boundary 0.5 — as close to the data as the prior permits.

This is a vivid warning: a prior that rules out the truth can never be overturned by data, no matter how much we collect.

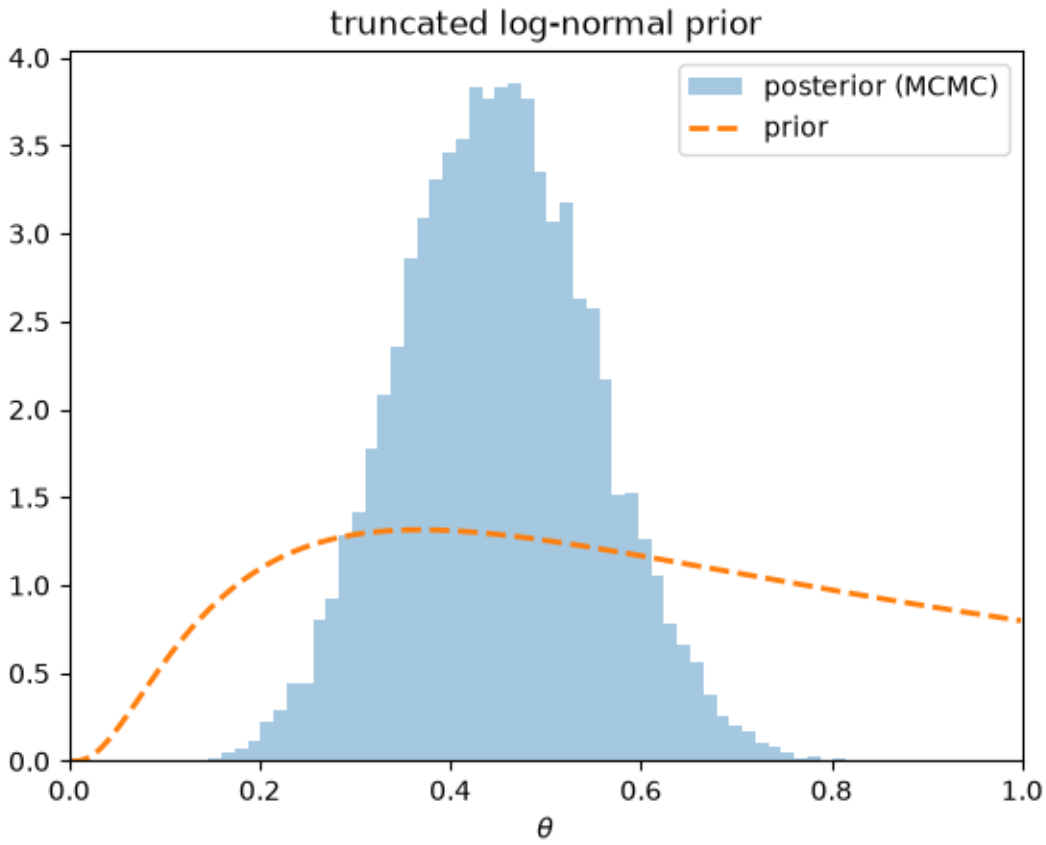
18.4.2 A truncated log-normal prior

A uniform prior is flat. A more realistic prior is smooth and asymmetric.

A convenient choice on $[0, 1]$ is a **truncated log-normal**: take $Z \sim N(\mu, \sigma)$ truncated to $Z \leq 0$, and set $\theta = e^Z$, which then lies in $(0, 1]$.

NumPyro builds this by feeding a `TruncatedNormal` through an `ExpTransform`.

```
def truncated_lognormal(μ, σ):
    "Log-normal distribution truncated to the unit interval (0, 1]."
    base = dist.TruncatedNormal(loc=μ, scale=σ, low=-jnp.inf, high=0.0)
    return dist.TransformedDistribution(base, dist.transforms.ExpTransform())

prior_ln = truncated_lognormal(0.0, 1.0)
mcmc_ln = run_nuts(binomial_model, prior_ln, k, n)
plot_prior_posterior(prior_ln, mcmc_ln.get_samples()["θ"],
                     title="truncated log-normal prior")
```



The prior favors smaller values of θ, but with $\sigma = 1$ it is diffuse, so the likelihood pulls the posterior toward the sample frequency.

We keep `mcmc_ln` — we will compare it with variational inference below.

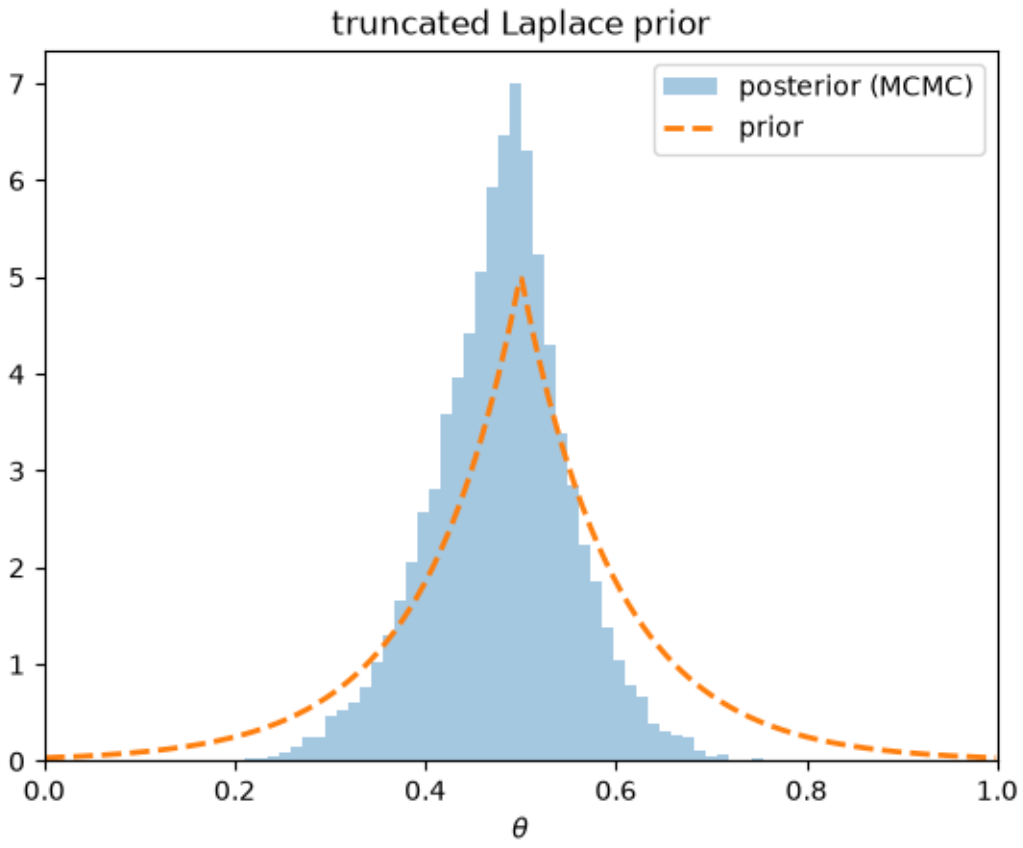
18.4.3 A truncated Laplace prior

Our final prior has a sharp, non-smooth peak.

A **Laplace** density $\propto e^{-|\theta-\mu|/b}$ has a kink at its center μ, expressing a strong belief that θ sits near μ while still allowing for surprises in the tails.

We truncate it to $[0, 1]$ and center it at 0.5.

```
def truncated_laplace(μ, b):
    "Laplace distribution truncated to the unit interval [0, 1]."
    return dist.TruncatedDistribution(dist.Laplace(μ, b), low=0.0, high=1.0)

prior_lp = truncated_laplace(0.5, 0.1)
mcmc_lp = run_nuts(binomial_model, prior_lp, k, n)
plot_prior_posterior(prior_lp, mcmc_lp.get_samples()["θ"],
                     title="truncated Laplace prior")
```



The spiked prior tugs the posterior toward 0.5, away from the sample frequency near 0.4.

The pull is gentle here because the prior, though peaked, is not very tight; with a smaller b it would dominate the modest sample.

NUTS handles the kink in the prior without any special tuning — a practical advantage of gradient-based samplers paired with automatic differentiation.

18.5 Variational inference

MCMC approximates the posterior by *sampling* from it.

Variational inference (VI) takes a different route: it turns posterior approximation into an *optimization* problem.

We restrict attention to a tractable family of densities $q_\phi(\theta)$ — the **guide** — indexed by parameters ϕ, and we search for the member of that family closest to the posterior.

18.5.1 Why variational inference?

If NUTS already returns accurate posteriors, why introduce another method?

The answer is **scale**.

MCMC evaluates the likelihood over the entire dataset at every step, and the number of steps it needs tends to grow with the dimension of the parameter.

For large datasets or high-dimensional models — for instance the hierarchical models and neural networks common in machine learning — this can become too slow to be practical.

Variational inference scales much better, because the objective (the ELBO, introduced below) can be maximized with *stochastic* gradients computed on small random subsets of the data — the same machinery that trains deep learning models.

It also yields a compact parametric approximation that is cheap to store and to draw from afterwards.

The price is accuracy: VI returns only the best fit *within the guide family*, and it can understate uncertainty.

As a rule of thumb, prefer MCMC when you need an accurate posterior and the problem is small enough to afford it, and VI when the model is too large for MCMC or a fast, approximate answer is good enough.

18.5.2 The evidence lower bound

Let the prior be $p(\theta)$ and the likelihood be $p(Y \mid \theta)$, where Y denotes the observed data (here the head count k).

By Bayes' rule,

$$p(\theta \mid Y) = \frac{p(Y, \theta)}{p(Y)} = \frac{p(Y \mid \theta)\, p(\theta)}{p(Y)},$$

where

$$p(Y) = \int p(Y \mid \theta)\, p(\theta)\, d\theta. \tag{18.1}$$

The integral in (18.1) is the troublesome one: in the non-conjugate case it has no closed form.

We measure the discrepancy between the guide $q_\phi(\theta)$ and the posterior with the **Kullback–Leibler (KL) divergence**

$$D_{KL}(q_\phi(\theta) \,\|\, p(\theta \mid Y)) = -\int q_\phi(\theta)\, \log \frac{p(\theta \mid Y)}{q_\phi(\theta)}\, d\theta,$$

and we choose ϕ to minimize it.

The KL divergence still involves the intractable posterior, but we can rearrange it. Using $p(\theta \mid Y) = p(\theta, Y)/p(Y)$,

$$\begin{aligned} D_{KL}(q_\phi \,\|\, p(\theta \mid Y)) &= -\int q_\phi(\theta)\, \log \frac{p(\theta, Y)/p(Y)}{q_\phi(\theta)}\, d\theta \\ &= -\int q_\phi(\theta) \left[\log \frac{p(\theta, Y)}{q_\phi(\theta)} - \log p(Y)\right] d\theta \\ &= -\int q_\phi(\theta)\, \log \frac{p(\theta, Y)}{q_\phi(\theta)}\, d\theta + \log p(Y), \end{aligned}$$

where the last line uses $\int q_\phi(\theta)\, d\theta = 1$. Rearranging,

$$\log p(Y) = D_{KL}(q_\phi \,\|\, p(\theta \mid Y)) + \underbrace{\int q_\phi(\theta)\, \log \frac{p(\theta, Y)}{q_\phi(\theta)}\, d\theta}_{\text{ELBO}}.$$

The marginal likelihood $\log p(Y)$ on the left does not depend on ϕ.

Hence **minimizing** the KL divergence is equivalent to **maximizing** the second term, the **evidence lower bound (ELBO)**:

$$\text{ELBO}(\phi) \equiv \int q_\phi(\theta)\, \log \frac{p(\theta, Y)}{q_\phi(\theta)}\, d\theta = \mathbb{E}_{q_\phi(\theta)}\big[\log p(\theta, Y) - \log q_\phi(\theta)\big]. \tag{18.2}$$

Because $D_{KL} \geq 0$, the ELBO is a lower bound on $\log p(Y)$ — hence its name.

Crucially, (18.2) involves only the *joint* density $p(\theta, Y) = p(Y \mid \theta)\, p(\theta)$, which we can evaluate, not the intractable normalizing constant $p(Y)$.

The expectation can be estimated by sampling from q_ϕ, and ϕ improved by gradient ascent — this is **stochastic variational inference (SVI)**.

18.5.3 Implementing SVI in NumPyro

We need a guide q_ϕ.

The simplest choice is an **autoguide**: NumPyro inspects the model and automatically constructs a guide for us.

`AutoNormal` places an independent normal distribution on each latent variable, transformed to respect its support — here, to keep θ inside $(0, 1)$.

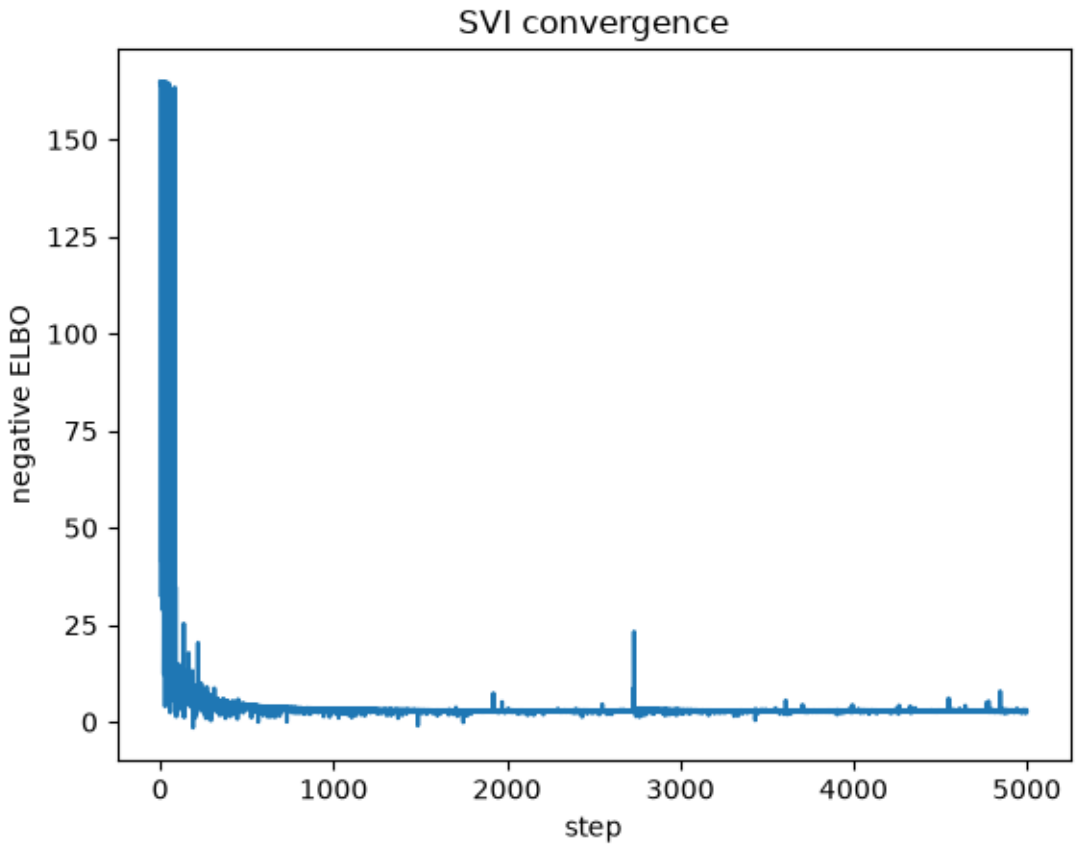
We apply SVI to the truncated log-normal model from above and maximize the ELBO with the Adam optimizer.

```
guide = AutoNormal(binomial_model)
optimizer = Adam(step_size=0.01)
svi = SVI(binomial_model, guide, optimizer, loss=Trace_ELBO())

svi_result = svi.run(random.key(0), 5000, prior_ln, k, n, progress_bar=False)
```

SVI maximizes the ELBO; equivalently, it minimizes its negative, which is the reported loss.

A loss curve that flattens out indicates convergence.

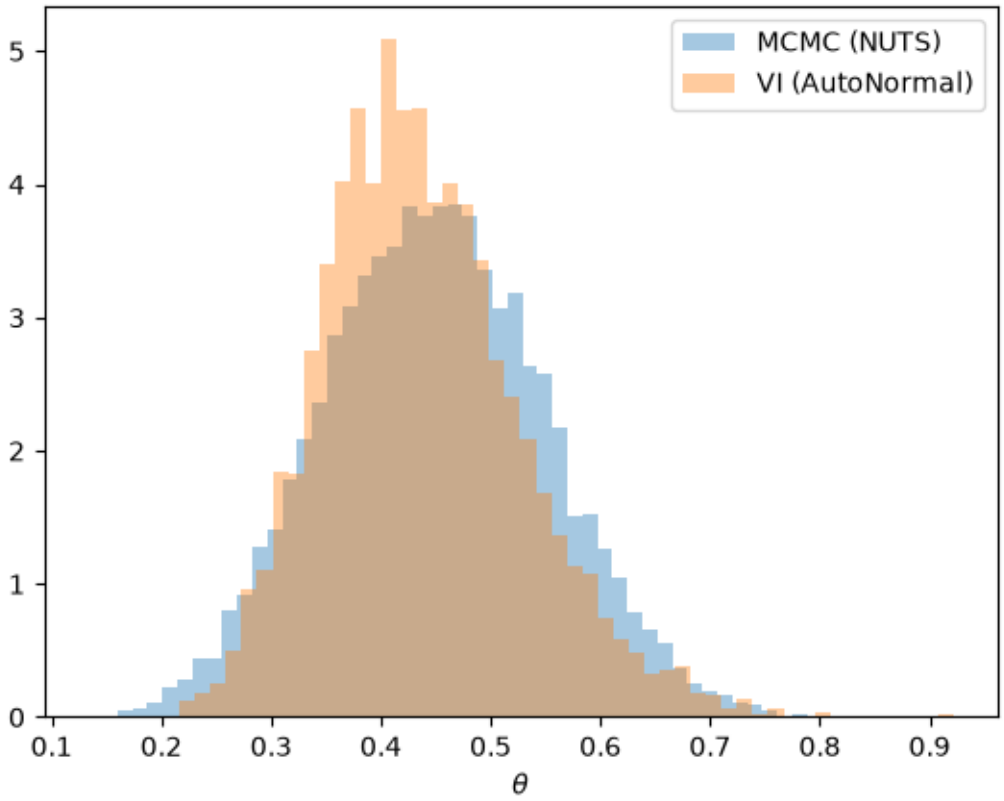
```
fig, ax = plt.subplots()
ax.plot(svi_result.losses)
ax.set_xlabel("step")
ax.set_ylabel("negative ELBO")
ax.set_title("SVI convergence")
plt.show()
```



18.5.4 Comparing VI with MCMC

To assess the approximation, we draw samples from the fitted guide and compare them with the NUTS posterior for the same (log-normal-prior) model.

```
vi_samples = guide.sample_posterior(
    random.key(1), svi_result.params, sample_shape=(4000,)
)["θ"]
nuts_samples = mcmc_ln.get_samples()["θ"]

fig, ax = plt.subplots()
ax.hist(np.asarray(nuts_samples), bins=50, density=True, alpha=0.4,
        label="MCMC (NUTS)")
ax.hist(np.asarray(vi_samples), bins=50, density=True, alpha=0.4,
        label="VI (AutoNormal)")
ax.set_xlabel(r"$\theta$")
ax.legend()
plt.show()
```



The two approximations broadly agree on the location and spread of the posterior.

They need not agree perfectly.

MCMC samples the true posterior (up to Monte Carlo error), whereas VI reports the best fit *within its guide family*.

A mean-field normal guide is symmetric on the transformed scale and can miss skewness or heavy tails in the true posterior.

The trade-off is one of cost against fidelity: VI replaces sampling with optimization and is often much faster in high dimensions, but it delivers an approximation whose quality is capped by the flexibility of the guide.

18.6 Where to next

This lecture showed how to compute posteriors when prior and likelihood are not conjugate, using NUTS and stochastic variational inference in NumPyro.

The same tools carry over to richer models.

The lectures *Posterior Distributions for AR(1) Parameters* and *Forecasting an AR(1) Process* apply NumPyro to Bayesian estimation and forecasting of autoregressive time series, where the parameter is a vector and conjugate analysis is unavailable.

CHAPTER

NINETEEN

POSTERIOR DISTRIBUTIONS FOR AR(1) PARAMETERS

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

In addition to what's included in base Anaconda, this lecture requires the following libraries.

We first install `numpyro` and `jax`:

```
!pip install numpyro jax
```

We also install `arviz` and `pymc`:

```
!pip install arviz pymc
```

19.1 Overview

This lecture uses Bayesian methods offered by pymc and numpyro to make statistical inferences about two parameters of a univariate first-order autoregression.

The model is a good laboratory for illustrating consequences of alternative ways of modeling the distribution of the initial y_0:

- As a fixed number
- As a random variable drawn from the stationary distribution of the $\{y_t\}$ stochastic process

19.1.1 Setting

The first component of the statistical model is

$$y_{t+1} = \rho y_t + \sigma_x \epsilon_{t+1}, \quad t \geq 0 \tag{19.1}$$

where

- the scalars ρ and σ_x satisfy $|\rho| < 1$ and $\sigma_x > 0$
- $\{\epsilon_{t+1}\}$ is a sequence of IID normal random variables with mean 0 and variance 1.

The second component of the statistical model is

$$y_0 \sim N(\mu_0, \sigma_0^2) \tag{19.2}$$

Unknown parameters are ρ, σ_x.

We have independent *prior probability distributions* for ρ, σ_x and want to compute a posterior probability distribution after observing a sample $\{y_t\}_{t=0}^T$.

We want to study how inferences about the unknown parameters (ρ, σ_x) depend on what is assumed about the parameters μ_0, σ_0 of the distribution of y_0.

We study two assumptions about the initial value y_0, and we refer to them by name throughout the lecture.

Under the **conditioning assumption** we take the observed y_0 as given.

Formally we set $(\mu_0, \sigma_0) = (y_0, 0)$, so the density of y_0 is a spike at its observed value.

This density does not depend on ρ or σ_x, so y_0 carries no information about the parameters; in effect we **condition on** y_0 and model only what happens after it.

Under the **stationary assumption** we treat y_0 as a draw from the stationary distribution of the process,

$$y_0 \sim N\left(0, \frac{\sigma_x^2}{1-\rho^2}\right).$$

This density *does* depend on ρ and σ_x, so now the observed y_0 carries information about the parameters.

The whole lecture is about how this one difference affects our estimates.

> **Note**
>
> We do not treat a third possible case in which μ_0, σ_0 are free parameters to be estimated.

19.1.2 Libraries

We use PyMC and NumPyro to compute the posterior distribution of ρ, σ_x.

We use two libraries because they make different trade-offs.

PyMC offers a mature and highly readable modeling syntax together with a rich set of diagnostic tools, which makes it convenient for learning and prototyping.

NumPyro is built on JAX, so it compiles to fast machine code and can run on a GPU, which helps it scale to larger models and datasets.

Because both libraries fit the same model, running them side by side also lets us check that they agree.

Both libraries support the NUTS sampler, which we use to draw samples from the posterior.

We treat NUTS as a black box here; see the *introduction to NUTS* in *Non-Conjugate Priors* for a brief account of how it works.

19.1.3 Imports

Let's start with some Python imports.

```
import arviz as az
import pymc as pm
import numpyro
from numpyro import distributions as dist

import numpy as np
import jax.numpy as jnp
from jax import random
import matplotlib.pyplot as plt

import logging
logging.basicConfig()
logger = logging.getLogger('pymc')
logger.setLevel(logging.CRITICAL)
```

19.2 Estimation

Let's turn to estimation, starting with the likelihood function.

19.2.1 Likelihood function

For a sample $\{y_t\}_{t=0}^T$ from the AR(1) model, the likelihood function can be *factored* as follows:

$$f(y_T, y_{T-1}, \cdots, y_0) = f(y_T|y_{T-1})f(y_{T-1}|y_{T-2}) \cdots f(y_1|y_0)f(y_0)$$

(We use f to denote a generic probability density.)

The statistical model (19.1)-(19.2) implies

$$\begin{aligned} f(y_t|y_{t-1}) &\sim N(\rho y_{t-1}, \sigma_x^2) \\ f(y_0) &\sim N(\mu_0, \sigma_0^2) \end{aligned}$$

We shall use Bayes' law to construct a posterior distribution under the alternative assumptions.

As discussed above, the way that we select the initial value y_0 matters.

- If we believe y_0 really is a draw from the stationary distribution, the stationary assumption is a good choice, because then y_0 carries useful information about ρ and σ_x.
- If we suspect y_0 is far out in the tail — so that early observations carry a large **transient component** — the conditioning assumption is better.

To illustrate the issue, we'll begin by choosing an initial y_0 that is far out in a tail of the stationary distribution.

19.2.2 Simulation code

We will work with simulated data, fixing parameters ρ and σ_x.

Then we will pretend that we don't know these parameters and try to estimate them under the two assumptions discussed above (conditioning and stationary).

The following function simulates a path of the AR(1) process from a given initial condition.

```
def ar1_simulate(ρ, σ, y0, T, rng):

    # Allocate space and draw shocks
    y = np.empty(T)
    ε = rng.normal(0, σ, T)

    # Initial condition and step forward
    y[0] = y0
    for t in range(1, T):
        y[t] = ρ * y[t-1] + ε[t]

    return y
```

We use the following parameterization.

```
σ_true = 1.0   # Fix a value for σ_x
ρ_true = 0.5   # Fix a value for ρ
```

Our simulated time series will be relatively short, so that the priors matter:

```
T = 50   # Length of time series
```

As mentioned above, we choose an initial y_0 that is far out in a tail of the stationary distribution:

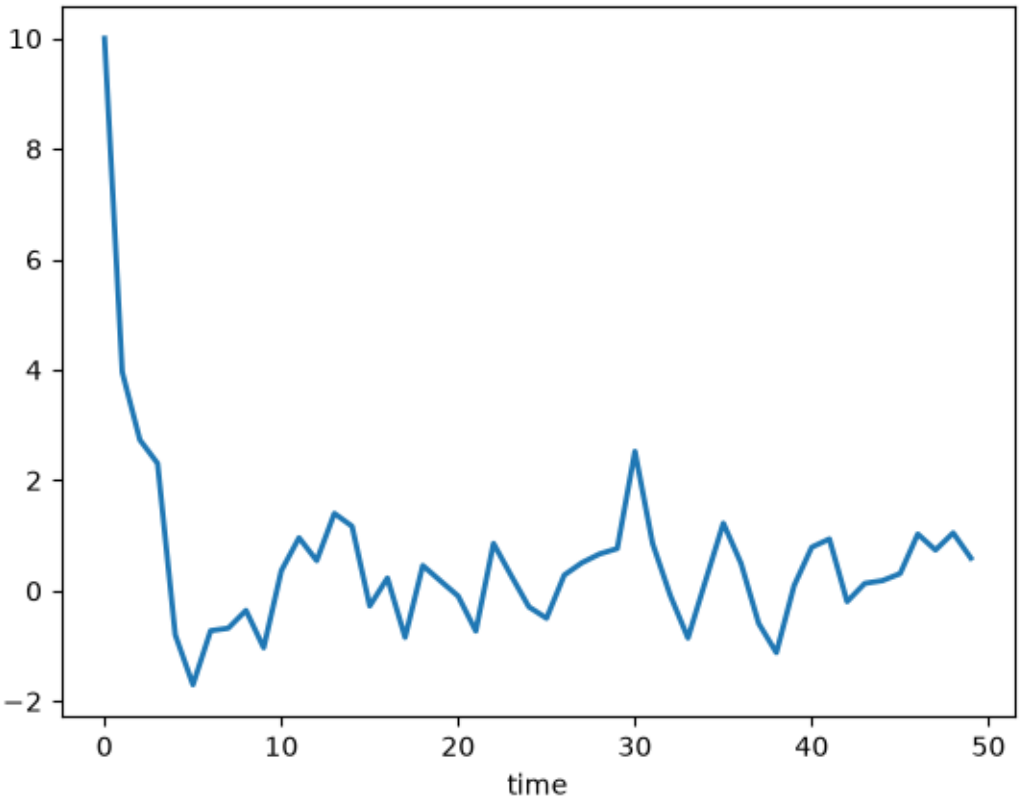
```
y_0 = 10
```

Now let's simulate and generate our data:

```
rng = np.random.default_rng(42)
y = ar1_simulate(ρ_true, σ_true, y_0, T, rng)
```

Here's a plot of the simulated series.

```
fig, ax = plt.subplots()
ax.plot(y, lw=2)
ax.set_xlabel('time')
plt.show()
```



You can see that the initial condition is unusually large — the series moves away from it quickly and fluctuates in a lower band.

19.2.3 PyMC implementation

In this section we use PyMC to compute the posterior under each assumption about y_0 — first the conditioning assumption, then the stationary assumption.

We parameterize each normal distribution in PyMC by its standard deviation σ, via the `sigma` argument.

```
AR1_model = pm.Model()

with AR1_model:

    # Start with priors
    ρ = pm.Uniform('rho', lower=-1., upper=1.)  # Assume stable ρ
    σ = pm.HalfNormal('sigma', sigma=np.sqrt(10))

    # Expected value of y at the next period (ρ * y)
    yhat = ρ * y[:-1]

    # Likelihood of the actual realization
    y_like = pm.Normal('y_obs', mu=yhat, sigma=σ, observed=y[1:])
```

Let's unpack what this model declares.

Inside the `with AR1_model:` block, each `pm` statement adds a random variable to the model.

The first two lines are the **priors** — a uniform prior on ρ over $(-1, 1)$, which builds in stationarity, and a half-normal prior on σ.

The line `yhat = ρ * y[:-1]` is just the vector of conditional means ρy_{t-1} for $t = 1, \ldots, T$.

The last line is the **likelihood**: the keyword `observed=y[1:]` tells PyMC that the values `y[1:]` are data, drawn from $N(\rho y_{t-1}, \sigma^2)$.

Because `yhat` and `y[1:]` are whole vectors, this single line encodes the entire product $\prod_{t=1}^{T} f(y_t \mid y_{t-1})$ from the factorization above.

PyMC multiplies this likelihood by the priors to form the posterior, which we sample from below.

Note

Notice what is *absent*: we never write down a density for y_0 itself.

It enters the model only inside `y[:-1]`, as the conditioning value for the first transition $f(y_1 \mid y_0)$ — never as something the model has to explain.

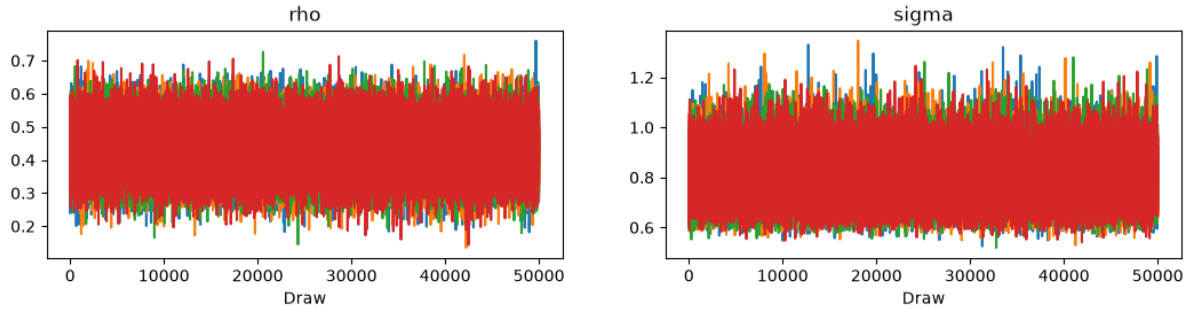
Leaving out the $f(y_0)$ factor in this way is precisely the **conditioning assumption**.

pm.sample by default uses the NUTS sampler to generate samples, as shown in the cell below:

```
with AR1_model:
    trace = pm.sample(50000, tune=10000, return_inferencedata=True)
```

We plot the trace and the posterior densities for the two parameters.

```
with AR1_model:
    az.plot_trace(trace)
```



Recall that we generated the data with $\rho = 0.5$ and $\sigma_x = 1$.

The posteriors concentrate near these values, so conditioning on y_0 recovers the parameters reasonably well, even from this short sample.

Note

The fit is good but not exact — the posterior for ρ sits a little below its true value.

This is partly the classic **Hurwicz bias**: in a first-order autoregression the least-squares (and hence posterior) estimate of ρ is biased downward in small samples, with the bias shrinking as the sample grows (see Hurwicz [1950] and [Orcutt and Winokur, 1969]).

Here is a numerical summary of the posterior.

```
with AR1_model:
    summary = az.summary(trace, round_to=4)

summary
```

```
          mean      sd  eti89_lb  eti89_ub     ess_bulk     ess_tail  r_hat  \
rho     0.4374  0.0632    0.3370    0.5387  180098.7978  133934.1226    1.0
sigma   0.7922  0.0837    0.6704    0.9352  175119.6815  125654.5530    1.0

        mcse_mean  mcse_sd
rho        0.0001   0.0001
sigma      0.0002   0.0002
```

Now we switch to the stationary assumption, using the same data.

Recall that this means

$$y_0 \sim N\left(0, \frac{\sigma_x^2}{1-\rho^2}\right).$$

We alter the code as follows:

```
AR1_model_y0 = pm.Model()

with AR1_model_y0:

    # Start with priors
    ρ = pm.Uniform('rho', lower=-1., upper=1.)  # Assume stable ρ
    σ = pm.HalfNormal('sigma', sigma=np.sqrt(10))

    # Standard deviation of the stationary distribution
    y_sd = σ / np.sqrt(1 - ρ**2)

    # Expected value of y at the next period (ρ * y)
    yhat = ρ * y[:-1]
    y_data = pm.Normal('y_obs', mu=yhat, sigma=σ, observed=y[1:])

    # Density for y0 -- this term imposes the stationary assumption
    y0_data = pm.Normal('y0_obs', mu=0., sigma=y_sd, observed=y[0])
```

The only change from the first model is that final line.

Note

The new line adds a density for y_0 — the stationary density $N(0, \sigma_x^2/(1-\rho^2))$ — through a second `observed` term.

This restores the $f(y_0)$ factor we dropped before, and so *is* the **stationary assumption**.

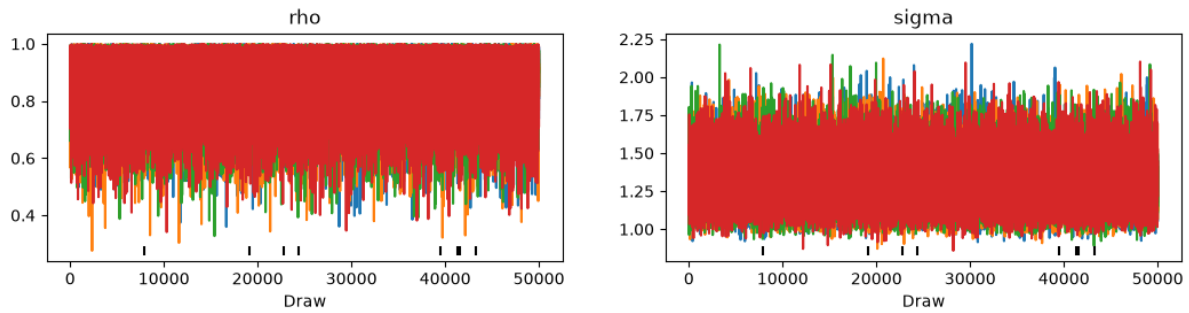
Everything else is identical, so any difference between the two posteriors comes entirely from this one term.

As before, we sample from the posterior.

```
with AR1_model_y0:
    trace_y0 = pm.sample(50000, tune=10000, return_inferencedata=True)
```

In the trace plot below, any grey vertical lines mark sampler divergences.

```
with AR1_model_y0:
    az.plot_trace(trace_y0)
```



Here is a summary of the posterior.

```
with AR1_model_y0:
    summary_y0 = az.summary(trace_y0, round_to=4)

summary_y0
```

```
         mean      sd  eti89_lb  eti89_ub     ess_bulk     ess_tail  r_hat  \
rho    0.8796  0.0852    0.7209    0.9818  112764.9852   89290.4204    1.0
sigma  1.3164  0.1383    1.1159    1.5534  139291.8797  116539.8696    1.0

       mcse_mean  mcse_sd
rho       0.0002   0.0002
sigma     0.0004   0.0003
```

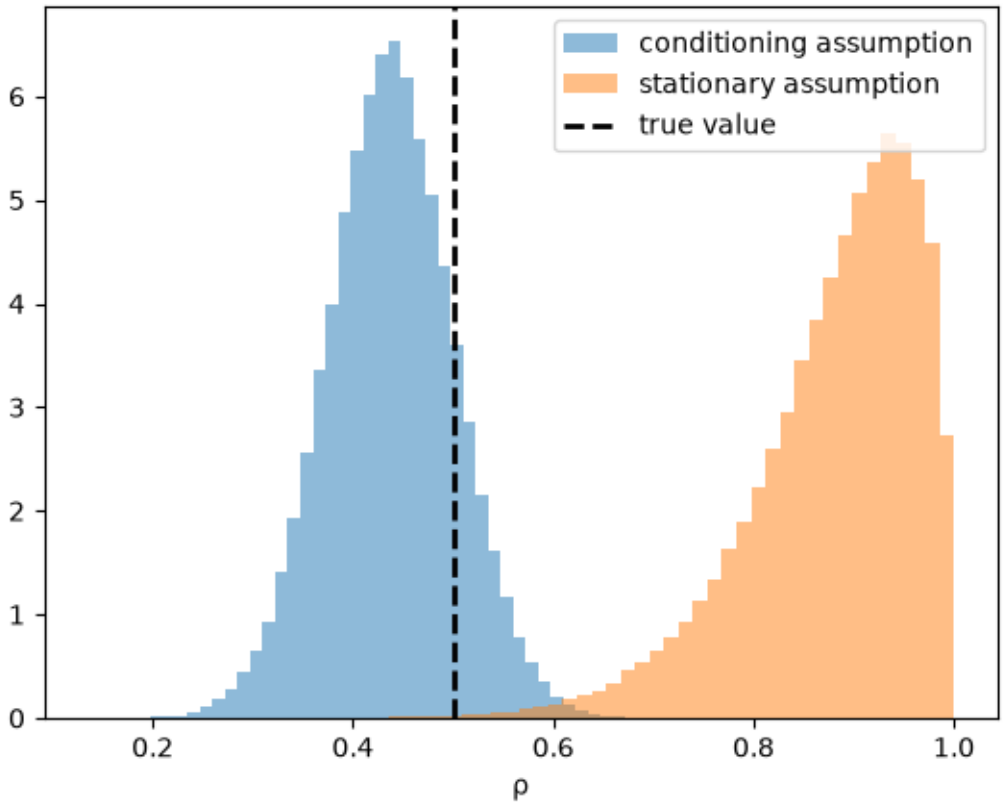
The posterior for ρ has clearly moved when we changed our assumption about y_0.

19.2.4 Comparing the two posteriors

Let's put the two posteriors for ρ side by side to see what changed.

The figure below overlays the posterior for ρ under each assumption, with the true value marked by a dashed line.

```
ρ_cond = trace.posterior['rho'].values.flatten()
ρ_stat = trace_y0.posterior['rho'].values.flatten()

fig, ax = plt.subplots()
ax.hist(ρ_cond, bins=50, density=True, alpha=0.5,
        label='conditioning assumption')
ax.hist(ρ_stat, bins=50, density=True, alpha=0.5,
        label='stationary assumption')
ax.axvline(ρ_true, color='k', linestyle='--', lw=2, label='true value')
ax.set_xlabel('ρ')
ax.legend()
plt.show()
```



The posterior from the conditioning assumption sits close to the true value of 0.5.

It is centred a little below 0.5, which is the small-sample Hurwicz bias mentioned above — a mild downward pull that shrinks as the sample grows.

The posterior from the stationary assumption is different: it is pushed well to the right, toward $\rho = 1$.

Here is why.

We chose a starting value $y_0 = 10$ that lies far out in the tail of the stationary distribution.

If y_0 really were a draw from that distribution, such an extreme value would be very unlikely.

So when we force the model to explain y_0 this way, Bayes' law looks for parameter values that make the extreme y_0 plausible.

It does this by pushing ρ toward 1, because a larger ρ inflates the stationary variance $\sigma_x^2/(1-\rho^2)$ and makes a large y_0 less surprising.

A single unusual starting value therefore drags the whole posterior away from the truth.

This is the sense in which the conditioning assumption is more accurate here: it does not let one atypical observation distort our view of ρ and σ_x.

19.2.5 NumPyro implementation

We now redo both computations with NumPyro.

Because it fits the same two models, we expect its posteriors to match those from PyMC.

The models are the same; only the syntax differs.

NumPyro describes a model as an ordinary Python function rather than a `with` block, each `numpyro.sample('name', distribution)` plays the role that a `pm` random variable did above, and the keyword `obs=` is NumPyro's counterpart of PyMC's `observed=`.

Everything we said about priors, the vectorized likelihood, and how the two assumptions are imposed carries over unchanged; see *Non-Conjugate Priors* for a fuller introduction to NumPyro models.

We start with a helper function that plots the trace and posterior histogram for the sampled parameters.

```
def plot_posterior(sample):
    """
    Plot trace and histogram
    """
    # To np array
    ρs = np.array(sample['rho'])
    σs = np.array(sample['sigma'])

    fig, axs = plt.subplots(2, 2, figsize=(17, 6))
    # Plot trace
    axs[0, 0].plot(ρs, lw=2)
    axs[1, 0].plot(σs, lw=2)

    # Plot posterior
    axs[0, 1].hist(ρs, bins=50, density=True, alpha=0.7)
    axs[0, 1].set_xlim([0, 1])
    axs[1, 1].hist(σs, bins=50, density=True, alpha=0.7)

    axs[0, 0].set_ylabel('ρ')
    axs[1, 0].set_ylabel('σ')
    axs[0, 1].set_xlabel('ρ')
    axs[1, 1].set_xlabel('σ')
    plt.show()
```

The first model uses the conditioning assumption.

```
def AR1_model(data):
    # Set prior
    ρ = numpyro.sample('rho', dist.Uniform(low=-1., high=1.))
    σ = numpyro.sample('sigma', dist.HalfNormal(scale=np.sqrt(10)))

    # Expected value of y at the next period (ρ * y)
    yhat = ρ * data[:-1]

    # Likelihood of the actual realization
    y_data = numpyro.sample('y_obs', dist.Normal(loc=yhat, scale=σ), obs=data[1:])
```

We convert the data to a JAX array, build the NUTS sampler, and run MCMC.

```
# Make jnp array
y = jnp.array(y)
```

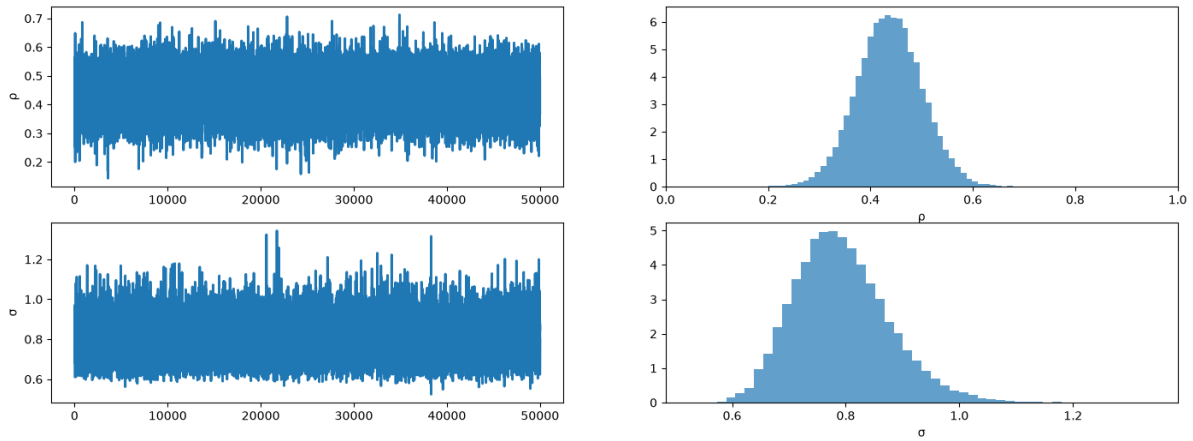
(continues on next page)

(continued from previous page)

```
# Set NUTS kernel
NUTS_kernel = numpyro.infer.NUTS(AR1_model)

# Run MCMC
mcmc = numpyro.infer.MCMC(NUTS_kernel, num_samples=50000, num_warmup=10000, progress_
↪bar=False)
mcmc.run(rng_key=random.PRNGKey(1), data=y)
```

We plot the trace and posterior.

```
plot_posterior(mcmc.get_samples())
```



Here is the posterior summary.

```
mcmc.print_summary()
```

```
                mean       std    median      5.0%     95.0%     n_eff     r_hat
       rho       0.44      0.06      0.44      0.33      0.54  33558.19      1.00
     sigma       0.79      0.08      0.79      0.66      0.93  32561.73      1.00

Number of divergences: 0
```

Next we use the stationary assumption, where

$$y_0 \sim N\left(0, \frac{\sigma_x^2}{1-\rho^2}\right).$$

Here's the new code to achieve this.

```
def AR1_model_y0(data):
    # Set prior
    ρ = numpyro.sample('rho', dist.Uniform(low=-1., high=1.))
    σ = numpyro.sample('sigma', dist.HalfNormal(scale=np.sqrt(10)))

    # Standard deviation of the stationary distribution
    y_sd = σ / jnp.sqrt(1 - ρ**2)

    # Expected value of y at the next period (ρ * y)
    yhat = ρ * data[:-1]
```

(continues on next page)

(continued from previous page)

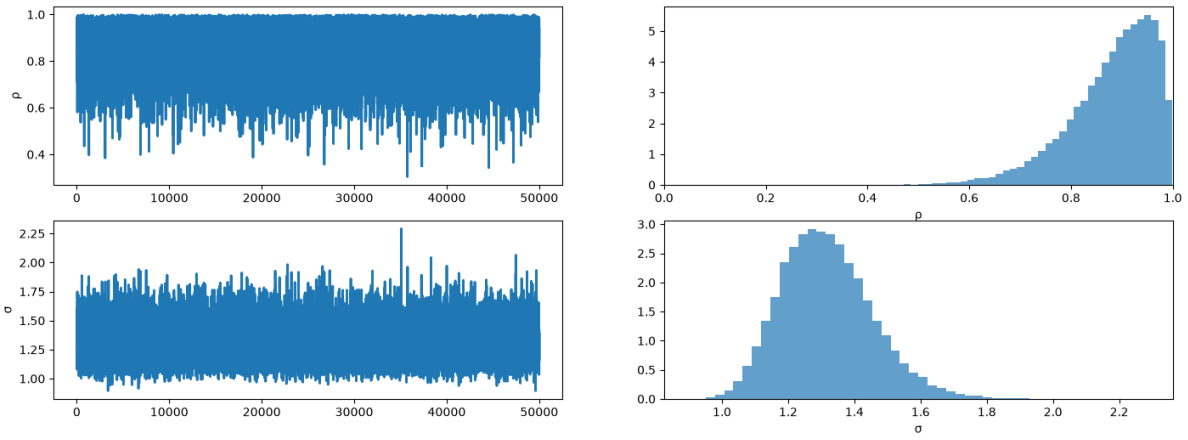
```
    # Likelihood of the actual realization
    y_data = numpyro.sample('y_obs', dist.Normal(loc=yhat, scale=σ), obs=data[1:])

    # Density for y0 -- this term imposes the stationary assumption
    y0_data = numpyro.sample('y0_obs', dist.Normal(loc=0., scale=y_sd), obs=data[0])
```

We build the sampler for this model and run MCMC.

```
# Set NUTS kernel
NUTS_kernel = numpyro.infer.NUTS(AR1_model_y0)

# Run MCMC
mcmc2 = numpyro.infer.MCMC(NUTS_kernel, num_samples=50000, num_warmup=10000, progress_
↪bar=False)
mcmc2.run(rng_key=random.PRNGKey(1), data=y)
```

Again we plot the trace and posterior.

```
plot_posterior(mcmc2.get_samples())
```



And here is the posterior summary.

```
mcmc2.print_summary()
```

```
                mean       std    median      5.0%     95.0%     n_eff     r_hat
       rho       0.88      0.09      0.90      0.76      1.00  32784.94      1.00
     sigma       1.32      0.14      1.31      1.09      1.53  24454.89      1.00

Number of divergences: 0
```

As with PyMC, the posterior for ρ shifts toward 1 once we switch to the stationary assumption.

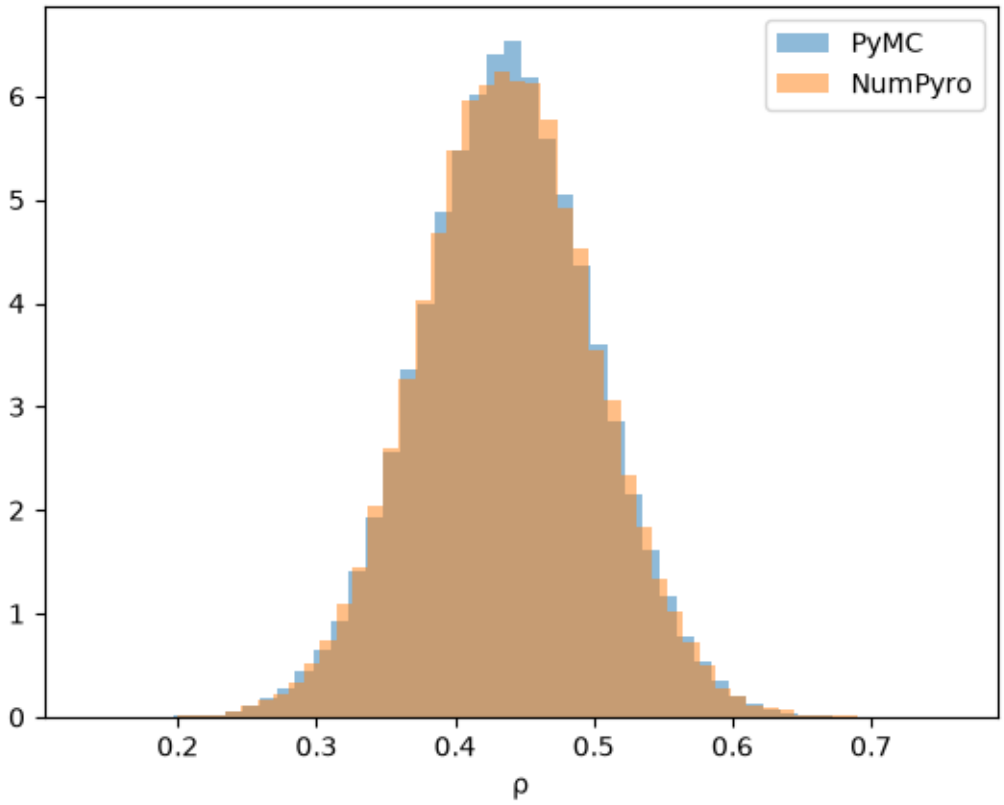
To confirm that the two libraries agree, we overlay their posteriors for ρ under the conditioning assumption.

```
ρ_pymc = trace.posterior['rho'].values.flatten()
ρ_numpyro = np.array(mcmc.get_samples()['rho'])

fig, ax = plt.subplots()
ax.hist(ρ_pymc, bins=50, density=True, alpha=0.5, label='PyMC')
ax.hist(ρ_numpyro, bins=50, density=True, alpha=0.5, label='NumPyro')
ax.set_xlabel('ρ')
```

(continues on next page)

(continued from previous page)

```
ax.legend()
plt.show()
```



The two posteriors line up, as they should.

19.3 Conclusion

This lecture showed that what we assume about the initial value y_0 can have a large effect on our estimates of an AR(1) process.

When the sample is short and y_0 might be unusual, the conditioning assumption is the safer choice.

It lets the data speak about ρ and σ_x without forcing the model to explain the starting value.

The stationary assumption adds information, and that information is valuable when the assumption is true.

But when y_0 is in fact far from typical, the same assumption misleads us — here it pushed ρ toward 1 and away from the truth.

A simple rule of thumb:

- use the conditioning assumption when early observations look transient or the starting point may be atypical;
- use the stationary assumption when you are confident the process has been running near its long-run behaviour.

See also

Forecasting an AR(1) Process builds on the posterior computed here to forecast nonlinear sample-path statistics of an AR(1) process, such as the time until the next turning point.

CHAPTER

TWENTY

FORECASTING AN AR(1) PROCESS

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

```
!pip install numpyro jax arviz
```

This lecture describes methods for forecasting statistics that are functions of future values of a univariate autoregressive process.

The methods are designed to take into account two possible sources of uncertainty about these statistics:

- random shocks that impinge on the transition law
- uncertainty about the parameter values of the AR(1) process

We consider two sorts of statistics:

- prospective values y_{t+j} of a random process $\{y_t\}$ that is governed by the AR(1) process
- sample path properties that are defined as non-linear functions of future values $\{y_{t+j}\}_{j\geq 1}$ at time t

Sample path properties are things like "time to next turning point" or "time to next recession".

To investigate sample path properties we'll use a simulation procedure recommended by Wecker [Wecker, 1979].

To acknowledge uncertainty about parameters, we'll deploy `numpyro` to construct a Bayesian joint posterior distribution for unknown parameters.

This lecture builds on *Posterior Distributions for AR(1) Parameters*, which studies Bayesian inference for the parameters of exactly this AR(1) model in detail.

We recommend reading that lecture first, since here we treat the construction of the posterior more briefly.

Let's start with some imports.

```
import matplotlib.pyplot as plt
import seaborn as sns
from typing import NamedTuple

# numpyro
```

(continues on next page)

(continued from previous page)

```
import numpyro
import numpyro.distributions as dist
from numpyro.infer import MCMC, NUTS

# jax
import jax
import jax.random as random
import jax.numpy as jnp
from jax import lax

# arviz
import arviz as az

sns.set_style('white')
key = random.PRNGKey(0)
```

20.1 A Univariate First-Order Autoregressive Process

Consider the univariate AR(1) model, which is the same model studied in *Posterior Distributions for AR(1) Parameters*:

$$y_{t+1} = \rho y_t + \sigma \epsilon_{t+1}, \quad t \geq 0 \tag{20.1}$$

where

- the scalars ρ and σ satisfy $|\rho| < 1$ and $\sigma > 0$
- $\{\epsilon_{t+1}\}$ is a sequence of IID normal random variables with mean 0 and variance 1.

The initial condition y_0 is a known number.

Equation (20.1) implies that for $t \geq 0$, the conditional density of y_{t+1} is

$$f(y_{t+1}|y_t; \rho, \sigma) \sim \mathcal{N}(\rho y_t, \sigma^2) \tag{20.2}$$

Further, equation (20.1) also implies that for $t \geq 0$, the conditional density of y_{t+j} for $j \geq 1$ is

$$f(y_{t+j}|y_t; \rho, \sigma) \sim \mathcal{N}\left(\rho^j y_t, \sigma^2 \frac{1-\rho^{2j}}{1-\rho^2}\right) \tag{20.3}$$

The predictive distribution (20.3) assumes that the parameters ρ, σ are known, which we express by conditioning on them.

We also want to compute a predictive distribution that does not condition on ρ, σ but instead takes account of our uncertainty about them.

We form this predictive distribution by integrating (20.3) with respect to a joint posterior distribution $\pi_t(\rho, \sigma|y^t)$ that conditions on an observed history $y^t = \{y_s\}_{s=0}^t$:

$$\begin{aligned} f(y_{t+j}|y^t) &= \int f(y_{t+j}|y^t, \rho, \sigma)\pi_t(\rho, \sigma|y^t)d\rho d\sigma \\ &= \int f(y_{t+j}|y_t, \rho, \sigma)\pi_t(\rho, \sigma|y^t)d\rho d\sigma \end{aligned} \tag{20.4}$$

Predictive distribution (20.3) assumes that parameters (ρ, σ) are known.

Predictive distribution (20.4) assumes that parameters (ρ, σ) are uncertain, but have a known probability distribution $\pi_t(\rho, \sigma|y^t)$. Notice the second equality follows because $\{y_t\}$ is an AR(1) process when (ρ, σ) are given.

We also want to compute some predictive distributions of "sample path statistics" that might include, for example

- the time until the next "recession",
- the minimum value of Y over the next 8 periods,
- "severe recession", and
- the time until the next turning point (positive or negative).

To accomplish that for situations in which we are uncertain about parameter values, we shall extend Wecker's [Wecker, 1979] approach in the following way.

- first simulate an initial path of length T_0;
- for a given prior, draw a sample of size N from the posterior joint distribution of parameters (ρ, σ) after observing the initial path;
- for each draw $n = 0, 1, ..., N$, simulate a "future path" of length T_1 with parameters (ρ_n, σ_n) and compute our "sample path statistics";
- finally, plot the desired statistics from the N samples as an empirical distribution.

20.2 Implementation

First, we'll simulate a sample path from which to launch our forecasts.

In addition to plotting the sample path, under the assumption that the true parameter values are known, we'll plot 0.9 and 0.95 coverage intervals using conditional distribution (20.3) described above.

We'll also plot a bunch of samples of sequences of future values and watch where they fall relative to the coverage interval.

```
class AR1(NamedTuple):
    """
    Represents a univariate first-order autoregressive (AR(1)) process.

    Parameters
    ----------
    ρ : float
        Autoregressive coefficient, must satisfy |ρ| < 1 for stationarity.
    σ : float
        Standard deviation of the error term.
    y0 : float
        Initial value of the process at time t=0.
    T0 : int, optional
        Length of the initial observed path (default is 100).
    T1 : int, optional
        Length of the future path to simulate (default is 100).
    """
    ρ: float
    σ: float
    y0: float
    T0: int = 100
    T1: int = 100
```

We create instances through a factory function that validates the parameters.

```
def create_ar1(ρ=0.9, σ=1.0, y0=10.0, T0=100, T1=100):
    """Create an AR(1) instance, checking the parameter restrictions."""
    if not abs(ρ) < 1:
        raise ValueError("ρ must satisfy |ρ| < 1 for stationarity")
```

(continues on next page)

(continued from previous page)

```
    if not σ > 0:
        raise ValueError("σ must be positive")
    return AR1(ρ=ρ, σ=σ, y0=y0, T0=T0, T1=T1)
```

Using the `AR1` class, we can simulate paths more conveniently. The following function simulates an initial path of length T_0.

```
def AR1_simulate_past(ar1: AR1, key=key):
    """
    Simulate a realization of the AR(1) process for T0 periods.

    Parameters
    ----------
    ar1 : AR1
        AR1 named tuple containing parameters (ρ, σ, y0, T0, T1).
    key : jax.random.PRNGKey
        JAX random key for generating random noise.

    Returns
    -------
    initial_path : jax.numpy.ndarray
        Simulated path of the AR(1) process and the initial y0.
    """
    ρ, σ, y0, T0 = ar1.ρ, ar1.σ, ar1.y0, ar1.T0
    # Draw εs
    ε = σ * random.normal(key, (T0,))

    # Set step function
    def ar1_step(y_prev, t_ρ_ε):
        ρ, ε_t = t_ρ_ε
        y_t = ρ * y_prev + ε_t
        return y_t, y_t

    # Scan over time steps
    _, y_seq = lax.scan(ar1_step, y0, (jnp.full(T0, ρ), ε))

    # Concatenate initial value
    initial_path = jnp.concatenate([jnp.array([y0]), y_seq])

    return initial_path
```

Now we define the simulation function that generates a realization of the AR(1) process for the future T_1 periods.

```
def AR1_simulate_future(ar1: AR1, y_T0, N=10, key=key):
    """
    Simulate a realization of the AR(1) process for T1 periods.

    Parameters
    ----------
    ar1 : AR1
        AR1 named tuple containing parameters (ρ, σ, y0, T0, T1).
    y_T0 : float
        Value of the process at time T0.
    N: int
        Number of paths to simulate.
    key : jax.random.PRNGKey
        JAX random key for generating random noise.
```

(continues on next page)

(continued from previous page)

```
    Returns
    -------
    future_path : jax.numpy.ndarray
        Simulated N paths of the AR(1) process of length T1.
    """
    ρ, σ, T1 = ar1.ρ, ar1.σ, ar1.T1

    def single_path_scan(y_T0, subkey):
        ε = σ * random.normal(subkey, (T1,))

        def ar1_step(y_prev, t_ρ_ε):
            ρ, ε_t = t_ρ_ε
            y_t = ρ * y_prev + ε_t
            return y_t, y_t
        _, y = lax.scan(ar1_step, y_T0, (jnp.full(T1, ρ), ε))
        return y

    # Split key to generate different paths
    subkeys = random.split(key, num=N)

    # Simulate N paths
    future_path = jax.vmap(single_path_scan, in_axes=(None, 0))(y_T0, subkeys)

    return future_path
```

The following function plots the initial observed AR(1) path and simulated future paths along with predictive confidence intervals.

```
def plot_path(ar1, initial_path, future_path, ax, key=key):
    """
    Plot the initial observed AR(1) path and simulated future paths,
    along with predictive confidence intervals.

    Parameters
    ----------
    ar1 : AR1
        AR1 named tuple containing process parameters (ρ, σ, T0, T1).
    initial_path : array-like
        Simulated initial path of the AR(1) process, shape(T0+1,).
    future_path : array-like
        Simulated future paths of the AR(1) process, shape (N, T1).
    ax : matplotlib.axes.Axes
        Matplotlib axes object to plot on.
    key : jax.random.PRNGKey, optional
        JAX random key for reproducible sampling.

    Plots
    -----
    - Initial path (historical data)
    - Multiple simulated future paths
    - 90% and 95% predictive confidence intervals
    - Expected future path
    """
    ρ, σ, T0, T1 = ar1.ρ, ar1.σ, ar1.T0, ar1.T1

    # Compute moments and confidence intervals
```

(continues on next page)

(continued from previous page)

```
    y_T0 = initial_path[-1]
    j = jnp.arange(1, T1+1)
    center = ρ**j * y_T0
    vars = σ**2 * (1 - ρ**(2 * j)) / (1 - ρ**2)

    # 95% CI
    y_upper_c95 = center + 1.96 * jnp.sqrt(vars)
    y_lower_c95 = center - 1.96 * jnp.sqrt(vars)

    # 90% CI
    y_upper_c90 = center + 1.65 * jnp.sqrt(vars)
    y_lower_c90 = center - 1.65 * jnp.sqrt(vars)

    # Plot
    ax.plot(jnp.arange(-T0, 1), initial_path, lw=2)
    ax.axvline(0, linestyle='--', alpha=.4, color='k', lw=1)

    # Choose 10 future paths to plot
    index = random.choice(
        key, jnp.arange(future_path.shape[0]), (10,), replace=False
        )
    for i in index:
        ax.plot(jnp.arange(1, T1+1), future_path[i, :], color='grey', alpha=.5)

    # Plot 90% and 95% CI
    ax.fill_between(
        jnp.arange(1, T1+1), y_upper_c95, y_lower_c95, alpha=.3, label='95% CI'
        )
    ax.fill_between(
        jnp.arange(1, T1+1), y_upper_c90, y_lower_c90, alpha=.35, label='90% CI'
        )
    ax.plot(
        jnp.arange(1, T1+1), center, color='red', alpha=.7, lw=2,
        label='expectation'
        )
    ax.set_xlim([-T0, T1])
    ax.set_xlabel("time", fontsize=13)
    ax.set_ylabel("y", fontsize=13)
    ax.legend(fontsize=8)
```

```
ar1 = create_ar1(ρ=0.9, σ=1.0, y0=10.0)

# Simulate
initial_path = AR1_simulate_past(ar1)
future_path = AR1_simulate_future(ar1, initial_path[-1])

# Plot
fig, ax = plt.subplots(1, 1)
plot_path(ar1, initial_path, future_path, ax)
plt.show()
```

As functions of forecast horizon, the coverage intervals have shapes like those described in *Permanent Income II: LQ Techniques*

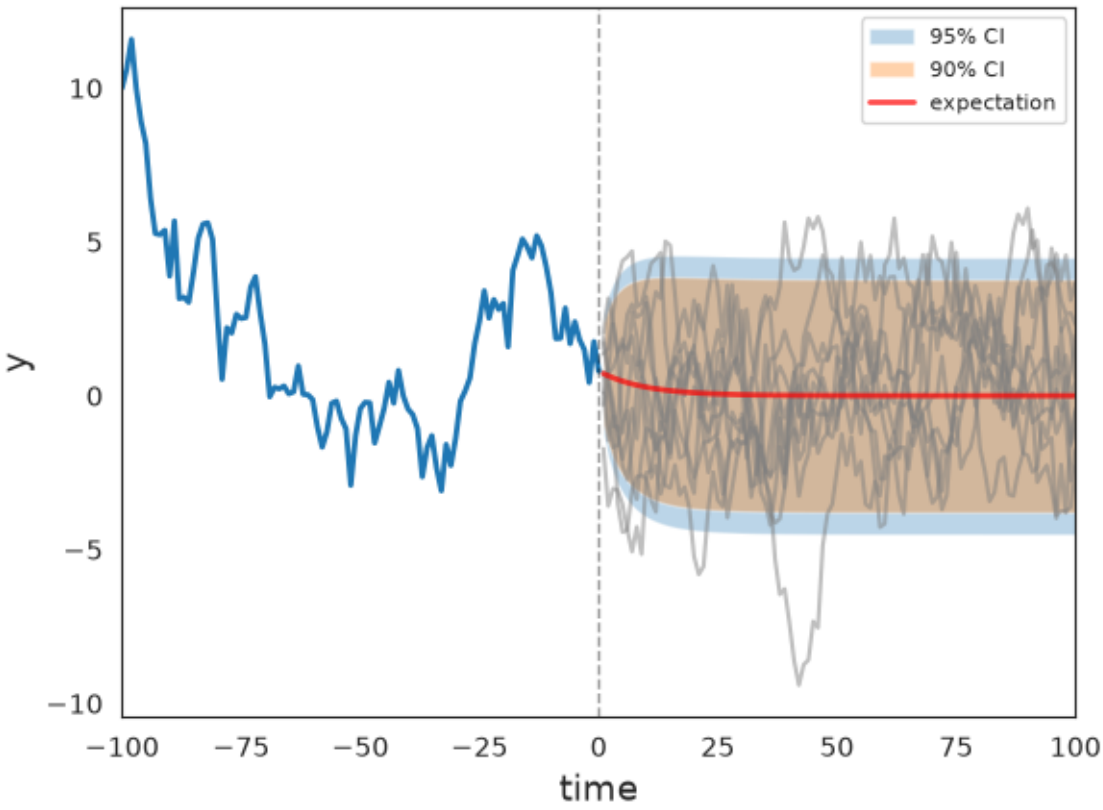


Fig. 20.1: Initial and predictive future paths

20.3 Predictive Distributions of Path Properties

Wecker [Wecker, 1979] proposed using simulation techniques to characterize the predictive distribution of some statistics that are non-linear functions of y.

He called these functions "path properties" to contrast them with properties of single data points.

He studied two special prospective path properties of a given series $\{y_t\}$.

The first was *time until the next turning point*.

- he defined a **"turning point"** to be the date of the second of two successive declines in y.

For example, if $y_t(\omega) < y_{t-1}(\omega) < y_{t-2}(\omega)$, then period t is a turning point.

To examine the *time until the next turning point*, let Z be an indicator process

$$Z_t(\omega) := \begin{cases} 1 & \text{if } Y_t(\omega) < Y_{t-1}(\omega) < Y_{t-2}(\omega) \geq Y_{t-3}(\omega) \\ 0 & \text{otherwise} \end{cases}$$

Here $\omega \in \Omega$ is a sequence of events, and $Y_t : \Omega \to \mathbb{R}$ gives y_t according to ω and the AR(1) process.

By Wecker's definition, period t is a turning point, and $Y_{t-2}(\omega) \geq Y_{t-3}(\omega)$ excludes the possibility that period $t-1$ is a turning point.

Then the random variable **time until the next turning point** is defined as the following *stopping time* with respect to Z:

$$W_t(\omega) := \inf\{k \geq 1 \mid Z_{t+k}(\omega) = 1\}$$

In the following code, we name this statistic *time until the next recession* to distinguish it from another concept of *turning point*.

Moreover, the statistic *time until the next severe recession* is defined in a similar way, except the decline between periods is greater than 0.02.

Wecker [Wecker, 1979] also studied **the minimum value of Y over the next 8 quarters**, which can be defined as the random variable

$$M_t(\omega) := \min\{Y_{t+1}(\omega); Y_{t+2}(\omega); \dots; Y_{t+8}(\omega)\}$$

It is interesting to study yet another possible concept of a **turning point**.

Thus, let

$$T_t(\omega) := \begin{cases} 1 & \text{if } Y_{t-2}(\omega) > Y_{t-1}(\omega) > Y_t(\omega) \text{ and } Y_t(\omega) < Y_{t+1}(\omega) < Y_{t+2}(\omega) \\ -1 & \text{if } Y_{t-2}(\omega) < Y_{t-1}(\omega) < Y_t(\omega) \text{ and } Y_t(\omega) > Y_{t+1}(\omega) > Y_{t+2}(\omega) \\ 0 & \text{otherwise} \end{cases}$$

Define a **positive turning point today or tomorrow** statistic as

$$P_t(\omega) := \begin{cases} 1 & \text{if } T_t(\omega) = 1 \text{ or } T_{t+1}(\omega) = 1 \\ 0 & \text{otherwise} \end{cases}$$

This is designed to express the event

- "after one or two decrease(s), Y will grow for two consecutive quarters"

The **negative turning point today or tomorrow** N_t is defined in the same way.

Following [Wecker, 1979], we can use simulations to calculate probabilities of P_t and N_t for each period t.

However, in the following code, we only use $T_{t+1}(\omega) = 1$ to determine $P_t(\omega)$ and $N_t(\omega)$, because we only want to find the first positive turning point.

20.4 A Wecker-Like Algorithm

The procedure consists of the following steps:

- index a sample path by ω_i
- from a given date t, simulate I sample paths of length N

$$Y(\omega_i) = \{Y_{t+1}(\omega_i), Y_{t+2}(\omega_i), \dots, Y_{t+N}(\omega_i)\}_{i=1}^{I}$$

- for each path ω_i, compute the associated value of $W_t(\omega_i), W_{t+1}(\omega_i), \dots, W_{t+N}$
- consider the sets $\{W_t(\omega_i)\}_{i=1}^{I}$, $\{W_{t+1}(\omega_i)\}_{i=1}^{I}$, ..., $\{W_{t+N}(\omega_i)\}_{i=1}^{I}$ as samples from the predictive distributions $f(W_{t+1} \mid y_t, y_{t-1}, \dots, y_0)$, $f(W_{t+2} \mid y_t, y_{t-1}, \dots, y_0)$, ..., $f(W_{t+N} \mid y_t, y_{t-1}, \dots, y_0)$.

20.5 Using Simulations to Approximate a Posterior Distribution

The next code cells use `numpyro` to compute the time t posterior distribution of ρ, σ.

We construct this posterior just as in *Posterior Distributions for AR(1) Parameters*, to which we refer for a detailed discussion.

As there, in defining the likelihood function we condition on the initial value y_0.

This is the **conditioning assumption** of *Posterior Distributions for AR(1) Parameters*, and it is the appropriate choice here because our initial path starts from an atypical value $y_0 = 10$.

```
def draw_from_posterior(data, size=10000, dis_plot=True, key=key):
    """Draw a sample of the given size from the posterior distribution."""

    def model(data):
        # Start with priors
        ρ = numpyro.sample('ρ', dist.Uniform(-1, 1))  # Assume stable ρ
        σ = numpyro.sample('σ', dist.HalfNormal(jnp.sqrt(10)))

        # Expected value of y at the next period (ρ * y), conditioning on y_0
        yhat = ρ * data[:-1]

        # Likelihood of the actual realization
        numpyro.sample('y_obs', dist.Normal(yhat, σ), obs=data[1:])

    # Compute posterior distribution of parameters
    nuts_kernel = NUTS(model)

    # Define MCMC class to compute the posteriors
    mcmc = MCMC(
        nuts_kernel,
        num_warmup=5000,
        num_samples=size,
        num_chains=4,  # plot 4 chains in the trace
        progress_bar=False,
        chain_method='vectorized'
    )

    # Run MCMC
    mcmc.run(key, data=data)
```

(continues on next page)

(continued from previous page)

```
    # Get posterior samples
    post_sample = {
        'ρ': mcmc.get_samples()['ρ'],
        'σ': mcmc.get_samples()['σ'],
    }

    # Plot posterior distributions and trace plots
    if dis_plot:
        plot_data = az.from_numpyro(posterior=mcmc)
        az.plot_trace_dist(plot_data, var_names=['ρ', 'σ'])

    return post_sample

post_samples = draw_from_posterior(initial_path)
```

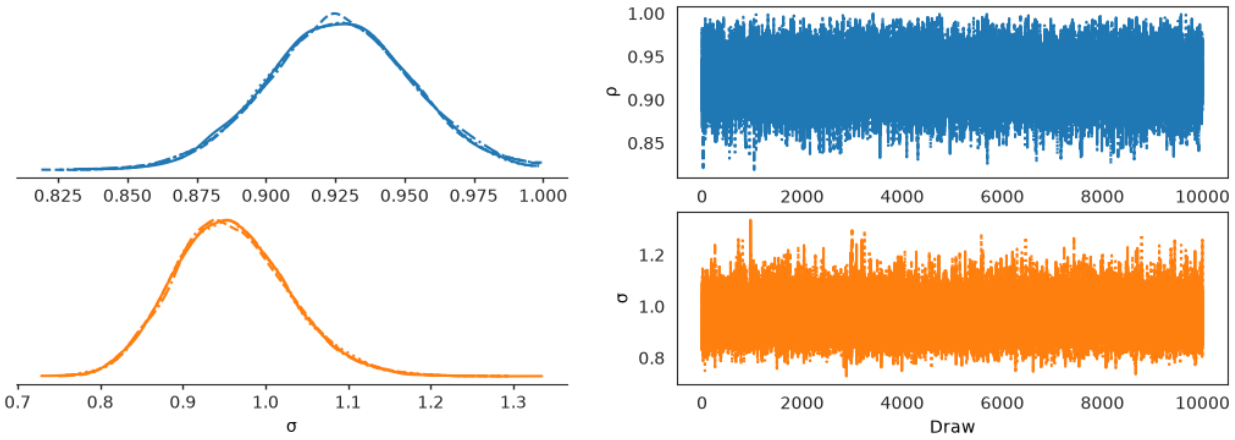


Fig. 20.2: Posterior and trace plots

The graphs above portray posterior distributions and trace plots. The posterior distributions (left column) show the marginal distributions of the parameters after observing the data, while the trace plots (right column) help diagnose MCMC convergence by showing how the sampler explored the parameter space over iterations.

20.6 Calculating Sample Path Statistics

Our next step is to prepare Python code to compute our sample path statistics.

These statistics were originally defined as random variables with respect to ω, but here we use $\{Y_t\}$ as the argument because ω is implicit.

These two kinds of definitions are equivalent because ω determines path statistics only through $\{Y_t\}$.

Moreover, we ignore all equality in the definitions, as equality occurs with zero probability for continuous random variables.

```
@jax.jit
def compute_path_statistics(initial_path, future_path):
    """Compute path statistics for the AR(1) process."""
```

(continues on next page)

(continued from previous page)

```
    # Prepend the last three observed values so we can identify a turn at t=0
    y = jnp.concatenate([initial_path[-3:], future_path])
    n = y.shape[0]

    def step(carry, i):
        # identify recession
        rec_cond = (y[i] < y[i-1]) & (y[i-1] < y[i-2]) & (y[i-2] > y[i-3])
        # identify severe recession
        sev_cond = (
            (y[i] - y[i-1] < -0.02) & (y[i-1] - y[i-2] < -0.02) & (y[i-2] > y[i-3])
            )

        # identify positive turning point
        up_cond = (
            (y[i-2] > y[i-1]) & (y[i-1] > y[i]) & (y[i] < y[i+1]) & (y[i+1] < y[i+2])
        )

        # identify negative turning point
        down_cond = (
            (y[i-2] < y[i-1]) & (y[i-1] < y[i]) & (y[i] > y[i+1]) & (y[i+1] > y[i+2])
        )

        # Convert to int
        rec = jnp.where(rec_cond, 1, 0)
        sev = jnp.where(sev_cond, 1, 0)
        up = jnp.where(up_cond, 1, 0)
        down = jnp.where(down_cond, 1, 0)
        return carry, (rec, sev, up, down)

    _, (rec_seq, sev_seq, up_seq, down_seq) = lax.scan(step, None, jnp.arange(3, n-2))

    # Get the time until the first recession
    next_recession = jnp.where(
        jnp.any(rec_seq == 1), jnp.argmax(rec_seq == 1) + 1, len(y)
    )
    next_severe_recession = jnp.where(
        jnp.any(sev_seq == 1), jnp.argmax(sev_seq == 1) + 1, len(y)
    )

    # Minimum value in the next 8 periods
    min_val_8q = jnp.min(future_path[:8])

    # Get the time until the first turning point
    next_up_turn = jnp.where(
        jnp.any(up_seq == 1),
        jnp.maximum(jnp.argmax(up_seq == 1), 1), # Exclude 0 return
        len(y)
    )
    next_down_turn = jnp.where(
        jnp.any(down_seq == 1),
        jnp.maximum(jnp.argmax(down_seq == 1), 1),
        len(y)
    )
    path_stats = (
        next_recession, next_severe_recession, min_val_8q,
        next_up_turn, next_down_turn
        )
```

(continues on next page)

(continued from previous page)

```
    return path_stats
```

The following function creates visualizations of the path statistics in a subplot grid.

```
def plot_path_stats(next_recession, next_severe_recession, min_val_8q,
                    next_up_turn, next_down_turn, ax, label=None):
    """Plot the path statistics in subplots(3,2)"""
    # ax[0, 0] is for paths of y
    sns.histplot(next_recession, kde=True, stat='density', ax=ax[0, 1],
                 alpha=.8, label=label)
    ax[0, 1].set_xlabel("time until the next recession", fontsize=13)

    sns.histplot(next_severe_recession, kde=True, stat='density', ax=ax[1, 0],
                 alpha=.8, label=label)
    ax[1, 0].set_xlabel("time until the next severe recession", fontsize=13)

    sns.histplot(min_val_8q, kde=True, stat='density', ax=ax[1, 1],
                 alpha=.8, label=label)
    ax[1, 1].set_xlabel("minimum value in next 8 periods", fontsize=13)

    sns.histplot(next_up_turn, kde=True, stat='density', ax=ax[2, 0],
                 alpha=.8, label=label)
    ax[2, 0].set_xlabel("time until the next positive turn", fontsize=13)

    sns.histplot(next_down_turn, kde=True, stat='density', ax=ax[2, 1],
                 alpha=.8, label=label)
    ax[2, 1].set_xlabel("time until the next negative turn", fontsize=13)
```

20.7 Original Wecker Method

Now we apply Wecker's original method by simulating future paths and compute predictive distributions, conditioning on the true parameters associated with the data-generating model.

```
def plot_Wecker(ar1: AR1, initial_path, ax, N=1000, label='true parameters'):
    """
    Plot the predictive distributions from "pure" Wecker's method.

    Parameters
    ----------
    ar1 : AR1
        An AR1 named tuple containing the process parameters (ρ, σ, T0, T1).
    initial_path : array-like
        The initial observed path of the AR(1) process.
    N : int
        Number of future sample paths to simulate for predictive distributions.
    label : str
        Legend label for the plotted distributions.
    """
    # Plot simulated initial and future paths
    y_T0 = initial_path[-1]
    future_path = AR1_simulate_future(ar1, y_T0, N=N)
    plot_path(ar1, initial_path, future_path, ax[0, 0])

    next_reces = jnp.zeros(N)
```

(continues on next page)

(continued from previous page)

```
    severe_rec = jnp.zeros(N)
    min_val_8q = jnp.zeros(N)
    next_up_turn = jnp.zeros(N)
    next_down_turn = jnp.zeros(N)

    # Simulate future paths and compute statistics
    for n in range(N):
        future_temp = future_path[n, :]
        (next_reces_val, severe_rec_val, min_val_8q_val,
        next_up_turn_val, next_down_turn_val
         ) = compute_path_statistics(initial_path, future_temp)

        next_reces = next_reces.at[n].set(next_reces_val)
        severe_rec = severe_rec.at[n].set(severe_rec_val)
        min_val_8q = min_val_8q.at[n].set(min_val_8q_val)
        next_up_turn = next_up_turn.at[n].set(next_up_turn_val)
        next_down_turn = next_down_turn.at[n].set(next_down_turn_val)

    # Plot path statistics
    plot_path_stats(next_reces, severe_rec, min_val_8q,
                    next_up_turn, next_down_turn, ax, label=label)


fig, ax = plt.subplots(3, 2, figsize=(15, 12))
plot_Wecker(ar1, initial_path, ax)
plt.show()
```

Apart from the top-left panel, which repeats the initial path and coverage intervals, each panel shows the predictive distribution of one sample path statistic.

These are computed by simulating many future paths with the parameters held fixed at their true values.

The distributions therefore reflect only the uncertainty that comes from future shocks.

20.8 Extended Wecker Method

Now we apply our "extended" Wecker method based on predictive densities of y defined by (20.4) that acknowledge posterior uncertainty in the parameters ρ, σ.

To approximate the integration on the right side of (20.4), we repeatedly draw parameters from the joint posterior distribution each time we simulate a sequence of future values from model (20.1).

```
def plot_extended_Wecker(
    ar1: AR1, post_samples, initial_path, ax, N=1000,
    label='sampling from posterior'
    ):
    """Plot the extended Wecker's predictive distribution"""
    y0, T1 = ar1.y0, ar1.T1
    y_T0 = initial_path[-1]

    # Select a parameter sample
    index = random.choice(
        key, jnp.arange(len(post_samples['ρ'])), (N,), replace=False
```

(continues on next page)

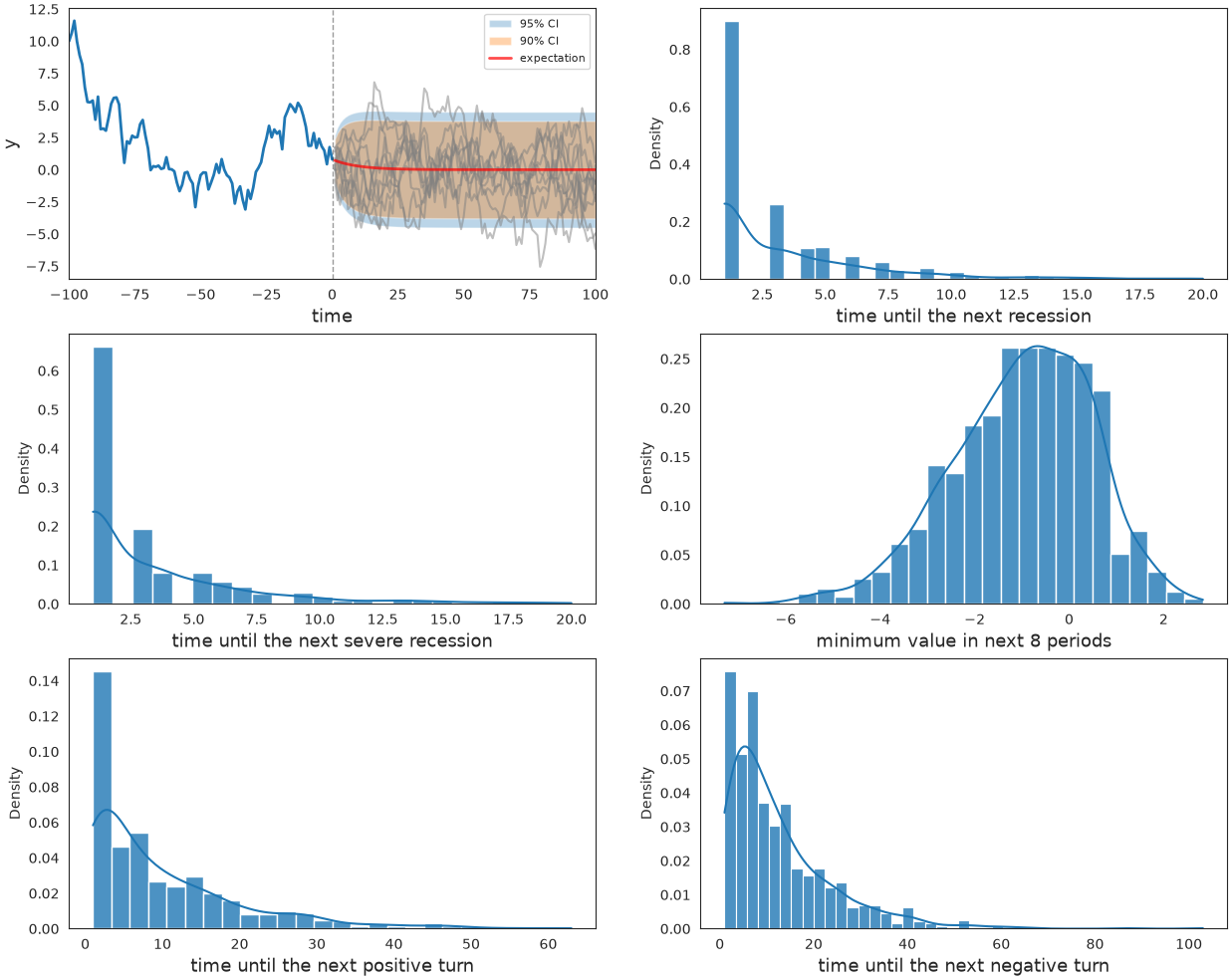


Fig. 20.3: Distributions from Wecker's method

(continued from previous page)

```
        )
    ρ_sample = post_samples['ρ'][index]
    σ_sample = post_samples['σ'][index]

    # Compute path statistics
    next_reces = jnp.zeros(N)
    severe_rec = jnp.zeros(N)
    min_val_8q = jnp.zeros(N)
    next_up_turn = jnp.zeros(N)
    next_down_turn = jnp.zeros(N)

    subkeys = random.split(key, num=N)

    # Simulate one future path per posterior draw
    future_paths = []
    for n in range(N):
        ar1_n = AR1(ρ=ρ_sample[n], σ=σ_sample[n], y0=y0, T1=T1)
        future_temp = AR1_simulate_future(
            ar1_n, y_T0, N=1, key=subkeys[n]
            ).reshape(-1)
        future_paths.append(future_temp)
        (next_reces_val, severe_rec_val, min_val_8q_val,
        next_up_turn_val, next_down_turn_val
         ) = compute_path_statistics(initial_path, future_temp)

        next_reces = next_reces.at[n].set(next_reces_val)
        severe_rec = severe_rec.at[n].set(severe_rec_val)
        min_val_8q = min_val_8q.at[n].set(min_val_8q_val)
        next_up_turn = next_up_turn.at[n].set(next_up_turn_val)
        next_down_turn = next_down_turn.at[n].set(next_down_turn_val)

    # Plot the initial path and the future paths drawn from the posterior
    future_paths = jnp.stack(future_paths)
    plot_path(ar1, initial_path, future_paths, ax[0, 0])

    # Plot path statistics
    plot_path_stats(next_reces, severe_rec, min_val_8q,
                    next_up_turn, next_down_turn, ax, label=label)

fig, ax = plt.subplots(3, 2, figsize=(12, 15))
plot_extended_Wecker(ar1, post_samples, initial_path, ax)
plt.show()
```

The panels show the same statistics, but now each future path is simulated with parameters drawn from the posterior.

These distributions therefore combine two sources of uncertainty: randomness in future shocks and our uncertainty about (ρ, σ).

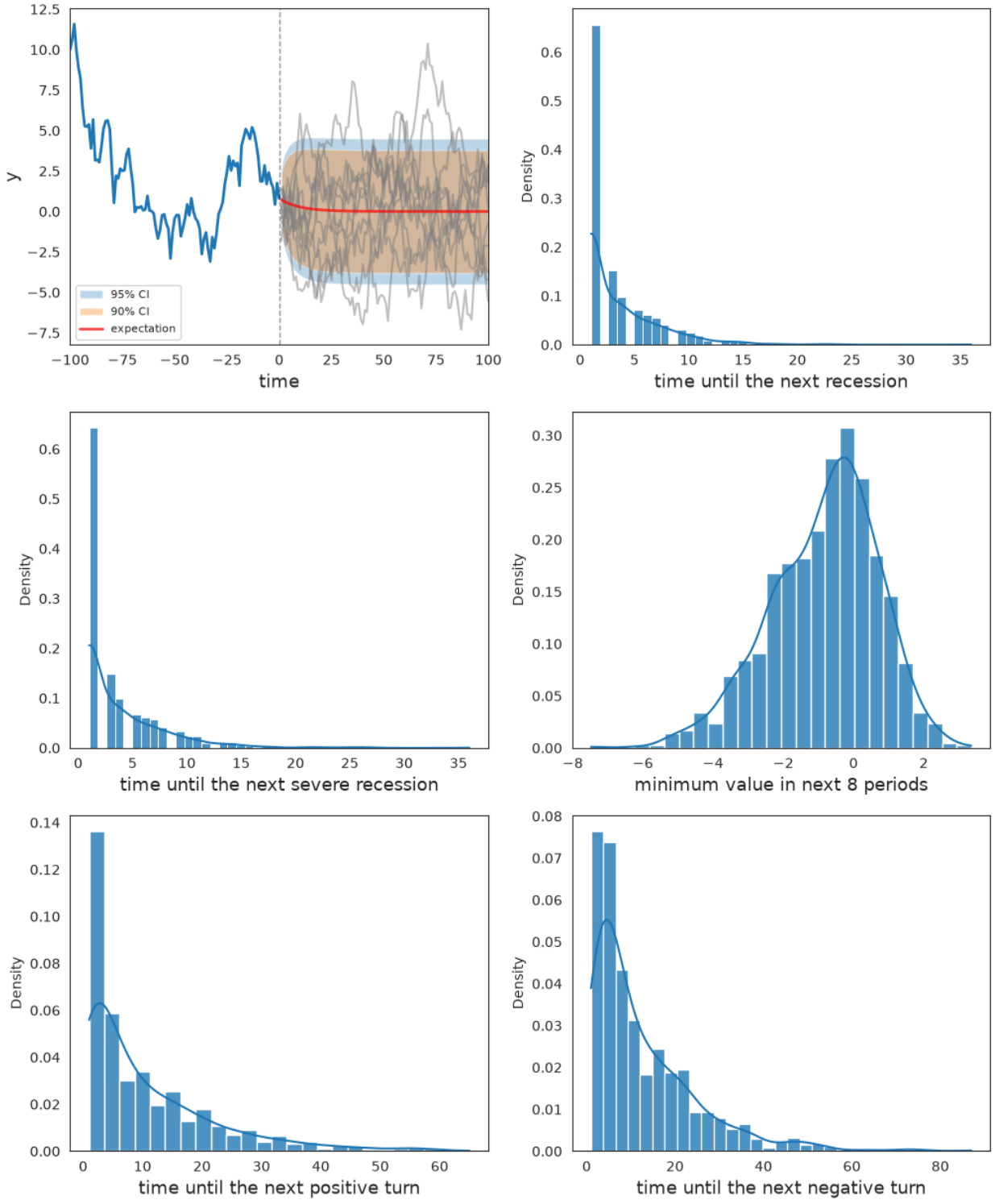


Fig. 20.4: Distributions from extended Wecker's method

20.9 Comparison

Finally, we plot both the original Wecker method and the extended method with parameter values drawn from the posterior together to compare the differences that emerge from pretending to know parameter values when they are actually uncertain.

```
fig, ax = plt.subplots(3, 2, figsize=(12, 15))
plot_Wecker(ar1, initial_path, ax)
ax[0, 0].clear()
plot_extended_Wecker(ar1, post_samples, initial_path, ax)
ax[0, 1].legend(fontsize=10)
plt.show()
```

The two sets of predictive distributions reveal the cost of pretending that we know the parameters.

The extended Wecker method draws (ρ, σ) from their posterior each time it simulates a future path.

It therefore layers parameter uncertainty on top of the shock uncertainty already present in the original method.

As a result, its predictive distributions are more dispersed than those computed with the parameters fixed at their true values.

20.10 Conclusion

This lecture combined two tools to forecast nonlinear functions of the future path of an AR(1) process.

Wecker's simulation method let us approximate predictive distributions of sample path statistics, such as the time until the next turning point.

A Bayesian posterior over (ρ, σ), computed as in *Posterior Distributions for AR(1) Parameters*, let us also account for our uncertainty about the parameters that govern the process.

Putting these together, the extended Wecker method produces predictive distributions that reflect both sources of uncertainty identified at the outset.

Ignoring parameter uncertainty, as the original Wecker method does, yields tighter distributions that overstate how much we really know about these future statistics.

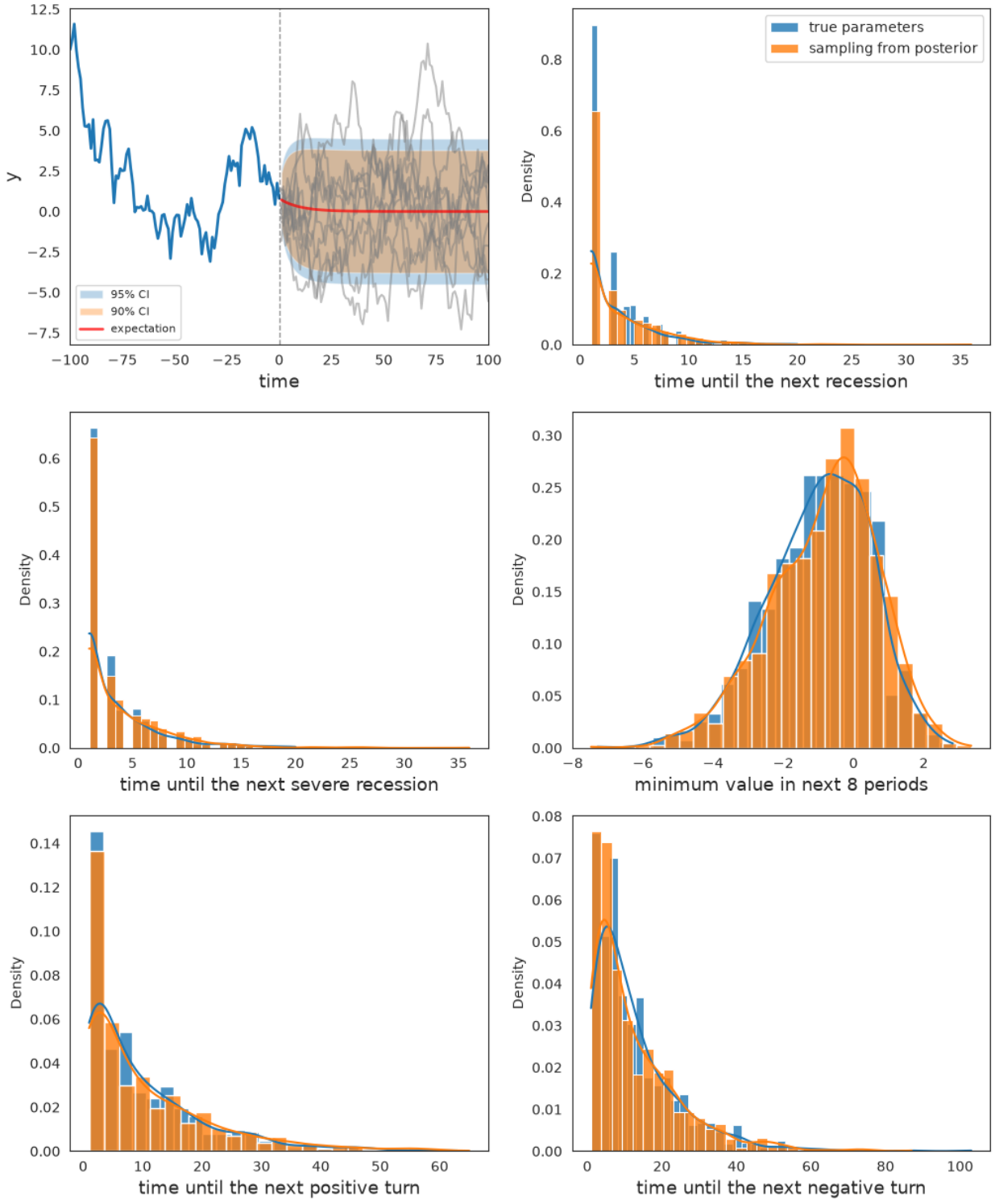


Fig. 20.5: Comparison of the two methods

Part IV

Statistics and Information

CHAPTER

TWENTYONE

STATISTICAL DIVERGENCE MEASURES

Contents

- *Statistical Divergence Measures*
  - *Overview*
  - *Primer on entropy, cross-entropy, KL divergence*
  - *Two Beta distributions: running example*
  - *Kullback–Leibler divergence*
  - *Jensen-Shannon divergence*
  - *Chernoff entropy*
  - *Comparing divergence measures*
  - *KL divergence and maximum-likelihood estimation*
  - *Related lectures*

21.1 Overview

A statistical divergence quantifies discrepancies between two distinct probability distributions that can be challenging to distinguish for the following reason:

- every event that has positive probability under one of the distributions also has positive probability under the other distribution
- this means that there is no "smoking gun" event whose occurrence tells a statistician that one of the probability distributions surely governs the data

A statistical divergence is a **function** that maps two probability distributions into a nonnegative real number.

Statistical divergence functions play important roles in statistics, information theory, and what many people now call "machine learning".

This lecture describes three divergence measures:

- **Kullback–Leibler (KL) divergence**
- **Jensen–Shannon (JS) divergence**
- **Chernoff entropy**

These will appear in several quantecon lectures.

Let's start by importing the necessary Python tools.

```
import matplotlib.pyplot as plt
import numpy as np
from numba import vectorize, jit
from math import gamma
from scipy.integrate import quad
from scipy.optimize import minimize_scalar
import pandas as pd
from IPython.display import display, Math
```

21.2 Primer on entropy, cross-entropy, KL divergence

Before diving in, we'll introduce some useful concepts in a simple setting.

We'll temporarily assume that f and g are two probability mass functions for discrete random variables on state space $I = \{1, 2, \ldots, n\}$ that satisfy $f_i \geq 0, \sum_i f_i = 1, g_i \geq 0, \sum_i g_i = 1$.

We follow some statisticians and information theorists who define the **surprise** or **surprisal** associated with having observed a single draw $x = i$ from distribution f as

$$\log\left(\frac{1}{f_i}\right)$$

They then define the **information** that you can anticipate to gather from observing a single realization as the expected surprisal

$$H(f) = \sum_i f_i \log\left(\frac{1}{f_i}\right).$$

Claude Shannon [Shannon, 1948] called $H(f)$ the **entropy** of distribution f.

Note

By maximizing $H(f)$ with respect to $\{f_1, f_2, \ldots, f_n\}$ subject to $\sum_i f_i = 1$, we can verify that the distribution that maximizes entropy is the uniform distribution $f_i = \frac{1}{n}$. Entropy $H(f)$ for the uniform distribution evidently equals $-\log(n)$.

Kullback and Leibler [Kullback and Leibler, 1951] define the amount of information that a single draw of x provides for distinguishing f from g as the log likelihood ratio

$$\log\frac{f(x)}{g(x)}$$

The following two concepts are widely used to compare two distributions f and g.

Cross-Entropy:

$$H(f, g) = -\sum_i f_i \log g_i \tag{21.1}$$

Kullback-Leibler (KL) Divergence:

$$D_{KL}(f \parallel g) = \sum_i f_i \log\left[\frac{f_i}{g_i}\right] \tag{21.2}$$

These concepts are related by the following equality.

$$D_{KL}(f \parallel g) = H(f,g) - H(f) \tag{21.3}$$

To prove (21.3), note that

$$\begin{aligned} D_{KL}(f \parallel g) &= \sum_i f_i \log \left[\frac{f_i}{g_i}\right] && (21.4)\\ &= \sum_i f_i \left[\log f_i - \log g_i\right] && (21.5)\\ &= \sum_i f_i \log f_i - \sum_i f_i \log g_i && (21.6)\\ &= -H(f) + H(f,g) && (21.7)\\ &= H(f,g) - H(f) && (21.8) \end{aligned}$$

Remember that $H(f)$ is the anticipated surprisal from drawing x from f.

Then the above equation tells us that the KL divergence is an anticipated "excess surprise" that comes from anticipating that x is drawn from f when it is actually drawn from g.

21.3 Two Beta distributions: running example

We'll use Beta distributions extensively to illustrate concepts.

The Beta distribution is particularly convenient as it's defined on $[0, 1]$ and exhibits diverse shapes by appropriately choosing its two parameters.

The density of a Beta distribution with parameters a and b is given by

$$f(z; a, b) = \frac{\Gamma(a+b) z^{a-1} (1-z)^{b-1}}{\Gamma(a)\Gamma(b)} \quad \text{where} \quad \Gamma(p) := \int_0^\infty x^{p-1} e^{-x} dx$$

We introduce two Beta distributions $f(x)$ and $g(x)$, which we will use to illustrate the different divergence measures.

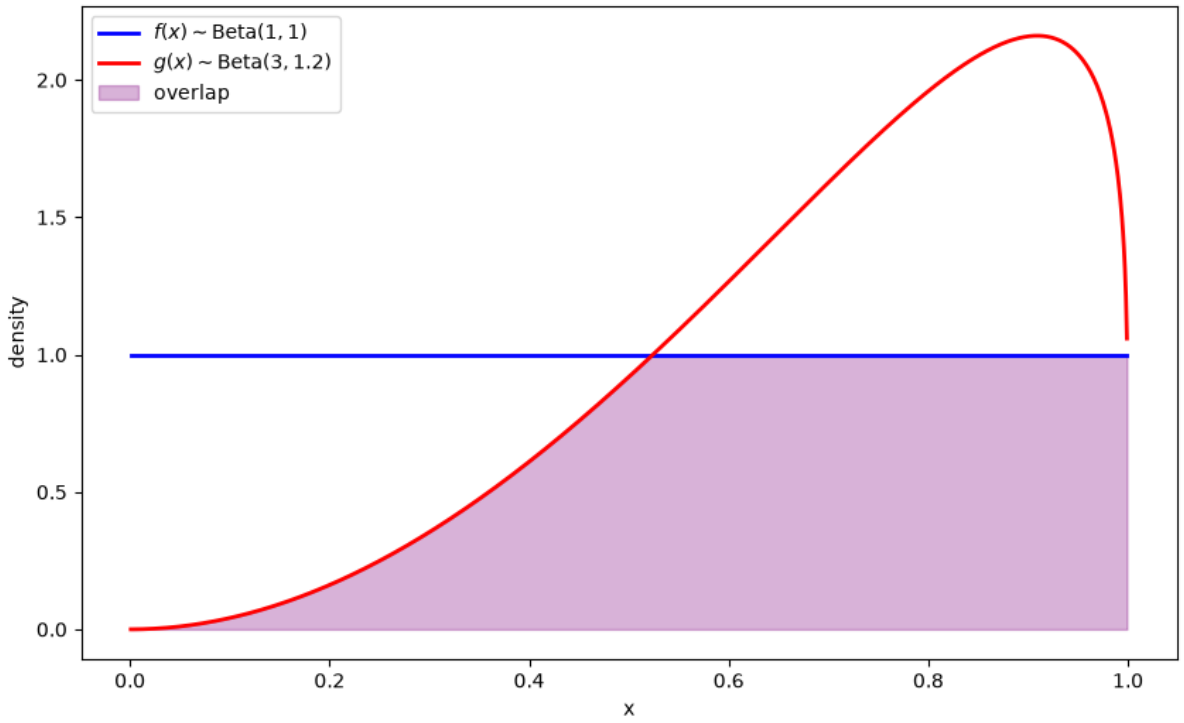
Let's define parameters and density functions in Python

```
# Parameters in the two Beta distributions
F_a, F_b = 1, 1
G_a, G_b = 3, 1.2

@vectorize
def p(x, a, b):
    r = gamma(a + b) / (gamma(a) * gamma(b))
    return r * x** (a-1) * (1 - x) ** (b-1)

# The two density functions
f = jit(lambda x: p(x, F_a, F_b))
g = jit(lambda x: p(x, G_a, G_b))

# Plot the distributions
x_range = np.linspace(0.001, 0.999, 1000)
f_vals = [f(x) for x in x_range]
g_vals = [g(x) for x in x_range]

plt.figure(figsize=(10, 6))
```

(continues on next page)

(continued from previous page)

```
plt.plot(x_range, f_vals, 'b-', linewidth=2, label=r'$f(x) \sim \text{Beta}(1,1)$')
plt.plot(x_range, g_vals, 'r-', linewidth=2, label=r'$g(x) \sim \text{Beta}(3,1.2)$')

# Fill overlap region
overlap = np.minimum(f_vals, g_vals)
plt.fill_between(x_range, 0, overlap, alpha=0.3, color='purple', label='overlap')

plt.xlabel('x')
plt.ylabel('density')
plt.legend()
plt.show()
```



21.4 Kullback–Leibler divergence

Our first divergence function is the **Kullback–Leibler (KL) divergence**.

For probability densities (or pmfs) f and g it is defined by

$$D_{KL}(f\|g) = KL(f,g) = \int f(x) \log \frac{f(x)}{g(x)}\, dx.$$

We can interpret $D_{KL}(f\|g)$ as the expected excess log loss (expected excess surprisal) incurred when we use g while the data are generated by f.

It has several important properties:

- Non-negativity (Gibbs' inequality): $D_{KL}(f\|g) \geq 0$ with equality if and only if $f = g$ almost everywhere.
- Asymmetry: $D_{KL}(f\|g) \neq D_{KL}(g\|f)$ in general (hence it is not a metric)

- Information decomposition: $D_{KL}(f\|g) = H(f,g) - H(f)$, where $H(f,g)$ is the cross entropy and $H(f)$ is the Shannon entropy of f.
- Chain rule: For joint distributions $f(x,y)$ and $g(x,y)$, $D_{KL}(f(x,y)\|g(x,y)) = D_{KL}(f(x)\|g(x)) + E_f\left[D_{KL}(f(y|x)\|g(y|x))\right]$

KL divergence plays a central role in statistical inference, including model selection and hypothesis testing.

Likelihood Ratio Processes describes a link between KL divergence and the expected log likelihood ratio, and the lecture *A Problem that Stumped Milton Friedman* connects it to the test performance of the sequential probability ratio test.

Let's compute the KL divergence between our example distributions f and g.

```
def compute_KL(f, g):
    """
    Compute KL divergence KL(f, g) via numerical integration
    """
    def integrand(w):
        fw = f(w)
        gw = g(w)
        return fw * np.log(fw / gw)
    val, _ = quad(integrand, 1e-5, 1-1e-5)
    return val

# Compute KL divergences between our example distributions
kl_fg = compute_KL(f, g)
kl_gf = compute_KL(g, f)

print(f"KL(f, g) = {kl_fg:.4f}")
print(f"KL(g, f) = {kl_gf:.4f}")
```

```
KL(f, g) = 0.7590
KL(g, f) = 0.3436
```

The asymmetry of KL divergence has important practical implications.

$D_{KL}(f\|g)$ penalizes regions where $f > 0$ but g is close to zero, reflecting the cost of using g to model f and vice versa.

21.5 Jensen-Shannon divergence

Sometimes we want a symmetric measure of divergence that captures the difference between two distributions without favoring one over the other.

This often arises in applications like clustering, where we want to compare distributions without assuming one is the true model.

The **Jensen-Shannon (JS) divergence** symmetrizes KL divergence by comparing both distributions to their mixture:

$$JS(f,g) = \frac{1}{2}D_{KL}(f\|m) + \frac{1}{2}D_{KL}(g\|m), \quad m = \frac{1}{2}(f+g).$$

where m is a mixture distribution that averages f and g

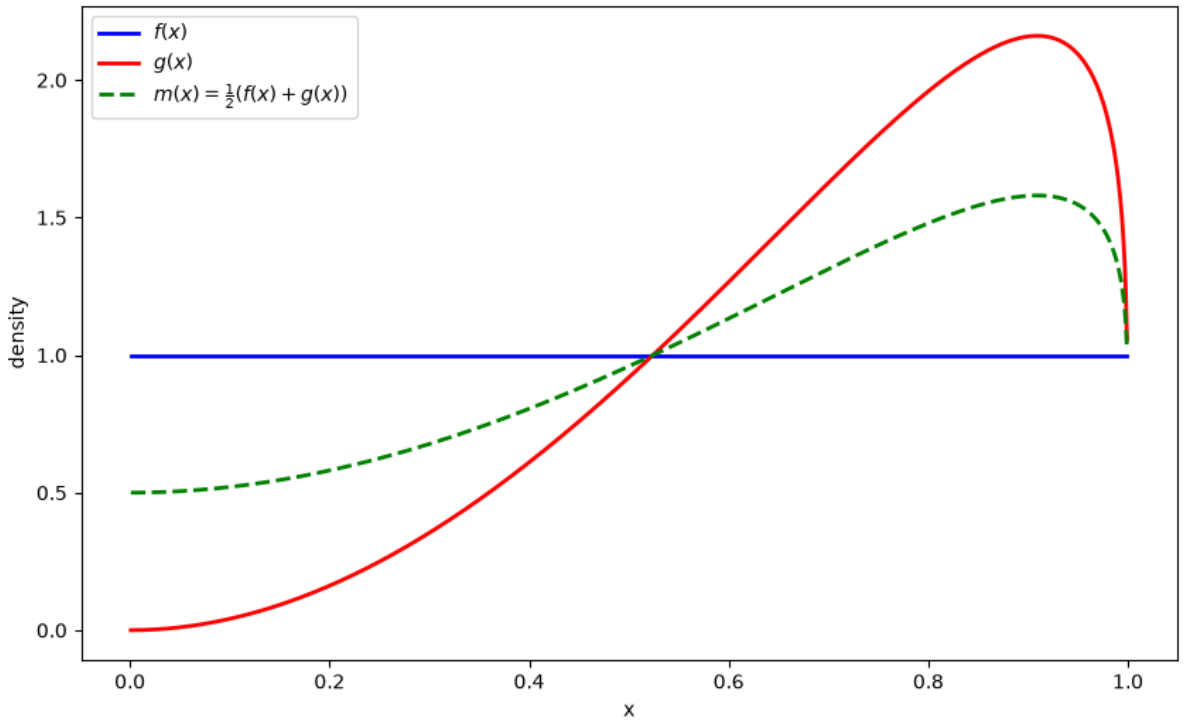
Let's also visualize the mixture distribution m:

```
def m(x):
    return 0.5 * (f(x) + g(x))
```

(continues on next page)

(continued from previous page)

```
m_vals = [m(x) for x in x_range]

plt.figure(figsize=(10, 6))
plt.plot(x_range, f_vals, 'b-', linewidth=2, label=r'$f(x)$')
plt.plot(x_range, g_vals, 'r-', linewidth=2, label=r'$g(x)$')
plt.plot(x_range, m_vals, 'g--', linewidth=2, label=r'$m(x) = \frac{1}{2}(f(x) + g(x))
↪$')

plt.xlabel('x')
plt.ylabel('density')
plt.legend()
plt.show()
```



The JS divergence has several useful properties:

- Symmetry: $JS(f, g) = JS(g, f)$.
- Boundedness: $0 \leq JS(f, g) \leq \log 2$.
- Its square root $\sqrt{JS}$ is a metric (Jensen–Shannon distance) on the space of probability distributions.
- JS divergence equals the mutual information between a binary random variable $Z \sim \text{Bernoulli}(1/2)$ indicating the source and a sample X drawn from f if $Z = 0$ or from g if $Z = 1$.

The Jensen–Shannon divergence plays a key role in the optimization of certain generative models, as it is bounded, symmetric, and smoother than KL divergence, often providing more stable gradients for training.

Let's compute the JS divergence between our example distributions f and g

```
def compute_JS(f, g):
    """Compute Jensen-Shannon divergence."""
    def m(w):
        return 0.5 * (f(w) + g(w))
```

(continues on next page)

(continued from previous page)

```
    js_div = 0.5 * compute_KL(f, m) + 0.5 * compute_KL(g, m)
    return js_div

js_div = compute_JS(f, g)
print(f"Jensen-Shannon divergence JS(f,g) = {js_div:.4f}")
```

```
Jensen-Shannon divergence JS(f,g) = 0.0984
```

We can easily generalize to more than two distributions using the generalized Jensen-Shannon divergence with weights $\alpha = (\alpha_i)_{i=1}^n$:

$$JS_\alpha(f_1, \ldots, f_n) = H\left(\sum_{i=1}^n \alpha_i f_i\right) - \sum_{i=1}^n \alpha_i H(f_i)$$

where:

- $\alpha_i \geq 0$ and $\sum_{i=1}^n \alpha_i = 1$, and
- $H(f) = -\int f(x) \log f(x) dx$ is the **Shannon entropy** of distribution f

21.6 Chernoff entropy

Chernoff entropy originates from early applications of the theory of large deviations, which refines central limit approximations by providing exponential decay rates for rare events.

For densities f and g the Chernoff entropy is

$$C(f, g) = -\log \min_{\phi \in (0,1)} \int f^\phi(x) g^{1-\phi}(x)\, dx.$$

Remarks:

- The inner integral is the **Chernoff coefficient**.
- At $\phi = 1/2$ it becomes the **Bhattacharyya coefficient** $\int \sqrt{fg}$.
- In binary hypothesis testing with T iid observations, the optimal error probability decays as $e^{-C(f,g)T}$.

We will see an example of the third point in the lecture *Likelihood Ratio Processes*, where we study the Chernoff entropy in the context of model selection.

Let's compute the Chernoff entropy between our example distributions f and g.

```
def chernoff_integrand(ϕ, f, g):
    """Integral entering Chernoff entropy for a given ϕ."""
    def integrand(w):
        return f(w)**ϕ * g(w)**(1-ϕ)
    result, _ = quad(integrand, 1e-5, 1-1e-5)
    return result

def compute_chernoff_entropy(f, g):
    """Compute Chernoff entropy C(f,g)."""
    def objective(ϕ):
        return chernoff_integrand(ϕ, f, g)
    result = minimize_scalar(objective, bounds=(1e-5, 1-1e-5), method='bounded')
    min_value = result.fun
```

(continues on next page)

(continued from previous page)

```
    ϕ_optimal = result.x
    chernoff_entropy = -np.log(min_value)
    return chernoff_entropy, ϕ_optimal

C_fg, ϕ_optimal = compute_chernoff_entropy(f, g)
print(f"Chernoff entropy C(f,g) = {C_fg:.4f}")
print(f"Optimal ϕ = {ϕ_optimal:.4f}")
```

```
Chernoff entropy C(f,g) = 0.1212
Optimal ϕ = 0.5969
```

21.7 Comparing divergence measures

We now compare these measures across several pairs of Beta distributions

Pair (f, g)	KL(f, g)	KL(g, f)	JS	C
$\text{Beta}(1,1), \text{Beta}(1.1, 1.05)$	0.0028	0.0026	0.0007	0.0007
$\text{Beta}(1,1), \text{Beta}(1.2, 1.1)$	0.0105	0.0092	0.0024	0.0025
$\text{Beta}(1,1), \text{Beta}(0.9, 0.8)$	0.0143	0.0166	0.0038	0.0039
$\text{Beta}(1,1), \text{Beta}(1.5, 1.2)$	0.0589	0.0437	0.0121	0.0126
$\text{Beta}(1,1), \text{Beta}(0.7, 0.6)$	0.0673	0.0924	0.0186	0.0201
$\text{Beta}(1,1), \text{Beta}(2, 1.5)$	0.1781	0.1081	0.0309	0.0339
$\text{Beta}(1,1), \text{Beta}(0.5, 0.5)$	0.1448	0.2190	0.0400	0.0461
$\text{Beta}(1,1), \text{Beta}(2.5, 1.8)$	0.3323	0.1731	0.0502	0.0577
$\text{Beta}(1,1), \text{Beta}(0.3, 0.4)$	0.3317	0.5572	0.0869	0.1203
$\text{Beta}(1,1), \text{Beta}(3, 1.2)$	0.7590	0.3436	0.0984	0.1212
$\text{Beta}(1,1), \text{Beta}(0.3, 0.3)$	0.3935	0.6516	0.1008	0.1456
$\text{Beta}(1,1), \text{Beta}(4, 1)$	1.6134	0.6362	0.1733	0.2341
$\text{Beta}(1,1), \text{Beta}(0.1, 0.2)$	0.9811	1.0036	0.1783	0.4556
$\text{Beta}(1,1), \text{Beta}(5, 1)$	2.3901	0.8094	0.2162	0.3128

We can clearly see co-movement across the divergence measures as we vary the parameters of the Beta distributions.

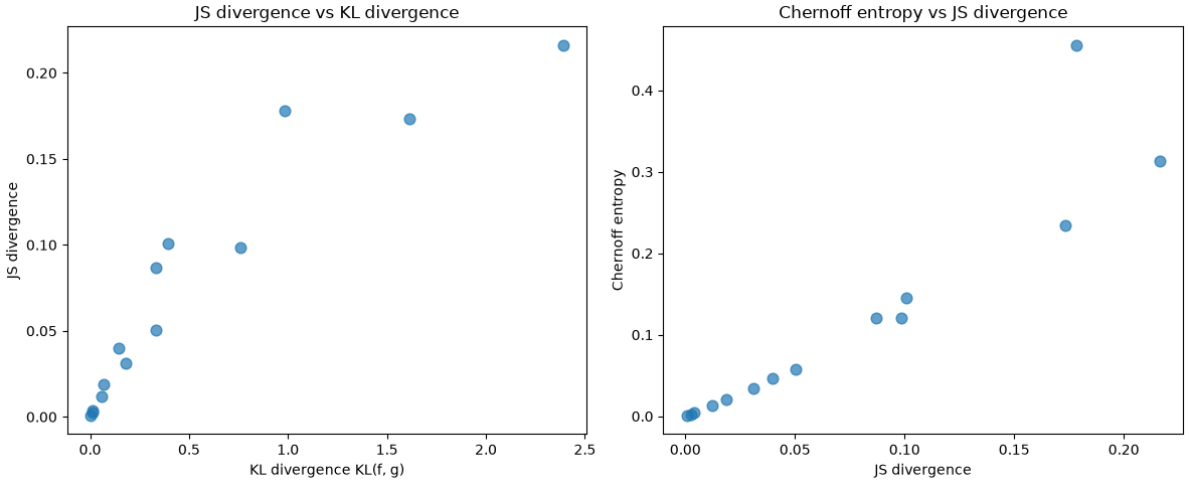
Next we visualize relationships among KL, JS, and Chernoff entropy.

```
kl_fg_values = [float(result['KL(f, g)']) for result in results]
js_values = [float(result['JS']) for result in results]
chernoff_values = [float(result['C']) for result in results]

fig, axes = plt.subplots(1, 2, figsize=(12, 5))

axes[0].scatter(kl_fg_values, js_values, alpha=0.7, s=60)
axes[0].set_xlabel('KL divergence KL(f, g)')
axes[0].set_ylabel('JS divergence')
axes[0].set_title('JS divergence vs KL divergence')

axes[1].scatter(js_values, chernoff_values, alpha=0.7, s=60)
axes[1].set_xlabel('JS divergence')
axes[1].set_ylabel('Chernoff entropy')
axes[1].set_title('Chernoff entropy vs JS divergence')
```

(continues on next page)

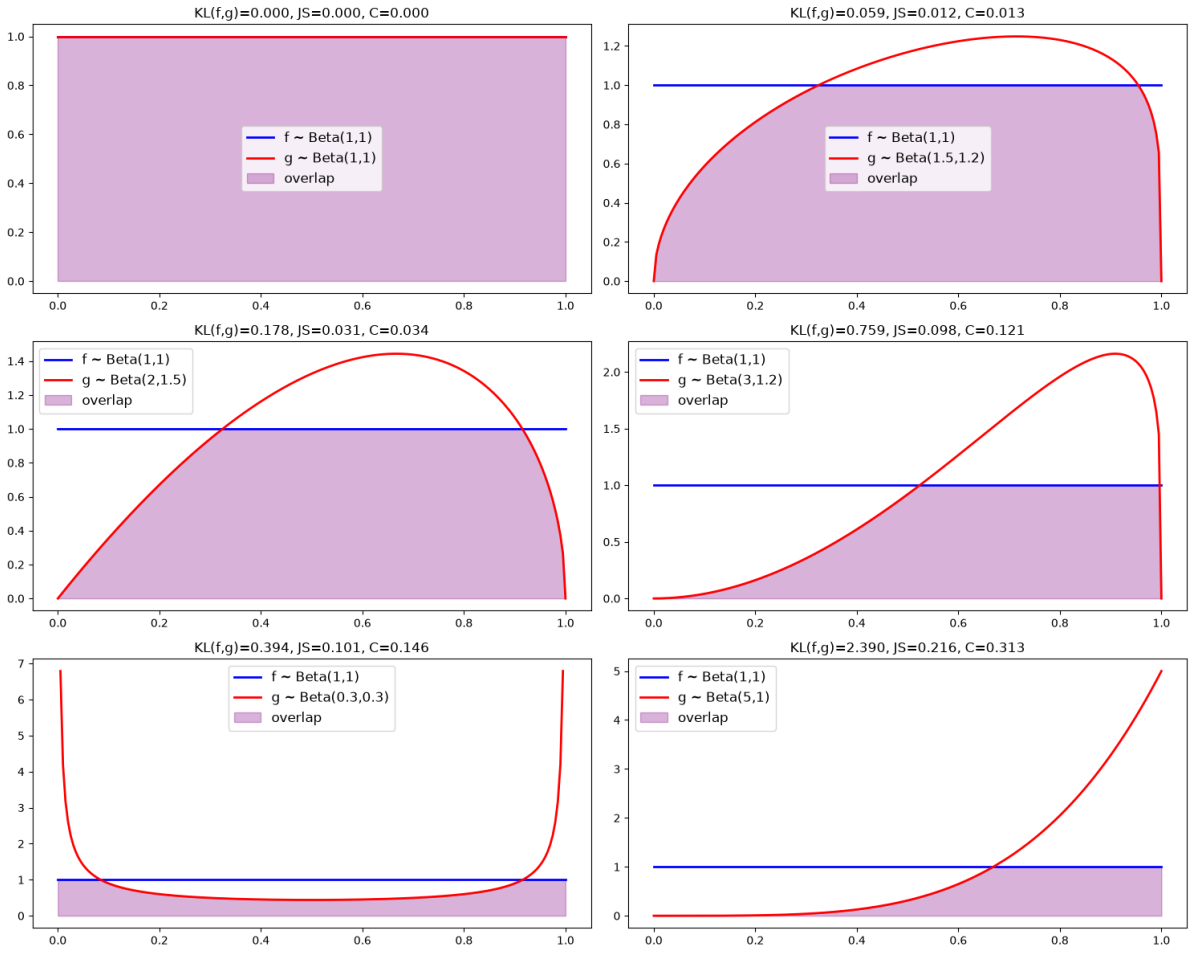
(continued from previous page)

```
plt.tight_layout()
plt.show()
```



We now generate plots illustrating how overlap visually diminishes as divergence measures increase.

```
param_grid = [
    ((1, 1), (1, 1)),
    ((1, 1), (1.5, 1.2)),
    ((1, 1), (2, 1.5)),
    ((1, 1), (3, 1.2)),
    ((1, 1), (0.3, 0.3)),
    ((1, 1), (5, 1))
]
```



21.8 KL divergence and maximum-likelihood estimation

Given a sample of n observations $X = \{x_1, x_2, \dots, x_n\}$, the **empirical distribution** is

$$p_e(x) = \frac{1}{n} \sum_{i=1}^{n} \delta(x - x_i)$$

where $\delta(x - x_i)$ is the Dirac delta function centered at x_i:

$$\delta(x - x_i) = \begin{cases} +\infty & \text{if } x = x_i \\ 0 & \text{if } x \neq x_i \end{cases}$$

- **Discrete probability measure**: Assigns probability $\frac{1}{n}$ to each observed data point
- **Empirical expectation**: $\langle X \rangle_{p_e} = \frac{1}{n} \sum_{i=1}^{n} x_i = \bar{\mu}$
- **Support**: Only on the observed data points $\{x_1, x_2, \dots, x_n\}$

The KL divergence from the empirical distribution p_e to a parametric model $p_\theta(x)$ is:

$$D_{KL}(p_e \parallel p_\theta) = \int p_e(x) \log \frac{p_e(x)}{p_\theta(x)} dx$$

Using the mathematics of the Dirac delta function, it follows that

$$D_{KL}(p_e \parallel p_\theta) = \sum_{i=1}^{n} \frac{1}{n} \log \frac{\left(\frac{1}{n}\right)}{p_\theta(x_i)}$$

$$= \frac{1}{n} \sum_{i=1}^{n} \log \frac{1}{n} - \frac{1}{n} \sum_{i=1}^{n} \log p_\theta(x_i)$$

$$= -\log n - \frac{1}{n} \sum_{i=1}^{n} \log p_\theta(x_i)$$

Since the log-likelihood function for parameter θ is:

$$\ell(\theta; X) = \sum_{i=1}^{n} \log p_\theta(x_i),$$

it follows that maximum likelihood chooses parameters to minimize

$$D_{KL}(p_e \parallel p_\theta)$$

Thus, MLE is equivalent to minimizing the KL divergence from the empirical distribution to the statistical model p_θ.

21.9 Related lectures

This lecture has introduced tools that we'll encounter elsewhere.

- Other quantecon lectures that apply connections between divergence measures and statistical inference include *Likelihood Ratio Processes*, *A Problem that Stumped Milton Friedman*, and *Incorrect Models*.
- Statistical divergence functions also take center stage in *Heterogeneous Beliefs and Financial Markets* that studies Lawrence Blume and David Easley's model of heterogeneous beliefs and financial markets.

CHAPTER

TWENTYTWO

LIKELIHOOD RATIO PROCESSES

Contents

- *Likelihood Ratio Processes*
  - *Overview*
  - *Likelihood Ratio Process*
  - *Nature permanently draws from density g*
  - *Peculiar property*
  - *Nature permanently draws from density f*
  - *Likelihood ratio test*
  - *Hypothesis testing and classification*
  - *Markov chains*
  - *Related lectures*
  - *Exercises*

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install --upgrade quantecon
```

22.1 Overview

This lecture describes likelihood ratio processes and some of their uses.

We'll study the same setting that is also used in *this lecture on exchangeability*.

Among the things that we'll learn are

- How a likelihood ratio process is a key ingredient in frequentist hypothesis testing
- How a **receiver operator characteristic curve** summarizes information about a false alarm probability and power in frequentist hypothesis testing
- How a statistician can combine frequentist probabilities of type I and type II errors to form posterior probabilities of mistakes in a model selection or in an individual-classification problem

- How to use a Kullback-Leibler divergence to quantify the difference between two probability distributions with the same support
- How during World War II the United States Navy devised a decision rule for doing quality control on lots of ammunition, a topic that sets the stage for *this lecture*
- A peculiar property of likelihood ratio processes

Let's start by importing some Python tools.

```
import matplotlib.pyplot as plt
import numpy as np
from numba import vectorize, jit
from math import gamma
from scipy.integrate import quad
from scipy.optimize import brentq, minimize_scalar
from scipy.stats import beta as beta_dist
import pandas as pd
from IPython.display import display, Math
import quantecon as qe
```

22.2 Likelihood Ratio Process

A nonnegative random variable W has one of two probability density functions, either f or g.

Before the beginning of time, nature once and for all decides whether she will draw a sequence of IID draws from either f or g.

We will sometimes let q be the density that nature chose once and for all, so that q is either f or g, permanently.

Nature knows which density it permanently draws from, but we the observers do not.

We know both f and g but we don't know which density nature chose.

But we want to know.

To do that, we use observations.

We observe a sequence $\{w_t\}_{t=1}^T$ of T IID draws that we know came from either f or g.

We want to use these observations to infer whether nature chose f or g.

A **likelihood ratio process** is a useful tool for this task.

To begin, we define a key component of a likelihood ratio process, namely, the time t likelihood ratio as the random variable

$$\ell(w_t) = \frac{f(w_t)}{g(w_t)}, \quad t \geq 1.$$

We assume that f and g both put positive probabilities on the same intervals of possible realizations of the random variable W.

That means that under the g density, $\ell(w_t) = \frac{f(w_t)}{g(w_t)}$ is a nonnegative random variable with mean 1.

A **likelihood ratio process** for sequence $\{w_t\}_{t=1}^\infty$ is defined as

$$L(w^t) = \prod_{i=1}^t \ell(w_i),$$

where $w^t = \{w_1, \ldots, w_t\}$ is a history of observations up to and including time t.

Sometimes for shorthand we'll write $L_t = L(w^t)$.

Notice that the likelihood process satisfies the *recursion*

$$L(w^t) = \ell(w_t) L(w^{t-1}).$$

The likelihood ratio and its logarithm are key tools for making inferences using a classic frequentist approach due to Neyman and Pearson [Neyman and Pearson, 1933].

To help us appreciate how things work, the following Python code evaluates f and g as two different Beta distributions, then computes and simulates an associated likelihood ratio process by generating a sequence w^t from one of the two probability distributions, for example, a sequence of IID draws from g.

```
# Parameters for the two Beta distributions
F_a, F_b = 1, 1
G_a, G_b = 3, 1.2

@vectorize
def p(x, a, b):
    """Beta distribution density function."""
    r = gamma(a + b) / (gamma(a) * gamma(b))
    return r * x** (a-1) * (1 - x) ** (b-1)

f = jit(lambda x: p(x, F_a, F_b))
g = jit(lambda x: p(x, G_a, G_b))

def create_beta_density(a, b):
    """Create a beta density function with specified parameters."""
    return jit(lambda x: p(x, a, b))

def likelihood_ratio(w, f_func, g_func):
    """Compute likelihood ratio for observation(s) w."""
    return f_func(w) / g_func(w)

@jit
def simulate_likelihood_ratios(a, b, f_func, g_func, T=50, N=500):
    """
    Generate N sets of T observations of the likelihood ratio.
    """
    l_arr = np.empty((N, T))
    for i in range(N):
        for j in range(T):
            w = np.random.beta(a, b)
            l_arr[i, j] = f_func(w) / g_func(w)
    return l_arr

def simulate_sequences(distribution, f_func, g_func,
        F_params=(1, 1), G_params=(3, 1.2), T=50, N=500):
    """
    Generate N sequences of T observations from specified distribution.
    """
    if distribution == 'f':
        a, b = F_params
    elif distribution == 'g':
        a, b = G_params
    else:
        raise ValueError("distribution must be 'f' or 'g'")

    l_arr = simulate_likelihood_ratios(a, b, f_func, g_func, T, N)
```

(continues on next page)

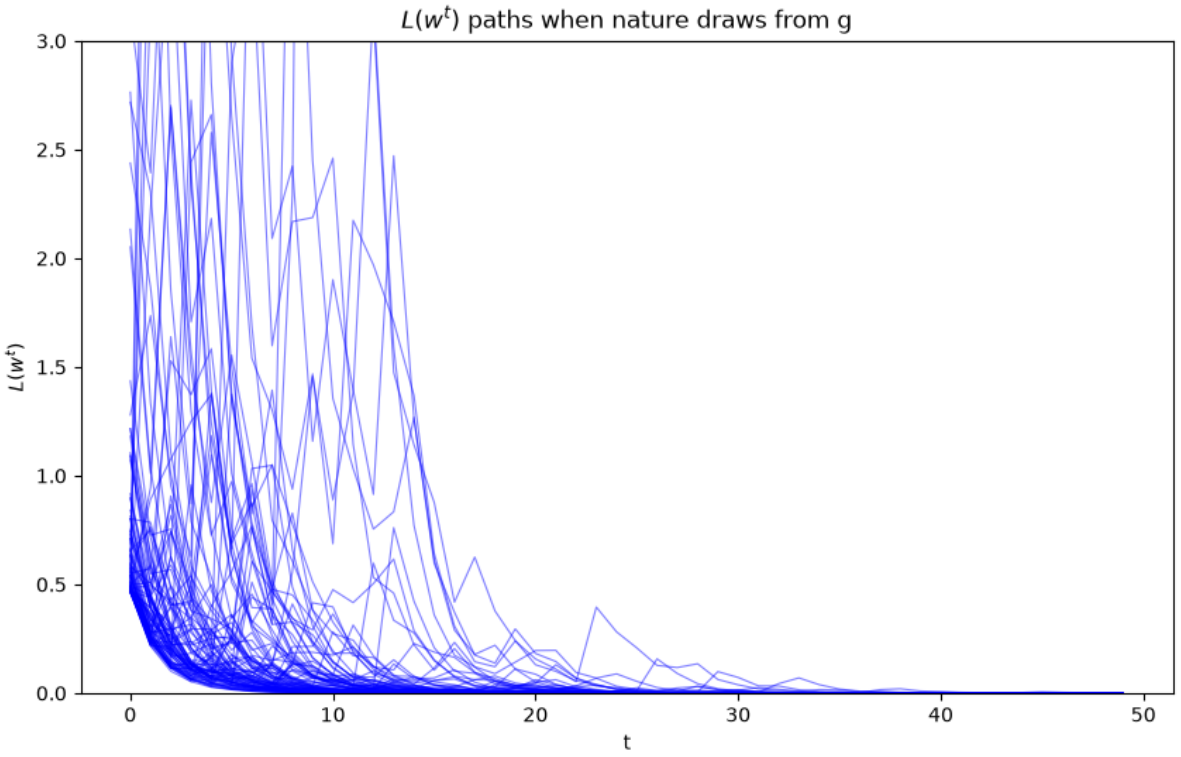
(continued from previous page)

```
    l_seq = np.cumprod(l_arr, axis=1)
    return l_arr, l_seq

def plot_likelihood_paths(l_seq, title="Likelihood ratio paths",
                          ylim=None, n_paths=None):
    """Plot likelihood ratio paths."""
    N, T = l_seq.shape
    n_show = n_paths or min(N, 100)

    plt.figure(figsize=(10, 6))
    for i in range(n_show):
        plt.plot(range(T), l_seq[i, :], color='b', lw=0.8, alpha=0.5)

    if ylim:
        plt.ylim(ylim)
    plt.title(title)
    plt.xlabel('t')
    plt.ylabel('$L(w^t)$')
    plt.show()
```

22.3 Nature permanently draws from density g

We first simulate the likelihood ratio process when nature permanently draws from g.

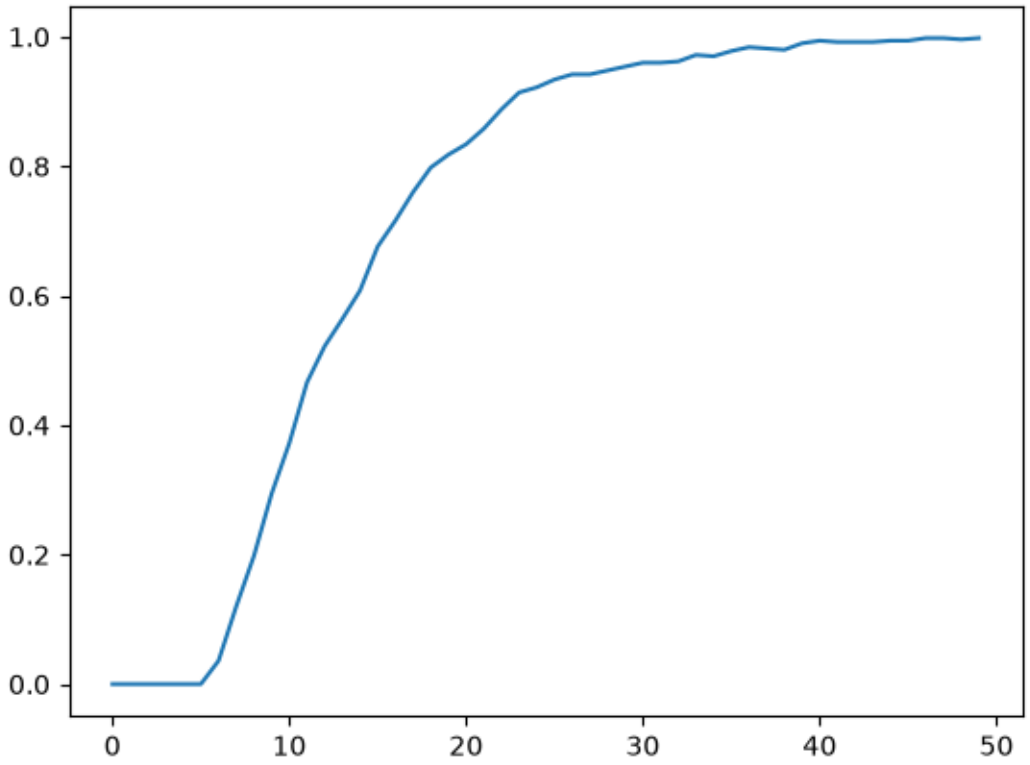
```
# Simulate when nature draws from g
l_arr_g, l_seq_g = simulate_sequences('g', f, g, (F_a, F_b), (G_a, G_b))
plot_likelihood_paths(l_seq_g,
                      title="$L(w^{t})$ paths when nature draws from g",
                      ylim=[0, 3])
```



Evidently, as sample length T grows, most probability mass shifts toward zero

To see this more clearly, we plot over time the fraction of paths $L\left(w^t\right)$ that fall in the interval $[0, 0.01]$.

```
N, T = l_arr_g.shape
plt.plot(range(T), np.sum(l_seq_g <= 0.01, axis=0) / N)
plt.show()
```



Despite the evident convergence of most probability mass to a very small interval near 0, the unconditional mean of $L(w^t)$ under probability density g is identically 1 for all t.

To verify this assertion, first notice that as mentioned earlier the unconditional mean $E[\ell(w_t) \mid q = g]$ is 1 for all t:

$$\begin{aligned} E[\ell(w_t) \mid q = g] &= \int \frac{f(w_t)}{g(w_t)} g(w_t)\, dw_t \\ &= \int f(w_t)\, dw_t \\ &= 1, \end{aligned}$$

which immediately implies

$$\begin{aligned} E[L(w^1) \mid q = g] &= E[\ell(w_1) \mid q = g] \\ &= 1. \end{aligned}$$

Because $L(w^t) = \ell(w_t) L(w^{t-1})$ and $\{w_t\}_{t=1}^t$ is an IID sequence, we have

$$\begin{aligned} E[L(w^t) \mid q = g] &= E[L(w^{t-1})\, \ell(w_t) \mid q = g] \\ &= E[L(w^{t-1})\, E[\ell(w_t) \mid q = g, w^{t-1}] \mid q = g] \\ &= E[L(w^{t-1})\, E[\ell(w_t) \mid q = g] \mid q = g] \\ &= E[L(w^{t-1}) \mid q = g] \end{aligned}$$

for any $t \geq 1$.

Mathematical induction implies $E[L(w^t) \mid q = g] = 1$ for all $t \geq 1$.

22.4 Peculiar property

How can $E\left[L\left(w^t\right) \mid q = g\right] = 1$ possibly be true when most probability mass of the likelihood ratio process is piling up near 0 as $t \to +\infty$?

The answer is that as $t \to +\infty$, the distribution of L_t becomes more and more fat-tailed: enough mass shifts to larger and larger values of L_t to make the mean of L_t continue to be one despite most of the probability mass piling up near 0.

To illustrate this peculiar property, we simulate many paths and calculate the unconditional mean of $L\left(w^t\right)$ by averaging across these many paths at each t.

```
l_arr_g, l_seq_g = simulate_sequences('g',
                f, g, (F_a, F_b), (G_a, G_b), N=50000)
```

It would be useful to use simulations to verify that unconditional means $E\left[L\left(w^t\right)\right]$ equal unity by averaging across sample paths.

But it would be too computer-time-consuming for us to do that here simply by applying a standard Monte Carlo simulation approach.

The reason is that the distribution of $L\left(w^t\right)$ is extremely skewed for large values of t.

Because the probability density in the right tail is close to 0, it just takes too much computer time to sample enough points from the right tail.

We explain the problem in more detail in *this lecture*.

There we describe an alternative way to compute the mean of a likelihood ratio by computing the mean of a *different* random variable by sampling from a *different* probability distribution.

22.5 Nature permanently draws from density f

Now suppose that before time 0 nature permanently decided to draw repeatedly from density f.

While the mean of the likelihood ratio $\ell\left(w_t\right)$ under density g is 1, its mean under the density f exceeds one.

To see this, we compute

$$
\begin{aligned}
E\left[\ell\left(w_t\right) \mid q = f\right] &= \int \frac{f\left(w_t\right)}{g\left(w_t\right)} f\left(w_t\right) dw_t \\
&= \int \frac{f\left(w_t\right)}{g\left(w_t\right)} \frac{f\left(w_t\right)}{g\left(w_t\right)} g\left(w_t\right) dw_t \\
&= \int \ell\left(w_t\right)^2 g\left(w_t\right) dw_t \\
&= E\left[\ell\left(w_t\right)^2 \mid q = g\right] \\
&= E\left[\ell\left(w_t\right) \mid q = g\right]^2 + Var\left(\ell\left(w_t\right) \mid q = g\right) \\
&> E\left[\ell\left(w_t\right) \mid q = g\right]^2 = 1
\end{aligned}
$$

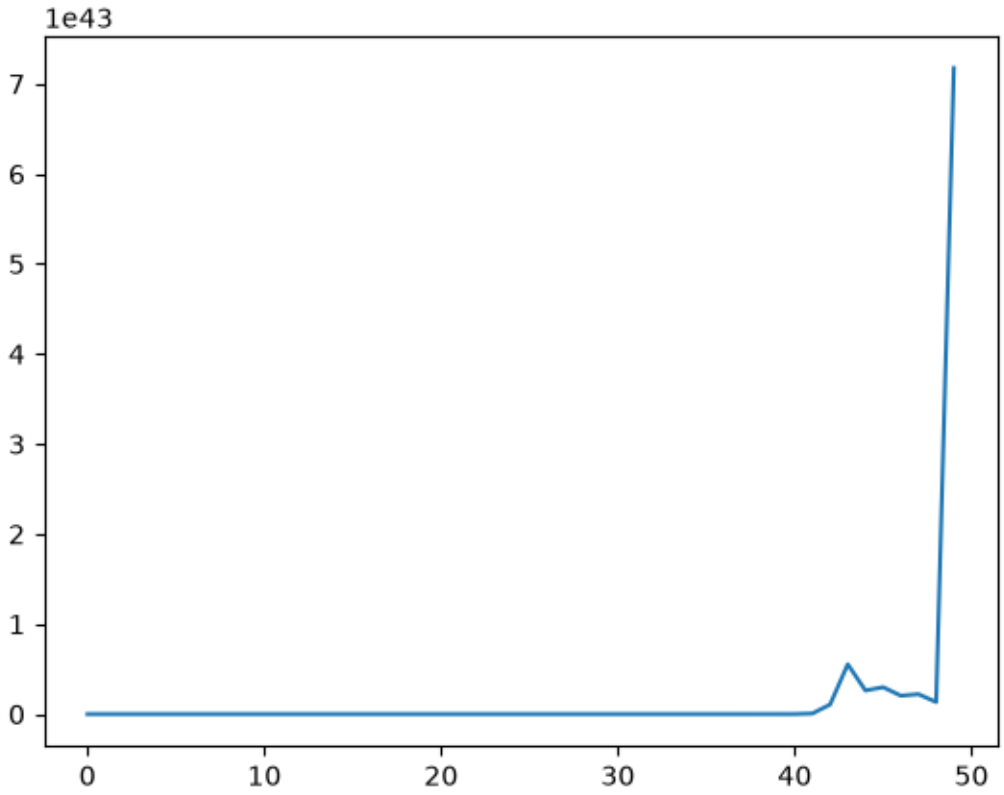
This in turn implies that the unconditional mean of the likelihood ratio process $L(w^t)$ diverges toward $+\infty$.

Simulations below confirm this conclusion.

Please note the scale of the y axis.

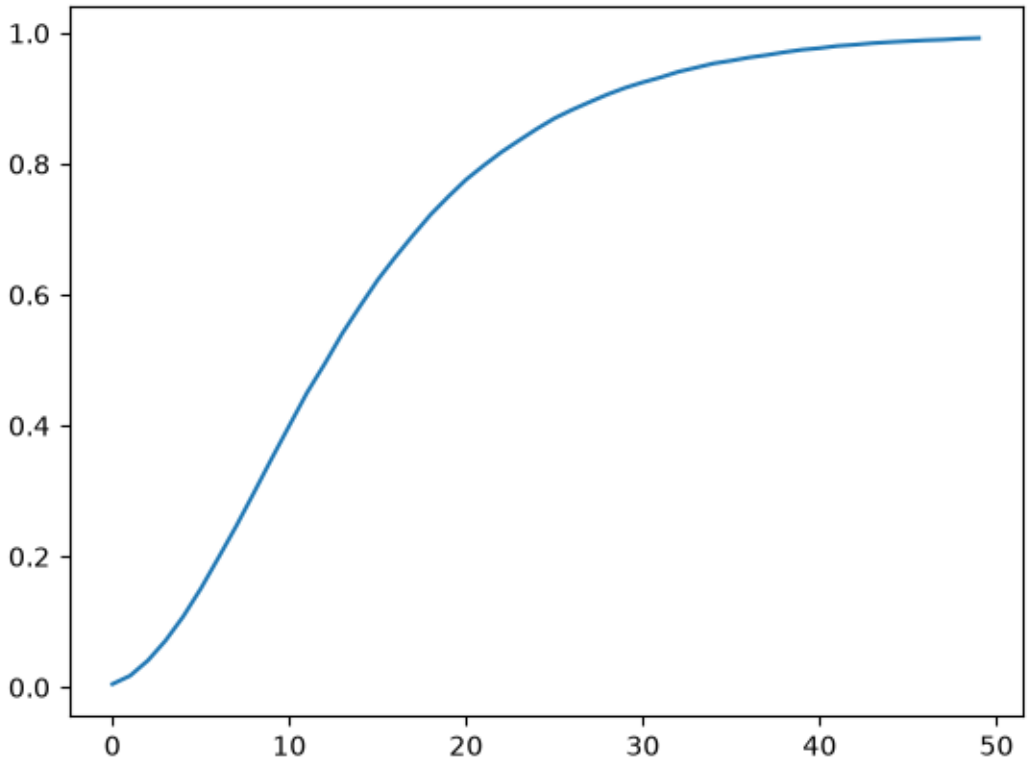
```
# Simulate when nature draws from f
l_arr_f, l_seq_f = simulate_sequences('f', f, g,
                        (F_a, F_b), (G_a, G_b), N=50000)
```

```
N, T = l_arr_f.shape
plt.plot(range(T), np.mean(l_seq_f, axis=0))
plt.show()
```



We also plot the probability that $L\left(w^t\right)$ falls into the interval $[10000, \infty)$ as a function of time and watch how fast probability mass diverges to $+\infty$.

```
plt.plot(range(T), np.sum(l_seq_f > 10000, axis=0) / N)
plt.show()
```



22.6 Likelihood ratio test

We now describe how to employ the machinery of Neyman and Pearson [Neyman and Pearson, 1933] to test the hypothesis that history w^t is generated by repeated IID draws from density f.

Denote q as the data generating process, so that $q = f$ or g.

Upon observing a sample $\{W_i\}_{i=1}^t$, we want to decide whether nature is drawing from g or from f by performing a (frequentist) hypothesis test.

We specify

- Null hypothesis H_0: $q = f$,
- Alternative hypothesis H_1: $q = g$.

Neyman and Pearson proved that the best way to test this hypothesis is to use a **likelihood ratio test** that takes the form:

- accept H_0 if $L(W^t) > c$,
- reject H_0 if $L(W^t) < c$,

where c is a given discrimination threshold.

Setting $c = 1$ is a common choice.

We'll discuss consequences of other choices of c below.

This test is *best* in the sense that it is **uniformly most powerful**.

To understand what this means, we have to define probabilities of two important events that allow us to characterize a test associated with a given threshold c.

The two probabilities are:

- Probability of a Type I error in which we reject H_0 when it is true:

$$\alpha \equiv \Pr\{L(w^t) < c \mid q = f\}$$

- Probability of a Type II error in which we accept H_0 when it is false:

$$\beta \equiv \Pr\{L(w^t) > c \mid q = g\}$$

These two probabilities underlie the following two concepts:

- Probability of false alarm (= significance level = probability of Type I error):

$$\alpha \equiv \Pr\{L(w^t) < c \mid q = f\}$$

- Probability of detection (= power = 1 minus probability of Type II error):

$$1 - \beta \equiv \Pr\{L(w^t) < c \mid q = g\}$$

The Neyman-Pearson Lemma states that among all possible tests, a likelihood ratio test maximizes the probability of detection for a given probability of false alarm.

Another way to say the same thing is that among all possible tests, a likelihood ratio test maximizes **power** for a given **significance level**.

We want a small probability of false alarm and a large probability of detection.

With sample size t fixed, we can change our two probabilities by adjusting c.

A troublesome "that's life" fact is that these two probabilities move in the same direction as we vary the critical value c.

Without specifying quantitative losses from making Type I and Type II errors, there is little that we can say about how we *should* trade off probabilities of the two types of mistakes.

We do know that increasing sample size t improves statistical inference.

Below we plot some informative figures that illustrate this.

We also present a classical frequentist method for choosing a sample size t.

Let's start with a case in which we fix the threshold c at 1.

```
c = 1
```

Below we plot empirical distributions of logarithms of the cumulative likelihood ratios simulated above, which are generated by either f or g.

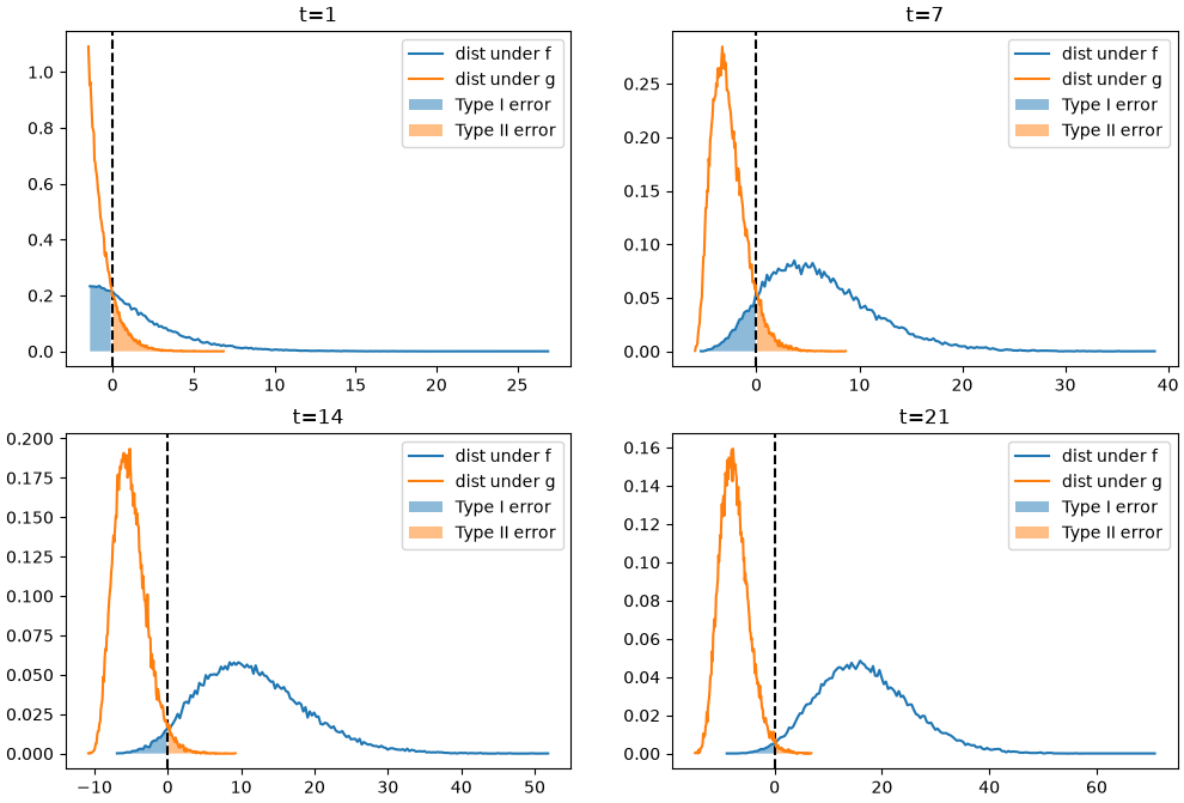
Taking logarithms has no effect on calculating the probabilities because the log is a monotonic transformation.

As t increases, the probabilities of making Type I and Type II errors both decrease, which is good.

This is because most of the probability mass of $\log(L(w^t))$ moves toward $-\infty$ when g is the data generating process, while $\log(L(w^t))$ goes to ∞ when data are generated by f.

That disparate behavior of $\log(L(w^t))$ under f and g is what makes it possible eventually to distinguish $q = f$ from $q = g$.

```
def plot_log_histograms(l_seq_f, l_seq_g, c=1, time_points=[1, 7, 14, 21]):
    """Plot log likelihood ratio histograms."""
    fig, axs = plt.subplots(2, 2, figsize=(12, 8))

    for i, t in enumerate(time_points):
        nr, nc = i // 2, i % 2

        axs[nr, nc].axvline(np.log(c), color="k", ls="--")

        hist_f, x_f = np.histogram(np.log(l_seq_f[:, t]), 200, density=True)
        hist_g, x_g = np.histogram(np.log(l_seq_g[:, t]), 200, density=True)

        axs[nr, nc].plot(x_f[1:], hist_f, label="dist under f")
        axs[nr, nc].plot(x_g[1:], hist_g, label="dist under g")

        # Fill error regions
        for j, (x, hist, label) in enumerate(
            zip([x_f, x_g], [hist_f, hist_g],
            ["Type I error", "Type II error"])):
            ind = x[1:] <= np.log(c) if j == 0 else x[1:] > np.log(c)
            axs[nr, nc].fill_between(x[1:][ind], hist[ind],
                                     alpha=0.5, label=label)

        axs[nr, nc].legend()
        axs[nr, nc].set_title(f"t={t}")

    plt.show()

plot_log_histograms(l_seq_f, l_seq_g, c=c)
```



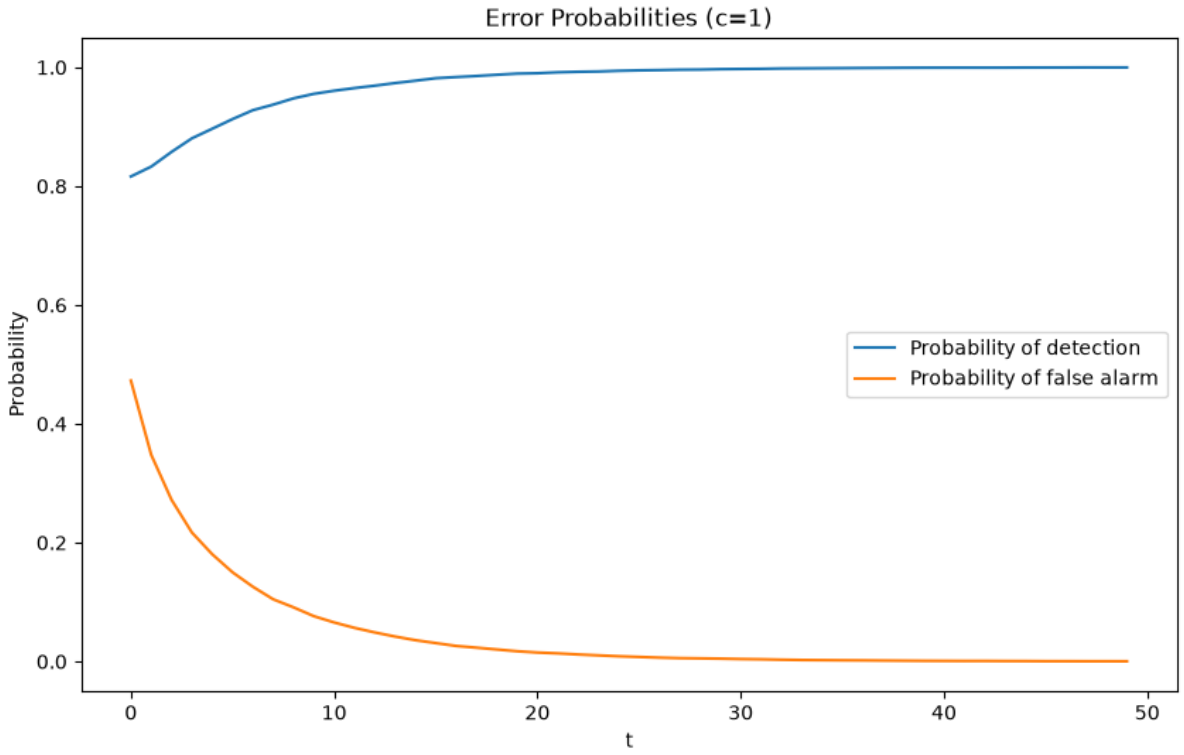
In the above graphs,

- the blue areas are related to but not equal to probabilities α of a type I error because they are integrals of $\log L_t$, not integrals of L_t, over rejection region $L_t < 1$
- the orange areas are related to but not equal to probabilities β of a type II error because they are integrals of $\log L_t$, not integrals of L_t, over acceptance region $L_t > 1$

When we hold c fixed at $c = 1$, the following graph shows that

- the probability of detection monotonically increases with increases in t
- the probability of a false alarm monotonically decreases with increases in t.

```
def compute_error_probabilities(l_seq_f, l_seq_g, c=1):
    """
    Compute Type I and Type II error probabilities.
    """
    N, T = l_seq_f.shape

    # Type I error (false alarm) - reject H0 when true
    PFA = np.array([np.sum(l_seq_f[:, t] < c) / N for t in range(T)])

    # Type II error - accept H0 when false
    beta = np.array([np.sum(l_seq_g[:, t] >= c) / N for t in range(T)])

    # Probability of detection (power)
    PD = np.array([np.sum(l_seq_g[:, t] < c) / N for t in range(T)])

    return {
        'alpha': PFA,
        'beta': beta,
        'PD': PD,
        'PFA': PFA
    }

def plot_error_probabilities(error_dict, T, c=1, title_suffix=""):
    """Plot error probabilities over time."""
    plt.figure(figsize=(10, 6))
    plt.plot(range(T), error_dict['PD'], label="Probability of detection")
    plt.plot(range(T), error_dict['PFA'], label="Probability of false alarm")
    plt.xlabel("t")
    plt.ylabel("Probability")
    plt.title(f"Error Probabilities (c={c}){title_suffix}")
    plt.legend()
    plt.show()

error_probs = compute_error_probabilities(l_seq_f, l_seq_g, c=c)
N, T = l_seq_f.shape
plot_error_probabilities(error_probs, T, c)
```



For a given sample size t, the threshold c uniquely pins down probabilities of both types of error.

If for a fixed t we now free up and move c, we will sweep out the probability of detection as a function of the probability of false alarm.

This produces a receiver operating characteristic curve (ROC curve).

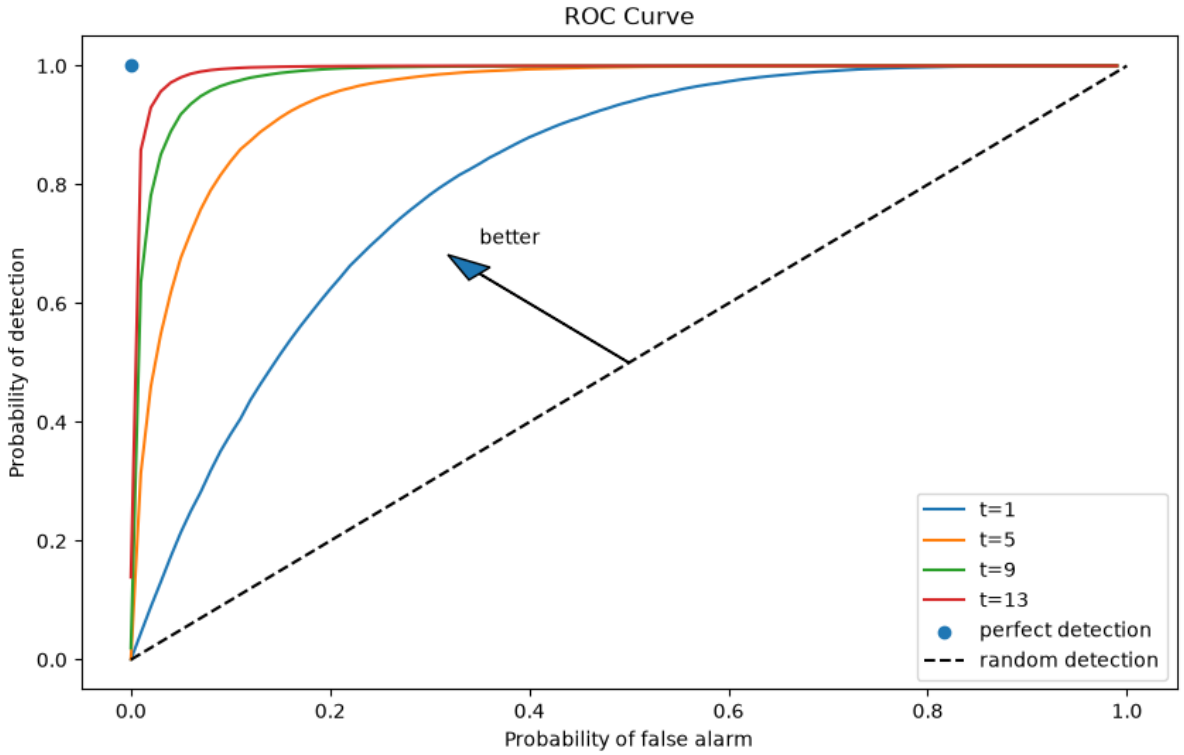
Below, we plot receiver operating characteristic curves for different sample sizes t.

```
def plot_roc_curves(l_seq_f, l_seq_g, t_values=[1, 5, 9, 13], N=None):
    """Plot ROC curves for different sample sizes."""
    if N is None:
        N = l_seq_f.shape[0]

    PFA = np.arange(0, 100, 1)

    plt.figure(figsize=(10, 6))
    for t in t_values:
        percentile = np.percentile(l_seq_f[:, t], PFA)
        PD = [np.sum(l_seq_g[:, t] < p) / N for p in percentile]
        plt.plot(PFA / 100, PD, label=f"t={t}")

    plt.scatter(0, 1, label="perfect detection")
    plt.plot([0, 1], [0, 1], color='k', ls='--', label="random detection")

    plt.arrow(0.5, 0.5, -0.15, 0.15, head_width=0.03)
    plt.text(0.35, 0.7, "better")
    plt.xlabel("Probability of false alarm")
    plt.ylabel("Probability of detection")
    plt.legend()
    plt.title("ROC Curve")
    plt.show()
```

(continues on next page)

(continued from previous page)

```
plot_roc_curves(l_seq_f, l_seq_g, t_values=range(1, 15, 4), N=N)
```



Notice that as t increases, we are assured a larger probability of detection and a smaller probability of false alarm associated with a given discrimination threshold c.

For a given sample size t, both α and β change as we vary c.

As we increase c

- $\alpha \equiv \Pr\{L(w^t) < c \mid q = f\}$ increases
- $\beta \equiv \Pr\{L(w^t) > c \mid q = g\}$ decreases

As $t \to +\infty$, we approach the perfect detection curve that is indicated by a right angle hinging on the blue dot.

For a given sample size t, the discrimination threshold c determines a point on the receiver operating characteristic curve.

It is up to the test designer to trade off probabilities of making the two types of errors.

But we know how to choose the smallest sample size to achieve given targets for the probabilities.

Typically, frequentists aim for a high probability of detection that respects an upper bound on the probability of false alarm.

Below we show an example in which we fix the probability of false alarm at 0.05.

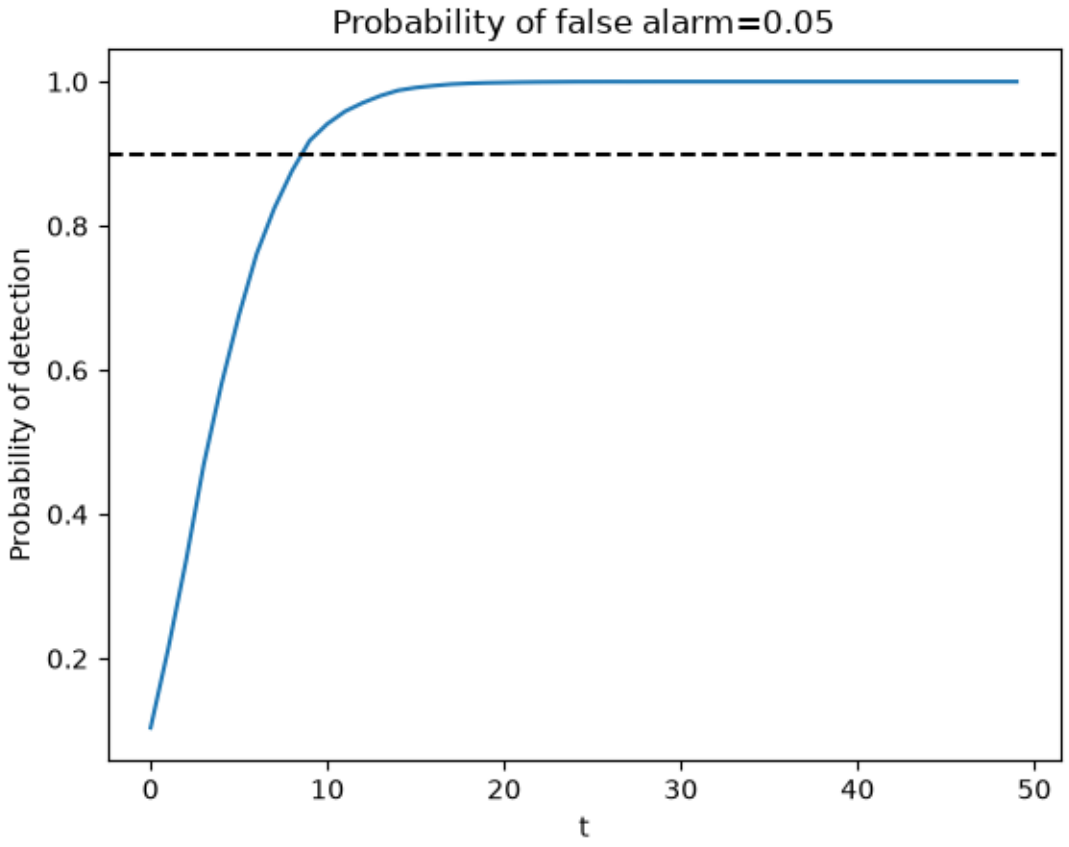
The required sample size for making a decision is then determined by a target probability of detection, for example, 0.9, as depicted in the following graph.

```
PFA = 0.05
PD = np.empty(T)
```

(continues on next page)

(continued from previous page)

```
for t in range(T):

    c = np.percentile(l_seq_f[:, t], PFA * 100)
    PD[t] = np.sum(l_seq_g[:, t] < c) / N

plt.plot(range(T), PD)
plt.axhline(0.9, color="k", ls="--")

plt.xlabel("t")
plt.ylabel("Probability of detection")
plt.title(f"Probability of false alarm={PFA}")
plt.show()
```



The United States Navy evidently used a procedure like this to select a sample size t for doing quality control tests during World War II.

A Navy Captain who had been ordered to perform tests of this kind had doubts about it that he presented to Milton Friedman, as we describe in *this lecture*.

22.6.1 A third distribution h

Now let's consider a case in which neither g nor f generates the data.

Instead, a third distribution h does.

Let's study how accumulated likelihood ratios L behave when h governs the data.

A key tool here is called **Kullback–Leibler divergence** we studied in *Statistical Divergence Measures*.

In our application, we want to measure how much f or g diverges from h

Two Kullback–Leibler divergences pertinent for us are K_f and K_g defined as

$$\begin{aligned} K_f = D_{KL}(h\|f) = KL(h,f) = E_h\left[\log\frac{h(w)}{f(w)}\right] \\ = \int \log\left(\frac{h(w)}{f(w)}\right) h(w)dw. \end{aligned}$$

$$\begin{aligned} K_g = D_{KL}(h\|g) = KL(h,g) = E_h\left[\log\frac{h(w)}{g(w)}\right] \\ = \int \log\left(\frac{h(w)}{g(w)}\right) h(w)dw. \end{aligned}$$

Let's compute the Kullback–Leibler discrepancies using the same code in *Statistical Divergence Measures*.

```
def compute_KL(f, g):
    """
    Compute KL divergence KL(f, g)
    """
    integrand = lambda w: f(w) * np.log(f(w) / g(w))
    val, _ = quad(integrand, 1e-5, 1-1e-5)
    return val

def compute_KL_h(h, f, g):
    """
    Compute KL divergences with respect to reference distribution h
    """
    Kf = compute_KL(h, f)
    Kg = compute_KL(h, g)
    return Kf, Kg
```

22.6.2 A helpful formula

There is a mathematical relationship between likelihood ratios and KL divergence.

When data is generated by distribution h, the expected log likelihood ratio is:

$$\frac{1}{t}E_h[\log L_t] = K_g - K_f \tag{22.1}$$

where $L_t = \prod_{j=1}^{t} \frac{f(w_j)}{g(w_j)}$ is the likelihood ratio process.

Equation (22.1) tells us that:

- When $K_g < K_f$ (i.e., g is closer to h than f is), the expected log likelihood ratio is negative, so $L(w^t) \to 0$.
- When $K_g > K_f$ (i.e., f is closer to h than g is), the expected log likelihood ratio is positive, so $L(w^t) \to +\infty$.

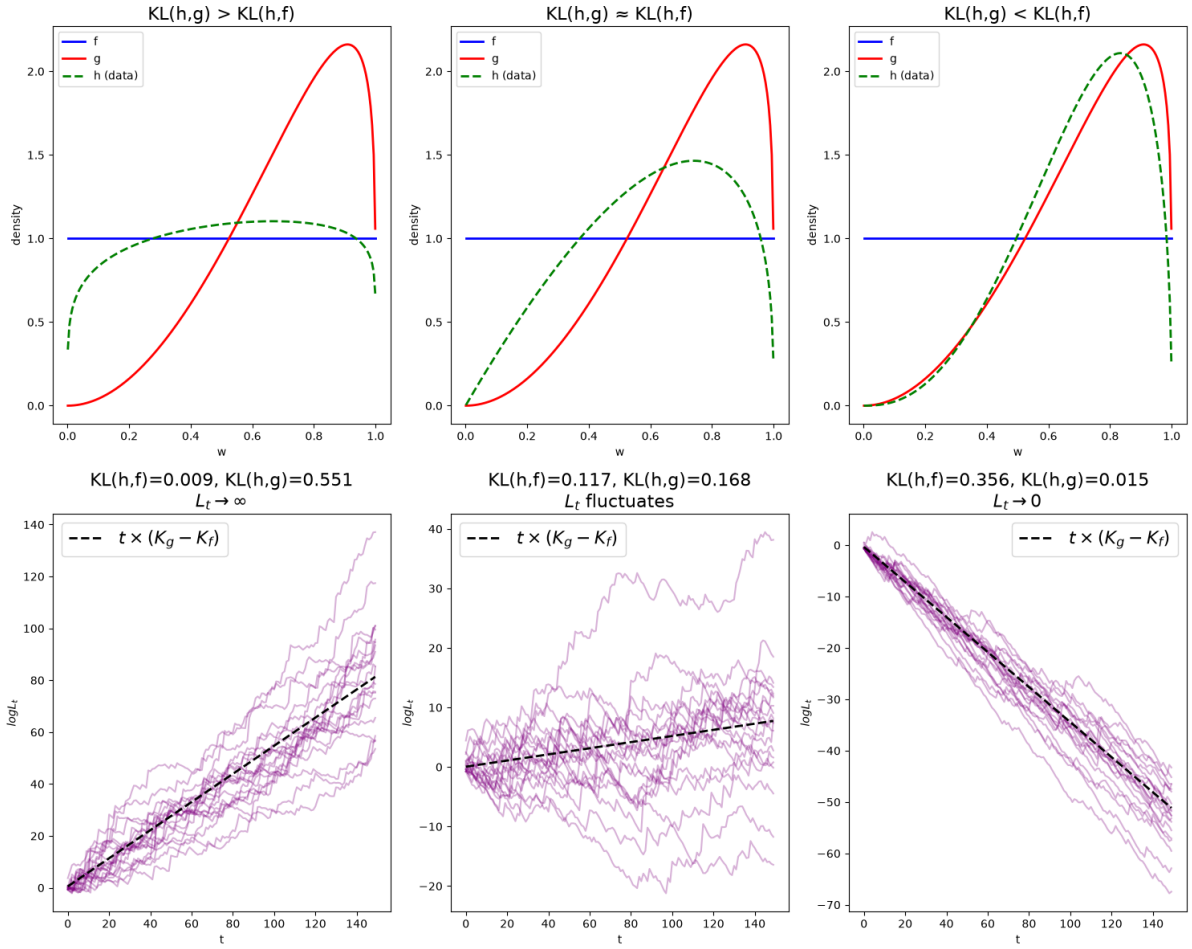
Let's verify this using simulation.

In the simulation, we generate multiple paths using Beta distributions f, g, and h, and compute the paths of $\log(L(w^t))$.

First, we write a function to compute the likelihood ratio process

```
def compute_likelihood_ratios(sequences, f, g):
    """Compute likelihood ratios and cumulative products."""
    l_ratios = f(sequences) / g(sequences)
    L_cumulative = np.cumprod(l_ratios, axis=1)
    return l_ratios, L_cumulative
```

We consider three cases: (1) h is closer to f, (2) f and g are approximately equidistant from h, and (3) h is closer to g.



Note that

- In the first figure, $\log L(w^t)$ diverges to ∞ because $K_g > K_f$.
- In the second figure, we still have $K_g > K_f$, but the difference is smaller, so $L(w^t)$ diverges to infinity at a slower pace.
- In the last figure, $\log L(w^t)$ diverges to $-\infty$ because $K_g < K_f$.
- The black dotted line, $t\left(D_{KL}(h\|g) - D_{KL}(h\|f)\right)$, closely fits the paths verifying (22.1).

These observations align with the theory.

In *Heterogeneous Beliefs and Financial Markets*, we will see an application of these ideas.

22.7 Hypothesis testing and classification

This section discusses another application of likelihood ratio processes.

We describe how a statistician can combine frequentist probabilities of type I and type II errors in order to

- compute an anticipated frequency of selecting a wrong model based on a sample length T
- compute an anticipated error rate in a classification problem

We consider a situation in which nature generates data by mixing known densities f and g with known mixing parameter $\pi_{-1} \in (0, 1)$ so that the random variable w is drawn from the density

$$h(w) = \pi_{-1} f(w) + (1 - \pi_{-1}) g(w)$$

We assume that the statistician knows the densities f and g and also the mixing parameter π_{-1}.

Below, we'll set $\pi_{-1} = .5$, although much of the analysis would follow through with other settings of $\pi_{-1} \in (0, 1)$.

We assume that f and g both put positive probabilities on the same intervals of possible realizations of the random variable W.

In the simulations below, we specify that f is a Beta$(1, 1)$ distribution and that g is Beta$(3, 1.2)$ distribution.

We consider two alternative timing protocols.

- Timing protocol 1 is for the model selection problem
- Timing protocol 2 is for the individual classification problem

Timing Protocol 1: Nature flips a coin only **once** at time $t = -1$ and with probability π_{-1} generates a sequence $\{w_t\}_{t=1}^T$ of IID draws from f and with probability $1 - \pi_{-1}$ generates a sequence $\{w_t\}_{t=1}^T$ of IID draws from g.

Timing Protocol 2. Nature flips a coin **often**. At each time $t \geq 0$, nature flips a coin and with probability π_{-1} draws w_t from f and with probability $1 - \pi_{-1}$ draws w_t from g.

Here is Python code that we'll use to implement timing protocol 1 and 2

```
def protocol_1(π_minus_1, T, N=1000, F_params=(1, 1), G_params=(3, 1.2)):
    """
    Simulate Protocol 1: Nature decides once at t=-1 which model to use.
    """
    F_a, F_b = F_params
    G_a, G_b = G_params

    # Single coin flip for the true model
    true_models_F = np.random.rand(N) < π_minus_1
    sequences = np.empty((N, T))

    n_f = np.sum(true_models_F)
    n_g = N - n_f

    if n_f > 0:
        sequences[true_models_F, :] = np.random.beta(F_a, F_b, (n_f, T))
    if n_g > 0:
        sequences[~true_models_F, :] = np.random.beta(G_a, G_b, (n_g, T))

    return sequences, true_models_F

def protocol_2(π_minus_1, T, N=1000, F_params=(1, 1), G_params=(3, 1.2)):
    """
```

(continues on next page)

(continued from previous page)

```
    Simulate Protocol 2: Nature decides at each time step which model to use.
    """
    F_a, F_b = F_params
    G_a, G_b = G_params

    # Coin flips for each time step
    true_models_F = np.random.rand(N, T) < π_minus_1
    sequences = np.empty((N, T))

    n_f = np.sum(true_models_F)
    n_g = N * T - n_f

    if n_f > 0:
        sequences[true_models_F] = np.random.beta(F_a, F_b, n_f)
    if n_g > 0:
        sequences[~true_models_F] = np.random.beta(G_a, G_b, n_g)

    return sequences, true_models_F
```

Remark: Under timing protocol 2, the $\{w_t\}_{t=1}^T$ is a sequence of IID draws from $h(w)$. Under timing protocol 1, the $\{w_t\}_{t=1}^T$ is not IID. It is **conditionally IID** – meaning that with probability π_{-1} it is a sequence of IID draws from $f(w)$ and with probability $1-\pi_{-1}$ it is a sequence of IID draws from $g(w)$. For more about this, see *this lecture about exchangeability*.

We again deploy a **likelihood ratio process** with time t component being the likelihood ratio

$$\ell(w_t) = \frac{f(w_t)}{g(w_t)}, \quad t \geq 1.$$

The **likelihood ratio process** for sequence $\{w_t\}_{t=1}^{\infty}$ is

$$L(w^t) = \prod_{i=1}^{t} \ell(w_i),$$

For shorthand we'll write $L_t = L(w^t)$.

22.7.1 Model selection mistake probability

We first study a problem that assumes timing protocol 1.

Consider a decision maker who wants to know whether model f or model g governs a data set of length T observations.

The decision makers has observed a sequence $\{w_t\}_{t=1}^T$.

On the basis of that observed sequence, a likelihood ratio test selects model f when $L_T \geq 1$ and model g when $L_T < 1$.

When model f generates the data, the probability that the likelihood ratio test selects the wrong model is

$$p_f = \text{Prob}\left(L_T < 1 \middle| f\right) = \alpha_T.$$

When model g generates the data, the probability that the likelihood ratio test selects the wrong model is

$$p_g = \text{Prob}\left(L_T \geq 1 \middle| g\right) = \beta_T.$$

We can construct a probability that the likelihood ratio selects the wrong model by assigning a Bayesian prior probability of $\pi_{-1} = .5$ that nature selects model f then averaging p_f and p_g to form the Bayesian posterior probability of a detection

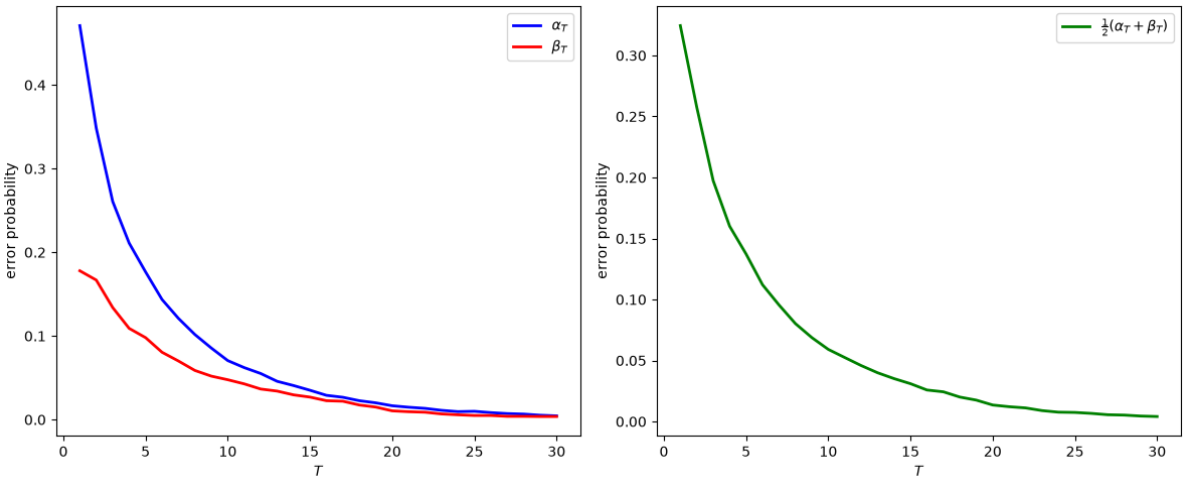
error equal to

$$p(\text{wrong decision}) = \frac{1}{2}(\alpha_T + \beta_T). \tag{22.2}$$

Now let's simulate timing protocol 1 and compute the error probabilities

```
def compute_protocol_1_errors(π_minus_1, T_max, N_simulations, f_func, g_func,
                              F_params=(1, 1), G_params=(3, 1.2)):
    """
    Compute error probabilities for Protocol 1.
    """
    sequences, true_models = protocol_1(
        π_minus_1, T_max, N_simulations, F_params, G_params)
    l_ratios, L_cumulative = compute_likelihood_ratios(sequences,
                                    f_func, g_func)

    T_range = np.arange(1, T_max + 1)

    mask_f = true_models
    mask_g = ~true_models

    L_f = L_cumulative[mask_f, :]
    L_g = L_cumulative[mask_g, :]

    α_T = np.mean(L_f < 1, axis=0)
    β_T = np.mean(L_g >= 1, axis=0)
    error_prob = 0.5 * (α_T + β_T)

    return {
        'T_range': T_range,
        'alpha': α_T,
        'beta': β_T,
        'error_prob': error_prob,
        'L_cumulative': L_cumulative,
        'true_models': true_models
    }
```

The following code visualizes the error probabilities for timing protocol 1



```
At T=30:
α_30 = 0.0045
β_30 = 0.0037
Model selection error probability = 0.0041
```

Notice how the model selection error probability approaches zero as T grows.

22.7.2 Classification

We now consider a problem that assumes timing protocol 2.

A decision maker wants to classify components of an observed sequence $\{w_t\}_{t=1}^T$ as having been drawn from either f or g.

The decision maker uses the following classification rule:

$$w_t \text{ is from } f \text{ if } l_t > 1$$
$$w_t \text{ is from } g \text{ if } l_t \leq 1.$$

Under this rule, the expected misclassification rate is

$$p(\text{misclassification}) = \frac{1}{2}(\tilde{\alpha}_t + \tilde{\beta}_t) \tag{22.3}$$

where $\tilde{\alpha}_t = \text{Prob}(l_t < 1 \mid f)$ and $\tilde{\beta}_t = \text{Prob}(l_t \geq 1 \mid g)$.

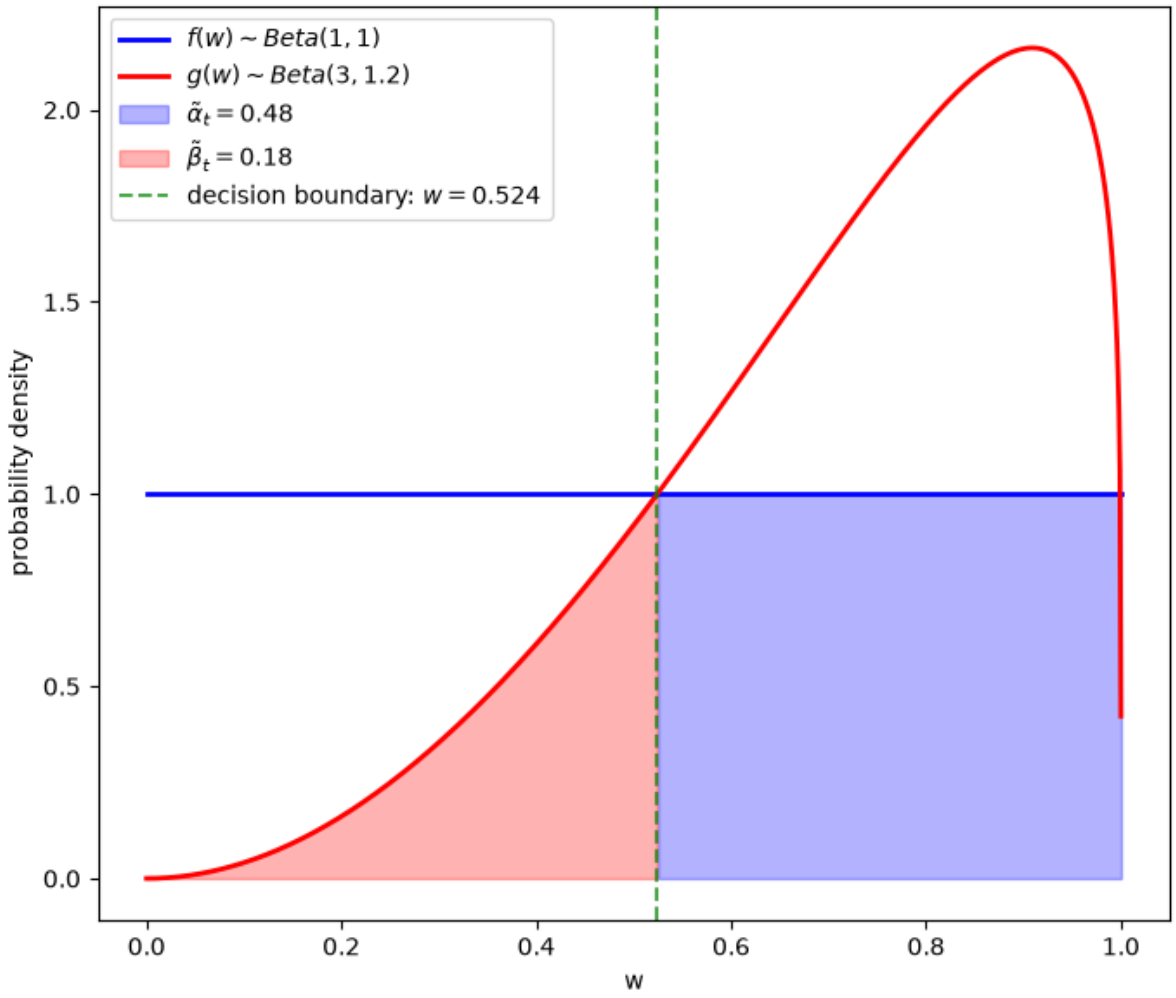
Now let's write some code to simulate it

```
def compute_protocol_2_errors(π_minus_1, T_max, N_simulations, f_func, g_func,
                              F_params=(1, 1), G_params=(3, 1.2)):
    """
    Compute error probabilities for Protocol 2.
    """
    sequences, true_models = protocol_2(π_minus_1,
                        T_max, N_simulations, F_params, G_params)
    l_ratios, _ = compute_likelihood_ratios(sequences, f_func, g_func)

    T_range = np.arange(1, T_max + 1)

    accuracy = np.empty(T_max)
    for t in range(T_max):
        predictions = (l_ratios[:, t] >= 1)
        actual = true_models[:, t]
        accuracy[t] = np.mean(predictions == actual)

    return {
        'T_range': T_range,
        'accuracy': accuracy,
        'l_ratios': l_ratios,
        'true_models': true_models
    }
```

Since for each t, the decision boundary is the same, the decision boundary can be computed as

```
root = brentq(lambda w: f(w) / g(w) - 1, 0.001, 0.999)
```

we can plot the distributions of f and g and the decision boundary



To the left of the green vertical line $g < f$, so $l_t > 1$; therefore a w_t that falls to the left of the green line is classified as a type f individual.

- The shaded red area equals β – the probability of classifying someone as a type g individual when it is really a type f individual.

To the right of the green vertical line $g > f$, so $l_t < 1$; therefore a w_t that falls to the right of the green line is classified as a type g individual.

- The shaded blue area equals α – the probability of classifying someone as a type f when it is really a type g individual.

This gives us clues about how to compute the theoretical classification error probability

```
# Compute theoretical tilde α_t and tilde β_t
def α_integrand(w):
    """Integrand for tilde α_t = P(l_t < 1 | f)"""
    return f(w) if f(w) / g(w) < 1 else 0

def β_integrand(w):
    """Integrand for tilde β_t = P(l_t >= 1 | g)"""
    return g(w) if f(w) / g(w) >= 1 else 0
```

(continues on next page)

(continued from previous page)

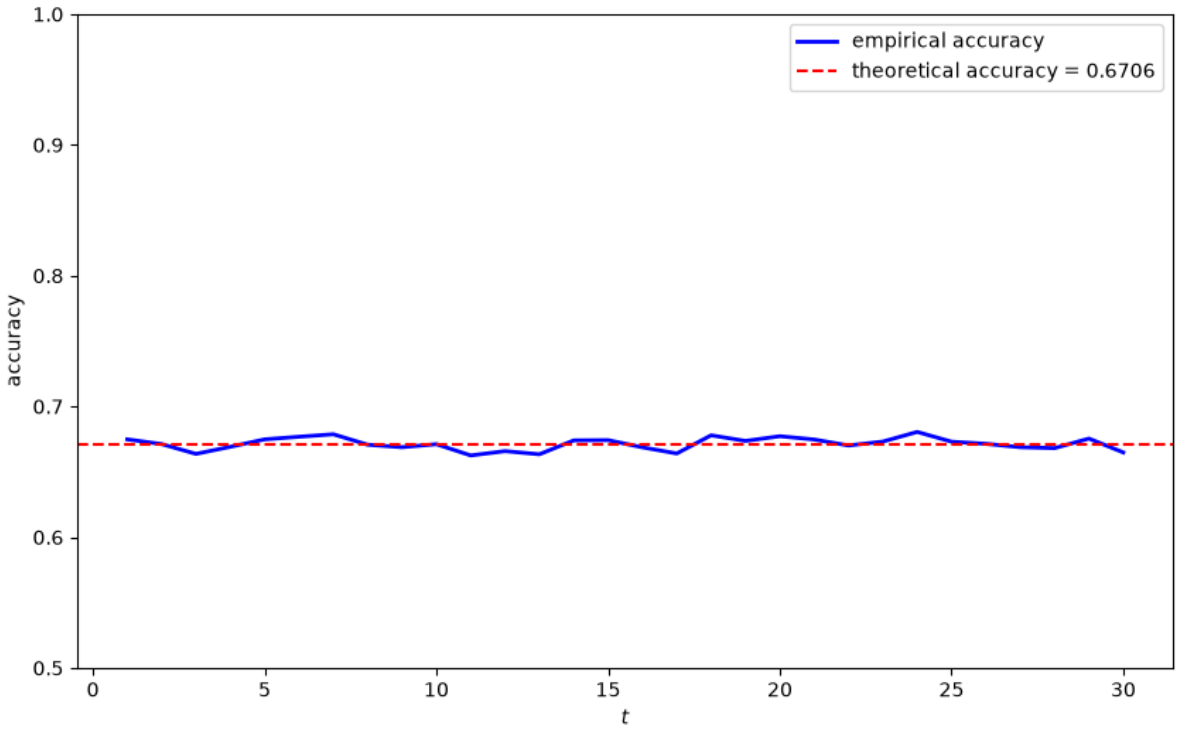
```
# Compute the integrals
α_theory, _ = quad(α_integrand, 0, 1, limit=100)
β_theory, _ = quad(β_integrand, 0, 1, limit=100)

theory_error = 0.5 * (α_theory + β_theory)

print(f"theoretical tilde α_t = {α_theory:.4f}")
print(f"theoretical tilde β_t = {β_theory:.4f}")
print(f"theoretical classification error probability = {theory_error:.4f}")
```

```
theoretical tilde α_t = 0.4752
theoretical tilde β_t = 0.1836
theoretical classification error probability = 0.3294
```

Now we simulate timing protocol 2 and compute the classification error probability.

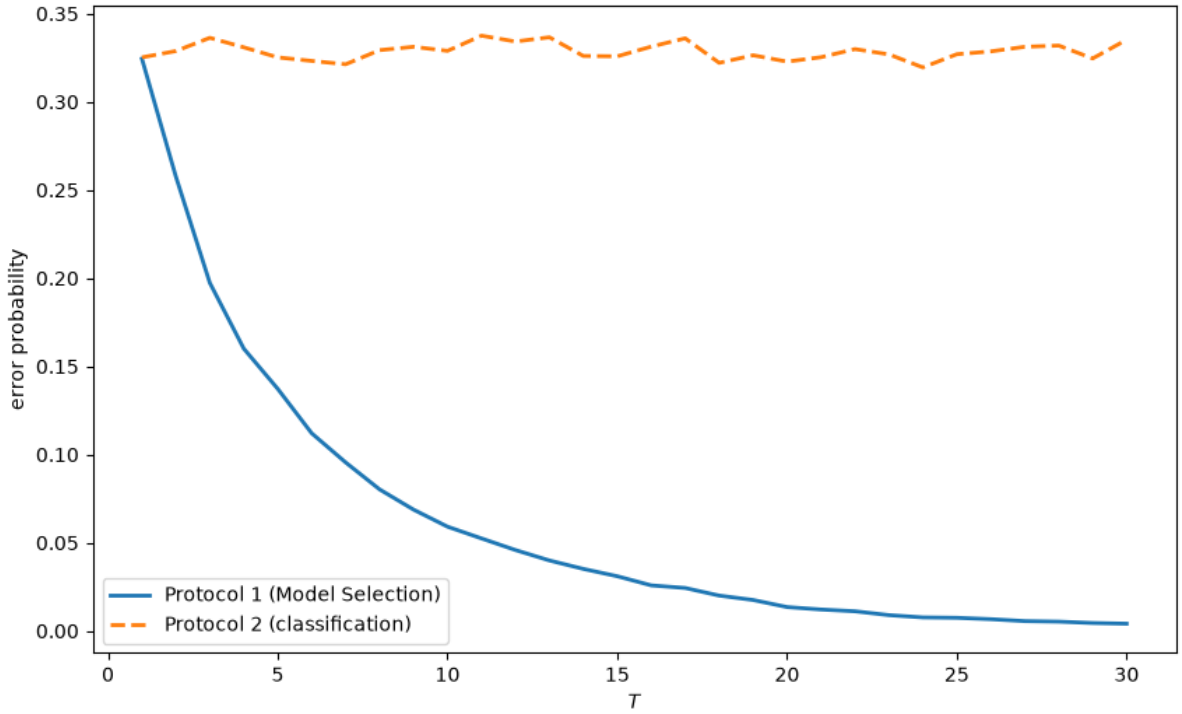
In the next cell, we also compare the theoretical classification accuracy to the empirical classification accuracy

```
def analyze_protocol_2(π_minus_1, T_max, N_simulations, f_func, g_func,
                       theory_error=None, F_params=(1, 1), G_params=(3, 1.2)):
    """Analyze Protocol 2."""
    result = compute_protocol_2_errors(π_minus_1, T_max, N_simulations,
                                       f_func, g_func, F_params, G_params)

    # Plot results
    plt.figure(figsize=(10, 6))
    plt.plot(result['T_range'], result['accuracy'],
             'b-', linewidth=2, label='empirical accuracy')

    if theory_error is not None:
        plt.axhline(1 - theory_error, color='r', linestyle='--',
                    label=f'theoretical accuracy = {1 - theory_error:.4f}')

    plt.xlabel('$t$')
    plt.ylabel('accuracy')
    plt.legend()
    plt.ylim(0.5, 1.0)
    plt.show()

    return result

# Analyze Protocol 2
result_p2 = analyze_protocol_2(π_minus_1, T_max, N_simulations, f, g,
                               theory_error, (F_a, F_b), (G_a, G_b))
```



Let's watch decisions made by the two timing protocols as more and more observations accrue.

```
def compare_protocols(result1, result2):
    """Compare results from both protocols."""
    plt.figure(figsize=(10, 6))

    plt.plot(result1['T_range'], result1['error_prob'], linewidth=2,
             label='Protocol 1 (Model Selection)')
    plt.plot(result2['T_range'], 1 - result2['accuracy'],
             linestyle='--', linewidth=2,
             label='Protocol 2 (classification)')

    plt.xlabel('$T$')
    plt.ylabel('error probability')
    plt.legend()
    plt.show()

compare_protocols(result_p1, result_p2)
```



From the figure above, we can see:

- For both timing protocols, the error probability starts at the same level, subject to a little randomness.
- For timing protocol 1, the error probability decreases as the sample size increases because we are making just **one** decision – i.e., selecting whether f or g governs **all** individuals. More data provides better evidence.
- For timing protocol 2, the error probability remains constant because we are making **many** decisions – one classification decision for each observation.

Remark: Think about how laws of large numbers are applied to compute error probabilities for the model selection problem and the classification problem.

22.7.3 Error probability and divergence measures

A plausible guess is that the ability of a likelihood ratio to distinguish distributions f and g depends on how "different" they are.

We have learnt some measures of "difference" between distributions in *Statistical Divergence Measures*.

Let's now study two more measures of "difference" between distributions that are useful in the context of model selection and classification.

Recall that Chernoff entropy between probability densities f and g is defined as:

$$C(f,g) = -\log \min_{\phi \in (0,1)} \int f^{\phi}(x) g^{1-\phi}(x) dx$$

An upper bound on model selection error probability is

$$e^{-C(f,g)T}.$$

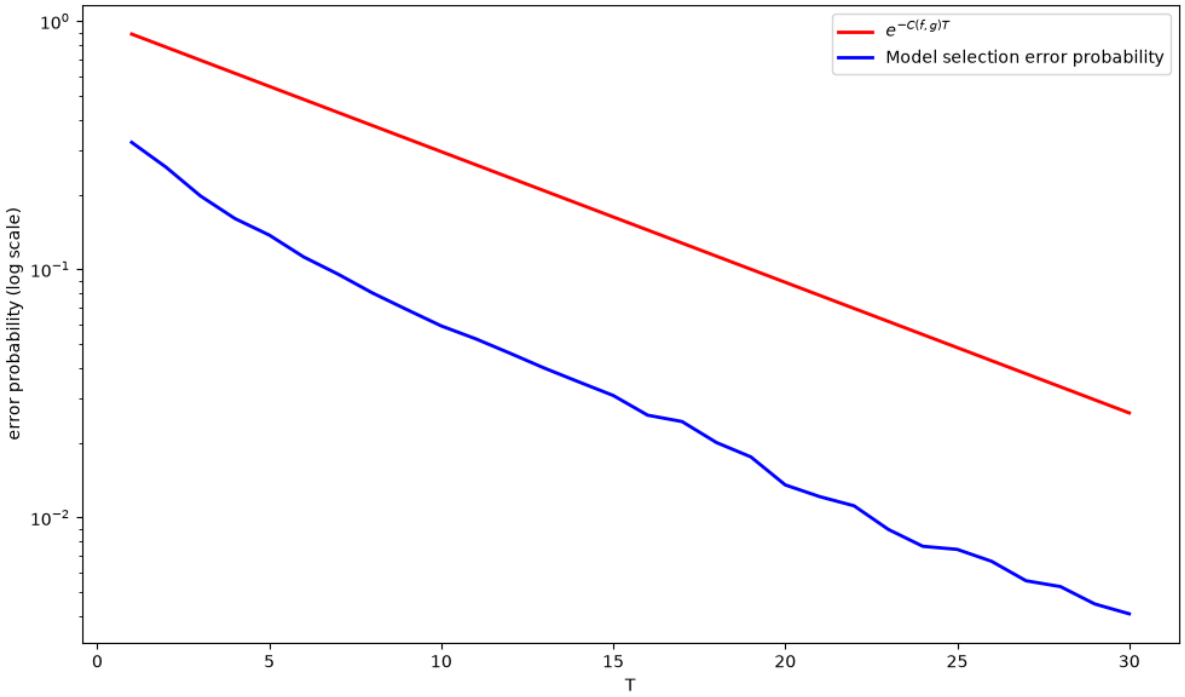
Let's compute Chernoff entropy numerically with some Python code

```
def chernoff_integrand(ϕ, f, g):
    """
    Compute the integrand for Chernoff entropy
    """
    def integrand(w):
        return f(w)**ϕ * g(w)**(1-ϕ)

    result, _ = quad(integrand, 1e-5, 1-1e-5)
    return result

def compute_chernoff_entropy(f, g):
    """
    Compute Chernoff entropy C(f,g)
    """
    def objective(ϕ):
        return chernoff_integrand(ϕ, f, g)

    # Find the minimum over ϕ in (0,1)
    result = minimize_scalar(objective,
                             bounds=(1e-5, 1-1e-5),
                             method='bounded')
    min_value = result.fun
    ϕ_optimal = result.x

    chernoff_entropy = -np.log(min_value)
    return chernoff_entropy, ϕ_optimal
C_fg, ϕ_optimal = compute_chernoff_entropy(f, g)
print(f"Chernoff entropy C(f,g) = {C_fg:.4f}")
print(f"Optimal ϕ = {ϕ_optimal:.4f}")
```

```
Chernoff entropy C(f,g) = 0.1212
Optimal ϕ = 0.5969
```

Now let's examine how $e^{-C(f,g)T}$ behaves as a function of T and compare it to the model selection error probability

```
T_range = np.arange(1, T_max+1)
chernoff_bound = np.exp(-C_fg * T_range)

# Plot comparison
fig, ax = plt.subplots(figsize=(10, 6))

ax.semilogy(T_range, chernoff_bound, 'r-', linewidth=2,
           label=f'$e^{{-C(f,g)T}}$')
ax.semilogy(T_range, result_p1['error_prob'], 'b-', linewidth=2,
           label='Model selection error probability')

ax.set_xlabel('T')
ax.set_ylabel('error probability (log scale)')
ax.legend()
plt.tight_layout()
plt.show()
```



Evidently, $e^{-C(f,g)T}$ is an upper bound on the error rate.

In *Statistical Divergence Measures*, we also studied **Jensen-Shannon divergence** as a symmetric measure of distance between distributions.

We can use Jensen-Shannon divergence to measure the distance between distributions f and g and compute how it covaries with the model selection error probability.

We also compute Jensen-Shannon divergence numerically with some Python code

```
def compute_JS(f, g):
    """
    Compute Jensen-Shannon divergence
    """
    def m(w):
        return 0.5 * (f(w) + g(w))

    js_div = 0.5 * compute_KL(f, m) + 0.5 * compute_KL(g, m)
    return js_div
```

Now let's return to our guess that the error probability at large sample sizes is related to the Chernoff entropy between two distributions.

We verify this by computing the correlation between the log of the error probability at $T = 50$ under Timing Protocol 1 and the divergence measures.

In the simulation below, nature draws $N/2$ sequences from g and $N/2$ sequences from f.

Note

Nature does this rather than flipping a fair coin to decide whether to draw from g or f once and for all before each simulation of length T.

We use the following pairs of Beta distributions for f and g as test cases

```
distribution_pairs = [
    # (f_params, g_params)
    ((1, 1), (0.1, 0.2)),
    ((1, 1), (0.3, 0.3)),
    ((1, 1), (0.3, 0.4)),
    ((1, 1), (0.5, 0.5)),
    ((1, 1), (0.7, 0.6)),
    ((1, 1), (0.9, 0.8)),
    ((1, 1), (1.1, 1.05)),
    ((1, 1), (1.2, 1.1)),
    ((1, 1), (1.5, 1.2)),
    ((1, 1), (2, 1.5)),
    ((1, 1), (2.5, 1.8)),
    ((1, 1), (3, 1.2)),
    ((1, 1), (4, 1)),
    ((1, 1), (5, 1))
]
```

Now let's run the simmulation

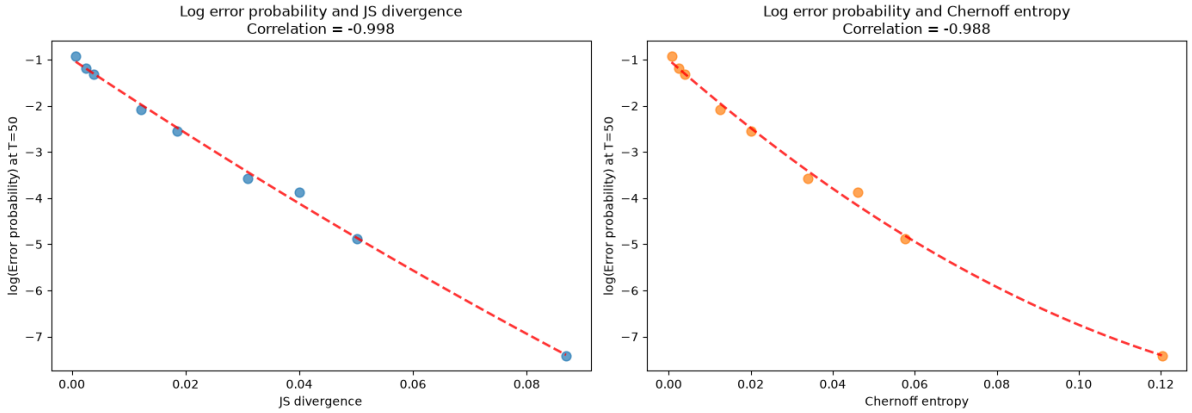
```
# Parameters for simulation
T_large = 50
N_sims = 5000
N_half = N_sims // 2

# Initialize arrays
n_pairs = len(distribution_pairs)
kl_fg_vals = np.zeros(n_pairs)
kl_gf_vals = np.zeros(n_pairs)
js_vals = np.zeros(n_pairs)
chernoff_vals = np.zeros(n_pairs)
error_probs = np.zeros(n_pairs)
pair_names = []

for i, ((f_a, f_b), (g_a, g_b)) in enumerate(distribution_pairs):
    # Create density functions
    f = jit(lambda x, a=f_a, b=f_b: p(x, a, b))
    g = jit(lambda x, a=g_a, b=g_b: p(x, a, b))

    # Compute divergence measures
    kl_fg_vals[i] = compute_KL(f, g)
    kl_gf_vals[i] = compute_KL(g, f)
    js_vals[i] = compute_JS(f, g)
    chernoff_vals[i], _ = compute_chernoff_entropy(f, g)

    # Generate samples
    sequences_f = np.random.beta(f_a, f_b, (N_half, T_large))
    sequences_g = np.random.beta(g_a, g_b, (N_half, T_large))

    # Compute likelihood ratios and cumulative products
    _, L_cumulative_f = compute_likelihood_ratios(sequences_f, f, g)
    _, L_cumulative_g = compute_likelihood_ratios(sequences_g, f, g)

    # Get final values
    L_cumulative_f = L_cumulative_f[:, -1]
    L_cumulative_g = L_cumulative_g[:, -1]
```

(continues on next page)

(continued from previous page)

```
    # Calculate error probabilities
    error_probs[i] = 0.5 * (np.mean(L_cumulative_f < 1) +
                            np.mean(L_cumulative_g >= 1))
    pair_names.append(f"Beta({f_a},{f_b}) and Beta({g_a},{g_b})")

cor_data = {
    'kl_fg': kl_fg_vals,
    'kl_gf': kl_gf_vals,
    'js': js_vals,
    'chernoff': chernoff_vals,
    'error_prob': error_probs,
    'names': pair_names,
    'T': T_large}
```

Now let's visualize the correlations



Evidently, Chernoff entropy and Jensen-Shannon entropy each covary tightly with the model selection error probability.

We'll encounter related ideas in *A Problem that Stumped Milton Friedman* very soon.

22.8 Markov chains

Let's now look at a likelihood ratio process for a sequence of random variables that is not independently and identically distributed.

Here we assume that the sequence is generated by a Markov chain on a finite state space.

We consider two n-state irreducible and aperiodic Markov chain models on the same state space $\{1, 2, \ldots, n\}$ with positive transition matrices $P^{(f)}$, $P^{(g)}$ and initial distributions $\pi_0^{(f)}$, $\pi_0^{(g)}$.

We assume that nature samples from chain f.

For a sample path $(x_0, x_1, \ldots, x_T)$, let N_{ij} count transitions from state i to j.

The likelihood process under model $m \in \{f, g\}$ is

$$L_T^{(m)} = \pi_{0,x_0}^{(m)} \prod_{i=1}^{n} \prod_{j=1}^{n} \left(P_{ij}^{(m)}\right)^{N_{ij}}$$

Hence,

$$\log L_T^{(m)} = \log \pi_{0,x_0}^{(m)} + \sum_{i,j} N_{ij} \log P_{ij}^{(m)}$$

The log-likelihood ratio is

$$\log \frac{L_T^{(f)}}{L_T^{(g)}} = \log \frac{\pi_{0,x_0}^{(f)}}{\pi_{0,x_0}^{(g)}} + \sum_{i,j} N_{ij} \log \frac{P_{ij}^{(f)}}{P_{ij}^{(g)}} \tag{22.4}$$

22.8.1 KL divergence rate

By the ergodic theorem for irreducible, aperiodic Markov chains, we have

$$\frac{N_{ij}}{T} \xrightarrow{a.s.} \pi_i^{(f)} P_{ij}^{(f)} \quad \text{as } T \to \infty$$

where $\pi^{(f)}$ is the stationary distribution satisfying $\pi^{(f)} = \pi^{(f)} P^{(f)}$.

Therefore,

$$\frac{1}{T} \log \frac{L_T^{(f)}}{L_T^{(g)}} = \frac{1}{T} \log \frac{\pi_{0,x_0}^{(f)}}{\pi_{0,x_0}^{(g)}} + \frac{1}{T} \sum_{i,j} N_{ij} \log \frac{P_{ij}^{(f)}}{P_{ij}^{(g)}}$$

Taking the limit as $T \to \infty$, we have:

- The first term: $\frac{1}{T} \log \frac{\pi_{0,x_0}^{(f)}}{\pi_{0,x_0}^{(g)}} \to 0$
- The second term: $\frac{1}{T} \sum_{i,j} N_{ij} \log \frac{P_{ij}^{(f)}}{P_{ij}^{(g)}} \xrightarrow{a.s.} \sum_{i,j} \pi_i^{(f)} P_{ij}^{(f)} \log \frac{P_{ij}^{(f)}}{P_{ij}^{(g)}}$

Define the **KL divergence rate** as

$$h_{KL}(f,g) = \sum_{i=1}^{n} \pi_i^{(f)} \underbrace{\sum_{j=1}^{n} P_{ij}^{(f)} \log \frac{P_{ij}^{(f)}}{P_{ij}^{(g)}}}_{=: KL(P_{i\cdot}^{(f)}, P_{i\cdot}^{(g)})}$$

where $KL(P_{i\cdot}^{(f)}, P_{i\cdot}^{(g)})$ is the row-wise KL divergence.

By the ergodic theorem, we have

$$\frac{1}{T} \log \frac{L_T^{(f)}}{L_T^{(g)}} \xrightarrow{a.s.} h_{KL}(f,g) \quad \text{as } T \to \infty$$

Taking expectations and using the dominated convergence theorem, we obtain

$$\frac{1}{T} E_f \left[\log \frac{L_T^{(f)}}{L_T^{(g)}} \right] \to h_{KL}(f,g) \quad \text{as } T \to \infty$$

Here we invite readers to pause and compare this result with (22.1).

Let's confirm this in the simulation below.

22.8.2 Simulations

Let's implement simulations to illustrate these concepts with a three-state Markov chain.

We start by writing functions to compute the stationary distribution and the KL divergence rate for Markov chain models.

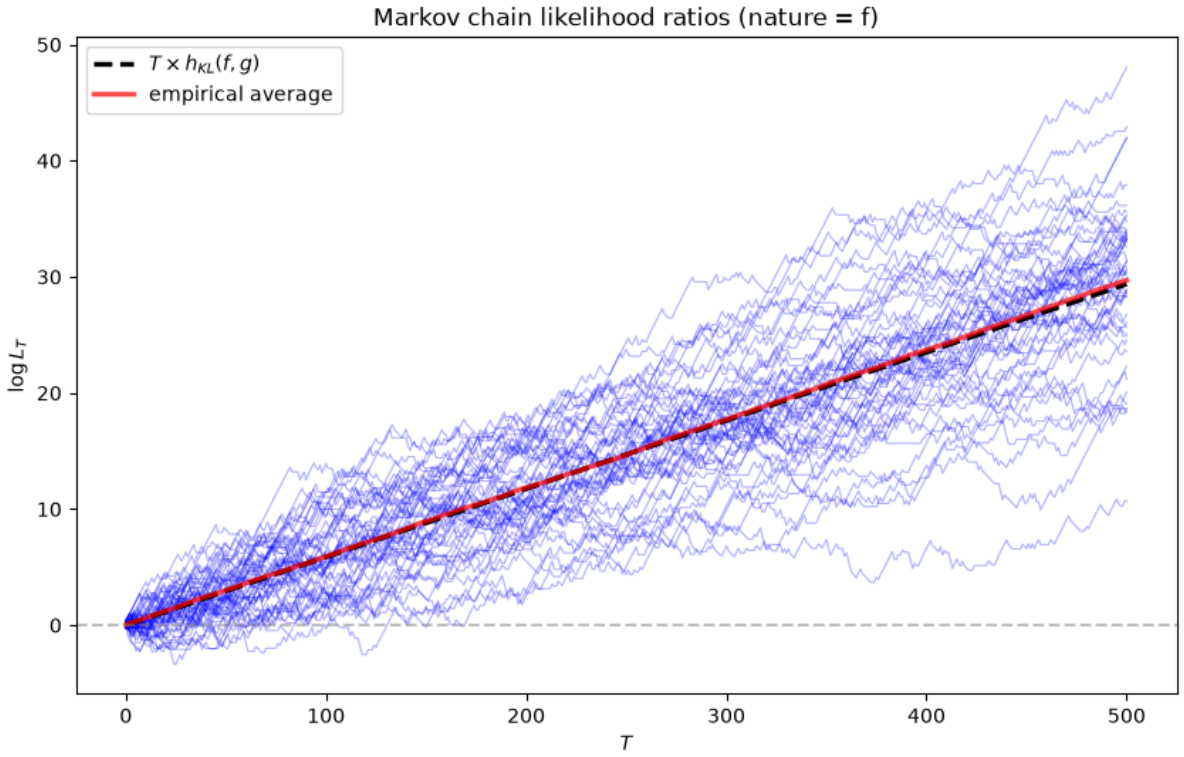
Now let's create an example with two different 3-state Markov chains.

We are now ready to simulate paths and visualize how likelihood ratios evolve.

We verify $\frac{1}{T} E_f \left[\log \frac{L_T^{(f)}}{L_T^{(g)}} \right] = h_{KL}(f, g)$ starting from the stationary distribution by plotting both the empirical average and the line predicted by the theory

```
# Define example Markov chain transition matrices
P_f = np.array([[0.7, 0.2, 0.1],
                [0.3, 0.5, 0.2],
                [0.1, 0.3, 0.6]])

P_g = np.array([[0.5, 0.3, 0.2],
                [0.2, 0.6, 0.2],
                [0.2, 0.2, 0.6]])

markov_results = analyze_markov_chains(P_f, P_g)
```

```
Stationary distribution (f): [0.41176471 0.32352941 0.26470588]
Stationary distribution (g): [0.28571429 0.38095238 0.33333333]

KL divergence rate h(f, g): 0.0588
KL divergence rate h(g, f): 0.0563
```



22.9 Related lectures

Likelihood processes play an important role in Bayesian learning, as described in *Likelihood Ratio Processes and Bayesian Learning* and as applied in *Job Search VIII: Search with Learning*.

Likelihood ratio processes are central to Lawrence Blume and David Easley's answer to their question "If you're so smart, why aren't you rich?" [Blume and Easley, 2006], the subject of the lecture*Heterogeneous Beliefs and Financial Markets*.

Likelihood ratio processes also appear in Additive and Multiplicative Functionals, which contains another illustration of the **peculiar property** of likelihood ratio processes described above.

22.10 Exercises

Exercise 22.10.1

Consider the setting where nature generates data from a third density h.

Let $\{w_t\}_{t=1}^T$ be IID draws from h, and let $L_t = L(w^t)$ be the likelihood ratio process defined as in the lecture.

Show that:

$$\frac{1}{t} E_h[\log L_t] = K_g - K_f$$

with finite K_g, K_f, $E_h|\log f(W)| < \infty$ and $E_h|\log g(W)| < \infty$.

Hint: Start by expressing $\log L_t$ as a sum of $\log \ell(w_i)$ terms and compare with the definition of K_f and K_g.

Solution

Since $w_1, \ldots, w_t$ are IID draws from h, we can write

$$\log L_t = \log \prod_{i=1}^t \ell(w_i) = \sum_{i=1}^t \log \ell(w_i) = \sum_{i=1}^t \log \frac{f(w_i)}{g(w_i)}$$

Taking the expectation under h

$$E_h[\log L_t] = E_h\left[\sum_{i=1}^t \log \frac{f(w_i)}{g(w_i)}\right] = \sum_{i=1}^t E_h\left[\log \frac{f(w_i)}{g(w_i)}\right]$$

Since the w_i are identically distributed

$$E_h[\log L_t] = t \cdot E_h\left[\log \frac{f(w)}{g(w)}\right]$$

where $w \sim h$.

Therefore

$$\frac{1}{t} E_h[\log L_t] = E_h\left[\log \frac{f(w)}{g(w)}\right] = E_h[\log f(w)] - E_h[\log g(w)]$$

Now, from the definition of Kullback-Leibler divergence

$$K_f = \int h(w) \log \frac{h(w)}{f(w)} dw = E_h[\log h(w)] - E_h[\log f(w)]$$

This gives us

$$E_h[\log f(w)] = E_h[\log h(w)] - K_f$$

Similarly

$$E_h[\log g(w)] = E_h[\log h(w)] - K_g$$

Substituting back

$$\begin{aligned}\frac{1}{t}E_h[\log L_t] &= E_h[\log f(w)] - E_h[\log g(w)] \\ &= [E_h[\log h(w)] - K_f] - [E_h[\log h(w)] - K_g] \\ &= K_g - K_f\end{aligned}$$

Exercise 22.10.2

Building on Exercise 22.10.1, use the result to explain what happens to L_t as $t \to \infty$ in the following cases:

1. When $K_g > K_f$ (i.e., f is "closer" to h than g is)
2. When $K_g < K_f$ (i.e., g is "closer" to h than f is)

Relate your answer to the simulation results shown in *this section*.

Solution

From Exercise 22.10.1, we know that:

$$\frac{1}{t}E_h[\log L_t] = K_g - K_f$$

Case 1: When $K_g > K_f$

Here, f is "closer" to h than g is. Since $K_g - K_f > 0$

$$E_h[\log L_t] = t \cdot (K_g - K_f) \to +\infty \text{ as } t \to \infty$$

By the Law of Large Numbers, $\frac{1}{t}\log L_t \to K_g - K_f > 0$ almost surely.

Therefore $L_t \to +\infty$ almost surely.

Case 2: When $K_g < K_f$

Here, g is "closer" to h than f is. Since $K_g - K_f < 0$

$$E_h[\log L_t] = t \cdot (K_g - K_f) \to -\infty \text{ as } t \to \infty$$

Therefore by similar reasoning $L_t \to 0$ almost surely.

CHAPTER

TWENTYTHREE

HETEROGENEOUS BELIEFS AND FINANCIAL MARKETS

Contents

- *Heterogeneous Beliefs and Financial Markets*
  - *Overview*
  - *Review: likelihood ratio processes*
  - *Blume and Easley's setting*
  - *Nature and agents' beliefs*
  - *A socialist risk-sharing arrangement*
  - *Social planner's allocation problem*
  - *If you're so smart, …*
  - *Competitive equilibrium prices*
  - *Simulations*
  - *Related lectures*
  - *Exercises*

23.1 Overview

A likelihood ratio process lies behind Lawrence Blume and David Easley's answer to their question "If you're so smart, why aren't you rich?" [Blume and Easley, 2006].

Blume and Easley constructed formal models to study how differences of opinions about probabilities governing risky income processes would influence outcomes and be reflected in prices of stocks, bonds, and insurance policies that individuals use to share and hedge risks.

Note

[Alchian, 1950] and [Friedman, 1953] conjectured that, by rewarding traders with more realistic probability models, competitive markets in financial securities put wealth in the hands of better informed traders and help make prices of risky assets reflect realistic probability assessments.

Here we'll provide an example that illustrates basic components of Blume and Easley's analysis.

We'll focus only on their analysis of an environment with complete markets in which trades in all conceivable risky securities are possible.

We'll study two alternative arrangements:

- perfect socialism in which individuals surrender their endowments of consumption goods each period to a central planner who then dictatorially allocates those goods
- a decentralized system of competitive markets in which selfish price-taking individuals voluntarily trade with each other in competitive markets

The fundamental theorems of welfare economics will apply and assure us that these two arrangements end up producing exactly the same allocation of consumption goods to individuals **provided** that the social planner assigns an appropriate set of **Pareto weights**.

Note

You can learn about how the two welfare theorems are applied in modern macroeconomic models in *this lecture on a planning problem* and *this lecture on a related competitive equilibrium*. *This quantecon lecture* presents a recursive formulation of complete markets models with homogeneous beliefs.

Let's start by importing some Python tools.

```
import matplotlib.pyplot as plt
import numpy as np
from numba import vectorize, jit, prange
from math import gamma
from scipy.integrate import quad
```

23.2 Review: likelihood ratio processes

We'll begin by reminding ourselves definitions and properties of likelihood ratio processes.

A nonnegative random variable W has one of two probability density functions, either f or g.

Before the beginning of time, nature once and for all decides whether she will draw a sequence of IID draws from either f or g.

We will sometimes let q be the density that nature chose once and for all, so that q is either f or g, permanently.

Nature knows which density it permanently draws from, but we the observers do not.

We know both f and g but we don't know which density nature chose.

But we want to know.

To do that, we use observations.

We observe a sequence $\{w_t\}_{t=1}^T$ of T IID draws that we know came from either f or g.

We want to use these observations to infer whether nature chose f or g.

A **likelihood ratio process** is a useful tool for this task.

To begin, we define a key component of a likelihood ratio process, namely, the time t likelihood ratio as the random variable

$$\ell(w_t) = \frac{f(w_t)}{g(w_t)}, \quad t \geq 1.$$

We assume that f and g both put positive probabilities on the same intervals of possible realizations of the random variable W.

That means that under the g density, $\ell(w_t) = \frac{f(w_t)}{g(w_t)}$ is a nonnegative random variable with mean 1.

A **likelihood ratio process** for sequence $\{w_t\}_{t=1}^{\infty}$ is defined as

$$L(w^t) = \prod_{i=1}^{t} \ell(w_i),$$

where $w^t = \{w_1, \ldots, w_t\}$ is a history of observations up to and including time t.

Sometimes for shorthand we'll write $L_t = L(w^t)$.

Notice that the likelihood process satisfies the *recursion*

$$L(w^t) = \ell(w_t) L(w^{t-1}).$$

The likelihood ratio and its logarithm are key tools for making inferences using a classic frequentist approach due to Neyman and Pearson [Neyman and Pearson, 1933].

To help us appreciate how things work, the following Python code evaluates f and g as two different Beta distributions, then computes and simulates an associated likelihood ratio process by generating a sequence w^t from one of the two probability distributions, for example, a sequence of IID draws from g.

```
# Parameters in the two Beta distributions.
F_a, F_b = 1, 1
G_a, G_b = 3, 1.2

@vectorize
def p(x, a, b):
    r = gamma(a + b) / (gamma(a) * gamma(b))
    return r * x** (a-1) * (1 - x) ** (b-1)

# The two density functions.
f = jit(lambda x: p(x, F_a, F_b))
g = jit(lambda x: p(x, G_a, G_b))
```

```
@jit
def simulate(a, b, T=50, N=500):
    '''
    Generate N sets of T observations of the likelihood ratio,
    return as N x T matrix.
    '''

    l_arr = np.empty((N, T))

    for i in range(N):
        for j in range(T):
            w = np.random.beta(a, b)
            l_arr[i, j] = f(w) / g(w)

    return l_arr
```

23.3 Blume and Easley's setting

Let the random variable $s_t \in (0, 1)$ at time $t = 0, 1, 2, \ldots$ be distributed according to the same Beta distribution with parameters $\theta = \{\theta_1, \theta_2\}$.

We'll denote this probability density as

$$\pi(s_t|\theta)$$

Below, we'll often just write $\pi(s_t)$ instead of $\pi(s_t|\theta)$ to save space.

Let $s_t \equiv y_t^1$ be the endowment of a nonstorable consumption good that a person we'll call "agent 1" receives at time t.

Let a history $s^t = [s_t, s_{t-1}, \ldots, s_0]$ be a sequence of i.i.d. random variables with joint distribution

$$\pi_t(s^t) = \pi(s_t)\pi(s_{t-1}) \cdots \pi(s_0)$$

So in our example, the history s^t is a comprehensive record of agent 1's endowments of the consumption good from time 0 up to time t.

If agent 1 were to live on an island by himself, agent 1's consumption $c^1(s_t)$ at time t is

$$c^1(s_t) = y_t^1 = s_t.$$

But in our model, agent 1 is not alone.

23.4 Nature and agents' beliefs

Nature draws i.i.d. sequences $\{s_t\}_{t=0}^{\infty}$ from $\pi_t(s^t)$.

- so π without a superscript is nature's model
- but in addition to nature, there are other entities inside our model – artificial people that we call "agents"
- each agent has a sequence of probability distributions over s^t for $t = 0, \ldots$
- agent i thinks that nature draws i.i.d. sequences $\{s_t\}_{t=0}^{\infty}$ from $\{\pi_t^i(s^t)\}_{t=0}^{\infty}$
  - agent i is mistaken unless $\pi_t^i(s^t) = \pi_t(s^t)$

Note

A **rational expectations** model would set $\pi_t^i(s^t) = \pi_t(s^t)$ for all agents i.

There are two agents named $i = 1$ and $i = 2$.

At time t, agent 1 receives an endowment

$$y_t^1 = s_t$$

of a nonstorable consumption good, while agent 2 receives an endowment of

$$y_t^2 = 1 - s_t$$

The aggregate endowment of the consumption good is

$$y_t^1 + y_t^2 = 1$$

at each date $t \geq 0$.

At date t agent i consumes $c_t^i(s^t)$ of the good.

A (non wasteful) feasible allocation of the aggregate endowment of 1 each period satisfies

$$c_t^1 + c_t^2 = 1.$$

23.5 A socialist risk-sharing arrangement

In order to share risks, a benevolent social planner dictates a history-dependent consumption allocation that takes the form of a sequence of functions

$$c_t^i = c_t^i(s^t)$$

that satisfy

$$c_t^1(s^t) + c_t^2(s^t) = 1 \tag{23.1}$$

for all s^t for all $t \geq 0$.

To design a socially optimal allocation, the social planner wants to know what each agent i believes about the endowment sequence and how they feel about bearing risks.

As for the endowment sequences, agent i believes that nature draws i.i.d. sequences from joint densities

$$\pi_t^i(s^t) = \pi^i(s_t)\pi^i(s_{t-1}) \cdots \pi^i(s_0)$$

As for attitudes toward bearing risks, agent i has a one-period utility function

$$u(c_t^i) = \ln(c_t^i)$$

with marginal utility of consumption in period t

$$u'(c_t^i) = \frac{1}{c_t^i}$$

Putting its beliefs about its random endowment sequence and its attitudes toward bearing risks together, agent i has intertemporal utility function

$$V^i = \sum_{t=0}^{\infty} \sum_{s^t} \delta^t u(c_t^i(s^t))\pi_t^i(s^t), \tag{23.2}$$

where $\delta \in (0, 1)$ is an intertemporal discount factor, and $u(\cdot)$ is a strictly increasing, concave one-period utility function.

23.6 Social planner's allocation problem

The benevolent dictator has all the information it requires to choose a consumption allocation that maximizes the social welfare criterion

$$W = \lambda V^1 + (1 - \lambda)V^2 \tag{23.3}$$

where $\lambda \in [0, 1]$ is a Pareto weight that tells how much the planner likes agent 1 and $1 - \lambda$ is a Pareto weight that tells how much the social planner likes agent 2.

Setting $\lambda = .5$ expresses "egalitarian" social preferences.

Notice how social welfare criterion (23.3) takes into account both agents' preferences as represented by formula (23.2).

This means that the social planner knows and respects

- each agent's one period utility function $u(\cdot) = \ln(\cdot)$
- each agent i's probability model $\{\pi_t^i(s^t)\}_{t=0}^{\infty}$

Consequently, it is natural to anticipate that these objects will appear in the social planner's rule for allocating the aggregate endowment each period.

First-order necessary conditions for maximizing welfare criterion (23.3) subject to the feasibility constraint (23.1) are

$$\frac{\pi_t^2(s^t)}{\pi_t^1(s^t)} \frac{(1/c_t^2(s^t))}{(1/c_t^1(s^t))} = \frac{\lambda}{1-\lambda}$$

which can be rearranged to become

$$\frac{c_t^1(s^t)}{c_t^2(s^t)} = \frac{\lambda}{1-\lambda} l_t(s^t) \tag{23.4}$$

where

$$l_t(s^t) = \frac{\pi_t^1(s^t)}{\pi_t^2(s^t)}$$

is the likelihood ratio of agent 1's joint density to agent 2's joint density.

Using

$$c_t^1(s^t) + c_t^2(s^t) = 1$$

we can rewrite allocation rule (23.4) as

$$\frac{c_t^1(s^t)}{1 - c_t^1(s^t)} = \frac{\lambda}{1-\lambda} l_t(s^t)$$

or

$$c_t^1(s^t) = \frac{\lambda}{1-\lambda} l_t(s^t)(1 - c_t^1(s^t))$$

which implies that the social planner's allocation rule is

$$c_t^1(s^t) = \frac{\lambda l_t(s^t)}{1 - \lambda + \lambda l_t(s^t)} \tag{23.5}$$

If we define a temporary or **continuation Pareto weight** process as

$$\lambda_t(s^t) = \frac{\lambda l_t(s^t)}{1 - \lambda + \lambda l_t(s^t)},$$

then we can represent the social planner's allocation rule as

$$c_t^1(s^t) = \lambda_t(s^t)$$

and of course

$$c_t^2(s^t) = 1 - \lambda_t(s^t).$$

23.7 If you're so smart, …

Let's compute some values of limiting allocations (23.5) for some interesting possible limiting values of the likelihood ratio process $l_t(s^t)$.

As our first case, let's suppose that

$$l_\infty(s^\infty) = 1; \quad c^1_\infty = \lambda$$

- In this case, both agents are equally smart (or equally not smart) and the consumption allocation stays put at a $\lambda, 1-\lambda$ split between the two agents.

As our second case, let suppose that

$$l_\infty(s^\infty) = 0; \quad c^1_\infty = 0$$

- In this case, agent 2 is "smarter" than agent 1, and agent 1's share of the aggregate endowment converges to zero.

As our third case, let's suppose that

$$l_\infty(s^\infty) = \infty; \quad c^1_\infty = 1$$

- In this case, agent 1 is smarter than agent 2, and agent 1's share of the aggregate endowment converges to 1.

Note

These three cases are somehow telling us about how relative **wealths** of the agents evolve as time passes.

- when the two agents are equally smart and $\lambda \in (0,1)$, agent 1's wealth share stays at λ perpetually.
- when agent 1 is smarter and $\lambda \in (0,1)$, agent 1 eventually "owns" the entire continuation endowment and agent 2 eventually "owns" nothing.
- when agent 2 is smarter and $\lambda \in (0,1)$, agent 2 eventually "owns" the entire continuation endowment and agent 1 eventually "owns" nothing.

We can define continuation wealths precisely after we construct a competitive equilibrium **price** system below.

Soon we'll do some simulations that will shed further light on possible outcomes.

But before we do that, let's take a detour and study some "shadow prices" for the social planning problem.

We shall soon see that these shadow prices can readily be converted to "equilibrium prices" for a competitive equilibrium.

Doing this will allow us to connect our analysis with claims by [Alchian, 1950] and [Friedman, 1953] that transfers of wealth coming from trades in competitive asset markets eventually make prices of risky assets reflect realistic probability assessments.

23.8 Competitive equilibrium prices

Two fundamental welfare theorems for general equilibrium models lead us to anticipate that there is a connection between the allocation that solves the social planning problem we have been studying and the allocation in a **competitive equilibrium** with complete markets in contingent claims to time- and history-dependent consumption goods.

Note

For the two welfare theorems and their history, see https://en.wikipedia.org/wiki/Fundamental_theorems_of_welfare_economics. Again, for applications to a classic macroeconomic growth model, see *this lecture on a planning problem* and *this lecture on a related competitive equilibrium*

Such a connection prevails for our model.

We'll sketch it now.

In a competitive equilibrium, no social planner dictatorially collects everybody's endowments and then reallocates them.

Instead, there is a comprehensive centralized market that meets at one point in time.

There are **prices** at which price-taking agents can buy or sell whatever goods that they want.

Trade is multilateral in the sense that that there is a "Walrasian auctioneer" who lives outside the model and whose job it is to verify that each agent's budget constraint is satisfied.

That budget constraint requires that the total value of the agent's endowment stream be at least as the total value of its consumption stream.

These values are computed at price vectors that the agents take as given – the agents are "price-takers" who assume that they can buy or sell whatever quantities that they want at those prices.

Suppose that at time -1, before time 0 starts, agent i can purchase one unit $c_t(s^t)$ of consumption at time t after history s^t at price $p_t(s^t)$.

Notice that there is (very long) **vector** of prices.

- there is one price $p_t(s^t)$ for each history s^t at every date $t = 0, 1, \ldots,$.
- so there are as many prices as there are histories and dates.

These prices determined at time -1 before the economy starts.

The market meets once at time -1.

At times $t = 0, 1, 2, \ldots$ trades made at time -1 are simply executed, i.e., the promised deliveries are made.

- in the background, there is an "enforcement" procedure that forces agents to carry out the exchanges or "deliveries" that they agreed to at time -1.

We want to study how agents' probability models influence equilibrium prices.

Agent i faces a **single** intertemporal budget constraint

$$\sum_{t=0}^{\infty} \sum_{s^t} p_t(s^t) c_t^i(s^t) \leq \sum_{t=0}^{\infty} \sum_{s^t} p_t(s^t) y_t^i(s^t) \tag{23.6}$$

According to budget constraint (23.6), trade is **multilateral** in the following sense

- imagine that agent i first sells his random endowment stream $\{y_t^i(s^t)\}$ and then uses the proceeds (i.e., his "wealth") to purchase a random consumption stream $\{c_t^i(s^t)\}$.

Agent i attaches a Lagrange multiplier μ_i to budget constraint (23.6) and once-and-for-all chooses a consumption plan $\{c_t^i(s^t)\}_{t=0}^{\infty}$ that maximizes criterion (23.2) subject to budget constraint (23.6).

This means that the agent i chooses many objects, namely, $c_t^i(s^t)$ for all s^t for $t = 0, 1, 2, \ldots$.

For convenience, let's remind ourselves of criterion V^i defined in (23.2):

$$V^i = \sum_{t=0}^{\infty} \sum_{s^t} \delta^t u(c_t^i(s^t)) \pi_t^i(s^t)$$

First-order necessary conditions for maximizing objective V^i defined in (23.2) with respect to $c_t^i(s^t)$ are

$$\delta^t u'(c_t^i(s^t))\pi_t^i(s^t) = \mu_i p_t(s^t),$$

which we can rearrange to obtain

$$p_t(s^t) = \frac{\delta^t \pi_t^i(s^t)}{\mu_i c_t^i(s^t)} \tag{23.7}$$

for $i = 1, 2$.

If we divide equation (23.7) for agent 1 by the appropriate version of equation (23.7) for agent 2, impose the feasibility condition $c_t^2(s^t) = 1 - c_t^1(s^t)$, and do some algebra, we'll obtain

$$c_t^1(s^t) = \frac{\mu_1 l_t(s^t)}{\mu_2 + \mu_1 l_t(s^t)}. \tag{23.8}$$

We now embark on an extended "guess-and-verify" exercise that involves matching objects in our competitive equilibrium with objects in our social planning problem.

- we'll match consumption allocations in the planning problem with equilibrium consumption allocations in the competitive equilibrium
- we'll match "shadow" prices in the planning problem with competitive equilibrium prices.

Notice that if we set $\mu_1 = 1 - \lambda$ and $\mu_2 = \lambda$, then formula (23.8) agrees with formula (23.5).

- doing this amounts to choosing a **numeraire** or normalization for the price system $\{p_t(s^t)\}_{t=0}^{\infty}$

Note

For information about how a numeraire must be chosen to pin down the absolute price level in a model like ours that determines only relative prices, see https://en.wikipedia.org/wiki/Numéraire.

If we substitute formula (23.8) for $c_t^1(s^t)$ into formula (23.7) and rearrange, we obtain

$$p_t(s^t) = \frac{\delta^t}{\lambda(1-\lambda)}\pi_t^2(s^t)[1 - \lambda + \lambda l_t(s^t)]$$

or

$$p_t(s^t) = \frac{\delta^t}{\lambda(1-\lambda)}[(1-\lambda)\pi_t^2(s^t) + \lambda\pi_t^1(s^t)] \tag{23.9}$$

According to formula (23.9), we have the following possible limiting cases:

- when $l_\infty = 0$, $c_\infty^1 = 0$ and tails of competitive equilibrium prices reflect agent 2's probability model $\pi_t^2(s^t)$ according to $p_t(s^t) \propto \delta^t \pi_t^2(s^t)$
- when $l_\infty = \infty$, $c_\infty^1 = 1$ and tails of competitive equilibrium prices reflect agent 1's probability model $\pi_t^1(s^t)$ according to $p_t(s^t) \propto \delta^t \pi_t^1(s^t)$
- for small t's, competitive equilibrium prices reflect both agents' probability models.

We leave the verification of the shadow prices to the reader since it follows from the same reasoning.

23.9 Simulations

Now let's implement some simulations when agent 1 believes marginal density

$$\pi^1(s_t) = f(s_t)$$

and agent 2 believes marginal density

$$\pi^2(s_t) = g(s_t)$$

where f and g are Beta distributions like ones that we used in earlier sections of this lecture.

Meanwhile, we'll assume that nature believes a marginal density

$$\pi(s_t) = h(s_t)$$

where $h(s_t)$ is perhaps a mixture of f and g.

First, we write a function to compute the likelihood ratio process

```
def compute_likelihood_ratios(sequences, f, g):
    """Compute likelihood ratios and cumulative products."""
    l_ratios = f(sequences) / g(sequences)
    L_cumulative = np.cumprod(l_ratios, axis=1)
    return l_ratios, L_cumulative
```

Let's compute the Kullback–Leibler discrepancies by quadrature integration.

```
def compute_KL(f, g):
    """
    Compute KL divergence KL(f, g)
    """
    integrand = lambda w: f(w) * np.log(f(w) / g(w))
    val, _ = quad(integrand, 1e-5, 1-1e-5)
    return val
```

We also create a helper function to compute KL divergence with respect to a reference distribution h

```
def compute_KL_h(h, f, g):
    """
    Compute KL divergence with reference distribution h
    """

    Kf = compute_KL(h, f)
    Kg = compute_KL(h, g)

    return Kf, Kg
```

Let's write a Python function that computes agent 1's consumption share

```
def simulate_blume_easley(sequences, f_belief=f, g_belief=g, λ=0.5):
    """Simulate Blume-Easley model consumption shares."""
    l_ratios, l_cumulative = compute_likelihood_ratios(sequences, f_belief, g_belief)
    c1_share = λ * l_cumulative / (1 - λ + λ * l_cumulative)
    return l_cumulative, c1_share
```

Now let's use this function to generate sequences in which

- nature draws from f each period, or

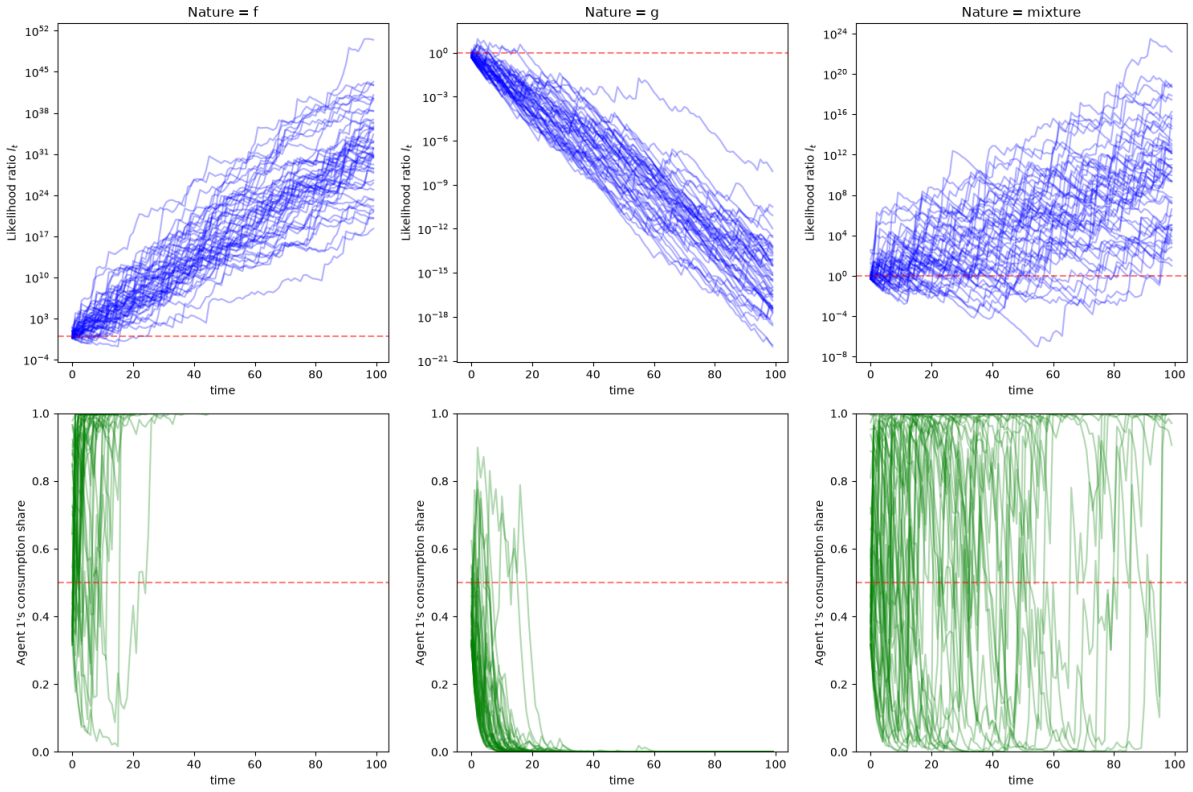
- nature draws from g each period, or
- nature flips a fair coin each period to decide whether to draw from f or g

```
λ = 0.5
T = 100
N = 10000

# Nature follows f, g, or mixture
s_seq_f = np.random.beta(F_a, F_b, (N, T))
s_seq_g = np.random.beta(G_a, G_b, (N, T))

h = jit(lambda x: 0.5 * f(x) + 0.5 * g(x))
model_choices = np.random.rand(N, T) < 0.5
s_seq_h = np.empty((N, T))
s_seq_h[model_choices] = np.random.beta(F_a, F_b, size=model_choices.sum())
s_seq_h[~model_choices] = np.random.beta(G_a, G_b, size=(~model_choices).sum())

l_cum_f, c1_f = simulate_blume_easley(s_seq_f)
l_cum_g, c1_g = simulate_blume_easley(s_seq_g)
l_cum_h, c1_h = simulate_blume_easley(s_seq_h)
```

Before looking at the figure below, have some fun by guessing whether agent 1 or agent 2 will have a larger and larger consumption share as time passes in our three cases.

To make better guesses, let's visualize instances of the likelihood ratio processes in the three cases.

```
fig, axes = plt.subplots(2, 3, figsize=(15, 10))

titles = ["Nature = f", "Nature = g", "Nature = mixture"]
data_pairs = [(l_cum_f, c1_f), (l_cum_g, c1_g), (l_cum_h, c1_h)]

for i, ((l_cum, c1), title) in enumerate(zip(data_pairs, titles)):
    # Likelihood ratios
    ax = axes[0, i]
    for j in range(min(50, l_cum.shape[0])):
        ax.plot(l_cum[j, :], alpha=0.3, color='blue')
    ax.set_yscale('log')
    ax.set_xlabel('time')
    ax.set_ylabel('Likelihood ratio $l_t$')
    ax.set_title(title)
    ax.axhline(y=1, color='red', linestyle='--', alpha=0.5)

    # Consumption shares
    ax = axes[1, i]
    for j in range(min(50, c1.shape[0])):
        ax.plot(c1[j, :], alpha=0.3, color='green')
    ax.set_xlabel('time')
    ax.set_ylabel("Agent 1's consumption share")
    ax.set_ylim([0, 1])
    ax.axhline(y=λ, color='red', linestyle='--', alpha=0.5)

plt.tight_layout()
plt.show()
```



In the left panel, nature chooses f. Agent 1's consumption reaches 1 very quickly.

In the middle panel, nature chooses g. Agent 1's consumption ratio tends to move towards 0 but not as fast as in the first case.

In the right panel, nature flips coins each period. We see a very similar pattern to the processes in the left panel.

The figures in the top panel remind us of the discussion in *this section*.

We invite readers to revisit *that section* and try to infer the relationships among $D_{KL}(f\|g)$, $D_{KL}(g\|f)$, $D_{KL}(h\|f)$, and $D_{KL}(h\|g)$.

Let's compute values of KL divergence

```
shares = [np.mean(c1_f[:, -1]), np.mean(c1_g[:, -1]), np.mean(c1_h[:, -1])]
Kf_g, Kg_f = compute_KL(f, g), compute_KL(g, f)
Kf_h, Kg_h = compute_KL_h(h, f, g)

print(f"Final shares: f={shares[0]:.3f}, g={shares[1]:.3f}, mix={shares[2]:.3f}")
print(f"KL divergences: \nKL(f,g)={Kf_g:.3f}, KL(g,f)={Kg_f:.3f}")
print(f"KL(h,f)={Kf_h:.3f}, KL(h,g)={Kg_h:.3f}")
```

```
Final shares: f=1.000, g=0.000, mix=0.927
KL divergences:
KL(f,g)=0.759, KL(g,f)=0.344
KL(h,f)=0.073, KL(h,g)=0.281
```

We find that $KL(f,g) > KL(g,f)$ and $KL(h,g) > KL(h,f)$.

The first inequality tells us that the average "surprise" from having belief g when nature chooses f is greater than the "surprise" from having belief f when nature chooses g.

This explains the difference between the first two panels we noted above.

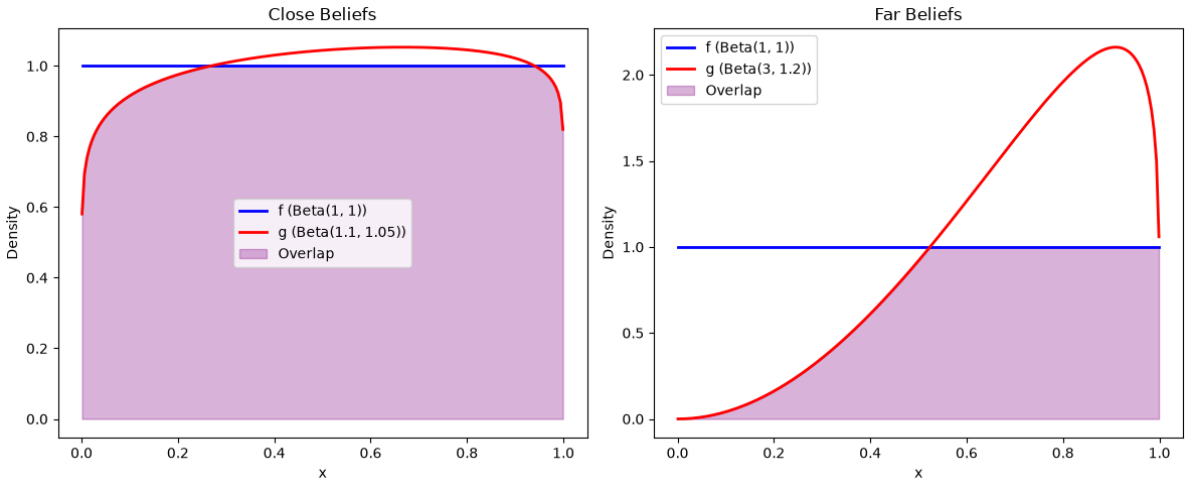
The second inequality tells us that agent 1's belief distribution f is closer to nature's pick than agent 2's belief g.

To make this idea more concrete, let's compare two cases:

- agent 1's belief distribution f is close to agent 2's belief distribution g;
- agent 1's belief distribution f is far from agent 2's belief distribution g.

We use the two distributions visualized below

```
def plot_distribution_overlap(ax, x_range, f_vals, g_vals,
                              f_label='f', g_label='g',
                              f_color='blue', g_color='red'):
    """Plot two distributions with their overlap region."""
    ax.plot(x_range, f_vals, color=f_color, linewidth=2, label=f_label)
    ax.plot(x_range, g_vals, color=g_color, linewidth=2, label=g_label)

    overlap = np.minimum(f_vals, g_vals)
    ax.fill_between(x_range, 0, overlap, alpha=0.3, color='purple', label='Overlap')
    ax.set_xlabel('x')
    ax.set_ylabel('Density')
    ax.legend()

# Define close and far belief distributions
f_close = jit(lambda x: p(x, 1, 1))
g_close = jit(lambda x: p(x, 1.1, 1.05))

f_far = jit(lambda x: p(x, 1, 1))
g_far = jit(lambda x: p(x, 3, 1.2))

# Visualize the belief distributions
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))

x_range = np.linspace(0.001, 0.999, 200)

# Close beliefs
f_close_vals = [f_close(x) for x in x_range]
g_close_vals = [g_close(x) for x in x_range]
plot_distribution_overlap(ax1, x_range, f_close_vals, g_close_vals,
                          f_label='f (Beta(1, 1))', g_label='g (Beta(1.1, 1.05))')
ax1.set_title(f'Close Beliefs')

# Far beliefs
f_far_vals = [f_far(x) for x in x_range]
g_far_vals = [g_far(x) for x in x_range]
plot_distribution_overlap(ax2, x_range, f_far_vals, g_far_vals,
                          f_label='f (Beta(1, 1))', g_label='g (Beta(3, 1.2))')
ax2.set_title(f'Far Beliefs')

plt.tight_layout()
plt.show()
```



Let's draw the same consumption ratio plots as above for agent 1.

We replace the simulation paths with median and percentiles to make the figure cleaner.

Staring at the figure below, can we infer the relation between $KL(f,g)$ and $KL(g,f)$?

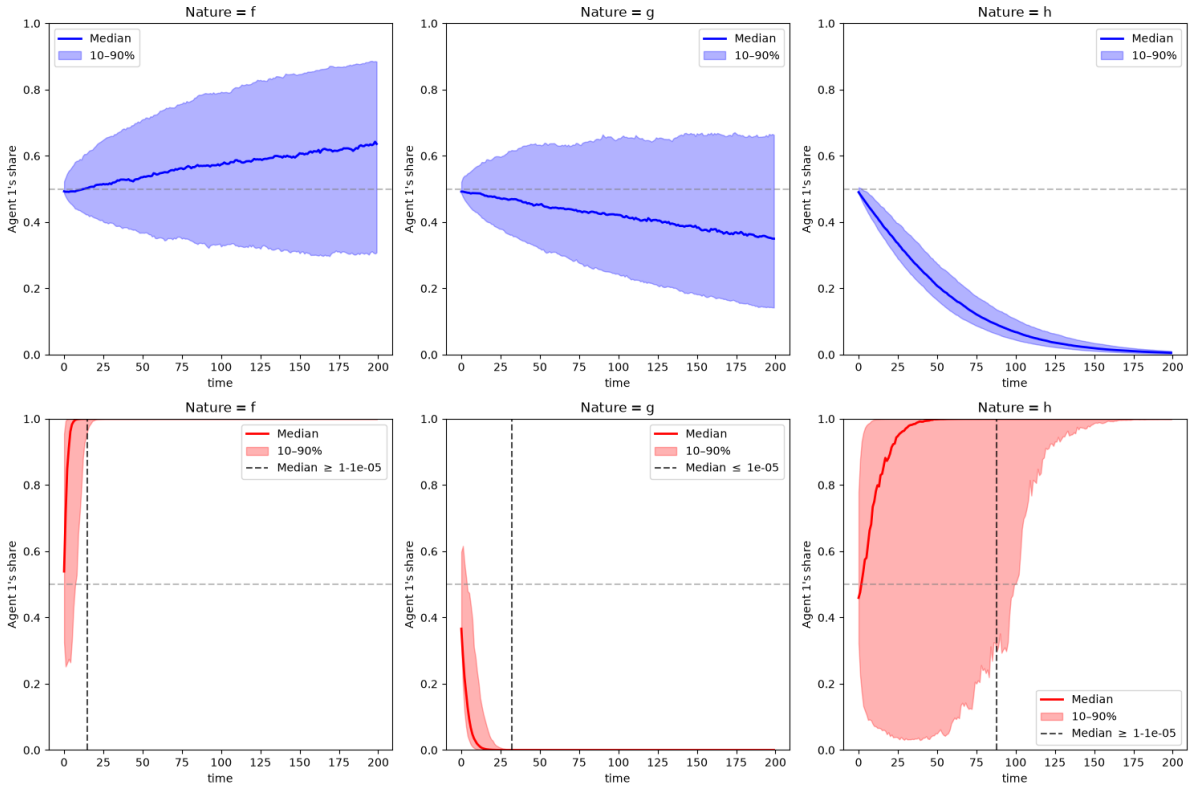
From the right panel, can we infer the relation between $KL(h,g)$ and $KL(h,f)$?

```
fig, axes = plt.subplots(2, 3, figsize=(15, 10))
nature_params = {'close': [(1, 1), (1.1, 1.05), (2, 1.5)],
                 'far':   [(1, 1), (3, 1.2),    (2, 1.5)]}
nature_labels = ["Nature = f", "Nature = g", "Nature = h"]
colors = {'close': 'blue', 'far': 'red'}

threshold = 1e-5  # "close to zero" cutoff

for row, (f_belief, g_belief, label) in enumerate([
                        (f_close, g_close, 'close'),
                        (f_far, g_far, 'far')]):

    for col, nature_label in enumerate(nature_labels):
        params = nature_params[label][col]
        s_seq = np.random.beta(params[0], params[1], (1000, 200))
        _, c1 = simulate_blume_easley(s_seq, f_belief, g_belief, λ)

        median_c1 = np.median(c1, axis=0)
        p10, p90 = np.percentile(c1, [10, 90], axis=0)

        ax = axes[row, col]
        color = colors[label]
        ax.plot(median_c1, color=color, linewidth=2, label='Median')
        ax.fill_between(range(len(median_c1)), p10, p90, alpha=0.3, color=color,␣
↪label='10-90%')
        ax.set_xlabel('time')
        ax.set_ylabel("Agent 1's share")
        ax.set_ylim([0, 1])
        ax.set_title(nature_label)
        ax.axhline(y=λ, color='gray', linestyle='--', alpha=0.5)
        below = np.where(median_c1 < threshold)[0]
        above = np.where(median_c1 > 1-threshold)[0]
        if below.size > 0: first_zero = (below[0], True)
        elif above.size > 0: first_zero = (above[0], False)
```

(continues on next page)

(continued from previous page)

```
        else: first_zero = None
        if first_zero is not None:
            ax.axvline(x=first_zero[0], color='black', linestyle='--',
                       alpha=0.7,
                       label=fr'Median $\leq$ {threshold}' if first_zero[1]
                       else fr'Median $\geq$ 1-{threshold}')
        ax.legend()

plt.tight_layout()
plt.show()
```



Holding to our guesses, let's calculate the four values

```
# Close case
Kf_g, Kg_f = compute_KL(f_close, g_close), compute_KL(g_close, f_close)
Kf_h, Kg_h = compute_KL_h(h, f_close, g_close)

print(f"KL divergences (close): \nKL(f,g)={Kf_g:.3f}, KL(g,f)={Kg_f:.3f}")
print(f"KL(h,f)={Kf_h:.3f}, KL(h,g)={Kg_h:.3f}")

# Far case
Kf_g, Kg_f = compute_KL(f_far, g_far), compute_KL(g_far, f_far)
Kf_h, Kg_h = compute_KL_h(h, f_far, g_far)

print(f"KL divergences (far): \nKL(f,g)={Kf_g:.3f}, KL(g,f)={Kg_f:.3f}")
print(f"KL(h,f)={Kf_h:.3f}, KL(h,g)={Kg_h:.3f}")
```

```
KL divergences (close):
```

(continues on next page)

(continued from previous page)

```
KL(f,g)=0.003, KL(g,f)=0.003
KL(h,f)=0.073, KL(h,g)=0.061
KL divergences (far):
KL(f,g)=0.759, KL(g,f)=0.344
KL(h,f)=0.073, KL(h,g)=0.281
```

We find that in the first case, $KL(f,g) \approx KL(g,f)$ and both are relatively small, so although either agent 1 or agent 2 will eventually consume everything, convergence displayed in the first two panels on the top is pretty slow.

In the first two panels at the bottom, we see convergence occurring faster (as indicated by the black dashed line) because the divergence gaps $KL(f,g)$ and $KL(g,f)$ are larger.

Since $KL(f,g) > KL(g,f)$, we see faster convergence in the first panel at the bottom when nature chooses f than in the second panel where nature chooses g.

This ties in nicely with (22.1).

23.10 Related lectures

Complete markets models with homogeneous beliefs, a kind often used in macroeconomics and finance, are studied in this quantecon lecture *Competitive Equilibria with Arrow Securities*.

[Blume *et al.*, 2018] discuss a paternalistic case against complete markets. They study the consequences of assuming that a social planner disregards individuals preferences in the sense that it ignores the subjective belief components of their preferences and replaces it with the social planner's beliefs about probabilities.

Likelihood processes play an important role in Bayesian learning, as described in *Likelihood Ratio Processes and Bayesian Learning* and as applied in *Job Search VIII: Search with Learning*.

Likelihood ratio processes appear again in Additive and Multiplicative Functionals.

23.11 Exercises

Exercise 23.11.1

Starting from (23.7), show that the competitive equilibrium prices can be expressed as

$$p_t(s^t) = \frac{\delta^t}{\lambda(1-\lambda)} \pi_t^2(s^t)[1 - \lambda + \lambda l_t(s^t)]$$

Solution

Starting from

$$p_t(s^t) = \frac{\delta^t \pi_t^i(s^t)}{\mu_i c_t^i(s^t)}, \qquad i = 1, 2.$$

Since both expressions equal the same price, we can equate them

$$\frac{\pi_t^1(s^t)}{\mu_1 c_t^1(s^t)} = \frac{\pi_t^2(s^t)}{\mu_2 c_t^2(s^t)}$$

Rearranging gives

$$\frac{c_t^1(s^t)}{c_t^2(s^t)} = \frac{\mu_2}{\mu_1} l_t(s^t)$$

where $l_t(s^t) \equiv \pi_t^1(s^t)/\pi_t^2(s^t)$ is the likelihood ratio process.

Using $c_t^2(s^t) = 1 - c_t^1(s^t)$:

$$\frac{c_t^1(s^t)}{1 - c_t^1(s^t)} = \frac{\mu_2}{\mu_1} l_t(s^t)$$

Solving for $c_t^1(s^t)$

$$c_t^1(s^t) = \frac{\mu_2 l_t(s^t)}{\mu_1 + \mu_2 l_t(s^t)}$$

The planner's solution gives

$$c_t^1(s^t) = \frac{\lambda l_t(s^t)}{1 - \lambda + \lambda l_t(s^t)}$$

To match agent 1's choice in a competitive equilibrium with the planner's choice for agent 1, the following equality must hold

$$\frac{\mu_2}{\mu_1} = \frac{\lambda}{1 - \lambda}$$

Hence we have

$$\mu_1 = 1 - \lambda, \qquad \mu_2 = \lambda$$

With $\mu_1 = 1 - \lambda$ and $c_t^1(s^t) = \frac{\lambda l_t(s^t)}{1-\lambda+\lambda l_t(s^t)}$, we have

$$\begin{aligned} p_t(s^t) &= \frac{\delta^t \pi_t^1(s^t)}{(1-\lambda) c_t^1(s^t)} \\ &= \frac{\delta^t \pi_t^1(s^t)}{(1-\lambda)} \cdot \frac{1 - \lambda + \lambda l_t(s^t)}{\lambda l_t(s^t)} \\ &= \frac{\delta^t \pi_t^1(s^t)}{(1-\lambda)\lambda l_t(s^t)} [1 - \lambda + \lambda l_t(s^t)]. \end{aligned}$$

Since $\pi_t^1(s^t) = l_t(s^t)\pi_t^2(s^t)$, we have

$$\begin{aligned} p_t(s^t) &= \frac{\delta^t l_t(s^t)\pi_t^2(s^t)}{(1-\lambda)\lambda l_t(s^t)} [1 - \lambda + \lambda l_t(s^t)] \\ &= \frac{\delta^t \pi_t^2(s^t)}{(1-\lambda)\lambda} [1 - \lambda + \lambda l_t(s^t)] \\ &= \frac{\delta^t}{\lambda(1-\lambda)} \pi_t^2(s^t) [1 - \lambda + \lambda l_t(s^t)]. \end{aligned}$$

Exercise 23.11.2

In this exercise, we'll study two agents, each of whom updates its posterior probability as data arrive.

- each agent applies Bayes' law in the way studied in *Likelihood Ratio Processes and Bayesian Learning*.

The following two models are on the table

$$f(s^t) = f(s_1)f(s_2)\cdots f(s_t)$$

and

$$g(s^t) = g(s_1)g(s_2)\cdots g(s_t)$$

as is an associated likelihood ratio process

$$L(s^t) = \frac{f(s^t)}{g(s^t)}.$$

Let $\pi_0 \in (0,1)$ be a prior probability and

$$\pi_t = \frac{\pi_0 L(s^t)}{\pi_0 L(s^t) + (1-\pi_0)}.$$

Each of our two agents deploys its own version of the mixture model

$$m(s^t) = \pi_t f(s^t) + (1-\pi_t)g(s^t) \tag{23.10}$$

We'll endow each type of consumer with model (23.10).

- The two agents share the same f and g, but
- they have different initial priors, say π_0^1 and π_0^2

Thus, consumer i's probability model is

$$m^i(s^t) = \pi_t^i f(s^t) + (1-\pi_t^i)g(s^t) \tag{23.11}$$

We now hand probability models (23.11) for $i = 1, 2$ to the social planner.

We want to deduce allocation $c^i(s^t), i = 1, 2$, and watch what happens when

- nature's model is f
- nature's model is g

We expect that consumers will eventually learn the "truth", but that one of them will learn faster.

To explore things, please set $f \sim \text{Beta}(1.5, 1)$ and $g \sim \text{Beta}(1, 1.5)$.

Please write Python code that answers the following questions.

- How do consumption shares evolve?
- Which agent learns faster when nature follows f?
- Which agent learns faster when nature follows g?
- How does a difference in initial priors π_0^1 and π_0^2 affect the convergence speed?

Solution

First, let's write helper functions that compute model components including each agent's subjective belief function.

```
def bayesian_update(π_0, L_t):
    """
    Bayesian update of belief probability given likelihood ratio.
    """
    return (π_0 * L_t) / (π_0 * L_t + (1 - π_0))

def mixture_density_belief(s_seq, f_func, g_func, π_seq):
    """
    Compute the mixture density beliefs m^i(s^t) for agent i.
    """
    f_vals = f_func(s_seq)
    g_vals = g_func(s_seq)
    return π_seq * f_vals + (1 - π_seq) * g_vals
```

Now let's write code that simulates the Blume-Easley model with our two agents.

```
def simulate_learning_blume_easley(sequences, f_belief, g_belief,
                                   π_0_1, π_0_2, λ=0.5):
    """
    Simulate Blume-Easley model with learning agents.
    """
    N, T = sequences.shape

    # Initialize arrays to store results
    π_1_seq = np.full((N, T), np.nan)
    π_2_seq = np.full((N, T), np.nan)
    c1_share = np.full((N, T), np.nan)
    l_agents_seq = np.full((N, T), np.nan)

    π_1_seq[:, 0] = π_0_1
    π_2_seq[:, 0] = π_0_2

    for n in range(N):
        # Initialize cumulative likelihood ratio for beliefs
        L_cumul = 1.0

        # Initialize likelihood ratio between agent densities
        l_agents_cumul = 1.0

        for t in range(1, T):
            s_t = sequences[n, t]

            # Compute likelihood ratio for this observation
            l_t = f_belief(s_t) / g_belief(s_t)

            # Update cumulative likelihood ratio
            L_cumul *= l_t

            # Bayesian update of beliefs
            π_1_t = bayesian_update(π_0_1, L_cumul)
            π_2_t = bayesian_update(π_0_2, L_cumul)

            # Store beliefs
            π_1_seq[n, t] = π_1_t
            π_2_seq[n, t] = π_2_t

            # Compute mixture densities for each agent
            m1_t = π_1_t * f_belief(s_t) + (1 - π_1_t) * g_belief(s_t)
```

```
            m2_t = π_2_t * f_belief(s_t) + (1 - π_2_t) * g_belief(s_t)

            # Update cumulative likelihood ratio between agents
            l_agents_cumul *= (m1_t / m2_t)
            l_agents_seq[n, t] = l_agents_cumul

            # c_t^1(s^t) = λ * l_t(s^t) / (1 - λ + λ * l_t(s^t))
            # where l_t(s^t) is the cumulative likelihood ratio between agents
            c1_share[n, t] = λ * l_agents_cumul / (1 - λ + λ * l_agents_cumul)

    return {
        'π_1': π_1_seq,
        'π_2': π_2_seq,
        'c1_share': c1_share,
        'l_agents': l_agents_seq
    }
```

Let's run simulations for different scenarios.

We use $\lambda = 0.5$, $T = 40$, and $N = 1000$.

```
λ = 0.5
T = 40
N = 1000

F_a, F_b = 1.5, 1
G_a, G_b = 1, 1.5

f = jit(lambda x: p(x, F_a, F_b))
g = jit(lambda x: p(x, G_a, G_b))
```

We'll start with different initial priors $\pi_0^i \in (0, 1)$ and widen the gap between them.

```
# Different initial priors
π_0_scenarios = [
    (0.3, 0.7),
    (0.7, 0.3),
    (0.1, 0.9),
]
```

Now we can run simulations for different scenarios

```
# Nature follows f
s_seq_f = np.random.beta(F_a, F_b, (N, T))

# Nature follows g
s_seq_g = np.random.beta(G_a, G_b, (N, T))

results_f = {}
results_g = {}

for i, (π_0_1, π_0_2) in enumerate(π_0_scenarios):
    # When nature follows f
    results_f[i] = simulate_learning_blume_easley(
            s_seq_f, f, g, π_0_1, π_0_2, λ)
    # When nature follows g
    results_g[i] = simulate_learning_blume_easley(
            s_seq_g, f, g, π_0_1, π_0_2, λ)
```

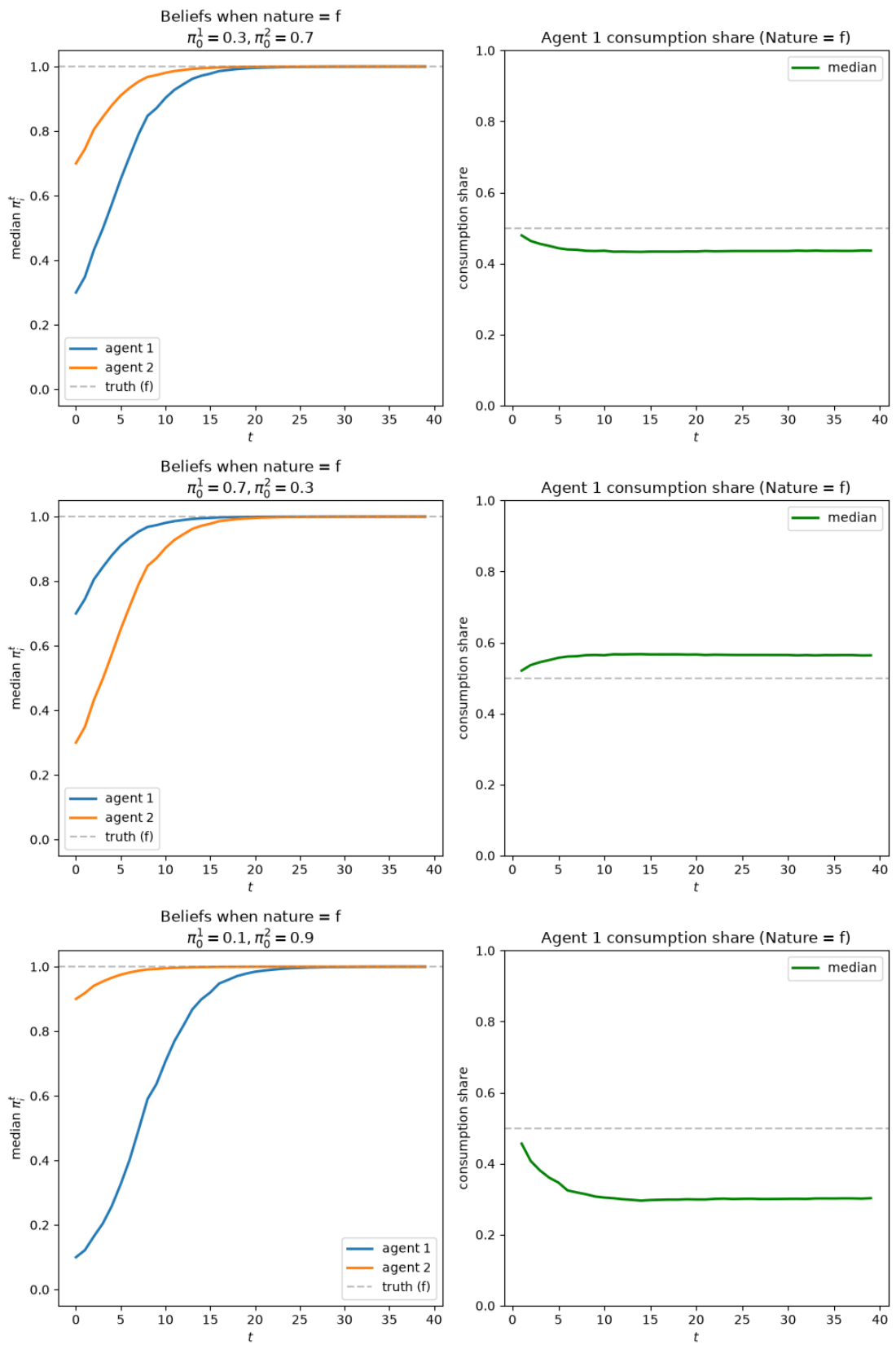
Let's visualize the results

```
def plot_learning_results(results, π_0_scenarios, nature_type, truth_value):
    """
```

```
    Plot beliefs and consumption shares for learning agents.
    """

    fig, axes = plt.subplots(3, 2, figsize=(10, 15))

    scenario_labels = [
        rf'$\pi_0^1 = {π_0_1}, \pi_0^2 = {π_0_2}$'
        for π_0_1, π_0_2 in π_0_scenarios
    ]

    for row, (scenario_idx, scenario_label) in enumerate(
                zip(range(3), scenario_labels)):

        res = results[scenario_idx]

        # Plot beliefs
        ax = axes[row, 0]
        π_1_med = np.median(res['π_1'], axis=0)
        π_2_med = np.median(res['π_2'], axis=0)
        ax.plot(π_1_med, 'C0', label=r'agent 1', linewidth=2)
        ax.plot(π_2_med, 'C1', label=r'agent 2', linewidth=2)
        ax.axhline(y=truth_value, color='gray', linestyle='--',
                   alpha=0.5, label=f'truth ({nature_type})')
        ax.set_title(f'Beliefs when nature = {nature_type}\n{scenario_label}')
        ax.set_ylabel(r'median $\pi_i^t$')
        ax.set_ylim([-0.05, 1.05])
        ax.legend()

        # Plot consumption shares
        ax = axes[row, 1]
        c1_med = np.median(res['c1_share'], axis=0)
        ax.plot(c1_med, 'g-', linewidth=2, label='median')
        ax.axhline(y=0.5, color='gray', linestyle='--',
                   alpha=0.5)
        ax.set_title(f'Agent 1 consumption share (Nature = {nature_type})')
        ax.set_ylabel('consumption share')
        ax.set_ylim([0, 1])
        ax.legend()

        # Add x-labels
        for col in range(2):
            axes[row, col].set_xlabel('$t$')

    plt.tight_layout()
    return fig, axes
```

Now we'll plot outcome when nature follows f:

```
fig_f, axes_f = plot_learning_results(
                results_f, π_0_scenarios, 'f', 1.0)
plt.show()
```


Beliefs when nature = f
$\pi_0^1 = 0.3, \pi_0^2 = 0.7$
median π_t^i
agent 1
agent 2
truth (f)
t
Agent 1 consumption share (Nature = f)
median
consumption share
t
Beliefs when nature = f
$\pi_0^1 = 0.7, \pi_0^2 = 0.3$
median π_t^i
agent 1
agent 2
truth (f)
t
Agent 1 consumption share (Nature = f)
median
consumption share
t
Beliefs when nature = f
$\pi_0^1 = 0.1, \pi_0^2 = 0.9$
median π_t^i
agent 1
agent 2
truth (f)
t
Agent 1 consumption share (Nature = f)
median
consumption share
t

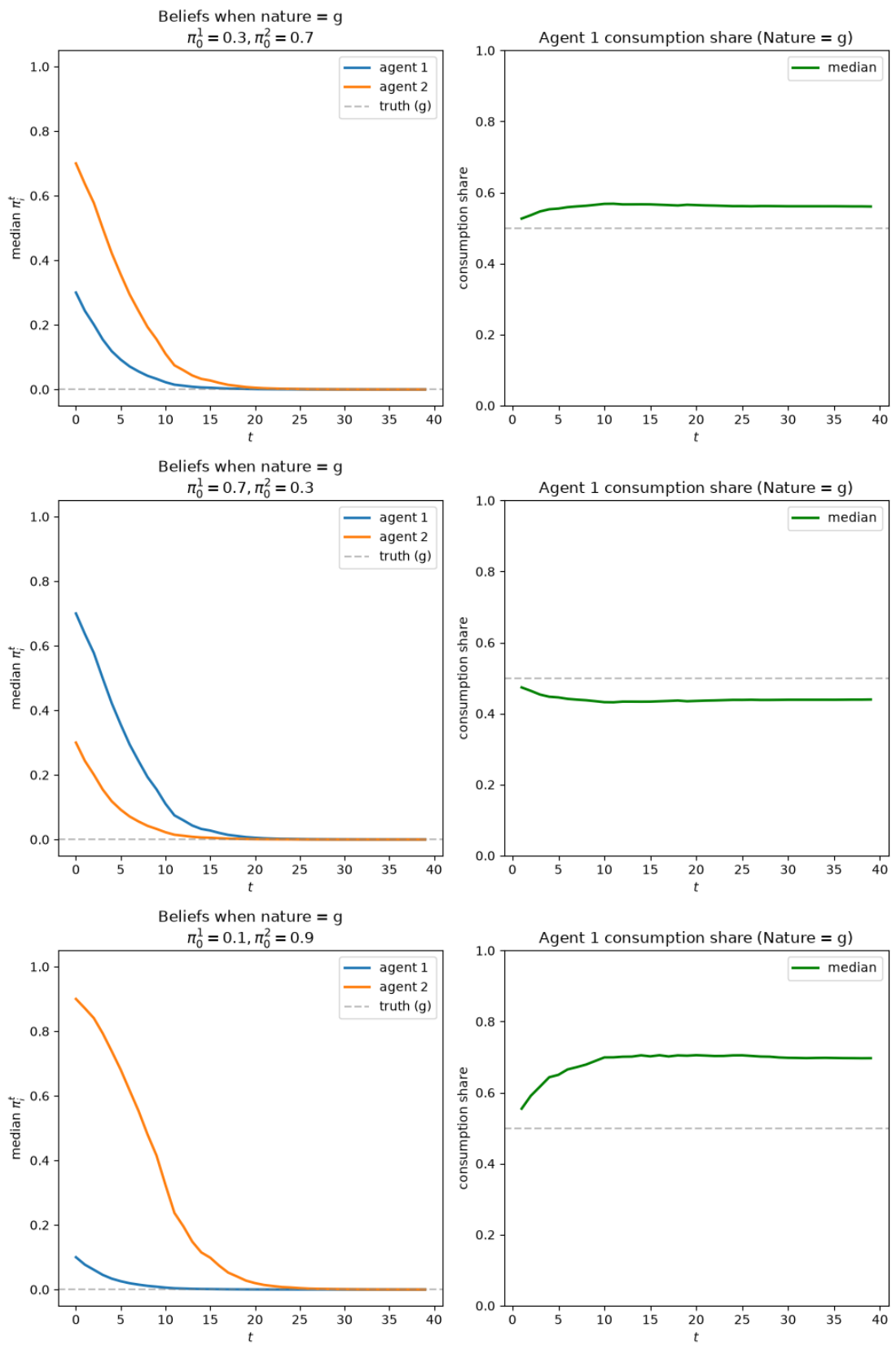
We can see that the agent with the more accurate belief gets higher consumption share.

Moreover, the further apart are initial beliefs, the longer it takes for the consumption ratio to converge.

The longer it takes for the "less accurate" agent to learn, the lower its ultimate consumption share.

Now let's plot outcomes when nature follows g:

```
fig_g, axes_g = plot_learning_results(results_g, π_0_scenarios, 'g', 0.0)
plt.show()
```


Beliefs when nature = g
$\pi_0^1 = 0.3, \pi_0^2 = 0.7$
agent 1
agent 2
truth (g)
median π_t^i
t
Agent 1 consumption share (Nature = g)
median
consumption share
t
Beliefs when nature = g
$\pi_0^1 = 0.7, \pi_0^2 = 0.3$
agent 1
agent 2
truth (g)
median π_t^i
t
Agent 1 consumption share (Nature = g)
median
consumption share
t
Beliefs when nature = g
$\pi_0^1 = 0.1, \pi_0^2 = 0.9$
agent 1
agent 2
truth (g)
median π_t^i
t
Agent 1 consumption share (Nature = g)
median
consumption share
t

We observe symmetrical outcomes.

Exercise 23.11.3

In the previous exercise, we purposefully set the two beta distributions to be relatively close to each other.

That made it challenging to distinguish the distributions.

Now let's study outcomes when the distributions are further apart.

Let's set $f \sim \text{Beta}(2, 5)$ and $g \sim \text{Beta}(5, 2)$.

Please use the Python code you have written to study outcomes.

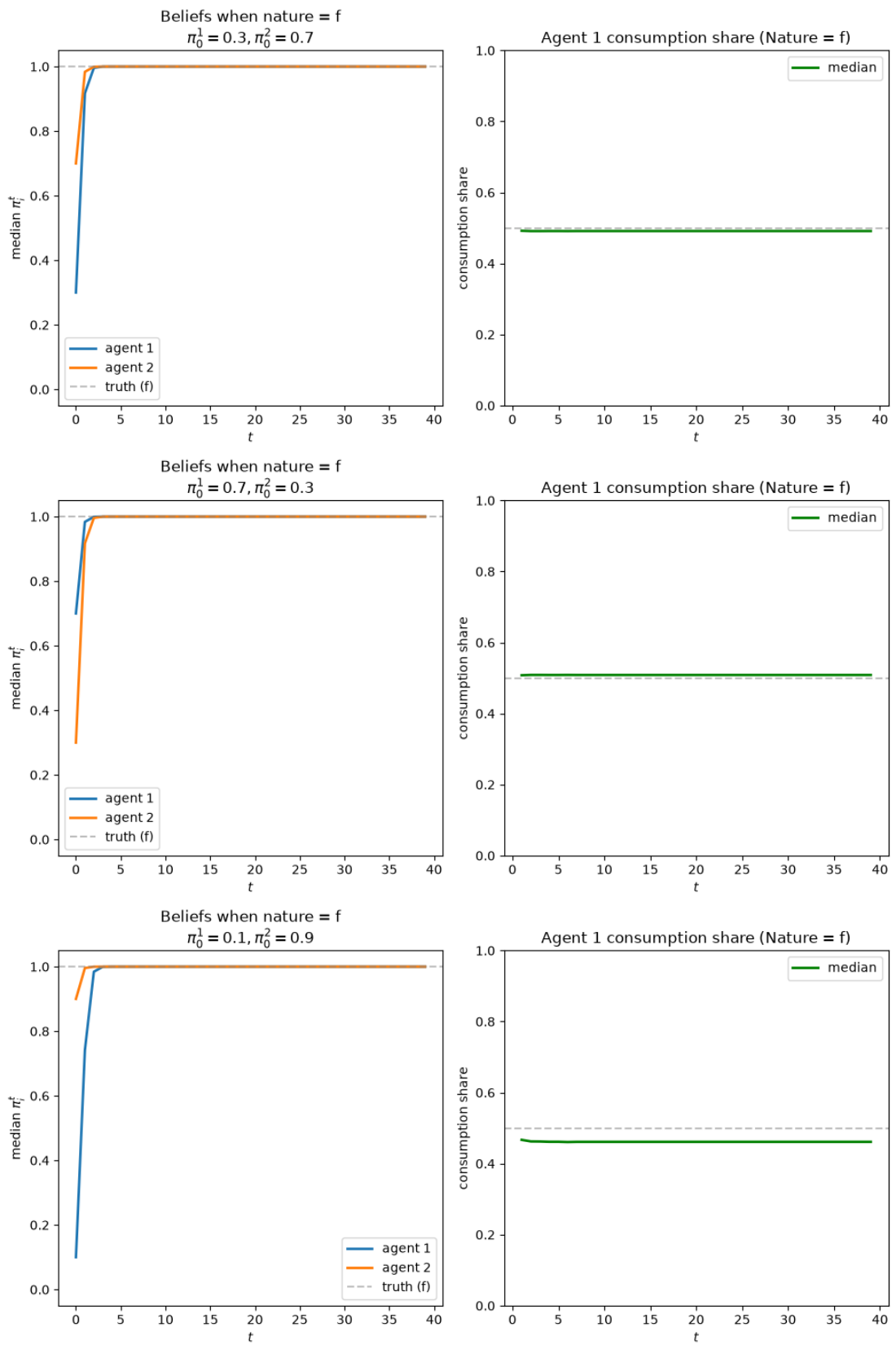
Solution

Here is one solution

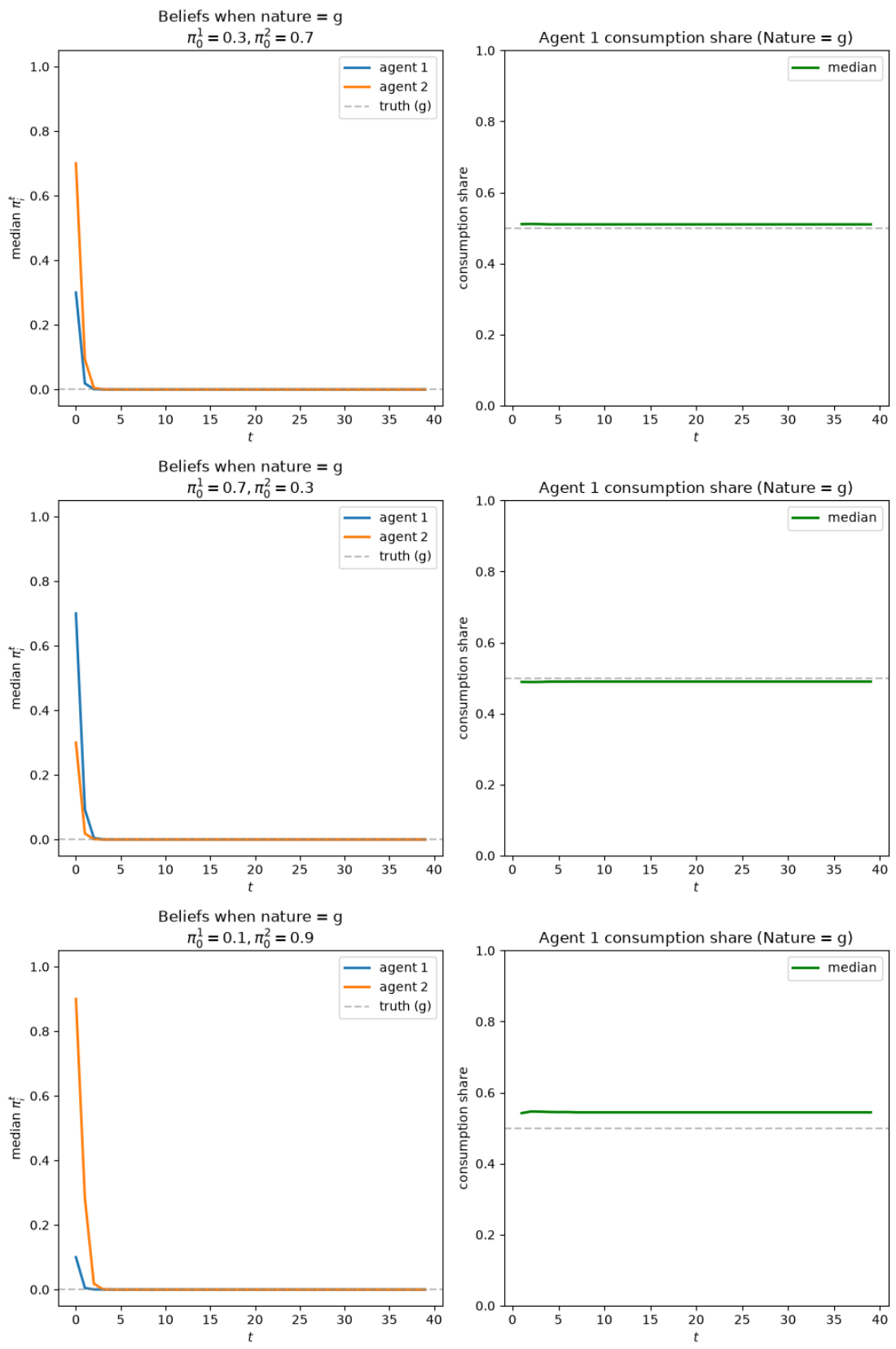
```
λ = 0.5
T = 40
N = 1000

F_a, F_b = 2, 5
G_a, G_b = 5, 2

f = jit(lambda x: p(x, F_a, F_b))
g = jit(lambda x: p(x, G_a, G_b))

π_0_scenarios = [
    (0.3, 0.7),
    (0.7, 0.3),
    (0.1, 0.9),
]

s_seq_f = np.random.beta(F_a, F_b, (N, T))
s_seq_g = np.random.beta(G_a, G_b, (N, T))

results_f = {}
results_g = {}

for i, (π_0_1, π_0_2) in enumerate(π_0_scenarios):
    # When nature follows f
    results_f[i] = simulate_learning_blume_easley(
            s_seq_f, f, g, π_0_1, π_0_2, λ)
    # When nature follows g
    results_g[i] = simulate_learning_blume_easley(
            s_seq_g, f, g, π_0_1, π_0_2, λ)
```

Now let's visualize the results

```
fig_f, axes_f = plot_learning_results(results_f, π_0_scenarios, 'f', 1.0)
plt.show()
```


Beliefs when nature = f
$\pi_0^1 = 0.3, \pi_0^2 = 0.7$
median π_t^i
agent 1
agent 2
truth (f)
t
Agent 1 consumption share (Nature = f)
median
consumption share
t
Beliefs when nature = f
$\pi_0^1 = 0.7, \pi_0^2 = 0.3$
median π_t^i
agent 1
agent 2
truth (f)
t
Agent 1 consumption share (Nature = f)
median
consumption share
t
Beliefs when nature = f
$\pi_0^1 = 0.1, \pi_0^2 = 0.9$
median π_t^i
agent 1
agent 2
truth (f)
t
Agent 1 consumption share (Nature = f)
median
consumption share
t

```
fig_g, axes_g = plot_learning_results(results_g, π_0_scenarios, 'g', 0.0)
plt.show()
```


Beliefs when nature = g
$\pi_0^1 = 0.3, \pi_0^2 = 0.7$
agent 1
agent 2
truth (g)
median π_t^i
t
Agent 1 consumption share (Nature = g)
median
consumption share
t
Beliefs when nature = g
$\pi_0^1 = 0.7, \pi_0^2 = 0.3$
agent 1
agent 2
truth (g)
median π_t^i
t
Agent 1 consumption share (Nature = g)
median
consumption share
t
Beliefs when nature = g
$\pi_0^1 = 0.1, \pi_0^2 = 0.9$
agent 1
agent 2
truth (g)
median π_t^i
t
Agent 1 consumption share (Nature = g)
median
consumption share
t

Evidently, because the two distributions are further apart, it is easier to distinguish them.

So learning occurs more quickly.

So do consumption shares.

Exercise 23.11.4

Two agents have different beliefs about three possible models.

Assume $f(x) \geq 0$, $g(x) \geq 0$, and $h(x) \geq 0$ for $x \in X$ with:

- $\int_X f(x)dx = 1$
- $\int_X g(x)dx = 1$
- $\int_X h(x)dx = 1$

We'll consider two agents:

- Agent 1: $\pi_0^g = 1 - \pi_0^f$, $\pi_0^f \in (0, 1)$, $\pi_0^h = 0$ (attaches positive probability only to models f and g)
- Agent 2: $\pi_0^g = \pi_0^f = 1/3$, $\pi_0^h = 1/3$ (attaches equal weights to all three models)

Let f and g be two beta distributions with $f \sim \text{Beta}(3, 2)$ and $g \sim \text{Beta}(2, 3)$, and set $h = \pi_0^f f + (1 - \pi_0^f)g$ with $\pi_0^f = 0.5$.

Bayes' Law tells us that posterior probabilities on models f and g evolve according to

$$\pi^f(s^t) := \frac{\pi_0^f f(s^t)}{\pi_0^f f(s^t) + \pi_0^g g(s^t) + (1 - \pi_0^f - \pi_0^g)h(s^t)}$$

and

$$\pi^g(s^t) := \frac{\pi_0^g g(s^t)}{\pi_0^f f(s^t) + \pi_0^g g(s^t) + (1 - \pi_0^f - \pi_0^g)h(s^t)}$$

Please simulate and visualize evolutions of posterior probabilities and consumption allocations when:

- Nature permanently draws from f
- Nature permanently draws from g

Solution

Let's implement this three-model case with two agents having different beliefs.

Let's define f and g far apart, with h being a mixture of f and g.

```
F_a, F_b = 3, 2
G_a, G_b = 2, 3
λ = 0.5
π_f_0 = 0.5

f = jit(lambda x: p(x, F_a, F_b))
g = jit(lambda x: p(x, G_a, G_b))
h = jit(lambda x: π_f_0 * f(x) + (1 - π_f_0) * g(x))
```

Now we can define the belief updating for the model

```
@jit(parallel=True)
def compute_posterior_three_models(
    s_seq, f_func, g_func, h_func, π_f_0, π_g_0):
    """
    Compute posterior probabilities for three models.
    """
    N, T = s_seq.shape
    π_h_0 = 1 - π_f_0 - π_g_0

    π_f = np.zeros((N, T))
    π_g = np.zeros((N, T))
    π_h = np.zeros((N, T))

    for n in prange(N):
        # Initialize with priors
        π_f[n, 0] = π_f_0
        π_g[n, 0] = π_g_0
        π_h[n, 0] = π_h_0

        # Compute cumulative likelihoods
        f_cumul = 1.0
        g_cumul = 1.0
        h_cumul = 1.0

        for t in range(1, T):
            s_t = s_seq[n, t]

            # Update cumulative likelihoods
            f_cumul *= f_func(s_t)
            g_cumul *= g_func(s_t)
            h_cumul *= h_func(s_t)

            # Compute posteriors using Bayes' rule
            denominator = π_f_0 * f_cumul + π_g_0 * g_cumul + π_h_0 * h_cumul

            π_f[n, t] = π_f_0 * f_cumul / denominator
            π_g[n, t] = π_g_0 * g_cumul / denominator
            π_h[n, t] = π_h_0 * h_cumul / denominator

    return π_f, π_g, π_h
```

Let's also write simulation code along the lines of earlier exercises

```
@jit
def bayesian_update_three_models(π_f_0, π_g_0, L_f, L_g, L_h):
    """Bayesian update for three models."""
    π_h_0 = 1 - π_f_0 - π_g_0
    denom = π_f_0 * L_f + π_g_0 * L_g + π_h_0 * L_h
    return π_f_0 * L_f / denom, π_g_0 * L_g / denom, π_h_0 * L_h / denom

@jit
def compute_mixture_density(π_f, π_g, π_h, f_val, g_val, h_val):
    """Compute mixture density for an agent."""
    return π_f * f_val + π_g * g_val + π_h * h_val

@jit(parallel=True)
def simulate_three_model_allocation(sequences, f_func, g_func, h_func,
                                    π_f_0_1, π_g_0_1, π_f_0_2, π_g_0_2, λ=0.5):
```

```
    """
    Simulate Blume-Easley model with learning agents and three models.
    """
    N, T = sequences.shape

    # Initialize arrays to store results
    beliefs_1 = {k: np.full((N, T), np.nan) for k in ['π_f', 'π_g', 'π_h']}
    beliefs_2 = {k: np.full((N, T), np.nan) for k in ['π_f', 'π_g', 'π_h']}
    c1_share = np.full((N, T), np.nan)
    l_agents_seq = np.full((N, T), np.nan)

    # Set initial beliefs
    beliefs_1['π_f'][:, 0] = π_f_0_1
    beliefs_1['π_g'][:, 0] = π_g_0_1
    beliefs_1['π_h'][:, 0] = 1 - π_f_0_1 - π_g_0_1
    beliefs_2['π_f'][:, 0] = π_f_0_2
    beliefs_2['π_g'][:, 0] = π_g_0_2
    beliefs_2['π_h'][:, 0] = 1 - π_f_0_2 - π_g_0_2

    for n in range(N):
        # Initialize cumulative likelihoods
        L_cumul = {'f': 1.0, 'g': 1.0, 'h': 1.0}
        l_agents_cumul = 1.0

        # Calculate initial consumption share at t=0
        l_agents_seq[n, 0] = 1.0
        c1_share[n, 0] = λ * 1.0 / (1 - λ + λ * 1.0)  # This equals λ

        for t in range(1, T):
            s_t = sequences[n, t]

            # Compute densities for current observation
            densities = {
                'f': f_func(s_t),
                'g': g_func(s_t),
                'h': h_func(s_t)
            }

            # Update cumulative likelihoods
            for model in L_cumul:
                L_cumul[model] *= densities[model]

            # Bayesian updates for both agents
            π_f_1, π_g_1, π_h_1 = bayesian_update_three_models(
                π_f_0_1, π_g_0_1, L_cumul['f'], L_cumul['g'], L_cumul['h'])
            π_f_2, π_g_2, π_h_2 = bayesian_update_three_models(
                π_f_0_2, π_g_0_2, L_cumul['f'], L_cumul['g'], L_cumul['h'])

            # Store beliefs
            beliefs_1['π_f'][n, t] = π_f_1
            beliefs_1['π_g'][n, t] = π_g_1
            beliefs_1['π_h'][n, t] = π_h_1
            beliefs_2['π_f'][n, t] = π_f_2
            beliefs_2['π_g'][n, t] = π_g_2
            beliefs_2['π_h'][n, t] = π_h_2

            # Compute mixture densities
```

```
            m1_t = compute_mixture_density(
                π_f_1, π_g_1, π_h_1, densities['f'],
                densities['g'], densities['h'])
            m2_t = compute_mixture_density(
                π_f_2, π_g_2, π_h_2, densities['f'],
                densities['g'], densities['h'])

            # Update cumulative likelihood ratio between agents
            l_agents_cumul *= (m1_t / m2_t)
            l_agents_seq[n, t] = l_agents_cumul

            # Consumption share for agent 1
            c1_share[n, t] = λ * l_agents_cumul / (1 - λ + λ * l_agents_cumul)

    return {
        'π_f_1': beliefs_1['π_f'],
        'π_g_1': beliefs_1['π_g'],
        'π_h_1': beliefs_1['π_h'],
        'π_f_2': beliefs_2['π_f'],
        'π_g_2': beliefs_2['π_g'],
        'π_h_2': beliefs_2['π_h'],
        'c1_share': c1_share,
        'l_agents': l_agents_seq
    }
```

The following code cell defines a plotting function to show evolutions of beliefs and consumption ratios

Now let's run the simulation.

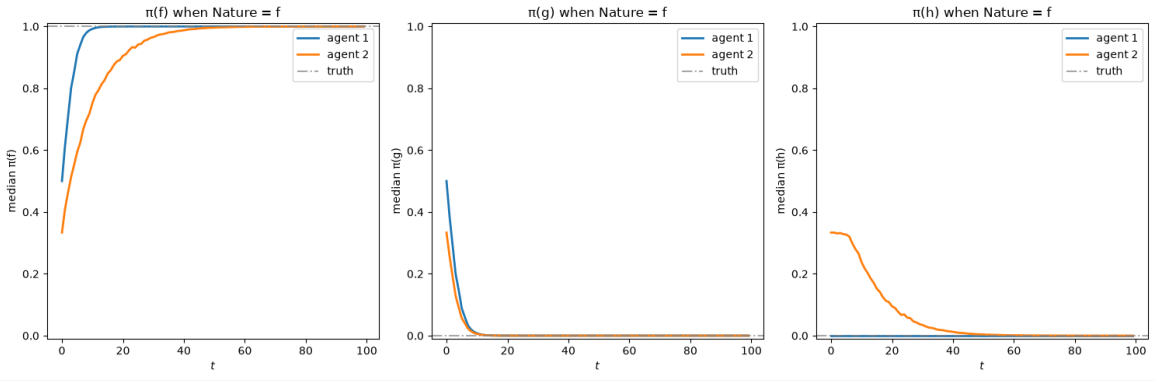
In the simulation below, agent 1 assigns positive probabilities only to f and g, while agent 2 puts equal weights on all three models.

```
T = 100
N = 1000

# Generate sequences for nature f and g
s_seq_f = np.random.beta(F_a, F_b, (N, T))
s_seq_g = np.random.beta(G_a, G_b, (N, T))

# Run simulations
results_f = simulate_three_model_allocation(s_seq_f,
                        f, g, h, π_f_0, 1-π_f_0,
                        1/3, 1/3, λ)
results_g = simulate_three_model_allocation(s_seq_g,
                        f, g, h, π_f_0, 1-π_f_0,
                        1/3, 1/3, λ)
```

Plots below show the evolution of beliefs for each model (f, g, h) separately.

First we show the figure when nature chooses f

```
plot_belief_evolution(results_f, nature='f', figsize=(15, 5))
plt.show()
```



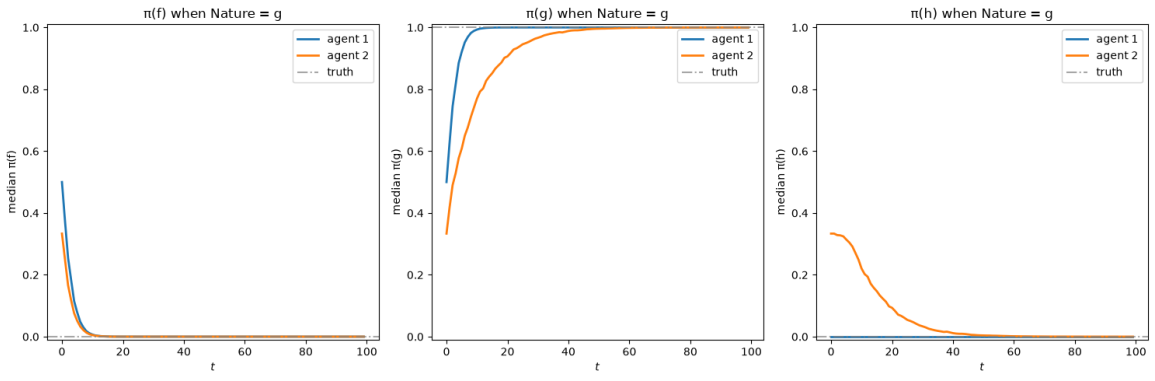
Agent 1's posterior beliefs are depicted in blue and agent 2's posterior beliefs are depicted in orange.

Evidently, when nature draws from f, agent 1 learns faster than agent 2, who, unlike agent 1, attaches a positive prior probability to model h:

- In the leftmost panel, both agents' beliefs for $\pi(f)$ converge toward 1 (the truth)
- Agent 2's belief in model h (rightmost panel) gradually converges to 0 after an initial rise

Now let's plot the belief evolution when nature chooses g:

```
plot_belief_evolution(results_g, nature='g', figsize=(15, 5))
plt.show()
```

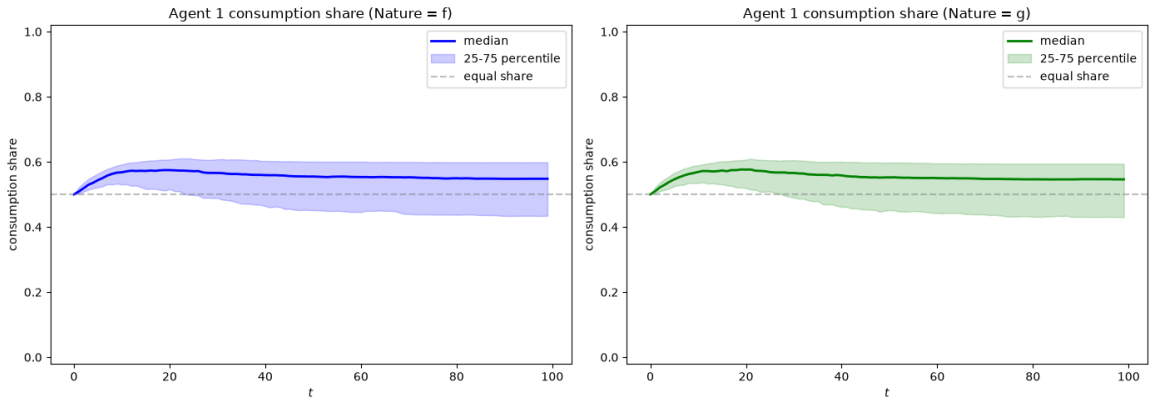


Again, agent 1 learns faster than agent 2.

Before reading the next figure, please guess how consumption shares evolve.

Remember that agent 1 reaches the correct model faster than agent 2

```
plot_consumption_dynamics(results_f, results_g, λ=0.5, figsize=(14, 5))
plt.show()
```



As we expected, agent 1 has a higher consumption share compared to agent 2.

In this exercise, the "truth" is among possible outcomes according to both agents.

Agent 2's model is "more general" because it allows a possibility – that nature is drawing from h – that agent 1's model does not include.

Agent 1 learns more quickly because he uses a simpler model.

Exercise 23.11.5

Now consider two agents with extreme priors about three models.

Consider the same setup as the previous exercise, but now:

- Agent 1: $\pi_0^g = \pi_0^f = \frac{\epsilon}{2} > 0$, where ϵ is close to 0 (e.g., $\epsilon = 0.01$)
- Agent 2: $\pi_0^g = \pi_0^f = 0$ (rigid belief in model h)

Choose h to be close but not equal to either f or g as measured by KL divergence. For example, set $h \sim \text{Beta}(1.2, 1.1)$ and $f \sim \text{Beta}(1, 1)$.

Please simulate and visualize evolutions of posterior probabilities and consumption allocations when:

- Nature permanently draws from f
- Nature permanently draws from g

Solution

To explore this exercise, we increase T to 1000.

Let's specify f, g, and h and verify that h and f are closer than h and g

```
F_a, F_b = 1, 1
G_a, G_b = 3, 1.2
H_a, H_b = 1.2, 1.1

f = jit(lambda x: p(x, F_a, F_b))
g = jit(lambda x: p(x, G_a, G_b))
h = jit(lambda x: p(x, H_a, H_b))

Kh_f = compute_KL(h, f)
```

```
Kh_g = compute_KL(h, g)
Kf_h = compute_KL(f, h)
Kg_h = compute_KL(g, h)

print(f"KL divergences:")
print(f"KL(h,f) = {Kh_f:.4f}, KL(h,g) = {Kh_g:.4f}")
print(f"KL(f,h) = {Kf_h:.4f}, KL(g,h) = {Kg_h:.4f}")
```

```
KL divergences:
KL(h,f) = 0.0092, KL(h,g) = 0.5514
KL(f,h) = 0.0105, KL(g,h) = 0.2919
```

Now we can set the belief models for the two agents

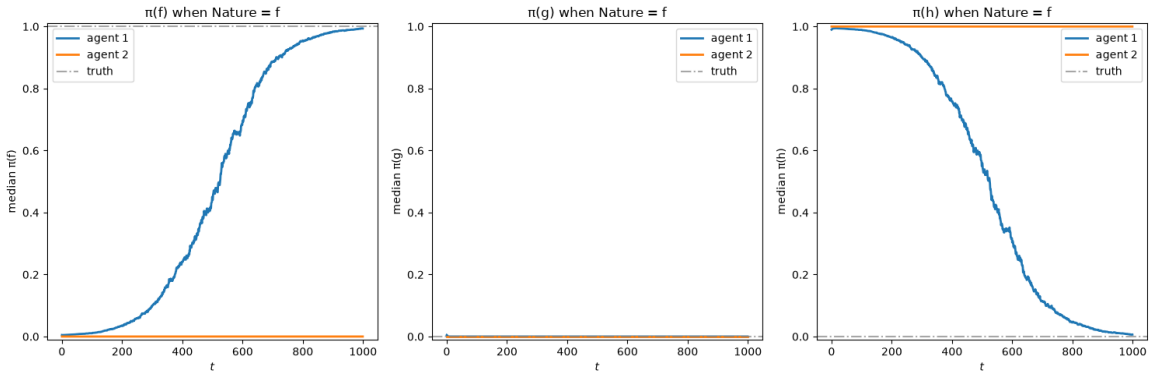
```
ε = 0.01
λ = 0.5

# Agent 1: π_f = ε/2, π_g = ε/2, π_h = 1-ε
# (almost rigid about h)
π_f_1 = ε/2
π_g_1 = ε/2

# Agent 2: π_f = 0, π_g = 0, π_h = 1
# (fully rigid about h)
π_f_2 = 1e-10
π_g_2 = 1e-10
```

Now we can run the simulation

```
T = 1000
N = 1000

# Generate sequences for different nature scenarios
s_seq_f = np.random.beta(F_a, F_b, (N, T))
s_seq_g = np.random.beta(G_a, G_b, (N, T))

# Run simulations for both scenarios
results_f = simulate_three_model_allocation(
                                s_seq_f,
                                f, g, h,
                                π_f_1, π_g_1, π_f_2, π_g_2, λ)
results_g = simulate_three_model_allocation(
                                s_seq_g,
                                f, g, h,
                                π_f_1, π_g_1, π_f_2, π_g_2, λ)
```

Let's plot the belief evolution when nature chooses f

```
plot_belief_evolution(results_f, nature='f', figsize=(15, 5))
plt.show()
```



Observe how slowly agent 1 learns the truth in the leftmost panel showing $\pi(f)$.

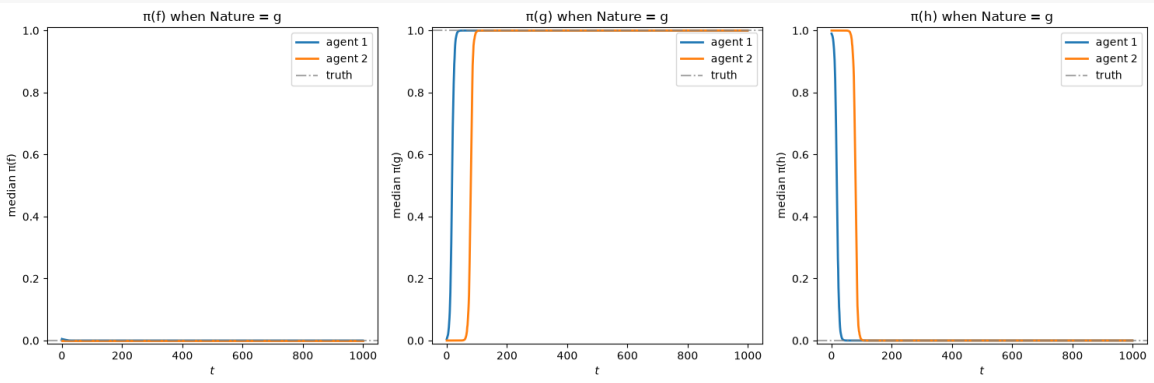
Also note that agent 2 is not updating.

This is because we have specified that f is very difficult to distinguish from h as measured by $KL(f, h)$.

The rigidity regarding h prevents agent 2 from updating its beliefs when observing a very similar model f

Now let's plot the belief evolution when nature chooses g

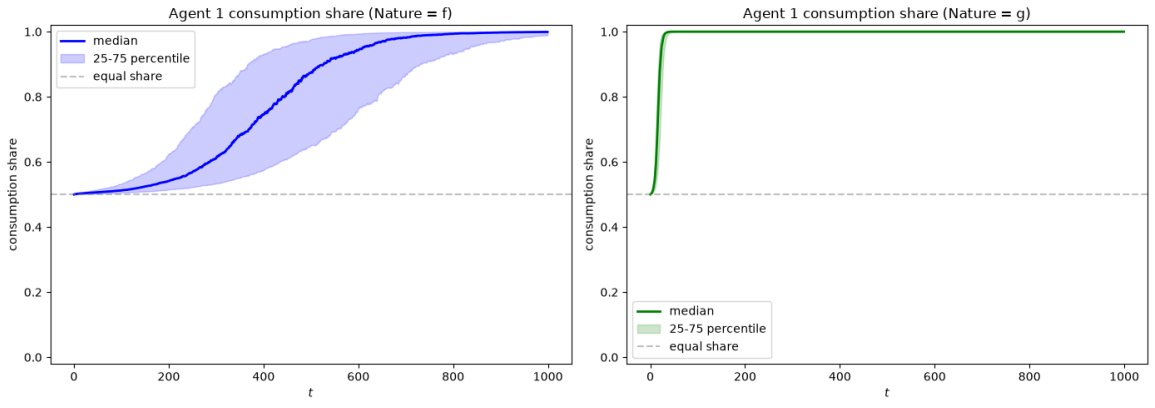
```
plot_belief_evolution(results_g, nature='g', figsize=(15, 5))
plt.show()
```



When nature draws from g, it is further away from h as measured by the KL divergence.

This helps both agents learn the truth more quickly.

```
plot_consumption_dynamics(results_f, results_g,
                          λ=0.5, figsize=(14, 5))
plt.show()
```



In the consumption dynamics plot, notice that agent 1's consumption share converges to 1 both when nature permanently draws from f and when nature permanently draws from g.

CHAPTER

TWENTYFOUR

LIKELIHOOD PROCESSES FOR VAR MODELS

Contents

- *Likelihood Processes For VAR Models*
  - *Overview*
  - *VAR model setup*
  - *Likelihood ratio process*
  - *Example 1: two AR(1) processes*
  - *Example 2: bivariate VAR models*
  - *Application: Samuelson multiplier-accelerator*

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install quantecon
```

24.1 Overview

This lecture extends our analysis of likelihood ratio processes to Vector Autoregressions (VARs).

We'll

- Construct likelihood functions for VAR models
- Form likelihood ratio processes for comparing two VAR models
- Visualize the evolution of likelihood ratios over time
- Connect VAR likelihood ratios to the Samuelson multiplier-accelerator model

Our analysis builds on concepts from:

- *Likelihood Ratio Processes*
- *Linear State Space Models*
- *Samuelson Multiplier-Accelerator*

Let's start by importing helpful libraries:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import linalg
from scipy.stats import multivariate_normal as mvn
from quantecon import LinearStateSpace
import quantecon as qe
from numba import jit
from typing import NamedTuple, Optional, Tuple
from collections import namedtuple
```

24.2 VAR model setup

Consider a VAR model of the form:

$$x_{t+1} = Ax_t + Cw_{t+1}$$
$$x_0 \sim \mathcal{N}(\mu_0, \Sigma_0)$$

where:

- x_t is an $n \times 1$ state vector
- $w_{t+1} \sim \mathcal{N}(0, I)$ is an $m \times 1$ vector of shocks
- A is an $n \times n$ transition matrix
- C is an $n \times m$ volatility matrix

Let's define the necessary data structures for the VAR model

```
VARModel = namedtuple('VARModel', ['A', 'C', 'μ_0', 'Σ_0',
                                   'CC', 'CC_inv', 'log_det_CC',
                                   'Σ_0_inv', 'log_det_Σ_0'])
def compute_stationary_var(A, C):
    """
    Compute stationary mean and covariance for VAR model
    """
    n = A.shape[0]

    # Check stability
    eigenvalues = np.linalg.eigvals(A)
    if np.max(np.abs(eigenvalues)) >= 1:
        raise ValueError("VAR is not stationary")

    μ_0 = np.zeros(n)

    # Stationary covariance: solve discrete Lyapunov equation
    # Σ_0 = A @ Σ_0 @ A.T + C @ C.T
    CC = C @ C.T
    Σ_0 = linalg.solve_discrete_lyapunov(A, CC)

    return μ_0, Σ_0

def create_var_model(A, C, μ_0=None, Σ_0=None, stationary=True):
    """
    Create a VAR model with parameters and precomputed matrices
    """
    A = np.asarray(A)
```

(continues on next page)

(continued from previous page)

```
    C = np.asarray(C)
    n = A.shape[0]
    CC = C @ C.T

    if stationary:
        μ_0_comp, Σ_0_comp = compute_stationary_var(A, C)
    else:
        μ_0_comp = μ_0 if μ_0 is not None else np.zeros(n)
        Σ_0_comp = Σ_0 if Σ_0 is not None else np.eye(n)

    # Check if CC is singular
    det_CC = np.linalg.det(CC)
    if np.abs(det_CC) < 1e-10:
        # Use pseudo-inverse for singular case
        CC_inv = np.linalg.pinv(CC)
        CC_reg = CC + 1e-10 * np.eye(CC.shape[0])
        log_det_CC = np.log(np.linalg.det(CC_reg))
    else:
        CC_inv = np.linalg.inv(CC)
        log_det_CC = np.log(det_CC)

    # Same check for Σ_0
    det_Σ_0 = np.linalg.det(Σ_0_comp)
    if np.abs(det_Σ_0) < 1e-10:
        Σ_0_inv = np.linalg.pinv(Σ_0_comp)
        Σ_0_reg = Σ_0_comp + 1e-10 * np.eye(Σ_0_comp.shape[0])
        log_det_Σ_0 = np.log(np.linalg.det(Σ_0_reg))
    else:
        Σ_0_inv = np.linalg.inv(Σ_0_comp)
        log_det_Σ_0 = np.log(det_Σ_0)

    return VARModel(A=A, C=C, μ_0=μ_0_comp, Σ_0=Σ_0_comp,
                    CC=CC, CC_inv=CC_inv, log_det_CC=log_det_CC,
                    Σ_0_inv=Σ_0_inv, log_det_Σ_0=log_det_Σ_0)
```

24.2.1 Joint distribution

The joint probability distribution $f(x_T, x_{T-1}, \dots, x_0)$ can be factored as:

$$f(x_T, \dots, x_0) = f(x_T|x_{T-1}) f(x_{T-1}|x_{T-2}) \cdots f(x_1|x_0) f(x_0)$$

Since the VAR is Markovian, $f(x_{t+1}|x_t, \dots, x_0) = f(x_{t+1}|x_t)$.

24.2.2 Conditional densities

Given the Gaussian structure, the conditional distribution $f(x_{t+1}|x_t)$ is Gaussian with:

- Mean: Ax_t
- Covariance: CC'

The log conditional density is

$$\log f(x_{t+1}|x_t) = -\frac{n}{2}\log(2\pi) - \frac{1}{2}\log\det(CC') - \frac{1}{2}(x_{t+1} - Ax_t)'(CC')^{-1}(x_{t+1} - Ax_t) \tag{24.1}$$

```
def log_likelihood_transition(x_next, x_curr, model):
    """
    Compute log likelihood of transition from x_curr to x_next
    """
    x_next = np.atleast_1d(x_next)
    x_curr = np.atleast_1d(x_curr)
    n = len(x_next)
    diff = x_next - model.A @ x_curr
    return -0.5 * (n * np.log(2 * np.pi) + model.log_det_CC +
                   diff @ model.CC_inv @ diff)
```

The log density of the initial state is:

$$\log f(x_0) = -\frac{n}{2}\log(2\pi) - \frac{1}{2}\log\det(\Sigma_0) - \frac{1}{2}(x_0 - \mu_0)'\Sigma_0^{-1}(x_0 - \mu_0)$$

```
def log_likelihood_initial(x_0, model):
    """
    Compute log likelihood of initial state
    """
    x_0 = np.atleast_1d(x_0)
    n = len(x_0)
    diff = x_0 - model.μ_0
    return -0.5 * (n * np.log(2 * np.pi) + model.log_det_Σ_0 +
                   diff @ model.Σ_0_inv @ diff)
```

Now let's group the likelihood computations into a single function that computes the log likelihood of an entire path

```
def log_likelihood_path(X, model):
    """
    Compute log likelihood of entire path
    """

    T = X.shape[0] - 1
    log_L = log_likelihood_initial(X[0], model)

    for t in range(T):
        log_L += log_likelihood_transition(X[t+1], X[t], model)

    return log_L

def simulate_var(model, T, N_paths=1):
    """
    Simulate paths from the VAR model
    """
    n = model.A.shape[0]
    m = model.C.shape[1]
    paths = np.zeros((N_paths, T+1, n))

    for i in range(N_paths):
        # Draw initial state
        x = mvn.rvs(mean=model.μ_0, cov=model.Σ_0)
        x = np.atleast_1d(x)
        paths[i, 0] = x

        # Simulate forward
        for t in range(T):
            w = np.random.randn(m)
```

(continues on next page)

(continued from previous page)

```
            x = model.A @ x + model.C @ w
            paths[i, t+1] = x

    return paths if N_paths > 1 else paths[0]
```

24.3 Likelihood ratio process

Now let's compute likelihood ratio processes for comparing two VAR models.

For a VAR model with state vector x_t, the log likelihood ratio at time t is

$$\ell_t = \log \frac{p_f(x_t|x_{t-1})}{p_g(x_t|x_{t-1})}$$

where p_f and p_g are the conditional densities under models f and g respectively.

The cumulative log likelihood ratio process is

$$L_t = \sum_{s=1}^{t} \ell_s = \sum_{s=1}^{t} \log \frac{p_f(x_s|x_{s-1})}{p_g(x_s|x_{s-1})}$$

where $p_f(x_t|x_{t-1})$ and $p_g(x_t|x_{t-1})$ are given by their respective conditional densities defined in (24.1).

Let's write those equations in Python

```
def compute_likelihood_ratio_var(paths, model_f, model_g):
    """
    Compute likelihood ratio process for VAR models
    """
    if paths.ndim == 2:
        paths = paths[np.newaxis, :]

    N_paths, T_plus_1, n = paths.shape
    T = T_plus_1 - 1
    log_L_ratios = np.zeros((N_paths, T+1))

    for i in range(N_paths):
        X = paths[i]

        # Initial log likelihood ratio
        log_L_f_0 = log_likelihood_initial(X[0], model_f)
        log_L_g_0 = log_likelihood_initial(X[0], model_g)
        log_L_ratios[i, 0] = log_L_f_0 - log_L_g_0

        # Recursive computation
        for t in range(1, T+1):
            log_L_f_t = log_likelihood_transition(X[t], X[t-1], model_f)
            log_L_g_t = log_likelihood_transition(X[t], X[t-1], model_g)

            # Update log likelihood ratio
            log_diff = log_L_f_t - log_L_g_t

            log_L_prev = log_L_ratios[i, t-1]
            log_L_new = log_L_prev + log_diff
            log_L_ratios[i, t] = log_L_new
```

(continues on next page)

(continued from previous page)

```
    return log_L_ratios if N_paths > 1 else log_L_ratios[0]
```

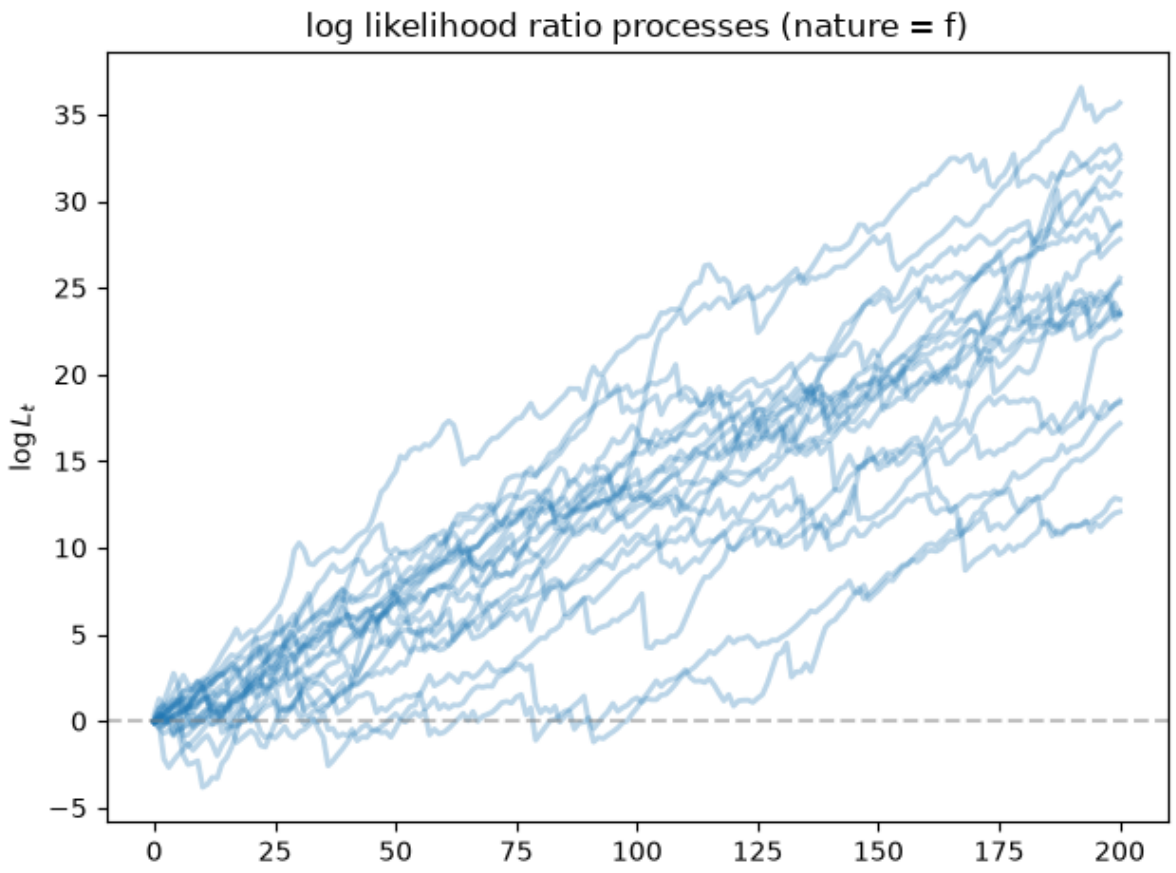
24.4 Example 1: two AR(1) processes

Let's start with a simple example comparing two univariate AR(1) processes with $A_f = 0.8$, $A_g = 0.5$, and $C_f = 0.3$, $C_g = 0.4$

```
# Model f: AR(1) with persistence ρ = 0.8
A_f = np.array([[0.8]])
C_f = np.array([[0.3]])

# Model g: AR(1) with persistence ρ = 0.5
A_g = np.array([[0.5]])
C_g = np.array([[0.4]])

# Create VAR models
model_f = create_var_model(A_f, C_f)
model_g = create_var_model(A_g, C_g)
```

Let's generate 100 paths of length 200 from model f and compute the likelihood ratio processes

```
# Simulate from model f
T = 200
N_paths = 100
paths_from_f = simulate_var(model_f, T, N_paths)

L_ratios_f = compute_likelihood_ratio_var(paths_from_f, model_f, model_g)

fig, ax = plt.subplots()

for i in range(min(20, N_paths)):
    ax.plot(L_ratios_f[i], alpha=0.3, color='C0', lw=2)

ax.axhline(y=0, color='gray', linestyle='--', alpha=0.5)
ax.set_ylabel(r'$\log L_t$')
ax.set_title('log likelihood ratio processes (nature = f)')

plt.tight_layout()
plt.show()
```



As we expected, the likelihood ratio processes goes to $+\infty$ as T increases, indicating that model f is chosen correctly by our algorithm.

24.5 Example 2: bivariate VAR models

Now let's consider an example with bivariate VAR models with

$$A_f = \begin{bmatrix} 0.7 & 0.2 \\ 0.1 & 0.6 \end{bmatrix}, \quad C_f = \begin{bmatrix} 0.3 & 0.1 \\ 0.1 & 0.3 \end{bmatrix}$$

and

$$A_g = \begin{bmatrix} 0.5 & 0.3 \\ 0.2 & 0.5 \end{bmatrix}, \quad C_g = \begin{bmatrix} 0.4 & 0.0 \\ 0.0 & 0.4 \end{bmatrix}$$

```
A_f = np.array([[0.7, 0.2],
                [0.1, 0.6]])

C_f = np.array([[0.3, 0.1],
                [0.1, 0.3]])

A_g = np.array([[0.5, 0.3],
                [0.2, 0.5]])

C_g = np.array([[0.4, 0.0],
```

(continues on next page)

(continued from previous page)

```
                [0.0, 0.4]])

# Create VAR models
model2_f = create_var_model(A_f, C_f)
model2_g = create_var_model(A_g, C_g)

# Check stationarity
print("model f eigenvalues:", np.linalg.eigvals(A_f))
print("model g eigenvalues:", np.linalg.eigvals(A_g))
```

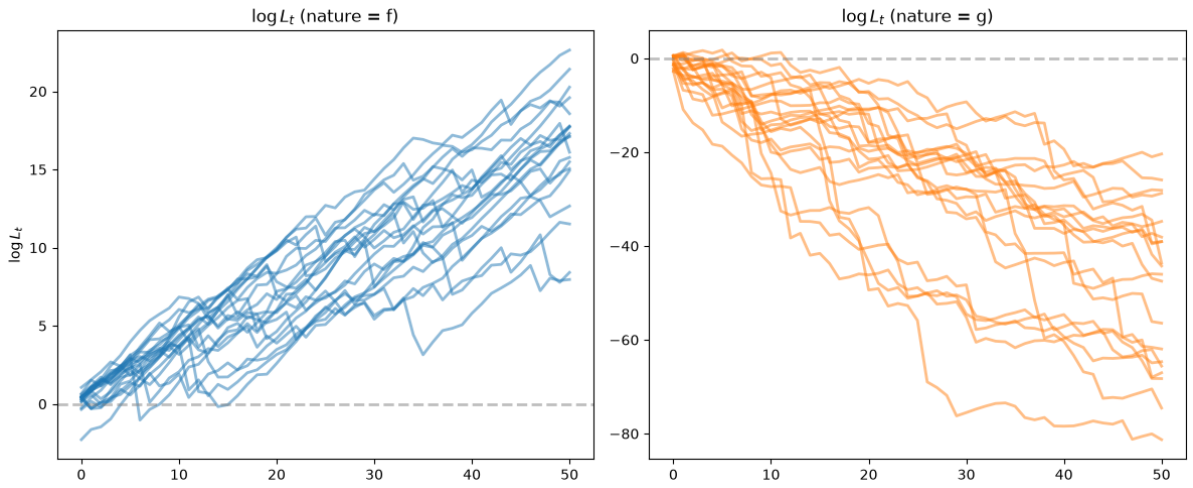
```
model f eigenvalues: [0.8 0.5]
model g eigenvalues: [0.74494897 0.25505103]
```

Let's generate 50 paths of length 50 from both models and compute the likelihood ratio processes

```
# Simulate from both models
T = 50
N_paths = 50

paths_from_f = simulate_var(model2_f, T, N_paths)
paths_from_g = simulate_var(model2_g, T, N_paths)

# Compute likelihood ratios
L_ratios_ff = compute_likelihood_ratio_var(paths_from_f, model2_f, model2_g)
L_ratios_gf = compute_likelihood_ratio_var(paths_from_g, model2_f, model2_g)
```

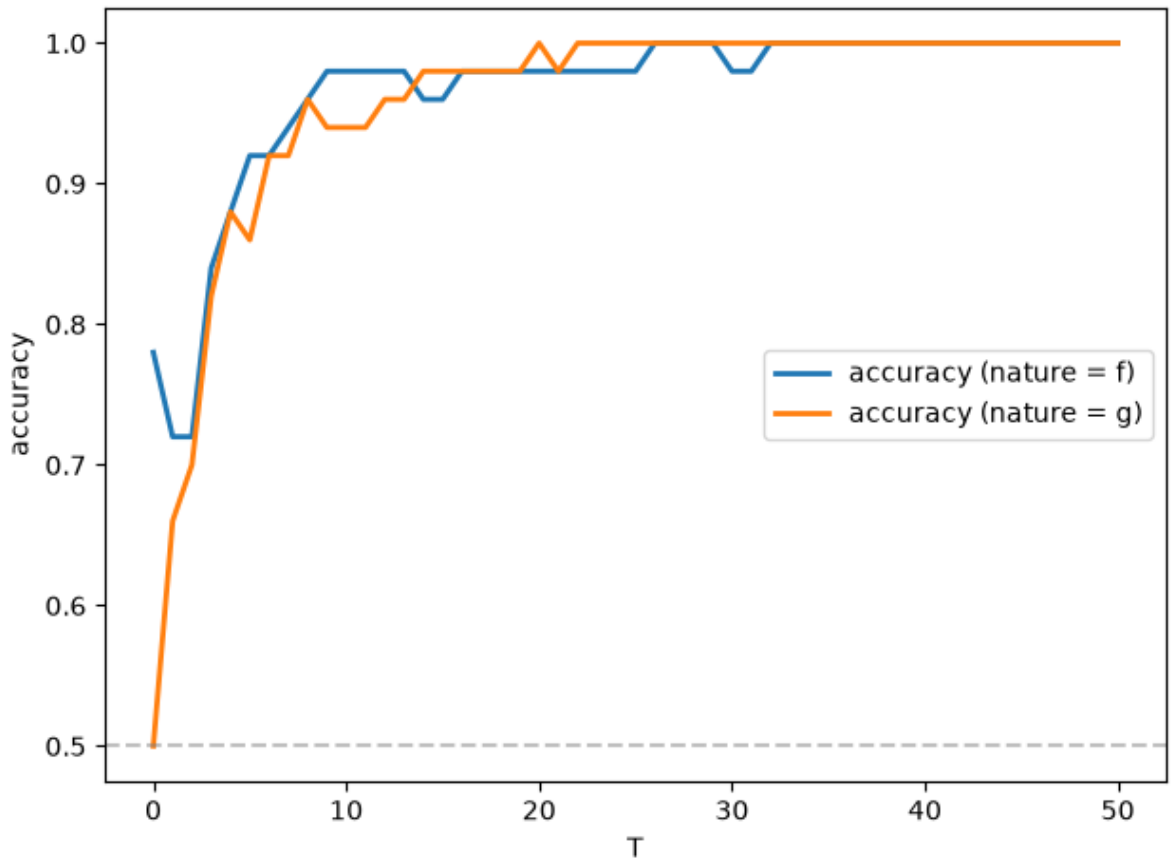
We can see that for paths generated from model f, the likelihood ratio processes tend to go to $+\infty$, while for paths from model g, they tend to go to $-\infty$.

```
# Visualize the results
fig, axes = plt.subplots(1, 2, figsize=(12, 5))

ax = axes[0]
for i in range(min(20, N_paths)):
    ax.plot(L_ratios_ff[i], alpha=0.5, color='C0', lw=2)
ax.axhline(y=0, color='gray', linestyle='--', alpha=0.5, lw=2)
ax.set_title(r'$\log L_t$ (nature = f)')
ax.set_ylabel(r'$\log L_t$')

ax = axes[1]
for i in range(min(20, N_paths)):
    ax.plot(L_ratios_gf[i], alpha=0.5, color='C1', lw=2)
ax.axhline(y=0, color='gray', linestyle='--', alpha=0.5, lw=2)
ax.set_title(r'$\log L_t$ (nature = g)')
plt.tight_layout()
plt.show()
```



Let's apply a Neyman-Pearson frequentist decision rule described in *Likelihood Ratio Processes* that selects model f when $\log L_T \geq 0$ and model g when $\log L_T < 0$

```
fig, ax = plt.subplots()
T_values = np.arange(0, T+1)
accuracy_f = np.zeros(len(T_values))
accuracy_g = np.zeros(len(T_values))

for i, t in enumerate(T_values):
    # Correct selection when data from f
    accuracy_f[i] = np.mean(L_ratios_ff[:, t] > 0)
    # Correct selection when data from g
    accuracy_g[i] = np.mean(L_ratios_gf[:, t] < 0)

ax.plot(T_values, accuracy_f, 'C0', linewidth=2, label='accuracy (nature = f)')
ax.plot(T_values, accuracy_g, 'C1', linewidth=2, label='accuracy (nature = g)')
ax.axhline(y=0.5, color='gray', linestyle='--', alpha=0.5)
ax.set_xlabel('T')
ax.set_ylabel('accuracy')
ax.legend()

plt.tight_layout()
plt.show()
```



Evidently, the accuracy approaches 1 as T increases, and it does so very quickly.

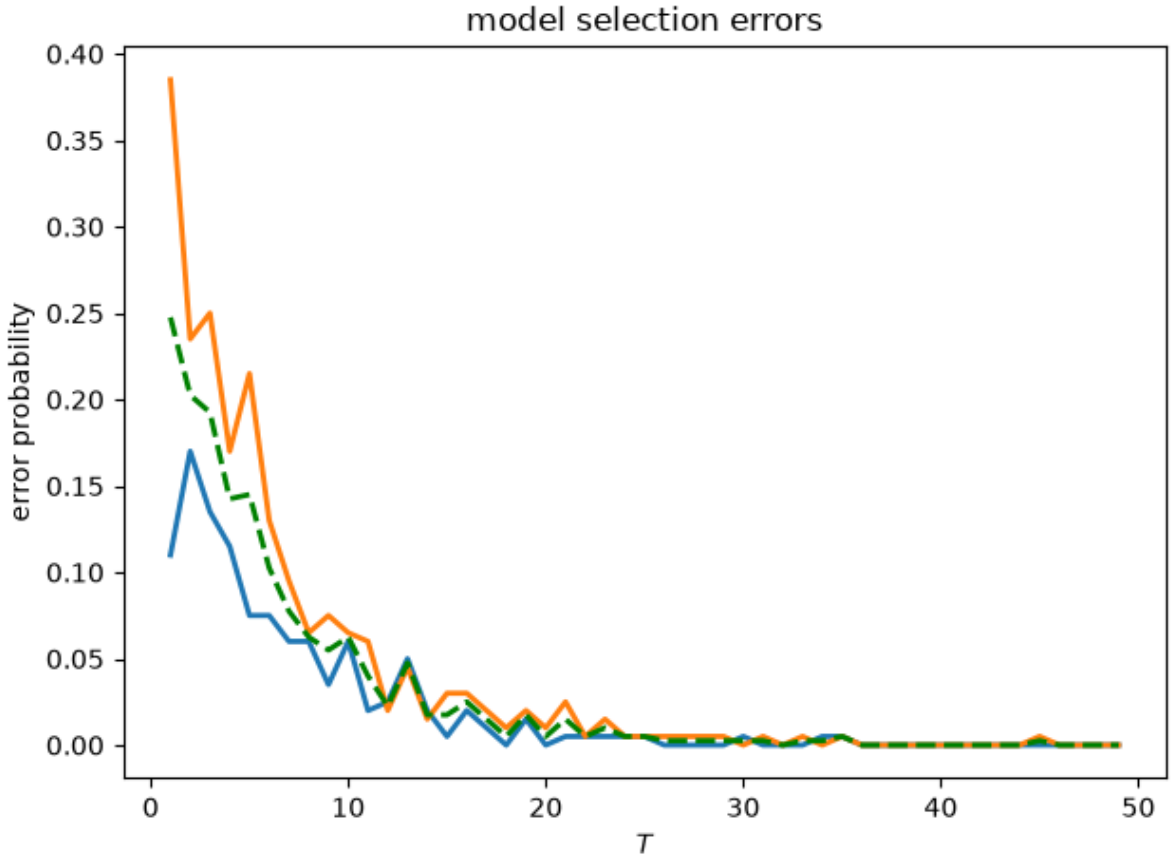
Let's also check the type I and type II errors as functions of T

```
def model_selection_analysis(T_values, model_f, model_g, N_sim=500):
    """
    Analyze model selection performance for different sample sizes
    """
    errors_f = []  # Type I errors
    errors_g = []  # Type II errors

    for T in T_values:
        # Simulate from model f
        paths_f = simulate_var(model_f, T, N_sim//2)
        L_ratios_f = compute_likelihood_ratio_var(paths_f, model_f, model_g)

        # Simulate from model g
        paths_g = simulate_var(model_g, T, N_sim//2)
        L_ratios_g = compute_likelihood_ratio_var(paths_g, model_f, model_g)

        # Decision rule: choose f if log L_T >= 0
        errors_f.append(np.mean(L_ratios_f[:, -1] < 0))
        errors_g.append(np.mean(L_ratios_g[:, -1] >= 0))

    return np.array(errors_f), np.array(errors_g)

T_values = np.arange(1, 50, 1)
```

(continues on next page)

(continued from previous page)

```
errors_f, errors_g = model_selection_analysis(T_values, model2_f, model2_g, N_sim=400)

fig, ax = plt.subplots()

ax.plot(T_values, errors_f, 'C0', linewidth=2, label='type I error')
ax.plot(T_values, errors_g, 'C1', linewidth=2, label='type II error')
ax.plot(T_values, 0.5 * (errors_f + errors_g), 'g--',
linewidth=2, label='average error')
ax.set_xlabel('$T$')
ax.set_ylabel('error probability')
ax.set_title('model selection errors')
plt.tight_layout()
plt.show()
```



24.6 Application: Samuelson multiplier-accelerator

Now let's connect to the Samuelson multiplier-accelerator model.

The model consists of:

- Consumption: $C_t = \gamma + aY_{t-1}$ where $a \in (0, 1)$ is the marginal propensity to consume
- Investment: $I_t = b(Y_{t-1} - Y_{t-2})$ where $b > 0$ is the accelerator coefficient
- Government spending: $G_t = G$ (constant)

We have the national income identity

$$Y_t = C_t + I_t + G_t$$

Equations yields the second-order difference equation:

$$Y_t = (\gamma + G) + (a + b)Y_{t-1} - bY_{t-2} + \sigma\epsilon_t$$

With $\rho_1 = a + b$ and $\rho_2 = -b$, we have:

$$Y_t = (\gamma + G) + \rho_1 Y_{t-1} + \rho_2 Y_{t-2} + \sigma\epsilon_t$$

To fit into our discussion, we write it into state-space representation.

To handle the constant term properly, we use an augmented state vector $\mathbf{x}_t = [1, Y_t, Y_{t-1}]'$:

$$\mathbf{x}_{t+1} = \begin{bmatrix} 1 \\ Y_{t+1} \\ Y_t \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ \gamma + G & \rho_1 & \rho_2 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ Y_t \\ Y_{t-1} \end{bmatrix} + \begin{bmatrix} 0 \\ \sigma \\ 0 \end{bmatrix} \epsilon_{t+1}$$

The observation equation extracts the economic variables:

$$\mathbf{y}_t = \begin{bmatrix} Y_t \\ C_t \\ I_t \end{bmatrix} = \begin{bmatrix} \gamma + G & \rho_1 & \rho_2 \\ \gamma & a & 0 \\ 0 & b & -b \end{bmatrix} \begin{bmatrix} 1 \\ Y_t \\ Y_{t-1} \end{bmatrix}$$

This gives us:

- $Y_t = (\gamma + G) \cdot 1 + \rho_1 Y_{t-1} + \rho_2 Y_{t-2}$ (total output)
- $C_t = \gamma \cdot 1 + aY_{t-1}$ (consumption)
- $I_t = b(Y_{t-1} - Y_{t-2})$ (investment)

```
def samuelson_to_var(a, b, γ, G, σ):
    """
    Convert Samuelson model parameters to VAR form with augmented state

    Samuelson model:
    - Y_t = C_t + I_t + G
    - C_t = γ + a*Y_{t-1}
    - I_t = b*(Y_{t-1} - Y_{t-2})

    Reduced form: Y_t = (γ+G) + (a+b)*Y_{t-1} - b*Y_{t-2} + σ*ε_t

    State vector is [1, Y_t, Y_{t-1}]'
    """
    ρ_1 = a + b
    ρ_2 = -b

    # State transition matrix for augmented state
    A = np.array([[1,       0,      0],
                  [γ + G,   ρ_1,    ρ_2],
                  [0,       1,      0]])

    # Shock loading matrix
    C = np.array([[0],
                  [σ],
                  [0]])
```

(continues on next page)

(continued from previous page)

```
    # Observation matrix (extracts Y_t, C_t, I_t)
    G_obs = np.array([[γ + G,  ρ_1,  ρ_2],    # Y_t
                      [γ,      a,    0],      # C_t
                      [0,      b,   -b]])     # I_t

    return A, C, G_obs
```

We define functions in the code cell below to get the initial conditions and check stability

Let's implement it and inspect the likelihood ratio processes induced by two Samuelson models with different parameters.

```
def create_samuelson_var_model(a, b, γ, G, σ, stationary_init=False,
                               y_0=None, y_m1=None):
    """
    Create a VAR model from Samuelson parameters
    """
    A, C, G_obs = samuelson_to_var(a, b, γ, G, σ)

    μ_0, Σ_0 = get_samuelson_initial_conditions(
        a, b, γ, G, y_0, y_m1, stationary_init
    )

    # Create VAR model
    model = create_var_model(A, C, μ_0, Σ_0, stationary=False)
    is_stable, roots, max_root, dynamics = check_samuelson_stability(a, b)
    info = {
        'a': a, 'b': b, 'γ': γ, 'G': G, 'σ': σ,
        'ρ_1': a + b, 'ρ_2': -b,
        'steady_state': (γ + G) / (1 - a - b),
        'is_stable': is_stable,
        'roots': roots,
        'max_abs_root': max_root,
        'dynamics': dynamics
    }

    return model, G_obs, info

def simulate_samuelson(model, G_obs, T, N_paths=1):
    """
    Simulate Samuelson model
    """
    # Simulate state paths
    states = simulate_var(model, T, N_paths)

    # Extract observables using G matrix
    if N_paths == 1:
        # Single path: states is (T+1, 3)
        observables = (G_obs @ states.T).T
    else:
        # Multiple paths: states is (N_paths, T+1, 3)
        observables = np.zeros((N_paths, T+1, 3))
        for i in range(N_paths):
            observables[i] = (G_obs @ states[i].T).T

    return states, observables
```

Now let's simulate two Samuelson models with different accelerator coefficients and plot their sample paths

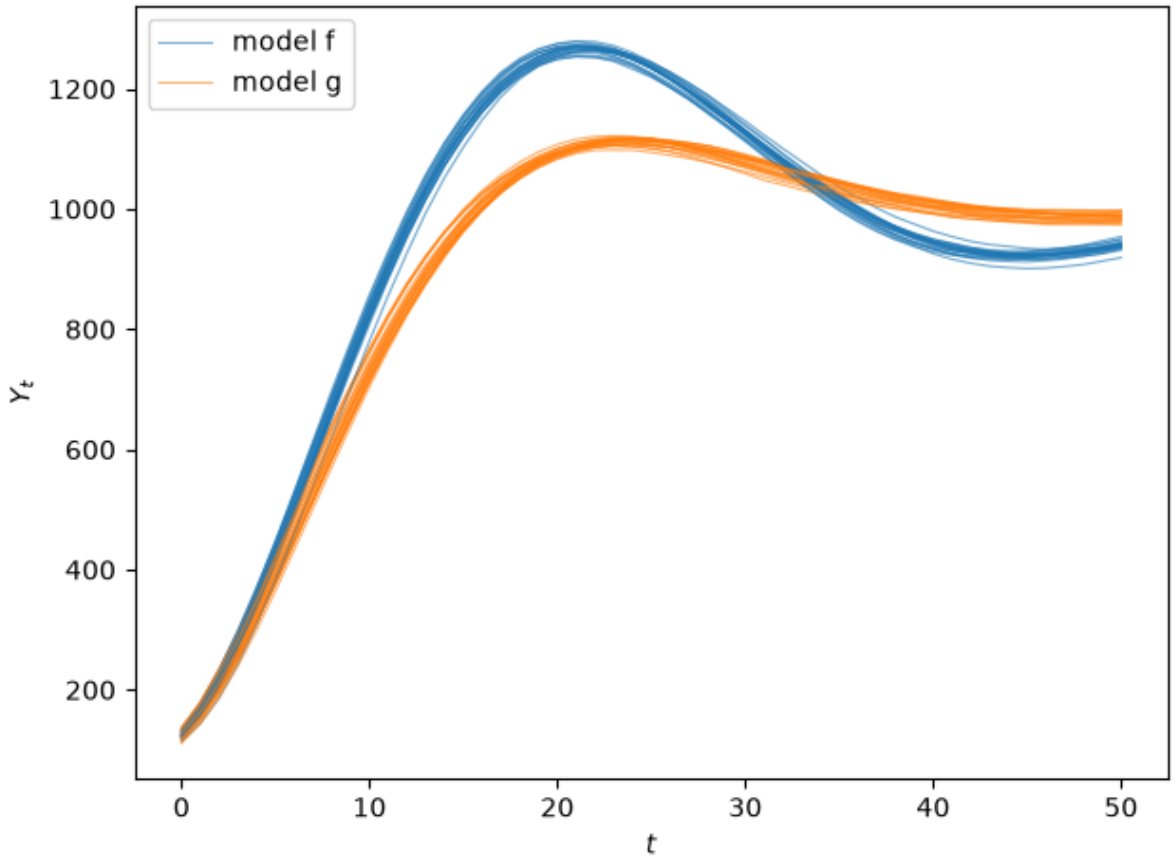
```
# Model f: Higher accelerator coefficient
a_f, b_f = 0.98, 0.9
γ_f, G_f, σ_f = 10, 10, 0.5

# Model g: Lower accelerator coefficient
a_g, b_g = 0.98, 0.85
γ_g, G_g, σ_g = 10, 10, 0.5


model_sam_f, G_obs_f, info_f = create_samuelson_var_model(
    a_f, b_f, γ_f, G_f, σ_f,
    stationary_init=False,
    y_0=100, y_m1=95
)

model_sam_g, G_obs_g, info_g = create_samuelson_var_model(
    a_g, b_g, γ_g, G_g, σ_g,
    stationary_init=False,
    y_0=100, y_m1=95
)

T = 50
N_paths = 50

# Get both states and observables
states_f, obs_f = simulate_samuelson(model_sam_f, G_obs_f, T, N_paths)
states_g, obs_g = simulate_samuelson(model_sam_g, G_obs_g, T, N_paths)

output_paths_f = obs_f[:, :, 0]
output_paths_g = obs_g[:, :, 0]

print("model f:")
print(f"  ρ_1 = a + b = {info_f['ρ_1']:.2f}")
print(f"  ρ_2 = -b = {info_f['ρ_2']:.2f}")
print(f"  roots: {info_f['roots']}")
print(f"  dynamics: {info_f['dynamics']}")

print("\nmodel g:")
print(f"  ρ_1 = a + b = {info_g['ρ_1']:.2f}")
print(f"  ρ_2 = -b = {info_g['ρ_2']:.2f}")
print(f"  roots: {info_g['roots']}")
print(f"  dynamics: {info_g['dynamics']}")


fig, ax = plt.subplots(1, 1)

for i in range(min(20, N_paths)):
    ax.plot(output_paths_f[i], alpha=0.6, color='C0', linewidth=0.8)
    ax.plot(output_paths_g[i], alpha=0.6, color='C1', linewidth=0.8)
ax.set_xlabel('$t$')
ax.set_ylabel('$Y_t$')
ax.legend(['model f', 'model g'], loc='upper left')
plt.tight_layout()
plt.show()
```

```
model f:
  ρ_1 = a + b = 1.88
```

(continues on next page)

(continued from previous page)

```
  ρ_2 = -b = -0.90
  roots: [0.94+0.12806248j 0.94-0.12806248j]
  dynamics: Damped oscillations

model g:
  ρ_1 = a + b = 1.83
  ρ_2 = -b = -0.85
  roots: [0.915+0.11302655j 0.915-0.11302655j]
  dynamics: Damped oscillations
```

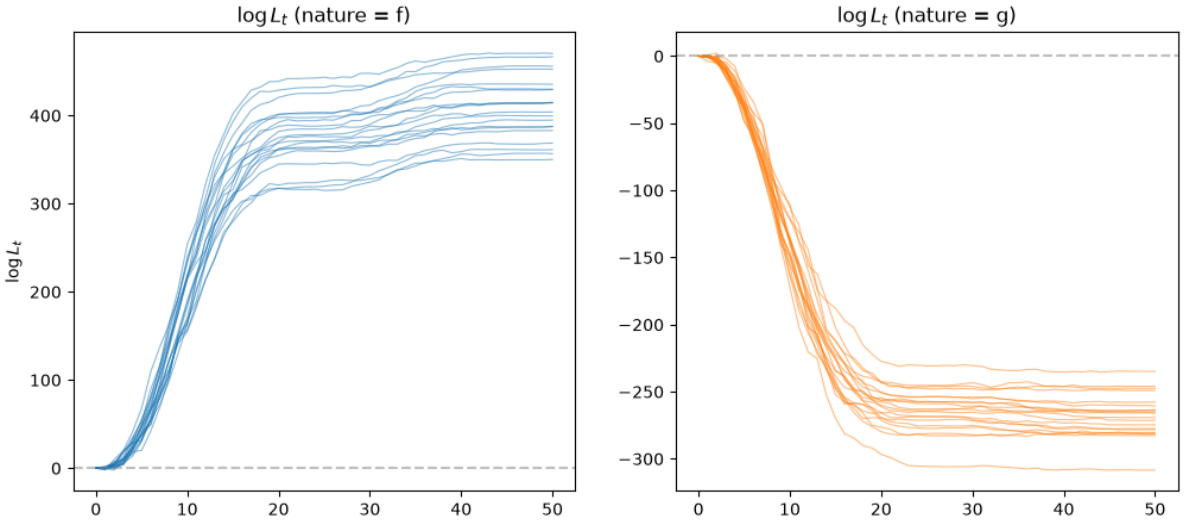



```
# Compute likelihood ratios
L_ratios_ff = compute_likelihood_ratio_var(states_f, model_sam_f, model_sam_g)
L_ratios_gf = compute_likelihood_ratio_var(states_g, model_sam_f, model_sam_g)

fig, axes = plt.subplots(1, 2, figsize=(12, 5))

ax = axes[0]
for i in range(min(20, N_paths)):
    ax.plot(L_ratios_ff[i], alpha=0.5, color='C0', lw=0.8)
ax.axhline(y=0, color='gray', linestyle='--', alpha=0.5)
ax.set_title(r'$\log L_t$ (nature = f)')
ax.set_ylabel(r'$\log L_t$')

ax = axes[1]
for i in range(min(20, N_paths)):
```

(continues on next page)

(continued from previous page)

```
    ax.plot(L_ratios_gf[i], alpha=0.5, color='C1', lw=0.8)
ax.axhline(y=0, color='gray', linestyle='--', alpha=0.5)
ax.set_title(r'$\log L_t$ (nature = g)')
plt.show()
```



In the figure on the left, data are generated by f and the likelihood ratio diverges to plus infinity.

In the figure on the right, data are generated by g and the likelihood ratio diverges to negative infinity.

In both cases, we applied a lower and upper threshold for the log likelihood ratio process for numerical stability since they grow unbounded very quickly.

In both cases, the likelihood ratio processes eventually lead us to select the correct model.

CHAPTER

TWENTYFIVE

MEAN OF A LIKELIHOOD RATIO PROCESS

Contents

- *Mean of a Likelihood Ratio Process*
  - *Overview*
  - *Mathematical expectation of likelihood ratio*
  - *Importance sampling*
  - *Selecting a sampling distribution*
  - *Approximating a cumulative likelihood ratio*
  - *Distribution of sample mean*
  - *Choosing a sampling distribution*

25.1 Overview

In *this lecture* we described a peculiar property of a likelihood ratio process, namely, that its mean equals one for all $t \geq 0$ despite it's converging to zero almost surely.

While it is easy to verify that peculiar properly analytically (i.e., in population), it is challenging to use a computer simulation to verify it via an application of a law of large numbers that entails studying sample averages of repeated simulations.

To confront this challenge, this lecture puts **importance sampling** to work to accelerate convergence of sample averages to population means.

We use importance sampling to estimate the mean of a cumulative likelihood ratio $L\left(\omega^t\right) = \prod_{i=1}^t \ell\left(\omega_i\right)$.

We start by importing some Python packages.

```
import numpy as np
from numba import jit, vectorize, prange
import matplotlib.pyplot as plt
from math import gamma
```

25.2 Mathematical expectation of likelihood ratio

In *this lecture*, we studied a likelihood ratio $\ell(\omega_t)$

$$\ell(\omega_t) = \frac{f(\omega_t)}{g(\omega_t)}$$

where f and g are densities for Beta distributions with parameters F_a, F_b, G_a, G_b.

Assume that an i.i.d. random variable $\omega_t \in \Omega$ is generated by g.

The **cumulative likelihood ratio** $L(\omega^t)$ is

$$L(\omega^t) = \prod_{i=1}^{t} \ell(\omega_i)$$

Our goal is to approximate the mathematical expectation $E[L(\omega^t)]$ well.

In *this lecture*, we showed that $E[L(\omega^t)]$ equals 1 for all t.

We want to check out how well this holds if we replace E by with sample averages from simulations.

This turns out to be easier said than done because for Beta distributions assumed above, $L(\omega^t)$ has a very skewed distribution with a very long tail as $t \to \infty$.

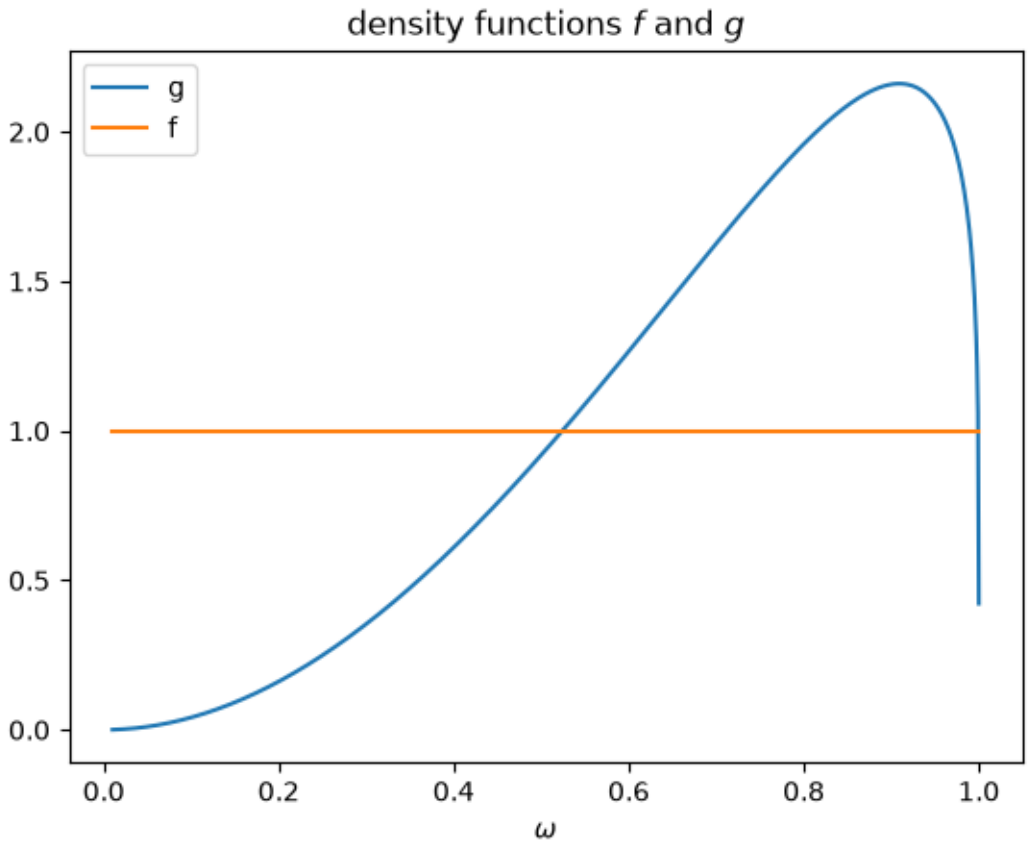
This property makes it difficult efficiently and accurately to estimate the mean by standard Monte Carlo simulation methods.

In this lecture we explore how a standard Monte Carlo method fails and how **importance sampling** provides a more computationally efficient way to approximate the mean of the cumulative likelihood ratio.

We first take a look at the density functions `f` and `g` .

```
# Parameters in the two beta distributions.
F_a, F_b = 1, 1
G_a, G_b = 3, 1.2

@vectorize
def p(w, a, b):
    r = gamma(a + b) / (gamma(a) * gamma(b))
    return r * w ** (a-1) * (1 - w) ** (b-1)

# The two density functions.
f = jit(lambda w: p(w, F_a, F_b))
g = jit(lambda w: p(w, G_a, G_b))
```

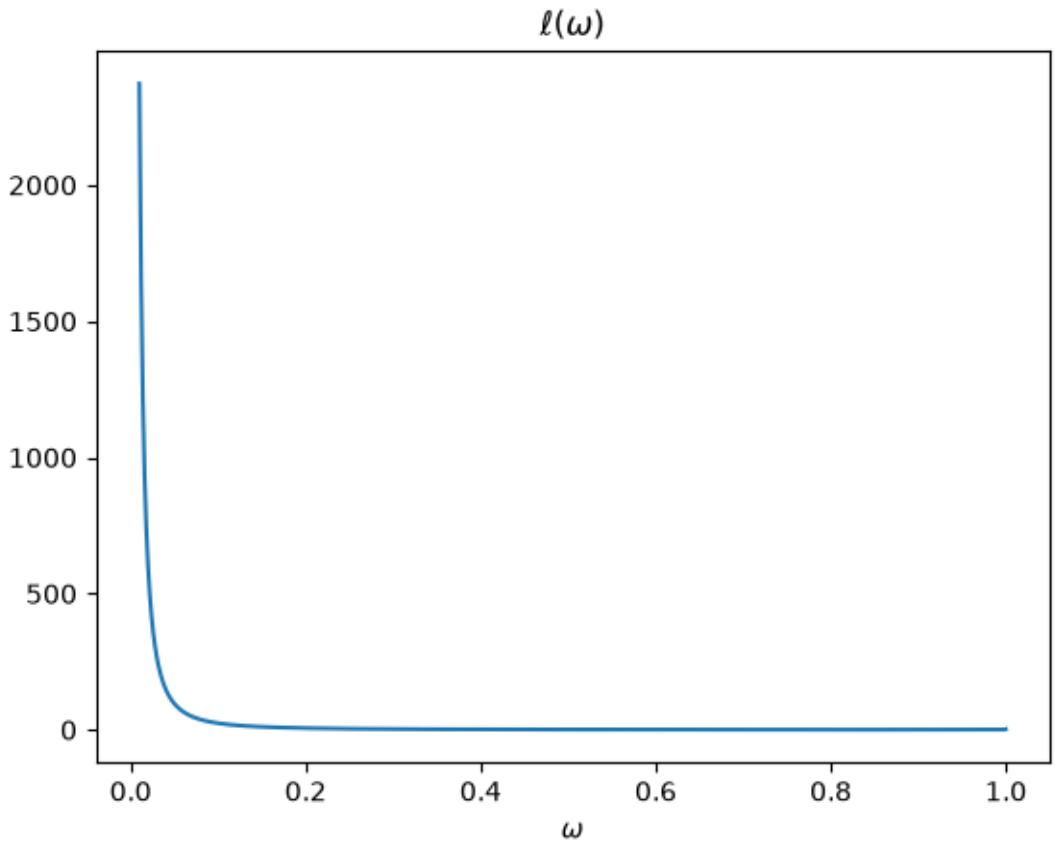
```
w_range = np.linspace(1e-2, 1-1e-5, 1000)

plt.plot(w_range, g(w_range), label='g')
plt.plot(w_range, f(w_range), label='f')
plt.xlabel(r'$\omega$')
plt.legend()
plt.title('density functions $f$ and $g$')
plt.show()
```



The likelihood ratio is `l(w)=f(w)/g(w)`.

```
l = jit(lambda w: f(w) / g(w))
```

```
plt.plot(w_range, l(w_range))
plt.title(r'$\ell(\omega)$')
plt.xlabel(r'$\omega$')
plt.show()
```



The above plots shows that as $\omega \to 0$, $f(\omega)$ is unchanged and $g(\omega) \to 0$, so the likelihood ratio approaches infinity.

A Monte Carlo approximation of $\hat{E}[L(\omega^t)] = \hat{E}\left[\prod_{i=1}^{t} \ell(\omega_i)\right]$ would repeatedly draw ω from g, calculate the likelihood ratio $\ell(\omega) = \frac{f(\omega)}{g(\omega)}$ for each draw, then average these over all draws.

Because $g(\omega) \to 0$ as $\omega \to 0$, such a simulation procedure undersamples a part of the sample space $[0, 1]$ that it is important to visit often in order to do a good job of approximating the mathematical expectation of the likelihood ratio $\ell(\omega)$.

We illustrate this numerically below.

25.3 Importance sampling

We circumvent the issue by using a *change of distribution* called **importance sampling**.

Instead of drawing from g to generate data during the simulation, we use an alternative distribution h to generate draws of ω.

The idea is to design h so that it oversamples the region of Ω where $\ell(\omega_t)$ has large values but low density under g.

After we construct a sample in this way, we must then weight each realization by the likelihood ratio of g and h when we compute the empirical mean of the likelihood ratio.

By doing this, we properly account for the fact that we are using h and not g to simulate data.

To illustrate, suppose were interested in $E[\ell(\omega)]$.

We could simply compute:

$$\hat{E}^g \left[\ell \left(\omega \right) \right] = \frac{1}{N} \sum_{i=1}^{N} \ell(w_i^g)$$

where ω_i^g indicates that ω_i is drawn from g.

But using our insight from importance sampling, we could instead calculate the object:

$$\hat{E}^h \left[\ell \left(\omega \right) \frac{g(w)}{h(w)} \right] = \frac{1}{N} \sum_{i=1}^{N} \ell(w_i^h) \frac{g(w_i^h)}{h(w_i^h)}$$

where w_i is now drawn from importance distribution h.

Notice that the above two are exactly the same population objects:

$$E^g \left[\ell \left(\omega \right) \right] = \int_\Omega \ell(\omega) g(\omega) d\omega = \int_\Omega \ell(\omega) \frac{g(\omega)}{h(\omega)} h(\omega) d\omega = E^h \left[\ell \left(\omega \right) \frac{g(\omega)}{h(\omega)} \right]$$

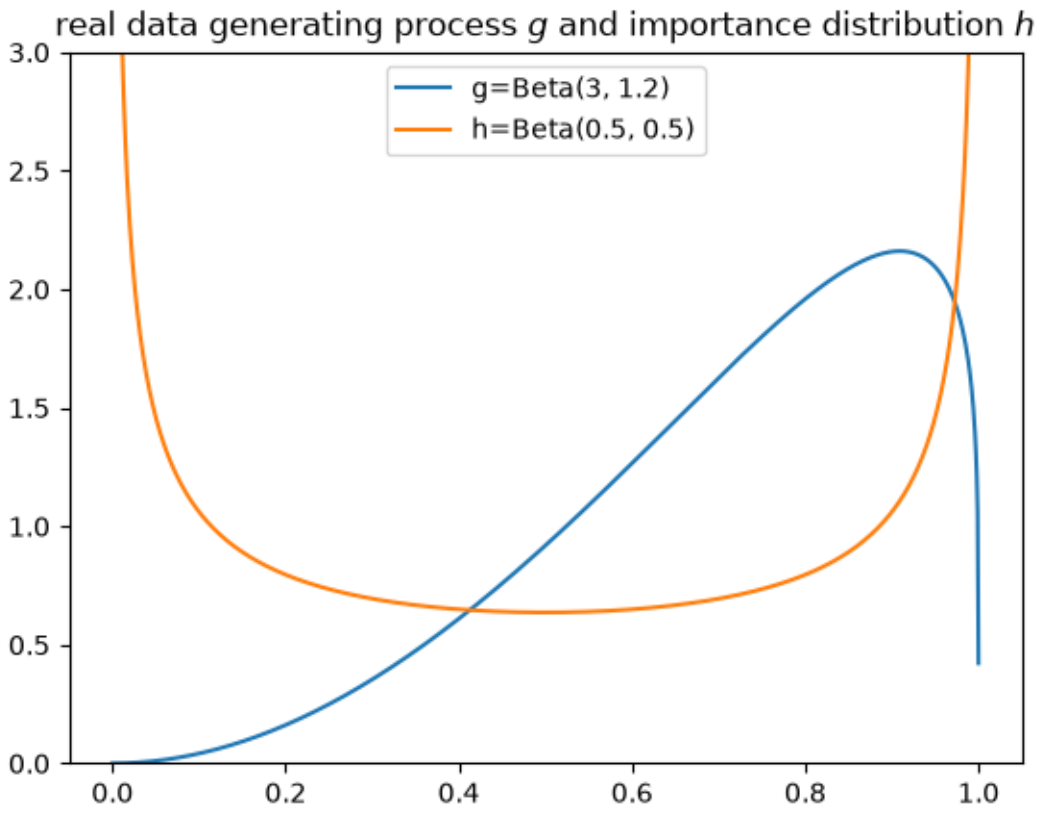
25.4 Selecting a sampling distribution

Since we must use an h that has larger mass in parts of the distribution to which g puts low mass, we use $h = Beta(0.5, 0.5)$ as our importance distribution.

The plots compare g and h.

```
g_a, g_b = G_a, G_b
h_a, h_b = 0.5, 0.5
```

```
w_range = np.linspace(1e-5, 1-1e-5, 1000)

plt.plot(w_range, g(w_range), label=f'g=Beta({g_a}, {g_b})')
plt.plot(w_range, p(w_range, 0.5, 0.5), label=f'h=Beta({h_a}, {h_b})')
plt.title('real data generating process $g$ and importance distribution $h$')
plt.legend()
plt.ylim([0., 3.])
plt.show()
```



25.5 Approximating a cumulative likelihood ratio

We now study how to use importance sampling to approximate $E\left[L(\omega^t)\right] = \left[\prod_{i=1}^T \ell\left(\omega_i\right)\right]$.

As above, our plan is to draw sequences ω^t from q and then re-weight the likelihood ratio appropriately:

$$\hat{E}^p\left[L\left(\omega^t\right)\right] = \hat{E}^p\left[\prod_{t=1}^T \ell\left(\omega_t\right)\right] = \hat{E}^q\left[\prod_{t=1}^T \ell\left(\omega_t\right)\frac{p\left(\omega_t\right)}{q\left(\omega_t\right)}\right] = \frac{1}{N}\sum_{i=1}^N\left(\prod_{t=1}^T \ell(\omega_{i,t}^h)\frac{p\left(\omega_{i,t}^h\right)}{q\left(\omega_{i,t}^h\right)}\right)$$

where the last equality uses $\omega_{i,t}^h$ drawn from the importance distribution q.

Here $\frac{p\left(\omega_{i,t}^q\right)}{q\left(\omega_{i,t}^q\right)}$ is the weight we assign to each data point $\omega_{i,t}^q$.

Below we prepare a Python function for computing the importance sampling estimates given any beta distributions p, q.

```
@jit(parallel=True)
def estimate(p_a, p_b, q_a, q_b, T=1, N=10000):

    μ_L = 0
    for i in prange(N):

        L = 1
        weight = 1
        for t in range(T):
            w = np.random.beta(q_a, q_b)
```

(continues on next page)

(continued from previous page)

```
            l = f(w) / g(w)

            L *= l
            weight *= p(w, p_a, p_b) / p(w, q_a, q_b)

        μ_L += L * weight

    μ_L /= N

    return μ_L
```

Consider the case when $T = 1$, which amounts to approximating $E_0 [\ell (\omega)]$

For the standard Monte Carlo estimate, we can set $p = g$ and $q = g$.

```
estimate(g_a, g_b, g_a, g_b, T=1, N=10000)
```

```
0.9426375445517388
```

For our importance sampling estimate, we set $q = h$.

```
estimate(g_a, g_b, h_a, h_b, T=1, N=10000)
```

```
0.993923633793145
```

Evidently, even at $T = 1$, our importance sampling estimate is closer to 1 than is the Monte Carlo estimate.

Bigger differences arise when computing expectations over longer sequences, $E_0 [L (\omega^t)]$.

Setting $T = 10$, we find that the Monte Carlo method severely underestimates the mean while importance sampling still produces an estimate close to its theoretical value of unity.

```
estimate(g_a, g_b, g_a, g_b, T=10, N=10000)
```

```
0.44966574972451884
```

```
estimate(g_a, g_b, h_a, h_b, T=10, N=10000)
```

```
1.0239033942875697
```

The Monte Carlo method underestimates because the likelihood ratio $L(\omega^T) = \prod_{t=1}^{T} \frac{f(\omega_t)}{g(\omega_t)}$ has a highly skewed distribution under g.

Most samples from g produce small likelihood ratios, while the true mean requires occasional very large values that are rarely sampled.

In our case, since $g(\omega) \to 0$ as $\omega \to 0$ while $f(\omega)$ remains constant, the Monte Carlo procedure undersamples precisely where the likelihood ratio $\frac{f(\omega)}{g(\omega)}$ is largest.

As T increases, this problem worsens exponentially, making standard Monte Carlo increasingly unreliable.

Importance sampling with $q = h$ fixes this by sampling more uniformly from regions important to both f and g.

25.6 Distribution of sample mean

We next study the bias and efficiency of the Monte Carlo and importance sampling approaches.

The code below produces distributions of estimates using both Monte Carlo and importance sampling methods.

```
def simulate(p_a, p_b, q_a, q_b, N_simu, T=1):

    μ_L_p = np.empty(N_simu)
    μ_L_q = np.empty(N_simu)

    for i in range(N_simu):
        μ_L_p[i] = estimate(p_a, p_b, p_a, p_b, T=T)
        μ_L_q[i] = estimate(p_a, p_b, q_a, q_b, T=T)

    return μ_L_p, μ_L_q
```

Again, we first consider estimating $E[\ell(\omega)]$ by setting T=1.

We simulate 1000 times for each method.

```
N_simu = 1000
μ_L_p, μ_L_q = simulate(g_a, g_b, h_a, h_b, N_simu)
```

```
# standard Monte Carlo (mean and std)
np.nanmean(μ_L_p), np.nanvar(μ_L_p)
```

```
(np.float64(0.9962473697194036), np.float64(0.012879091687113795))
```

```
# importance sampling (mean and std)
np.nanmean(μ_L_q), np.nanvar(μ_L_q)
```

```
(np.float64(1.0002781954262128), np.float64(2.3585400343706583e-05))
```

Although both methods tend to provide a mean estimate of $E[\ell(\omega)]$ close to 1, the importance sampling estimates have smaller variance.

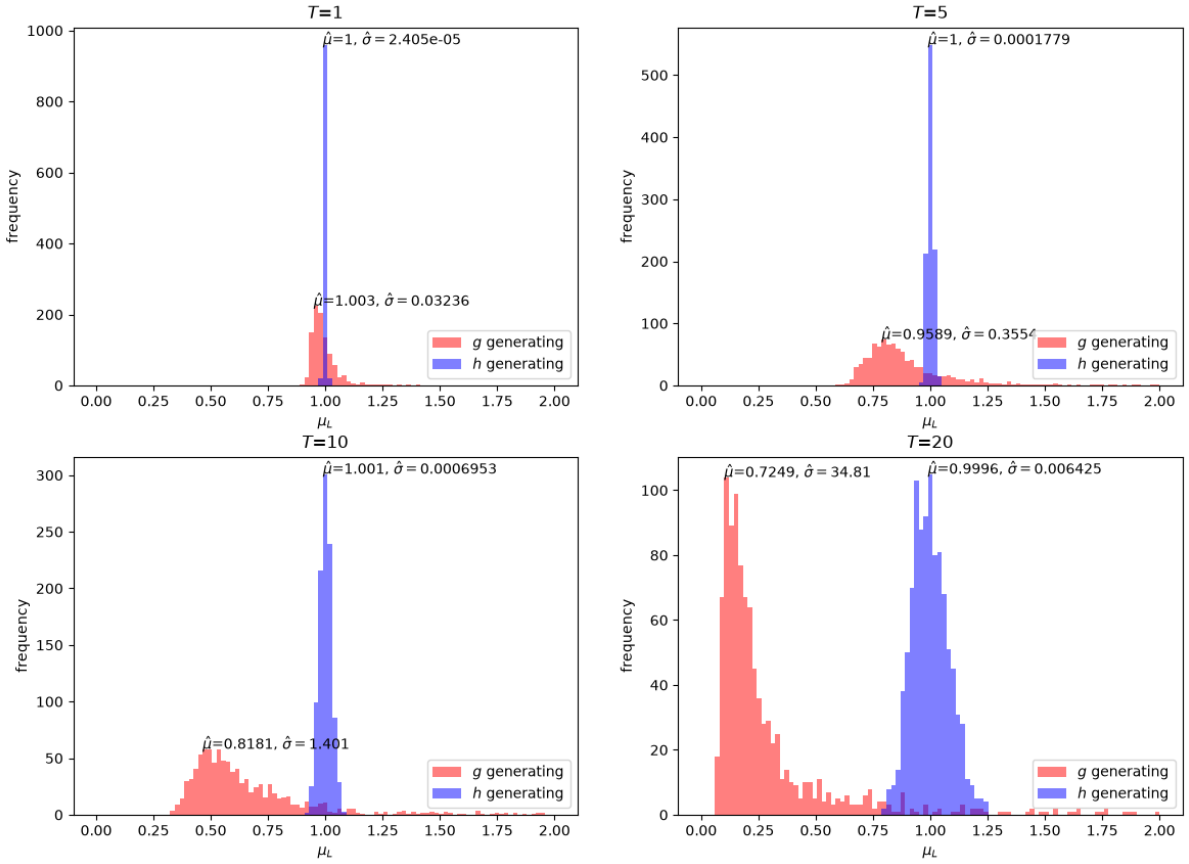
Next, we present distributions of estimates for $\hat{E}[L(\omega^t)]$, in cases for $T = 1, 5, 10, 20$.

```
fig, axs = plt.subplots(2, 2, figsize=(14, 10))

μ_range = np.linspace(0, 2, 100)

for i, t in enumerate([1, 5, 10, 20]):
    row = i // 2
    col = i % 2

    μ_L_p, μ_L_q = simulate(g_a, g_b, h_a, h_b, N_simu, T=t)
    μ_hat_p, μ_hat_q = np.nanmean(μ_L_p), np.nanmean(μ_L_q)
    σ_hat_p, σ_hat_q = np.nanvar(μ_L_p), np.nanvar(μ_L_q)

    axs[row, col].set_xlabel('$μ_L$')
    axs[row, col].set_ylabel('frequency')
    axs[row, col].set_title(f'$T$={t}')
    n_p, bins_p, _ = axs[row, col].hist(μ_L_p, bins=μ_range, color='r', alpha=0.5,␣
↪label='$g$ generating')
```

(continues on next page)

(continued from previous page)

```
    n_q, bins_q, _ = axs[row, col].hist(μ_L_q, bins=μ_range, color='b', alpha=0.5,␣
↪label='$h$ generating')
    axs[row, col].legend(loc=4)

    for n, bins, μ_hat, σ_hat in [[n_p, bins_p, μ_hat_p, σ_hat_p],
                                  [n_q, bins_q, μ_hat_q, σ_hat_q]]:
        idx = np.argmax(n)
        axs[row, col].text(bins[idx], n[idx], r'$\hat{μ}$='+f'{μ_hat:.4g}'+r', $\hat
↪{σ}=$'+f'{σ_hat:.4g}')

plt.show()
```



The simulation exercises above show that the importance sampling estimates are unbiased under all T while the standard Monte Carlo estimates are biased downwards.

Evidently, the bias increases with increases in T.

25.7 Choosing a sampling distribution

Above, we arbitrarily chose $h = Beta(0.5, 0.5)$ as the importance distribution.

Is there an optimal importance distribution?

In our particular case, since we know in advance that $E_0 \left[L \left(\omega^t \right) \right] = 1$, we can use that knowledge to our advantage.

Thus, suppose that we simply use $h = f$.

When estimating the mean of the likelihood ratio (T=1), we get:

$$\hat{E}^f \left[\ell(\omega) \frac{g(\omega)}{f(\omega)} \right] = \hat{E}^f \left[\frac{f(\omega)}{g(\omega)} \frac{g(\omega)}{f(\omega)} \right] = \frac{1}{N} \sum_{i=1}^{N} \ell(w_i^f) \frac{g(w_i^f)}{f(w_i^f)} = 1$$

```
μ_L_p, μ_L_q = simulate(g_a, g_b, F_a, F_b, N_simu)
```

```
# importance sampling (mean and std)
np.nanmean(μ_L_q), np.nanvar(μ_L_q)
```

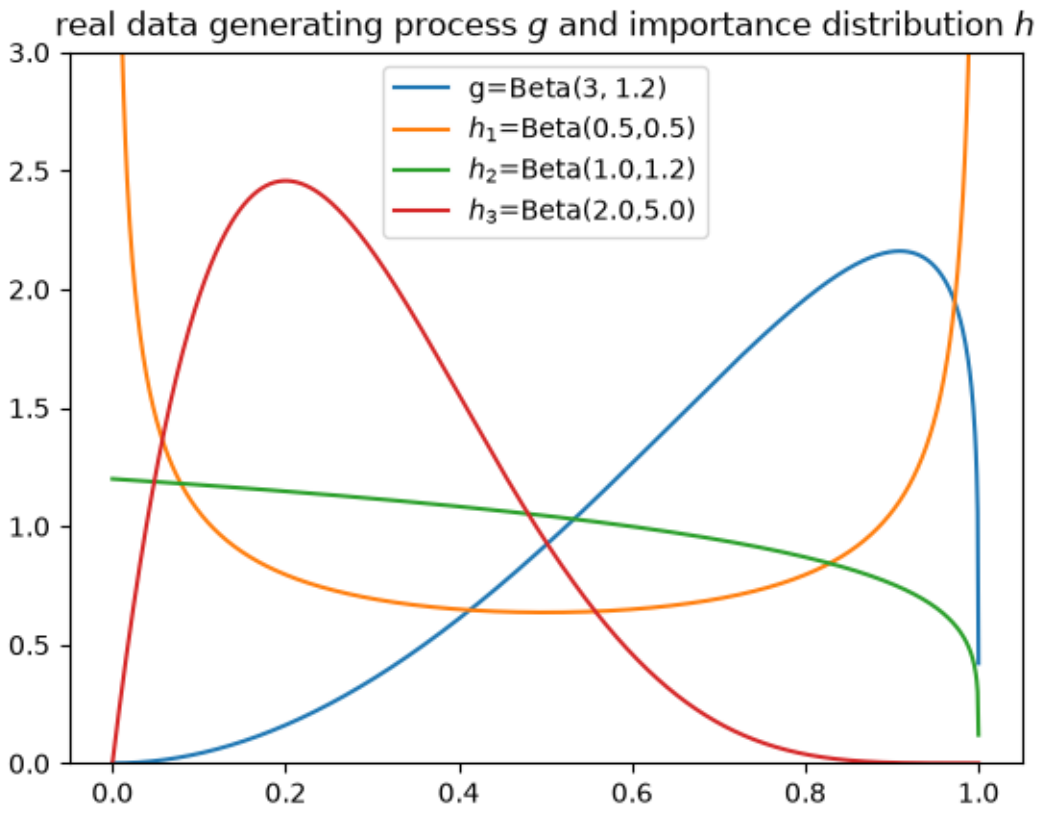
```
(np.float64(1.0), np.float64(0.0))
```

We could also use other distributions as our importance distribution.

Below we choose just a few and compare their sampling properties.

```
a_list = [0.5, 1., 2.]
b_list = [0.5, 1.2, 5.]
```

```
w_range = np.linspace(1e-5, 1-1e-5, 1000)

plt.plot(w_range, g(w_range), label=f'g=Beta({g_a}, {g_b})')
plt.plot(w_range, p(w_range, a_list[0], b_list[0]), label=f'$h_1$=Beta({a_list[0]},{b_
↪list[0]})')
plt.plot(w_range, p(w_range, a_list[1], b_list[1]), label=f'$h_2$=Beta({a_list[1]},{b_
↪list[1]})')
plt.plot(w_range, p(w_range, a_list[2], b_list[2]), label=f'$h_3$=Beta({a_list[2]},{b_
↪list[2]})')
plt.title('real data generating process $g$ and importance distribution $h$')
plt.legend()
plt.ylim([0., 3.])
plt.show()
```



We consider two additional distributions.

As a reminder h_1 is the original $Beta(0.5, 0.5)$ distribution that we used above.

h_2 is the $Beta(1, 1.2)$ distribution.

Note how h_2 has a similar shape to g at higher values of distribution but more mass at lower values.

Our hunch is that h_2 should be a good importance sampling distribution.

h_3 is the $Beta(2, 5)$ distribution.

Note how h_3 has zero mass at values very close to 0 and at values close to 1.

Our hunch is that h_3 will be a poor importance sampling distribution.

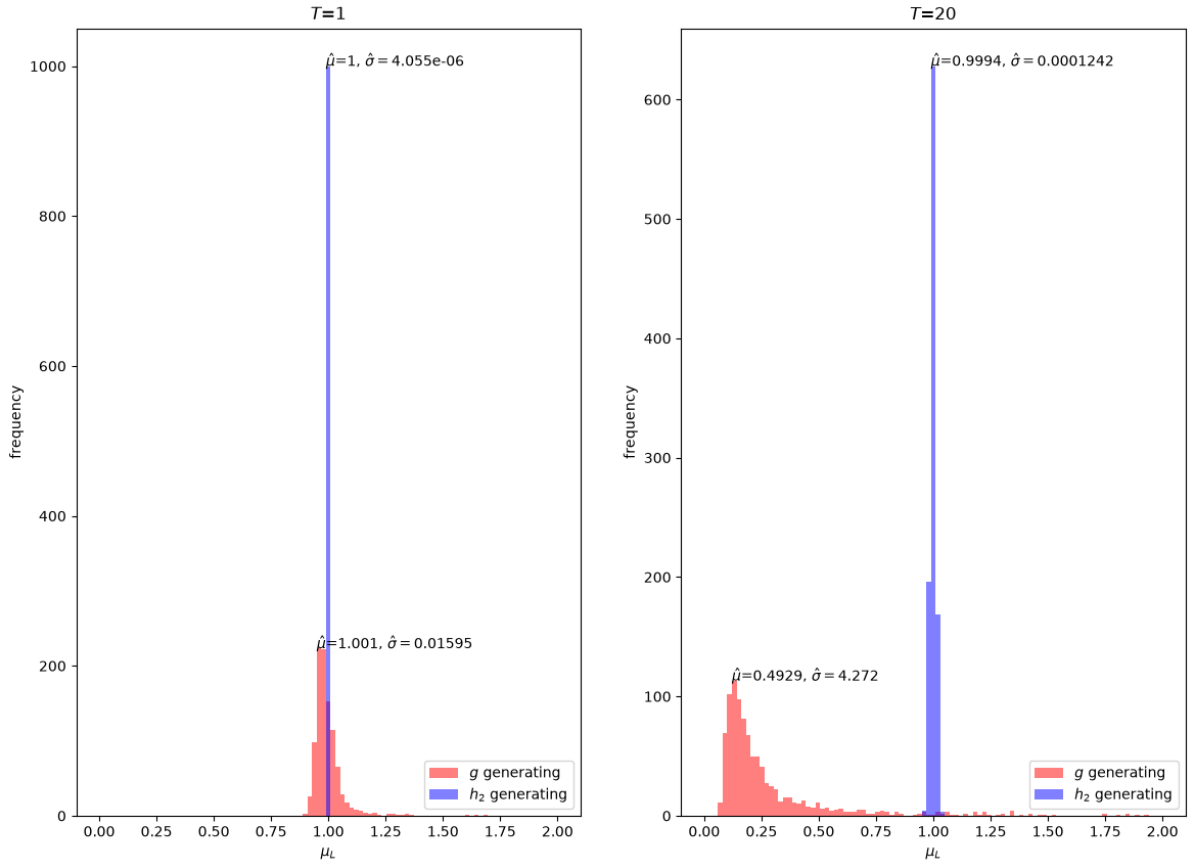
We first simulate a plot the distribution of estimates for $\hat{E}\left[L\left(\omega^t\right)\right]$ using h_2 as the importance sampling distribution.

```
h_a = a_list[1]
h_b = b_list[1]

fig, axs = plt.subplots(1,2, figsize=(14, 10))

μ_range = np.linspace(0, 2, 100)

for i, t in enumerate([1, 20]):


    μ_L_p, μ_L_q = simulate(g_a, g_b, h_a, h_b, N_simu, T=t)
    μ_hat_p, μ_hat_q = np.nanmean(μ_L_p), np.nanmean(μ_L_q)
    σ_hat_p, σ_hat_q = np.nanvar(μ_L_p), np.nanvar(μ_L_q)
```

(continues on next page)

(continued from previous page)

```
    axs[i].set_xlabel('$μ_L$')
    axs[i].set_ylabel('frequency')
    axs[i].set_title(f'$T$={t}')
    n_p, bins_p, _ = axs[i].hist(μ_L_p, bins=μ_range, color='r', alpha=0.5, label='$g
↪$ generating')
    n_q, bins_q, _ = axs[i].hist(μ_L_q, bins=μ_range, color='b', alpha=0.5, label='$h_
↪2$ generating')
    axs[i].legend(loc=4)

    for n, bins, μ_hat, σ_hat in [[n_p, bins_p, μ_hat_p, σ_hat_p],
                                  [n_q, bins_q, μ_hat_q, σ_hat_q]]:
        idx = np.argmax(n)
        axs[i].text(bins[idx], n[idx], r'$\hat{μ}$='+f'{μ_hat:.4g}'+r', $\hat{σ}=$'+f'
↪{σ_hat:.4g}')

plt.show()
```



Our simulations suggest that indeed h_2 is a quite good importance sampling distribution for our problem.

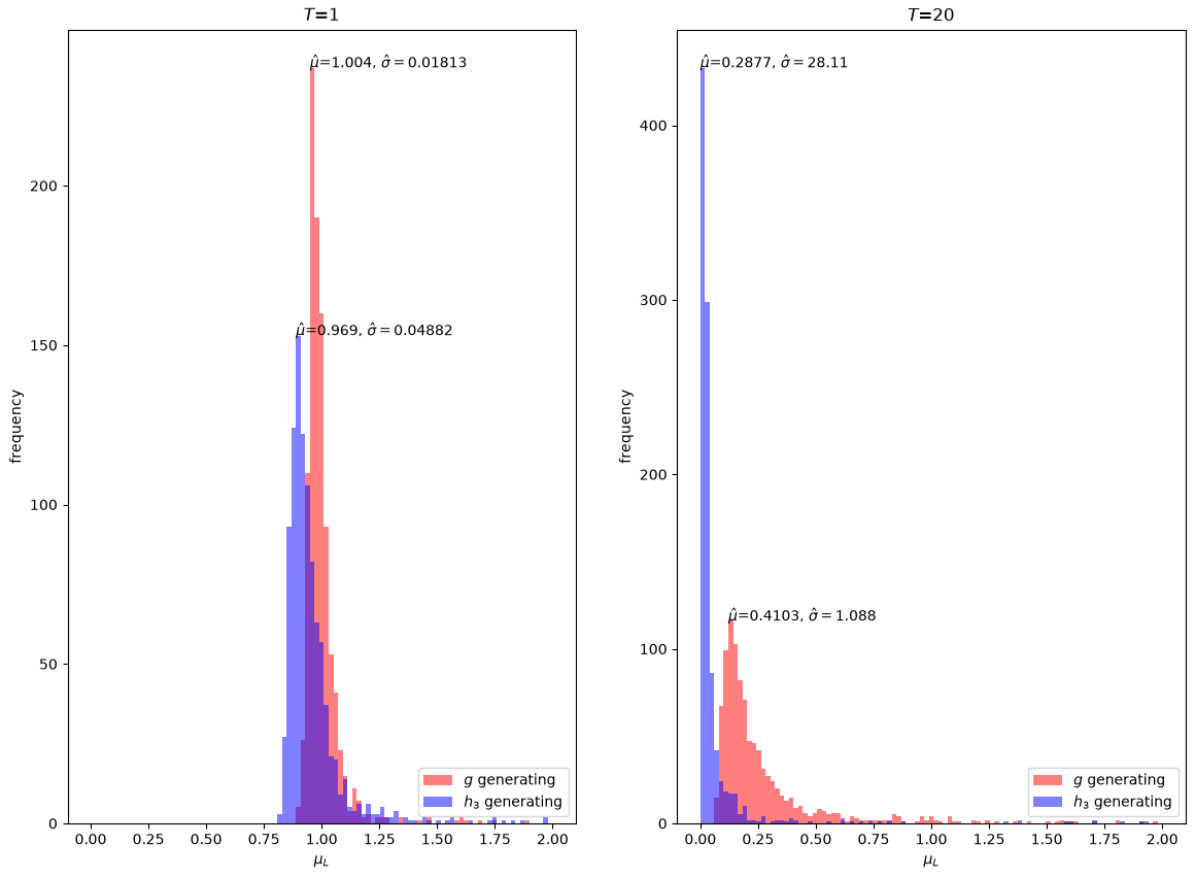
Even at $T = 20$, the mean is very close to 1 and the variance is small.

```
h_a = a_list[2]
h_b = b_list[2]

fig, axs = plt.subplots(1,2, figsize=(14, 10))

μ_range = np.linspace(0, 2, 100)
```

(continues on next page)

(continued from previous page)

```
for i, t in enumerate([1, 20]):


    μ_L_p, μ_L_q = simulate(g_a, g_b, h_a, h_b, N_simu, T=t)
    μ_hat_p, μ_hat_q = np.nanmean(μ_L_p), np.nanmean(μ_L_q)
    σ_hat_p, σ_hat_q = np.nanvar(μ_L_p), np.nanvar(μ_L_q)

    axs[i].set_xlabel('$μ_L$')
    axs[i].set_ylabel('frequency')
    axs[i].set_title(f'$T$={t}')
    n_p, bins_p, _ = axs[i].hist(μ_L_p, bins=μ_range, color='r', alpha=0.5, label='$g
↪$ generating')
    n_q, bins_q, _ = axs[i].hist(μ_L_q, bins=μ_range, color='b', alpha=0.5, label='$h_
↪3$ generating')
    axs[i].legend(loc=4)

    for n, bins, μ_hat, σ_hat in [[n_p, bins_p, μ_hat_p, σ_hat_p],
                                  [n_q, bins_q, μ_hat_q, σ_hat_q]]:
        idx = np.argmax(n)
        axs[i].text(bins[idx], n[idx], r'$\hat{μ}$='+f'{μ_hat:.4g}'+r', $\hat{σ}=$'+f'
↪{σ_hat:.4g}')

plt.show()
```



However, h_3 is evidently a poor importance sampling distribution for our problem, with a mean estimate far away from

1 for $T = 20$.

Notice that even at $T = 1$, the mean estimate with importance sampling is more biased than sampling with just g itself.

Thus, our simulations suggest that for our problem we would be better off simply using Monte Carlo approximations under g than using h_3 as an importance sampling distribution.

CHAPTER

TWENTYSIX

A PROBLEM THAT STUMPED MILTON FRIEDMAN

(and that Abraham Wald solved by inventing sequential analysis)

Contents

- *A Problem that Stumped Milton Friedman*
  - *Overview*
  - *Source of the Problem*
  - *Neyman-Pearson formulation*
  - *Wald's sequential formulation*
  - *Links between A, B and α, β*
  - *Simulations*
  - *Related lectures*
  - *Exercises*

26.1 Overview

This is the first of two lectures about a statistical decision problem that a US Navy Captain presented to Milton Friedman and W. Allen Wallis during World War II when they were analysts at the U.S. Government's Statistical Research Group at Columbia University.

This problem led Abraham Wald [Wald, 1947] to formulate **sequential analysis**, an approach to statistical decision problems that is intimately related to dynamic programming.

In the spirit of *this earlier lecture*, the present lecture and its *sequel* approach the problem from two distinct points of view, one frequentist, the other Bayesian.

In this lecture, we describe Wald's formulation of the problem from the perspective of a statistician working within the Neyman-Pearson tradition of a frequentist statistician who thinks about testing hypotheses and consequently use laws of large numbers to investigate limiting properties of particular statistics under a given **hypothesis**, i.e., a vector of **parameters** that pins down a particular member of a manifold of statistical models that interest the statistician.

- From *this lecture on frequentist and bayesian statistics*, please remember that a frequentist statistician routinely calculates functions of sequences of random variables, conditioning on a vector of parameters.

In *this related lecture* we'll discuss another formulation that adopts the perspective of a **Bayesian statistician** who views parameters as random variables that are jointly distributed with observable variables that he is concerned about.

Because we are taking a frequentist perspective that is concerned about relative frequencies conditioned on alternative parameter values, i.e., alternative **hypotheses**, key ideas in this lecture

- Type I and type II statistical errors
  - a type I error occurs when you reject a null hypothesis that is true
  - a type II error occurs when you accept a null hypothesis that is false
- The **power** of a frequentist statistical test
- The **size** of a frequentist statistical test
- The **critical region** of a statistical test
- A **uniformly most powerful test**
- The role of a Law of Large Numbers (LLN) in interpreting **power** and **size** of a frequentist statistical test
- Abraham Wald's **sequential probability ratio test**

We'll begin with some imports:

```
import numpy as np
import matplotlib.pyplot as plt
from numba import njit, prange, vectorize, jit
from numba.experimental import jitclass
from math import gamma
from scipy.integrate import quad
from scipy.stats import beta
from collections import namedtuple
import pandas as pd
import scipy as sp
```

This lecture uses ideas studied in *the lecture on likelihood ratio processes* and *the lecture on Bayesian learning*.

26.2 Source of the Problem

On pages 137-139 of his 1998 book *Two Lucky People* with Rose Friedman [Friedman and Friedman, 1998], Milton Friedman described a problem presented to him and Allen Wallis during World War II, when they worked at the US Government's Statistical Research Group at Columbia University.

Note

See pages 25 and 26 of Allen Wallis's 1980 article [Wallis, 1980] about the Statistical Research Group at Columbia University during World War II for his account of the episode and for important contributions that Harold Hotelling made to formulating the problem. Also see chapter 5 of Jennifer Burns' book about Milton Friedman [Burns, 2023].

Let's listen to Milton Friedman tell us what happened

> In order to understand the story, it is necessary to have an idea of a simple statistical problem, and of the standard procedure for dealing with it. The actual problem out of which sequential analysis grew will serve. The Navy has two alternative designs (say A and B) for a projectile. It wants to determine which is superior. To do so it undertakes a series of paired firings. On each round, it assigns the value 1 or 0 to A accordingly as

> its performance is superior or inferior to that of B and conversely 0 or 1 to B. The Navy asks the statistician how to conduct the test and how to analyze the results.
>
> The standard statistical answer was to specify a number of firings (say 1,000) and a pair of percentages (e.g., 53% and 47%) and tell the client that if A receives a 1 in more than 53% of the firings, it can be regarded as superior; if it receives a 1 in fewer than 47%, B can be regarded as superior; if the percentage is between 47% and 53%, neither can be so regarded.
>
> When Allen Wallis was discussing such a problem with (Navy) Captain Garret L. Schuyler, the captain objected that such a test, to quote from Allen's account, may prove wasteful. If a wise and seasoned ordnance officer like Schuyler were on the premises, he would see after the first few thousand or even few hundred [rounds] that the experiment need not be completed either because the new method is obviously inferior or because it is obviously superior beyond what was hoped for ….

Friedman and Wallis worked on the problem for a while but didn't completely solve it.

Realizing that, they told Abraham Wald about the problem.

That set Wald on a path that led him to create *Sequential Analysis* [Wald, 1947].

26.3 Neyman-Pearson formulation

It is useful to begin by describing the theory underlying the test that the U.S. Navy told Captain G. S. Schuyler to use.

Captain Schuyler's doubts motivated him to tell Milton Friedman and Allen Wallis his conjecture that superior practical procedures existed.

Evidently, the Navy had told Captain Schuyler to use what was then a state-of-the-art Neyman-Pearson hypothesis test.

We'll rely on Abraham Wald's [Wald, 1947] elegant summary of Neyman-Pearson theory.

Watch for these features of the setup:

- the assumption of a *fixed* sample size n
- the application of laws of large numbers, conditioned on alternative probability models, to interpret probabilities α and β of the type I and type II errors defined in the Neyman-Pearson theory

In chapter 1 of **Sequential Analysis** [Wald, 1947] Abraham Wald summarizes the Neyman-Pearson approach to hypothesis testing.

Wald frames the problem as making a decision about a probability distribution that is partially known.

(You have to assume that *something* is already known in order to state a well-posed problem – usually, *something* means *a lot*)

By limiting what is unknown, Wald uses the following simple structure to illustrate the main ideas:

- A decision-maker wants to decide which of two distributions f_0, f_1 govern an IID random variable z.
- The null hypothesis H_0 is the statement that f_0 governs the data.
- The alternative hypothesis H_1 is the statement that f_1 governs the data.
- The problem is to devise and analyze a test of hypothesis H_0 against the alternative hypothesis H_1 on the basis of a sample of a fixed number n independent observations $z_1, z_2, \ldots, z_n$ of the random variable z.

To quote Abraham Wald,

> A test procedure leading to the acceptance or rejection of the [null] hypothesis in question is simply a rule specifying, for each possible sample of size n, whether the [null] hypothesis should be accepted or rejected on the basis of the sample. This may also be expressed as follows: A test procedure is simply a subdivision of the totality of all possible samples of size n into two mutually exclusive parts, say part 1 and part 2, together

> with the application of the rule that the [null] hypothesis be accepted if the observed sample is contained in part 2. Part 1 is also called the critical region. Since part 2 is the totality of all samples of size n which are not included in part 1, part 2 is uniquely determined by part 1. Thus, choosing a test procedure is equivalent to determining a critical region.

Let's listen to Wald longer:

> As a basis for choosing among critical regions the following considerations have been advanced by Neyman and Pearson: In accepting or rejecting H_0 we may commit errors of two kinds. We commit an error of the first kind if we reject H_0 when it is true; we commit an error of the second kind if we accept H_0 when H_1 is true. After a particular critical region W has been chosen, the probability of committing an error of the first kind, as well as the probability of committing an error of the second kind is uniquely determined. The probability of committing an error of the first kind is equal to the probability, determined by the assumption that H_0 is true, that the observed sample will be included in the critical region W. The probability of committing an error of the second kind is equal to the probability, determined on the assumption that H_1 is true, that the probability will fall outside the critical region W. For any given critical region W we shall denote the probability of an error of the first kind by α and the probability of an error of the second kind by β.

Let's listen carefully to how Wald applies law of large numbers to interpret α and β:

> The probabilities α and β have the following important practical interpretation: Suppose that we draw a large number of samples of size n. Let M be the number of such samples drawn. Suppose that for each of these M samples we reject H_0 if the sample is included in W and accept H_0 if the sample lies outside W. In this way we make M statements of rejection or acceptance. Some of these statements will in general be wrong. If H_0 is true and if M is large, the probability is nearly 1 (i.e., it is practically certain) that the proportion of wrong statements (i.e., the number of wrong statements divided by M) will be approximately α. If H_1 is true, the probability is nearly 1 that the proportion of wrong statements will be approximately β. Thus, we can say that in the long run [here Wald applies law of large numbers by driving $M \to \infty$ (our comment, not Wald's)] the proportion of wrong statements will be α if H_0 is true and β if H_1 is true.

The quantity α is called the *size* of the critical region, and the quantity $1 - \beta$ is called the *power* of the critical region.

Wald notes that

> one critical region W is more desirable than another if it has smaller values of α and β. Although either α or β can be made arbitrarily small by a proper choice of the critical region W, it is impossible to make both α and β arbitrarily small for a fixed value of n, i.e., a fixed sample size.

Wald summarizes Neyman and Pearson's setup as follows:

> Neyman and Pearson show that a region consisting of all samples $(z_1, z_2, \ldots, z_n)$ which satisfy the inequality
>
> $$\frac{f_1(z_1) \cdots f_1(z_n)}{f_0(z_1) \cdots f_0(z_n)} \geq k$$
>
> is a most powerful critical region for testing the hypothesis H_0 against the alternative hypothesis H_1. The term k on the right side is a constant chosen so that the region will have the required size α.

Wald goes on to discuss Neyman and Pearson's concept of *uniformly most powerful* test.

Here is how Wald introduces the notion of a sequential test

> A rule is given for making one of the following three decisions at any stage of the experiment (at the m th trial for each integral value of m): (1) to accept the hypothesis H, (2) to reject the hypothesis H, (3) to continue the experiment by making an additional observation. Thus, such a test procedure is carried out sequentially. On the basis of the first observation, one of the aforementioned decision is made. If the first or second decision is made, the process is terminated. If the third decision is made, a second trial is performed. Again, on the basis of the first two observations, one of the three decision is made. If the third decision is made, a third trial is performed, and so on. The process is continued until either the first or the second

decisions is made. The number n of observations required by such a test procedure is a random variable, since the value of n depends on the outcome of the observations.

26.4 Wald's sequential formulation

By way of contrast to Neyman and Pearson's formulation of the problem, in Wald's formulation

- The sample size n is not fixed but rather a random variable.
- Two parameters A and B that are related to but distinct from Neyman and Pearson's α and β; A and B characterize cut-off rules that Wald uses to determine the random variable n as a function of random outcomes.

Here is how Wald sets up the problem.

A decision-maker can observe a sequence of draws of a random variable z.

He (or she) wants to know which of two probability distributions f_0 or f_1 governs z.

We use beta distributions as examples.

We will also work with Jensen-Shannon divergence introduced in *Statistical Divergence Measures*.

```
@vectorize
def p(x, a, b):
    """Beta distribution density function."""
    r = gamma(a + b) / (gamma(a) * gamma(b))
    return r * x** (a-1) * (1 - x) ** (b-1)

def create_beta_density(a, b):
    """Create a beta density function with specified parameters."""
    return jit(lambda x: p(x, a, b))

def compute_KL(f, g):
    """Compute KL divergence KL(f, g)"""
    integrand = lambda w: f(w) * np.log(f(w) / g(w))
    val, _ = quad(integrand, 1e-5, 1-1e-5)
    return val

def compute_JS(f, g):
    """Compute Jensen-Shannon divergence"""
    def m(w):
        return 0.5 * (f(w) + g(w))

    js_div = 0.5 * compute_KL(f, m) + 0.5 * compute_KL(g, m)
    return js_div
```

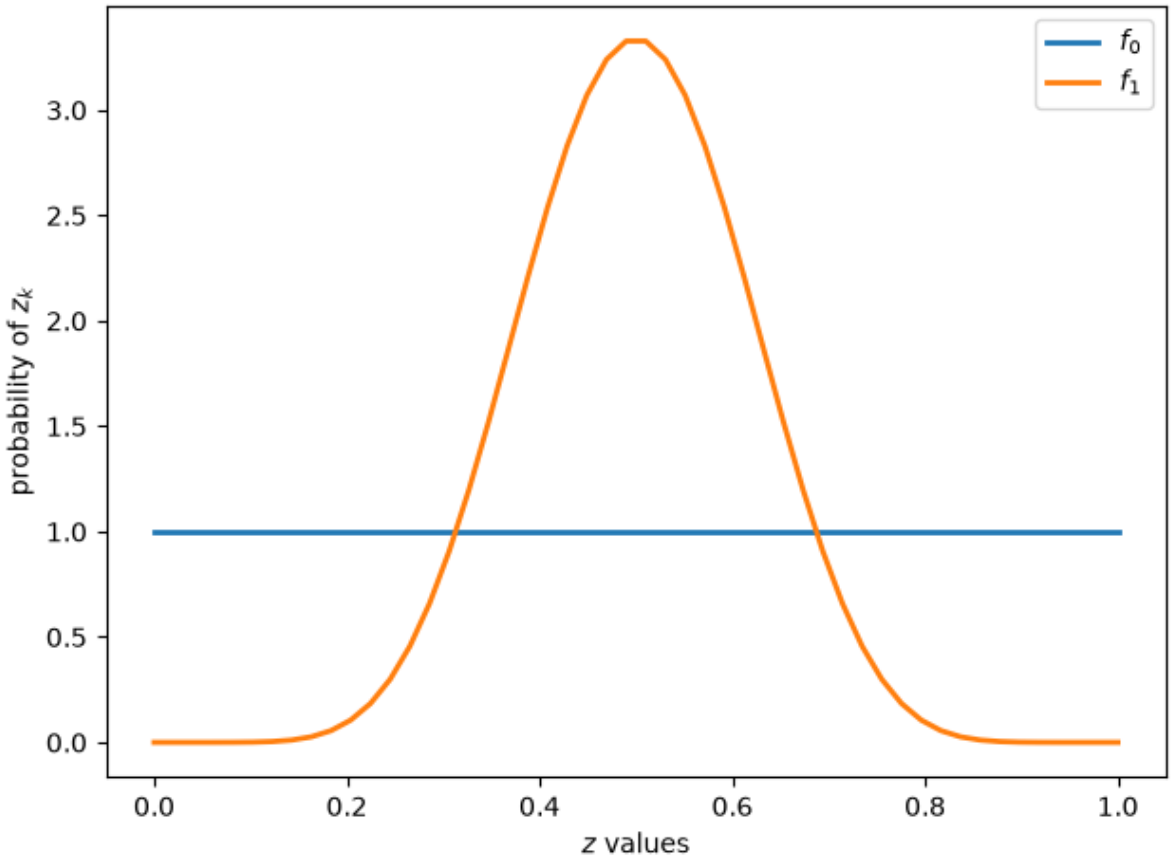
The next figure shows two beta distributions

```
f0 = create_beta_density(1, 1)
f1 = create_beta_density(9, 9)
grid = np.linspace(0, 1, 50)

fig, ax = plt.subplots()
ax.plot(grid, f0(grid), lw=2, label="$f_0$")
ax.plot(grid, f1(grid), lw=2, label="$f_1$")
ax.legend()
ax.set(xlabel="$z$ values", ylabel="probability of $z_k$")
```

(continues on next page)

(continued from previous page)

```
plt.tight_layout()
plt.show()
```



Conditional on knowing that successive observations are drawn from distribution f_0, the sequence of random variables is independently and identically distributed (IID).

Conditional on knowing that successive observations are drawn from distribution f_1, the sequence of random variables is also independently and identically distributed (IID).

But the observer does not know which of the two distributions generated the sequence.

For reasons explained in Exchangeability and Bayesian Updating, this means that the observer thinks that the sequence is not IID.

Consequently, the observer has something to learn, namely, whether the observations are drawn from f_0 or from f_1.

The decision maker wants to decide which of the two distributions is generating outcomes.

26.4.1 Type I and type II errors

If we regard $f = f_0$ as a null hypothesis and $f = f_1$ as an alternative hypothesis, then

- a type I error is an incorrect rejection of a true null hypothesis (a "false positive")
- a type II error is a failure to reject a false null hypothesis (a "false negative")

To repeat ourselves

- α is the probability of a type I error
- β is the probability of a type II error

26.4.2 Choices

After observing $z_k, z_{k-1}, \ldots, z_1$, the decision-maker chooses among three distinct actions:

- He decides that $f = f_0$ and draws no more z's
- He decides that $f = f_1$ and draws no more z's
- He postpones deciding and instead chooses to draw z_{k+1}

Wald defines

- $p_{0m} = f_0(z_1) \cdots f_0(z_m)$
- $p_{1m} = f_1(z_1) \cdots f_1(z_m)$
- $L_m = \frac{p_{1m}}{p_{0m}}$

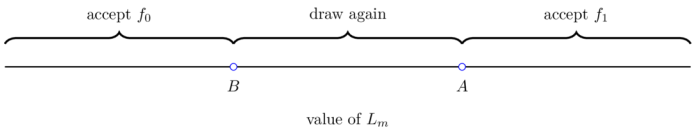
Here $\{L_m\}_{m=0}^{\infty}$ is a **likelihood ratio process**.

Wald's sequential decision rule is parameterized by real numbers $B < A$.

For a given pair A, B, the decision rule is

$$\begin{aligned} \text{accept } f = f_1 \text{ if } L_m &\geq A \\ \text{accept } f = f_0 \text{ if } L_m &\leq B \\ \text{draw another } z \text{ if } B < L_m &< A \end{aligned}$$

The following figure illustrates aspects of Wald's procedure.



26.5 Links between A, B and α, β

In chapter 3 of **Sequential Analysis** [Wald, 1947] Wald establishes the inequalities

$$\begin{aligned} \frac{\alpha}{1-\beta} &\leq \frac{1}{A} \\ \frac{\beta}{1-\alpha} &\leq B \end{aligned}$$

His analysis of these inequalities leads Wald to recommend the following approximations as rules for setting A and B that come close to attaining a decision maker's target values for probabilities α of a type I and β of a type II error:

$$
\begin{aligned}
A \approx a(\alpha, \beta) &\equiv \frac{1-\beta}{\alpha} \\
B \approx b(\alpha, \beta) &\equiv \frac{\beta}{1-\alpha}
\end{aligned}
\tag{26.1}
$$

For small values of α and β, Wald shows that approximation (26.1) provides a good way to set A and B.

In particular, Wald constructs a mathematical argument that leads him to conclude that the use of approximation (26.1) rather than the true functions $A(\alpha, \beta), B(\alpha, \beta)$ for setting A and B

> ... cannot result in any appreciable increase in the value of either α or β. In other words, for all practical purposes the test corresponding to $A = a(\alpha, \beta), B = b(\alpha, \beta)$ provides as least the same protection against wrong decisions as the test corresponding to $A = A(\alpha, \beta)$ and $B = b(\alpha, \beta)$.
>
> Thus, the only disadvantage that may arise from using $a(\alpha, \beta), b(\alpha, \beta)$ instead of $A(\alpha, \beta), B(\alpha, \beta)$, respectively, is that it may result in an appreciable increase in the number of observations required by the test.

We'll write some Python code to help us illustrate Wald's claims about how α and β are related to the parameters A and B that characterize his sequential probability ratio test.

26.6 Simulations

We experiment with different distributions f_0 and f_1 to examine how Wald's test performs under various conditions.

Our goal in conducting these simulations is to understand trade-offs between decision speed and accuracy associated with Wald's **sequential probability ratio test**.

Specifically, we will watch how:

- The decision thresholds A and B (or equivalently the target error rates α and β) affect the average stopping time
- The discrepancy between distributions f_0 and f_1 affects average stopping times

We will focus on the case where f_0 and f_1 are beta distributions since it is easy to control the overlapping regions of the two densities by adjusting their shape parameters.

First, we define a namedtuple to store all the parameters we need for our simulation studies.

We also compute Wald's recommended thresholds A and B based on the target type I and type II errors α and β

```
SPRTParams = namedtuple('SPRTParams',
                ['α', 'β',  # Target type I and type II errors
                'a0', 'b0', # Shape parameters for f_0
                'a1', 'b1', # Shape parameters for f_1
                'N',        # Number of simulations
                'seed'])

@njit
def compute_wald_thresholds(α, β):
    """Compute Wald's recommended thresholds."""
    A = (1 - β) / α
    B = β / (1 - α)
    return A, B, np.log(A), np.log(B)
```

Now we can run the simulation following Wald's recommendation.

We'll compare the log-likelihood ratio to logarithms of the thresholds $\log(A)$ and $\log(B)$.

The following algorithm underlies our simulations.

1. Compute thresholds $A = \frac{1-\beta}{\alpha}$, $B = \frac{\beta}{1-\alpha}$ and work with $\log A$, $\log B$.
2. Given true distribution (either f_0 or f_1):
   - Initialize log-likelihood ratio $\log L_0 = 0$
   - Repeat:
     - Draw observation z from the true distribution
     - Update: $\log L_{n+1} \leftarrow \log L_n + (\log f_1(z) - \log f_0(z))$
     - If $\log L_{n+1} \geq \log A$: stop, reject H_0
     - If $\log L_{n+1} \leq \log B$: stop, accept H_0
3. Repeat step 2 for N replications with $N/2$ replications for each distribution, compute the empirical type I error $\hat{\alpha}$ and type II error $\hat{\beta}$ with

$$\hat{\alpha} = \frac{\#\text{ of times reject } H_0 \text{ when } f_0 \text{ is true}}{\#\text{ of replications with } f_0 \text{ true}}$$

$$\hat{\beta} = \frac{\#\text{ of times accept } H_0 \text{ when } f_1 \text{ is true}}{\#\text{ of replications with } f_1 \text{ true}}$$

```
@njit
def sprt_single_run(a0, b0, a1, b1, logA, logB, true_f0, seed):
    """Run a single SPRT until a decision is reached."""
    log_L = 0.0
    n = 0
    np.random.seed(seed)

    while True:
        z = np.random.beta(a0, b0) if true_f0 else np.random.beta(a1, b1)
        n += 1

        # Update log-likelihood ratio
        log_L += np.log(p(z, a1, b1)) - np.log(p(z, a0, b0))

        # Check stopping conditions
        if log_L >= logA:
            return n, False  # Reject H0
        elif log_L <= logB:
            return n, True   # Accept H0

@njit(parallel=True)
def run_sprt_simulation(a0, b0, a1, b1, α, β, N, seed):
    """SPRT simulation."""
    A, B, logA, logB = compute_wald_thresholds(α, β)

    stopping_times = np.zeros(N, dtype=np.int64)
    decisions_h0 = np.zeros(N, dtype=np.bool_)
    truth_h0 = np.zeros(N, dtype=np.bool_)

    for i in prange(N):
        true_f0 = (i % 2 == 0)
        truth_h0[i] = true_f0
```

(continues on next page)

(continued from previous page)

```
        n, accept_f0 = sprt_single_run(
                        a0, b0, a1, b1,
                        logA, logB,
                        true_f0, seed + i)
        stopping_times[i] = n
        decisions_h0[i] = accept_f0

    return stopping_times, decisions_h0, truth_h0

def run_sprt(params):
    """Run SPRT simulations with given parameters."""
    stopping_times, decisions_h0, truth_h0 = run_sprt_simulation(
        params.a0, params.b0, params.a1, params.b1,
        params.α, params.β, params.N, params.seed
    )

    # Calculate error rates
    truth_h0_bool = truth_h0.astype(bool)
    decisions_h0_bool = decisions_h0.astype(bool)

    type_I = np.sum(truth_h0_bool & ~decisions_h0_bool) \
             / np.sum(truth_h0_bool)
    type_II = np.sum(~truth_h0_bool & decisions_h0_bool) \
              / np.sum(~truth_h0_bool)

    return {
        'stopping_times': stopping_times,
        'decisions_h0': decisions_h0_bool,
        'truth_h0': truth_h0_bool,
        'type_I': type_I,
        'type_II': type_II
    }

# Run simulation
params = SPRTParams(α=0.05, β=0.10, a0=2, b0=5, a1=5, b1=2, N=20000, seed=1)
results = run_sprt(params)

print(f"Average stopping time: {results['stopping_times'].mean():.2f}")
print(f"Empirical type I  error: {results['type_I']:.3f} (target = {params.α})")
print(f"Empirical type II error: {results['type_II']:.3f} (target = {params.β})")
```

```
Average stopping time: 1.59
Empirical type I  error: 0.012 (target = 0.05)
Empirical type II error: 0.022 (target = 0.1)
```

As anticipated in the passage above in which Wald discussed the quality of $a(\alpha, \beta), b(\alpha, \beta)$ given in approximation (26.1), we find that the algorithm actually gives **lower** type I and type II error rates than the target values.

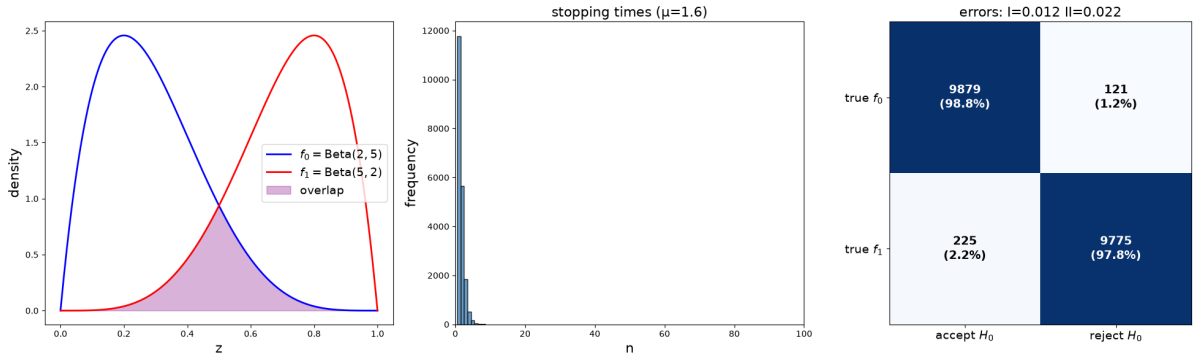
Note

For recent work on the quality of approximation (26.1), see, e.g., [Fischer and Ramdas, 2024].

The following code creates a few graphs that illustrate the results of our simulation.

Let's plot the results of our simulation

```
plot_sprt_results(results, params)
```



In this example, the stopping time stays below 10.

We can construct a 2×2 "confusion matrix" whose diagonal elements count the number of times that Wald's decision rule correctly accepts and rejects the null hypothesis.

```
print("Confusion Matrix data:")
print(f"Type I error: {results['type_I']:.3f}")
print(f"Type II error: {results['type_II']:.3f}")
```

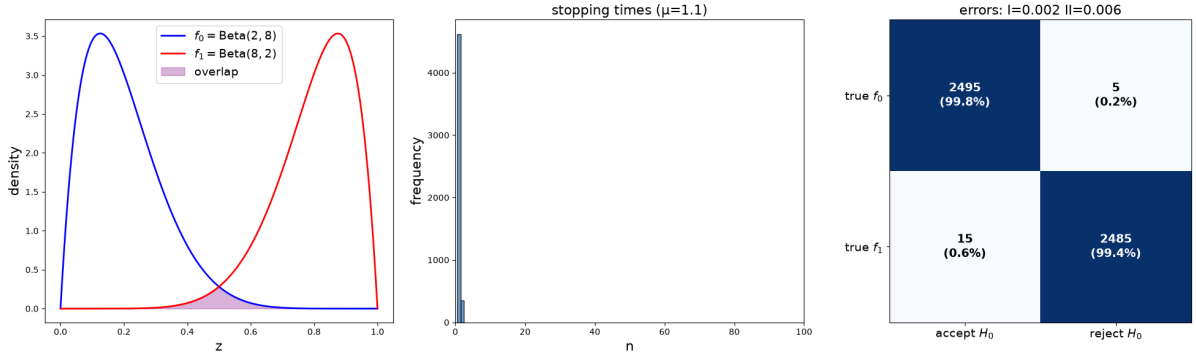
```
Confusion Matrix data:
Type I error: 0.012
Type II error: 0.022
```

Next we use our code to study three different f_0, f_1 pairs having different discrepancies between distributions.

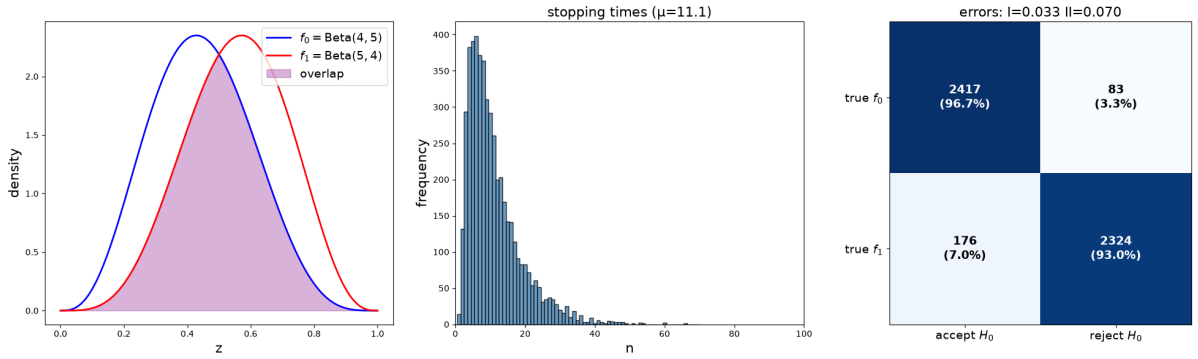
We plot the same three graphs we used above for each pair of distributions

```
params_1 = SPRTParams(α=0.05, β=0.10, a0=2, b0=8, a1=8, b1=2, N=5000, seed=42)
results_1 = run_sprt(params_1)

params_2 = SPRTParams(α=0.05, β=0.10, a0=4, b0=5, a1=5, b1=4, N=5000, seed=42)
results_2 = run_sprt(params_2)

params_3 = SPRTParams(α=0.05, β=0.10, a0=0.5, b0=0.4, a1=0.4,
                      b1=0.5, N=5000, seed=42)
results_3 = run_sprt(params_3)
```

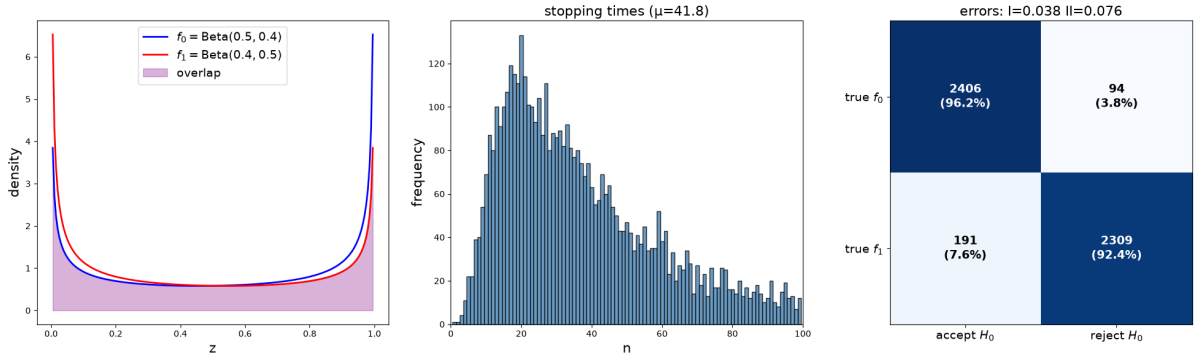
```
plot_sprt_results(results_1, params_1)
```



```
plot_sprt_results(results_2, params_2)
```




```
plot_sprt_results(results_3, params_3)
```



Notice that the stopping times are less when the two distributions are farther apart.

This makes sense.

When two distributions are "far apart", it should not take too long to decide which one is generating the data.

When two distributions are "close", it should takes longer to decide which one is generating the data.

It is tempting to link this pattern to our discussion of *Kullback–Leibler divergence* in *Likelihood Ratio Processes*.

While, KL divergence is larger when two distributions differ more, KL divergence is not symmetric, meaning that the KL divergence of distribution f from distribution g is not necessarily equal to the KL divergence of g from f.

If we want a symmetric measure of divergence that actually a metric, we can instead use Jensen-Shannon distance.

That is what we shall do now.

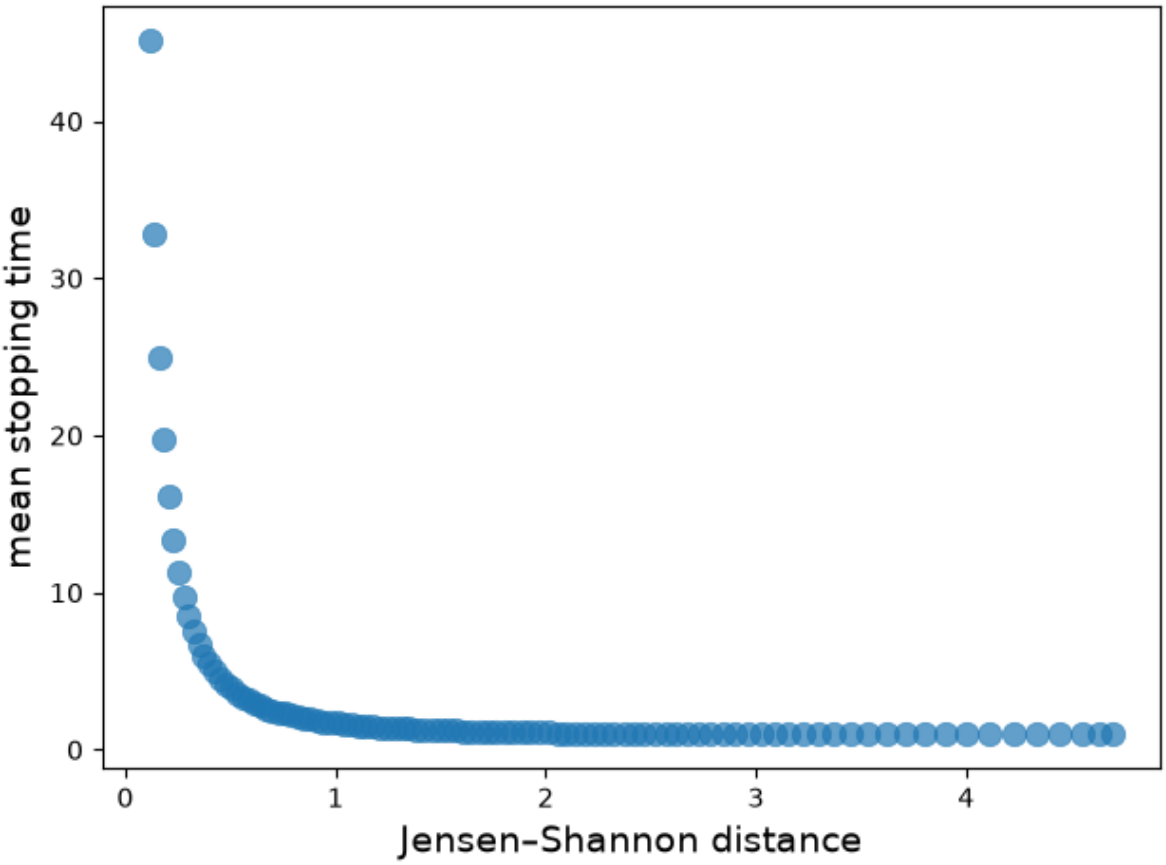
We shall compute Jensen-Shannon distance and plot it against the average stopping times.

```
def js_dist(a0, b0, a1, b1):
    """Jensen-Shannon distance"""
    f0 = create_beta_density(a0, b0)
    f1 = create_beta_density(a1, b1)

    # Mixture
    m = lambda w: 0.5*(f0(w) + f1(w))
    return np.sqrt(0.5*compute_KL(m, f0) + 0.5*compute_KL(m, f1))

def generate_β_pairs(N=100, T=10.0, d_min=0.5, d_max=9.5):
    ds = np.linspace(d_min, d_max, N)
```

(continues on next page)

(continued from previous page)

```
    a0 = (T - ds) / 2
    b0 = (T + ds) / 2
    return list(zip(a0, b0, b0, a0))

param_comb = generate_β_pairs()

# Run simulations for each parameter combination
js_dists = []
mean_stopping_times = []
param_list = []

for a0, b0, a1, b1 in param_comb:
    # Compute KL divergence
    js_div = js_dist(a1, b1, a0, b0)

    # Run SPRT simulation with a fixed set of parameters d d
    params = SPRTParams(α=0.05, β=0.10, a0=a0, b0=b0,
                        a1=a1, b1=b1, N=5000, seed=42)
    results = run_sprt(params)

    js_dists.append(js_div)
    mean_stopping_times.append(results['stopping_times'].mean())
    param_list.append((a0, b0, a1, b1))

# Create the plot
fig, ax = plt.subplots()

scatter = ax.scatter(js_dists, mean_stopping_times,
                     s=80, alpha=0.7, linewidth=0.5)

ax.set_xlabel('Jensen-Shannon distance', fontsize=14)
ax.set_ylabel('mean stopping time', fontsize=14)

plt.tight_layout()
plt.show()
```



The plot demonstrates a clear negative correlation between relative entropy and mean stopping time.

As Jensen-Shannon divergence increases (distributions become more separated), the mean stopping time decreases exponentially.

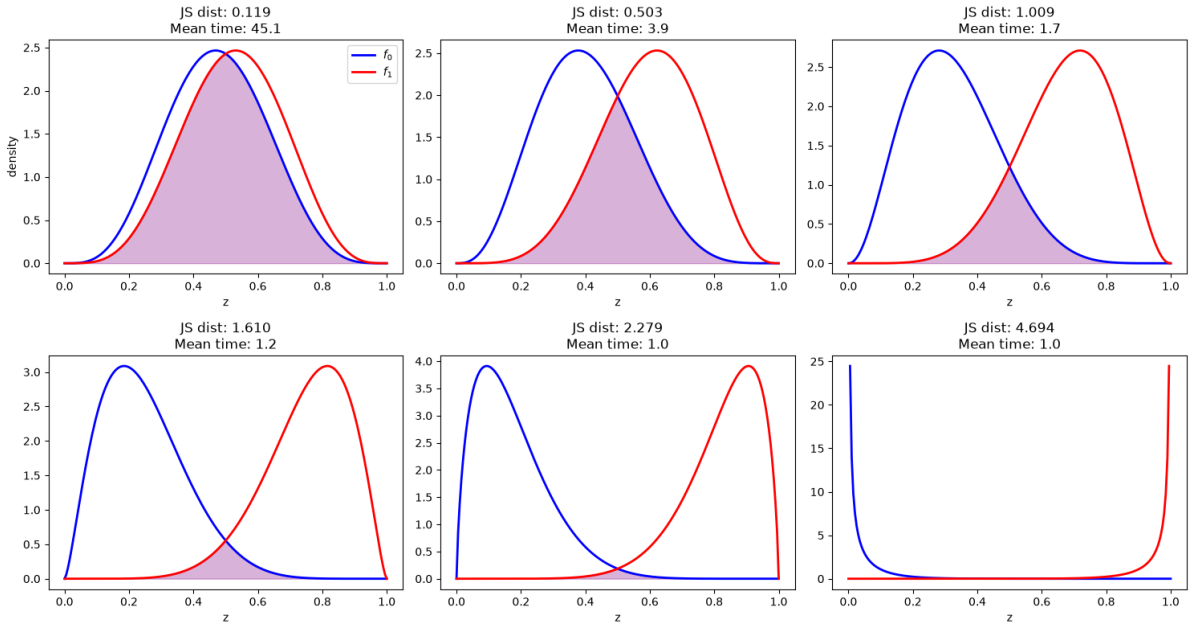
Below are sampled examples from the experiments we have above

```
def plot_beta_distributions_grid(param_list, js_dists, mean_stopping_times,
                                 selected_indices=None):
    """Plot grid of beta distributions with JS distance and stopping times."""
    if selected_indices is None:
        selected_indices = [0, len(param_list)//6, len(param_list)//3,
                            len(param_list)//2, 2*len(param_list)//3, -1]

    fig, axes = plt.subplots(2, 3, figsize=(15, 8))
    z_grid = np.linspace(0, 1, 200)

    for i, idx in enumerate(selected_indices):
        row, col = i // 3, i % 3
        a0, b0, a1, b1 = param_list[idx]

        f0 = create_beta_density(a0, b0)
        f1 = create_beta_density(a1, b1)

        axes[row, col].plot(z_grid, f0(z_grid), 'b-', lw=2, label='$f_0$')
        axes[row, col].plot(z_grid, f1(z_grid), 'r-', lw=2, label='$f_1$')
        axes[row, col].fill_between(z_grid, 0,
```

(continues on next page)

(continued from previous page)

```
                                np.minimum(f0(z_grid), f1(z_grid)),
                                alpha=0.3, color='purple')

        axes[row, col].set_title(f'JS dist: {js_dists[idx]:.3f}'
                                 f'\nMean time: {mean_stopping_times[idx]:.1f}',
                                 fontsize=12)
        axes[row, col].set_xlabel('z', fontsize=10)
        if i == 0:
            axes[row, col].set_ylabel('density', fontsize=10)
            axes[row, col].legend(fontsize=10)

    plt.tight_layout()
    plt.show()

plot_beta_distributions_grid(param_list, js_dists, mean_stopping_times)
```



Again, we find that the stopping time is shorter when the distributions are more separated, as measured by Jensen-Shannon distance.

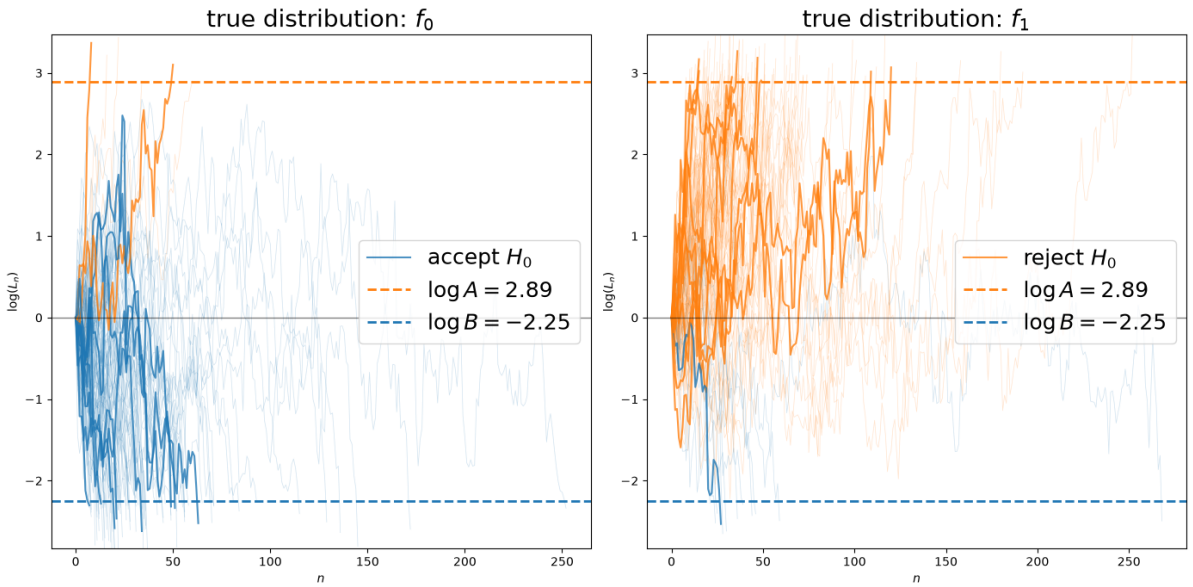
Let's visualize individual likelihood ratio processes to see how they evolve toward the decision boundaries.

```
def plot_likelihood_paths(params, n_highlight=10, n_background=200):
    """visualize likelihood ratio paths."""
    A, B, logA, logB = compute_wald_thresholds(params.α, params.β)
    f0, f1 = map(lambda ab: create_beta_density(*ab),
                 [(params.a0, params.b0),
                  (params.a1, params.b1)])

    fig, axes = plt.subplots(1, 2, figsize=(14, 7))

    for dist_idx, (true_f0, ax, title) in enumerate([
        (True, axes[0], 'true distribution: $f_0$'),
        (False, axes[1], 'true distribution: $f_1$')
    ]):
        rng = np.random.default_rng(seed=42 + dist_idx)
```

(continues on next page)

(continued from previous page)

```
        paths_data = []

        # Generate paths
        for path in range(n_background + n_highlight):
            log_L_path, log_L, n = [0.0], 0.0, 0

            while True:
                z = rng.beta(params.a0, params.b0) if true_f0 \
                    else rng.beta(params.a1, params.b1)
                n += 1
                log_L += np.log(f1(z)) - np.log(f0(z))
                log_L_path.append(log_L)

                if log_L >= logA or log_L <= logB:
                    paths_data.append((log_L_path, n, log_L >= logA))
                    break

        # Plot background paths
        for path, _, decision in paths_data[:n_background]:
            ax.plot(range(len(path)), path, color='C1' if decision else 'C0',
                   alpha=0.2, linewidth=0.5)

        # Plot highlighted paths with labels
        for i, (path, _, decision) in enumerate(paths_data[n_background:]):
            ax.plot(range(len(path)), path, color='C1' if decision else 'C0',
                   alpha=0.8, linewidth=1.5,
                   label='reject $H_0$' if decision and i == 0 else (
                         'accept $H_0$' if not decision and i == 0 else ''))

        # Add threshold lines and formatting
        ax.axhline(y=logA, color='C1', linestyle='--', linewidth=2,
                  label=f'$\\log A = {logA:.2f}$')
        ax.axhline(y=logB, color='C0', linestyle='--', linewidth=2,
                  label=f'$\\log B = {logB:.2f}$')
        ax.axhline(y=0, color='black', linestyle='-', alpha=0.5, linewidth=1)

        ax.set_xlabel(r'$n$')
        ax.set_ylabel(r'$\log(L_n)$')
        ax.set_title(title, fontsize=20)
        ax.legend(fontsize=18, loc='center right')

        y_margin = max(abs(logA), abs(logB)) * 0.2
        ax.set_ylim(logB - y_margin, logA + y_margin)

    plt.tight_layout()
    plt.show()

plot_likelihood_paths(params_3, n_highlight=10, n_background=100)
```



Next, let's adjust the decision thresholds A and B and examine how the mean stopping time and the type I and type II error rates change.

In the code below, we adjust Wald's rule by adjusting the thresholds A and B using factors A_f and B_f.

```
@njit(parallel=True)
def run_adjusted_thresholds(a0, b0, a1, b1, α, β, N, seed, A_f, B_f):
    """SPRT simulation with adjusted thresholds."""

    # Calculate original thresholds
    A_original = (1 - β) / α
    B_original = β / (1 - α)

    # Apply adjustment factors
    A_adj = A_original * A_f
    B_adj = B_original * B_f
    logA = np.log(A_adj)
    logB = np.log(B_adj)

    # Pre-allocate arrays
    stopping_times = np.zeros(N, dtype=np.int64)
    decisions_h0 = np.zeros(N, dtype=np.bool_)
    truth_h0 = np.zeros(N, dtype=np.bool_)

    # Run simulations in parallel
    for i in prange(N):
        true_f0 = (i % 2 == 0)
        truth_h0[i] = true_f0

        n, accept_f0 = sprt_single_run(a0, b0, a1, b1,
                        logA, logB, true_f0, seed + i)
        stopping_times[i] = n
        decisions_h0[i] = accept_f0

    return stopping_times, decisions_h0, truth_h0, A_adj, B_adj

def run_adjusted(params, A_f=1.0, B_f=1.0):
```

(continues on next page)

(continued from previous page)

```
    """Wrapper to run SPRT with adjusted A and B thresholds."""

    stopping_times, decisions_h0, truth_h0, A_adj, B_adj = run_adjusted_thresholds(
        params.a0, params.b0, params.a1, params.b1,
        params.α, params.β, params.N, params.seed, A_f, B_f
    )
    truth_h0_bool = truth_h0.astype(bool)
    decisions_h0_bool = decisions_h0.astype(bool)

    # Calculate error rates
    type_I = np.sum(truth_h0_bool
                    & ~decisions_h0_bool) / np.sum(truth_h0_bool)
    type_II = np.sum(~truth_h0_bool
                     & decisions_h0_bool) / np.sum(~truth_h0_bool)

    return {
        'stopping_times': stopping_times,
        'type_I': type_I,
        'type_II': type_II,
        'A_used': A_adj,
        'B_used': B_adj
    }

adjustments = [
    (5.0, 0.5),
    (1.0, 1.0),
    (0.3, 3.0),
    (0.2, 5.0),
    (0.15, 7.0),
]

results_table = []
for A_f, B_f in adjustments:
    result = run_adjusted(params_2, A_f, B_f)
    results_table.append([
        A_f, B_f,
        f"{result['stopping_times'].mean():.1f}",
        f"{result['type_I']:.3f}",
        f"{result['type_II']:.3f}"
    ])

df = pd.DataFrame(results_table,
                  columns=["A_f", "B_f", "mean stop time",
                           "Type I error", "Type II error"])
df = df.set_index(["A_f", "B_f"])
df
```

A_f	B_f	mean stop time	Type I error	Type II error
5.00	0.5	16.1	0.006	0.036
1.00	1.0	11.1	0.033	0.070
0.30	3.0	5.5	0.086	0.195
0.20	5.0	3.4	0.120	0.304
0.15	7.0	2.2	0.146	0.410

Let's pause and think about the table more carefully by referring back to (26.1).

Recall that $A = \frac{1-\beta}{\alpha}$ and $B = \frac{\beta}{1-\alpha}$.

When we multiply A by a factor less than 1 (making A smaller), we are effectively making it easier to reject the null hypothesis H_0.

This increases the probability of Type I errors.

When we multiply B by a factor greater than 1 (making B larger), we are making it easier to accept the null hypothesis H_0.

This increases the probability of Type II errors.

The table confirms this intuition: as A decreases and B increases from their optimal Wald values, both Type I and Type II error rates increase, while the mean stopping time decreases.

26.7 Related lectures

We'll dig deeper into some of the ideas used here in the following earlier and later lectures:

- In *this sequel*, we reformulate the problem from the perspective of a **Bayesian statistician** who views parameters as vectors of random variables that are jointly distributed with the observables they are concerned about.
- The concept of **exchangeability**, which underlies much of statistical learning, is explored in depth in our *lecture on exchangeable random variables*.
- For a deeper understanding of likelihood ratio processes and their role in frequentist and Bayesian statistical theories, see *Likelihood Ratio Processes*.
- Building on that foundation, *Likelihood Ratio Processes and Bayesian Learning* examines the role of likelihood ratio processes in **Bayesian learning**.
- Finally, *this later lecture* revisits the subject discussed here and examines whether the frequentist decision rule that the Navy ordered the captain to use would perform better or worse than Abraham Wald's sequential decision rule.

26.8 Exercises

In the two exercises below, please try to rewrite the entire SPRT suite in this lecture.

Exercise 26.8.1

In the first exercise, we apply the sequential probability ratio test to distinguish two models generated by 3-state Markov chains

(For a review on likelihood ratio processes for Markov chains, see *this section*.)

Consider distinguishing between two 3-state Markov chain models using Wald's sequential probability ratio test.

You have competing hypotheses about the transition probabilities:

- H_0: The chain follows transition matrix $P^{(0)}$
- H_1: The chain follows transition matrix $P^{(1)}$

Given transition matrices:

$$P^{(0)} = \begin{bmatrix} 0.7 & 0.2 & 0.1 \\ 0.3 & 0.5 & 0.2 \\ 0.1 & 0.3 & 0.6 \end{bmatrix}, \quad P^{(1)} = \begin{bmatrix} 0.5 & 0.3 & 0.2 \\ 0.2 & 0.6 & 0.2 \\ 0.2 & 0.2 & 0.6 \end{bmatrix}$$

For a sequence of observations $(x_0, x_1, \ldots, x_t)$, the likelihood ratio is:

$$\Lambda_t = \frac{\pi^{(1)}_{x_0}}{\pi^{(0)}_{x_0}} \prod_{s=1}^{t} \frac{P^{(1)}_{x_{s-1},x_s}}{P^{(0)}_{x_{s-1},x_s}}$$

where $\pi^{(i)}$ is the stationary distribution under hypothesis i.

Tasks:

1. Implement the likelihood ratio computation for Markov chains
2. Implement Wald's sequential test with Type I error $\alpha = 0.05$ and Type II error $\beta = 0.10$
3. Run 1000 simulations under each hypothesis and compute empirical error rates
4. Analyze the distribution of stopping times

The test stops when:

- $\Lambda_t \geq A = \frac{1-\beta}{\alpha} = 18$: Reject H_0
- $\Lambda_t \leq B = \frac{\beta}{1-\alpha} = 0.105$: Accept H_0

Solution

Below is one solution to the exercise.

In the lecture, we write the code more verbosely to illustrate the concepts clearly.

In the code below, we simplified some of the code structure for a shorter presentation.

First we define the parameters for the Markov chain SPRT

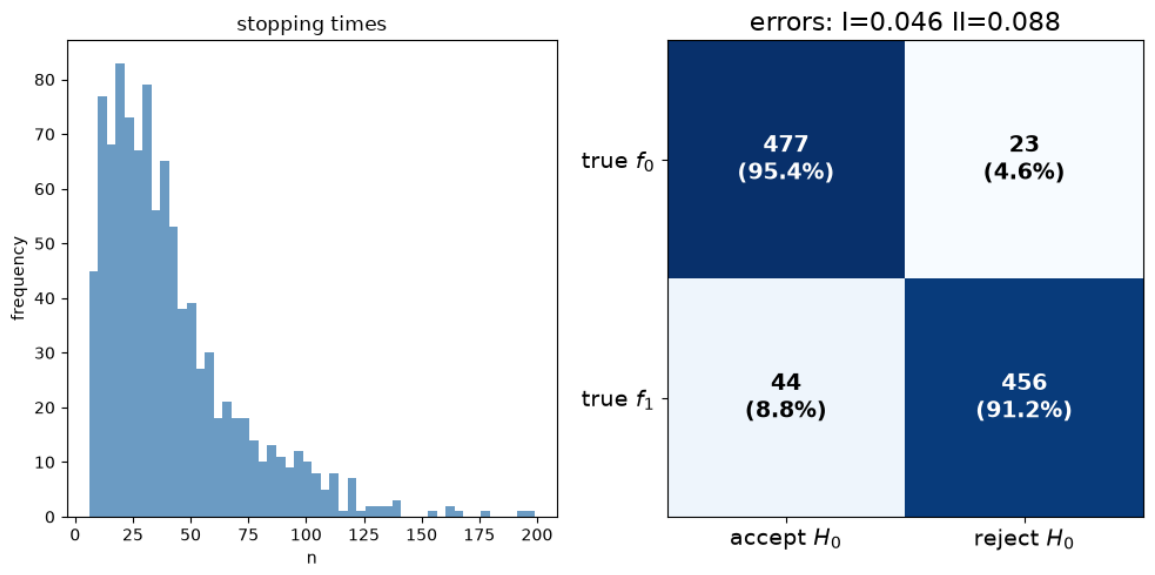
```
MarkovSPRTParams = namedtuple('MarkovSPRTParams',
            ['α', 'β', 'P_0', 'P_1', 'N', 'seed'])

def compute_stationary_distribution(P):
    """Compute stationary distribution of transition matrix P."""
    eigenvalues, eigenvectors = np.linalg.eig(P.T)
    idx = np.argmin(np.abs(eigenvalues - 1))
    pi = np.real(eigenvectors[:, idx])
    return pi / pi.sum()

@njit
def simulate_markov_chain(P, pi_0, T, seed):
    """Simulate a Markov chain path."""
    np.random.seed(seed)
    path = np.zeros(T, dtype=np.int32)

    cumsum_pi = np.cumsum(pi_0)
    path[0] = np.searchsorted(cumsum_pi, np.random.uniform())

    for t in range(1, T):
        cumsum_row = np.cumsum(P[path[t-1]])
        path[t] = np.searchsorted(cumsum_row, np.random.uniform())

    return path
```

Here we define the function that runs SPRT for Markov chains

```
@njit
def markov_sprt_single_run(P_0, P_1, π_0, π_1,
                           logA, logB, true_P, true_π, seed):
    """Run single SPRT for Markov chains."""
    max_n = 10000
    path = simulate_markov_chain(true_P, true_π, max_n, seed)

    log_L = np.log(π_1[path[0]] / π_0[path[0]])
    if log_L >= logA: return 1, False
    if log_L <= logB: return 1, True

    for t in range(1, max_n):
        prev_state, curr_state = path[t-1], path[t]
        p_1, p_0 = P_1[prev_state, curr_state], P_0[prev_state, curr_state]

        if p_0 > 0:
            log_L += np.log(p_1 / p_0)
        elif p_1 > 0:
            log_L = np.inf

        if log_L >= logA: return t+1, False
        if log_L <= logB: return t+1, True

    return max_n, log_L < 0

def run_markov_sprt(params):
    """Run SPRT for Markov chains."""
    π_0 = compute_stationary_distribution(params.P_0)
    π_1 = compute_stationary_distribution(params.P_1)
    A, B, logA, logB = compute_wald_thresholds(params.α, params.β)

    stopping_times = np.zeros(params.N, dtype=np.int64)
    decisions_h0 = np.zeros(params.N, dtype=bool)
    truth_h0 = np.zeros(params.N, dtype=bool)

    for i in range(params.N):
        true_P, true_π = (params.P_0, π_0) if i % 2 == 0 else (params.P_1, π_1)
        truth_h0[i] = i % 2 == 0

        n, accept_h0 = markov_sprt_single_run(
            params.P_0, params.P_1, π_0, π_1, logA, logB,
            true_P, true_π, params.seed + i)

        stopping_times[i] = n
        decisions_h0[i] = accept_h0

    type_I = np.sum(truth_h0 & ~decisions_h0) / np.sum(truth_h0)
    type_II = np.sum(~truth_h0 & decisions_h0) / np.sum(~truth_h0)

    return {
        'stopping_times': stopping_times, 'decisions_h0': decisions_h0,
        'truth_h0': truth_h0, 'type_I': type_I, 'type_II': type_II
    }
```

Now we can run the SPRT for the Markov chain models and visualize the results

```
# Run Markov chain SPRT
P_0 = np.array([[0.7, 0.2, 0.1],
                [0.3, 0.5, 0.2],
                [0.1, 0.3, 0.6]])

P_1 = np.array([[0.5, 0.3, 0.2],
                [0.2, 0.6, 0.2],
                [0.2, 0.2, 0.6]])

params_markov = MarkovSPRTParams(α=0.05, β=0.10,
                         P_0=P_0, P_1=P_1, N=1000, seed=42)
results_markov = run_markov_sprt(params_markov)


fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 5))

ax1.hist(results_markov['stopping_times'],
            bins=50, color="steelblue", alpha=0.8)
ax1.set_title("stopping times")
ax1.set_xlabel("n")
ax1.set_ylabel("frequency")

plot_confusion_matrix(results_markov, ax2)

plt.tight_layout()
plt.show()
```



Exercise 26.8.2

In this exercise, apply Wald's sequential test to distinguish between two VAR(1) models with different dynamics and noise structures.

For a review of the likelihood ratio process with VAR models, see *Likelihood Processes For VAR Models*.

Given VAR models under each hypothesis:

- H_0: $x_{t+1} = A^{(0)} x_t + C^{(0)} w_{t+1}$
- H_1: $x_{t+1} = A^{(1)} x_t + C^{(1)} w_{t+1}$

where $w_t \sim \mathcal{N}(0, I)$ and:

$$A^{(0)} = \begin{bmatrix} 0.8 & 0.1 \\ 0.2 & 0.7 \end{bmatrix}, \quad C^{(0)} = \begin{bmatrix} 0.3 & 0.1 \\ 0.1 & 0.3 \end{bmatrix}$$

$$A^{(1)} = \begin{bmatrix} 0.6 & 0.2 \\ 0.3 & 0.5 \end{bmatrix}, \quad C^{(1)} = \begin{bmatrix} 0.4 & 0 \\ 0 & 0.4 \end{bmatrix}$$

Tasks:

1. Implement the VAR likelihood ratio using the functions from the VAR lecture
2. Implement Wald's sequential test with $\alpha = 0.05$ and $\beta = 0.10$
3. Analyze performance under both hypotheses and with model misspecification
4. Compare with the Markov chain case in terms of stopping times and accuracy

Solution

Below is one solution to the exercise.

First we define the parameters for the VAR models and simulator

```
VARSPRTParams = namedtuple('VARSPRTParams',
            ['α', 'β', 'A_0', 'C_0', 'A_1', 'C_1', 'N', 'seed'])

def create_var_model(A, C):
    """Create VAR model."""
    μ_0 = np.zeros(A.shape[0])
    CC = C @ C.T
    Σ_0 = sp.linalg.solve_discrete_lyapunov(A, CC)

    CC_inv = np.linalg.inv(CC + 1e-10 * np.eye(CC.shape[0]))
    Σ_0_inv = np.linalg.inv(Σ_0 + 1e-10 * np.eye(Σ_0.shape[0]))

    return {
        'A': A, 'C': C, 'μ_0': μ_0, 'Σ_0': Σ_0,
        'CC_inv': CC_inv, 'Σ_0_inv': Σ_0_inv,
        'log_det_CC': np.log(
            np.linalg.det(CC + 1e-10 * np.eye(CC.shape[0]))),
        'log_det_Σ_0': np.log(
            np.linalg.det(Σ_0 + 1e-10 * np.eye(Σ_0.shape[0])))
    }
```

Now we define the likelihood ratio for the VAR models and the SPRT function similar to the Markov chain case

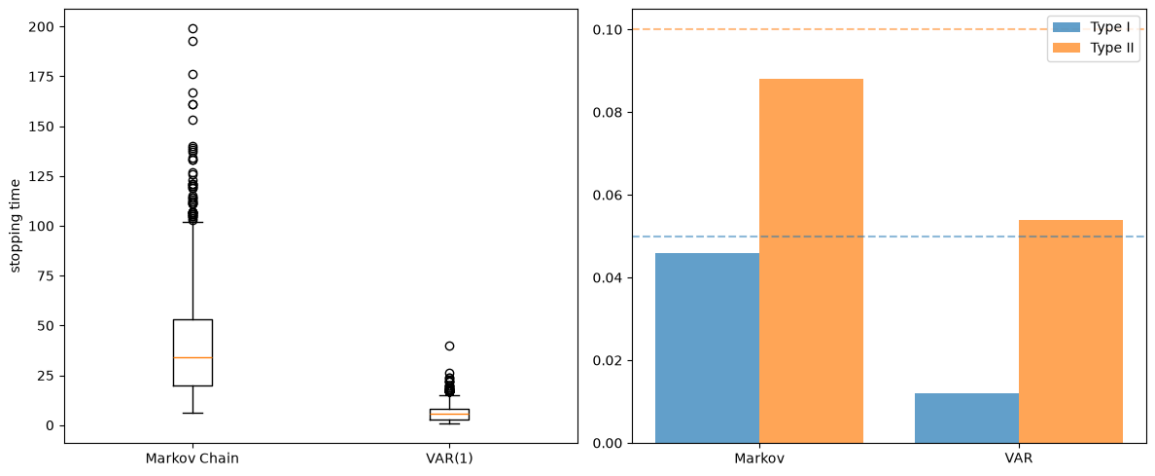
```
def var_log_likelihood(x_curr, x_prev, model, initial=False):
    """Compute VAR log-likelihood."""
    n = len(x_curr)
    if initial:
        diff = x_curr - model['μ_0']
        return -0.5 * (n * np.log(2 * np.pi) + model['log_det_Σ_0'] +
                       diff @ model['Σ_0_inv'] @ diff)
    else:
        diff = x_curr - model['A'] @ x_prev
```

```
        return -0.5 * (n * np.log(2 * np.pi) + model['log_det_CC'] +
                       diff @ model['CC_inv'] @ diff)

def var_sprt_single_run(model_0, model_1, model_true,
                        logA, logB, seed):
    """Single VAR SPRT run."""
    np.random.seed(seed)
    max_T = 500

    # Generate VAR path
    Σ_chol = np.linalg.cholesky(model_true['Σ_0'])
    x = model_true['μ_0'] + Σ_chol @ np.random.randn(
                len(model_true['μ_0']))

    # Initial likelihood ratio
    log_L = (var_log_likelihood(x, None, model_1, True) -
             var_log_likelihood(x, None, model_0, True))

    if log_L >= logA: return 1, False
    if log_L <= logB: return 1, True

    # Sequential updates
    for t in range(1, max_T):
        x_prev = x.copy()
        w = np.random.randn(model_true['C'].shape[1])
        x = model_true['A'] @ x + model_true['C'] @ w

        log_L += (var_log_likelihood(x, x_prev, model_1) -
                  var_log_likelihood(x, x_prev, model_0))

        if log_L >= logA: return t+1, False
        if log_L <= logB: return t+1, True

    return max_T, log_L < 0

def run_var_sprt(params):
    """Run VAR SPRT."""

    model_0 = create_var_model(params.A_0, params.C_0)
    model_1 = create_var_model(params.A_1, params.C_1)
    A, B, logA, logB = compute_wald_thresholds(params.α, params.β)

    stopping_times = np.zeros(params.N)
    decisions_h0 = np.zeros(params.N, dtype=bool)
    truth_h0 = np.zeros(params.N, dtype=bool)

    for i in range(params.N):
        model_true = model_0 if i % 2 == 0 else model_1
        truth_h0[i] = i % 2 == 0

        n, accept_h0 = var_sprt_single_run(model_0, model_1, model_true,
                                           logA, logB, params.seed + i)
        stopping_times[i] = n
        decisions_h0[i] = accept_h0

    type_I = np.sum(truth_h0 & ~decisions_h0) / np.sum(truth_h0)
    type_II = np.sum(~truth_h0 & decisions_h0) / np.sum(~truth_h0)
```

```
    return {'stopping_times': stopping_times,
            'decisions_h0': decisions_h0,
            'truth_h0': truth_h0,
            'type_I': type_I, 'type_II': type_II}
```

Let's run SPRT and visualize the results

```
# Run VAR SPRT
A_0 = np.array([[0.8, 0.1],
                [0.2, 0.7]])
C_0 = np.array([[0.3, 0.1],
                [0.1, 0.3]])
A_1 = np.array([[0.6, 0.2],
                [0.3, 0.5]])
C_1 = np.array([[0.4, 0.0],
                [0.0, 0.4]])

params_var = VARSPRTParams(α=0.05, β=0.10,
                A_0=A_0, C_0=C_0, A_1=A_1, C_1=C_1,
                N=1000, seed=42)
results_var = run_var_sprt(params_var)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))

ax1.boxplot([results_markov['stopping_times'],
             results_var['stopping_times']],
            tick_labels=['Markov Chain', 'VAR(1)'])
ax1.set_ylabel('stopping time')

x = np.arange(2)
ax2.bar(x - 0.2, [results_markov['type_I'], results_var['type_I']],
        0.4, label='Type I', alpha=0.7)
ax2.bar(x + 0.2, [results_markov['type_II'], results_var['type_II']],
        0.4, label='Type II', alpha=0.7)
ax2.axhline(y=0.05, linestyle='--', alpha=0.5, color='C0')
ax2.axhline(y=0.10, linestyle='--', alpha=0.5, color='C1')
ax2.set_xticks(x), ax2.set_xticklabels(['Markov', 'VAR'])
ax2.legend()
plt.tight_layout()
plt.show()
```



CHAPTER

TWENTYSEVEN

A BAYESIAN FORMULATION OF FRIEDMAN AND WALD'S PROBLEM

Contents

- *A Bayesian Formulation of Friedman and Wald's Problem*
  - *Overview*
  - *A Dynamic Programming Approach*
  - *Implementation*
  - *Analysis*

27.1 Overview

This lecture revisits the statistical decision problem presented to Milton Friedman and W. Allen Wallis during World War II when they were analysts at the U.S. Government's Statistical Research Group at Columbia University.

In *an earlier lecture*, we described how Abraham Wald [Wald, 1947] solved the problem by extending frequentist hypothesis testing techniques and formulating the problem sequentially.

Note

Wald's idea of formulating the problem sequentially created links to the **dynamic programming** that Richard Bellman developed in the 1950s.

As we learned in *Elementary Probability with Matrices* and *Two Meanings of Probability*, a frequentist statistician views a probability distribution as measuring relative frequencies of a statistic that he anticipates constructing from a very long sequence of i.i.d. draws from a known probability distribution.

That known probability distribution is his 'hypothesis'.

A frequentist statistician studies the distribution of that statistic under that known probability distribution

- when the distribution is a member of a set of parameterized probability distributions, his hypothesis takes the form of a particular parameter vector.
- this is what we mean when we say that the frequentist statistician 'conditions on the parameters'
- he regards the parameters as fixed numbers that are known to nature, but not to him.

- the statistician copes with his ignorance of those parameters by constructing type I and type II errors associated with frequentist hypothesis testing.

In this lecture, we reformulate Friedman and Wald's problem by transforming our point of view from the 'objective' frequentist perspective of *the lecture on Wald's sequential analysis* to an explicitly 'subjective' perspective taken by a Bayesian decision maker who regards parameters not as fixed numbers but as (hidden) random variables that are jointly distributed with the random variables that he can observe by sampling from that joint distribution.

To form that joint distribution, the Bayesian statistician supplements the conditional distributions used by the frequentist statistician with a prior probability distribution over the parameters that represents his personal, subjective opinion about them.

That lets the Bayesian statistician calculate the joint distribution that he requires to calculate the conditional distributions that he wants.

To proceed in this way, we endow our decision maker with

- an initial prior subjective probability $\pi_{-1} \in (0, 1)$ that nature uses to generate $\{z_k\}$ as a sequence of i.i.d. draws from f_1 rather than f_0.
- faith in Bayes' law as a way to revise his subjective beliefs as observations on $\{z_k\}$ sequence arrive.
- a loss function that tells how the decision maker values type I and type II errors.

In our *previous frequentist version*, key ideas in play were:

- Type I and type II statistical errors
  - a type I error occurs when you reject a null hypothesis that is true
  - a type II error occurs when you accept a null hypothesis that is false
- Abraham Wald's **sequential probability ratio test**
- The **power** of a statistical test
- The **critical region** of a statistical test
- A **uniformly most powerful test**

In this lecture about a Bayesian reformulation of the problem, additional ideas at work are

- an initial prior probability π_{-1} that model f_1 generates the data
- Bayes' Law
- a sequence of posterior probabilities that model f_1 is generating the data
- dynamic programming

This lecture uses ideas studied in the lectures on *likelihood ratio processes*, *their roles in Bayesian learning*, and *this lecture on exchangeability*.

We'll begin with some imports:

```
import numpy as np
import matplotlib.pyplot as plt
from numba import jit, prange, float64, int64
from numba.experimental import jitclass
from math import gamma
```

27.2 A Dynamic Programming Approach

The following presentation of the problem closely follows Dmitri Bertsekas's treatment in **Dynamic Programming and Stochastic Control** [Bertsekas, 1976].

A decision-maker can observe a sequence of draws of a random variable z.

He (or she) wants to know which of two probability distributions f_0 or f_1 governs z.

Conditional on knowing that successive observations are drawn from distribution f_0, the sequence of random variables is independently and identically distributed (IID).

Conditional on knowing that successive observations are drawn from distribution f_1, the sequence of random variables is also independently and identically distributed (IID).

But the observer does not know which of the two distributions generated the sequence.

For reasons explained in Exchangeability and Bayesian Updating, this means that the sequence is not IID.

The observer has something to learn, namely, whether the observations are drawn from f_0 or from f_1.

The decision maker wants to decide which of the two distributions is generating outcomes.

We adopt a Bayesian formulation.

The decision maker begins with a prior probability

$$\pi_{-1} = \mathbb{P}\{f = f_1 \mid \text{ no observations}\} \in (0,1)$$

Note

In Bertsekas [1976], the belief is associated with the distribution f_0, but here we associate the belief with the distribution f_1 to match the discussions in *the lecture on Wald's sequential analysis*.

After observing $k+1$ observations $z_k, z_{k-1}, \ldots, z_0$, he updates his personal probability that the observations are described by distribution f_1 to

$$\pi_k = \mathbb{P}\{f = f_1 \mid z_k, z_{k-1}, \ldots, z_0\}$$

which is calculated recursively by applying Bayes' law:

$$\pi_{k+1} = \frac{\pi_k f_1(z_{k+1})}{(1-\pi_k) f_0(z_{k+1}) + \pi_k f_1(z_{k+1})}, \quad k = -1, 0, 1, \ldots$$

After observing $z_k, z_{k-1}, \ldots, z_0$, the decision-maker believes that z_{k+1} has probability distribution

$$f_{\pi_k}(v) = (1-\pi_k) f_0(v) + \pi_k f_1(v),$$

which is a mixture of distributions f_0 and f_1, with the weight on f_1 being the posterior probability that $f = f_1$[1].

To illustrate such a distribution, let's inspect some mixtures of beta distributions.

The density of a beta probability distribution with parameters a and b is

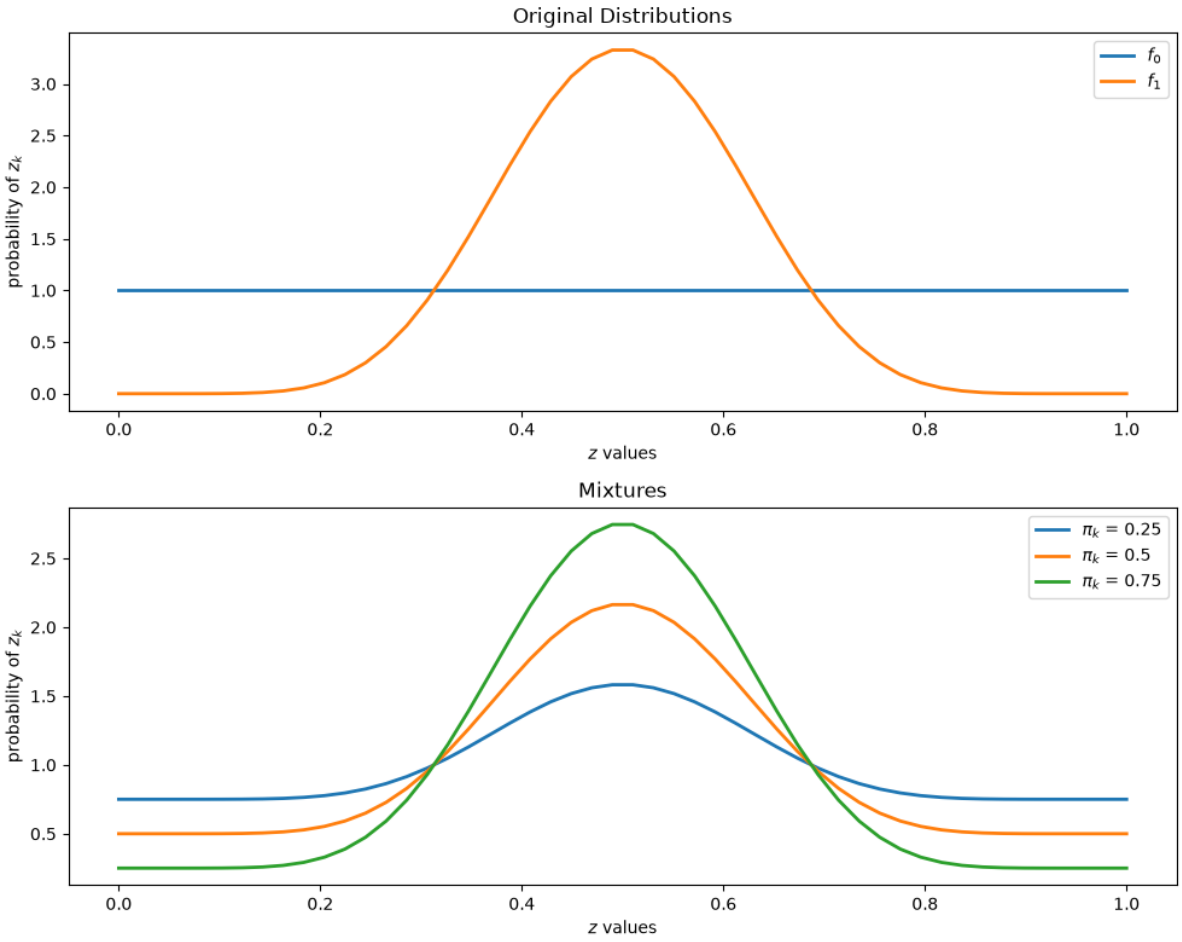
$$f(z; a, b) = \frac{\Gamma(a+b) z^{a-1} (1-z)^{b-1}}{\Gamma(a)\Gamma(b)} \quad \text{where} \quad \Gamma(t) := \int_0^\infty x^{t-1} e^{-x} dx$$

The next figure shows two beta distributions in the top panel.

The bottom panel presents mixtures of these distributions, with various mixing probabilities π_k

[1] The decision maker acts as if he believes that the sequence of random variables $[z_0, z_1, \ldots]$ is *exchangeable*. See Exchangeability and Bayesian Updating and [Kreps, 1988] chapter 11, for discussions of exchangeability.

```
@jit
def p(x, a, b):
    r = gamma(a + b) / (gamma(a) * gamma(b))
    return r * x**(a-1) * (1 - x)**(b-1)

f0 = lambda x: p(x, 1, 1)
f1 = lambda x: p(x, 9, 9)
grid = np.linspace(0, 1, 50)

fig, axes = plt.subplots(2, figsize=(10, 8))

axes[0].set_title("Original Distributions")
axes[0].plot(grid, f0(grid), lw=2, label="$f_0$")
axes[0].plot(grid, f1(grid), lw=2, label="$f_1$")

axes[1].set_title("Mixtures")
for π in 0.25, 0.5, 0.75:
    y = (1 - π) * f0(grid) + π * f1(grid)
    axes[1].plot(grid, y, lw=2, label=fr"$\pi_k$ = {π}")

for ax in axes:
    ax.legend()
    ax.set(xlabel="$z$ values", ylabel="probability of $z_k$")

plt.tight_layout()
plt.show()
```



27.2.1 Losses and Costs

After observing $z_k, z_{k-1}, \ldots, z_0$, the decision-maker chooses among three distinct actions:

- He decides that $f = f_0$ and draws no more z's
- He decides that $f = f_1$ and draws no more z's
- He postpones deciding now and instead chooses to draw a z_{k+1}

Associated with these three actions, the decision-maker can suffer three kinds of losses:

- A loss L_0 if he decides $f = f_0$ when actually $f = f_1$
- A loss L_1 if he decides $f = f_1$ when actually $f = f_0$
- A cost c if he postpones deciding and chooses instead to draw another z

27.2.2 Digression on Type I and Type II Errors

If we regard $f = f_0$ as a null hypothesis and $f = f_1$ as an alternative hypothesis, then L_1 and L_0 are losses associated with two types of statistical errors

- a type I error is an incorrect rejection of a true null hypothesis (a "false positive")
- a type II error is a failure to reject a false null hypothesis (a "false negative")

So when we treat $f = f_0$ as the null hypothesis

- We can think of L_1 as the loss associated with a type I error.
- We can think of L_0 as the loss associated with a type II error.

27.2.3 Intuition

Before proceeding, let's try to guess what an optimal decision rule might look like.

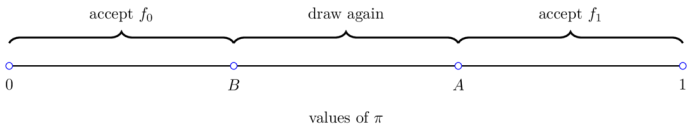
Suppose at some given point in time that π is close to 1.

Then our prior beliefs and the evidence so far point strongly to $f = f_1$.

If, on the other hand, π is close to 0, then $f = f_0$ is strongly favored.

Finally, if π is in the middle of the interval $[0, 1]$, then we are confronted with more uncertainty.

This reasoning suggests a sequential decision rule that we illustrate in the following figure:



As we'll see, this is indeed the correct form of the decision rule.

Our problem is to determine threshold values A, B that somehow depend on the parameters described above.

You might like to pause at this point and try to predict the impact of a parameter such as c or L_0 on A or B.

27.2.4 A Bellman Equation

Let $J(\pi)$ be the total loss for a decision-maker with current belief π who chooses optimally.

Principles of **dynamic programming** teach us that an optimal loss function J satisfies the following the Bellman functional equation

$$J(\pi) = \min \left\{ \underbrace{\pi L_0}_{\text{accept } f_0} , \underbrace{(1-\pi)L_1}_{\text{accept } f_1} , \underbrace{c + \mathbb{E}[J(\pi')]}_{\text{draw again}} \right\} \tag{27.1}$$

where π' is the random variable defined by Bayes' Law

$$\pi' = \kappa(z', \pi) = \frac{\pi f_1(z')}{(1-\pi) f_0(z') + \pi f_1(z')}$$

when π is fixed and z' is drawn from the current best guess, which is the distribution f defined by

$$f_\pi(v) = (1-\pi) f_0(v) + \pi f_1(v)$$

In the Bellman equation, minimization is over three actions:

1. Accept the hypothesis that $f = f_0$
2. Accept the hypothesis that $f = f_1$
3. Postpone deciding and draw again

We can represent the Bellman equation as

$$J(\pi) = \min\{\pi L_0,\ (1-\pi)L_1,\ h(\pi)\} \tag{27.2}$$

where $\pi \in [0,1]$ and

- πL_0 is the expected loss associated with accepting f_0 (i.e., the cost of making a type II error).
- $(1-\pi)L_1$ is the expected loss associated with accepting f_1 (i.e., the cost of making a type I error).
- $h(\pi) := c + \mathbb{E}[J(\pi')]$; this is the continuation value; i.e., the expected cost associated with drawing one more z.

The optimal decision rule is characterized by two numbers $A, B \in (0,1) \times (0,1)$ that satisfy

$$\pi L_0 < \min\{(1-\pi)L_1, c + \mathbb{E}[J(\pi')]\} \text{ if } \pi \leq B$$

and

$$(1-\pi)L_1 < \min\{\pi L_0, c + \mathbb{E}[J(\pi')]\} \text{ if } \pi \geq A$$

The optimal decision rule is then

$$\begin{aligned} \text{accept } f = f_1 \text{ if } \pi \geq A \\ \text{accept } f = f_0 \text{ if } \pi \leq B \\ \text{draw another } z \text{ if } B < \pi < A \end{aligned}$$

Our aim is to compute the cost function J as well as the associated cutoffs A and B.

To help make our computations more manageable, we can use (27.2) to write the continuation cost $h(\pi)$ as

$$\begin{aligned} h(\pi) &= c + \mathbb{E}[J(\pi')] \\ &= c + \mathbb{E}_{\pi'} \min\{\pi' L_0, (1-\pi')L_1, h(\pi')\} \\ &= c + \int \min\{\kappa(z',\pi)L_0, (1-\kappa(z',\pi))L_1, h(\kappa(z',\pi))\} f_\pi(z') dz' \end{aligned} \tag{27.3}$$

The equality

$$h(\pi) = c + \int \min\{\kappa(z',\pi)L_0, (1-\kappa(z',\pi))L_1, h(\kappa(z',\pi))\} f_\pi(z') dz' \tag{27.4}$$

is an equation in an unknown function h.

Note

Such an equation is called a **functional equation**.

Using the functional equation, (27.4), for the continuation cost, we can back out optimal choices using the right side of (27.2).

This functional equation can be solved by taking an initial guess and iterating to find a fixed point.

Thus, we iterate with an operator Q, where

$$Qh(\pi) = c + \int \min\{\kappa(z',\pi)L_0, (1-\kappa(z',\pi))L_1, h(\kappa(z',\pi))\} f_\pi(z') dz'$$

27.3 Implementation

First, we will construct a `jitclass` to store the parameters of the model

```
wf_data = [('a0', float64),          # Parameters of beta distributions
           ('b0', float64),
           ('a1', float64),
           ('b1', float64),
           ('c', float64),           # Cost of another draw
           ('π_grid_size', int64),
           ('L0', float64),          # Cost of selecting f0 when f1 is true
           ('L1', float64),          # Cost of selecting f1 when f0 is true
           ('π_grid', float64[:]),
           ('mc_size', int64),
           ('z0', float64[:]),
           ('z1', float64[:])]
```

```
@jitclass(wf_data)
class WaldFriedman:

    def __init__(self,
                 c=1.25,
                 a0=1,
                 b0=1,
                 a1=3,
                 b1=1.2,
                 L0=25,
                 L1=25,
                 π_grid_size=200,
                 mc_size=1000):

        self.a0, self.b0 = a0, b0
        self.a1, self.b1 = a1, b1
        self.c, self.π_grid_size = c, π_grid_size
        self.L0, self.L1 = L0, L1
        self.π_grid = np.linspace(0, 1, π_grid_size)
        self.mc_size = mc_size

        self.z0 = np.random.beta(a0, b0, mc_size)
        self.z1 = np.random.beta(a1, b1, mc_size)

    def f0(self, x):

        return p(x, self.a0, self.b0)

    def f1(self, x):

        return p(x, self.a1, self.b1)

    def f0_rvs(self):
        return np.random.beta(self.a0, self.b0)

    def f1_rvs(self):
        return np.random.beta(self.a1, self.b1)

    def κ(self, z, π):
        """
```

(continues on next page)

(continued from previous page)

```
        Updates π using Bayes' rule and the current observation z
        """

        f0, f1 = self.f0, self.f1

        π_f0, π_f1 = (1 - π) * f0(z), π * f1(z)
        π_new = π_f1 / (π_f0 + π_f1)

        return π_new
```

As in *Optimal Savings III: Stochastic Returns*, to approximate a continuous value function

- We iterate at a finite grid of possible values of π.
- When we evaluate $\mathbb{E}[J(\pi')]$ between grid points, we use linear interpolation.

We define the operator function `Q` below.

```
@jit(nopython=True, parallel=True)
def Q(h, wf):

    c, π_grid = wf.c, wf.π_grid
    L0, L1 = wf.L0, wf.L1
    z0, z1 = wf.z0, wf.z1
    mc_size = wf.mc_size

    κ = wf.κ

    h_new = np.empty_like(π_grid)
    h_func = lambda p: np.interp(p, π_grid, h)

    for i in prange(len(π_grid)):
        π = π_grid[i]

        # Find the expected value of J by integrating over z
        integral_f0, integral_f1 = 0, 0
        for m in range(mc_size):
            π_0 = κ(z0[m], π)  # Draw z from f0 and update π
            integral_f0 += min(π_0 * L0, (1 - π_0) * L1, h_func(π_0))

            π_1 = κ(z1[m], π)  # Draw z from f1 and update π
            integral_f1 += min(π_1 * L0, (1 - π_1) * L1, h_func(π_1))

        integral = ((1 - π) * integral_f0 + π * integral_f1) / mc_size

        h_new[i] = c + integral

    return h_new
```

To solve the key functional equation, we will iterate using `Q` to find the fixed point

```
@jit
def solve_model(wf, tol=1e-4, max_iter=1000):
    """
    Compute the continuation cost function

    * wf is an instance of WaldFriedman
    """
```

(continues on next page)

(continued from previous page)

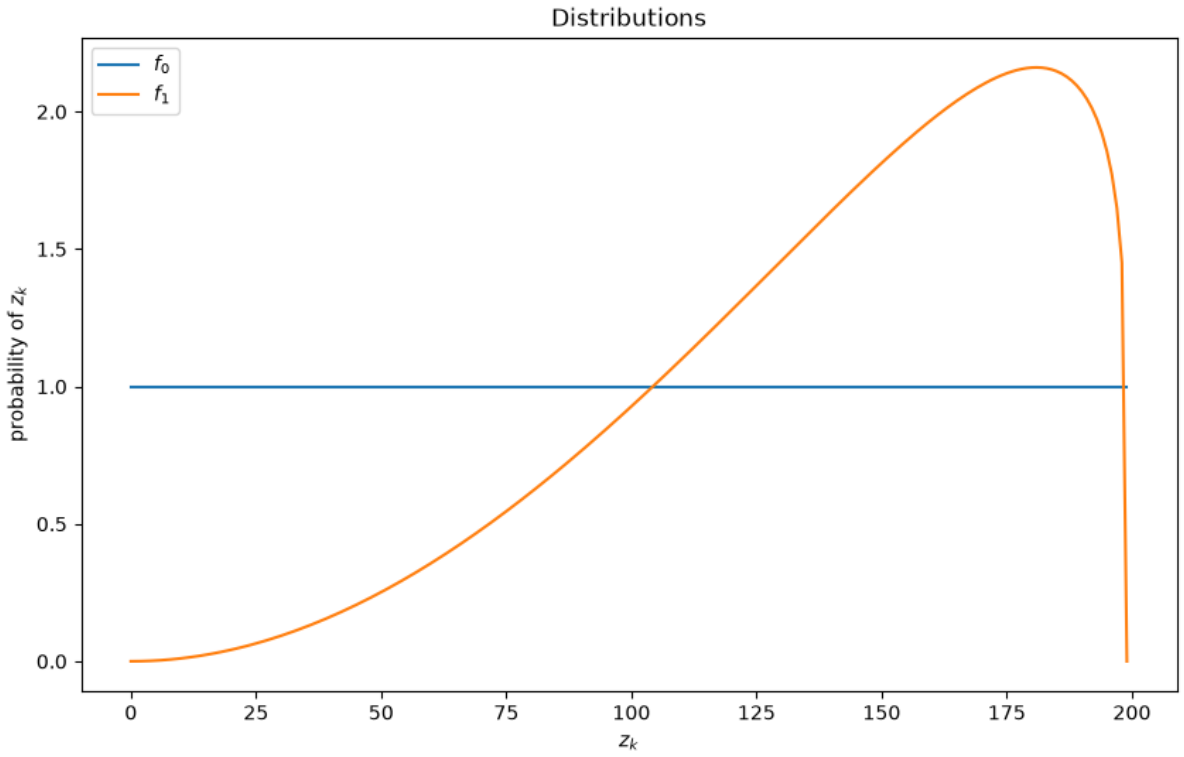
```
    # Set up loop
    h = np.zeros(len(wf.π_grid))
    i = 0
    error = tol + 1

    while i < max_iter and error > tol:
        h_new = Q(h, wf)
        error = np.max(np.abs(h - h_new))
        i += 1
        h = h_new

    if error > tol:
        print("Failed to converge!")

    return h_new
```

27.4 Analysis

Let's inspect outcomes.

We will be using the default parameterization with distributions like so

```
wf = WaldFriedman()

fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(wf.f0(wf.π_grid), label="$f_0$")
ax.plot(wf.f1(wf.π_grid), label="$f_1$")
ax.set(ylabel="probability of $z_k$", xlabel="$z_k$", title="Distributions")
ax.legend()

plt.show()
```



27.4.1 Cost Function

To solve the model, we will call our `solve_model` function

```
h_star = solve_model(wf)     # Solve the model
```

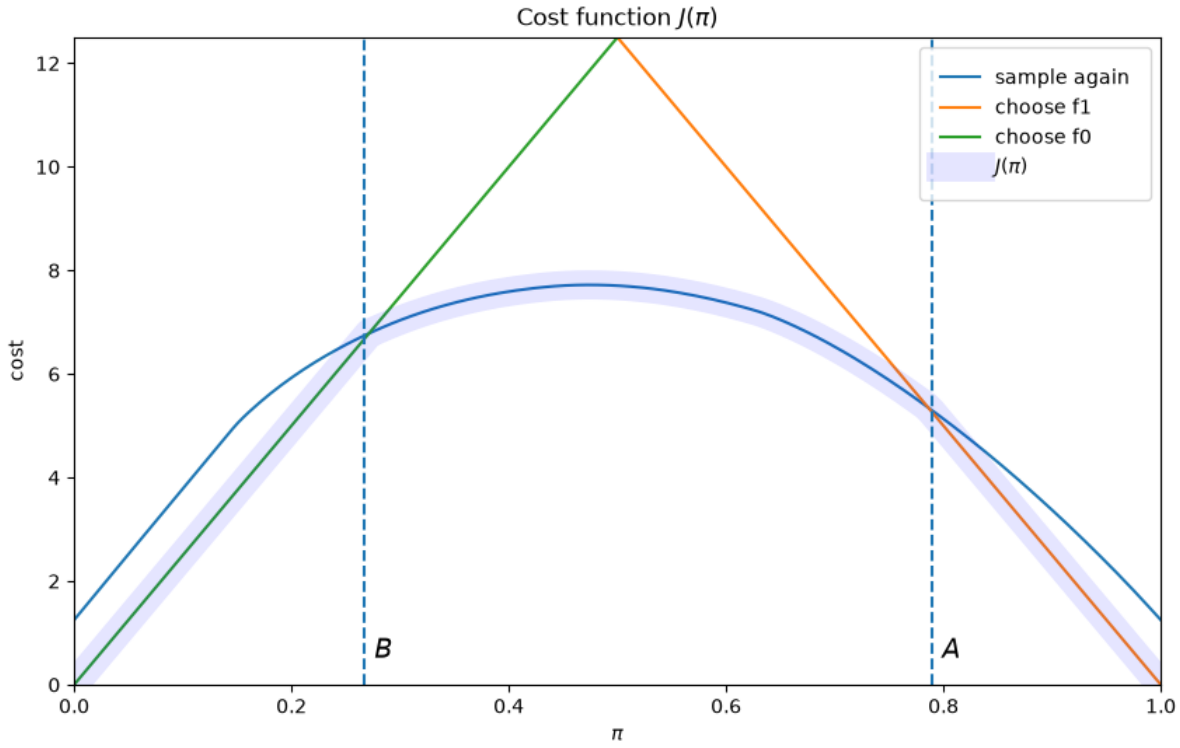
We will also set up a function to compute the cutoffs A and B and plot these on our cost function plot

```
@jit
def find_cutoff_rule(wf, h):

    """
    This function takes a continuation cost function and returns the
    corresponding cutoffs of where you transition between continuing and
    choosing a specific model
    """

    π_grid = wf.π_grid
    L0, L1 = wf.L0, wf.L1

    # Evaluate cost at all points on grid for choosing a model
    cost_f0 = π_grid * L0
    cost_f1 = (1 - π_grid) * L1

    # Find B: largest π where cost_f0 <= min(cost_f1, h)
    optimal_cost = np.minimum(np.minimum(cost_f0, cost_f1), h)
    choose_f0 = (cost_f0 <= cost_f1) & (cost_f0 <= h)

    if np.any(choose_f0):
```

(continues on next page)

(continued from previous page)

```
        B = π_grid[choose_f0][-1]  # Last point where we choose f0
    else:
        assert False, "No point where we choose f0"

    # Find A: smallest π where cost_f1 <= min(cost_f0, h)
    choose_f1 = (cost_f1 <= cost_f0) & (cost_f1 <= h)

    if np.any(choose_f1):
        A = π_grid[choose_f1][0]  # First point where we choose f1
    else:
        assert False, "No point where we choose f1"

    return (B, A)

B, A = find_cutoff_rule(wf, h_star)
cost_L0 = wf.π_grid * wf.L0
cost_L1 = (1 - wf.π_grid) * wf.L1

fig, ax = plt.subplots(figsize=(10, 6))

ax.plot(wf.π_grid, h_star, label='sample again')
ax.plot(wf.π_grid, cost_L1, label='choose f1')
ax.plot(wf.π_grid, cost_L0, label='choose f0')
ax.plot(wf.π_grid,
        np.amin(np.column_stack([h_star, cost_L0, cost_L1]),axis=1),
        lw=15, alpha=0.1, color='b', label=r'$J(\pi)$')

ax.annotate(r"$B$", xy=(B + 0.01, 0.5), fontsize=14)
ax.annotate(r"$A$", xy=(A + 0.01, 0.5), fontsize=14)

plt.vlines(B, 0, (1 - B) * wf.L1, linestyle="--")
plt.vlines(A, 0, A * wf.L0, linestyle="--")

ax.set(xlim=(0, 1), ylim=(0, 0.5 * max(wf.L0, wf.L1)), ylabel="cost",
       xlabel=r"$\pi$", title=r"Cost function $J(\pi)$")

plt.legend(borderpad=1.1)
plt.show()
```



The cost function J equals πL_0 for $\pi \leq B$, and $(1-\pi)L_1$ for $\pi \geq A$.

The slopes of the two linear pieces of the cost function $J(\pi)$ are determined by L_0 and $-L_1$.

The cost function J is smooth in the interior region, where the posterior probability assigned to f_1 is in the indecisive region $\pi \in (B, A)$.

The decision-maker continues to sample until the probability that he attaches to model f_1 falls below B or above A.

27.4.2 Simulations

The next figure shows the outcomes of 500 simulations of the decision process.

On the left is a histogram of **stopping times**, i.e., the number of draws of z_k required to make a decision.

The average number of draws is around 6.6.

On the right is the fraction of correct decisions at the stopping time.

In this case, the decision-maker is correct 80% of the time

```
def simulate(wf, true_dist, h_star, π_0=0.5):

    """
    This function takes an initial condition and simulates until it
    stops (when a decision is made)
    """

    f0, f1 = wf.f0, wf.f1
    f0_rvs, f1_rvs = wf.f0_rvs, wf.f1_rvs
    π_grid = wf.π_grid
    κ = wf.κ
```

(continues on next page)

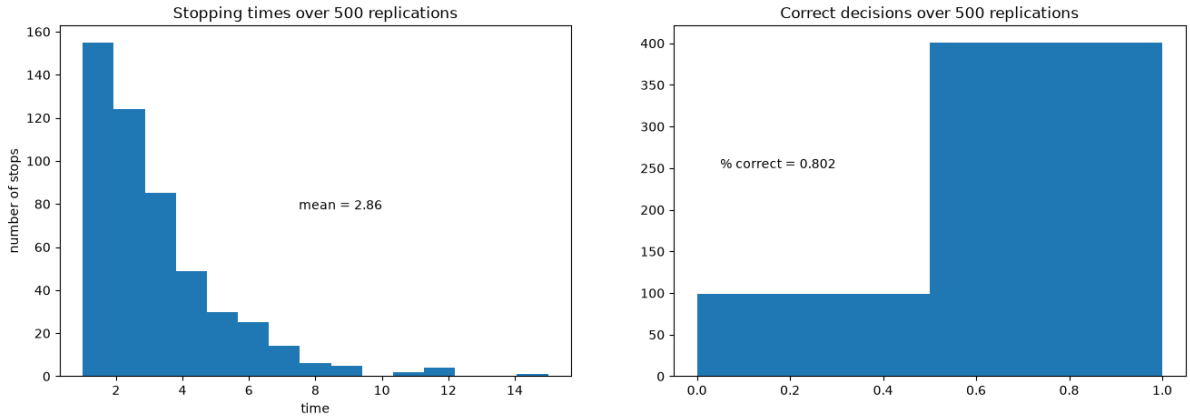
(continued from previous page)

```
    if true_dist == "f0":
        f, f_rvs = wf.f0, wf.f0_rvs
    elif true_dist == "f1":
        f, f_rvs = wf.f1, wf.f1_rvs

    # Find cutoffs
    B, A = find_cutoff_rule(wf, h_star)

    # Initialize a couple of useful variables
    decision_made = False
    π = π_0
    t = 0

    while decision_made is False:
        z = f_rvs()
        t = t + 1
        π = κ(z, π)
        if π < B:
            decision_made = True
            decision = 0
        elif π > A:
            decision_made = True
            decision = 1

    if true_dist == "f0":
        if decision == 0:
            correct = True
        else:
            correct = False

    elif true_dist == "f1":
        if decision == 1:
            correct = True
        else:
            correct = False

    return correct, π, t

def stopping_dist(wf, h_star, ndraws=250, true_dist="f0"):

    """
    Simulates repeatedly to get distributions of time needed to make a
    decision and how often they are correct
    """

    tdist = np.empty(ndraws, int)
    cdist = np.empty(ndraws, bool)

    for i in range(ndraws):
        correct, π, t = simulate(wf, true_dist, h_star)
        tdist[i] = t
        cdist[i] = correct

    return cdist, tdist

def simulation_plot(wf):
```

(continues on next page)

(continued from previous page)

```
    h_star = solve_model(wf)
    ndraws = 500
    cdist, tdist = stopping_dist(wf, h_star, ndraws)

    fig, ax = plt.subplots(1, 2, figsize=(16, 5))

    ax[0].hist(tdist, bins=np.max(tdist))
    ax[0].set_title(f"Stopping times over {ndraws} replications")
    ax[0].set(xlabel="time", ylabel="number of stops")
    ax[0].annotate(f"mean = {np.mean(tdist)}", xy=(max(tdist) / 2,
                   max(np.histogram(tdist, bins=max(tdist))[0]) / 2))

    ax[1].hist(cdist.astype(int), bins=2)
    ax[1].set_title(f"Correct decisions over {ndraws} replications")
    ax[1].annotate(f"% correct = {np.mean(cdist)}",
                   xy=(0.05, ndraws / 2))

    plt.show()

simulation_plot(wf)
```



27.4.3 Comparative Statics

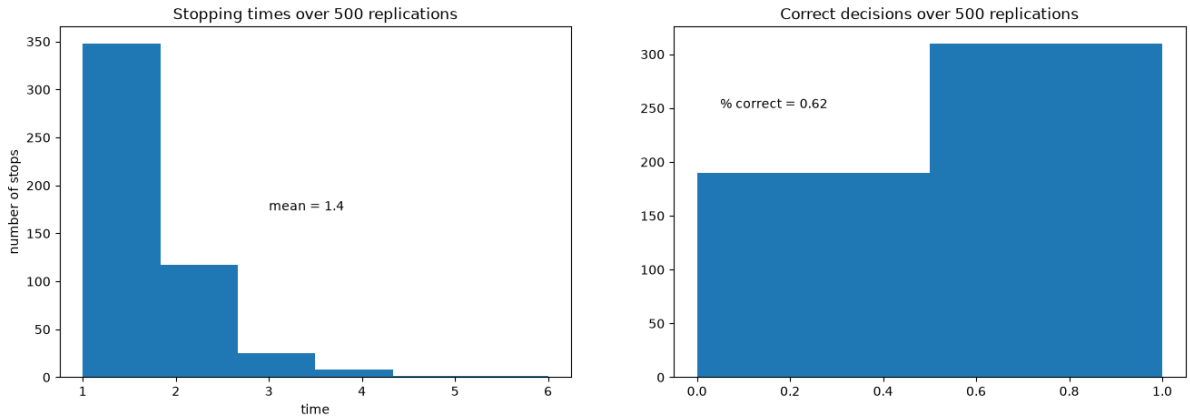
Now let's consider the following exercise.

We double the cost of drawing an additional observation.

Before you look, think about what will happen:

- Will the decision-maker be correct more or less often?
- Will he make decisions sooner or later?

```
wf = WaldFriedman(c=2.5)
simulation_plot(wf)
```



Increased cost per draw has induced the decision-maker to take fewer draws before deciding.

Because he decides with fewer draws, the percentage of time he is correct drops.

This leads to him having a higher expected loss when he puts equal weight on both models.

To facilitate comparative statics, we invite you to adjust the parameters of the model and investigate

- effects on the smoothness of the value function in the indecisive middle range as we increase the number of grid points in the piecewise linear approximation.
- effects of different settings for the cost parameters L_0, L_1, c, the parameters of two beta distributions f_0 and f_1, and the number of points and linear functions m to use in the piecewise continuous approximation to the value function.
- various simulations from f_0 and associated distributions of waiting times to making a decision.
- associated histograms of correct and incorrect decisions.

CHAPTER

TWENTYEIGHT

EXCHANGEABILITY AND BAYESIAN UPDATING

Contents

- *Exchangeability and Bayesian Updating*
  - *Overview*
  - *Independently and Identically Distributed*
  - *A Setting in Which Past Observations Are Informative*
  - *Relationship Between IID and Exchangeable*
  - *Exchangeability*
  - *Bayes' Law*
  - *More Details about Bayesian Updating*
  - *Appendix*
  - *Sequels*

28.1 Overview

This lecture studies learning via Bayes' Law.

We touch foundations of Bayesian statistical inference invented by Bruno DeFinetti [de Finetti, 1937].

The relevance of DeFinetti's work for economists is presented forcefully by David Kreps in chapter 11 of [Kreps, 1988].

An example that we study in this lecture is a key component of *this lecture* that augments the *classic* job search model of McCall [McCall, 1970] by presenting an unemployed worker with a statistical inference problem.

Here we create graphs that illustrate the role that a likelihood ratio plays in Bayes' Law.

We'll use such graphs to provide insights into mechanics driving outcomes in *this lecture* about learning in an augmented McCall job search model.

Among other things, this lecture discusses connections between the statistical concepts of sequences of random variables that are

- independently and identically distributed
- exchangeable (also known as *conditionally* independently and identically distributed)

Understanding these concepts is essential for appreciating how Bayesian updating works.

You can read about exchangeability here.

Because another term for **exchangeable** is **conditionally independent**, we want to convey an answer to the question *conditional on what?*

We also tell why an assumption of independence precludes learning while an assumption of conditional independence makes learning possible.

Below, we'll often use

- W to denote a random variable
- w to denote a particular realization of a random variable W

Let's start with some imports:

```
import matplotlib.pyplot as plt
from numba import jit, vectorize
from math import gamma
import scipy.optimize as op
from scipy.integrate import quad
import numpy as np
```

28.2 Independently and Identically Distributed

We begin by looking at the notion of an **independently and identically distributed sequence** of random variables.

An independently and identically distributed sequence is often abbreviated as IID.

Two notions are involved

- **independence**
- **identically distributed**

A sequence $W_0, W_1, ...$ is **independently distributed** if the joint probability density of the sequence is the **product** of the densities of the components of the sequence.

The sequence $W_0, W_1, ...$ is **independently and identically distributed** (IID) if in addition the marginal density of W_t is the same for all $t = 0, 1, ...$

For example, let $p(W_0, W_1, ...)$ be the **joint density** of the sequence and let $p(W_t)$ be the **marginal density** for a particular W_t for all $t = 0, 1, ...$

Then the joint density of the sequence $W_0, W_1, ...$ is IID if

$$p(W_0, W_1, ...) = p(W_0)p(W_1)\cdots$$

so that the joint density is the product of a sequence of identical marginal densities.

28.2.1 IID Means Past Observations Don't Tell Us Anything About Future Observations

If a sequence is random variables is IID, past information provides no information about future realizations.

Therefore, there is **nothing to learn** from the past about the future.

To understand these statements, let the joint distribution of a sequence of random variables $\{W_t\}_{t=0}^T$ that is not necessarily IID be

$$p(W_T, W_{T-1}, \dots, W_1, W_0)$$

Using the laws of probability, we can always factor such a joint density into a product of conditional densities:

$$\begin{aligned} p(W_T, W_{T-1}, \dots, W_1, W_0) =& p(W_T|W_{T-1}, \dots, W_0)p(W_{T-1}|W_{T-2}, \dots, W_0)\cdots \\ & \cdots p(W_1|W_0)p(W_0) \end{aligned}$$

In general,

$$p(W_t|W_{t-1}, \dots, W_0) \neq p(W_t)$$

which states that the **conditional density** on the left side does not equal the **marginal density** on the right side.

But in the special IID case,

$$p(W_t|W_{t-1}, \dots, W_0) = p(W_t),$$

so that the partial history $W_{t-1}, \dots, W_0$ contains no information about the probability of W_t.

So in the IID case, there is **nothing to learn** about the densities of future random variables from past random variables.

But when the sequence is not IID, there is something to learn about the future from observations of past random variables.

We turn next to an instance of the general case in which the sequence is not IID.

Please watch for what can be learned from the past and when.

28.3 A Setting in Which Past Observations Are Informative

Let $\{W_t\}_{t=0}^\infty$ be a sequence of nonnegative scalar random variables with a joint probability distribution constructed as follows.

There are two distinct cumulative distribution functions F and G that have densities f and g, respectively, for a nonnegative scalar random variable W.

Before the start of time, say at time $t = -1$, "nature" once and for all selects **either** f **or** g.

Thereafter at each time $t \geq 0$, nature draws a random variable W_t from the selected distribution.

So the data are permanently generated as independently and identically distributed (IID) draws from **either** F **or** G.

We could say that *objectively*, meaning *after* nature has chosen either F or G, the probability that the data are generated as draws from F is either 0 or 1.

We now drop into this setting a partially informed decision maker who

- knows both F and G, but
- does not know whether at $t = -1$ nature had drawn F or whether nature had drawn G once-and-for-all

Thus, although our decision maker knows F and knows G, he does not know which of these two known distributions nature had selected to draw from.

The decision maker describes his ignorance with a **subjective probability** $\tilde{\pi}$ and reasons as if nature had selected F with probability $\tilde{\pi} \in (0, 1)$ and G with probability $1 - \tilde{\pi}$.

Thus, we assume that the decision maker

- **knows** both F and G
- **doesn't know** which of these two distributions that nature has drawn
- expresses his ignorance by **acting as if** or **thinking that** nature chose distribution F with probability $\tilde{\pi} \in (0, 1)$ and distribution G with probability $1 - \tilde{\pi}$
- at date $t \geq 0$ knows the partial history $w_t, w_{t-1}, \ldots, w_0$

To proceed, we want to know the decision maker's belief about the joint distribution of the partial history.

We'll discuss that next and in the process describe the concept of **exchangeability**.

28.4 Relationship Between IID and Exchangeable

Conditional on nature selecting F, the joint density of the sequence $W_0, W_1, \ldots$ is

$$f(W_0)f(W_1)\cdots$$

Conditional on nature selecting G, the joint density of the sequence $W_0, W_1, \ldots$ is

$$g(W_0)g(W_1)\cdots$$

Thus, **conditional on nature having selected** F, the sequence $W_0, W_1, \ldots$ is independently and identically distributed.

Furthermore, **conditional on nature having selected** G, the sequence $W_0, W_1, \ldots$ is also independently and identically distributed.

But what about the **unconditional distribution** of a partial history?

The unconditional distribution of $W_0, W_1, \ldots$ is evidently

$$h(W_0, W_1, \ldots) \equiv \tilde{\pi}[f(W_0)f(W_1)\cdots] + (1 - \tilde{\pi})[g(W_0)g(W_1)\cdots] \tag{28.1}$$

Under the unconditional distribution $h(W_0, W_1, \ldots)$, the sequence $W_0, W_1, \ldots$ is **not** independently and identically distributed.

To verify this claim, it is sufficient to notice, for example, that

$$h(W_0, W_1) = \tilde{\pi}f(W_0)f(W_1) + (1 - \tilde{\pi})g(W_0)g(W_1) \neq (\tilde{\pi}f(W_0) + (1 - \tilde{\pi})g(W_0))(\tilde{\pi}f(W_1) + (1 - \tilde{\pi})g(W_1))$$

Thus, the conditional distribution

$$h(W_1|W_0) \equiv \frac{h(W_0, W_1)}{(\tilde{\pi}f(W_0) + (1 - \tilde{\pi})g(W_0))} \neq (\tilde{\pi}f(W_1) + (1 - \tilde{\pi})g(W_1))$$

This means that random variable W_0 contains information about random variable W_1.

So there is something to learn from the past about the future.

28.5 Exchangeability

While the sequence $W_0, W_1, \ldots$ is not IID, it can be verified that it is **exchangeable**, which means that the joint distributions $h(W_0, W_1)$ and $h(W_1, W_0)$ of the "re-ordered" sequences satisfy

$$h(W_0, W_1) = h(W_1, W_0)$$

and so on.

More generally, a sequence of random variables is said to be **exchangeable** if the joint probability distribution for a sequence does not change when the positions in the sequence in which finitely many of random variables appear are altered.

Equation (28.1) represents our instance of an exchangeable joint density over a sequence of random variables as a **mixture** of two IID joint densities over a sequence of random variables.

A Bayesian statistician interprets the mixing parameter $\tilde{\pi} \in (0, 1)$ as a decision maker's subjective belief – the decision maker's **prior probability** – that nature had selected probability distribution F.

Note

DeFinetti [de Finetti, 1937] established a related representation of an exchangeable process created by mixing sequences of IID Bernoulli random variables with parameter $\theta \in (0, 1)$ and mixing probability density $\pi(\theta)$ that a Bayesian statistician would interpret as a prior over the unknown Bernoulli parameter θ.

28.6 Bayes' Law

We noted above that in our example model there is something to learn about about the future from past data drawn from our particular instance of a process that is exchangeable but not IID.

But how can we learn?

And about what?

The answer to the *about what* question is $\tilde{\pi}$.

The answer to the *how* question is to use Bayes' Law.

Another way to say *use Bayes' Law* is to say *from a (subjective) joint distribution, compute an appropriate conditional distribution*.

Let's dive into Bayes' Law in this context.

Let q represent the distribution that nature actually draws w from and let

$$\pi = \mathbb{P}\{q = f\}$$

where we regard π as a decision maker's **subjective probability** (also called a **personal probability**).

Suppose that at $t \geq 0$, the decision maker has observed a history $w^t \equiv [w_t, w_{t-1}, \ldots, w_0]$.

We let

$$\pi_t = \mathbb{P}\{q = f | w^t\}$$

where we adopt the convention

$$\pi_{-1} = \tilde{\pi}$$

The distribution of w_{t+1} conditional on w^t is then

$$\pi_t f + (1 - \pi_t) g.$$

Bayes' rule for updating π_{t+1} is

$$\pi_{t+1} = \frac{\pi_t f(w_{t+1})}{\pi_t f(w_{t+1}) + (1 - \pi_t) g(w_{t+1})} \tag{28.2}$$

Equation (28.2) follows from Bayes' rule, which tells us that

$$\mathbb{P}\{q = f \,|\, W = w\} = \frac{\mathbb{P}\{W = w \,|\, q = f\}\mathbb{P}\{q = f\}}{\mathbb{P}\{W = w\}}$$

where

$$\mathbb{P}\{W = w\} = \sum_{a \in \{f, g\}} \mathbb{P}\{W = w \,|\, q = a\}\mathbb{P}\{q = a\}$$

28.7 More Details about Bayesian Updating

Let's stare at and rearrange Bayes' Law as represented in equation (28.2) with the aim of understanding how the **posterior** probability π_{t+1} is influenced by the **prior** probability π_t and the **likelihood ratio**

$$l(w) = \frac{f(w)}{g(w)}$$

It is convenient for us to rewrite the updating rule (28.2) as

$$\pi_{t+1} = \frac{\pi_t f\left(w_{t+1}\right)}{\pi_t f\left(w_{t+1}\right) + \left(1 - \pi_t\right) g\left(w_{t+1}\right)} = \frac{\pi_t \frac{f(w_{t+1})}{g(w_{t+1})}}{\pi_t \frac{f(w_{t+1})}{g(w_{t+1})} + \left(1 - \pi_t\right)} = \frac{\pi_t l\left(w_{t+1}\right)}{\pi_t l\left(w_{t+1}\right) + \left(1 - \pi_t\right)}$$

This implies that

$$\frac{\pi_{t+1}}{\pi_t} = \frac{l\left(w_{t+1}\right)}{\pi_t l\left(w_{t+1}\right) + \left(1 - \pi_t\right)} \begin{cases} > 1 & \text{if } l\left(w_{t+1}\right) > 1 \\ \leq 1 & \text{if } l\left(w_{t+1}\right) \leq 1 \end{cases} \tag{28.3}$$

Notice how the likelihood ratio and the prior interact to determine whether an observation w_{t+1} leads the decision maker to increase or decrease the subjective probability he/she attaches to distribution F.

When the likelihood ratio $l(w_{t+1})$ exceeds one, the observation w_{t+1} nudges the probability π put on distribution F upward, and when the likelihood ratio $l(w_{t+1})$ is less that one, the observation w_{t+1} nudges π downward.

Representation (28.3) is the foundation of some graphs that we'll use to display the dynamics of $\{\pi_t\}_{t=0}^{\infty}$ that are induced by Bayes' Law.

We'll plot $l\left(w\right)$ as a way to enlighten us about how learning – i.e., Bayesian updating of the probability π that nature has chosen distribution f – works.

To create the Python infrastructure to do our work for us, we construct a wrapper function that displays informative graphs given parameters of f and g.

```
@vectorize
def p(x, a, b):
    "The general beta distribution function."
    r = gamma(a + b) / (gamma(a) * gamma(b))
```

(continues on next page)

(continued from previous page)

```
    return r * x ** (a-1) * (1 - x) ** (b-1)

def learning_example(F_a=1, F_b=1, G_a=3, G_b=1.2):
    """
    A wrapper function that displays the updating rule of belief π,
    given the parameters which specify F and G distributions.
    """

    f = jit(lambda x: p(x, F_a, F_b))
    g = jit(lambda x: p(x, G_a, G_b))

    # l(w) = f(w) / g(w)
    l = lambda w: f(w) / g(w)
    # objective function for solving l(w) = 1
    obj = lambda w: l(w) - 1

    x_grid = np.linspace(0, 1, 100)
    π_grid = np.linspace(1e-3, 1-1e-3, 100)

    w_max = 1
    w_grid = np.linspace(1e-12, w_max-1e-12, 100)

    # the mode of beta distribution
    # use this to divide w into two intervals for root finding
    G_mode = (G_a - 1) / (G_a + G_b - 2)
    roots = np.empty(2)
    roots[0] = op.root_scalar(obj, bracket=[1e-10, G_mode]).root
    roots[1] = op.root_scalar(obj, bracket=[G_mode, 1-1e-10]).root

    fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(18, 5))

    ax1.plot(l(w_grid), w_grid, label='$l$', lw=2)
    ax1.vlines(1., 0., 1., linestyle="--")
    ax1.hlines(roots, 0., 2., linestyle="--")
    ax1.set_xlim([0., 2.])
    ax1.legend(loc=4)
    ax1.set(xlabel='$l(w)=f(w)/g(w)$', ylabel='$w$')

    ax2.plot(f(x_grid), x_grid, label='$f$', lw=2)
    ax2.plot(g(x_grid), x_grid, label='$g$', lw=2)
    ax2.vlines(1., 0., 1., linestyle="--")
    ax2.hlines(roots, 0., 2., linestyle="--")
    ax2.legend(loc=4)
    ax2.set(xlabel='$f(w), g(w)$', ylabel='$w$')

    area1 = quad(f, 0, roots[0])[0]
    area2 = quad(g, roots[0], roots[1])[0]
    area3 = quad(f, roots[1], 1)[0]

    ax2.text((f(0) + f(roots[0])) / 4, roots[0] / 2, f"{area1: .3g}")
    ax2.fill_between([0, 1], 0, roots[0], color='blue', alpha=0.15)
    ax2.text(np.mean(g(roots)) / 2, np.mean(roots), f"{area2: .3g}")
    w_roots = np.linspace(roots[0], roots[1], 20)
    ax2.fill_betweenx(w_roots, 0, g(w_roots), color='orange', alpha=0.15)
    ax2.text((f(roots[1]) + f(1)) / 4, (roots[1] + 1) / 2, f"{area3: .3g}")
    ax2.fill_between([0, 1], roots[1], 1, color='blue', alpha=0.15)
```

(continues on next page)

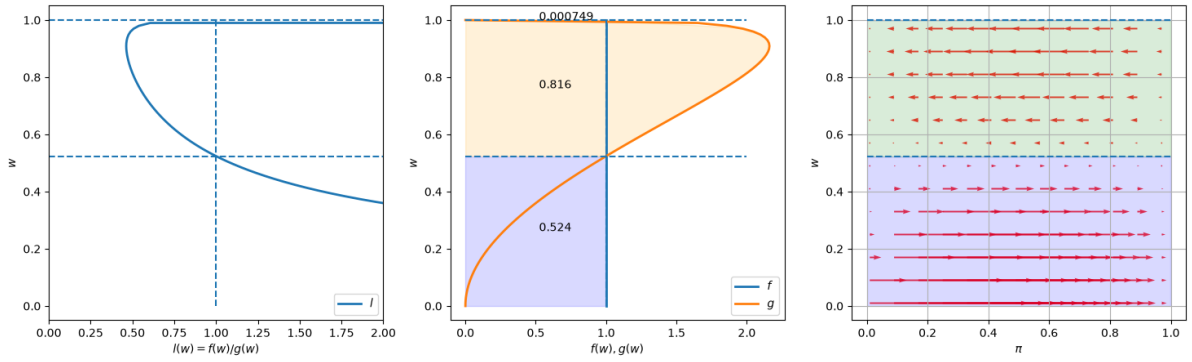
(continued from previous page)

```
    W = np.arange(0.01, 0.99, 0.08)
    Π = np.arange(0.01, 0.99, 0.08)

    ΔW = np.zeros((len(W), len(Π)))
    ΔΠ = np.empty((len(W), len(Π)))
    for i, w in enumerate(W):
        for j, π in enumerate(Π):
            lw = l(w)
            ΔΠ[i, j] = π * (lw / (π * lw + 1 - π) - 1)

    q = ax3.quiver(Π, W, ΔΠ, ΔW, scale=2, color='r', alpha=0.8)

    ax3.fill_between(π_grid, 0, roots[0], color='blue', alpha=0.15)
    ax3.fill_between(π_grid, roots[0], roots[1], color='green', alpha=0.15)
    ax3.fill_between(π_grid, roots[1], w_max, color='blue', alpha=0.15)
    ax3.hlines(roots, 0., 1., linestyle="--")
    ax3.set(xlabel=r'$\pi$', ylabel='$w$')
    ax3.grid()

    plt.show()
```

Now we'll create a group of graphs that illustrate dynamics induced by Bayes' Law.

We'll begin with Python function default values of various objects, then change them in a subsequent example.

```
learning_example()
```



Please look at the three graphs above created for an instance in which f is a uniform distribution on $[0, 1]$ (i.e., a Beta distribution with parameters $F_a = 1, F_b = 1$), while g is a Beta distribution with the default parameter values $G_a = 3, G_b = 1.2$.

The graph on the left plots the likelihood ratio $l(w)$ as the absciassa axis against w as the ordinate.

The middle graph plots both $f(w)$ and $g(w)$ against w, with the horizontal dotted lines showing values of w at which the likelihood ratio equals 1.

The graph on the right plots arrows to the right that show when Bayes' Law makes π increase and arrows to the left that show when Bayes' Law make π decrease.

Lengths of the arrows show magnitudes of the force from Bayes' Law impelling π to change.

These lengths depend on both the prior probability π on the abscissa axis and the evidence in the form of the current draw of w on the ordinate axis.

The fractions in the colored areas of the middle graphs are probabilities under F and G, respectively, that realizations of w fall into the interval that updates the belief π in a correct direction (i.e., toward 0 when G is the true distribution, and

toward 1 when F is the true distribution).

For example, in the above example, under true distribution F, π will be updated toward 0 if w falls into the interval $[0.524, 0.999]$, which occurs with probability $1 - .524 = .476$ under F.

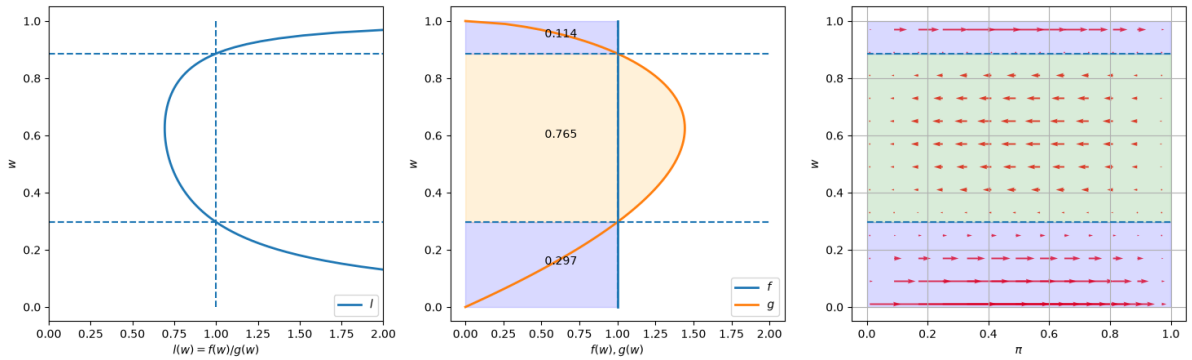
But this would occur with probability 0.816 if G were the true distribution.

The fraction 0.816 in the orange region is the integral of $g(w)$ over this interval.

Next we use our code to create graphs for another instance of our model.

We keep F the same as in the preceding instance, namely a uniform distribution, but now assume that G is a Beta distribution with parameters $G_a = 2, G_b = 1.6$.

```
learning_example(G_a=2, G_b=1.6)
```



Notice how the likelihood ratio, the middle graph, and the arrows compare with the previous instance of our example.

28.8 Appendix

28.8.1 Sample Paths of π_t

Now we'll have some fun by plotting multiple realizations of sample paths of π_t under two possible assumptions about nature's choice of distribution, namely

- that nature permanently draws from F
- that nature permanently draws from G

Outcomes depend on a peculiar property of likelihood ratio processes discussed in this lecture.

To proceed, we create some Python code.

```
def function_factory(F_a=1, F_b=1, G_a=3, G_b=1.2):

    # define f and g
    f = jit(lambda x: p(x, F_a, F_b))
    g = jit(lambda x: p(x, G_a, G_b))

    @jit
    def update(a, b, π):
        "Update π by drawing from beta distribution with parameters a and b"

        # Draw
        w = np.random.beta(a, b)
```

(continues on next page)

(continued from previous page)

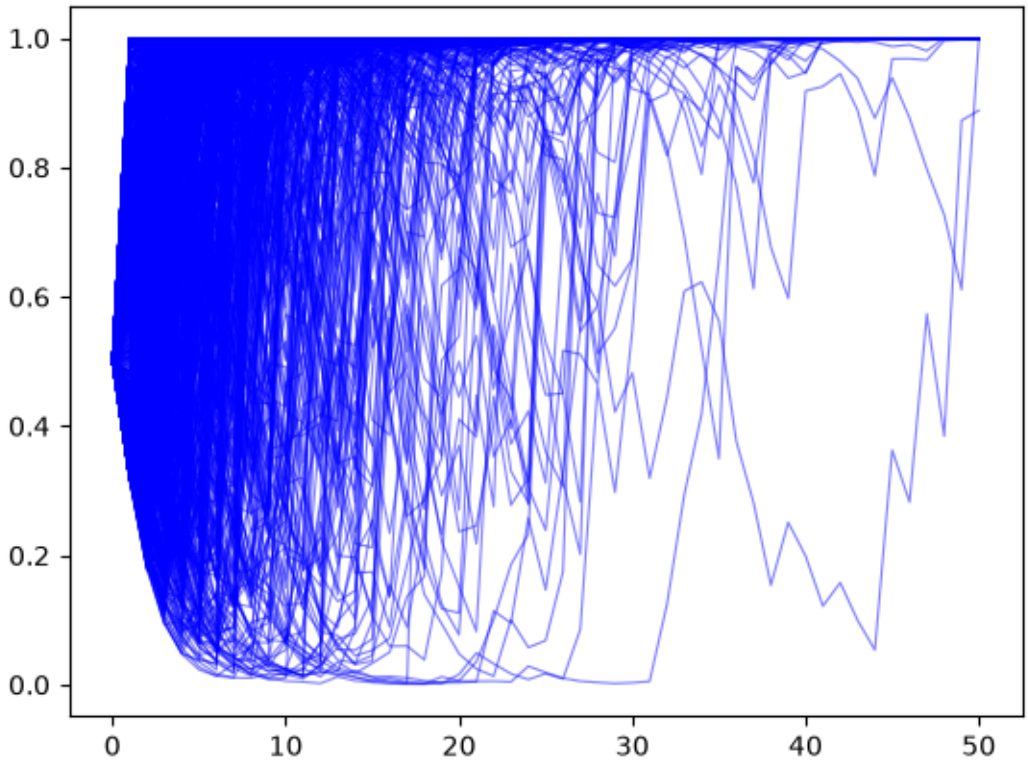
```
        # Update belief
        π = 1 / (1 + ((1 - π) * g(w)) / (π * f(w)))

        return π

    @jit
    def simulate_path(a, b, T=50):
        "Simulates a path of beliefs π with length T"

        π = np.empty(T+1)

        # initial condition
        π[0] = 0.5

        for t in range(1, T+1):
            π[t] = update(a, b, π[t-1])

        return π

    def simulate(a=1, b=1, T=50, N=200, display=True):
        "Simulates N paths of beliefs π with length T"

        π_paths = np.empty((N, T+1))
        if display:
            fig = plt.figure()

        for i in range(N):
            π_paths[i] = simulate_path(a=a, b=b, T=T)
            if display:
                plt.plot(range(T+1), π_paths[i], color='b', lw=0.8, alpha=0.5)

        if display:
            plt.show()

        return π_paths

    return simulate
```

```
simulate = function_factory()
```

We begin by generating N simulated $\{\pi_t\}$ paths with T periods when the sequence is truly IID draws from F. We set an initial prior $\pi_{-1} = .5$.

```
T = 50
```

```
# when nature selects F
π_paths_F = simulate(a=1, b=1, T=T, N=1000)
```

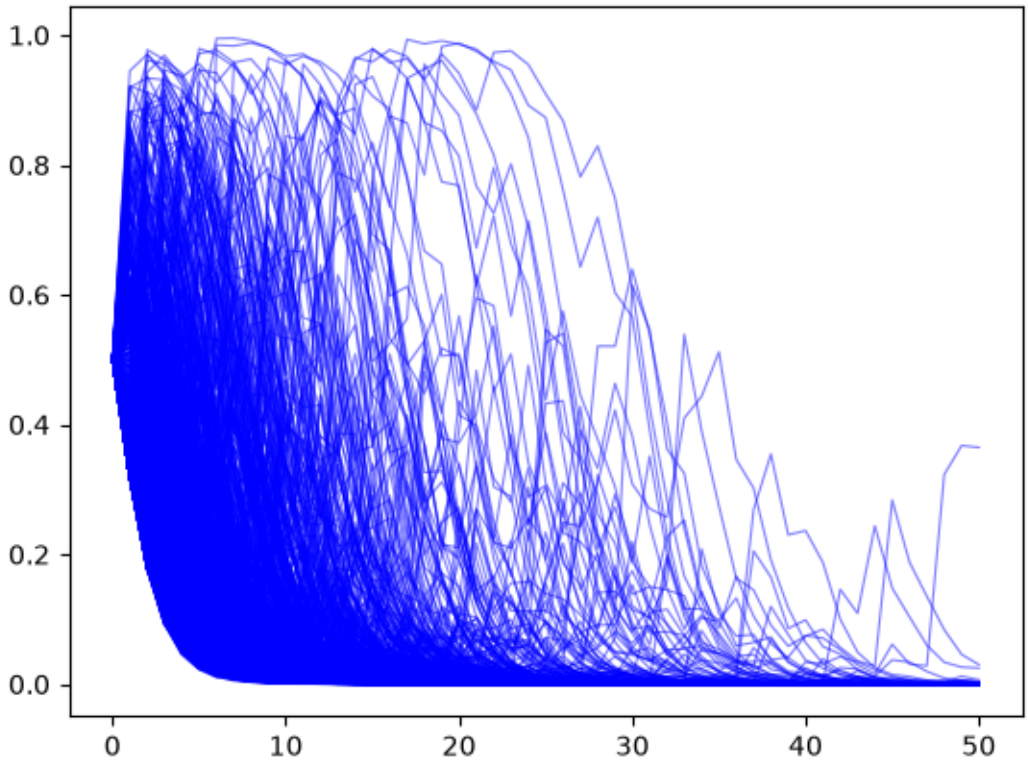


In the above example, for most paths $\pi_t \to 1$.

So Bayes' Law evidently eventually discovers the truth for most of our paths.

Next, we generate paths with T periods when the sequence is truly IID draws from G. Again, we set the initial prior $\pi_{-1} = .5$.

```
# when nature selects G
π_paths_G = simulate(a=3, b=1.2, T=T, N=1000)
```



In the above graph we observe that now most paths $\pi_t \to 0$.

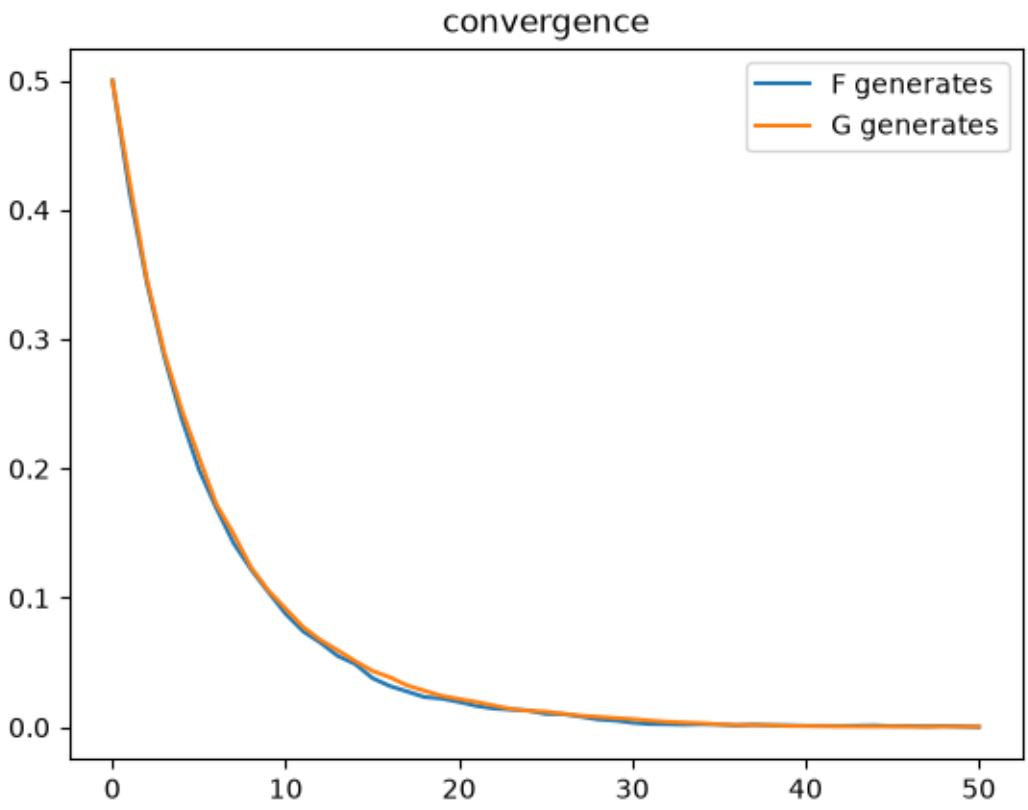
28.8.2 Rates of convergence

We study rates of convergence of π_t to 1 when nature generates the data as IID draws from F and of convergence of π_t to 0 when nature generates IID draws from G.

We do this by averaging across simulated paths of $\{\pi_t\}_{t=0}^T$.

Using N simulated π_t paths, we compute $1 - \sum_{i=1}^N \pi_{i,t}$ at each t when the data are generated as draws from F and compute $\sum_{i=1}^N \pi_{i,t}$ when the data are generated as draws from G.

```
plt.plot(range(T+1), 1 - np.mean(π_paths_F, 0), label='F generates')
plt.plot(range(T+1), np.mean(π_paths_G, 0), label='G generates')
plt.legend()
plt.title("convergence");
```



From the above graph, rates of convergence appear not to depend on whether F or G generates the data.

28.8.3 Graph of Ensemble Dynamics of π_t

More insights about the dynamics of $\{\pi_t\}$ can be gleaned by computing conditional expectations of $\frac{\pi_{t+1}}{\pi_t}$ as functions of π_t via integration with respect to the pertinent probability distribution:

$$
\begin{aligned}
E\left[\frac{\pi_{t+1}}{\pi_t}\,\middle|\, q=a,\pi_t\right] &= E\left[\frac{l\left(w_{t+1}\right)}{\pi_t l\left(w_{t+1}\right)+\left(1-\pi_t\right)}\,\middle|\, q=a,\pi_t\right], \\
&= \int_0^1 \frac{l\left(w_{t+1}\right)}{\pi_t l\left(w_{t+1}\right)+\left(1-\pi_t\right)} a\left(w_{t+1}\right) dw_{t+1}
\end{aligned}
$$

where $a = f, g$.

The following code approximates the integral above:

```
def expected_ratio(F_a=1, F_b=1, G_a=3, G_b=1.2):

    # define f and g
    f = jit(lambda x: p(x, F_a, F_b))
    g = jit(lambda x: p(x, G_a, G_b))

    l = lambda w: f(w) / g(w)
    integrand_f = lambda w, π: f(w) * l(w) / (π * l(w) + 1 - π)
    integrand_g = lambda w, π: g(w) * l(w) / (π * l(w) + 1 - π)

    π_grid = np.linspace(0.02, 0.98, 100)
```

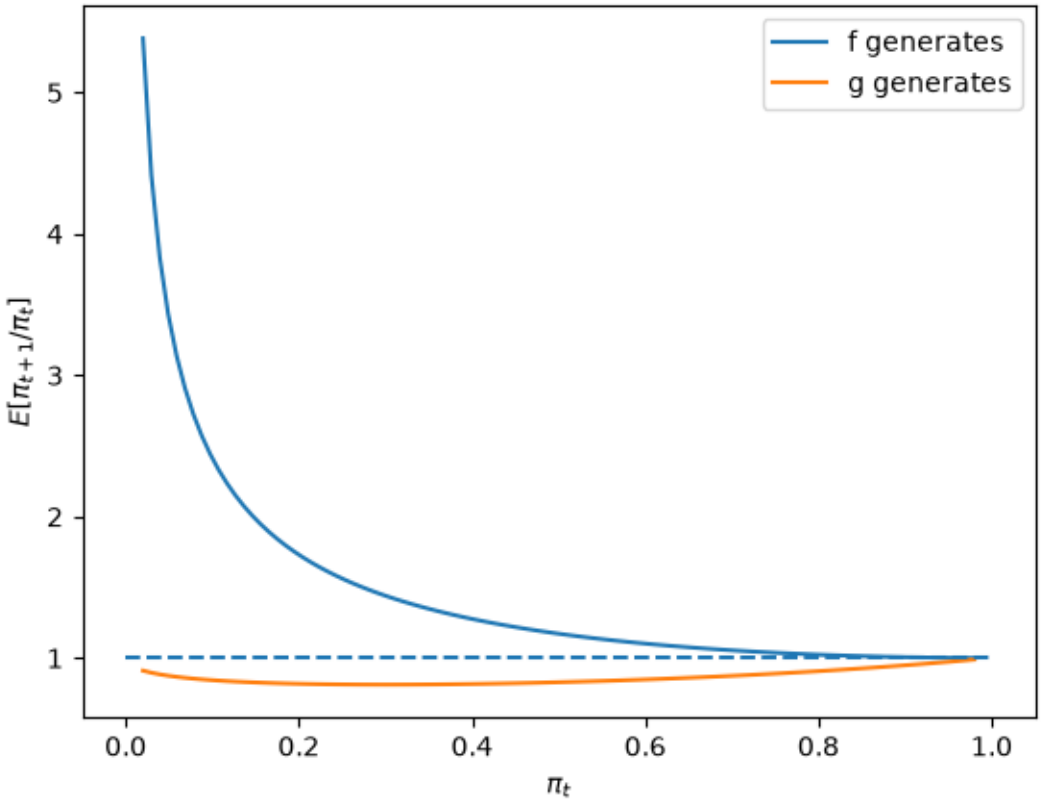
(continues on next page)

(continued from previous page)

```
    expected_rario = np.empty(len(π_grid))
    for q, inte in zip(["f", "g"], [integrand_f, integrand_g]):
        for i, π in enumerate(π_grid):
            expected_rario[i]= quad(inte, 0, 1, args=(π,))[0]
        plt.plot(π_grid, expected_rario, label=f"{q} generates")

    plt.hlines(1, 0, 1, linestyle="--")
    plt.xlabel(r"$\pi_t$")
    plt.ylabel(r"$E[\pi_{t+1}/\pi_t]$")
    plt.legend()

    plt.show()
```

First, consider the case where $F_a = F_b = 1$ and $G_a = 3, G_b = 1.2$.

```
expected_ratio()
```

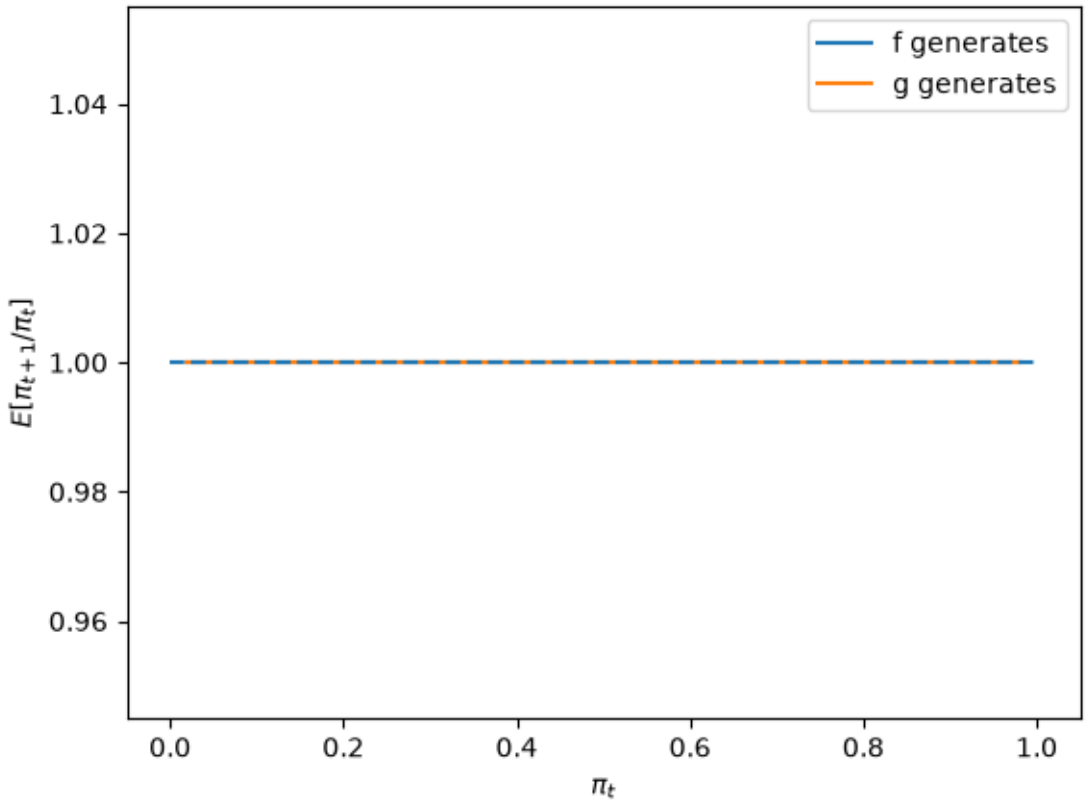


The above graphs shows that when F generates the data, π_t on average always heads north, while when G generates the data, π_t heads south.

Next, we'll look at a degenerate case in whcih f and g are identical beta distributions, and $F_a = G_a = 3, F_b = G_b = 1.2$.

In a sense, here there is nothing to learn.

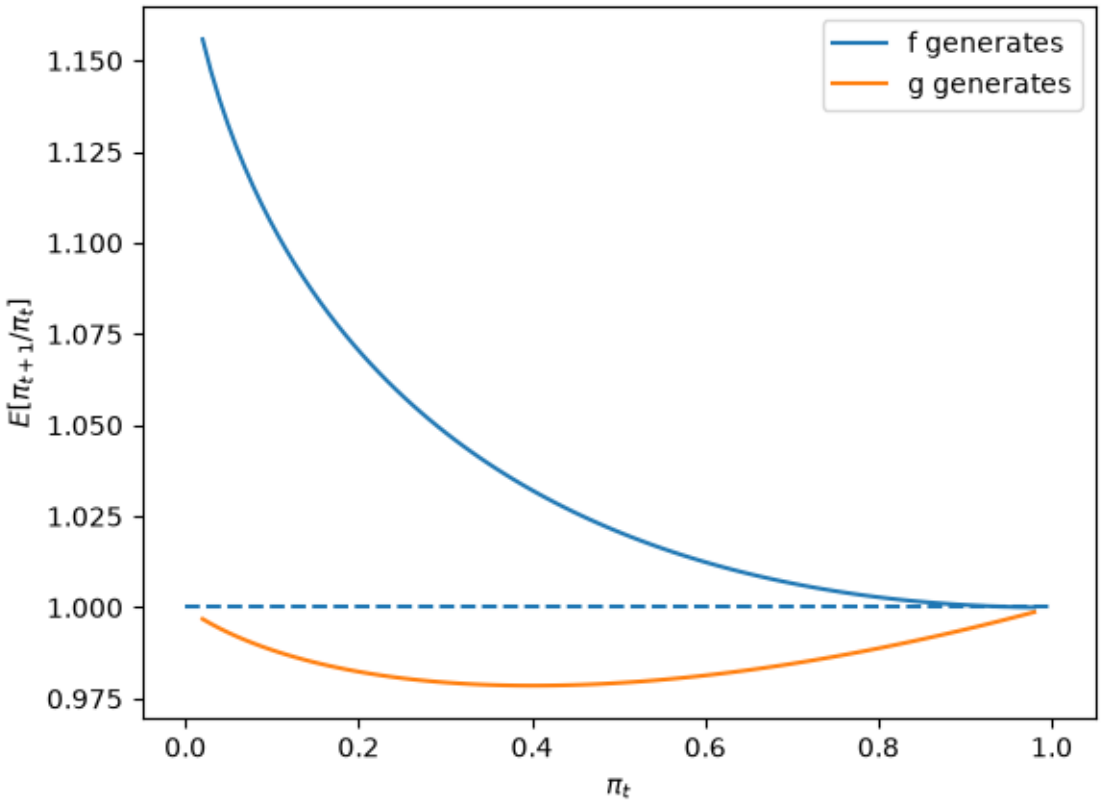
```
expected_ratio(F_a=3, F_b=1.2)
```



The above graph says that π_t is inert and remains at its initial value.

Finally, let's look at a case in which f and g are neither very different nor identical, in particular one in which $F_a = 2, F_b = 1$ and $G_a = 3, G_b = 1.2$.

```
expected_ratio(F_a=2, F_b=1, G_a=3, G_b=1.2)
```



28.9 Sequels

We'll apply and dig deeper into some of the ideas presented in this lecture:

- *this lecture* describes **likelihood ratio processes** and their role in frequentist and Bayesian statistical theories
- *this lecture* studies whether a World War II US Navy Captain's hunch that a (frequentist) decision rule that the Navy had told him to use was inferior to a sequential rule that Abraham Wald had not yet designed.

CHAPTER

TWENTYNINE

LIKELIHOOD RATIO PROCESSES AND BAYESIAN LEARNING

29.1 Overview

This lecture describes the role that **likelihood ratio processes** play in **Bayesian learning**.

As in *this lecture*, we'll use a simple statistical setting from *this lecture*.

We'll focus on how a likelihood ratio process and a **prior** probability determine a **posterior** probability.

We'll derive a convenient recursion for today's posterior as a function of yesterday's posterior and today's multiplicative increment to a likelihood process.

We'll also present a useful generalization of that formula that represents today's posterior in terms of an initial prior and today's realization of the likelihood ratio process.

We'll study how, at least in our setting, a Bayesian eventually learns the probability distribution that generates the data, an outcome that rests on the asymptotic behavior of likelihood ratio processes studied in *this lecture*.

We'll also drill down into the psychology of our Bayesian learner and study dynamics under his subjective beliefs.

This lecture provides technical results that underly outcomes to be studied in *this lecture* and *this lecture* and *this lecture*.

We'll begin by loading some Python modules.

```
import matplotlib.pyplot as plt
import numpy as np
from numba import vectorize, jit, prange
from math import gamma
import pandas as pd
from scipy.integrate import quad


import seaborn as sns
colors = sns.color_palette()

@jit
def set_seed():
    np.random.seed(142857)
set_seed()
```

29.2 The Setting

We begin by reviewing the setting in *this lecture*, which we adopt here too.

A nonnegative random variable W has one of two probability density functions, either f or g.

Before the beginning of time, nature once and for all decides whether she will draw a sequence of IID draws from f or from g.

We will sometimes let q be the density that nature chose once and for all, so that q is either f or g, permanently.

Nature knows which density it permanently draws from, but we the observers do not.

We do know both f and g, but we don't know which density nature chose.

But we want to know.

To do that, we use observations.

We observe a sequence $\{w_t\}_{t=1}^T$ of T IID draws from either f or g.

We want to use these observations to infer whether nature chose f or g.

A **likelihood ratio process** is a useful tool for this task.

To begin, we define the key component of a likelihood ratio process, namely, the time t likelihood ratio as the random variable

$$\ell(w_t) = \frac{f(w_t)}{g(w_t)}, \quad t \geq 1.$$

We assume that f and g both put positive probabilities on the same intervals of possible realizations of the random variable W.

That means that under the g density, $\ell(w_t) = \frac{f(w_t)}{g(w_t)}$ is evidently a nonnegative random variable with mean 1.

A **likelihood ratio process** for sequence $\{w_t\}_{t=1}^\infty$ is defined as

$$L(w^t) = \prod_{i=1}^t \ell(w_i),$$

where $w^t = \{w_1, \ldots, w_t\}$ is a history of observations up to and including time t.

Sometimes for shorthand we'll write

$$L_t = L(w^t) = \frac{f(w^t)}{g(w^t)}$$

where we use the conventions that $f(w^t) = f(w_1)f(w_2)\ldots f(w_t)$ and $g(w^t) = g(w_1)g(w_2)\ldots g(w_t)$.

Notice that the likelihood process satisfies the **recursion** or **multiplicative decomposition**

$$L(w^t) = \ell(w_t)L(w^{t-1}).$$

The likelihood ratio and its logarithm are key tools for making inferences using a classic frequentist approach due to Neyman and Pearson [Neyman and Pearson, 1933].

We'll again deploy the following Python code from *this lecture* that evaluates f and g as two different beta distributions, then computes and simulates an associated likelihood ratio process by generating a sequence w^t from *some* probability distribution, for example, a sequence of IID draws from g.

```
# Parameters in the two beta distributions.
F_a, F_b = 1, 1
G_a, G_b = 3, 1.2

@vectorize
def p(x, a, b):
    r = gamma(a + b) / (gamma(a) * gamma(b))
    return r * x** (a-1) * (1 - x) ** (b-1)

# The two density functions.
f = jit(lambda x: p(x, F_a, F_b))
g = jit(lambda x: p(x, G_a, G_b))
```

```
@jit
def simulate(a, b, T=50, N=500):
    '''
    Generate N sets of T observations of the likelihood ratio,
    return as N x T matrix.

    '''

    l_arr = np.empty((N, T))

    for i in range(N):

        for j in range(T):
            w = np.random.beta(a, b)
            l_arr[i, j] = f(w) / g(w)

    return l_arr
```

We'll also use the following Python code to prepare some informative simulations

```
l_arr_g = simulate(G_a, G_b, N=50000)
l_seq_g = np.cumprod(l_arr_g, axis=1)
```

```
l_arr_f = simulate(F_a, F_b, N=50000)
l_seq_f = np.cumprod(l_arr_f, axis=1)
```

29.3 Likelihood Ratio Processes and Bayes' Law

Let $\pi_0 \in [0,1]$ be a Bayesian statistician's prior probability that nature generates w^t as a sequence of i.i.d. draws from distribution f.

- here "probability" is to be interpreted as a way to summarize or express a subjective opinion
- it does **not** mean an anticipated relative frequency as sample size grows without limit

Let π_{t+1} be a Bayesian posterior probability defined as

$$\pi_{t+1} = \text{Prob}(q = f | w^{t+1}) \tag{29.1}$$

The likelihood ratio process is a principal actor in the formula that governs the evolution of the posterior probability π_t, an instance of **Bayes' Law**.

Let's derive a couple of formulas for π_{t+1}, one in terms of likelihood ratio $l(w_t)$, the other in terms of $L(w^t)$.

To begin, we use the notational conventions

- $f(w^{t+1}) \equiv f(w_1)f(w_2)\cdots f(w_{t+1})$
- $g(w^{t+1}) \equiv g(w_1)g(w_2)\cdots g(w_{t+1})$
- $\pi_0 = \text{Prob}(q = f|\emptyset)$
- $\pi_t = \text{Prob}(q = f|w^t)$

Here the symbol $\emptyset$ means "empty set" or "no data".

With no data in hand, our Bayesian statistician thinks that the probability density of the sequence w^{t+1} is

$$\text{Prob}(w^{t+1}|\emptyset) = \pi_0 f(w^{t+1}) + (1-\pi_0)g(w^{t+1})$$

Laws of probability say that the joint distribution $\text{Prob}(AB)$ of events A and B are connected to the conditional distributions $\text{Prob}(A|B)$ and $\text{Prob}(B|A)$ by

$$\text{Prob}(AB) = \text{Prob}(A|B)\text{Prob}(B) = \text{Prob}(B|A)\text{Prob}(A). \tag{29.2}$$

We are interested in events

$$A = \{q = f\}, \quad B = \{w^{t+1}\},$$

where braces $\{\cdot\}$ are our shorthand for "event".

So in our setting, probability laws (29.2) imply that

$$\text{Prob}(q = f|w^{t+1})\text{Prob}(w^{t+1}|\emptyset) = \text{Prob}(w^{t+1}|q = f)\text{Prob}(q = f|\emptyset)$$

or

$$\pi_{t+1}\left[\pi_0 f(w^{t+1}) + (1-\pi_0)g(w^{t+1})\right] = f(w^{t+1})\pi_0$$

or

$$\pi_{t+1} = \frac{f(w^{t+1})\pi_0}{\pi_0 f(w^{t+1}) + (1-\pi_0)g(w^{t+1})}$$

Dividing both the numerator and the denominator on the right side of the above equation by $g(w^{t+1})$ implies

$$\pi_{t+1} = \frac{\pi_0 L\left(w^{t+1}\right)}{\pi_0 L\left(w^{t+1}\right) + 1 - \pi_0}. \tag{29.3}$$

Formula (29.3) can be regarded as a one step revision of prior probability π_0 after seeing the batch of data $\{w_i\}_{i=1}^{t+1}$.

Formula (29.3) shows the key role that the likelihood ratio process $L\left(w^{t+1}\right)$ plays in determining the posterior probability π_{t+1}.

Formula (29.3) is the foundation for the insight that, because of how the likelihood ratio process behaves as $t \to +\infty$, the likelihood ratio process dominates the initial prior π_0 in determining the limiting behavior of π_t.

29.3.1 A recursive formula

We can use a similar line of argument to get a recursive version of formula (29.3).

The laws of probability imply

$$\text{Prob}(q = f|w^{t+1}) = \frac{\text{Prob}(q = f|w^t)f(w_{t+1})}{\text{Prob}(q = f|w^t)f(w_{t+1}) + (1 - \text{Prob}(q = f|w^t))g(w_{t+1})}$$

or

$$\pi_{t+1} = \frac{\pi_t f(w_{t+1})}{\pi_t f(w_{t+1}) + (1-\pi_t) g(w_{t+1})} \tag{29.4}$$

Evidently, the above equation asserts that

$$\text{Prob}(q = f | w^{t+1}) = \frac{\text{Prob}(q = f | w^t) f(w_{t+1})}{\text{Prob}(w_{t+1})}$$

Dividing both the numerator and the denominator on the right side of the equation (29.4) by $g(w_{t+1})$ implies the recursion

$$\pi_{t+1} = \frac{\pi_t l_t(w_{t+1})}{\pi_t l_t(w_{t+1}) + 1 - \pi_t} \tag{29.5}$$

with π_0 being a Bayesian prior probability that $q = f$, i.e., a personal or subjective belief about q based on our having seen no data.

Formula (29.3) can be deduced by iterating on equation (29.5).

Below we define a Python function that updates belief π using likelihood ratio ℓ according to recursion (29.5)

```
@jit
def update(π, l):
    "Update π using likelihood l"

    # Update belief
    π = π * l / (π * l + 1 - π)

    return π
```

As mentioned above, formula (29.3) shows the key role that the likelihood ratio process $L\left(w^{t+1}\right)$ plays in determining the posterior probability π_{t+1}.

As $t \to +\infty$, the likelihood ratio process dominates the initial prior π_0 in determining the limiting behavior of π_t.

To illustrate this insight, below we will plot graphs showing **one** simulated path of the likelihood ratio process L_t along with two paths of π_t that are associated with the *same* realization of the likelihood ratio process but *different* initial prior probabilities π_0.

First, we tell Python two values of π_0.

```
π1, π2 = 0.2, 0.8
```

Next we generate paths of the likelihood ratio process L_t and the posterior π_t for a history of IID draws from density f.

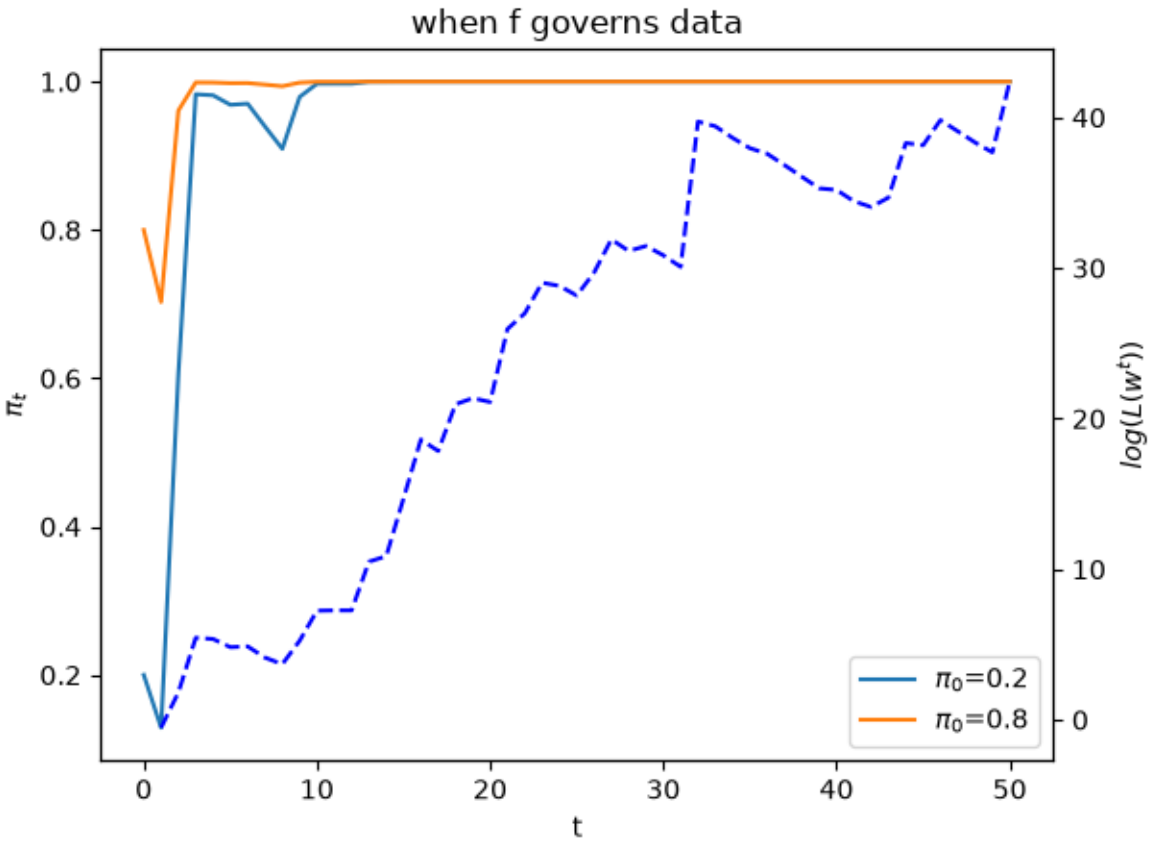
```
T = l_arr_f.shape[1]
π_seq_f = np.empty((2, T+1))
π_seq_f[:, 0] = π1, π2

for t in range(T):
    for i in range(2):
        π_seq_f[i, t+1] = update(π_seq_f[i, t], l_arr_f[0, t])
```

```
fig, ax1 = plt.subplots()

for i in range(2):
    ax1.plot(range(T+1), π_seq_f[i, :], label=fr"$\pi_0$={π_seq_f[i, 0]}")
```

(continues on next page)

(continued from previous page)

```
ax1.set_ylabel(r"$\pi_t$")
ax1.set_xlabel("t")
ax1.legend()
ax1.set_title("when f governs data")

ax2 = ax1.twinx()
ax2.plot(range(1, T+1), np.log(l_seq_f[0, :]), '--', color='b')
ax2.set_ylabel("$log(L(w^{t}))$")

plt.show()
```



The dotted line in the graph above records the logarithm of the likelihood ratio process $\log L(w^t)$.

Please note that there are two different scales on the y axis.

Now let's study what happens when the history consists of IID draws from density g

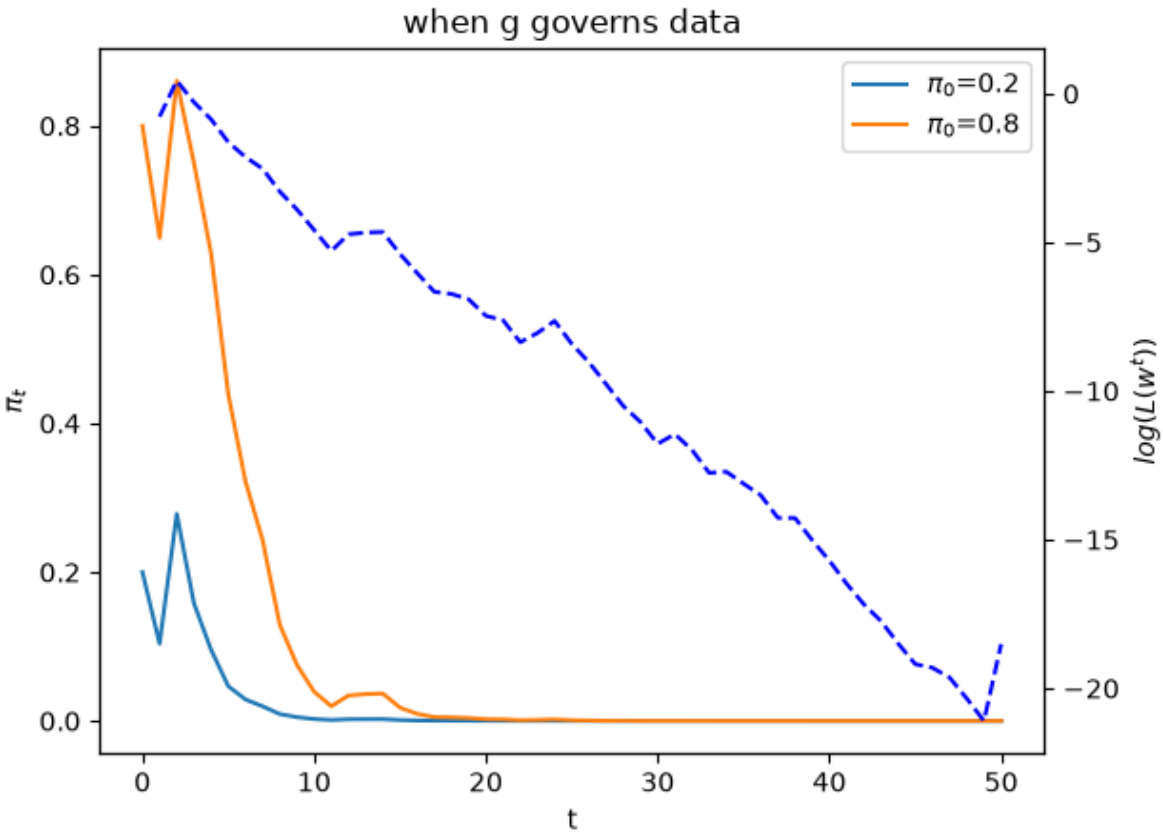
```
T = l_arr_g.shape[1]
π_seq_g = np.empty((2, T+1))
π_seq_g[:, 0] = π1, π2

for t in range(T):
    for i in range(2):
        π_seq_g[i, t+1] = update(π_seq_g[i, t], l_arr_g[0, t])
```

```
fig, ax1 = plt.subplots()
```

(continues on next page)

(continued from previous page)

```
for i in range(2):
    ax1.plot(range(T+1), π_seq_g[i, :], label=fr"$\pi_0$={π_seq_g[i, 0]}")

ax1.set_ylabel(r"$\pi_t$")
ax1.set_xlabel("t")
ax1.legend()
ax1.set_title("when g governs data")

ax2 = ax1.twinx()
ax2.plot(range(1, T+1), np.log(l_seq_g[0, :]), '--', color='b')
ax2.set_ylabel("$log(L(w^{t}))$")

plt.show()
```



Below we offer Python code that verifies that nature chose permanently to draw from density f.

```
π_seq = np.empty((2, T+1))
π_seq[:, 0] = π1, π2

for i in range(2):
    πL = π_seq[i, 0] * l_seq_f[0, :]
    π_seq[i, 1:] = πL / (πL + 1 - π_seq[i, 0])
```

```
np.abs(π_seq - π_seq_f).max() < 1e-10
```

```
np.True_
```

We thus conclude that the likelihood ratio process is a key ingredient of the formula (29.3) for a Bayesian's posterior probability that nature has drawn history w^t as repeated draws from density f.

29.4 Another timing protocol

Let's study how the posterior probability $\pi_t = \text{Prob}(q = f|w^t)$ behaves when nature generates the history $w^t = \{w_1, w_2, \ldots, w_t\}$ under a different timing protocol.

Until now we assumed that before time 1 nature somehow chose to draw w^t as an iid sequence from **either** f **or** g.

Nature's decision about whether to draw from f or g was thus **permanent**.

We now assume a different timing protocol in which before **each period** $t = 1, 2, \ldots$ nature

- flips an x-weighted coin, then
- draws from f if it has drawn a "head"
- draws from g if it has drawn a "tail".

Under this timing protocol, nature draws permanently from **neither** f **nor** g, so a statistician who thinks that nature is drawing i.i.d. draws **permanently** from one of them is mistaken.

- in truth, nature actually draws **permanently** from an x-mixture of f and g – a distribution that is neither f nor g when $x \in (0, 1)$

Thus, the Bayesian prior π_0 and the sequence of posterior probabilities described by equation (29.3) should **not** be interpreted as the statistician's opinion about the mixing parameter x under the alternative timing protocol in which nature draws from an x-mixture of f and g.

This is clear when we remember the definition of π_t in equation (29.1), which for convenience we repeat here:

$$\pi_{t+1} = \text{Prob}(q = f|w^{t+1})$$

Let's write some Python code to study how π_t behaves when nature actually generates data as i.i.d. draws from neither f nor from g but instead as i.i.d. draws from an x-mixture of two beta distributions.

Note

This is a situation in which the statistician's model is misspecified, so we should anticipate that a Kullback-Liebler divergence with respect to an x-mixture distribution will shape outcomes.

We can study how π_t would behave for various values of nature's mixing probability x.

First, let's create a function to simulate data under the mixture timing protocol:

```
@jit
def simulate_mixture_path(x_true, T):
    """
    Simulate T observations under mixture timing protocol.
    """
    w = np.empty(T)
    for t in range(T):
        if np.random.rand() < x_true:
```

(continues on next page)

(continued from previous page)

```
            w[t] = np.random.beta(F_a, F_b)
        else:
            w[t] = np.random.beta(G_a, G_b)
    return w
```

Let's generate a sequence of observations from this mixture model with a true mixing probability of $x = 0.5$.

We will first use this sequence to study how π_t behaves.

Note

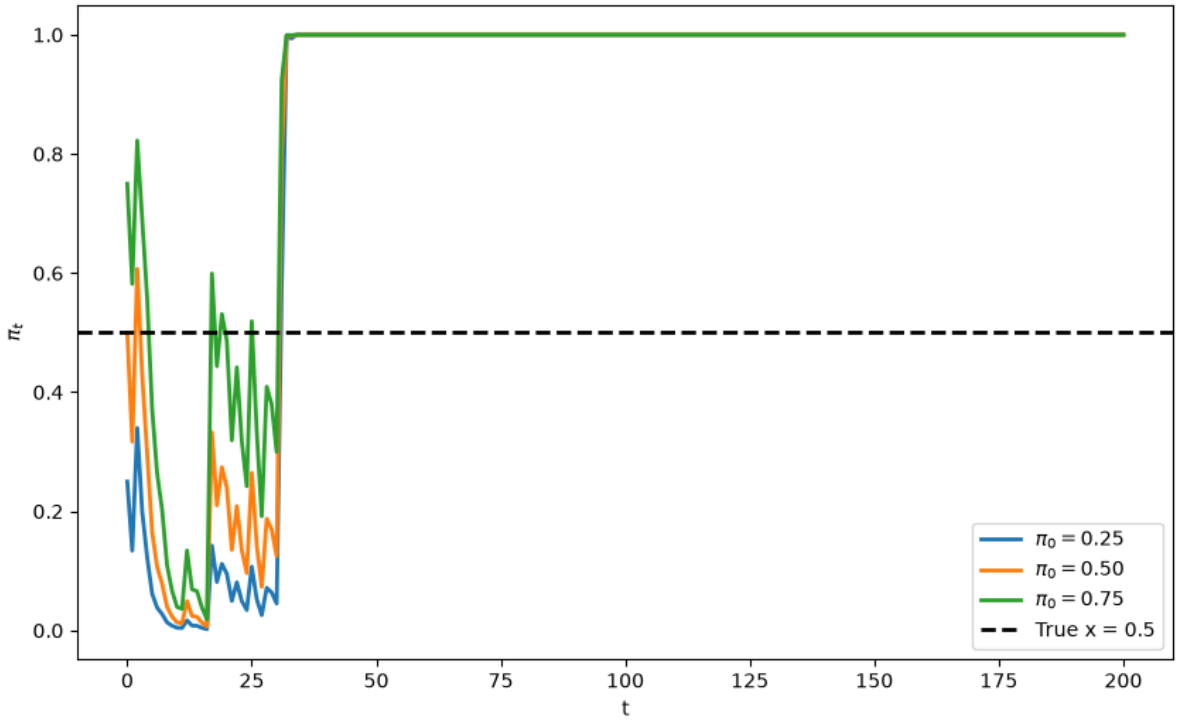
Later, we can use it to study how a statistician who knows that nature generates data from an x-mixture of f and g could construct maximum likelihood or Bayesian estimators of x along with the free parameters of f and g.

```
x_true = 0.5
T_mix = 200

# Three different priors with means 0.25, 0.5, 0.75
prior_params = [(1, 3), (1, 1), (3, 1)]
prior_means = [a/(a+b) for a, b in prior_params]

# Generate one path of observations from the mixture
set_seed()
w_mix = simulate_mixture_path(x_true, T_mix)
```

29.4.1 Behavior of π_t under wrong model

Let's study how the posterior probability π_t that nature permanently draws from f behaves when data are actually generated by an x-mixture of f and g.

```
fig, ax = plt.subplots(figsize=(10, 6))
T_plot = 200

for i, mean0 in enumerate(prior_means):
    π_wrong = np.empty(T_plot + 1)
    π_wrong[0] = mean0

    # Compute likelihood ratios for the mixture data
    for t in range(T_plot):
        l_t = f(w_mix[t]) / g(w_mix[t])
        π_wrong[t + 1] = update(π_wrong[t], l_t)

    ax.plot(range(T_plot + 1), π_wrong,
            label=fr'$\pi_0 = ${mean0:.2f}',
            color=colors[i], linewidth=2)

ax.axhline(y=x_true, color='black', linestyle='--',
           label=f'True x = {x_true}', linewidth=2)
ax.set_xlabel('t')
ax.set_ylabel(r'$\pi_t$')
ax.legend()
plt.show()
```



Evidently, π_t converges to 1.

This indicates that the model concludes that the data is generated by f.

Why does this happen?

Given $x = 0.5$, the data generating process is a mixture of f and g: $m(w) = \frac{1}{2}f(w) + \frac{1}{2}g(w)$.

Let's check the *KL divergence* of the mixture distribution m from both f and g.

```
def compute_KL(f, g):
    """
    Compute KL divergence KL(f, g)
    """
    integrand = lambda w: f(w) * np.log(f(w) / g(w))
    val, _ = quad(integrand, 1e-5, 1-1e-5)
    return val


def compute_div_m(f, g):
    """
    Compute Jensen-Shannon divergence
    """
    def m(w):
        return 0.5 * (f(w) + g(w))

    return compute_KL(m, f), compute_KL(m, g)


KL_f, KL_g = compute_div_m(f, g)

print(f'KL(m, f) = {KL_f:.3f}\nKL(m, g) = {KL_g:.3f}')
```

```
KL(m, f) = 0.073
KL(m, g) = 0.281
```

Since $KL(m, f) < KL(m, g)$, f is "closer" to the mixture distribution m.

Hence by our discussion on KL divergence and likelihood ratio process in *Likelihood Ratio Processes*, $\log(L_t) \to \infty$ as $t \to \infty$.

Now looking back to the key equation (29.3).

Consider the function

$$h(z) = \frac{\pi_0 z}{\pi_0 z + 1 - \pi_0}.$$

The limit $\lim_{z \to \infty} h(z)$ is 1.

Hence $\pi_t \to 1$ as $t \to \infty$ for any $\pi_0 \in (0, 1)$.

This explains what we observed in the plot above.

But how can we learn the true mixing parameter x?

This topic is taken up in *Incorrect Models*.

We explore how to learn the true mixing parameter x in the exercise of *Incorrect Models*.

29.5 Behavior of Posterior Probability $\{\pi_t\}$ Under Subjective Probability Distribution

We'll end this lecture by briefly studying what our Bayesian learner expects to learn under the subjective beliefs π_t cranked out by Bayes' law.

This will provide us with some perspective on our application of Bayes's law as a theory of learning.

As we shall see, at each time t, the Bayesian learner knows that he will be surprised.

But he expects that new information will not lead him to change his beliefs.

And it won't on average under his subjective beliefs.

We'll continue with our setting in which a McCall worker knows that successive draws of his wage are drawn from either F or G, but does not know which of these two distributions nature has drawn once-and-for-all before time 0.

We'll review and reiterate and rearrange some formulas that we have encountered above and in associated lectures.

The worker's initial beliefs induce a joint probability distribution over a potentially infinite sequence of draws $w_0, w_1, \ldots$.

Bayes' law is simply an application of laws of probability to compute the conditional distribution of the tth draw w_t conditional on $[w_0, \ldots, w_{t-1}]$.

After our worker puts a subjective probability π_{-1} on nature having selected distribution F, we have in effect assumed from the start that the decision maker **knows** the joint distribution for the process $\{w_t\}_{t=0}$.

We assume that the worker also knows the laws of probability theory.

A respectable view is that Bayes' law is less a theory of learning than a statement about the consequences of information inflows for a decision maker who thinks he knows the truth (i.e., a joint probability distribution) from the beginning.

29.5.1 Mechanical details again

At time 0 **before** drawing a wage offer, the worker attaches probability $\pi_{-1} \in (0,1)$ to the distribution being F.

Before drawing a wage at time 0, the worker thus believes that the density of w_0 is

$$h(w_0;\pi_{-1}) = \pi_{-1} f(w_0) + (1-\pi_{-1}) g(w_0).$$

Let $a \in \{f, g\}$ be an index that indicates whether nature chose permanently to draw from distribution f or from distribution g.

After drawing w_0, the worker uses Bayes' law to deduce that the posterior probability $\pi_0 = \text{Prob}(a = f|w_0)$ that the density is $f(w)$ is

$$\pi_0 = \frac{\pi_{-1} f(w_0)}{\pi_{-1} f(w_0) + (1-\pi_{-1}) g(w_0)}.$$

More generally, after making the tth draw and having observed $w_t, w_{t-1}, \ldots, w_0$, the worker believes that the probability that w_{t+1} is being drawn from distribution F is

$$\pi_t = \pi_t(w_t|\pi_{t-1}) \equiv \frac{\pi_{t-1} f(w_t)/g(w_t)}{\pi_{t-1} f(w_t)/g(w_t) + (1-\pi_{t-1})} \tag{29.6}$$

or

$$\pi_t = \frac{\pi_{t-1} l_t(w_t)}{\pi_{t-1} l_t(w_t) + 1 - \pi_{t-1}}$$

and that the density of w_{t+1} conditional on $w_t, w_{t-1}, \ldots, w_0$ is

$$h(w_{t+1};\pi_t) = \pi_t f(w_{t+1}) + (1-\pi_t) g(w_{t+1}).$$

Notice that

$$\begin{aligned} E(\pi_t|\pi_{t-1}) &= \int \Big[\frac{\pi_{t-1} f(w)}{\pi_{t-1} f(w) + (1-\pi_{t-1}) g(w)}\Big] \Big[\pi_{t-1} f(w) + (1-\pi_{t-1}) g(w)\Big] dw \\ &= \pi_{t-1} \int f(w) dw \\ &= \pi_{t-1}, \end{aligned}$$

so that the process π_t is a **martingale**.

Indeed, it is a **bounded martingale** because each π_t, being a probability, is between 0 and 1.

In the first line in the above string of equalities, the term in the first set of brackets is just π_t as a function of w_t, while the term in the second set of brackets is the density of w_t conditional on $w_{t-1}, \ldots, w_0$ or equivalently conditional on the *sufficient statistic* π_{t-1} for $w_{t-1}, \ldots, w_0$.

Notice that here we are computing $E(\pi_t|\pi_{t-1})$ under the **subjective** density described in the second term in brackets.

Because $\{\pi_t\}$ is a bounded martingale sequence, it follows from the **martingale convergence theorem** that π_t converges almost surely to a random variable in $[0,1]$.

Practically, this means that probability one is attached to sample paths $\{\pi_t\}_{t=0}^{\infty}$ that converge.

According to the theorem, different sample paths can converge to different limiting values.

Thus, let $\{\pi_t(\omega)\}_{t=0}^{\infty}$ denote a particular sample path indexed by a particular $\omega \in \Omega$.

We can think of nature as drawing an $\omega \in \Omega$ from a probability distribution ProbΩ and then generating a single realization (or *simulation*) $\{\pi_t(\omega)\}_{t=0}^{\infty}$ of the process.

The limit points of $\{\pi_t(\omega)\}_{t=0}^{\infty}$ as $t \to +\infty$ are realizations of a random variable that is swept out as we sample ω from Ω and construct repeated draws of $\{\pi_t(\omega)\}_{t=0}^{\infty}$.

By staring at law of motion (29.5) or (29.6) , we can figure out some things about the probability distribution of the limit points

$$\pi_\infty(\omega) = \lim_{t \to +\infty} \pi_t(\omega).$$

Evidently, since the likelihood ratio $\ell(w_t)$ differs from 1 when $f \neq g$, as we have assumed, the only possible fixed points of (29.6) are

$$\pi_\infty(\omega) = 1$$

and

$$\pi_\infty(\omega) = 0$$

Thus, for some realizations, $\lim_{\to+\infty} \pi_t(\omega) = 1$ while for other realizations, $\lim_{\to+\infty} \pi_t(\omega) = 0$.

Now let's remember that $\{\pi_t\}_{t=0}^{\infty}$ is a martingale and apply the law of iterated expectations.

The law of iterated expectations implies

$$E_t \pi_{t+j} = \pi_t$$

and in particular

$$E_{-1} \pi_{t+j} = \pi_{-1}.$$

Applying the above formula to π_∞, we obtain

$$E_{-1} \pi_\infty(\omega) = \pi_{-1}$$

where the mathematical expectation E_{-1} here is taken with respect to the probability measure $\text{Prob}(\Omega)$.

Since the only two values that $\pi_\infty(\omega)$ can take are 1 and 0, we know that for some $\lambda \in [0, 1]$

$$\text{Prob}\Big(\pi_\infty(\omega) = 1\Big) = \lambda, \quad \text{Prob}\Big(\pi_\infty(\omega) = 0\Big) = 1 - \lambda$$

and consequently that

$$E_{-1} \pi_\infty(\omega) = \lambda \cdot 1 + (1 - \lambda) \cdot 0 = \lambda$$

Combining this equation with equation (20), we deduce that the probability that $\text{Prob}(\Omega)$ attaches to $\pi_\infty(\omega)$ being 1 must be π_{-1}.

Thus, under the worker's subjective distribution, π_{-1} of the sample paths of $\{\pi_t\}$ will converge pointwise to 1 and $1 - \pi_{-1}$ of the sample paths will converge pointwise to 0.

29.5.2 Some simulations

Let's watch the martingale convergence theorem at work in some simulations of our learning model under the worker's subjective distribution.

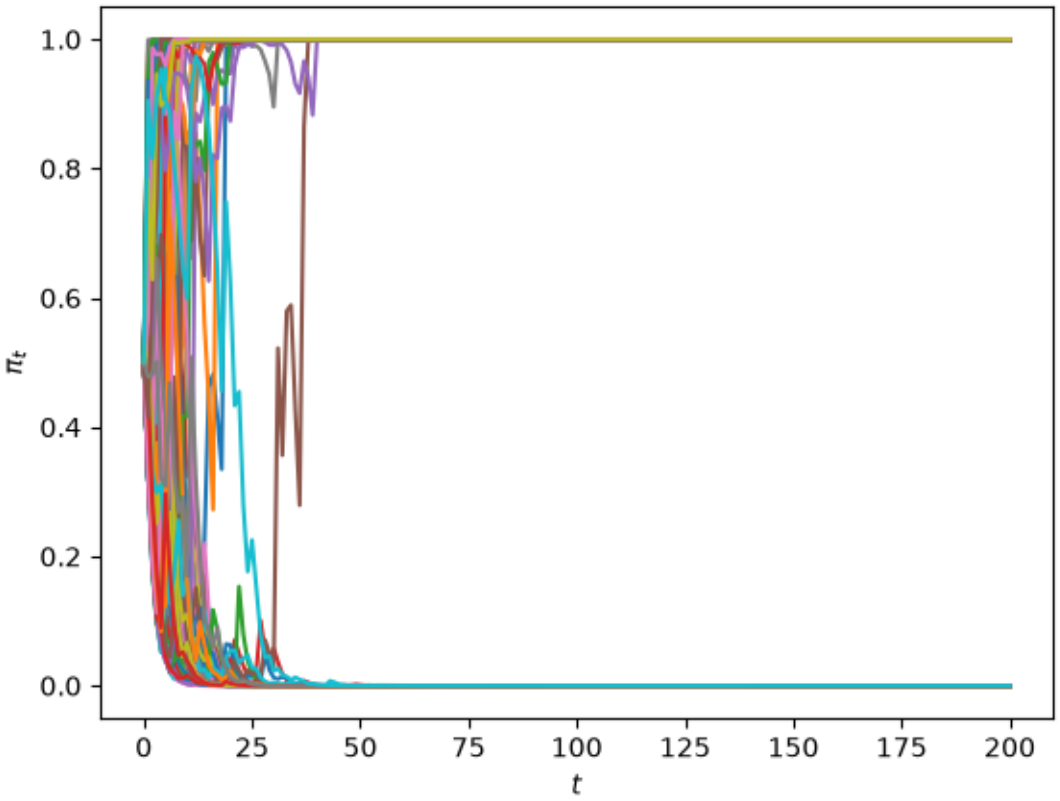
Let us simulate $\{\pi_t\}_{t=0}^{T}$, $\{w_t\}_{t=0}^{T}$ paths where for each $t \geq 0$, w_t is drawn from the subjective distribution

$$\pi_{t-1} f(w_t) + (1 - \pi_{t-1}) g(w_t)$$

We'll plot a large sample of paths.

```
@jit
def martingale_simulate(π0, N=5000, T=200):

    π_path = np.empty((N,T+1))
    w_path = np.empty((N,T))
    π_path[:,0] = π0

    for n in range(N):
        π = π0
        for t in range(T):
            # draw w
            if np.random.rand() <= π:
                w = np.random.beta(F_a, F_b)
            else:
                w = np.random.beta(G_a, G_b)
            π = π*f(w)/g(w)/(π*f(w)/g(w) + 1 - π)
            π_path[n,t+1] = π
            w_path[n,t] = w

    return π_path, w_path

def fraction_0_1(π0, N, T, decimals):

    π_path, w_path = martingale_simulate(π0, N=N, T=T)
    values, counts = np.unique(np.round(π_path[:,-1], decimals=decimals), return_
↪counts=True)
    return values, counts

def create_table(π0s, N=10000, T=500, decimals=2):

    outcomes = []
    for π0 in π0s:
        values, counts = fraction_0_1(π0, N=N, T=T, decimals=decimals)
        freq = counts/N
        outcomes.append(dict(zip(values, freq)))
    table = pd.DataFrame(outcomes).sort_index(axis=1).fillna(0)
    table.index = π0s
    return table

# simulate
T = 200
π0 = .5

π_path, w_path = martingale_simulate(π0=π0, T=T, N=10000)
```

```
fig, ax = plt.subplots()
for i in range(100):
    ax.plot(range(T+1), π_path[i, :])

ax.set_xlabel('$t$')
ax.set_ylabel(r'$\pi_t$')
plt.show()
```

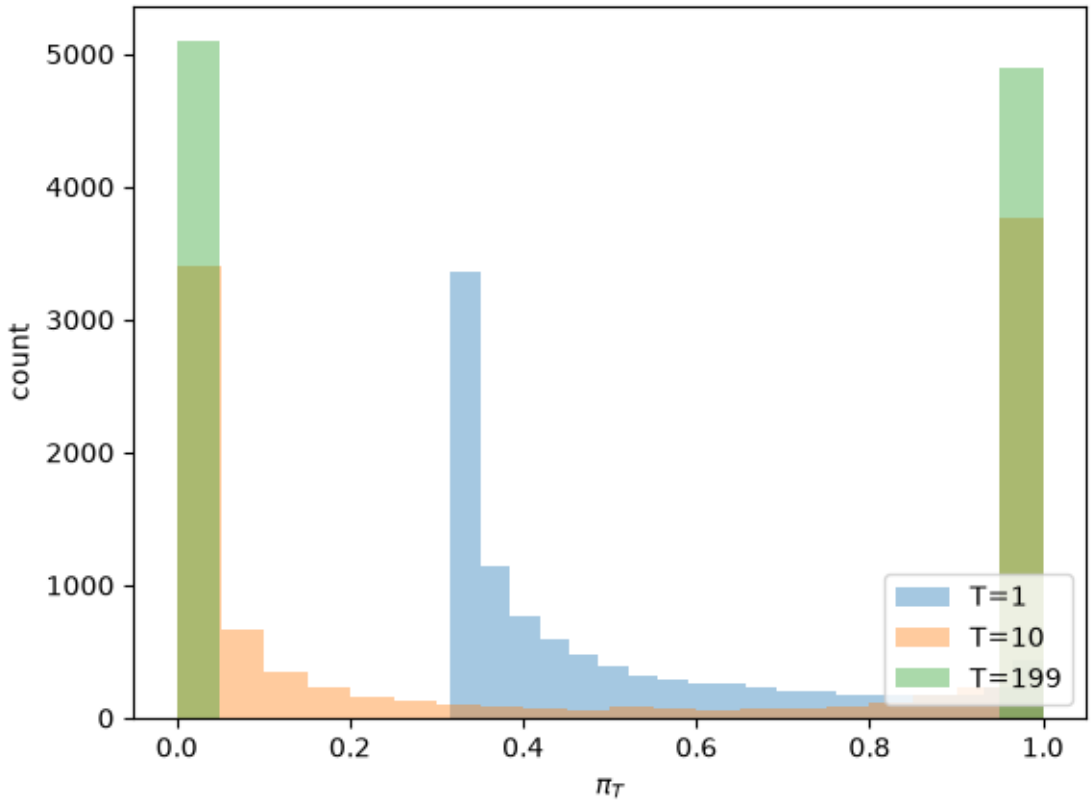


The above graph indicates that

- each of paths converges
- some of the paths converge to 1
- some of the paths converge to 0
- none of the paths converge to a limit point not equal to 0 or 1

Convergence actually occurs pretty fast, as the following graph of the cross-ensemble distribution of π_t for various small t's indicates.

```
fig, ax = plt.subplots()
for t in [1, 10, T-1]:
    ax.hist(π_path[:,t], bins=20, alpha=0.4, label=f'T={t}')

ax.set_ylabel('count')
ax.set_xlabel(r'$\pi_T$')
ax.legend(loc='lower right')
plt.show()
```



Evidently, by $t = 199$, π_t has converged to either 0 or 1.

The fraction of paths that have converged to 1 is .5

The fractions of paths that have converged to 0 is also .5.

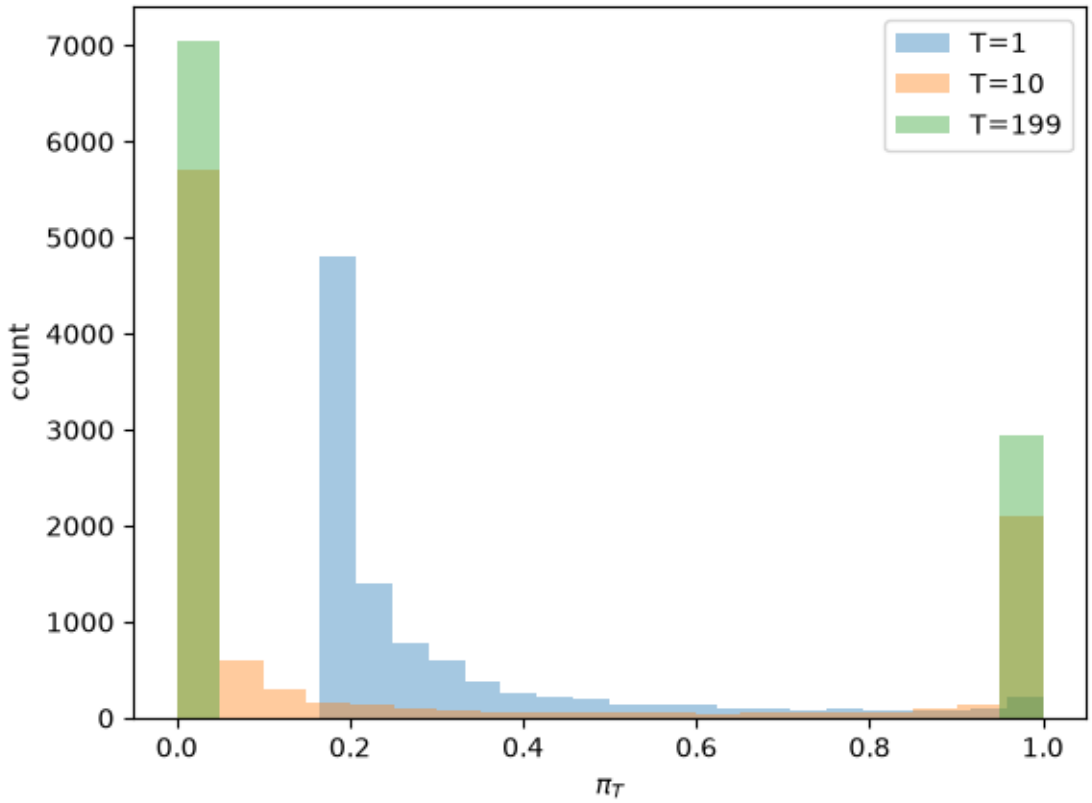
Does the fraction .5 ring a bell?

Yes, it does: it equals the value of $\pi_0 = .5$ that we used to generate each sequence in the ensemble.

So let's change π_0 to .3 and watch what happens to the distribution of the ensemble of π_t's for various t's.

```
# simulate
T = 200
π0 = .3

π_path3, w_path3 = martingale_simulate(π0=π0, T=T, N=10000)
```

```
fig, ax = plt.subplots()
for t in [1, 10, T-1]:
    ax.hist(π_path3[:,t], bins=20, alpha=0.4, label=f'T={t}')

ax.set_ylabel('count')
ax.set_xlabel(r'$\pi_T$')
ax.legend(loc='upper right')
plt.show()
```

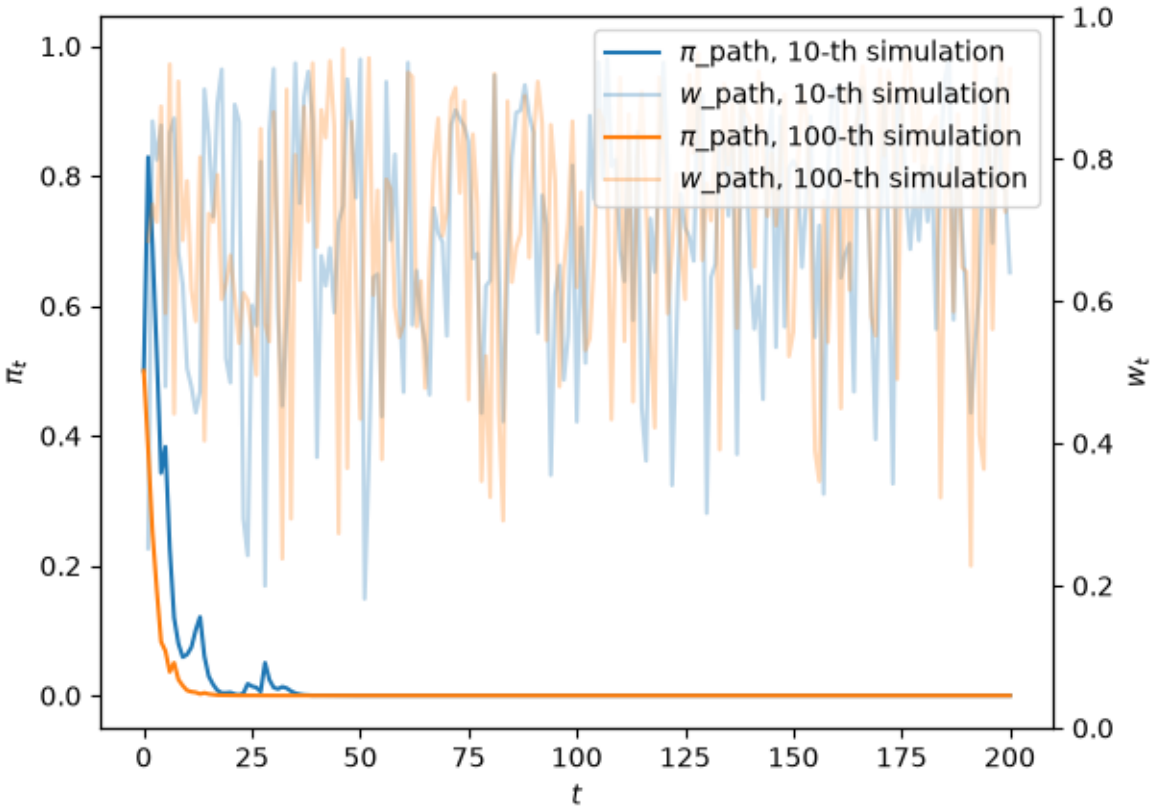


For the preceding ensemble that assumed $\pi_0 = .5$, the following graph shows two paths of w_t's and the π_t sequences that gave rise to them.

Notice that one of the paths involves systematically higher w_t's, outcomes that push π_t upward.

The luck of the draw early in a simulation pushes the subjective distribution to draw from F more frequently along a sample path, and this pushes π_t toward 0.

```
fig, ax = plt.subplots()
for i, j in enumerate([10, 100]):
    ax.plot(range(T+1), π_path[j,:], color=colors[i], label=fr'$\pi$_path, {j}-th␣
↪simulation')
    ax.plot(range(1,T+1), w_path[j,:], color=colors[i], label=fr'$w$_path, {j}-th␣
↪simulation', alpha=0.3)

ax.legend(loc='upper right')
ax.set_xlabel('$t$')
ax.set_ylabel(r'$\pi_t$')
ax2 = ax.twinx()
ax2.set_ylabel("$w_t$")
plt.show()
```



29.6 Initial Prior is Verified by Paths Drawn from Subjective Conditional Densities

Now let's use our Python code to generate a table that checks out our earlier claims about the probability distribution of the pointwise limits $\pi_\infty(\omega)$.

We'll use our simulations to generate a histogram of this distribution.

In the following table, the left column in bold face reports an assumed value of π_{-1}.

The second column reports the fraction of $N = 10000$ simulations for which π_t had converged to 0 at the terminal date $T = 500$ for each simulation.

The third column reports the fraction of $N = 10000$ simulations for which π_t had converged to 1 at the terminal date $T = 500$ for each simulation.

```
# create table
table = create_table(list(np.linspace(0,1,11)), N=10000, T=500)
table
```

```
        0.0     1.0
0.0  1.0000  0.0000
0.1  0.8984  0.1016
0.2  0.8000  0.2000
0.3  0.6981  0.3019
0.4  0.6004  0.3996
```

(continues on next page)

(continued from previous page)

```
0.5  0.4968  0.5032
0.6  0.3995  0.6005
0.7  0.3007  0.6993
0.8  0.2074  0.7926
0.9  0.0964  0.9036
1.0  0.0000  1.0000
```

The fraction of simulations for which π_t had converged to 1 is indeed always close to π_{-1}, as anticipated.

29.7 Drilling Down a Little Bit

To understand how the local dynamics of π_t behaves, it is enlightening to consult the variance of π_t conditional on π_{t-1}.

Under the subjective distribution this conditional variance is defined as

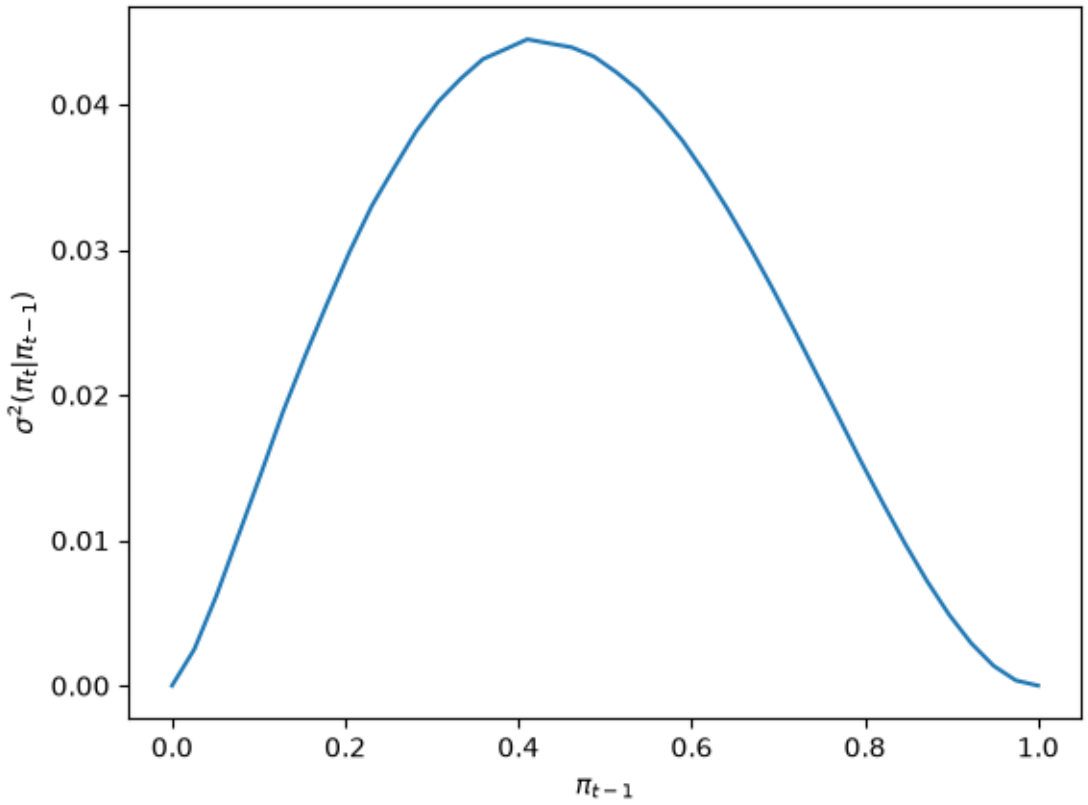
$$\sigma^2(\pi_t|\pi_{t-1}) = \int \Big[\frac{\pi_{t-1}f(w)}{\pi_{t-1}f(w) + (1-\pi_{t-1})g(w)} - \pi_{t-1}\Big]^2 \Big[\pi_{t-1}f(w) + (1-\pi_{t-1})g(w)\Big] dw$$

We can use a Monte Carlo simulation to approximate this conditional variance.

We approximate it for a grid of points $\pi_{t-1} \in [0, 1]$.

Then we'll plot it.

```
@jit
def compute_cond_var(pi, mc_size=int(1e6)):
    # create monte carlo draws
    mc_draws = np.zeros(mc_size)

    for i in prange(mc_size):
        if np.random.rand() <= pi:
            mc_draws[i] = np.random.beta(F_a, F_b)
        else:
            mc_draws[i] = np.random.beta(G_a, G_b)

    dev = pi*f(mc_draws)/(pi*f(mc_draws) + (1-pi)*g(mc_draws)) - pi
    return np.mean(dev**2)

pi_array = np.linspace(0, 1, 40)
cond_var_array = []

for pi in pi_array:
    cond_var_array.append(compute_cond_var(pi))

fig, ax = plt.subplots()
ax.plot(pi_array, cond_var_array)
ax.set_xlabel(r'$\pi_{t-1}$')
ax.set_ylabel(r'$\sigma^{2}(\pi_{t}\vert \pi_{t-1})$')
plt.show()
```



The shape of the conditional variance as a function of π_{t-1} is informative about the behavior of sample paths of $\{\pi_t\}$.

Notice how the conditional variance approaches 0 for π_{t-1} near either 0 or 1.

The conditional variance is nearly zero only when the agent is almost sure that w_t is drawn from F, or is almost sure it is drawn from G.

29.8 Related Lectures

This lecture has been devoted to building some useful infrastructure that will help us understand inferences that are the foundations of results described in *this lecture* and *this lecture* and *this lecture*.

CHAPTER

THIRTY

BLACKWELL'S THEOREM ON COMPARING EXPERIMENTS

Contents

- *Blackwell's Theorem on Comparing Experiments*
  - *Overview*
  - *Experiments and stochastic transformations*
  - *Three equivalent criteria*
  - *The main theorem*
  - *Kihlstrom's Bayesian interpretation*
  - *Simulating the Blackwell order with many states*
  - *The DeGroot uncertainty function*
  - *Connection to second-order stochastic dominance*
  - *Application 1: product quality information*
  - *Application 2: sequential experimental design*
  - *Summary*
  - *The Data Processing Inequality and Coarse-Graining*
  - *Relation to Bayesian likelihood-ratio learning*

30.1 Overview

This lecture studies *Blackwell's theorem* [Blackwell, 1951, Blackwell, 1953] on ranking statistical experiments.

Our presentation brings in findings from a Bayesian interpretation of Blackwell's theorem by Kihlstrom [1984].

Blackwell and Kihlstrom study statistical model-selection questions closely related to those encountered in this QuantEcon lecture *Likelihood Ratio Processes and Bayesian Learning*.

To appreciate the connection involved, it is helpful to appreciate how Blackwell's notion of an **experiment** is related to the concept of a "probability distribution" or "parameterized statistical model" appearing in *Likelihood Ratio Processes and Bayesian Learning*

Blackwell studies a situation in which a decision maker wants to know the value of a state s that lives in a space S.

For Blackwell, an **experiment** is a **conditional probability model** $\{\mu(\cdot \mid s) : s \in S\}$, i.e., a family of probability distributions that are conditioned by the same state $s \in S$.

We are free to interpret the "state" as a "parameter" or "parameter vector".

In a two-state case $S = \{s_1, s_2\}$, the two conditional densities $f(\cdot) = \mu(\cdot \mid s_1)$ and $g(\cdot) = \mu(\cdot \mid s_2)$ are the ones used repeatedly in our studies of classical hypothesis testing and Bayesian inference in this QuantEcon lecture *Likelihood Ratio Processes and Bayesian Learning* as well as several other lectures in this suite of QuantEcon lectures.

Kihlstrom [1984] interprets the question *which experiment is more informative?* as asking which conditional probability model allows a Bayesian decision maker with a prior over $\{s_1, s_2\}$ to gather higher expected utility.

We'll use the terms "signal" and "experiment" as synyomyms.

Thus, suppose that two signals, $\tilde{x}_\mu$ and $\tilde{x}_\nu$, are both informative about an unknown state $\tilde{s}$.

Signal μ is **at least as informative as** signal ν if every Bayesian decision maker can attain weakly higher expected utility with μ than with ν.

This economic criterion is equivalent to two statistical criteria:

- *Sufficiency* (Blackwell): $\tilde{x}_\nu$ can be generated from $\tilde{x}_\mu$ by an additional randomization.
- *Uncertainty reduction* (DeGroot [1962]): $\tilde{x}_\mu$ lowers expected uncertainty at least as much as $\tilde{x}_\nu$ for every concave uncertainty function.

Kihlstrom's formulation focuses on the *posterior distribution*.

More informative experiments generate posterior distributions that are more dispersed in convex order.

In the two-state case, this becomes a mean-preserving-spread comparison on $[0, 1]$, which can be checked with the integrated-CDF test used for second-order stochastic dominance.

The lecture proceeds as follows:

1. Set up notation and define experiments as Markov matrices.
2. Define stochastic transformations using Markov kernels.
3. State the three equivalent criteria.
4. State and sketch the proof of the main theorem.
5. Develop the Bayesian interpretation via standard experiments and mean-preserving spreads.
6. Illustrate each idea with Python simulations.

We begin with some imports.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import minimize
```

30.2 Experiments and stochastic transformations

30.2.1 The state space and experiments

Let $S = \{s_1, \ldots, s_N\}$ be a finite set of possible states of the world.

An **experiment** is described by the conditional distribution of an observed signal $\tilde{x}$ given the state $\tilde{s}$.

When the signal space is also finite, say $X = \{x_1, \dots, x_M\}$, an experiment reduces to an $N \times M$ *Markov matrix*

$$\mu = [\mu_{ij}], \qquad \mu_{ij} = \Pr(\tilde{x}_\mu = x_j \mid \tilde{s} = s_i) \geq 0, \quad \sum_{j=1}^{M} \mu_{ij} = 1 \ \forall\, i.$$

Each row i gives the distribution of signals when the true state is s_i.

```
μ = np.array([[0.6, 0.3, 0.1],
              [0.1, 0.3, 0.6]])

Q = np.array([[1.0, 0.0],
              [0.5, 0.5],
              [0.0, 1.0]])

ν = μ @ Q

print("Experiment μ (3 signals, rows sum to 1):")
print(μ)
print("\nStochastic transformation Q (3 × 2):")
print(Q)
print("\nExperiment ν = μ @ Q (2 signals):")
print(ν)
print("\nRow sums μ:", μ.sum(axis=1))
print("Row sums ν:", ν.sum(axis=1))
```

```
Experiment μ (3 signals, rows sum to 1):
[[0.6 0.3 0.1]
 [0.1 0.3 0.6]]

Stochastic transformation Q (3 × 2):
[[1.  0. ]
 [0.5 0.5]
 [0.  1. ]]

Experiment ν = μ @ Q (2 signals):
[[0.75 0.25]
 [0.25 0.75]]

Row sums μ: [1. 1.]
Row sums ν: [1. 1.]
```

30.2.2 Stochastic transformations

A **stochastic transformation** Q maps signals from one experiment to signals from another by further randomization.

In the discrete setting with M input signals and K output signals, Q is an $M \times K$ Markov matrix: $q_{lk} \geq 0$ and $\sum_k q_{lk} = 1$ for every row l.

Definition 30.2.1 (Sufficiency)

Experiment μ is *sufficient for* ν if there exists a stochastic transformation Q (an $M \times K$ Markov matrix) such that

$$\nu = \mu\, Q,$$

meaning that an observer of $\tilde{x}_\mu$ can generate the distribution of $\tilde{x}_\nu$ by passing their signal through Q.

If you observe the more informative signal $\tilde{x}_\mu$, then you can always *throw away* information to reproduce a less informative signal.

The reverse is not possible: a less informative signal cannot be enriched to recover what was lost.

We can verify this numerically using the two experiments μ and ν defined above.

The function below searches for a stochastic transformation Q that minimizes $\|\nu - \mu Q\|$.

If an exact Q exists the residual will be close to zero; otherwise it will be large.

```
def find_stochastic_transform(μ, ν, tol=1e-8):
    """
    Find a row-stochastic matrix Q that minimizes ||ν - μ @ Q||.
    """
    _, M = μ.shape
    _, K = ν.shape

    def unpack(q_flat):
        return q_flat.reshape(M, K)

    def objective(q_flat):
        Q = unpack(q_flat)
        return np.linalg.norm(ν - μ @ Q)**2

    constraints = [
        {"type": "eq", "fun": lambda q_flat,
        row=i: unpack(q_flat)[row].sum() - 1.0}
        for i in range(M)
    ]
    bounds = [(0.0, 1.0)] * (M * K)
    Q0 = np.full((M, K), 1 / K).ravel()

    result = minimize(
        objective,
        Q0,
        method="SLSQP",
        bounds=bounds,
        constraints=constraints,
        options={"ftol": tol, "maxiter": 1_000},
    )

    Q = unpack(result.x)
    residual = np.linalg.norm(ν - μ @ Q)
    return Q, residual

# Forward: find Q such that ν = μ @ Q  (should succeed)
Q_fwd, res_fwd = find_stochastic_transform(μ, ν)
print("Forward (μ to ν):")
print(f"  residual = {res_fwd:.2e}")
print(f"  exact transformation exists: {res_fwd < 1e-6}")

# Reverse: find Q' such that μ = ν @ Q'  (should fail)
Q_rev, res_rev = find_stochastic_transform(ν, μ)
print("\nReverse (ν to μ):")
print(f"  residual = {res_rev:.2e}")
print(f"  exact transformation exists: {res_rev < 1e-6}")
```

```
Forward (μ to ν):
  residual = 5.05e-09
  exact transformation exists: True

Reverse (ν to μ):
  residual = 1.44e-01
  exact transformation exists: False
```

The forward residual is close to zero: a stochastic transformation from μ to ν exists, confirming that μ is sufficient for ν.

The reverse residual is large: no stochastic transformation can recover μ from ν.

No stochastic transformation can undo the information loss.

The key is that the inverse of a stochastic transformation in general is not a stochastic transformation.

In fact, the only stochastic transformations whose inverses are also stochastic are permutation matrices, which merely relabel signals without losing any information.

30.3 Three equivalent criteria

Blackwell's theorem establishes that three different ways of comparing experiments all turn out to be equivalent.

30.3.1 Criterion 1: the economic criterion

The first criterion compares experiments by their value to decision makers.

Let A be a compact convex set of actions and $u : A \times S \to \mathbb{R}$ a bounded utility function.

A decision maker observes $x \in X$, updates beliefs about $\tilde{s}$ by Bayes' rule, and chooses $d(x) \in A$ to maximize expected utility.

Let $p = (p_1, \ldots, p_N)$ be the prior over states, and write

$$P = \{(p_1, \ldots, p_N) : p_i \geq 0,\ \textstyle\sum_i p_i = 1\}$$

for the probability simplex.

For fixed A and u, the set of *achievable expected-utility vectors* under experiment μ is

$$B(\mu, A, u) = \Big\{ v \in \mathbb{R}^N : v_i = \textstyle\int_X u(f(x), s_i)\, \mu_i(dx) \text{ for some measurable } f : X \to A \Big\}.$$

Definition 30.3.1 (Economic criterion)

μ is **at least as informative as** ν in the economic sense if

$$B(\mu, A, u) \supseteq B(\nu, A, u)$$

for every compact convex action set A and every bounded utility function $u : A \times S \to \mathbb{R}$.

This criterion says that experiment μ is better than experiment ν if anything a decision maker can achieve after seeing ν, they can also achieve after seeing μ.

The reason is that a more informative experiment lets the agent imitate a less informative one by *ignoring* or *garbling* some of the extra information.

But the reverse need not be possible.

So $B(\mu, A, u) \supseteq B(\nu, A, u)$ means that μ gives the decision maker at least as many feasible expected-utility outcomes as ν.

Equivalently, every Bayesian decision maker attains weakly higher expected utility with $\tilde{x}_\mu$ than with $\tilde{x}_\nu$, for every prior $p \in P$.

30.3.2 Criterion 2: the sufficiency criterion

The second criterion uses the stochastic transformation idea introduced above.

Definition 30.3.2 (Blackwell sufficiency)

$\mu \geq \nu$ in Blackwell's sense if there exists a stochastic transformation Q from the signal space of μ to the signal space of ν such that

$$\nu_i(E) = (Q \circ \mu_i)(E) \quad \forall\, E \in \mathcal{G},\ i = 1, \ldots, N.$$

In matrix notation for finite experiments: $\nu = \mu\, Q$.

30.3.3 Criterion 3: the uncertainty criterion

The third criterion compares experiments by how much they reduce uncertainty about the state.

DeGroot [1962] calls any concave function $U : P \to \mathbb{R}$ an **uncertainty function**.

The prototypical example is Shannon entropy:

$$U(p) = -\sum_{i=1}^{N} p_i \log p_i.$$

Definition 30.3.3 (DeGroot uncertainty criterion)

μ **reduces expected uncertainty at least as much as** ν if, for every prior $p \in P$ and every concave $U : P \to \mathbb{R}$,

$$\int_P U(q)\, \hat{\mu}^p(dq) \ \leq \ \int_P U(q)\, \hat{\nu}^p(dq),$$

where $\hat{\mu}^p$ is the distribution of posterior beliefs induced by experiment μ under prior p.

To see this, let $Q = p^\mu(X)$ denote the random posterior induced by experiment μ.

Then Q has distribution $\hat{\mu}^p$, so

$$\mathbb{E}[U(Q)] = \int_P U(q)\, \hat{\mu}^p(dq).$$

Since U is concave, Jensen's inequality gives

$$\mathbb{E}[U(Q)] \leq U(\mathbb{E}[Q]) = U(p).$$

Hence

$$\int_P U(q)\,\hat{\mu}^p(dq) \le U(p),$$

so any experiment weakly lowers expected uncertainty.

Kihlstrom's standard-experiment construction will later let us compare posterior distributions under the uniform prior $c = (1/N, \dots, 1/N)$.

30.4 The main theorem

Theorem 30.4.1 (Blackwell's theorem)

The following three conditions are equivalent:

(i) Economic criterion: $B(\mu, A, u) \supseteq B(\nu, A, u)$ for every compact convex A and every bounded utility function u.

(ii) Sufficiency criterion: There exists a stochastic transformation Q from the signal space of μ to the signal space of ν such that $\nu = Q \circ \mu$.

(iii) Uncertainty criterion: $\int_P U(q)\,\hat{\mu}^p(dq) \le \int_P U(q)\,\hat{\nu}^p(dq)$ for every prior $p \in P$ and every concave U.

See also Blackwell [1951], Bohnenblust *et al.* [1949], and DeGroot [1962].

The hard part is the equivalence between the economic and sufficiency criteria.

Sketch (ii ⇒ i): If $\nu = \mu Q$, then any decision rule based on $\tilde{x}_\nu$ can be replicated by first observing $\tilde{x}_\mu$, then drawing a synthetic $\tilde{x}_\nu$ from Q, and then applying the same rule.

Sketch (i ⇒ ii): Since $B(\mu, A, u) \supseteq B(\nu, A, u)$ for every A and u, a separating-hyperplane (duality) argument implies the existence of a posterior-space mean-preserving kernel D sending the standard experiment of ν into that of μ. Passing from these posterior laws back to the original signal spaces then yields the required garbling Q with $\nu = \mu Q$. Thus D is an intermediate randomization on posterior beliefs, not literally the signal-space kernel Q.

Sketch (ii ⇒ iii): Under a garbling, the posterior from the coarser experiment is the conditional expectation of the posterior from the finer experiment, so Jensen's inequality gives the result for every concave U.

Sketch (iii ⇒ ii): The converse, that the inequality for all concave U forces the existence of Q, is proved in [Blackwell, 1953]. Kihlstrom's posterior-based representation makes the geometry transparent.

30.5 Kihlstrom's Bayesian interpretation

30.5.1 Posteriors and standard experiments

The key object in Kihlstrom's analysis is the *posterior belief vector*.

When prior p holds and experiment μ produces signal x, Bayes' rule gives

$$p_i^\mu(x) = \Pr(\tilde{s} = s_i \mid \tilde{x}_\mu = x) = \frac{\mu_{ix}\, p_i}{\sum_j \mu_{jx}\, p_j}, \qquad i = 1, \dots, N.$$

The posterior $p^\mu(x) \in P$ is a random point in the simplex.

Property 30.5.1 (Mean preservation)

The prior p is the expectation of the posterior:

$$\mathbb{E}[p^\mu] = \sum_x \Pr(\tilde{x}_\mu = x)\, p^\mu(x) = p.$$

This is sometimes called the *law of iterated expectations for beliefs*.

For a fixed prior c, Kihlstrom's **standard experiment** replaces the raw signals of μ with the posterior beliefs they generate.

Let $\hat{\mu}^c$ denote the distribution over posteriors induced by μ under prior c. Mean preservation implies $\int_P q\, \hat{\mu}^c(dq) = c$.

Two experiments are **informationally equivalent** when they induce the same posterior distribution.

The standard experiment strips away every detail of the signal except its posterior, so it provides a canonical Bayesian representation for comparing experiments.

A stochastic kernel on posterior beliefs lives on the simplex P, whereas a Blackwell garbling Q lives on the original signal space. Kihlstrom's construction uses the former to study convex order and then recovers the latter after passing to standard experiments.

Any two experiments that generate the same distribution over posteriors lead to identical decisions for every Bayesian decision maker, regardless of how different their raw signal spaces may look.

30.5.2 Mean-preserving spreads and Blackwell's order

Kihlstrom's key reformulation is the following.

Theorem 30.5.1 (Kihlstrom's Reformulation)

$\mu \geq \nu$ in Blackwell's sense if and only if $\hat{\mu}^c$ is a **mean-preserving spread** of $\hat{\nu}^c$; that is,

$$\int_P g(p)\, \hat{\mu}^c(dp) \;\geq\; \int_P g(p)\, \hat{\nu}^c(dp)$$

for every convex function $g : P \to \mathbb{R}$.

Equivalently, $\hat{\mu}^c$ is larger than $\hat{\nu}^c$ in convex order.

A better experiment spreads posterior beliefs farther from the prior while preserving their mean.

To see this concretely, we define two experiments for the two-state case and compute their posteriors.

```
def compute_posteriors(μ, prior, tol=1e-14):
    """
    Compute the posterior distribution for each signal realisation.
    """
    N, M = μ.shape
    signal_probs = μ.T @ prior
    numerators = μ.T * prior
    posteriors = np.zeros((M, N))
    np.divide(
        numerators,
        signal_probs[:, None],
        out=posteriors,
```

(continues on next page)

(continued from previous page)

```
        where=signal_probs[:, None] > tol,
    )
    return posteriors, signal_probs


def check_mean_preservation(posteriors, signal_probs, prior):
    """Verify E[posterior] == prior."""
    expected_posterior = (posteriors * signal_probs[:, None]).sum(axis=0)
    return expected_posterior, np.allclose(expected_posterior, prior)


N = 2
prior = np.array([0.5, 0.5])

μ_info = np.array([[0.8, 0.2],
                   [0.2, 0.8]])

ν_info = np.array([[0.6, 0.4],
                   [0.4, 0.6]])

post_μ, probs_μ = compute_posteriors(μ_info, prior)
post_ν, probs_ν = compute_posteriors(ν_info, prior)

print("Experiment μ (more informative):\n")
print("Signal probabilities:", probs_μ.round(3))
print("Posteriors (row = signal, col = state):")
print(post_μ.round(3))
mean_μ, ok_μ = check_mean_preservation(post_μ, probs_μ, prior)
print(f"E[posterior] = {mean_μ.round(4)}  (equals prior: {ok_μ})")

print("\n Experiment ν (less informative):\n")
print("Signal probabilities:", probs_ν.round(3))
print("Posteriors:")
print(post_ν.round(3))
mean_ν, ok_ν = check_mean_preservation(post_ν, probs_ν, prior)
print(f"E[posterior] = {mean_ν.round(4)}  (equals prior: {ok_ν})")
```

```
Experiment μ (more informative):

Signal probabilities: [0.5 0.5]
Posteriors (row = signal, col = state):
[[0.8 0.2]
 [0.2 0.8]]
E[posterior] = [0.5 0.5]  (equals prior: True)

 Experiment ν (less informative):

Signal probabilities: [0.5 0.5]
Posteriors:
[[0.6 0.4]
 [0.4 0.6]]
E[posterior] = [0.5 0.5]  (equals prior: True)
```

For $N = 2$ states, the simplex P is the unit interval $[0, 1]$ (the probability of state s_1).

We can directly plot the distribution of posteriors under experiments μ and ν.

```
def plot_posterior_distributions(μ_matrix, ν_matrix, prior,
                                 labels=("μ (more informative)",
                                         "ν (less informative)")):
    """
    For a two-state experiment, plot the distribution of posteriors
    (i.e., the standard experiment distribution) on [0,1].
    """
    posts_μ, probs_μ = compute_posteriors(μ_matrix, prior)
    posts_ν, probs_ν = compute_posteriors(ν_matrix, prior)

    fig, axes = plt.subplots(1, 2, figsize=(11, 4), sharey=False)
    prior_val = prior[0]

    for ax, posts, probs, label in zip(
        axes, [posts_μ, posts_ν], [probs_μ, probs_ν], labels):
        p_s1 = posts[:, 0]
        ax.vlines(p_s1, 0, probs, linewidth=6, color="steelblue", alpha=0.7)
        ax.axvline(prior_val, color="tomato", linestyle="--", linewidth=2,
                   label=f"prior = {prior_val:.2f}")
        ax.set_xlim(0, 1)
        ax.set_xlabel(r"posterior $p(s_1 \mid x)$", fontsize=12)
        ax.set_ylabel("probability mass", fontsize=12)
        mean_post = (p_s1 * probs).sum()
        ax.axvline(mean_post, color="green", linestyle=":", linewidth=2,
                   label=f"E[post] = {mean_post:.2f}")
        ax.text(0.03, 0.94, label, transform=ax.transAxes, va="top")
        ax.legend()

    plt.tight_layout()
    plt.show()

plot_posterior_distributions(μ_info, ν_info, prior)
```

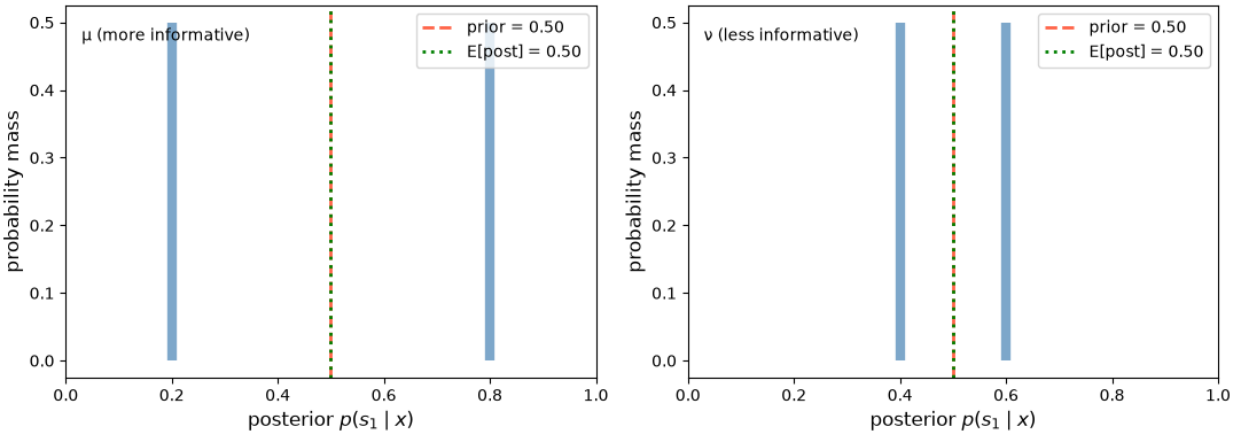


Fig. 30.1: Posterior distributions in the two-state case

This is the mean-preserving spread in action: both distributions have the same mean (equal to the prior), but the more informative experiment μ spreads its posteriors farther apart.

We can verify the mean-preserving spread condition numerically.

The key fact is that, up to an affine term, any convex function can be represented as a mixture of "call option" payoffs $g_t(p) = \max(p - t, 0)$.

Because the two posterior distributions being compared have the same mean, that affine term cancels in the comparison.

So it suffices to check $E[g_t(p^\mu)] \geq E[g_t(p^\nu)]$ for all thresholds $t \in [0, 1]$.

```
def check_mps_convex_functions(μ_matrix, ν_matrix, prior, n_functions=200):
    """
    Verify the mean-preserving spread condition using
    convex functions g(p) = max(p - t, 0).
    """
    posts_μ, probs_μ = compute_posteriors(μ_matrix, prior)
    posts_ν, probs_ν = compute_posteriors(ν_matrix, prior)

    p_μ = posts_μ[:, 0]
    p_ν = posts_ν[:, 0]

    thresholds = np.linspace(0, 1, n_functions)
    diffs = []
    for t in thresholds:
        Eg_μ = (np.maximum(p_μ - t, 0) * probs_μ).sum()
        Eg_ν = (np.maximum(p_ν - t, 0) * probs_ν).sum()
        diffs.append(Eg_μ - Eg_ν)

    fig, ax = plt.subplots(figsize=(8, 4))
    ax.plot(thresholds, diffs, color="steelblue", linewidth=2)
    ax.axhline(0, color="tomato", linestyle="--", linewidth=2)
    ax.fill_between(thresholds, diffs, 0,
                    where=np.array(diffs) >= 0,
                    alpha=0.25, color="steelblue",
                    label="$E[g(p^μ)] - E[g(p^ν)] \\geq 0$")
    ax.set_xlabel("threshold $t$", fontsize=12)
    ax.set_ylabel(r"$E[\max(p-t,0)]$ difference", fontsize=12)
    ax.legend(fontsize=11)
    plt.tight_layout()
    plt.show()

    all_non_negative = all(d >= -1e-10 for d in diffs)
    print(f"μ is a mean-preserving spread of ν: {all_non_negative}")
    return diffs

_ = check_mps_convex_functions(μ_info, ν_info, prior)
```

```
μ is a mean-preserving spread of ν: True
```

The difference $E[g_t(p^\mu)] - E[g_t(p^\nu)]$ is non-negative for every threshold t, confirming that $\hat{\mu}^c$ is a mean-preserving spread of $\hat{\nu}^c$ and therefore $\mu \geq \nu$ in the Blackwell order.

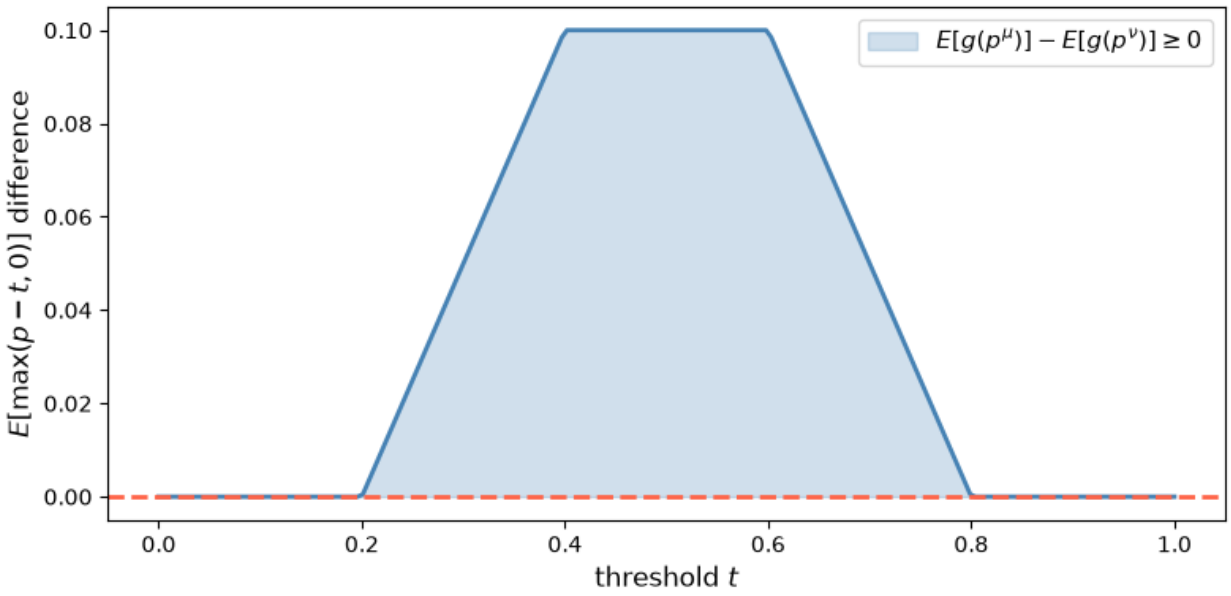


Fig. 30.2: Convex-order check in the two-state case

30.6 Simulating the Blackwell order with many states

We now move to a three-state example.

Experiment μ is strongly correlated with the state, and experiment ν is a garbling of μ.

```
N3 = 3
prior3 = np.array([1/3, 1/3, 1/3])

μ3 = np.array([[0.7, 0.2, 0.1],
               [0.1, 0.7, 0.2],
               [0.2, 0.1, 0.7]])

Q3 = np.array([[0.9, 0.05, 0.05],
               [0.05, 0.8, 0.15],
               [0.05, 0.15, 0.8]])

ν3 = μ3 @ Q3

print("μ (3×3):")
print(np.round(μ3, 2))
print("\nQ (garbling):")
print(np.round(Q3, 2))
print("\nν = μ @ Q:")
print(np.round(ν3, 3))
```

```
μ (3×3):
[[0.7 0.2 0.1]
 [0.1 0.7 0.2]
 [0.2 0.1 0.7]]

Q (garbling):
```

(continues on next page)

(continued from previous page)

```
[[0.9  0.05 0.05]
 [0.05 0.8  0.15]
 [0.05 0.15 0.8 ]]

ν = μ @ Q:
[[0.645 0.21  0.145]
 [0.135 0.595 0.27 ]
 [0.22  0.195 0.585]]
```

For three states, posterior beliefs live in a 2-simplex.

Let's visualize sampled posterior points under μ and ν

```
def sample_posteriors(μ_matrix, prior, n_draws=3000, rng=None):
    """
    Simulate n_draws observations from the experiment and compute
    the resulting posterior beliefs.
    Returns array of shape (n_draws, N).
    """
    if rng is None:
        rng = np.random.default_rng()
    N, M = μ_matrix.shape
    states = rng.choice(N, size=n_draws, p=prior)
    signals = np.array([rng.choice(M, p=μ_matrix[s]) for s in states])
    posteriors, _ = compute_posteriors(μ_matrix, prior)
    return posteriors[signals]


def simplex_to_cart(pts):
    """Convert 3-simplex barycentric coordinates to 2-D Cartesian."""
    corners = np.array([[0.0, 0.0],
                        [1.0, 0.0],
                        [0.5, np.sqrt(3)/2]])
    return pts @ corners


def plot_simplex_posteriors(μ_matrix, ν_matrix, prior3, n_draws=3000, seed=0):
    rng = np.random.default_rng(seed)
    posts_μ = sample_posteriors(μ_matrix, prior3, n_draws, rng=rng)
    posts_ν = sample_posteriors(ν_matrix, prior3, n_draws, rng=rng)

    cart_μ = simplex_to_cart(posts_μ)
    cart_ν = simplex_to_cart(posts_ν)
    prior_cart = simplex_to_cart(prior3[None, :])[0]

    corners = np.array([[0.0, 0.0],
                        [1.0, 0.0],
                        [0.5, np.sqrt(3)/2]])

    fig, axes = plt.subplots(1, 2, figsize=(12, 5))
    panel_labels = ["μ (more informative)", "ν (garbled)"]
    data = [(cart_μ, "steelblue"), (cart_ν, "darkorange")]
    labels = ["$s_1$", "$s_2$", "$s_3$"]
    offsets = [(-0.07, -0.05), (0.02, -0.05), (-0.02, 0.03)]

    for ax, (cart, c), panel_label in zip(axes, data, panel_labels):
        tri = plt.Polygon(corners, fill=False, edgecolor="black", linewidth=2)
```

(continues on next page)

(continued from previous page)

```
        ax.add_patch(tri)
        ax.scatter(cart[:, 0], cart[:, 1], s=4, alpha=0.25, color=c)
        ax.scatter(*prior_cart, s=120, color="red", zorder=5,
                   label="prior", marker="*")
        for i, (lbl, off) in enumerate(zip(labels, offsets)):
            ax.text(corners[i][0] + off[0], corners[i][1] + off[1],
                    lbl, fontsize=13)
        ax.set_xlim(-0.15, 1.15)
        ax.set_ylim(-0.1, np.sqrt(3)/2 + 0.1)
        ax.set_aspect("equal")
        ax.set_xticks([])
        ax.set_yticks([])
        ax.text(0.03, 0.94, panel_label, transform=ax.transAxes, va="top")
        ax.legend(fontsize=11, loc="upper right")

    plt.tight_layout()
    plt.show()

plot_simplex_posteriors(μ3, ν3, prior3)
```

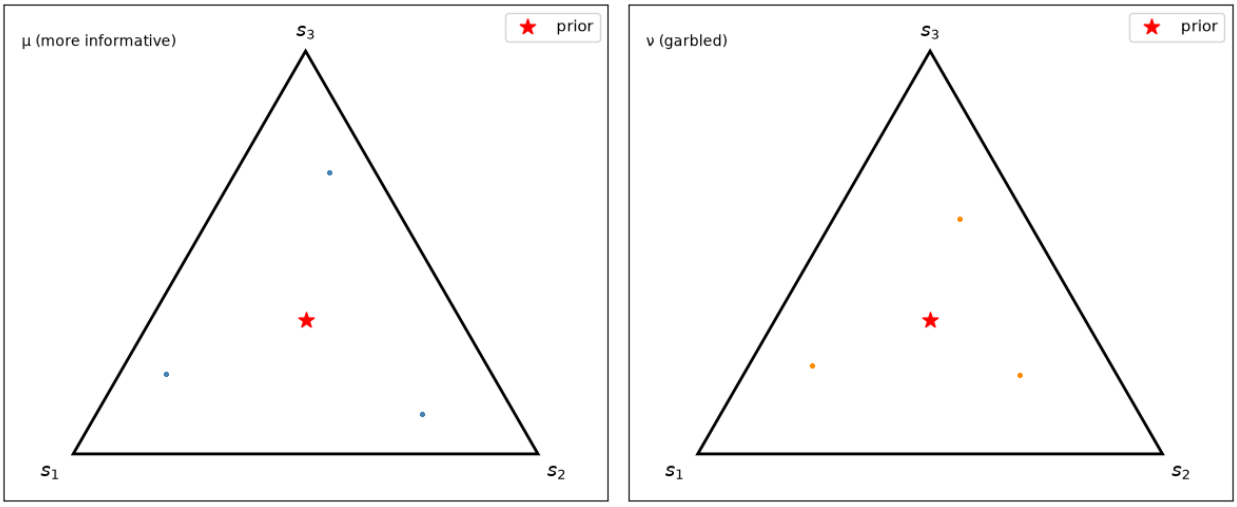


Fig. 30.3: Sampled posterior points on the 2-simplex

Because this example has only three signals, each panel consists of three posterior atoms sampled repeatedly rather than a continuous cloud.

Under μ, the sampled posterior points reach farther toward the vertices.

Under the garbled experiment ν, the sampled posterior points stay closer to the center.

30.7 The DeGroot uncertainty function

30.7.1 Concave uncertainty functions and the value of information

[DeGroot, 1962] formalizes the value of information through an **uncertainty function** $U : P \to \mathbb{R}$.

In DeGroot's axiomatization, an uncertainty function is:

- *Concave*: by Jensen, observing any signal weakly reduces expected uncertainty.
- *Symmetric*: it depends on the components of p, not their labeling.
- *Normalized*: it is maximized at $p = (1/N, \dots, 1/N)$ and minimized at vertices.

The *value of experiment* μ *given prior* p is

$$I(\tilde{x}_\mu;\, \tilde{s};\, U) = U(p) - \mathbb{E}[U(p^\mu)],$$

This quantity is the expected reduction in uncertainty.

Blackwell's order is equivalent to the statement that $I(\tilde{x}_\mu; \tilde{s}; U) \geq I(\tilde{x}_\nu; \tilde{s}; U)$ for *every* concave U.

30.7.2 Shannon entropy as a special case

The canonical uncertainty function is Shannon entropy

$$U_H(p) = -\sum_{i=1}^{N} p_i \log p_i.$$

Under the uniform prior $c = (1/N, \dots, 1/N)$, DeGroot's value formula becomes

$$I(\tilde{x}_\mu, c;\, U_H) = \log N - H(\tilde{s} \mid \tilde{x}_\mu),$$

where $H(\tilde{s} \mid \tilde{x}_\mu)$ is the conditional entropy of the state given the signal.

To see why, write $H(\tilde{s} \mid \tilde{x}_\mu) = \sum_x \Pr(\tilde{x}_\mu = x)\, H(\tilde{s} \mid \tilde{x}_\mu = x)$, where each conditional entropy term equals $-\sum_i p_i^\mu(x) \log p_i^\mu(x) = U_H(p^\mu(x))$.

Substituting into DeGroot's formula gives $I = U_H(c) - \mathbb{E}[U_H(p^\mu)] = \log N - H(\tilde{s} \mid \tilde{x}_\mu)$, which is exactly the *mutual information* between $\tilde{x}_\mu$ and $\tilde{s}$.

Note

The Blackwell ordering implies the entropy-based inequality, but the *converse fails*: entropy alone does not pin down the full Blackwell ordering.

Two experiments can have the same mutual information yet differ in Blackwell rank, because a single concave function cannot detect all differences in the dispersion of posteriors.

The full Blackwell ordering requires the inequality to hold for *every* concave U, not just Shannon entropy.

```
def entropy(p, ε=1e-12):
    """Shannon entropy of a probability vector."""
    p = np.asarray(p, dtype=float)
    p = np.clip(p, ε, 1.0)
    return -np.sum(p * np.log(p))
```

(continues on next page)

(continued from previous page)

```
def degroot_value(μ_matrix, prior, U_func):
    """
    Compute DeGroot's value of information I = U(prior) - E[U(posterior)].
    """
    posts, probs = compute_posteriors(μ_matrix, prior)
    prior_uncertainty = U_func(prior)
    expected_post_uncertainty = sum(
        probs[j] * U_func(posts[j]) for j in range(len(probs)))
    return prior_uncertainty - expected_post_uncertainty


def gini_impurity(p):
    """Gini impurity: 1 - sum(p_i^2)."""
    return 1.0 - np.sum(np.asarray(p)**2)


def tsallis_entropy(p, q=2):
    """Tsallis entropy of order q (concave for q>1)."""
    p = np.clip(p, 1e-12, 1.0)
    return (1 - np.sum(p**q)) / (q - 1)


def tsallis_q15(p):
    """Tsallis entropy with q=1.5 for an independent concavity check."""
    return tsallis_entropy(p, q=1.5)


def sqrt_index(p):
    """Concave uncertainty index based on sum(sqrt(p_i))."""
    p = np.clip(np.asarray(p), 0.0, 1.0)
    return np.sum(np.sqrt(p)) - 1.0

uncertainty_functions = {
    "Shannon entropy": entropy,
    "Gini impurity": gini_impurity,
    "Tsallis (q=1.5)": tsallis_q15,
    "Square-root index": sqrt_index,
}

header = (f"{'Uncertainty function':<22}  "
          f"{'I(μ)':<10}  {'I(ν)':<10}  "
          f"{'I(μ)>=I(ν)?'}")
print(header)
print("-" * 58)
for name, U in uncertainty_functions.items():
    I_μ = degroot_value(μ_info, prior, U)
    I_ν = degroot_value(ν_info, prior, U)
    print(f"{name:<22}  {I_μ:<10.4f}  {I_ν:<10.4f}  {I_μ >= I_ν - 1e-10}")
```

```
Uncertainty function    I(μ)        I(ν)        I(μ)>=I(ν)?
----------------------------------------------------------
Shannon entropy         0.1927      0.0201      True
Gini impurity           0.1800      0.0200      True
Tsallis (q=1.5)         0.1958      0.0213      True
Square-root index       0.0726      0.0072      True
```

As predicted by the theorem, $I(\mu) \geq I(\nu)$ for every concave uncertainty function once we know $\mu \geq \nu$ in the Blackwell sense.

30.7.3 Value of information as a function of experiment quality

We now parameterize a continuum of experiments between the uninformative and perfectly informative cases.

For $N = 2$ states, a natural family is

$$\mu(\theta) = (1-\theta) \cdot \tfrac{1}{2}\mathbf{1}\mathbf{1}^\top + \theta \cdot I_2, \quad \theta \in [0,1],$$

The first term is the completely mixed matrix and I_2 is the identity.

```
def make_experiment(θ, N=2):
    """Parameterized experiment: θ=0 is uninformative, θ=1 is perfect."""
    return (1 - θ) * np.ones((N, N)) / N + θ * np.eye(N)


θs = np.linspace(0, 1, 100)
prior2 = np.array([0.5, 0.5])

fig, ax = plt.subplots(figsize=(9, 4))
for name, U in uncertainty_functions.items():
    values = [degroot_value(make_experiment(θ), prior2, U) for θ in θs]
    vmin, vmax = values[0], values[-1]
    normed = (np.array(values) - vmin) / (vmax - vmin + 1e-15)
    ax.plot(θs, normed, label=name, linewidth=2)

ax.set_xlabel("experiment quality θ  (0 = uninformative, 1 = perfect)",
              fontsize=11)
ax.set_ylabel("normalized value of information I(μ(θ))", fontsize=11)
ax.legend(fontsize=10)
plt.tight_layout()
plt.show()
```

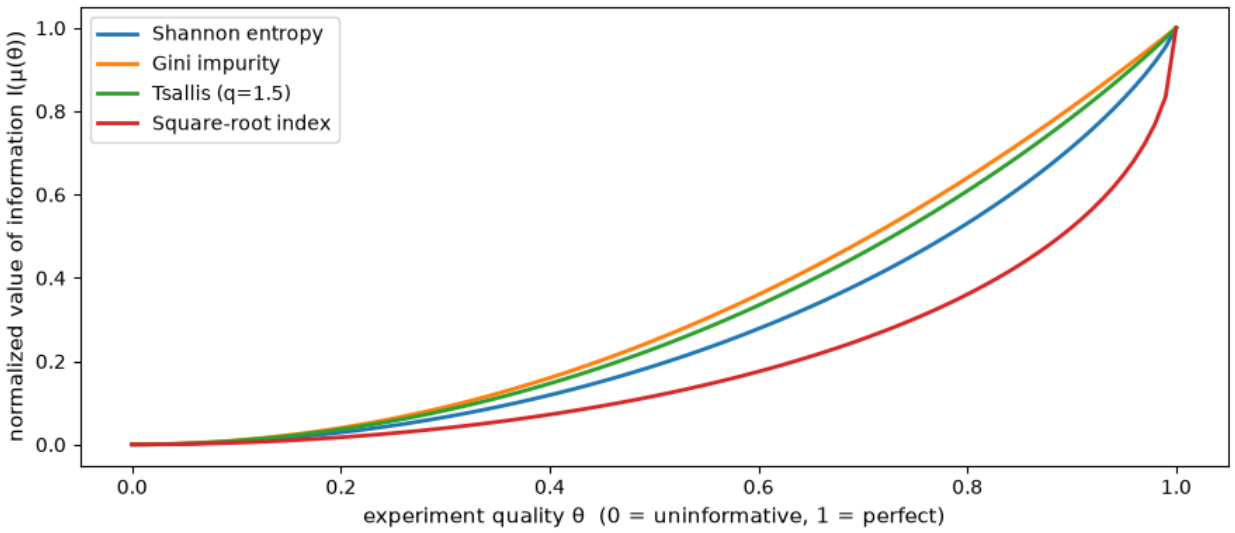


Fig. 30.4: Value of information and experiment quality

Every concave uncertainty function assigns weakly higher value to a more informative experiment.

30.8 Connection to second-order stochastic dominance

A random variable X **second-order stochastically dominates** Y (written $X \succeq_{\text{SOSD}} Y$) if $E[u(X)] \geq E[u(Y)]$ for every concave function u. Equivalently, Y is a mean-preserving spread of X.

The uncertainty-function representation makes the connection to SOSD explicit.

Because U is concave, $-U$ is convex, and the condition

$$\mathbb{E}[U(p^\mu)] \leq \mathbb{E}[U(p^\nu)] \quad \text{for all concave } U$$

is precisely the statement that $\hat{\mu}^c$ dominates $\hat{\nu}^c$ in convex order on P.

When $N = 2$, posterior beliefs are scalars in $[0, 1]$, and the SOSD comparison reduces to the classical integrated-CDF test.

Specifically, $\hat{\mu}^c$ is a mean-preserving spread of $\hat{\nu}^c$ if and only if $\int_0^t F_\mu(s)\, ds \geq \int_0^t F_\nu(s)\, ds$ for all $t \in [0, 1]$, where F_μ and F_ν are the CDFs of the posterior on s_1 under each experiment. Equivalently, in SOSD language, the less informative posterior under ν dominates the more dispersed posterior under μ.

We can verify this graphically for the two-state example above

```
def cdf_data_1d(weights, values):
    """Sort support points and cumulative masses for a discrete distribution."""
    idx = np.argsort(values)
    sorted_vals = values[idx]
    sorted_wts = weights[idx]
    cum_mass = np.cumsum(sorted_wts)
    return sorted_vals, cum_mass


def plot_sosd_posteriors(μ_matrix, ν_matrix, prior):
    """Plot CDFs and integrated CDFs for the posterior-on-s1 distributions."""
    posts_μ, probs_μ = compute_posteriors(μ_matrix, prior)
    posts_ν, probs_ν = compute_posteriors(ν_matrix, prior)

    p_μ = posts_μ[:, 0]
    p_ν = posts_ν[:, 0]

    sv_μ, cm_μ = cdf_data_1d(probs_μ, p_μ)
    sv_ν, cm_ν = cdf_data_1d(probs_ν, p_ν)

    fig, axes = plt.subplots(1, 2, figsize=(11, 4))

    ax = axes[0]
    for sv, cm, lbl, c in [(sv_μ, cm_μ, "μ", "steelblue"),
                           (sv_ν, cm_ν, "ν", "darkorange")]:
        xs = np.concatenate([[0], sv, [1]])
        ys = np.concatenate([[0], cm, [1]])
        ax.step(xs, ys, where="post", label=lbl, color=c, linewidth=2)
    ax.axvline(prior[0], linestyle="--", color="gray", alpha=0.6, linewidth=2,
               label="prior")
    ax.set_xlabel(r"posterior $p(s_1 \mid x)$", fontsize=12)
    ax.set_ylabel("cumulative probability", fontsize=12)
    ax.text(0.03, 0.94, "CDFs", transform=ax.transAxes, va="top")
```

(continues on next page)

(continued from previous page)

```
    ax.legend(fontsize=11)

    ax2 = axes[1]
    grid = np.linspace(0, 1, 200)

    def integrated_cdf(sorted_vals, cum_mass, grid):
        cdf = np.array([cum_mass[sorted_vals <= t].max()
                        if np.any(sorted_vals <= t) else 0.0
                        for t in grid])
        return np.cumsum(cdf) * (grid[1] - grid[0])

    int_μ = integrated_cdf(sv_μ, cm_μ, grid)
    int_ν = integrated_cdf(sv_ν, cm_ν, grid)

    ax2.plot(grid, int_μ, label=r"$\int F_\mu$", color="steelblue", linewidth=2)
    ax2.plot(grid, int_ν, color="darkorange",
             label=r"$\int F_\nu$", linewidth=2)
    ax2.fill_between(grid, int_ν, int_μ,
                     where=int_μ >= int_ν,
                     alpha=0.2, color="steelblue",
                     label=(r"$\int F_\mu \geq \int F_\nu$"
                            r" ($\mu$ is an MPS of $\nu$)"))
    ax2.set_xlabel(r"$t$", fontsize=12)
    ax2.set_ylabel("integrated CDF", fontsize=12)
    ax2.text(0.03, 0.94, "integrated CDFs", transform=ax2.transAxes, va="top")
    ax2.legend(fontsize=10)

    plt.tight_layout()
    plt.show()

plot_sosd_posteriors(μ_info, ν_info, prior)
```

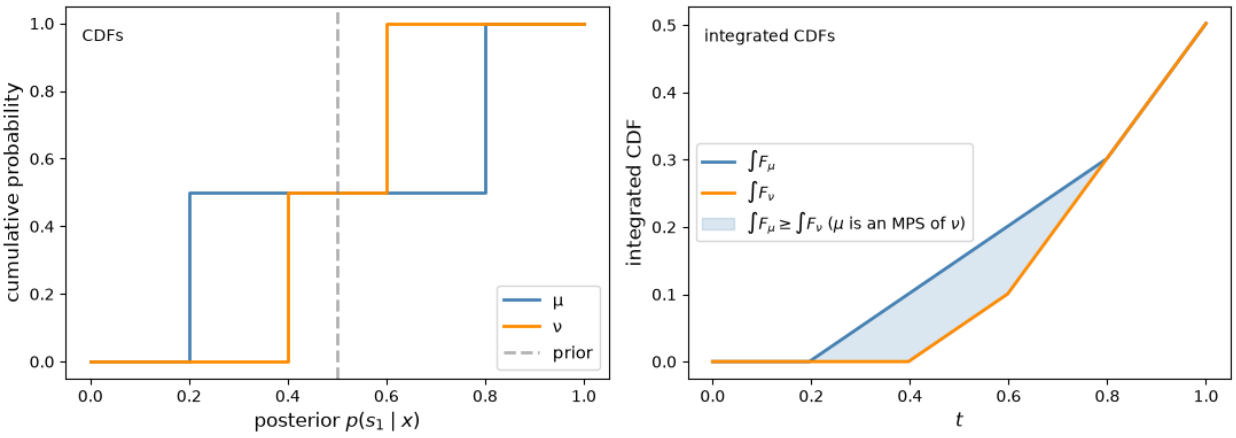


Fig. 30.5: Integrated-CDF check in the two-state case

30.9 Application 1: product quality information

Kihlstrom [1974] applies Blackwell's theorem to consumer demand for information about product quality.

- The unknown state $\tilde{s}$ is a product parameter θ.
- A consumer can purchase λ units of information at cost $c(\lambda)$.
- As λ rises, the experiment becomes more informative in the Blackwell sense.

The Blackwell order says that, absent costs, more information is always better for every expected-utility maximizer.

With costs, the consumer chooses quality investment θ to maximize *net value*.

If quality investment translates into experiment accuracy with diminishing returns – say, accuracy $\phi(\theta) = 1 - e^{-a\theta}$ for a rate parameter a – then the marginal value of information eventually decreases in θ.

With a convex cost $c(\theta) = c\,\theta^2$, the increasing marginal cost eventually overtakes the declining marginal value, producing an interior optimum.

```
def gross_value(θ, prior2, U=entropy, rate=2):
    """Gross value of quality investment θ (diminishing returns)."""
    accuracy = 1 - np.exp(-rate * θ)
    μ_t = (1 - accuracy) * np.ones((2, 2)) / 2 + accuracy * np.eye(2)
    return degroot_value(μ_t, prior2, U)


θ_fine = np.linspace(0, 1, 200)
c = 0.6

gross_vals = np.array([gross_value(θ, prior2) for θ in θ_fine])
cost_vals = c * θ_fine**2
net_vals = gross_vals - cost_vals
marginal_vals = np.gradient(gross_vals, θ_fine)
marginal_cost = 2 * c * θ_fine
opt_idx = int(np.argmax(net_vals))

fig, axes = plt.subplots(1, 2, figsize=(12, 4))

ax = axes[0]
ax.plot(θ_fine, gross_vals,
        label="Gross value I(θ)",
        color="steelblue", linewidth=2)
ax.plot(θ_fine, cost_vals,
        label=r"Cost $c\theta^2$",
        color="tomato", linestyle="--", linewidth=2)
ax.plot(θ_fine, net_vals,
        label="Net value", color="green", linewidth=2)
ax.axvline(θ_fine[opt_idx], color="green",
           linestyle=":", linewidth=2,
           label=f"θ* ≈ {θ_fine[opt_idx]:.2f}")
ax.set_xlabel("quality investment θ", fontsize=11)
ax.set_ylabel("value (entropy units)", fontsize=11)
ax.legend(fontsize=10)

ax2 = axes[1]
ax2.plot(θ_fine, marginal_vals,
         label="Marginal value I'(θ)",
         color="steelblue", linewidth=2)
```

(continues on next page)

(continued from previous page)

```
ax2.plot(θ_fine, marginal_cost,
         label=r"Marginal cost $2c\theta$",
         color="tomato", linestyle="--", linewidth=2)
ax2.axvline(θ_fine[opt_idx], color="green",
            linestyle=":", linewidth=2,
            label=f"θ* ≈ {θ_fine[opt_idx]:.2f}")
ax2.set_xlabel("quality investment θ", fontsize=11)
ax2.set_ylabel("marginal value / cost", fontsize=11)
ax2.legend(fontsize=10)

plt.tight_layout()
plt.show()
```

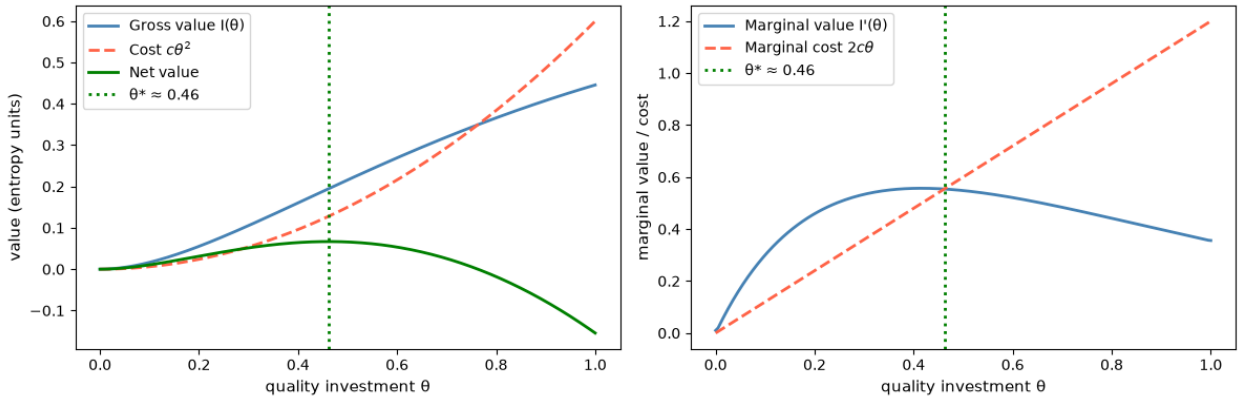


Fig. 30.6: Information demand with a quadratic cost

The optimal investment θ^* occurs where marginal value equals marginal cost.

Because experiment accuracy has diminishing returns in θ, the marginal value of investment eventually falls below the rising marginal cost, yielding a genuine interior optimum.

Raising c shifts the marginal cost curve up and reduces θ^*, while a more asymmetric prior shifts the marginal value curve and changes the optimum.

30.10 Application 2: sequential experimental design

DeGroot [1962] applies the uncertainty-function framework to *sequential experimental design*.

Each period a statistician observes one draw and updates the posterior.

The question is which sequence of experiments minimizes cumulative expected uncertainty.

If one experiment is more informative than another at every stage, then the Blackwell order favors using the better experiment at every date.

We now simulate sequential belief updating for experiments of different quality.

```
def sequential_update(μ_matrix, prior, T=20, seed=0):
    """Simulate T sequential belief updates under experiment μ."""
    rng = np.random.default_rng(seed)
```

(continues on next page)

(continued from previous page)

```
    N, M = μ_matrix.shape
    beliefs = np.zeros((T + 1, N))
    beliefs[0] = prior.copy()

    true_state = rng.choice(N, p=prior)

    for t in range(T):
        p = beliefs[t]
        signal = rng.choice(M, p=μ_matrix[true_state])
        unnorm = μ_matrix[:, signal] * p
        beliefs[t + 1] = unnorm / unnorm.sum()

    return beliefs, true_state


def plot_sequential_beliefs(θs_compare, prior2, T=25):
    fig, axes = plt.subplots(1, len(θs_compare), figsize=(14, 4), sharey=True)

    for ax, θ in zip(axes, θs_compare):
        μ_t = make_experiment(θ, N=2)
        for seed in range(15):
            beliefs, ts = sequential_update(μ_t, prior2, T=T, seed=seed)
            c = "steelblue" if ts == 0 else "darkorange"
            ax.plot(beliefs[:, 0], alpha=0.35, color=c, linewidth=2)
        ax.axhline(prior2[0], linestyle="--", color="gray", linewidth=2,
                   label="prior")
        ax.axhline(1.0, linestyle=":", color="steelblue", linewidth=2)
        ax.axhline(0.0, linestyle=":", color="darkorange", linewidth=2)
        ax.set_xlabel(r"period $t$", fontsize=11)
        if θ == θs_compare[0]:
            ax.set_ylabel(r"posterior $p(s_1 \mid x^t)$", fontsize=11)
        ax.set_ylim(-0.05, 1.05)
        ax.text(0.03, 0.94, f"θ = {θ}", transform=ax.transAxes, va="top")
        ax.legend(fontsize=9)

    plt.tight_layout()
    plt.show()

plot_sequential_beliefs([0.2, 0.5, 0.9], prior2, T=30)
```

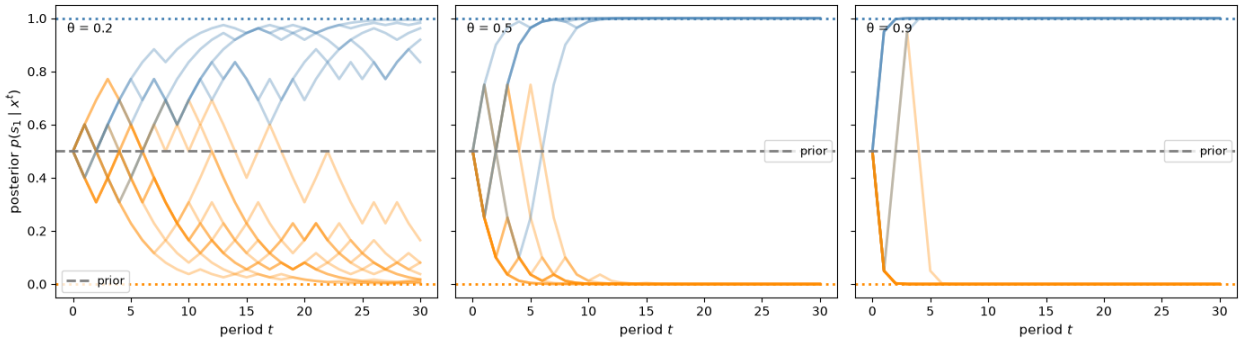


Fig. 30.7: Sequential posterior paths for different experiment qualities

More informative experiments make beliefs converge faster to the truth.

Under the correct prior, the posterior process is a martingale.

```
def check_martingale_mean(μ_matrix, prior, T=15, n_paths=2000, seed=0):
    """
    Simulate many belief paths and check E[p_t] = p_0.
    """
    rng = np.random.default_rng(seed)
    N, M = μ_matrix.shape
    all_paths = np.zeros((n_paths, T + 1, N))

    for k in range(n_paths):
        true_state = rng.choice(N, p=prior)
        p = prior.copy()
        all_paths[k, 0] = p
        for t in range(T):
            signal = rng.choice(M, p=μ_matrix[true_state])
            unnorm = μ_matrix[:, signal] * p
            p = unnorm / unnorm.sum()
            all_paths[k, t + 1] = p

    mean_path = all_paths[:, :, 0].mean(axis=0)

    fig, ax = plt.subplots(figsize=(8, 4))
    ax.plot(mean_path, color="steelblue", linewidth=2,
            label=r"$\bar p_t(s_1)$ (mean over paths)")
    ax.axhline(prior[0], linestyle="--", color="tomato", linewidth=2,
               label=fr"Prior $p_0 = {prior[0]:.2f}$")
    ax.set_xlabel(r"period $t$", fontsize=12)
    ax.set_ylabel(r"$E[p_t(s_1)]$", fontsize=12)
    ax.legend(fontsize=11)
    ax.set_ylim(0, 1)
    plt.tight_layout()
    plt.show()

    print(f"Prior = {prior[0]:.4f}")
    print(f"Average mean belief across dates: {mean_path.mean():.4f}")

check_martingale_mean(μ_info, prior, T=20, n_paths=5000)
```

```
Prior = 0.5000
Average mean belief across dates: 0.4942
```

The simulated cross-sectional mean stays close to the prior at every date.

This is the unconditional implication of the posterior martingale property.

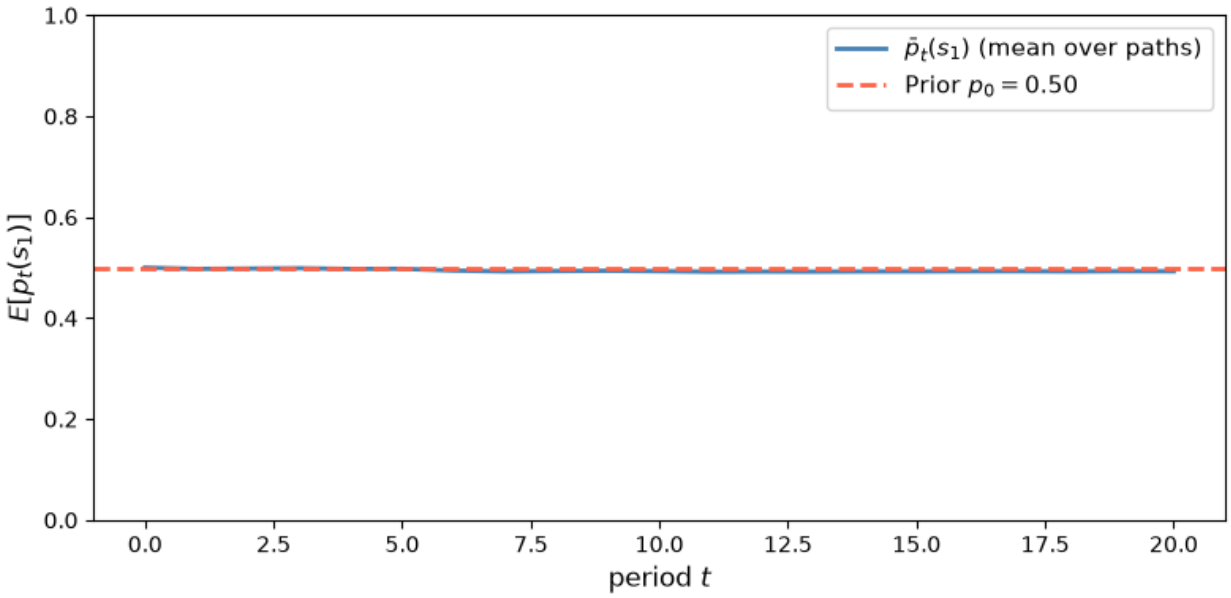


Fig. 30.8: Unconditional implication of the posterior martingale property

30.11 Summary

Blackwell's theorem identifies a *partial order* on statistical experiments with three equivalent characterizations:

Criterion	Condition
Economic	Every decision maker weakly prefers μ to ν: $B(\mu, A, u) \supseteq B(\nu, A, u)$
Sufficiency	ν is a garbling of μ: $\nu = \mu Q$ for some Markov Q
Uncertainty	μ reduces expected uncertainty more for every prior p and every concave U

Kihlstrom's Bayesian exposition places the *posterior distribution* at the center.

A more informative experiment generates a more dispersed posterior distribution with the same mean prior.

The right probabilistic language is convex order on the simplex of posterior beliefs.

In the two-state case this reduces to the familiar SOSD / integrated-CDF test on $[0, 1]$.

DeGroot's contribution is to extend the comparison from particular utility functions to the full class of concave uncertainty functions.

30.12 The Data Processing Inequality and Coarse-Graining

Blackwell's condition that $\nu = \mu Q$ for some Markov kernel Q is the same mathematical operation that underlies the **data processing inequality** (DPI) and the **coarse-graining theorem** in information theory, information geometry, and machine learning.

30.12.1 The DPI for f-divergences

An **f-divergence** between two probability distributions P and Q over a finite space Ω is

$$D_f(P\|Q) = \sum_{\omega\in\Omega} q_\omega \, f\left(\frac{p_\omega}{q_\omega}\right),$$

where $f : (0,\infty) \to \mathbb{R}$ is a convex function with $f(1) = 0$.

Special cases include:

Divergence	Generator $f(t)$
KL-divergence	$t \log t$
Squared Hellinger H^2	$(\sqrt{t}-1)^2/2$
Total variation TV	$\lvert t-1\rvert/2$
Chi-squared χ^2	$(t-1)^2$

The class of f-divergences was introduced independently by Ali and Silvey [1966], Csiszár [1963], and Morimoto [1963]; see also Liese [2012].

Theorem 30.12.1 (Data Processing Inequality)

For any f-divergence D_f and any Markov kernel (stochastic transformation) κ, with $P\kappa$ denoting the image of P under κ, we have

$$D_f(P\|Q) \geq D_f(P\kappa\|Q\kappa).$$

If κ is induced by a sufficient statistic for the pair $\{P, Q\}$, then equality holds.

A converse of this form requires additional hypotheses; a clean binary-model characterization is given below.

The proof follows from Jensen's inequality applied to the convex function f, using the fact that κ is a stochastic matrix [Csiszár, 1963].

30.12.2 Connection to Blackwell's sufficiency condition

In Blackwell's framework, μ and ν are experiments over the same state space $S = \{s_1, \ldots, s_N\}$.

For two states, each experiment has two rows: $\mu_1 = \mu(s_1, \cdot)$ and $\mu_2 = \mu(s_2, \cdot)$.

If $\nu = \mu Q$ (i.e., ν is a garbling of μ), then the pair $(\nu_1, \nu_2) = (\mu_1 Q, \mu_2 Q)$ is obtained by applying the Markov kernel Q to the pair (μ_1, μ_2).

The coarse-graining theorem then implies immediately:

$$D_f(\mu_1\|\mu_2) \geq D_f(\nu_1\|\nu_2) \quad \text{for every f-divergence } D_f,$$

whenever $\mu \geq \nu$ in the Blackwell order.

So a more informative experiment always produces *more separated* conditional signal distributions, in the sense of every f-divergence simultaneously.

The DPI is thus a statement about the *distinguishability* of states: garbling an experiment makes the states harder to tell apart under every statistical measure of separability.

For binary experiments, the equality condition links the DPI directly back to Blackwell: $D_f(\mu_1 Q \| \mu_2 Q) = D_f(\mu_1 \| \mu_2)$ for some strictly convex f if and only if Q is a sufficient statistic for (μ_1, μ_2).

Once sufficiency holds, equality follows for every convex f [Liese, 2012].

30.12.3 Information geometry: Chentsov's theorem

The DPI has an infinitesimal, differential-geometric companion.

Chentsov's theorem [Čencov, 1981] states that the **Fisher information matrix** $I_F(\theta)$ is, up to a constant rescaling, the *unique* Riemannian metric on a statistical manifold that contracts under every Markov morphism (coarse-graining):

$$I_F(\theta;\, \mu) \succeq I_F(\theta;\, \mu\kappa) \quad \text{for every differentiable family } \{\mu_\theta\} \text{ and every Markov kernel } \kappa.$$

Equality holds if and only if κ is a sufficient statistic for θ.

The uniqueness clause is deep: it says that the Fisher information is not merely *one* metric that happens to contract under coarse-graining, but the *only one* with that property.

See Amari and Nagaoka [2000] for a thorough treatment of information geometry and its connections to sufficiency.

30.12.4 The information bottleneck in machine learning

The **information bottleneck** method of Tishby *et al.* [1999] provides a prominent application of the DPI in machine learning.

Given a joint distribution $p(X, Y)$ over an input X and a target Y, the goal is to find a compressed representation T, formed by a stochastic mapping $p(T \mid X)$, that retains as much information about Y as possible while using as few bits as possible to describe X.

The method minimizes the Lagrangian

$$\mathcal{L}[p(T \mid X)] = I(X;\, T) - \beta\, I(T;\, Y),$$

where $I(\cdot\,;\,\cdot)$ denotes mutual information and $\beta \geq 0$ governs the compression–relevance trade-off.

Because $Y - X - T$ forms a Markov chain (T is derived from X alone), the DPI implies

$$I(T;\, Y) \leq I(X;\, Y),$$

with equality if and only if T is a **sufficient statistic** for Y given X.

The Blackwell ordering explains why no deterministic or random post-processing of X can increase the mutual information with Y: any Markov kernel applied to X is a garbling in Blackwell's sense, and the DPI is the mutual-information form of the coarse-graining theorem.

In machine learning language the information bottleneck searches among all garblings of X for the one that best preserves relevant information about Y subject to a compression budget.

In a deep neural network with input X and target Y and layers $X \to T_1 \to T_2 \to \cdots \to T_L \to \hat{Y}$, each layer's representation is a garbling of the previous one.

The DPI then implies the chain of inequalities

$$I(X;\, Y) \geq I(T_1;\, Y) \geq I(T_2;\, Y) \geq \cdots \geq I(T_L;\, Y),$$

so successive layers can only lose, never gain, information about Y.

This observation was placed at the center of the study of what deep networks learn by [Shwartz-Ziv and Tishby, 2017].

Fig. 30.4 already illustrates this: as experiment quality θ increases, every measure of informativeness rises monotonically.

The DPI says the same thing in reverse: garbling (decreasing θ) can only contract these measures.

30.12.5 Summary of the DPI–Blackwell correspondence

The table below collects the precise correspondence between Blackwell's framework and the data-processing and coarse-graining literature.

Blackwell / DeGroot	Data processing / coarse-graining
Garbling $\nu = \mu Q$	Applying Markov kernel κ to a pair $(P, Q) = (\mu_1, \mu_2)$
$\mu \geq \nu$ in Blackwell order	$D_f(\mu_1 \mid \mu_2) \geq D_f(\nu_1 \mid \nu_2)$ for every f-divergence
Sufficiency (Q discards nothing)	Equality in DPI; in binary models, one strictly convex f already characterizes sufficiency
DeGroot value $I(\mu; U_H)$	Mutual information $I(\tilde{x}_\mu;\, \tilde{s})$ (Shannon DPI)
Posterior spreads under μ vs ν	D_f between rows larger under μ
Blackwell theorem (economic $\Leftrightarrow$ garbling)	DPI for all $f \Leftrightarrow$ single Markov kernel witnesses dominance
Chentsov's uniqueness theorem	Fisher information is the unique coarse-graining-contracting metric
Information bottleneck $I(T;Y) \leq I(X;Y)$	DPI for mutual information applied to Markov chain $Y - X - T$

30.13 Relation to Bayesian likelihood-ratio learning

The lecture *Likelihood Ratio Processes and Bayesian Learning* is a dynamic two-state special case of the framework developed here.

Let $S = \{s_1, s_2\}$ with $s_1 \leftrightarrow f$ and $s_2 \leftrightarrow g$, where f and g are the two candidate data-generating densities.

Then a single observation is a Blackwell experiment with rows $f(\cdot)$ and $g(\cdot)$, and the history $w^t = (w_1, \ldots, w_t)$ defines a richer experiment μ_t.

Because one can always discard the last $t - s$ observations, μ_t Blackwell-dominates μ_s for every $t > s$.

The likelihood-ratio process

$$L(w^t) = \prod_{i=1}^{t} \frac{f(w_i)}{g(w_i)}$$

is a sufficient statistic for μ_t, and the posterior

$$\pi_t = \Pr(s_1 \mid w^t) = \frac{\pi_0 L(w^t)}{\pi_0 L(w^t) + 1 - \pi_0}$$

is Kihlstrom's standard experiment in this two-state setting.

Its martingale property, $E[\pi_t] = \pi_0$, is exactly the mean-preservation result proved above for posterior distributions.

Likewise, $\mu_t \geq \mu_s$ implies that the distribution of π_t is a mean-preserving spread of the distribution of π_s, so additional data pushes beliefs farther toward 0 and 1 while lowering expected uncertainty under every concave uncertainty function.

30.13.1 Summary table

The table below records the dictionary between the two lectures without repeating the earlier arguments.

Concept in *Likelihood Ratio Processes and Bayesian Learning*	Concept in this lecture
States $\{f, g\}$	State space $S = \{s_1, s_2\}$
Densities $f(\cdot)$, $g(\cdot)$	Rows of experiment matrix μ
Single draw w_t	Blackwell experiment with continuous signal space
History w^t of t IID draws	Richer experiment μ_t Blackwell-dominating μ_s, $s < t$
Likelihood ratio $L(w^t)$	Sufficient statistic for μ_t
Prior π_0	Prior $p \in P$ on the 1-simplex $[0, 1]$
Posterior π_t	Posterior on $P = [0, 1]$ (Kihlstrom's standard experiment)
Distribution of π_t across histories	$\widehat{\mu}^c$ (Kihlstrom's posterior distribution)
Martingale property $E[\pi_t] = \pi_0$	Mean preservation of $\widehat{\mu}^c$
$\pi_t \to 0$ or 1 almost surely	Posteriors spread to vertices (MPS in the limit)
Mutual information $I(\mu_t; U_H)$	DeGroot value of information
More draws $\Rightarrow$ better for all decision makers	Blackwell ordering $\mu_t \geq \mu_s$
Garbling (discard last $t - s$ draws)	Stochastic transformation Q with $\mu_s = \mu_t Q$

CHAPTER

THIRTYONE

INFORMATION AND MARKET EQUILIBRIUM

Contents

- *Information and Market Equilibrium*
  - *Overview*
  - *Setup*
  - *Price revelation*
  - *Invertibility and the elasticity of substitution*
  - *Bayesian price expectations in a dynamic economy*
  - *Simulating Bayesian learning from prices*
  - *Exercises*

31.1 Overview

This lecture studies two questions about the **informational role of prices** posed and answered by Kihlstrom and Mirman [1975].

1. *When do prices transmit inside information?*
   - An informed insider observes a private signal correlated with an unknown state of the world and adjusts demand accordingly.
   - Equilibrium prices shift.
   - Under what conditions can an outside observer *infer* the insider's posterior distribution from the equilibrium price?
2. *Do Bayesian price expectations converge?*
   - In a stationary stochastic exchange economy, an uninformed observer uses the history of market prices and Bayes' Law to form beliefs about the economy's structure and hence about its induced price distribution.
   - Do those expectations eventually agree with those of a fully informed observer?

Kihlstrom and Mirman's answers rely on two classical ideas from statistics:

- **Blackwell sufficiency**: a random variable $\tilde{y}$ is said to be *sufficient* for a random variable $\tilde{y}'$ with respect to an unknown state if knowing $\tilde{y}$ gives all the information about the state that $\tilde{y}'$ contains.

- **Bayesian consistency**: as the sample grows, posterior beliefs eliminate models that imply the wrong **price distribution**, so even when structure is not identified from prices the posterior mass on the true **reduced form** still converges to one.

Important findings of Kihlstrom and Mirman [1975] are:

- Equilibrium prices transmit inside information *if and only if* the map from the insider's posterior distribution to the equilibrium price is one-to-one on the set of posteriors that can actually arise from the signal.
  - For the two-state case ($S = 2$), invertibility holds when the informed agent's utility is homothetic and the elasticity of substitution is everywhere either below one or above one.
- In the dynamic economy, as information accumulates, Bayesian price expectations converge to **rational expectations**, even when the deep structure is not identified from prices alone.

Note

Kihlstrom and Mirman [1975] use the terms "reduced form" and "structural" models in a way that careful econometricians do.

Reduced-form and structural models come in pairs.

To each structure or structural model there is a reduced form, or collection of reduced forms, underlying different possible regressions.

The lecture is organized as follows.

1. Set up the static two-commodity model and define equilibrium.
2. State the price-revelation theorem and the invertibility conditions.
3. Illustrate invertibility and its failure with numerical examples using CES and Cobb-Douglas preferences.
4. Introduce the dynamic stochastic economy and derive the Bayesian convergence result.
5. Simulate Bayesian learning from price observations.

This lecture builds on ideas in *Blackwell's Theorem on Comparing Experiments* and *Likelihood Ratio Processes and Bayesian Learning*.

We start by importing some Python packages.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import brentq
from scipy.stats import norm
```

31.2 Setup

31.2.1 Preferences, endowments, and the unknown state

The economy has two goods.

Good 2 is the numeraire (price normalized to 1).

Good 1 trades at price $p > 0$.

An unknown parameter $\bar{a}$ affects the value of good 1.

Agent i's expected utility from a bundle (x_1^i, x_2^i) is

$$U^i(x_1^i, x_2^i) = \sum_{s=1}^{S} u^i(a_s x_1^i,\ x_2^i)\, P^i(\bar{a} = a_s),$$

where P^i is agent i's subjective probability distribution over the finite state space $A = \{a_1, \ldots, a_S\}$.

Each agent starts with an endowment w^i of good 2 and a share θ^i of the representative firm.

In the paper's formal model, a single firm transforms good 2 into good 1 according to $y_1 = f(y_2)$ with $f' < 0$ and chooses production to maximize

$$\pi(p) = \max_{y_2 \leq 0} \{pf(y_2) + y_2\}.$$

The firm's profit π is then distributed to households according to the shares θ^i.

Agent i's budget constraint is

$$px_1^i + x_2^i = w^i + \theta^i \pi.$$

Agents maximize expected utility subject to their budget constraints.

A **competitive equilibrium** is a price $\hat{p}$ that clears both markets simultaneously.

Under the maintained convexity assumptions equilibrium exists, and following Kihlstrom and Mirman [1975] we assume the equilibrium price is unique, so that we can write $\hat{p} = p(\mu)$ as a well-defined function of the informed agent's posterior.

For most of what follows, the production side matters only through the induced equilibrium price map, so when we turn to numerical illustrations we will suppress production and use a pure-exchange / portfolio interpretation to keep the calculations transparent.

31.2.2 The informed agent's problem

Suppose **agent 1** (the insider) observes a private signal $\tilde{y}$ correlated with $\bar{a}$ before trading, where $\tilde{y}$ takes values in a finite set Y.

Before the signal arrives, agent 1 has prior beliefs $\mu_0 = P^1$.

Upon observing $\tilde{y} = y$, agent 1 updates to the **posterior** $\mu_y = (\mu_{y1}, \ldots, \mu_{yS})$ via Bayes' rule:

$$\mu_{ys} = P(\bar{a} = a_s \mid \tilde{y} = y).$$

Because agent 1's demand depends on μ_y, the new equilibrium price satisfies

$$\hat{p} = p(\mu_y).$$

Outside observers who see $\hat{p}$ but not $\tilde{y}$ can try to *back out* the insider's posterior from the price.

Define the set of realized posteriors

$$M = \{\mu_y : y \in Y,\ P(\tilde{y} = y) > 0\}.$$

The key question is whether the map $\mu \mapsto p(\mu)$ is one-to-one on M.

To answer that question, we now translate "information in prices" into Blackwell's language of sufficiency.

31.3 Price revelation

31.3.1 Blackwell sufficiency

The price variable $p(\mu_{\tilde{y}})$ *accurately transmits* the insider's private information if observing the equilibrium price is just as informative about $\bar{a}$ as observing the signal $\tilde{y}$ directly.

In Blackwell's language (Blackwell [1951] and Blackwell [1953]), this means $p(\mu_{\tilde{y}})$ is **sufficient** for $\tilde{y}$.

Definition 31.3.1 (Sufficiency)

A random variable $\tilde{y}$ is *sufficient* for $\tilde{y}'$ with respect to $\bar{a}$ if there exists a conditional distribution $P(y' \mid y)$, **independent of** $\bar{a}$, such that

$$\phi'_a(y') = \sum_{y \in Y} P(y' \mid y)\, \phi_a(y) \quad \text{for all } a \text{ and all } y',$$

where $\phi_a(y) = P(\tilde{y} = y \mid \bar{a} = a)$.

Thus, once $\tilde{y}$ is known, $\tilde{y}'$ provides no additional information about $\bar{a}$.

Kihlstrom and Mirman [1975] show that

Lemma 31.3.1 (Posterior Sufficiency)

The posterior distribution $\mu_{\tilde{y}}$ is a sufficient statistic for $\tilde{y}$.

Proof. (Sketch) The posterior $\mu_{\tilde{y}}$ satisfies

$$P(\bar{a} = a_s \mid \mu_{\tilde{y}} = \mu_y,\ \tilde{y} = y) = \mu_{ys} = P(\bar{a} = a_s \mid \mu_{\tilde{y}} = \mu_y).$$

This identity says that once the posterior is known, conditioning on the original signal $\tilde{y}$ does not change beliefs about $\bar{a}$.

Equivalently, the conditional law of $\tilde{y}$ given $\mu_{\tilde{y}}$ is independent of $\bar{a}$, so $\mu_{\tilde{y}}$ is sufficient for $\tilde{y}$ in Blackwell's sense.

Now let's think about the mapping from belief to price.

Theorem 31.3.1 (Price Revelation)

In the model outlined above, the price random variable $p(\mu_{\tilde{y}})$ is sufficient for the random variable $\tilde{y}$ if and only if the function $p(P^1)$ is invertible on the set of prices

$$\mathcal{P} = \Big\{ p(\mu_y) : y \in Y,\ P(\tilde{y} = y) = \sum_{a \in A} \phi_a(y)\, \mu_0(a) > 0 \Big\}.$$

The logic is

$$\tilde{y} \quad \longrightarrow \quad \mu_{\tilde{y}} \quad \longrightarrow \quad p(\mu_{\tilde{y}}).$$

The first arrow loses no information about $\bar{a}$ by *Lemma 31.3.1*, and the theorem asks when the second arrow also loses no information.

The proof has two parts.

If $p(\cdot)$ is one-to-one on M, then observing the price is equivalent to observing the posterior itself because

$$P(\mu_{\tilde{y}} = \mu \mid p(\mu_{\tilde{y}}) = p) = \begin{cases} 1 & \text{if } \mu = p^{-1}(p), \\ 0 & \text{otherwise.} \end{cases}$$

This conditional distribution is independent of the state, so price is sufficient for the posterior; together with *Lemma 31.3.1*, price is therefore sufficient for the signal.

Conversely, if two different posteriors in M generated the same price, an observer of the price could not tell which posterior had occurred, and the paper shows formally that in this case the conditional distribution of the posterior given price would depend on the state, so price could not be sufficient.

Before turning to invertibility itself, it helps to keep in mind the two economic interpretations emphasized in the paper.

31.3.2 Two interpretations

Insider trading in a stock market

Good 1 is a risky asset with random return $\bar{a}$; good 2 is "money".

An insider's demand reveals private information about the return.

If the invertibility condition holds, outside observers can read the insider's posterior distribution – the useful information the insider's signal carries about $\bar{a}$ – from the equilibrium stock price.

Price as a quality signal

Good 1 has uncertain quality $\bar{a}$.

Experienced consumers (who have sampled the good) observe a signal correlated with quality and buy accordingly.

Uninformed consumers can infer quality from the market price, provided invertibility holds.

31.4 Invertibility and the elasticity of substitution

When does the belief-to-price map fail to be invertible?

Theorem 31.4.1 shows that for a two-state economy ($S = 2$), the answer depends on the **elasticity of substitution** σ of agent 1's utility function.

Before stating the theorem, it helps to see the two intermediate steps in the paper's argument.

Lemma 31.4.1 (Same Price Implies Same Allocation)

Assume that u^i has continuous first partial derivatives and that u^i is quasi-concave.

Let $p \in \mathcal{P}$.

If there exist two measures μ^* and μ' in M such that $p(\mu^*, P^2, \ldots, P^n) = p(\mu', P^2, \ldots, P^n) = p$, then

$$x^i(\mu^*, P^2, \ldots, P^n) = x^i(\mu', P^2, \ldots, P^n), \quad i = 1, \ldots, n.$$

Fix the beliefs of all agents except agent 1.

The lemma says that if two posterior beliefs μ^* and μ' for agent 1 both support the same equilibrium price p, then they support the same equilibrium allocation for every trader.

The intuition is that when the price is unchanged, the demands of the uninformed traders are unchanged too, so market clearing forces the informed agent's bundle to be unchanged as well.

This lemma lets us define the informed agent's equilibrium bundle as a function of price alone:

$$x(p) = (x_1(p), x_2(p)).$$

Throughout, u^i_j denotes the partial derivative of u^i with respect to its j-th argument.

Whenever the informed agent consumes positive amounts of both goods, optimality of $x(p)$ under posterior μ gives the interior first-order condition

$$p = \frac{\sum_{s=1}^{S} a_s u_1^1(a_s x_1(p), x_2(p))\, \mu(a_s)}{\sum_{s=1}^{S} u_2^1(a_s x_1(p), x_2(p))\, \mu(a_s)}.$$

For a fixed price p, the bundle $x(p)$ is fixed too, so invertibility boils down to whether this equation admits a unique posterior μ.

Lemma 31.4.2 (Unique Posterior at a Given Price)

Assume that the first partial derivatives of u^1 exist and that u^1 is quasi-concave.

Also assume that agent 1 always consumes positive quantities of both goods.

Then $p(P^1)$ is invertible on $\mathcal{P}$ if for each $p \in \mathcal{P}$ there exists a unique probability measure $\mu \in M$ such that

$$\frac{\sum_{s=1}^{S} a_s\, u_1^1(a_s x_1(p), x_2(p))\, \mu(a_s)}{\sum_{s=1}^{S} u_2^1(a_s x_1(p), x_2(p))\, \mu(a_s)} = p.$$

If two different posteriors gave the same price, then by *Lemma 31.4.1* they would share the same bundle $x(p)$, contradicting uniqueness of the posterior that solves the first-order condition at that price.

31.4.1 The two-state first-order condition

With $S = 2$ and $\mu = (q,\ 1-q)$, define

$$\alpha_s(p) = a_s\, u_1^1(a_s x_1(p),\ x_2(p)), \qquad \beta_s(p) = u_2^1(a_s x_1(p),\ x_2(p)), \qquad s = 1, 2.$$

Then the first-order condition becomes

$$p = \frac{\alpha_1(p)\, q + \alpha_2(p)\,(1-q)}{\beta_1(p)\, q + \beta_2(p)\,(1-q)}.$$

At a fixed price p, the quantities $\alpha_s(p)$ and $\beta_s(p)$ are constants, so uniqueness of the posterior is the same as uniqueness of the scalar q solving this equation.

Theorem 31.4.1 (Invertibility Conditions)

Assume that the first partial derivatives of u^1 exist and that u^1 is quasi-concave and homothetic.

Also suppose that the informed agent always consumes positive quantities of both goods in all equilibrium allocations.

If $S = 2$ and the elasticity of substitution of u^1 is either always less than one or always greater than one, then $p(P^1)$ is invertible on $\mathcal{P}$.

If u^1 is Cobb-Douglas (elasticity of substitution constant and equal to one), then $p(P^1)$ is constant on $\mathcal{P}$.

When $\sigma = 1$ the income and substitution effects exactly cancel, so agent 1's demand for good 1 does not respond to changes in beliefs about $\bar{a}$.

Because the demand is unchanged, the market-clearing price is unchanged too, and the price reveals nothing about the insider's signal.

31.4.2 CES utility

For concreteness we work with a simplified example with the **constant-elasticity-of-substitution** (CES) utility function

$$u(c_1, c_2) = \left(c_1^\rho + c_2^\rho\right)^{1/\rho}, \qquad \rho \in (-\infty, 0) \cup (0, 1),$$

whose elasticity of substitution is $\sigma = 1/(1-\rho)$.

- $\rho \to 0$: Cobb-Douglas ($\sigma = 1$).
- $\rho < 0$: $\sigma < 1$ (complements).
- $0 < \rho < 1$: $\sigma > 1$ (substitutes).

Pertinent partial derivatives are

$$u_1(c_1, c_2) = \left(c_1^\rho + c_2^\rho\right)^{1/\rho - 1} c_1^{\rho-1}, \qquad u_2(c_1, c_2) = \left(c_1^\rho + c_2^\rho\right)^{1/\rho - 1} c_2^{\rho-1}.$$

This CES example is only an illustration, because the theorem itself covers any homothetic utility with elasticity everywhere above one or everywhere below one.

With that example in hand, we can compute the equilibrium price directly as a function of the posterior.

31.4.3 Equilibrium price as a function of the posterior

We focus on agent 1 as the *only* informed trader who absorbs one unit of good 1 at equilibrium (i.e., $x_1 = 1$).

Let $W_1 = w^1 + \theta^1 \pi$ denote agent 1's total wealth (endowment plus profit share).

Agent 1's budget constraint then reduces to $x_2 = W_1 - p$, and the equilibrium price is the unique $p \in (0, W_1)$ satisfying the first-order condition

$$p[q\, u_2(a_1,\, W_1 - p) + (1-q)\, u_2(a_2,\, W_1 - p)] = q\, a_1\, u_1(a_1,\, W_1 - p) + (1-q)\, a_2\, u_1(a_2,\, W_1 - p).$$

For Cobb-Douglas utility ($\sigma = 1$), the first-order condition becomes $p = W_1 - p$, giving $p^* = W_1/2$ regardless of the posterior q, confirming that no information is transmitted through the price in the Cobb-Douglas case.

We compute first-order conditions numerically below.

```
def ces_derivatives(c1, c2, ρ):
    """
    Return CES marginal utilities.

    Use the Cobb-Douglas limit near rho = 0.
    """
    if abs(ρ) < 1e-4:
        u1 = 0.5 * np.sqrt(c2 / c1)
        u2 = 0.5 * np.sqrt(c1 / c2)
    else:
        common = (c1**ρ + c2**ρ)**(1 / ρ - 1)
        u1 = common * c1**(ρ - 1)
        u2 = common * c2**(ρ - 1)
    return u1, u2


def eq_price(q, a1, a2, W1, ρ):
    """Return the equilibrium price for posterior q."""
    def residual(p):
        x2 = W1 - p
        u1_s1, u2_s1 = ces_derivatives(a1, x2, ρ)
        u1_s2, u2_s2 = ces_derivatives(a2, x2, ρ)
        lhs = p * (q * u2_s1 + (1 - q) * u2_s2)
        rhs = q * a1 * u1_s1 + (1 - q) * a2 * u1_s2
        return lhs - rhs

    try:
        return brentq(residual, 1e-6, W1 - 1e-6, xtol=1e-10)
    except ValueError:
        return np.nan
```

```
a1, a2 = 2.0, 0.5     # state values (a1 > a2)
W1 = 4.0

q_grid = np.linspace(0.05, 0.95, 200)

ρ_values = [-0.5, 0.0, 0.5]
ρ_labels = [
    r"$\rho = -0.5$ ($\sigma = 0.67$, complements)",
    r"$\rho = 0$ ($\sigma = 1$, Cobb-Douglas)",
    r"$\rho = 0.5$ ($\sigma = 2$, substitutes)",
]
```

(continues on next page)

(continued from previous page)

```
fig, ax = plt.subplots(figsize=(8, 5))

for ρ, label in zip(ρ_values, ρ_labels):
    prices = [eq_price(q, a1, a2, W1, ρ) for q in q_grid]
    ax.plot(q_grid, prices, label=label, lw=2)

ax.set_xlabel(r"posterior probability $q = \Pr(\bar{a} = a_1)$", fontsize=12)
ax.set_ylabel("equilibrium price $p^*(q)$", fontsize=12)
ax.legend(fontsize=10)
plt.tight_layout()
plt.show()
```

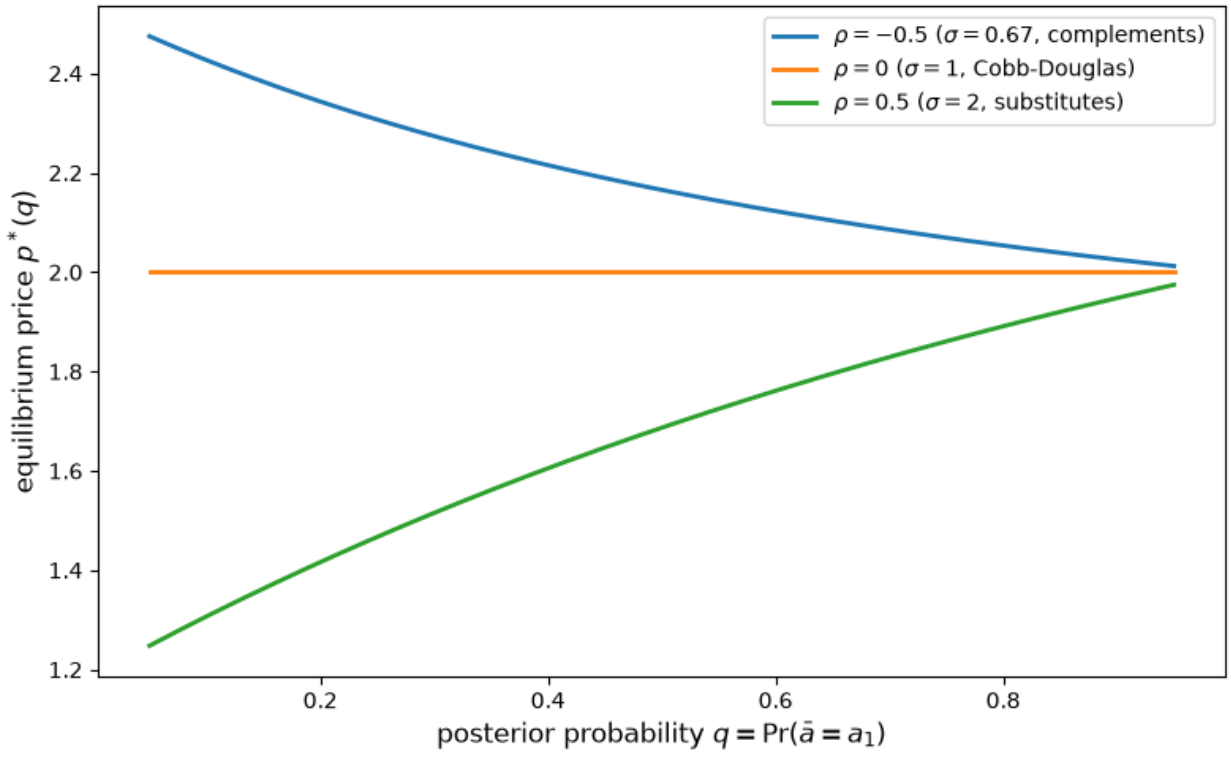


Fig. 31.1: equilibrium price vs posterior

The plot confirms *Theorem 31.4.1*.

For CES with $\sigma \neq 1$, the equilibrium price is strictly monotone in q.

An outside observer who knows the equilibrium map $p^*(\cdot)$ can therefore invert the price uniquely to recover q, so the inside information is fully transmitted.

For Cobb-Douglas ($\sigma = 1$), the price is flat in q, so information is never transmitted through the market.

```
p_cd = [eq_price(q, a1, a2, W1, ρ=0.0) for q in q_grid]

print(f"Cobb-Douglas (rho=0): min p* = {min(p_cd):.6f}, "
      f"max p* = {max(p_cd):.6f}, "
```

(continues on next page)

(continued from previous page)

```
      f"range = {max(p_cd)-min(p_cd):.2e}")
print(f"Analytical CD price  = W1/2 = {W1/2:.6f}")
```

```
Cobb-Douglas (rho=0): min p* = 2.000000, max p* = 2.000000, range = 8.88e-16
Analytical CD price  = W1/2 = 2.000000
```

Every entry equals $W_1/2 = 2.0$ exactly, confirming analytically that the Cobb-Douglas equilibrium price is independent of q and of the state values a_1, a_2.

The numerical plot shows monotonicity, and the next subsection connects that pattern back to the proof of *Theorem 31.4.1*.

31.4.4 Why monotonicity depends on σ

Fix a price p and treat $\alpha_s(p)$ and $\beta_s(p)$ as constants.

The right-hand side of the two-state first-order condition

$$\frac{\alpha_1(p)\,q + \alpha_2(p)\,(1-q)}{\beta_1(p)\,q + \beta_2(p)\,(1-q)}$$

is then a function of q alone, with derivative

$$\frac{\partial}{\partial q}\frac{\alpha_1 q + \alpha_2(1-q)}{\beta_1 q + \beta_2(1-q)} = \frac{\alpha_1\beta_2 - \alpha_2\beta_1}{\left[\beta_1 q + \beta_2(1-q)\right]^2}.$$

So the sign is determined by $\alpha_1\beta_2 - \alpha_2\beta_1$, and if that sign is constant then for each fixed price there is at most one posterior weight q consistent with the first-order condition, which is exactly what *Theorem 31.4.1* requires.

Using

$$\frac{\alpha_s}{\beta_s} = \frac{a_s\,u_1(a_s x_1, x_2)}{u_2(a_s x_1, x_2)} = a_s^{(\sigma-1)/\sigma}\left(\frac{x_2}{x_1}\right)^{1/\sigma},$$

one can show

$$\frac{\partial}{\partial a}\frac{\alpha}{\beta} = \frac{(\sigma-1)}{\sigma}\,a^{-1/\sigma}\left(\frac{x_2}{x_1}\right)^{1/\sigma}.$$

For the CES specification, this derivative is positive when $\sigma > 1$, negative when $\sigma < 1$, and *zero when* $\sigma = 1$.

In other words, for CES utility the ratio α_s/β_s moves monotonically with the state value a_s unless $\sigma = 1$, which makes the fixed-price first-order-condition expression monotone in q and in turn delivers invertibility.

The vanishing derivative in the Cobb-Douglas case means the marginal rate of substitution is independent of a_s, so the informed agent's demand, and hence the equilibrium price, does not respond to changes in beliefs.

Let us visualize the ratio α_s/β_s as a function of a_s for different values of σ:

```
a_vals = np.linspace(0.3, 3.0, 300)
x1_fix, x2_fix = 1.0, 1.0

fig, ax = plt.subplots(figsize=(7, 4))
for ρ in [-0.5, -1e-6, 0.5]:
    σ = 1 / (1 - ρ) if abs(ρ) > 1e-8 else 1.0
    ratios = []
    for a in a_vals:
```

(continues on next page)

(continued from previous page)

```
        u1, u2 = ces_derivatives(a * x1_fix, x2_fix, ρ)
        ratios.append(a * u1 / u2)
    ax.plot(a_vals, ratios, label=rf"$\sigma = {σ:.2f}$", lw=2)

ax.set_xlabel(r"state value $a_s$", fontsize=12)
ax.set_ylabel(r"$\alpha_s / \beta_s = a_s u_1 / u_2$", fontsize=12)
ax.axhline(y=1.0, color="black", lw=0.8, ls="--")
ax.legend(fontsize=10)
plt.tight_layout()
plt.show()
```

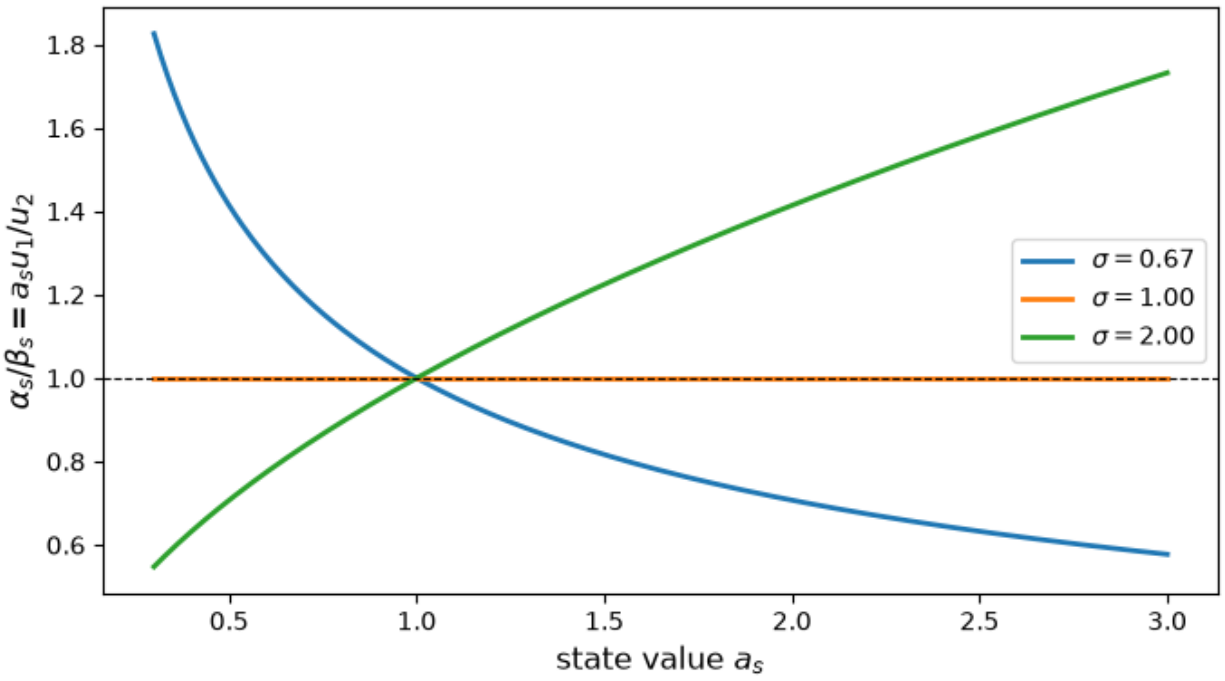


Fig. 31.2: marginal rate of substitution

When $\sigma = 1$ the ratio is constant across all a_s values, so information about the state has no effect on the marginal rate of substitution.

For $\sigma < 1$ the ratio is decreasing in a_s, and for $\sigma > 1$ it is increasing, making the equilibrium price strictly monotone in the posterior q in both cases.

The static analysis asks whether a current price reveals current private information, whereas the next section asks what a whole history of prices reveals over time.

31.5 Bayesian price expectations in a dynamic economy

We now turn to a question addressed in Section 3 of Kihlstrom and Mirman [1975].

31.5.1 A stochastic exchange economy

Time is discrete: $t = 1, 2, ...$

In each period t:

1. Consumer i receives a random endowment ω_i^t.
2. Markets open; competitive prices $p^t = p(\omega^t)$ clear all markets.
3. Consumers trade and consume.

The endowment vectors $\{\tilde{\omega}^t\}$ are **i.i.d.** with density $f(\omega^t \mid \lambda)$, where $\lambda = (\lambda_1, ..., \lambda_K)$ is a **structural parameter vector** (of dimension K) that is *fixed but unknown*.

The equilibrium price at time t is a deterministic function of ω^t, so $\{p^t\}$ is also i.i.d.

For any measurable price set P, let

$$W(P) = \{\omega^t : p(\omega^t) \in P\}.$$

Then

$$P_\lambda(p^t \in P) = P_\lambda(\omega^t \in W(P)) = \int_{W(P)} f(\omega^t \mid \lambda)\, d\omega^t.$$

The induced price density is denoted by $g(p^t \mid \lambda)$.

For a given structure λ, this density is the observable implication of the model, and when several structures imply the same density we group them into a single reduced-form class.

The next issue is therefore what an observer can and cannot infer about the structure from price data alone.

31.5.2 The identification problem

Because price observations identify only the induced price density $g(\cdot \mid \lambda)$, and because the structural-to-reduced-form map $\lambda \mapsto g(\cdot \mid \lambda)$ may be many-to-one, price data may identify only a reduced-form class rather than the exact structure.

In particular, it may be impossible to recover λ from $g(p \mid \lambda)$ even with infinite price data.

To handle this, partition Λ into equivalence classes μ such that $\lambda \in \mu$ and $\lambda' \in \mu$ whenever $g(p \mid \lambda) = g(p \mid \lambda')$ for all p.

The equivalence class μ containing the true λ is the **reduced form** relevant for price data.

An observer who knows the infinite price history learns μ but not necessarily λ.

Once that distinction is clear, Bayesian updating can be written down directly.

31.5.3 Bayesian updating

An uninformed observer begins with a prior $h(\lambda)$ over $\lambda \in \Lambda$.

If the observer could see endowments directly, the posterior would be

$$h(\lambda \mid \omega^1, \dots, \omega^t) = \frac{h(\lambda) \prod_{\tau=1}^{t} f(\omega^\tau \mid \lambda)}{\sum_{\lambda' \in \Lambda} h(\lambda') \prod_{\tau=1}^{t} f(\omega^\tau \mid \lambda')},$$

and the paper appeals to a Bayesian consistency result to conclude that this posterior concentrates on the true structure $\bar{\lambda}$.

After observing the price sequence $(p^1, \dots, p^t)$, the observer's Bayesian posterior is

$$h(\lambda \mid p^1, \dots, p^t) = \frac{h(\lambda) \prod_{\tau=1}^{t} g(p^\tau \mid \lambda)}{\sum_{\lambda' \in \Lambda} h(\lambda') \prod_{\tau=1}^{t} g(p^\tau \mid \lambda')}.$$

Price data cannot distinguish structures inside the same reduced-form class.

Indeed, if λ and λ' belong to the same class μ, then $g(\cdot \mid \lambda) = g(\cdot \mid \lambda')$, so

$$\frac{h(\lambda \mid p^1, \dots, p^t)}{h(\lambda' \mid p^1, \dots, p^t)} = \frac{h(\lambda)}{h(\lambda')}$$

for every sample history, so the relative odds within an observationally equivalent class never change.

At time t, the observer's price expectations for the next period are

$$g(p^{t+1} \mid p^1, \dots, p^t) = \sum_{\lambda \in \Lambda} g(p^{t+1} \mid \lambda)\, h(\lambda \mid p^1, \dots, p^t).$$

31.5.4 The convergence theorem

Theorem 31.5.1 (Bayesian Convergence)

Let $\bar{\lambda}$ be the true structural parameter and $\bar{\mu}$ the reduced form that contains $\bar{\lambda}$.

Assume the prior assigns positive probability to the reduced-form class $\bar{\mu}$.

Define the posterior mass on a reduced-form class by

$$H_t(\mu) = \sum_{\lambda \in \mu} h(\lambda \mid p^1, \dots, p^t).$$

Because all structures inside a class imply the same $g(\cdot \mid \lambda)$, the predictive density can equivalently be written as

$$g(p^{t+1} \mid p^1, \dots, p^t) = \sum_{\mu} g(p^{t+1} \mid \mu)\, H_t(\mu).$$

Then

$$\lim_{t \to \infty} H_t(\mu) = \begin{cases} 1 & \text{if } \mu = \bar{\mu}, \\ 0 & \text{otherwise,} \end{cases}$$

with probability one.

Consequently,

$$\lim_{t\to\infty} g(p^{t+1} \mid p^1, \ldots, p^t) = g(p \mid \bar{\mu}),$$

which equals the rational-expectations price distribution for a fully informed observer.

Note

Note that the theorem only requires the prior to assign positive probability to the reduced-form class $\bar{\mu}$ that contains the true structure $\bar{\lambda}$.

This is implied by, but weaker than, assigning positive probability to the true structural parameter $\bar{\lambda}$ itself.

A prior could place zero mass on $\bar{\lambda}$ while still placing positive mass on other structures inside $\bar{\mu}$.

The important distinction is that price observers need not learn $\bar{\lambda}$ itself.

They only learn which reduced-form class is correct.

That is enough for forecasting because every $\lambda \in \bar{\mu}$ generates the same price density $g(\cdot \mid \bar{\mu})$.

Rational price expectations emerge from learning the reduced form, not from identifying every structural detail of the economy.

Here "rational expectations" means that the observer's predictive distribution for next period's price matches the objective price distribution generated by the true reduced form.

Let's now turn to a simple simulation.

31.6 Simulating Bayesian learning from prices

We illustrate the theorem with a two-state example.

Two possible reduced forms μ_1 and μ_2 generate prices $p^t \sim N(\bar{p}_i, \sigma_p^2)$ for $i = 1, 2$ respectively.

The observer knows the two possible price distributions (the reduced forms) but not which one governs the data.

This is a **Bayesian model selection** problem we have seen in *Likelihood Ratio Processes and Bayesian Learning*.

With a prior h_0 on μ_1 and the observed price p^t, the posterior weight on μ_1 after period t is

$$h_t = \frac{h_{t-1}\, g(p^t \mid \mu_1)}{h_{t-1}\, g(p^t \mid \mu_1) + (1 - h_{t-1})\, g(p^t \mid \mu_2)}.$$

We consider a numerical example with two normal distributions with different means

```
def simulate_bayesian_learning(
    p_bar_true, p_bar_alt, σ_p, T, h0, n_paths, seed=42
):
    """Simulate posterior learning between two Gaussian reduced forms."""
    rng = np.random.default_rng(seed)
    h_paths = np.zeros((n_paths, T + 1))
    h_paths[:, 0] = h0

    for path in range(n_paths):
        h = h0
```

(continues on next page)

(continued from previous page)

```
        prices = rng.normal(p_bar_true, σ_p, size=T)
        for t, p in enumerate(prices):
            g_true = norm.pdf(p, loc=p_bar_true, scale=σ_p)
            g_alt = norm.pdf(p, loc=p_bar_alt, scale=σ_p)
            denom = h * g_true + (1 - h) * g_alt
            h = h * g_true / denom
            h_paths[path, t + 1] = h

    return h_paths


def plot_bayesian_learning(h_paths, p_bar_true, p_bar_alt, ax):
    """Plot posterior beliefs over time."""
    T = h_paths.shape[1] - 1
    t_grid = np.arange(T + 1)

    for path in h_paths:
        ax.plot(t_grid, path, alpha=0.25, lw=0.8, color="steelblue")

    median_path = np.median(h_paths, axis=0)
    ax.plot(t_grid, median_path, color="navy", lw=2, label="median posterior")

    ax.axhline(
        y=1.0,
        color="black",
        ls="--",
        lw=1.2,
        label="true model weight = 1",
    )
    ax.set_xlabel("period $t$", fontsize=12)
    ax.set_ylabel(r"$h_t$ = posterior weight on true model", fontsize=12)
    ax.legend(fontsize=10)
```

We consider two cases, one that is easy to learn and another one that is harder to learn, using $T = 300$ periods, $n = 40$ simulated paths, a diffuse prior $h_0 = 0.5$, and common standard deviation $\sigma_p = 0.4$.

- *Easy case*: true model $N(2.0,\ 0.4^2)$, alternative $N(1.2,\ 0.4^2)$.
- *Hard case*: true model $N(2.0,\ 0.4^2)$, alternative $N(1.8,\ 0.4^2)$.

Whether easy or hard to learn depends on "how close" the true distribution is compared to the alternative hypothesis.

```
T = 300
h0 = 0.5      # diffuse prior
n_paths = 40
σ_p = 0.4

fig, axes = plt.subplots(1, 2, figsize=(12, 5))

# Distinct reduced forms
p_bar_true, p_bar_alt = 2.0, 1.2
h_paths = simulate_bayesian_learning(p_bar_true, p_bar_alt, σ_p, T, h0, n_paths)
plot_bayesian_learning(h_paths, p_bar_true, p_bar_alt, axes[0])

# Similar reduced forms
p_bar_true, p_bar_alt = 2.0, 1.8
h_paths_hard = simulate_bayesian_learning(
    p_bar_true, p_bar_alt, σ_p, T, h0, n_paths
```

(continues on next page)

(continued from previous page)

```
)
plot_bayesian_learning(h_paths_hard, p_bar_true, p_bar_alt, axes[1])

plt.tight_layout()
plt.show()
```

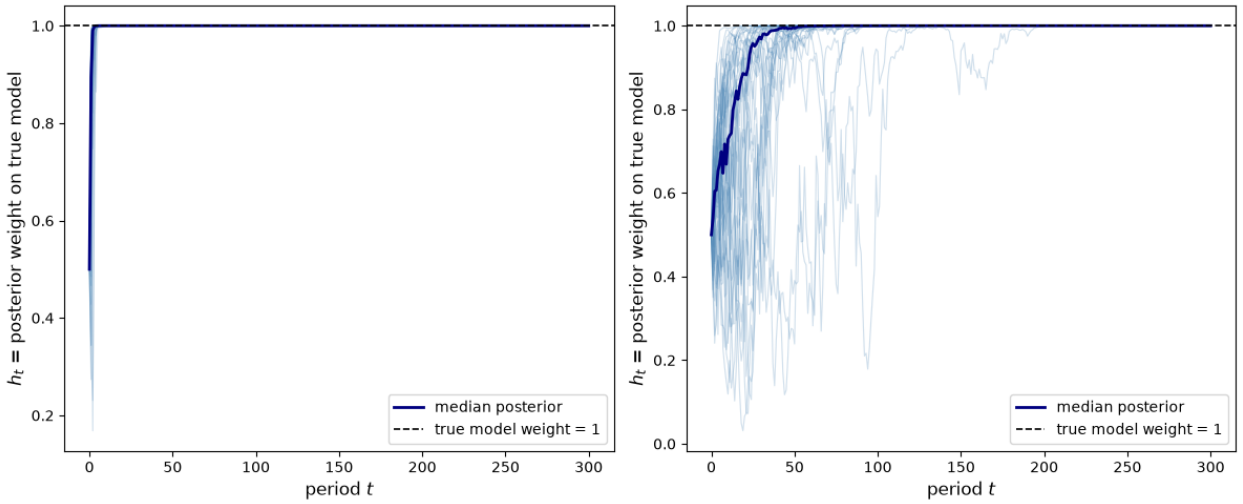


Fig. 31.3: bayesian learning across paths

In both panels the posterior weight on the true model converges to 1 with probability one, though convergence is slower when the two price distributions are similar (right panel).

31.6.1 Price expectations vs. rational expectations

We now verify that the observer's price expectations converge to the rational-expectations distribution $g(p \mid \bar{\mu})$.

We use the parameterization of the "hard-to-learn" example above ($\bar{p}_{\text{true}} = 2.0$, $\bar{p}_{\text{alt}} = 1.8$, $\sigma_p = 0.4$), extending to $T = 1{,}000$ periods with a single simulated path and prior $h_0 = 0.5$

```
def price_expectation(h_t, p_bar_true, p_bar_alt, σ_p, p_grid):
    """Return the predictive price density at posterior weight h_t."""
    return (
        h_t * norm.pdf(p_grid, loc=p_bar_true, scale=σ_p)
        + (1 - h_t) * norm.pdf(p_grid, loc=p_bar_alt, scale=σ_p)
    )


p_bar_true, p_bar_alt = 2.0, 1.8
σ_p = 0.4
n_paths = 1
T_long = 1000

h_paths_long = simulate_bayesian_learning(
    p_bar_true, p_bar_alt, σ_p, T_long, h0=0.5, n_paths=n_paths, seed=7
)
```

(continues on next page)

(continued from previous page)

```
p_grid = np.linspace(0.0, 3.5, 300)
re_density = norm.pdf(p_grid, loc=p_bar_true, scale=σ_p)

fig, ax = plt.subplots(figsize=(8, 5))
snapshots = [0, 25, 100, 300, 1000]
palette   = plt.cm.Blues(np.linspace(0.3, 1.0, len(snapshots)))

for t_snap, col in zip(snapshots, palette):
    h_t = h_paths_long[0, t_snap]
    dens = price_expectation(h_t, p_bar_true, p_bar_alt, σ_p, p_grid)
    ax.plot(
        p_grid,
        dens,
        color=col,
        lw=2,
        label=rf"$t = {t_snap}$, $h_t = {h_t:.3f}$",
    )

ax.plot(p_grid, re_density, "k--", lw=2,
        label=r"rational expectations $g(p \mid \bar{\mu})$")
ax.set_xlabel("price $p$", fontsize=12)
ax.set_ylabel("density", fontsize=12)
ax.legend(fontsize=9)
plt.tight_layout()
plt.show()
```

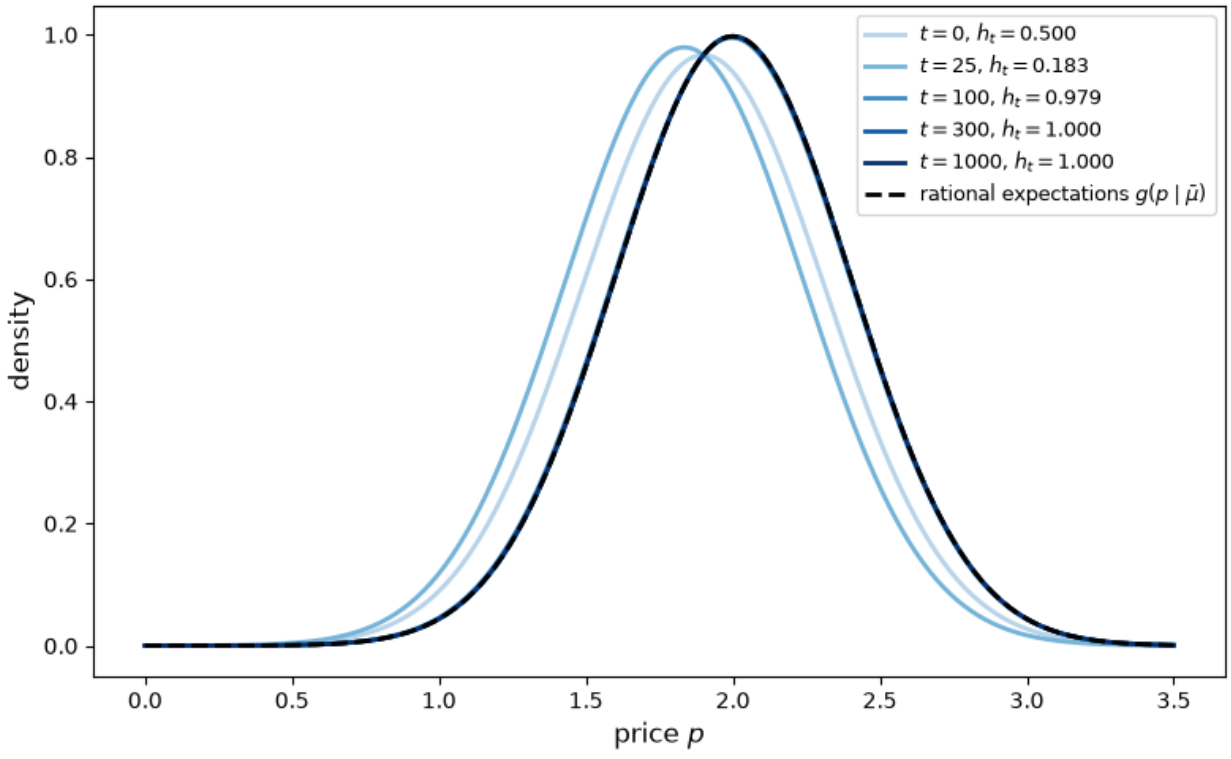


Fig. 31.4: price distribution convergence

The sequence of predictive densities (shades of blue) converges to the rational-expectations density (dashed black line) as experience accumulates.

This illustrates *Theorem 31.5.1*.

We can now sharpen the point by looking at a case in which the reduced form is learned but the underlying structure is not.

31.6.2 Learning the reduced form without identifying the structure

The convergence result is particularly striking because the observer converges to *rational expectations* even when the underlying **structure** λ is *not identified* by prices.

To illustrate this, consider a case with *three* possible structures $\lambda^{(1)}, \lambda^{(2)}, \lambda^{(3)}$ but only *two* reduced forms $\mu_1 = \{\lambda^{(1)}, \lambda^{(2)}\}$ and $\mu_2 = \{\lambda^{(3)}\}$ (because $\lambda^{(1)}$ and $\lambda^{(2)}$ generate the same price distribution).

We continue with the hard-to-learn parameterization, so the three structures have price means $\bar{p}_1 = \bar{p}_2 = 2.0$ and $\bar{p}_3 = 1.8$, with common standard deviation $\sigma_p = 0.4$, a uniform prior $h_0 = (1/3, 1/3, 1/3)$, and $T = 400$ periods over 30 paths.

The true structure is $\lambda^{(1)}$.

```
def simulate_learning_3struct(
    T, h0_vec, p_bar_vec, σ_p, true_idx, n_paths, seed=0
):
    """Simulate learning with three structures and two reduced forms."""
    rng = np.random.default_rng(seed)
    h_paths = np.zeros((n_paths, T + 1, 3))
    h_paths[:, 0, :] = h0_vec

    for path in range(n_paths):
        h = np.array(h0_vec, dtype=float)
        prices = rng.normal(p_bar_vec[true_idx], σ_p, size=T)
        for t, p in enumerate(prices):
            likelihoods = norm.pdf(p, loc=p_bar_vec, scale=σ_p)
            h = h * likelihoods
            h /= h.sum()
            h_paths[path, t + 1, :] = h

    return h_paths


# Structures 0 and 1 share the same reduced form
p_bar_vec = np.array([2.0, 2.0, 1.8])
h0_vec = np.array([1 / 3, 1 / 3, 1 / 3])
σ_p = 0.4
T = 400
true_idx = 0     # Structure 0 is observationally equivalent to 1

h_paths_3 = simulate_learning_3struct(
    T, h0_vec, p_bar_vec, σ_p, true_idx, n_paths=30
)
t_grid = np.arange(T + 1)

fig, axes = plt.subplots(1, 3, figsize=(13, 4), sharey=True)
struct_labels = [
```

(continues on next page)

(continued from previous page)

```
    r"$\lambda^{(1)}$",
    r"$\lambda^{(2)}$",
    r"$\lambda^{(3)}$",
]

for k, (ax, label) in enumerate(zip(axes, struct_labels)):
    for path in h_paths_3:
        ax.plot(t_grid, path[:, k], alpha=0.25, lw=0.8, color="steelblue")
    ax.plot(t_grid, np.median(h_paths_3[:, :, k], axis=0),
            color="navy", lw=2, label=f"median weight on {label}")
    ax.set_xlabel("period $t$", fontsize=11)
    ax.legend(fontsize=9)

axes[0].set_ylabel("posterior weight", fontsize=11)
plt.tight_layout()
plt.show()
```

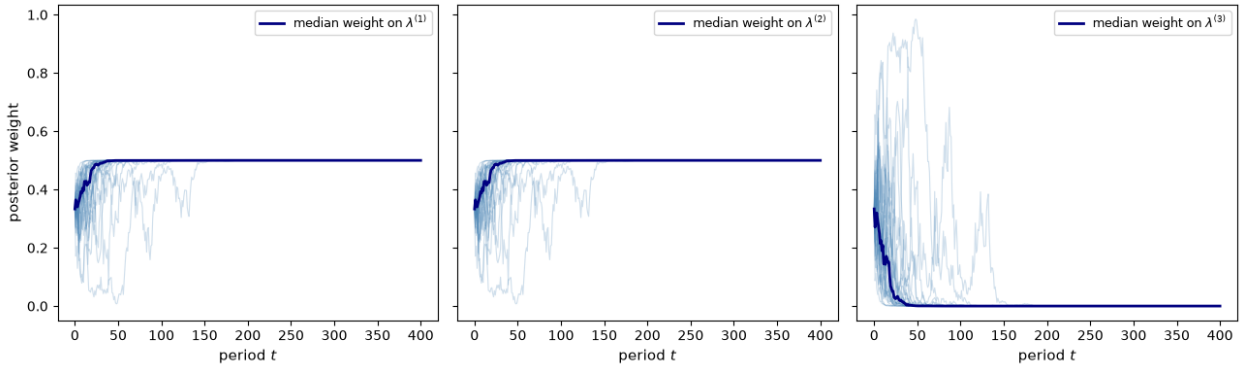


Fig. 31.5: learning with non-identification

The observer correctly rules out $\lambda^{(3)}$ (the wrong reduced form) with probability one, but cannot distinguish $\lambda^{(1)}$ from $\lambda^{(2)}$ because they generate an identical price distribution.

Nevertheless, the observer's **price expectations** converge to rational expectations because both structures imply the same reduced form $\bar{\mu}$.

31.7 Exercises

Exercise 31.7.1

CARA portfolio utility and the stock-market interpretation.

Consider a two-state economy ($a_1 = 2$, $a_2 = 0.5$) where the informed agent has **CARA** (constant absolute risk aversion) preferences over portfolio wealth:

$$u(W) = -e^{-\gamma W}, \quad W = x_2 + \bar{a}\, x_1.$$

The agent chooses x_1 to maximize

$$q\, u(W_1) + (1-q)\, u(W_2), \quad W_s = w - p\, x_1 + a_s\, x_1,$$

subject to the budget constraint $p\,x_1 + x_2 = w$.

Total supply of good 1 is $X_1 = 1$.

1. Derive the first-order condition for the informed agent's optimal x_1.
2. Use the market-clearing condition $x_1 = 1$ (the informed agent absorbs the entire supply) to obtain an implicit equation for the equilibrium price $p^*(q)$, and solve it numerically for $q \in (0, 1)$ and several values of γ.
3. Show *analytically* that $p^*(q)$ admits the closed form

$$p^*(q) = \frac{a_2 + R(q,\gamma)\,a_1}{1 + R(q,\gamma)}, \qquad R(q,\gamma) = \frac{q}{1-q}\,e^{-\gamma(a_1 - a_2)},$$

and verify that $p^*(q)$ is strictly increasing in q.

Solution

For the first-order condition, define $W_s = w + (a_s - p)\,x_1$ for $s = 1, 2$.

Then the FOC is

$$q\,(a_1 - p)\,\gamma\,e^{-\gamma W_1} = (1-q)\,(p - a_2)\,\gamma\,e^{-\gamma W_2},$$

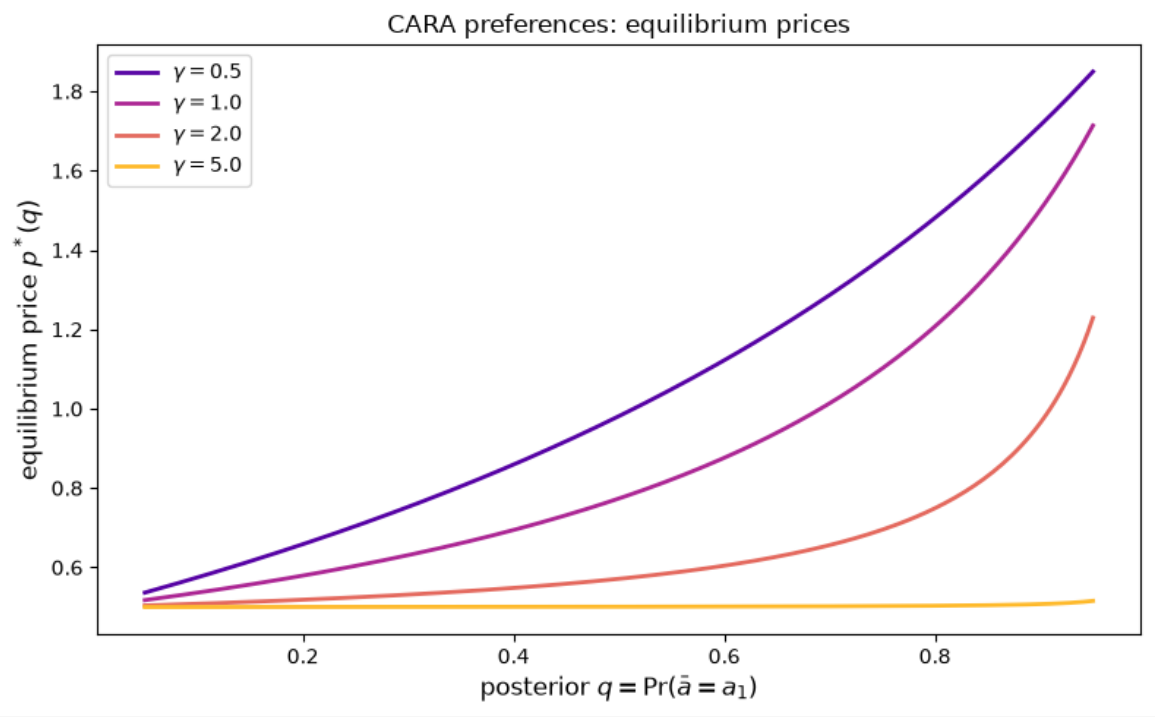
or equivalently (dividing by γ and rearranging)

$$q\,(a_1 - p)\,e^{-\gamma(a_1 - p)x_1} = (1-q)\,(p - a_2)\,e^{\gamma(p - a_2)x_1}.$$

Setting $x_1 = 1$ (the informed agent absorbs all supply), this becomes a scalar root-finding problem in p:

$$F(p;\,q,\gamma) \equiv q\,(a_1 - p)\,e^{-\gamma(a_1 - p)} - (1-q)\,(p - a_2)\,e^{\gamma(p - a_2)} = 0.$$

```
from scipy.optimize import brentq

def F_cara(p, q, a1, a2, γ, x1=1.0):
    """Residual for the CARA equilibrium condition."""
    return (q * (a1 - p) * np.exp(-γ * (a1 - p) * x1)
            - (1 - q) * (p - a2) * np.exp(γ * (p - a2) * x1))

a1, a2 = 2.0, 0.5
q_grid = np.linspace(0.05, 0.95, 200)
γ_values = [0.5, 1.0, 2.0, 5.0]
colors_sol = plt.cm.plasma(np.linspace(0.15, 0.85, len(γ_values)))

fig, ax = plt.subplots(figsize=(8, 5))
for γ, color in zip(γ_values, colors_sol):
    p_eq = [brentq(F_cara, a2, a1,
                   args=(q, a1, a2, γ))
            for q in q_grid]
    ax.plot(q_grid, p_eq, lw=2, color=color,
            label=rf"$\gamma = {γ}$")

ax.set_xlabel(r"posterior $q = \Pr(\bar a = a_1)$", fontsize=12)
ax.set_ylabel("equilibrium price $p^*(q)$", fontsize=12)
ax.set_title("CARA preferences: equilibrium prices", fontsize=12)
ax.legend(fontsize=10)
plt.tight_layout()
plt.show()
```



The price is strictly increasing in q for every $\gamma > 0$.

For the closed form, start from the FOC at $x_1 = 1$, divide both sides by $(a_1 - p)(p - a_2)$, and combine the exponentials:

$$\frac{q\,(a_1 - p)}{(1-q)\,(p - a_2)} = e^{\gamma(a_1 - a_2)}.$$

Rearranging gives

$$\frac{p - a_2}{a_1 - p} = \frac{q}{1-q}\, e^{-\gamma(a_1 - a_2)} \equiv R(q, \gamma),$$

and solving the resulting linear equation in p yields

$$p^*(q) = \frac{a_2 + R(q,\gamma)\, a_1}{1 + R(q,\gamma)}.$$

Since $R(q, \gamma)$ is strictly increasing in q and $dp^*/dR = (a_1 - a_2)/(1 + R)^2 > 0$, the equilibrium price $p^*(q)$ is strictly increasing in q.

This exercise uses the stock-market interpretation emphasized by Kihlstrom and Mirman [1975].

Portfolio wealth is $W = x_2 + \bar{a}\, x_1$, so ax_1 and x_2 are perfect substitutes in each state.

Hence the elasticity of substitution between the two arguments of $u(ax_1, x_2)$ is infinite, corresponding to the $\sigma > 1$ side of *Theorem 31.4.1*.

The difference is that this example is not the full equilibrium model the theorem analyzes, but rather a partial equilibrium model with a single informed agent and a fixed supply of the risky asset.

Exercise 31.7.2

In the Bayesian learning simulation, the speed of convergence to rational expectations is determined by the **Kullback-Leibler divergence** between the two reduced forms.

The KL divergence $D_{KL}(\mu_1 \| \mu_2)$ from $g(\cdot \mid \mu_1)$ to $g(\cdot \mid \mu_2)$, for two normal distributions with means $\bar{p}_1$ and $\bar{p}_2$ and common variance σ_p^2, is

$$D_{KL}(\mu_1 \| \mu_2) = \frac{(\bar{p}_1 - \bar{p}_2)^2}{2\sigma_p^2},$$

which is symmetric in the two means under equal variances.

1. For the "easy" case ($\bar{p}_1 = 2.0$, $\bar{p}_2 = 1.2$) and the "hard" case ($\bar{p}_1 = 2.0$, $\bar{p}_2 = 1.8$), compute D_{KL} for $\sigma_p = 0.4$.
2. Re-run the simulations from the lecture for both cases with $n = 100$ paths. For each path compute the first period $T_{0.99}$ at which $h_t \geq 0.99$. Plot histograms of $T_{0.99}$ for both cases.
3. How does the median $T_{0.99}$ scale with D_{KL}? Verify numerically that roughly $T_{0.99} \approx C/D_{KL}$ for some constant C.

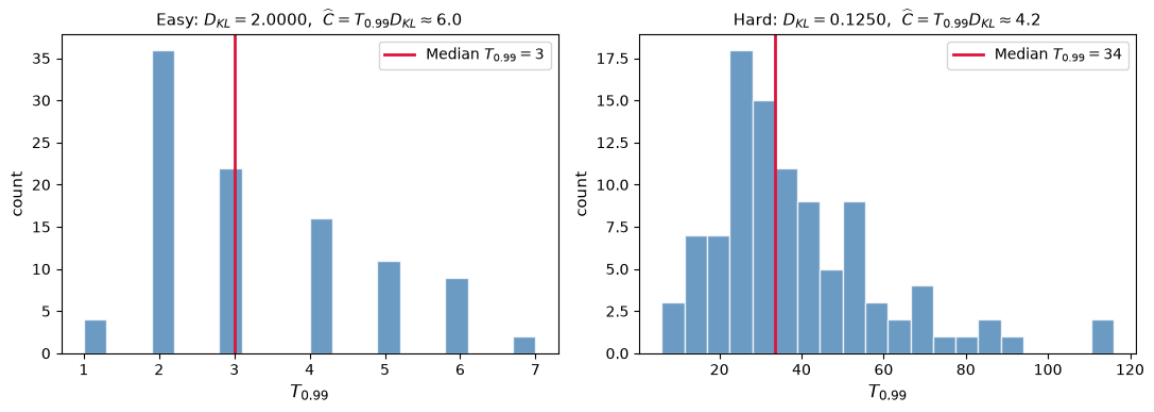
Solution

Here is one solution:

```
σ_p = 0.4

def kl_normal(p1, p2, σ):
    """Return the KL divergence for N(p1, σ^2) and N(p2, σ^2)."""
    return (p1 - p2)**2 / (2 * σ**2)

cases = [("Easy",  2.0, 1.2), ("Hard", 2.0, 1.8)]
for name, p1, p2 in cases:
    kl = kl_normal(p1, p2, σ_p)
    print(f"{name} case: D_KL = {kl:.4f}")

n_paths = 100

fig, axes = plt.subplots(1, 2, figsize=(11, 4))
for ax, (name, p1, p2) in zip(axes, cases):
    kl = kl_normal(p1, p2, σ_p)
    paths = simulate_bayesian_learning(p1, p2, σ_p, T=2000,
                                       h0=0.5, n_paths=n_paths, seed=42)
    # First period with posterior >= 0.99
    T99 = []
    for path in paths:
        idx = np.where(path >= 0.99)[0]
        T99.append(idx[0] if len(idx) > 0 else 2001)

    median_T = np.median(T99)
    ax.hist(T99, bins=20, color="steelblue", edgecolor="white", alpha=0.8)
    ax.axvline(median_T, color="crimson", lw=2,
               label=fr"Median $T_{{0.99}} = {median_T:.0f}$")
    ax.set_title(
        f"{name}: $D_{{KL}} = {kl:.4f}$,  "
        fr"$\widehat C = T_{{0.99}} D_{{KL}} \approx {median_T * kl:.1f}$",
        fontsize=11
```

```
    )
    ax.set_xlabel(r"$T_{0.99}$", fontsize=12)
    ax.set_ylabel("count", fontsize=11)
    ax.legend(fontsize=10)

plt.tight_layout()
plt.show()
```

```
Easy case: D_KL = 2.0000
Hard case: D_KL = 0.1250
```



The median $T_{0.99}$ scales as approximately C/D_{KL}, confirming that learning is faster when the two reduced forms are more easily distinguished (large D_{KL}).

Exercise 31.7.3

Theorem 31.5.1 requires the prior to assign positive probability to the true reduced-form class $\bar{\mu}$, equivalently to some structure that generates the true price distribution $g(\cdot \mid \bar{\mu})$.

In this exercise the true reduced form itself is excluded from the prior support, so we investigate what happens when no model in the prior generates the true price distribution.

Simulate $T = 1,000$ periods of prices from $N(2.0, 0.4^2)$ but use a prior that places equal weight on two *wrong* models: $N(1.5, 0.4^2)$ and $N(2.3, 0.4^2)$.

Plot the posterior weight on each model over time.

Discuss your findings.

Solution

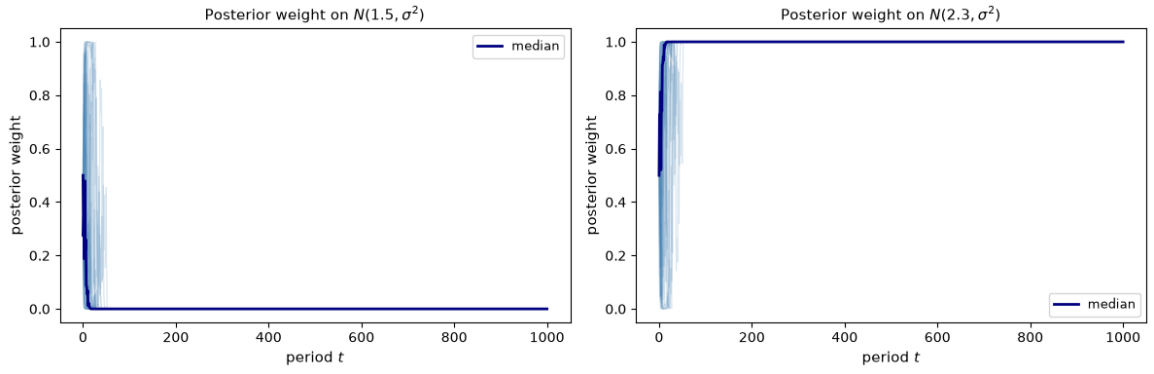
Here is one solution:

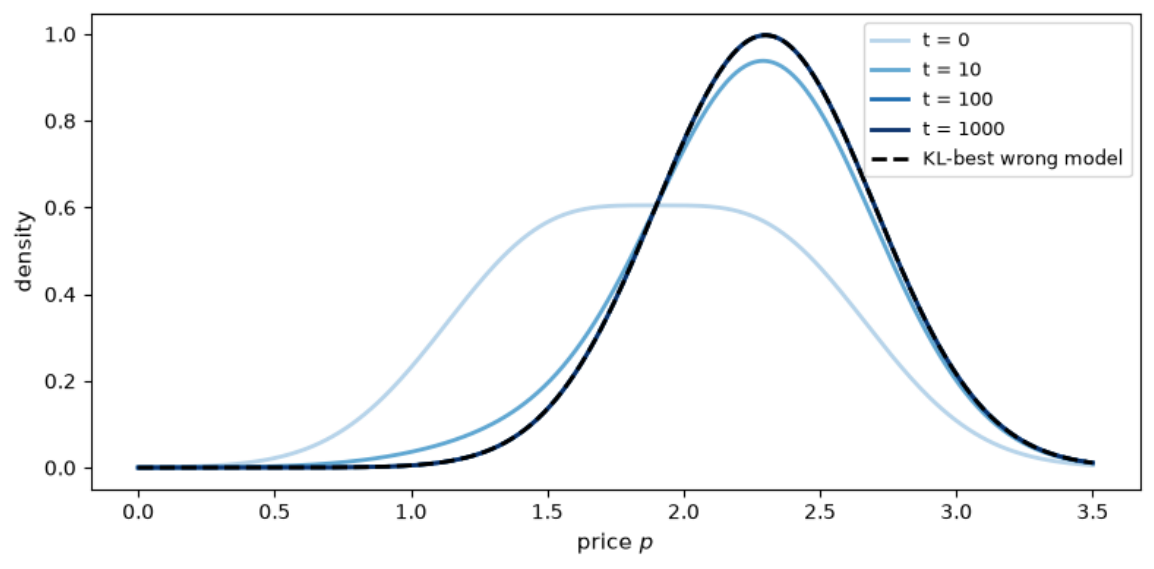
```
def simulate_misspecified(
    T, p_bar_true, p_bar_wrong, σ_p, h0, n_paths, seed=0
):
    """Simulate learning under a misspecified two-model prior."""
    rng = np.random.default_rng(seed)
    h_paths = np.zeros((n_paths, T + 1, 2))
    h_paths[:, 0, :] = h0
```

```
    for path in range(n_paths):
        h = np.array(h0, dtype=float)
        prices = rng.normal(p_bar_true, σ_p, size=T)
        for t, price in enumerate(prices):
            likes = norm.pdf(price, loc=p_bar_wrong, scale=σ_p)
            h = h * likes
            h /= h.sum()
            h_paths[path, t + 1, :] = h

    return h_paths


def predictive_density(weights, means, σ_p, p_grid):
    """Return the predictive density under the current posterior weights."""
    density = np.zeros_like(p_grid)
    for weight, mean in zip(weights, means):
        density += weight * norm.pdf(p_grid, loc=mean, scale=σ_p)
    return density


T = 1000
p_true = 2.0
p_wrong = np.array([1.5, 2.3])
σ_p = 0.4
h0 = np.array([0.5, 0.5])
n_paths = 30

h_misspec = simulate_misspecified(T, p_true, p_wrong, σ_p, h0, n_paths)

kl_vals = (p_true - p_wrong)**2 / (2 * σ_p**2)
for mean, kl in zip(p_wrong, kl_vals):
    print(f"KL(true || N({mean:.1f}, σ^2)) = {kl:.4f}")

t_grid = np.arange(T + 1)
fig, axes = plt.subplots(1, 2, figsize=(12, 4))

labels = [r"$N(1.5, \sigma^2)$", r"$N(2.3, \sigma^2)$"]
for ax, k, label in zip(axes, [0, 1], labels):
    for path in h_misspec:
        ax.plot(t_grid, path[:, k], alpha=0.2, lw=0.8, color="steelblue")
    ax.plot(t_grid, np.median(h_misspec[:, :, k], axis=0),
            color="navy", lw=2, label="median")
    ax.set_title(f"Posterior weight on {label}", fontsize=11)
    ax.set_xlabel("period $t$", fontsize=11)
    ax.set_ylabel("posterior weight", fontsize=11)
    ax.legend(fontsize=9)

plt.tight_layout()
plt.show()

# Predictive density and mean along the median posterior path
median_path = np.median(h_misspec, axis=0)
p_grid = np.linspace(0.0, 3.5, 300)
closer_idx = np.argmin(kl_vals)

fig, ax = plt.subplots(figsize=(8, 4))
colors = plt.cm.Blues(np.linspace(0.3, 1.0, 4))
```

```
for t_snap, color in zip([0, 10, 100, T], colors):
    dens = predictive_density(median_path[t_snap], p_wrong, σ_p, p_grid)
    ax.plot(p_grid, dens, color=color, lw=2, label=f"t = {t_snap}")

ax.plot(
    p_grid,
    norm.pdf(p_grid, loc=p_wrong[closer_idx], scale=σ_p),
    "k--",
    lw=2,
    label="KL-best wrong model",
)
ax.set_xlabel("price $p$", fontsize=11)
ax.set_ylabel("density", fontsize=11)
ax.legend(fontsize=9)
plt.tight_layout()
plt.show()

pred_mean = np.median(
    h_misspec[:, :, 0] * p_wrong[0] + h_misspec[:, :, 1] * p_wrong[1], axis=0
)
print(f"True mean: {p_true}")
print(f"Predictive mean at T={T}: {pred_mean[-1]:.4f}")
print(f"Closer misspecified mean: {p_wrong[np.argmin(kl_vals)]:.1f}")
```

```
KL(true || N(1.5, σ^2)) = 0.7812
KL(true || N(2.3, σ^2)) = 0.2812
True mean: 2.0
Predictive mean at T=1000: 2.3000
Closer misspecified mean: 2.3
```





Here

$$D_{KL}(N(2.0, 0.4^2) \,\|\, N(2.3, 0.4^2)) < D_{KL}(N(2.0, 0.4^2) \,\|\, N(1.5, 0.4^2)),$$

so the model with mean 2.3 is the KL-best approximation among the two wrong models, and in the simulation posterior weight concentrates on that model.

Posterior odds are cumulative *likelihood ratios*.

If we compare the two wrong Gaussian models f and g, then under the true distribution h the average log likelihood ratio satisfies

$$\frac{1}{t} E_h[\log L_t] = K(h, g) - K(h, f).$$

So if f is KL-closer to h than g is, $\log L_t$ has positive drift and posterior odds tilt toward f.

CHAPTER

THIRTYTWO

INCORRECT MODELS

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

```
!pip install numpyro
```

32.1 Overview

This is a sequel to *Likelihood Ratio Processes and Bayesian Learning*.

We discuss two ways to create a compound lottery and their consequences.

A compound lottery can be said to create a *mixture distribution*.

Our two ways of constructing a compound lottery will differ in their *timing*.

- in one, mixing between two possible probability distributions will occur once and all at the beginning of time
- in the other, mixing between the same two possible probability distributions will occur each period

The statistical setting is close but not identical to the problem studied in that lecture.

In that lecture, there were two IID processes that could possibly govern successive draws of a non-negative random variable W.

Nature decided once and for all whether to make a sequence of IID draws from either f or from g.

That lecture studied an agent who knew both f and g but did not know which distribution nature chose at time -1.

The agent represented that ignorance by assuming that nature had chosen f or g by flipping an unfair coin that put probability π_{-1} on probability distribution f.

That assumption allowed the agent to construct a subjective joint probability distribution over the random sequence $\{W_t\}_{t=0}^{\infty}$.

We studied how the agent would then use the laws of conditional probability and an observed history $w^t = \{w_s\}_{s=0}^t$ to form

$$\pi_t = \mathbb{E}[\text{nature chose distribution } f \mid w^t], \quad t = 0, 1, 2, ...$$

However, in the setting of this lecture, that rule imputes to the agent an incorrect model.

The reason is that now the wage sequence is actually described by a different statistical model.

Thus, we change the specification in *Likelihood Ratio Processes and Bayesian Learning* in the following way.

Now, **each period** $t \geq 0$, nature flips a possibly unfair coin that comes up f with probability α and g with probability $1 - \alpha$.

Thus, nature perpetually draws from the **mixture distribution** with CDF

$$H(w) = \alpha F(w) + (1 - \alpha)G(w), \quad \alpha \in (0, 1)$$

We'll study two agents who try to learn about the wage process, but who use different statistical models.

Both types of agent know f and g but neither knows α.

Our first type of agent erroneously thinks that at time -1, nature once and for all chose f or g and thereafter permanently draws from that distribution.

Our second type of agent knows, correctly, that nature mixes f and g with mixing probability $\alpha \in (0, 1)$ each period, though the agent doesn't know the mixing parameter.

Our first type of agent applies the learning algorithm described in *Likelihood Ratio Processes and Bayesian Learning*.

In the context of the statistical model that prevailed in that lecture, that was a good learning algorithm and it enabled the Bayesian learner eventually to learn the distribution that nature had drawn at time -1.

This is because the agent's statistical model was *correct* in the sense of being aligned with the data generating process.

But in the present context, our type 1 decision maker's model is incorrect because the model h that actually generates the data is neither f nor g and so is beyond the support of the models that the agent thinks are possible.

Nevertheless, we'll see that our first type of agent muddles through and eventually learns something interesting and useful, even though it is not *true*.

Instead, it turns out that our type 1 agent who is armed with a wrong statistical model ends up learning whichever probability distribution, f or g, is in a special sense *closest* to the h that actually generates the data.

We'll tell the sense in which it is closest.

Our second type of agent understands that nature mixes between f and g each period with a fixed mixing probability α.

But the agent doesn't know α.

The agent sets out to learn α using Bayes' law applied to his model.

His model is correct in the sense that it includes the actual data generating process h as a possible distribution.

In this lecture, we'll learn about

- how nature can *mix* between two distributions f and g to create a new distribution h.
- The Kullback-Leibler statistical divergence that governs statistical learning under an incorrect statistical model
- A useful Python function `numpy.searchsorted` that, in conjunction with a uniform random number generator, can be used to sample from an arbitrary distribution

As usual, we'll start by importing some Python tools.

```
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
from scipy.integrate import quad

import numpyro
import jax
import numpyro.distributions as dist
from numpyro.infer import MCMC, NUTS
import jax.numpy as jnp
import jax.scipy.stats as jsp
from jax import random
from jax.scipy.special import gamma

# Enable JAX to use 64-bit float operations
jax.config.update("jax_enable_x64", True)

colors = sns.color_palette()
```

Let's use Python to generate two Beta distributions.

```
# Parameters in the two beta distributions.
F_a, F_b = 1, 1
G_a, G_b = 3, 1.2


@jax.jit
def p(x, a, b):
    r = gamma(a + b) / (gamma(a) * gamma(b))
    return r * x**(a-1) * (1 - x)**(b-1)

# The two density functions.
f = jax.jit(lambda x: p(x, F_a, F_b))
g = jax.jit(lambda x: p(x, G_a, G_b))
```

```
def simulate(key, a, b, T=50, N=500):
    """
    Generate N sets of T observations of the likelihood ratio,
    return as N x T matrix.
    """
    w = jax.random.beta(key, a, b, (N, T))
    l_arr = f(w) / g(w)
    return l_arr
```

We'll also use the following Python code to prepare some informative simulations.

```
key = jax.random.key(42)
key_g, key_f = jax.random.split(key)

l_arr_g = simulate(key_g, G_a, G_b, N=50000)
l_seq_g = np.cumprod(l_arr_g, axis=1)
```

```
l_arr_f = simulate(key_f, F_a, F_b, N=50000)
l_seq_f = np.cumprod(l_arr_f, axis=1)
```

32.2 Sampling from compound lottery H

We implement two methods to draw samples from our mixture model $\alpha F + (1-\alpha)G$.

We'll generate samples using each of them and verify that they match well.

Here is pseudo code for a direct "method 1" for drawing from our compound lottery:

- Step one:
  - use a uniform random draw to flip an unfair coin that selects distribution F with prob α and G with prob $1-\alpha$
- Step two:
  - draw from either F or G, as determined by the coin flip.
- Step three:
  - put the first two steps in a big loop and do them for each realization of w

Our second method uses a uniform distribution and the following fact that we also described and used in *Elementary Probability with Matrices*:

- If a random variable X has CDF F, then a random variable $F^{-1}(U)$ also has CDF F, where U is a uniform random variable on $[0, 1]$.

In other words, if $X \sim F(x)$ we can generate a random sample from F by drawing a random sample from a uniform distribution on $[0, 1]$ and computing $F^{-1}(U)$.

We'll use this fact in conjunction with the `numpy.searchsorted` command to sample from H directly.

See the `numpy.searchsorted` documentation for details on the `searchsorted` function.

See the Mr. P Solver video on Monte Carlo simulation to see other applications of this powerful trick.

In the Python code below, we'll use both of our methods and confirm that each of them does a good job of sampling from our target mixture distribution.

```
def draw_lottery(key, p, N):
    """Draw from the compound lottery directly."""
    # split the key: one for the coin flip, one for each distribution
    key_uniform, key_beta1, key_beta2 = jax.random.split(key, 3)

    u = jax.random.uniform(key_uniform, shape=(N,))
    f_draws = jax.random.beta(key_beta1, F_a, F_b, shape=(N,))
    g_draws = jax.random.beta(key_beta2, G_a, G_b, shape=(N,))

    # select F with probability p, else G
    return jnp.where(u <= p, f_draws, g_draws)


# JIT-compile with `N` as a static argument
draw_lottery = jax.jit(draw_lottery, static_argnums=(2,))
```

```
def draw_lottery_MC(key, p, N):
    "Draw from the compound lottery using the Monte Carlo trick."

    xs = jnp.linspace(1e-8, 1 - (1e-8), 10000)
    CDF = p * jsp.beta.cdf(xs, F_a, F_b) + (1 - p) * jsp.beta.cdf(xs, G_a, G_b)

    Us = jax.random.uniform(key, shape=(N,))
```

(continues on next page)

(continued from previous page)

```
    draws = xs[jnp.searchsorted(CDF[:-1], Us)]
    return draws
```

```
# verify
N = 100000
α = 0.0

key1, key2 = jax.random.split(jax.random.key(42))
sample1 = draw_lottery(key1, α, N)
sample2 = draw_lottery_MC(key2, α, N)

# plot draws and density function
plt.hist(sample1, 50, density=True, alpha=0.5, label="direct draws")
plt.hist(sample2, 50, density=True, alpha=0.5, label="MC draws")

xs = np.linspace(0, 1, 1000)
plt.plot(xs, α * f(xs) + (1 - α) * g(xs), color="red", label="density", lw=2)

plt.legend()
plt.show()
```

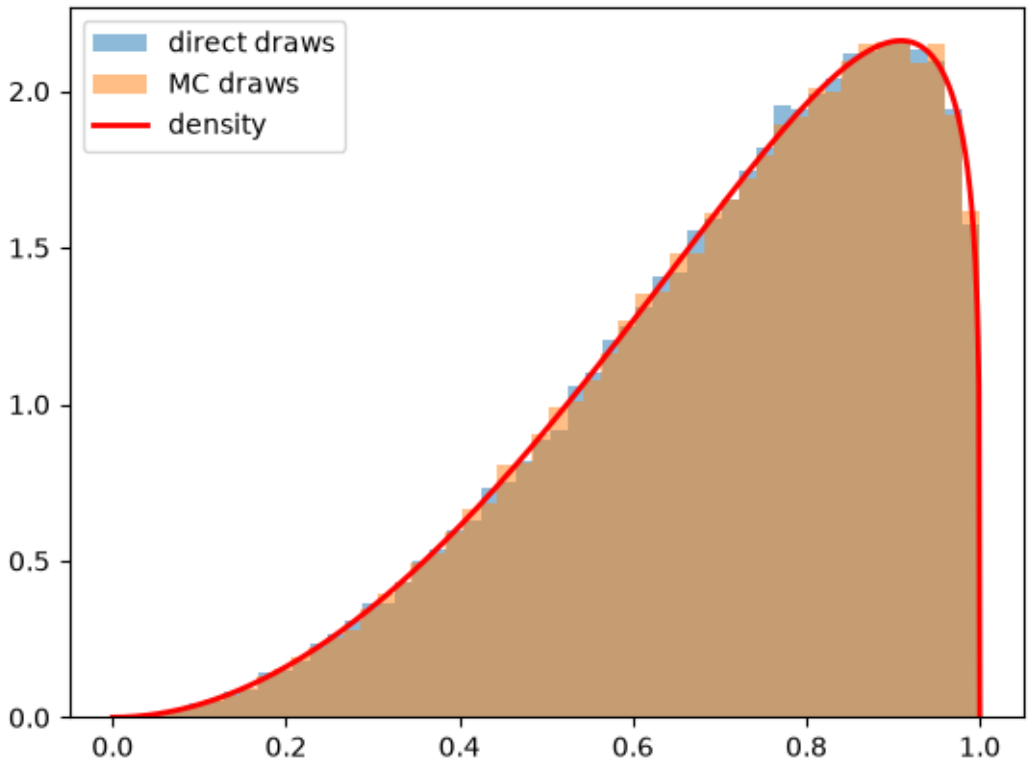


Fig. 32.1: Direct and Monte Carlo draws

32.3 Type 1 agent

We'll now study what our type 1 agent learns.

Remember that our type 1 agent uses the wrong statistical model, thinking that nature mixed between f and g once and for all at time -1.

The type 1 agent thus uses the learning algorithm studied in *Likelihood Ratio Processes and Bayesian Learning*.

We'll briefly review that learning algorithm now.

Let π_t be a Bayesian posterior defined as

$$\pi_t = \mathbb{P}\{q = f \mid w^t\}$$

The likelihood ratio process plays a principal role in the formula that governs the evolution of the posterior probability π_t, an instance of **Bayes' law**.

Bayes' law implies that $\{\pi_t\}$ obeys the recursion

$$\pi_t = \frac{\pi_{t-1} l_t(w_t)}{\pi_{t-1} l_t(w_t) + 1 - \pi_{t-1}} \tag{32.1}$$

with π_0 being a Bayesian prior probability that $q = f$, i.e., a personal or subjective belief about q based on our having seen no data.

Below we define a Python function that updates belief π using likelihood ratio ℓ according to recursion (32.1)

```
@jax.jit
def update(π, l):
    """Update π using likelihood l"""
    # Update belief
    π = π * l / (π * l + 1 - π)
    return π
```

Formula (32.1) can be generalized by iterating on it and thereby deriving an expression for the time t posterior π_{t+1} as a function of the time 0 prior π_0 and the likelihood ratio process $L(w^{t+1})$ at time t.

To begin, notice that the updating rule

$$\pi_{t+1} = \frac{\pi_t \ell\left(w_{t+1}\right)}{\pi_t \ell\left(w_{t+1}\right) + \left(1 - \pi_t\right)}$$

implies

$$\begin{aligned}
\frac{1}{\pi_{t+1}} &= \frac{\pi_t \ell\left(w_{t+1}\right) + \left(1 - \pi_t\right)}{\pi_t \ell\left(w_{t+1}\right)} \\
&= 1 - \frac{1}{\ell\left(w_{t+1}\right)} + \frac{1}{\ell\left(w_{t+1}\right)} \frac{1}{\pi_t}. \\
\Rightarrow \frac{1}{\pi_{t+1}} - 1 &= \frac{1}{\ell\left(w_{t+1}\right)} \left(\frac{1}{\pi_t} - 1\right).
\end{aligned}$$

Therefore

$$\frac{1}{\pi_{t+1}} - 1 = \frac{1}{\prod_{i=1}^{t+1} \ell\left(w_i\right)} \left(\frac{1}{\pi_0} - 1\right) = \frac{1}{L\left(w^{t+1}\right)} \left(\frac{1}{\pi_0} - 1\right).$$

Since $\pi_0 \in (0, 1)$ and $L\left(w^{t+1}\right) > 0$, we can verify that $\pi_{t+1} \in (0, 1)$.

After rearranging the preceding equation, we can express π_{t+1} as a function of $L\left(w^{t+1}\right)$, the likelihood ratio process at $t+1$, and the initial prior π_0

$$\pi_{t+1} = \frac{\pi_0 L\left(w^{t+1}\right)}{\pi_0 L\left(w^{t+1}\right) + 1 - \pi_0}. \tag{32.2}$$

Formula (32.2) generalizes formula (32.1).

Formula (32.2) can be regarded as a one step revision of prior probability π_0 after seeing the batch of data $\{w_i\}_{i=1}^{t+1}$.

32.4 What a type 1 agent learns when mixture H generates data

We now study what happens when the mixture distribution $h;\alpha$ truly generated the data each period.

The sequence π_t continues to converge, despite the agent's misspecified model, and the limit is either 0 or 1.

This is true even though in truth nature always mixes between f and g.

After verifying that claim about possible limit points of π_t sequences, we'll drill down and study what fundamental force determines the limiting value of π_t.

Let's set a value of α and then watch how π_t evolves.

```
def simulate_mixed(key, α, T=50, N=500):
    """
    Generate N sets of T observations of the likelihood ratio,
    return as N x T matrix, when the true density is mixed h;a
    """

    w_s = draw_lottery(key, α, N * T).reshape(N, T)
    l_arr = f(w_s) / g(w_s)
    return l_arr


def plot_π_seq(key, α, π1=0.2, π2=0.8, T=200):
    """
    Compute and plot π_seq and the log likelihood ratio process
    when the mixed distribution governs the data.
    """

    l_arr_mixed = simulate_mixed(key, α, T=T, N=50)
    l_seq_mixed = np.cumprod(l_arr_mixed, axis=1)

    T = l_arr_mixed.shape[1]
    π_seq_mixed = np.empty((2, T + 1))
    π_seq_mixed[:, 0] = π1, π2

    for t in range(T):
        for i in range(2):
            π_seq_mixed[i, t + 1] = update(
                π_seq_mixed[i, t], l_arr_mixed[0, t]
            )

    # plot
    fig, ax1 = plt.subplots()
    for i in range(2):
        ax1.plot(
```

(continues on next page)

(continued from previous page)

```
            range(T + 1),
            π_seq_mixed[i, :],
            label=rf"$\pi_0$={π_seq_mixed[i, 0]}",
            lw=2,
        )

    ax1.plot(
        np.nan, np.nan, "--", color="b",
        label="Log likelihood ratio process", lw=2,
    )
    ax1.set_ylabel(r"$\pi_t$")
    ax1.set_xlabel("$t$")
    ax1.legend()

    ax2 = ax1.twinx()
    ax2.plot(range(1, T + 1), np.log(l_seq_mixed[0, :]), "--", color="b", lw=2)
    ax2.set_ylabel(r"$\log(L(w^{t}))$")

    plt.show()
```

```
plot_π_seq(jax.random.key(42), α=0.6)
```

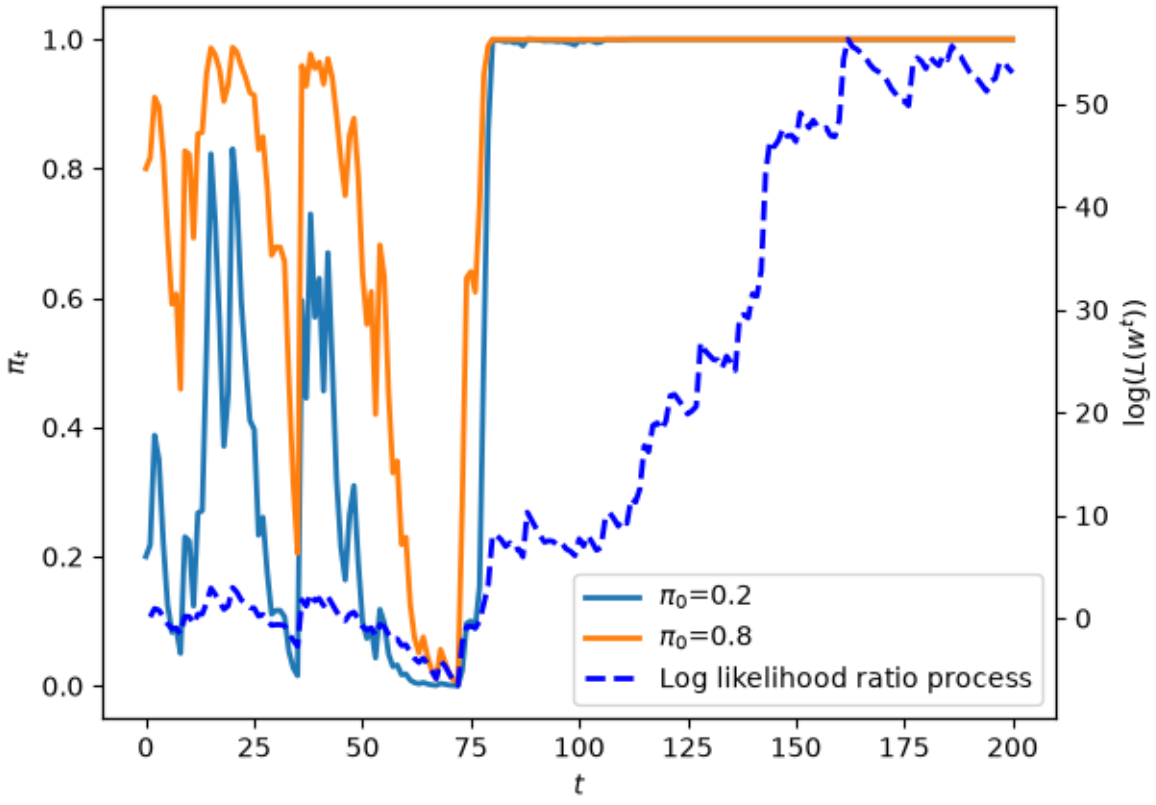


Fig. 32.2: Belief paths, $\alpha = 0.6$

The above graph shows a sample path of the log likelihood ratio process as the blue dotted line, together with sample

paths of π_t that start from two distinct initial conditions.

Let's see what happens when we change α.

```
plot_π_seq(jax.random.key(42), α=0.2)
```

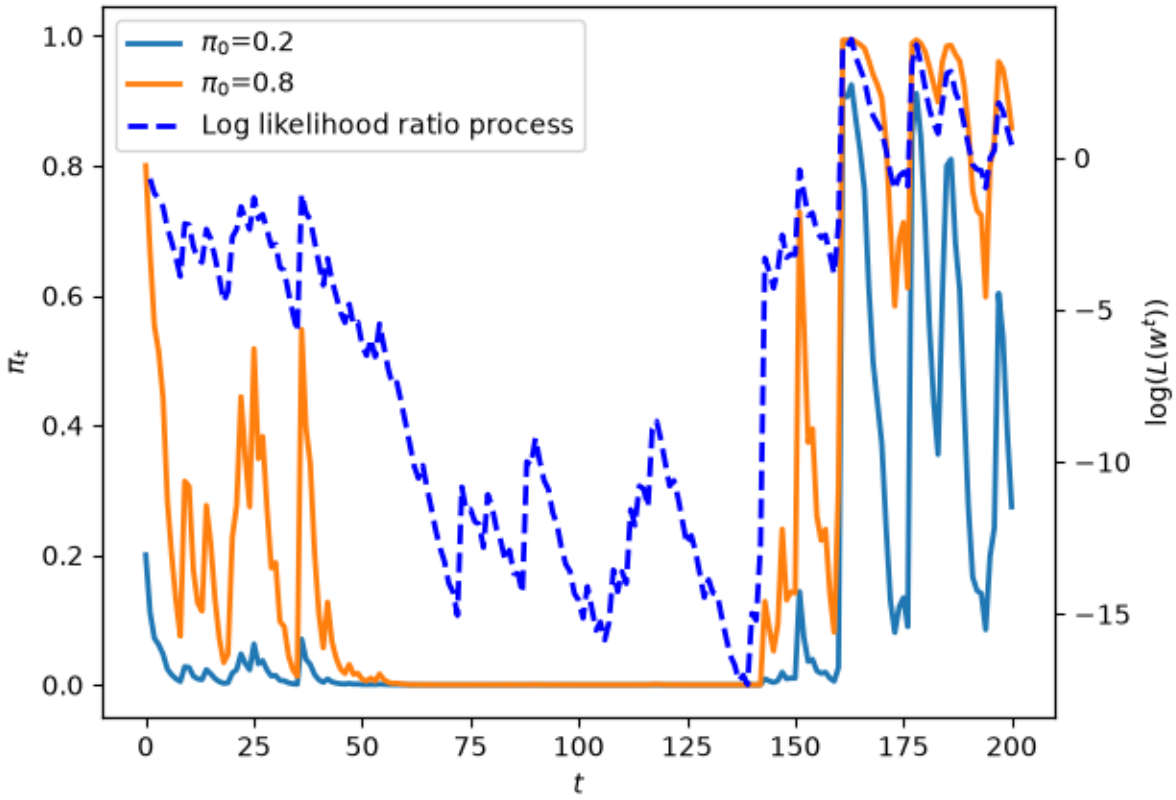


Fig. 32.3: Belief paths, $\alpha = 0.2$

Evidently, α is having a big effect on the destination of π_t as $t \to +\infty$.

32.5 Kullback-Leibler divergence governs limit of π_t

To understand what determines whether the limit point of π_t is 0 or 1 and how the answer depends on the true value of the mixing probability $\alpha \in (0, 1)$ that generates

$$h(w) \equiv h(w|\alpha) = \alpha f(w) + (1 - \alpha) g(w)$$

we shall compute the following two Kullback-Leibler divergences

$$KL_g(\alpha) = \int \log\left(\frac{h(w)}{g(w)}\right) h(w) dw$$

and

$$KL_f(\alpha) = \int \log\left(\frac{h(w)}{f(w)}\right) h(w) dw$$

We shall plot both of these functions against α as we use α to vary $h(w) = h(w|\alpha)$.

The limit of π_t is determined by

$$\min_{f,g}\{KL_g, KL_f\}$$

The only possible limits are 0 and 1.

As $t \to +\infty$, π_t goes to one if and only if $KL_f < KL_g$.

```
@jax.jit
def KL_g(α):
    "Compute the KL divergence KL(h, g)."
    err = 1e-8  # to avoid 0 at end points
    ws = jnp.linspace(err, 1 - err, 10000)
    gs, fs = g(ws), f(ws)
    hs = α * fs + (1 - α) * gs
    return jnp.sum(jnp.log(hs / gs) * hs) / 10000


KL_g_v = jax.vmap(KL_g)


@jax.jit
def KL_f(α):
    "Compute the KL divergence KL(h, f)."
    err = 1e-8  # to avoid 0 at end points
    ws = jnp.linspace(err, 1 - err, 10000)
    gs, fs = g(ws), f(ws)
    hs = α * fs + (1 - α) * gs
    return jnp.sum(jnp.log(hs / fs) * hs) / 10000


KL_f_v = jax.vmap(KL_f)


# compute KL using quad in Scipy
def KL_g_quad(α):
    "Compute the KL divergence KL(h, g) using scipy.integrate."
    h = lambda x: α * f(x) + (1 - α) * g(x)
    return quad(lambda x: h(x) * np.log(h(x) / g(x)), 0, 1)[0]


def KL_f_quad(α):
    "Compute the KL divergence KL(h, f) using scipy.integrate."
    h = lambda x: α * f(x) + (1 - α) * g(x)
    return quad(lambda x: h(x) * np.log(h(x) / f(x)), 0, 1)[0]


# vectorize
KL_g_quad_v = np.vectorize(KL_g_quad)
KL_f_quad_v = np.vectorize(KL_f_quad)


@jax.jit
def π_lim(key, α, T=5000, π_0=0.4):
    """Find limit of π sequence."""
    # Get lottery draws
    l_arr = simulate_mixed(key, α, T, N=1)[0]
```

(continues on next page)

(continued from previous page)

```
    def scan_fn(π_prev, l_t):
        π_new = update(π_prev, l_t)
        return π_new, π_new

    # Scan over lottery draws
    π_final, _ = jax.lax.scan(scan_fn, π_0, l_arr)

    return π_final


π_lim_v = jax.vmap(π_lim)
```

Let us first plot the KL divergences $KL_g(\alpha), KL_f(\alpha)$ for each α.

```
α_arr = np.linspace(0, 1, 100)
KL_g_arr = KL_g_v(α_arr)
KL_f_arr = KL_f_v(α_arr)

fig, ax = plt.subplots()

ax.plot(α_arr, KL_g_arr, label="KL(h, g)", lw=2)
ax.plot(α_arr, KL_f_arr, label="KL(h, f)", lw=2)
ax.set_ylabel("KL divergence")
ax.set_xlabel(r"$\alpha$")

ax.legend(loc="upper right")
plt.show()
```

Let's compute an α for which the KL divergence between h and g is the same as that between h and f.

```
# where KL_f = KL_g
discretion = α_arr[np.argmin(np.abs(KL_g_arr - KL_f_arr))]
```

We can compute and plot the convergence point π_∞ for each α to verify that the convergence is indeed governed by the KL divergence.

The blue circles show the limiting values of π_t that simulations discover for different values of α recorded on the x axis.

Thus, the graph below confirms how a minimum KL divergence governs what our type 1 agent eventually learns.

```
α_arr_x = α_arr[(α_arr < discretion) | (α_arr > discretion)]
keys = jax.random.split(jax.random.key(42), len(α_arr_x))
π_lim_arr = π_lim_v(keys, α_arr_x)

# plot
fig, ax = plt.subplots()

ax.plot(α_arr, KL_g_arr, label="KL(h, g)", lw=2)
ax.plot(α_arr, KL_f_arr, label="KL(h, f)", lw=2)
ax.set_ylabel("KL divergence")
ax.set_xlabel(r"$\alpha$")

# plot KL
ax2 = ax.twinx()
# plot limit point
```

(continues on next page)

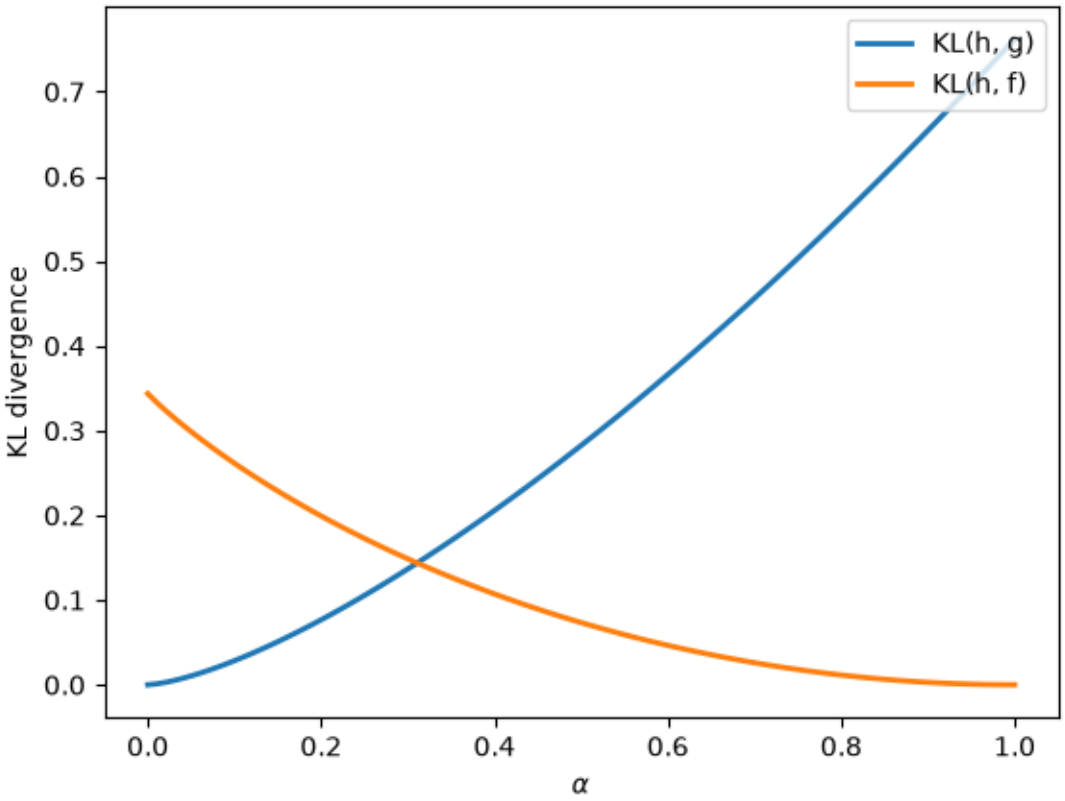


Fig. 32.4: KL divergences against α

(continued from previous page)

```
ax2.scatter(
    α_arr_x,
    π_lim_arr,
    facecolors="none",
    edgecolors="tab:blue",
    label=r"$\pi$ lim",
)
ax2.set_ylabel("π lim")

ax.legend(loc=[0.85, 0.8])
ax2.legend(loc=[0.85, 0.73])
plt.show()
```

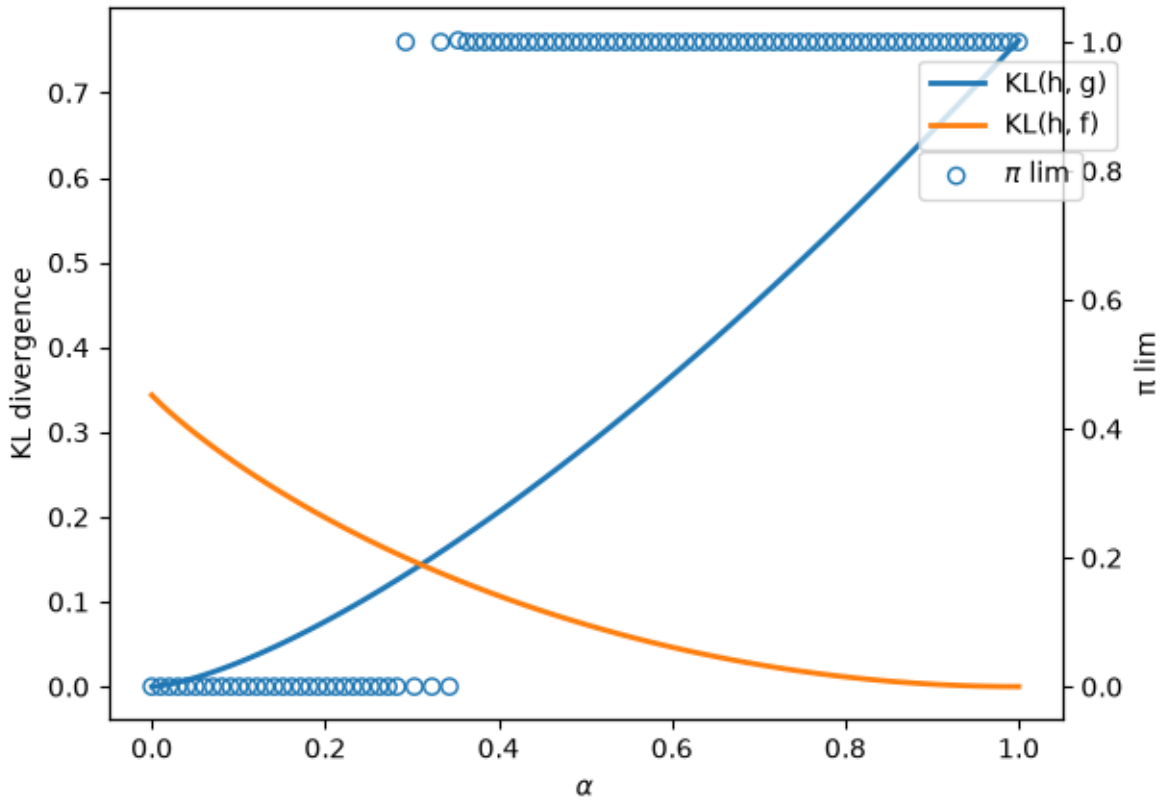


Fig. 32.5: Limit points and KL divergences

Evidently, our type 1 learner who applies Bayes' law to his misspecified set of statistical models eventually learns an approximating model that is as close as possible to the true model, as measured by its Kullback-Leibler divergence:

- When α is small, $KL_g < KL_f$, meaning the divergence of g from h is smaller than that of f and so the limit point of π_t is close to 0.
- When α is large, $KL_f < KL_g$, meaning the divergence of f from h is smaller than that of g and so the limit point of π_t is close to 1.

32.6 Type 2 agent

We now describe how our type 2 agent formulates his learning problem and what he eventually learns.

Our type 2 agent understands the correct statistical model but does not know α.

We apply Bayes' law to deduce an algorithm for learning α under the assumption that the agent knows that

$$h(w) = h(w|\alpha)$$

but does not know α.

We'll assume that the agent starts out with a prior probability $\pi_0(\alpha)$ on $\alpha \in (0,1)$ where the prior has one of the forms that we deployed in *Non-Conjugate Priors*.

We'll fire up `numpyro` and apply it to the present situation.

Bayes' law now takes the form

$$\pi_{t+1}(\alpha) = \frac{h(w_{t+1}|\alpha)\pi_t(\alpha)}{\int h(w_{t+1}|\hat{\alpha})\pi_t(\hat{\alpha})d\hat{\alpha}}$$

We'll use numpyro to approximate this equation.

We'll create graphs of the posterior $\pi_t(\alpha)$ as $t \to +\infty$ corresponding to ones presented in *Non-Conjugate Priors*.

We anticipate that a posterior distribution will collapse around the true α as $t \to +\infty$.

Let us try a uniform prior first.

We use the `Mixture` class in numpyro to construct the likelihood function.

```
a = 0.8

# simulate data with true a
data = draw_lottery(jax.random.key(42), a, 1000)
sizes = [5, 20, 50, 200, 1000, 25000]


def model(w):
    a = numpyro.sample("a", dist.Uniform(low=0.0, high=1.0))

    y_samp = numpyro.sample(
        "w",
        dist.Mixture(
            dist.Categorical(jnp.array([a, 1 - a])),
            [dist.Beta(F_a, F_b), dist.Beta(G_a, G_b)],
        ),
        obs=w,
    )


def MCMC_run(ws):
    "Compute posterior using MCMC with observed ws"

    kernel = NUTS(model)
    mcmc = MCMC(kernel, num_samples=5000, num_warmup=1000, progress_bar=False)

    mcmc.run(rng_key=random.key(42), w=jnp.array(ws))
    sample = mcmc.get_samples()
    return sample["a"]
```

The following code generates the graph below that displays Bayesian posteriors for α at various history lengths.

```
fig, ax = plt.subplots()

for i in range(len(sizes)):
    sample = MCMC_run(data[: sizes[i]])
    sns.histplot(
        data=sample,
        kde=True,
        stat="density",
        alpha=0.2,
        ax=ax,
        color=colors[i],
        binwidth=0.02,
        linewidth=0.05,
        label=f"t={sizes[i]}",
    )
ax.legend()
ax.set_xlabel(r"$\alpha$")
plt.show()
```

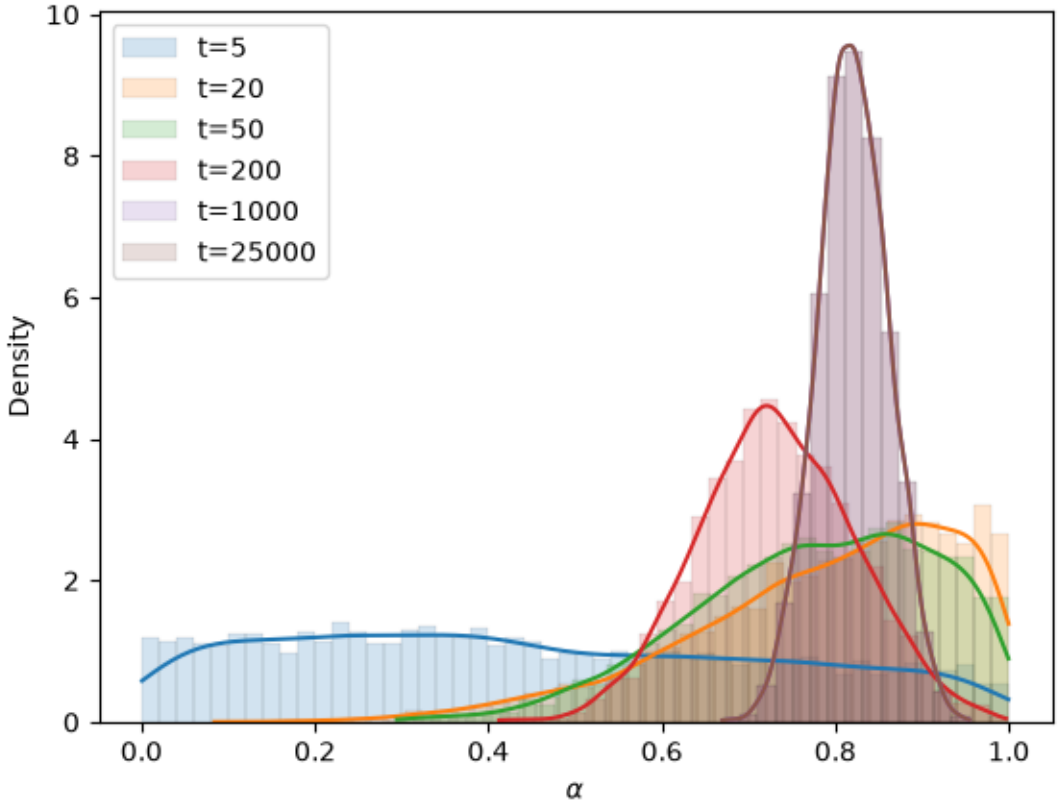


Fig. 32.6: Posterior for α as t grows

Evidently, the Bayesian posterior narrows in on the true value $\alpha = 0.8$ of the mixing parameter as the length of a history of observations grows.

32.7 Concluding remarks

Our type 1 agent deploys an incorrect statistical model.

He believes that either f or g generated the w process, but just doesn't know which one.

That is wrong because nature is actually mixing each period with mixing probability α.

Our type 1 agent eventually believes that either f or g generated the w sequence, the outcome being determined by the model, either f or g, whose KL divergence relative to h is smaller.

Our type 2 agent has a different statistical model, one that is correctly specified.

He knows the parametric form of the statistical model but not the mixing parameter α.

He knows that he does not know it.

But by using Bayes' law in conjunction with his statistical model and a history of data, he eventually acquires a more and more accurate inference about α.

This little laboratory exhibits some important general principles that govern outcomes of Bayesian learning of misspecified models.

Thus, the following situation prevails quite generally in empirical work.

A scientist approaches the data with a manifold S of statistical models $s(X|\theta)$, where s is a probability distribution over a random vector X, $\theta \in \Theta$ is a vector of parameters, and Θ indexes the manifold of models.

The scientist with observations that he interprets as realizations x of the random vector X wants to solve an **inverse problem** of somehow *inverting* $s(x|\theta)$ to infer θ from x.

But the scientist's model is misspecified, being only an approximation to an unknown model h that nature uses to generate X.

If the scientist uses Bayes' law or a related likelihood-based method to infer θ, it occurs quite generally that for large sample sizes the inverse problem infers a θ that minimizes the KL divergence of the scientist's model s relative to nature's model h.

32.8 Exercises

Exercise 32.8.1

In *Likelihood Ratio Processes and Bayesian Learning*, we studied the consequence of applying likelihood ratio and Bayes' law to a misspecified statistical model.

In that lecture, we used a model selection algorithm to study the case where the true data generating process is a mixture.

In this lecture, we studied how to correctly "learn" a model generated by a mixing process using a Bayesian approach.

To fix the algorithm we used in *Likelihood Ratio Processes and Bayesian Learning*, a correct Bayesian approach should directly model the uncertainty about x and update beliefs about it as new data arrives.

Here is the algorithm:

First we specify a prior distribution for x given by $x \sim \text{Beta}(\alpha_0, \beta_0)$ with expectation $\mathbb{E}[x] = \frac{\alpha_0}{\alpha_0+\beta_0}$.

The likelihood for a single observation w_t is $p(w_t|x) = xf(w_t) + (1-x)g(w_t)$.

For a sequence $w^t = (w_1, \ldots, w_t)$, the likelihood is $p(w^t|x) = \prod_{i=1}^{t} p(w_i|x)$.

The posterior distribution is updated using $p(x|w^t) \propto p(w^t|x)p(x)$.

Recursively, the posterior after w_t is $p(x|w^t) \propto p(w_t|x)p(x|w^{t-1})$.

Without a conjugate prior, we can approximate the posterior by discretizing x into a grid.

Your task is to implement this algorithm in Python.

You can verify your implementation by checking that the posterior mean converges to the true value of x as t increases in *Likelihood Ratio Processes and Bayesian Learning*.

Solution

Here is one solution:

First we define the mixture probability and parameters of prior distributions

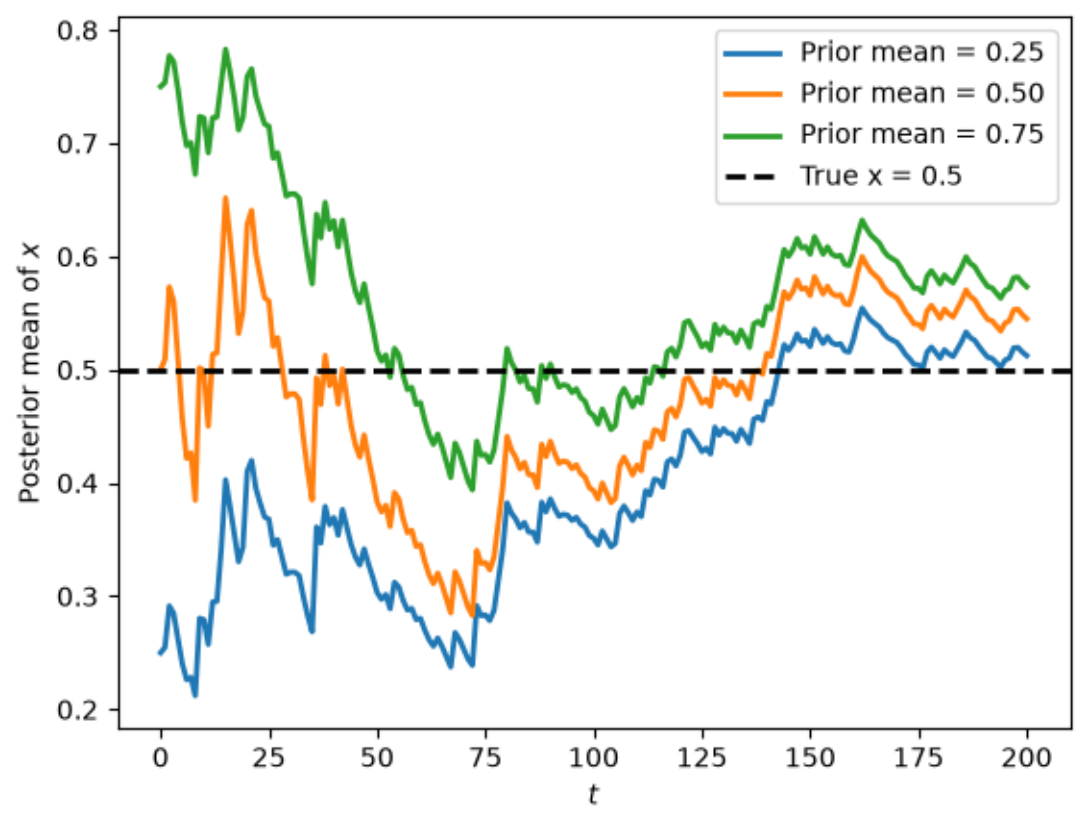
```
x_true = 0.5
T_mix = 200

# Three different priors with means 0.25, 0.5, 0.75
prior_params = [(1, 3), (1, 1), (3, 1)]
prior_means = [a / (a + b) for a, b in prior_params]

w_mix = draw_lottery(jax.random.key(42), x_true, T_mix)
```

```
@jax.jit
def learn_x_bayesian(observations, α0, β0, grid_size=2000):
    """
    Sequential Bayesian learning of the mixing probability x
    using a grid approximation.
    """
    w = jnp.asarray(observations)
    T = w.size

    x_grid = jnp.linspace(1e-3, 1 - 1e-3, grid_size)

    # Log prior
    log_prior = (α0 - 1) * jnp.log(x_grid) + (β0 - 1) * jnp.log1p(-x_grid)

    def scan_fn(log_post, wt):
        # P(w_t | x) = x f(w_t) + (1 - x) g(w_t)
        like = x_grid * f(wt) + (1 - x_grid) * g(wt)
        log_post = log_post + jnp.log(like)

        # Normalize using log-sum-exp trick
        log_post = log_post - jax.nn.logsumexp(log_post)
        post = jnp.exp(log_post)

        # Compute posterior mean
        μ = x_grid @ post

        return log_post, μ

    # Initial posterior mean
    μ_0 = α0 / (α0 + β0)

    # Scan over observations
```

```
    _, μ_path = jax.lax.scan(scan_fn, log_prior, w)

    # Prepend initial value
    return jnp.concatenate([jnp.array([μ_0]), μ_path])


# Vectorize over different prior parameters
def compute_all_posteriors(observations, prior_params):
    """Compute posterior means for all prior parameter pairs."""

    def single_posterior(params):
        α0, β0 = params
        return learn_x_bayesian(observations, α0, β0)

    return jax.vmap(single_posterior)(jnp.array(prior_params))


x_posterior_means = compute_all_posteriors(w_mix, jnp.array(prior_params))
```

Let's visualize how the posterior mean of x evolves over time, starting from three different prior beliefs.

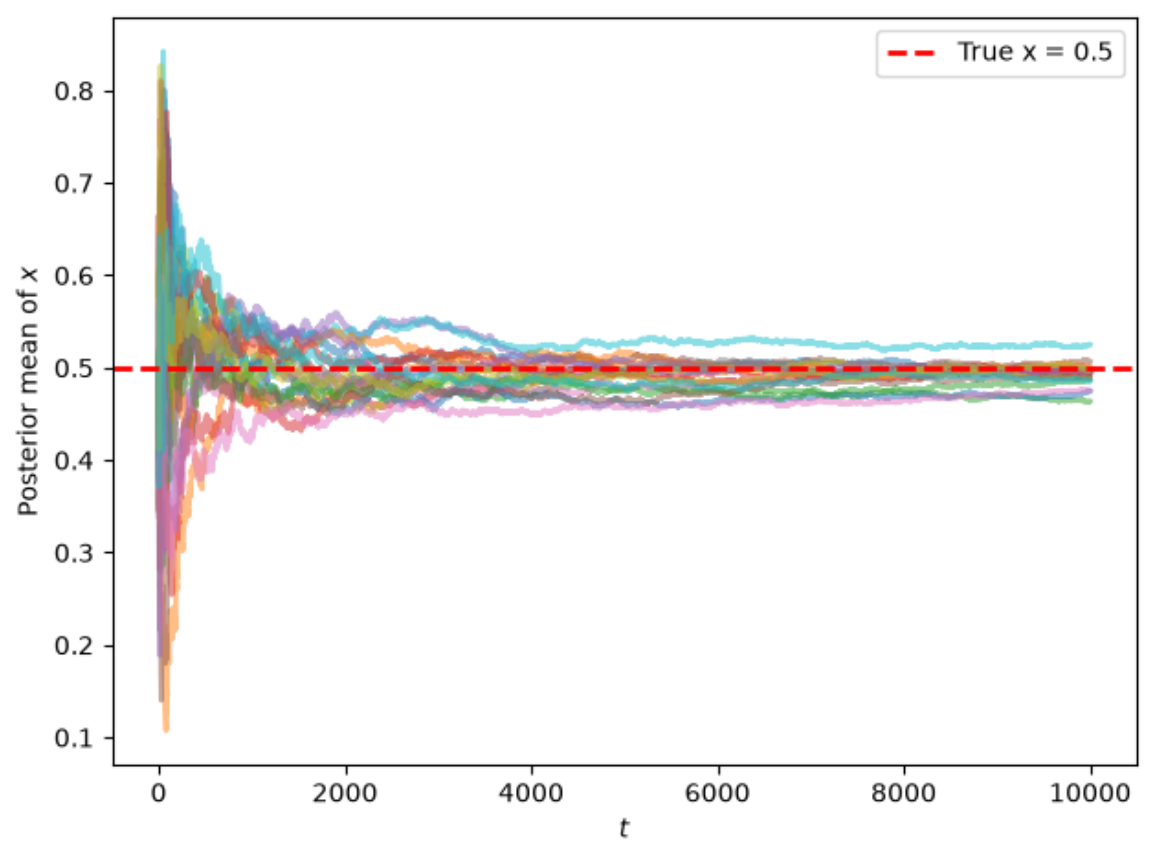
```
fig, ax = plt.subplots()

for i, (x_means, mean0) in enumerate(zip(x_posterior_means, prior_means)):
    ax.plot(
        range(T_mix + 1),
        x_means,
        label=rf"Prior mean = ${mean0:.2f}$",
        color=colors[i],
        linewidth=2,
    )

ax.axhline(
    y=x_true,
    color="black",
    linestyle="--",
    label=f"True x = {x_true}",
    linewidth=2,
)
ax.set_xlabel("$t$")
ax.set_ylabel("Posterior mean of $x$")
ax.legend()
plt.show()
```



The plot shows that regardless of the initial prior belief, all three posterior means eventually converge towards the true value of $x = 0.5$.

Next, let's look at multiple simulations with a longer time horizon, all starting from a uniform prior.

```
n_paths = 20
T_long = 10_000

keys = jax.random.split(jax.random.key(42), n_paths)

fig, ax = plt.subplots()

for j in range(n_paths):
    w_path = draw_lottery(keys[j], x_true, T_long)
    x_means = learn_x_bayesian(w_path, 1, 1)  # Uniform prior
    ax.plot(range(T_long + 1), x_means, alpha=0.5, lw=2)

ax.axhline(
    y=x_true,
    color="red",
    linestyle="--",
    label=f"True x = {x_true}",
    linewidth=2,
)
ax.set_ylabel("Posterior mean of $x$")
ax.set_xlabel("$t$")
ax.legend()
plt.tight_layout()
plt.show()
```



We can see that the posterior mean of x converges to the true value $x = 0.5$.

CHAPTER

THIRTYTHREE

BAYESIAN VERSUS FREQUENTIST DECISION RULES

Contents

- *Bayesian versus Frequentist Decision Rules*
  - *Overview*
  - *Setup*
  - *Frequentist Decision Rule*
  - *Bayesian Decision Rule*
  - *Was the Navy Captain's Hunch Correct?*
  - *More Details*
  - *Distribution of Bayesian Decision Rule's Time to Decide*
  - *Probability of Making Correct Decision*
  - *Distribution of Likelihood Ratios at Neyman-Pearson's t*


```
import matplotlib.pyplot as plt
import numpy as np
from numba import jit, prange, float64, int64
from numba.experimental import jitclass
from math import gamma
from scipy.optimize import minimize
```

33.1 Overview

This lecture follows up on ideas presented in the following lectures:

- *A Problem that Stumped Milton Friedman*
- *A Bayesian Formulation of Friedman and Wald's Problem*
- *Exchangeability and Bayesian Updating*
- *Likelihood Ratio Processes*

A Problem that Stumped Milton Friedman described a problem that a Navy Captain presented to Milton Friedman during World War II.

The Navy had told the Captain to use a decision rule for quality control.

In particular, the Navy had ordered the Captain to use an instance of a **frequentist decision rule**.

The Captain doubted that that rule was a good one.

Milton Friedman recognized the Captain's conjecture as posing a challenging statistical problem that he and other members of the US Government's Statistical Research Group at Columbia University proceeded to try to solve.

A member of the group, the great mathematician and economist Abraham Wald, soon solved the problem.

A good way to formulate the problem is to use some ideas from Bayesian statistics that we describe in this lecture *Exchangeability and Bayesian Updating* and in this lecture *Likelihood Ratio Processes*, which describes the link between Bayesian updating and likelihood ratio processes.

The present lecture uses Python to generate simulations that evaluate expected losses under the Neyman-Pearson **frequentist** procedure that the Navy captain questioned and the **Bayesian** decision rule described in *A Bayesian Formulation of Friedman and Wald's Problem*.

The simulations confirm the Navy Captain's hunch that there is a better rule than the Neyman-Pearson likelihood ratio test that the Navy had told him to use.

33.2 Setup

To formalize the problem that had confronted the Navy Captain, we consider a setting with the following parts.

- Each period a decision maker draws a non-negative random variable Z. He knows that two probability distributions are possible, f_0 and f_1, and that which ever distribution it is remains fixed over time. The decision maker believes that before the beginning of time, nature once and for all had selected either f_0 or f_1 and that the probability that it selected f_0 is probability π^*.
- The decision maker observes a sample $\{z_i\}_{i=0}^t$ from the distribution chosen by nature.

The decision maker wants to decide which distribution actually governs Z.

He is worried about two types of errors and the losses that they will impose on him.

- a loss $\bar{L}_1$ from a **type I error** that occurs if he decides that $f = f_1$ when actually $f = f_0$
- a loss $\bar{L}_0$ from a **type II error** that occurs if he decides that $f = f_0$ when actually $f = f_1$

The decision maker pays a cost c for drawing another z.

We mainly borrow parameters from the quantecon lecture *A Bayesian Formulation of Friedman and Wald's Problem* except that we increase both $\bar{L}_0$ and $\bar{L}_1$ from 25 to 100 to encourage the Bayesian decision rule to take more draws before deciding.

We set the cost c of taking one more draw at 1.25.

We set the probability distributions f_0 and f_1 to be beta distributions with $a_0 = b_0 = 1$, $a_1 = 3$, and $b_1 = 1.2$, respectively.

Below is some Python code that sets up these objects.

```
@jit
def p(x, a, b):
    "Beta distribution."

    r = gamma(a + b) / (gamma(a) * gamma(b))

    return r * x**(a-1) * (1 - x)**(b-1)
```

We start with defining a `jitclass` that stores parameters and functions we need to solve problems for both the Bayesian and frequentist Navy Captains.

```
wf_data = [
    ('c', float64),           # unemployment compensation
    ('a0', float64),          # parameters of beta distribution
    ('b0', float64),
    ('a1', float64),
    ('b1', float64),
    ('L0', float64),          # cost of selecting f0 when f1 is true
    ('L1', float64),          # cost of selecting f1 when f0 is true
    ('π_grid', float64[:]),   # grid of beliefs π
    ('π_grid_size', int64),
    ('mc_size', int64),       # size of Monto Carlo simulation
    ('z0', float64[:]),       # sequence of random values
    ('z1', float64[:])        # sequence of random values
]
```

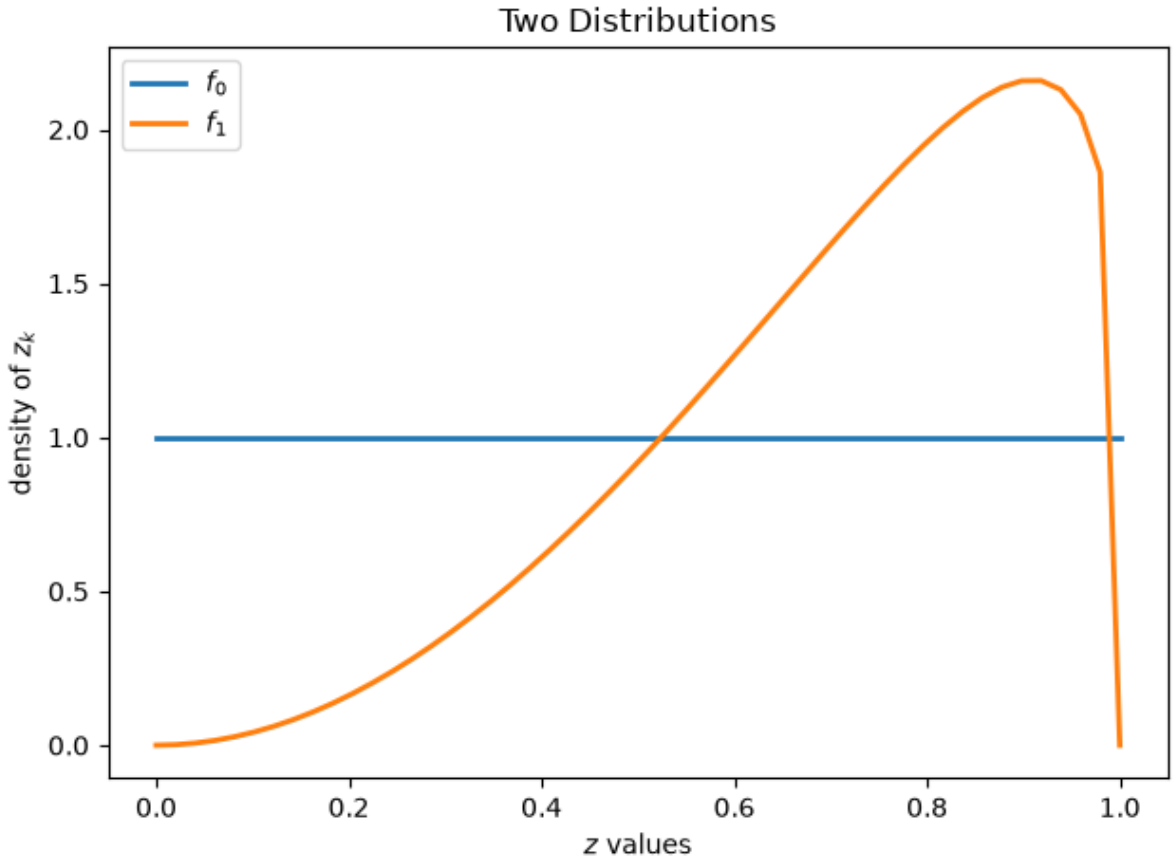
```
@jitclass(wf_data)
class WaldFriedman:

    def __init__(self,
                 c=1.25,
                 a0=1,
                 b0=1,
                 a1=3,
                 b1=1.2,
                 L0=100,
                 L1=100,
                 π_grid_size=200,
                 mc_size=1000):

        self.c, self.π_grid_size = c, π_grid_size
        self.a0, self.b0, self.a1, self.b1 = a0, b0, a1, b1
        self.L0, self.L1 = L0, L1
        self.π_grid = np.linspace(0, 1, π_grid_size)
        self.mc_size = mc_size

        self.z0 = np.random.beta(a0, b0, mc_size)
        self.z1 = np.random.beta(a1, b1, mc_size)

    def f0(self, x):

        return p(x, self.a0, self.b0)

    def f1(self, x):

        return p(x, self.a1, self.b1)

    def κ(self, z, π):
        """
        Updates π using Bayes' rule and the current observation z
        """

        a0, b0, a1, b1 = self.a0, self.b0, self.a1, self.b1

        π_f0, π_f1 = π * p(z, a0, b0), (1 - π) * p(z, a1, b1)
        π_new = π_f0 / (π_f0 + π_f1)
```

(continues on next page)

(continued from previous page)

```
        return n_new
```

```
wf = WaldFriedman()

grid = np.linspace(0, 1, 50)

plt.figure()

plt.title("Two Distributions")
plt.plot(grid, wf.f0(grid), lw=2, label="$f_0$")
plt.plot(grid, wf.f1(grid), lw=2, label="$f_1$")

plt.legend()
plt.xlabel("$z$ values")
plt.ylabel("density of $z_k$")

plt.tight_layout()
plt.show()
```



Above, we plot the two possible probability densities f_0 and f_1

33.3 Frequentist Decision Rule

The Navy told the Captain to use a Neyman-Pearson likelihood ratio decision rule.

That decision rule is characterized by

- a sample size t, and
- a cutoff value d of a likelihood ratio

Let $L\left(z^t\right) = \prod_{i=0}^{t} \frac{f_0(z_i)}{f_1(z_i)}$ be the likelihood ratio associated with observing the sequence $\{z_i\}_{i=0}^{t}$.

The decision rule associated with a sample size t is:

- decide that f_0 is the distribution if the likelihood ratio is greater than d
- decide that f_1 is the distribution if the likelihood ratio is less than d

For our purposes here, we want to compute an expected loss from using this rule, where we borrow loss parameters $\bar{L}_1$ and $\bar{L}_2$ from *A Bayesian Formulation of Friedman and Wald's Problem*.

Let null and alternative hypotheses be

- null: H_0: $f = f_0$,
- alternative H_1: $f = f_1$.

Given sample size t and cutoff d, under the model described above, the mathematical expectation of total loss is

$$\bar{V}_{fre}\left(t, d\right) = ct + \pi^* PFA \times \bar{L}_1 + \left(1 - \pi^*\right)\left(1 - PD\right) \times \bar{L}_0 \tag{33.1}$$

$$\begin{aligned} \text{where} \quad PFA &= \Pr\left\{L\left(z^t\right) < d \mid q = f_0\right\} \\ PD &= \Pr\left\{L\left(z^t\right) < d \mid q = f_1\right\} \end{aligned}$$

Here

- PFA denotes the probability of a **false alarm**, i.e., rejecting H_0 when it is true
- PD denotes the probability of a **detection error**, i.e., not rejecting H_0 when H_1 is true

For a given sample size t, the pairs (PFA, PD) lie on a **receiver operating characteristic curve**.

- by choosing d, we select a particular pair (PFA, PD) along the curve for a given t

To see some receiver operating characteristic curves, please see this lecture *Likelihood Ratio Processes*.

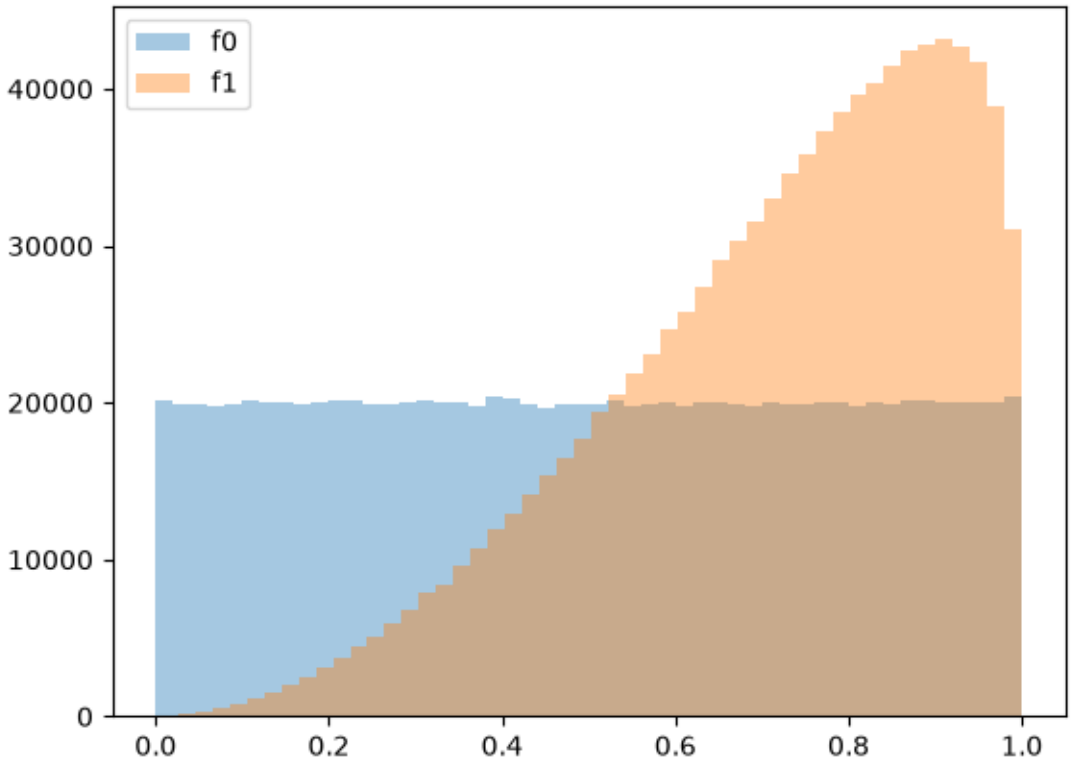
To solve for $\bar{V}_{fre}\left(t, d\right)$ numerically, we first simulate sequences of z when either f_0 or f_1 generates data.

Let's plot empirical distributions, i.e., histograms, associated with f_0 and f_1.

```
N = 10000
T = 100
```

```
z0_arr = np.random.beta(wf.a0, wf.b0, (N, T))
z1_arr = np.random.beta(wf.a1, wf.b1, (N, T))
```

```
plt.hist(z0_arr.flatten(), bins=50, alpha=0.4, label='f0')
plt.hist(z1_arr.flatten(), bins=50, alpha=0.4, label='f1')
plt.legend()
plt.show()
```



We can compute sequences of likelihood ratios using simulated samples.

```
l = lambda z: wf.f0(z) / wf.f1(z)
```

```
l0_arr = l(z0_arr)
l1_arr = l(z1_arr)

L0_arr = np.cumprod(l0_arr, 1)
L1_arr = np.cumprod(l1_arr, 1)
```

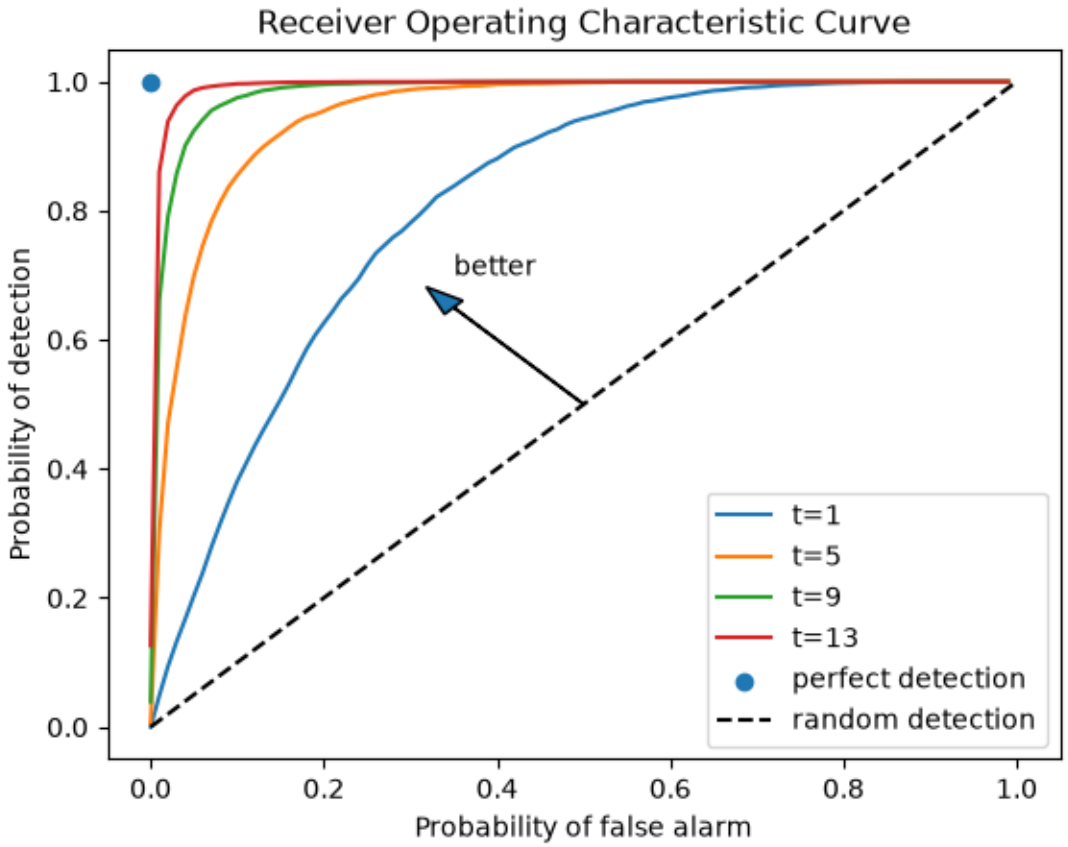
With an empirical distribution of likelihood ratios in hand, we can draw **receiver operating characteristic curves** by enumerating (PFA, PD) pairs given each sample size t.

```
PFA = np.arange(0, 100, 1)

for t in range(1, 15, 4):
    percentile = np.percentile(L0_arr[:, t], PFA)
    PD = [np.sum(L1_arr[:, t] < p) / N for p in percentile]

    plt.plot(PFA / 100, PD, label=f"t={t}")

plt.scatter(0, 1, label="perfect detection")
plt.plot([0, 1], [0, 1], color='k', ls='--', label="random detection")

plt.arrow(0.5, 0.5, -0.15, 0.15, head_width=0.03)
plt.text(0.35, 0.7, "better")
plt.xlabel("Probability of false alarm")
plt.ylabel("Probability of detection")
plt.legend()
plt.title("Receiver Operating Characteristic Curve")
```

(continues on next page)

(continued from previous page)

```
plt.show()
```



We can minimize the expected total loss presented in equation (33.1) by choosing (t, d).

Doing that delivers an expected loss

$$\bar{V}_{fre} = \min_{t,d} \bar{V}_{fre}(t, d).$$

We first consider the case in which $\pi^* = \Pr\{\text{nature selects } f_0\} = 0.5$.

We can solve the minimization problem in two steps.

First, we fix t and find the optimal cutoff d and consequently the minimal $\bar{V}_{fre}(t)$.

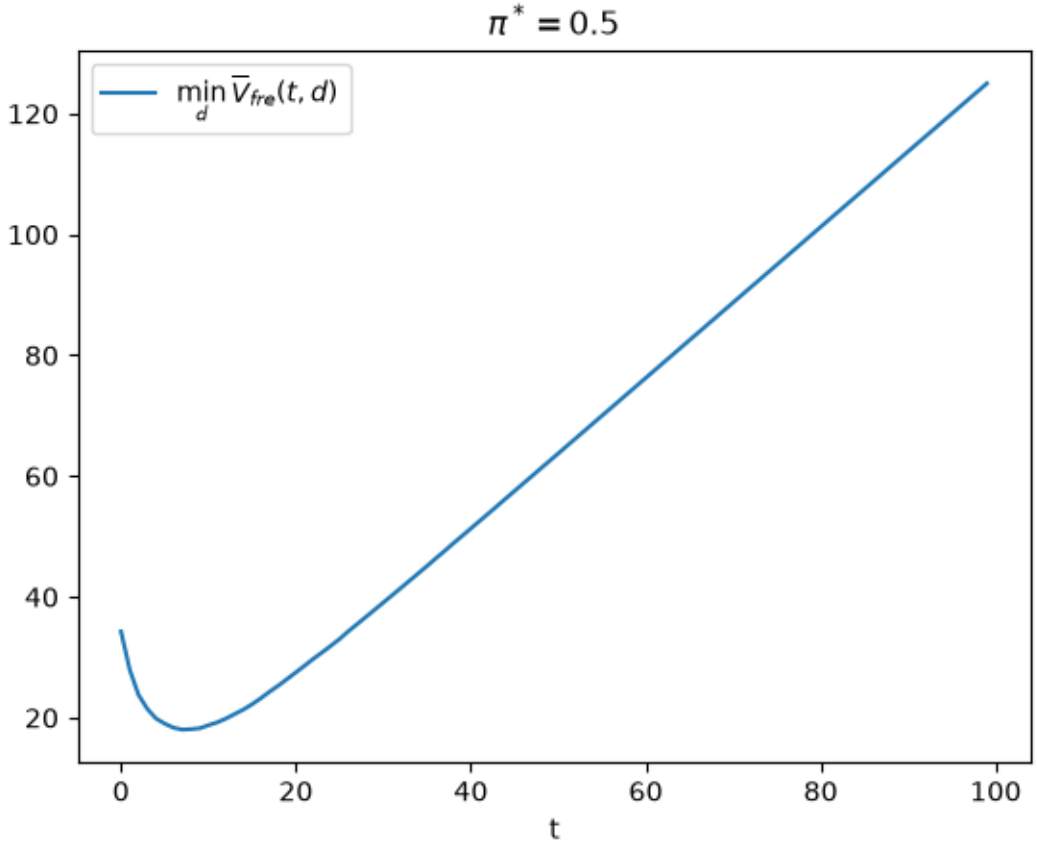
Here is Python code that does that and then plots a useful graph.

```
@jit
def V_fre_d_t(d, t, L0_arr, L1_arr, π_star, wf):

    N = L0_arr.shape[0]

    PFA = np.sum(L0_arr[:, t-1] < d) / N
    PD = np.sum(L1_arr[:, t-1] < d) / N

    V = π_star * PFA * wf.L1 + (1 - π_star) * (1 - PD) * wf.L0

    return V
```

```
def V_fre_t(t, L0_arr, L1_arr, π_star, wf):

    res = minimize(V_fre_d_t, 1, args=(t, L0_arr, L1_arr, π_star, wf), method='Nelder-
↪Mead')
    V = res.fun
    d = res.x

    PFA = np.sum(L0_arr[:, t-1] < d) / N
    PD = np.sum(L1_arr[:, t-1] < d) / N

    return V, PFA, PD
```

```
def compute_V_fre(L0_arr, L1_arr, π_star, wf):

    T = L0_arr.shape[1]

    V_fre_arr = np.empty(T)
    PFA_arr = np.empty(T)
    PD_arr = np.empty(T)

    for t in range(1, T+1):
        V, PFA, PD = V_fre_t(t, L0_arr, L1_arr, π_star, wf)
        V_fre_arr[t-1] = wf.c * t + V
        PFA_arr[t-1] = PFA
        PD_arr[t-1] = PD

    return V_fre_arr, PFA_arr, PD_arr
```

```
π_star = 0.5
V_fre_arr, PFA_arr, PD_arr = compute_V_fre(L0_arr, L1_arr, π_star, wf)

plt.plot(range(T), V_fre_arr, label=r'$\min_{d} \overline{V}_{fre}(t,d)$')
plt.xlabel('t')
plt.title(r'$\pi^*=0.5$')
plt.legend()
plt.show()
```




```
t_optimal = np.argmin(V_fre_arr) + 1
```

The above graph illustrates how minimizing over t tells the frequentist to draw t_{optimal} observations and then decide.

Let's now change the value of π^* and watch how the decision rule changes.

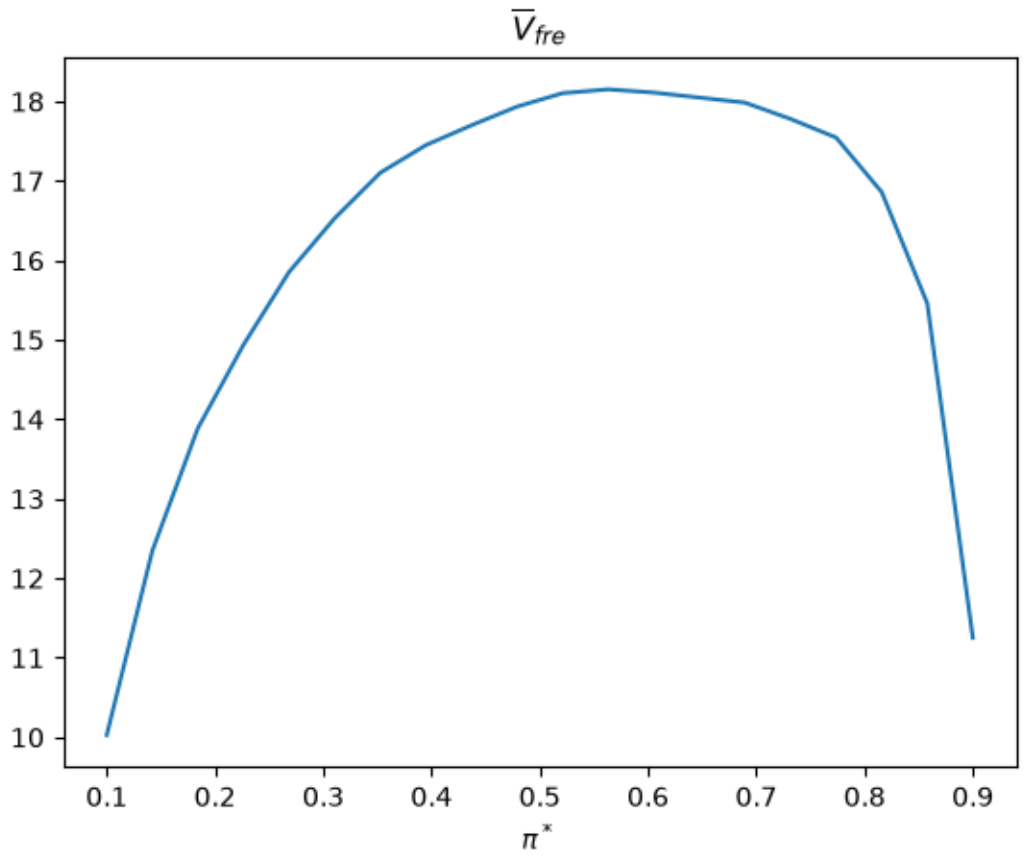
```
n_π = 20
π_star_arr = np.linspace(0.1, 0.9, n_π)

V_fre_bar_arr = np.empty(n_π)
t_optimal_arr = np.empty(n_π)
PFA_optimal_arr = np.empty(n_π)
PD_optimal_arr = np.empty(n_π)

for i, π_star in enumerate(π_star_arr):
    V_fre_arr, PFA_arr, PD_arr = compute_V_fre(L0_arr, L1_arr, π_star, wf)
    t_idx = np.argmin(V_fre_arr)

    V_fre_bar_arr[i] = V_fre_arr[t_idx]
    t_optimal_arr[i] = t_idx + 1
    PFA_optimal_arr[i] = PFA_arr[t_idx]
    PD_optimal_arr[i] = PD_arr[t_idx]
```

```
plt.plot(π_star_arr, V_fre_bar_arr)
plt.xlabel(r'$\pi^*$')
plt.title(r'$\overline{V}_{fre}$')
```

(continues on next page)

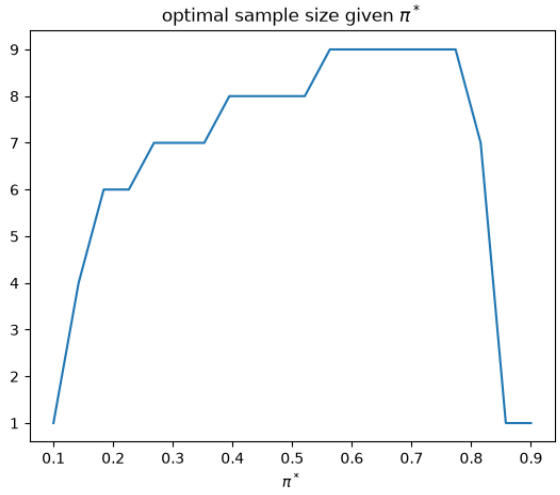
(continued from previous page)

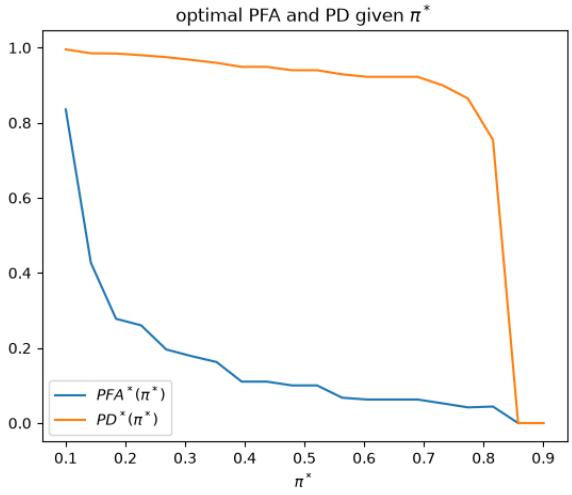
```
plt.show()
```



The following shows how optimal sample size t and targeted (PFA, PD) change as π^* varies.

```
fig, axs = plt.subplots(1, 2, figsize=(14, 5))

axs[0].plot(π_star_arr, t_optimal_arr)
axs[0].set_xlabel(r'$\pi^*$')
axs[0].set_title(r'optimal sample size given $\pi^*$')

axs[1].plot(π_star_arr, PFA_optimal_arr, label=r'$PFA^*(\pi^*)$')
axs[1].plot(π_star_arr, PD_optimal_arr, label=r'$PD^*(\pi^*)$')
axs[1].set_xlabel(r'$\pi^*$')
axs[1].legend()
axs[1].set_title(r'optimal PFA and PD given $\pi^*$')

plt.show()
```





33.4 Bayesian Decision Rule

In *A Problem that Stumped Milton Friedman*, we learned how Abraham Wald confirmed the Navy Captain's hunch that there is a better decision rule.

In *A Bayesian Formulation of Friedman and Wald's Problem* we presented a Bayesian procedure that makes decisions by comparing a Bayesian posterior probability π with cutoff probabilities called A and B.

To proceed, we borrow some Python code from the quantecon lecture *A Bayesian Formulation of Friedman and Wald's Problem* that computes optimal values of A and B.

```
@jit(parallel=True)
def Q(h, wf):

    c, π_grid = wf.c, wf.π_grid
    L0, L1 = wf.L0, wf.L1
    z0, z1 = wf.z0, wf.z1
    mc_size = wf.mc_size

    κ = wf.κ

    h_new = np.empty_like(π_grid)
    h_func = lambda p: np.interp(p, π_grid, h)

    for i in prange(len(π_grid)):
        π = π_grid[i]

        # Find the expected value of J by integrating over z
        integral_f0, integral_f1 = 0, 0
        for m in range(mc_size):
            π_0 = κ(z0[m], π)  # Draw z from f0 and update π
            integral_f0 += min((1 - π_0) * L0, π_0 * L1, h_func(π_0))

            π_1 = κ(z1[m], π)  # Draw z from f1 and update π
            integral_f1 += min((1 - π_1) * L0, π_1 * L1, h_func(π_1))

        integral = (π * integral_f0 + (1 - π) * integral_f1) / mc_size
```

(continues on next page)

(continued from previous page)

```
        h_new[i] = c + integral

    return h_new
```

```
@jit
def solve_model(wf, tol=1e-4, max_iter=1000):
    """
    Compute the continuation value function

    * wf is an instance of WaldFriedman
    """

    # Set up loop
    h = np.zeros(len(wf.π_grid))
    i = 0
    error = tol + 1

    while i < max_iter and error > tol:
        h_new = Q(h, wf)
        error = np.max(np.abs(h - h_new))
        i += 1
        h = h_new

    if error > tol:
        print("Failed to converge!")

    return h_new
```

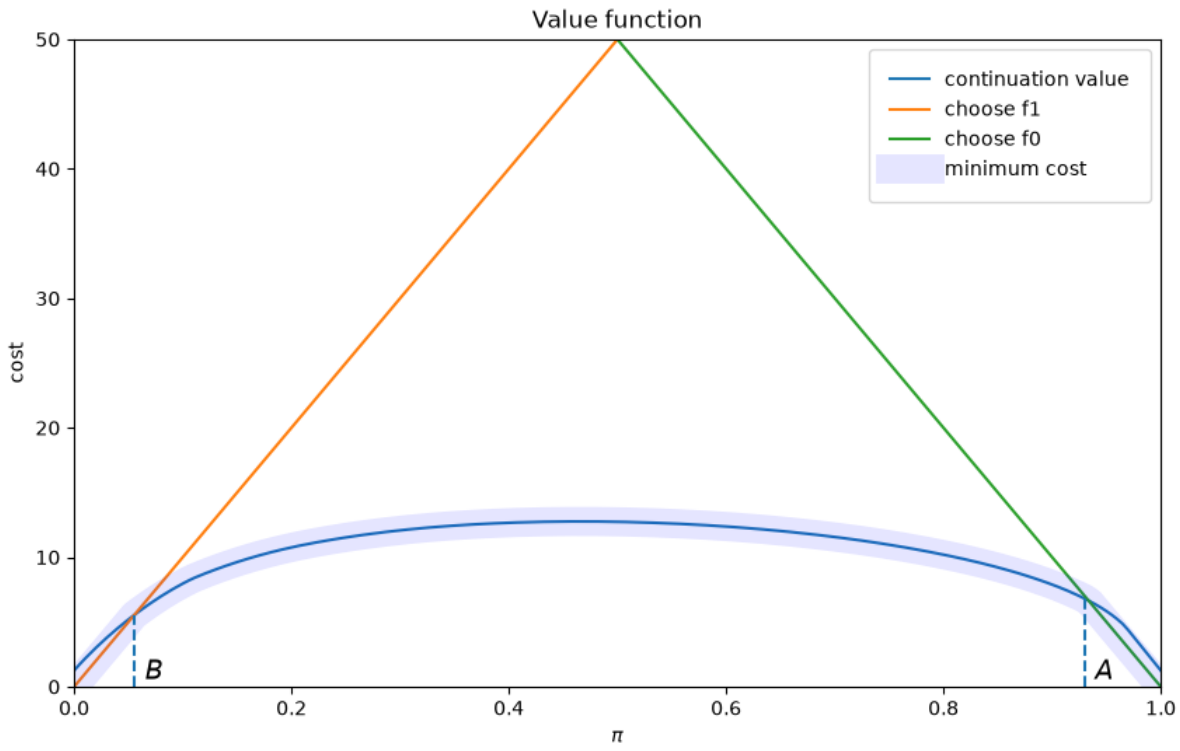
```
h_star = solve_model(wf)
```

```
@jit
def find_cutoff_rule(wf, h):

    """
    This function takes a continuation value function and returns the
    corresponding cutoffs of where you transition between continuing and
    choosing a specific model
    """

    π_grid = wf.π_grid
    L0, L1 = wf.L0, wf.L1

    # Evaluate cost at all points on grid for choosing a model
    payoff_f0 = (1 - π_grid) * L0
    payoff_f1 = π_grid * L1

    # The cutoff points can be found by differencing these costs with
    # The Bellman equation (J is always less than or equal to p_c_i)
    B = π_grid[np.searchsorted(
                               payoff_f1 - np.minimum(h, payoff_f0),
                               1e-10)
               - 1]
    A = π_grid[np.searchsorted(
                               np.minimum(h, payoff_f1) - payoff_f0,
                               1e-10)
               - 1]
```

(continues on next page)

(continued from previous page)

```
    return (B, A)

B, A = find_cutoff_rule(wf, h_star)
cost_L0 = (1 - wf.π_grid) * wf.L0
cost_L1 = wf.π_grid * wf.L1

fig, ax = plt.subplots(figsize=(10, 6))

ax.plot(wf.π_grid, h_star, label='continuation value')
ax.plot(wf.π_grid, cost_L1, label='choose f1')
ax.plot(wf.π_grid, cost_L0, label='choose f0')
ax.plot(wf.π_grid,
        np.amin(np.column_stack([h_star, cost_L0, cost_L1]),axis=1),
        lw=15, alpha=0.1, color='b', label='minimum cost')

ax.annotate(r"$B$", xy=(B + 0.01, 0.5), fontsize=14)
ax.annotate(r"$A$", xy=(A + 0.01, 0.5), fontsize=14)

plt.vlines(B, 0, B * wf.L0, linestyle="--")
plt.vlines(A, 0, (1 - A) * wf.L1, linestyle="--")

ax.set(xlim=(0, 1), ylim=(0, 0.5 * max(wf.L0, wf.L1)), ylabel="cost",
       xlabel=r"$\pi$", title="Value function")

plt.legend(borderpad=1.1)
plt.show()
```



The above figure portrays the value function plotted against the decision maker's Bayesian posterior.

It also shows the cutoff probabilities A and B.

The Bayesian decision rule is:

- accept H_0 if $\pi \geq A$
- accept H_1 if $\pi \leq B$
- delay deciding and draw another z if $B \leq \pi \leq A$

We can calculate two "objective" loss functions under this situation conditioning on knowing for sure that nature has selected f_0, in the first case, or f_1, in the second case.

1. under f_0,

$$V^0(\pi) = \begin{cases} 0 & \text{if} A \leq \pi, \\ c + EV^0(\pi') & \text{if } B \leq \pi < A, \\ \bar{L}_1 & \text{if } \pi < B. \end{cases}$$

2. under f_1

$$V^1(\pi) = \begin{cases} \bar{L}_0 & \text{if } A \leq \pi, \\ c + EV^1(\pi') & \text{if } B \leq \pi < A, \\ 0 & \text{if } \pi < B. \end{cases}$$

where $\pi' = \frac{\pi f_0(z')}{\pi f_0(z') + (1-\pi) f_1(z')}$.

Given a prior probability π_0, the expected loss for the Bayesian is

$$\bar{V}_{Bayes}(\pi_0) = \pi^* V^0(\pi_0) + (1 - \pi^*) V^1(\pi_0).$$

Below we write some Python code that computes $V^0(\pi)$ and $V^1(\pi)$ numerically.

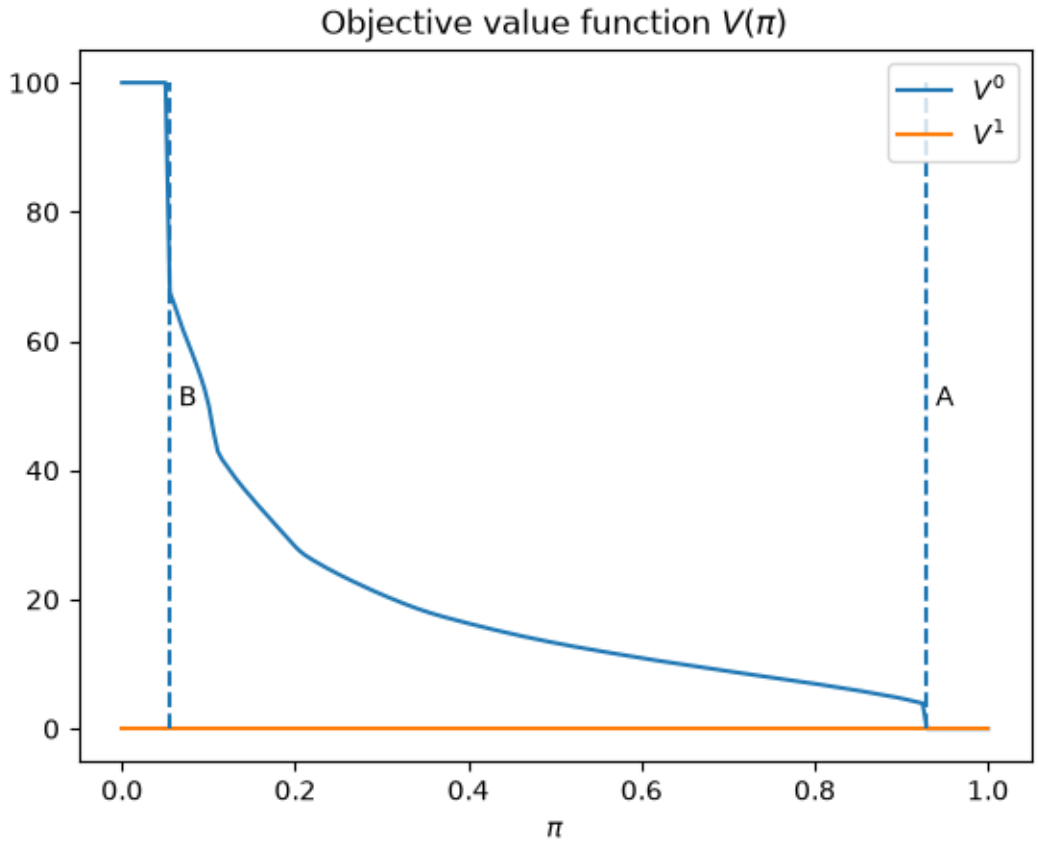
```
@jit(parallel=True)
def V_q(wf, flag):
    V = np.zeros(wf.π_grid_size)
    if flag == 0:
        z_arr = wf.z0
        V[wf.π_grid < B] = wf.L1
    else:
        z_arr = wf.z1
        V[wf.π_grid >= A] = wf.L0

    V_old = np.empty_like(V)

    while True:
        V_old[:] = V[:]
        V[(B <= wf.π_grid) & (wf.π_grid < A)] = 0

        for i in prange(len(wf.π_grid)):
            π = wf.π_grid[i]

            if π >= A or π < B:
                continue

            for j in prange(len(z_arr)):
                π_next = wf.κ(z_arr[j], π)
                V[i] += wf.c + np.interp(π_next, wf.π_grid, V_old)
```

(continues on next page)

(continued from previous page)

```
            V[i] /= wf.mc_size

        if np.abs(V - V_old).max() < 1e-5:
            break

    return V
```

```
V0 = V_q(wf, 0)
V1 = V_q(wf, 1)

plt.plot(wf.π_grid, V0, label='$V^0$')
plt.plot(wf.π_grid, V1, label='$V^1$')
plt.vlines(B, 0, wf.L0, linestyle='--')
plt.text(B+0.01, wf.L0/2, 'B')
plt.vlines(A, 0, wf.L0, linestyle='--')
plt.text(A+0.01, wf.L0/2, 'A')
plt.xlabel(r'$\pi$')
plt.title(r'Objective value function $V(\pi)$')
plt.legend()
plt.show()
```



Given an assumed value for $\pi^* = \Pr\{\text{nature selects } f_0\}$, we can then compute $\bar{V}_{Bayes}(\pi_0)$.

We can then determine an initial Bayesian prior π_0^* that minimizes this objective concept of expected loss.

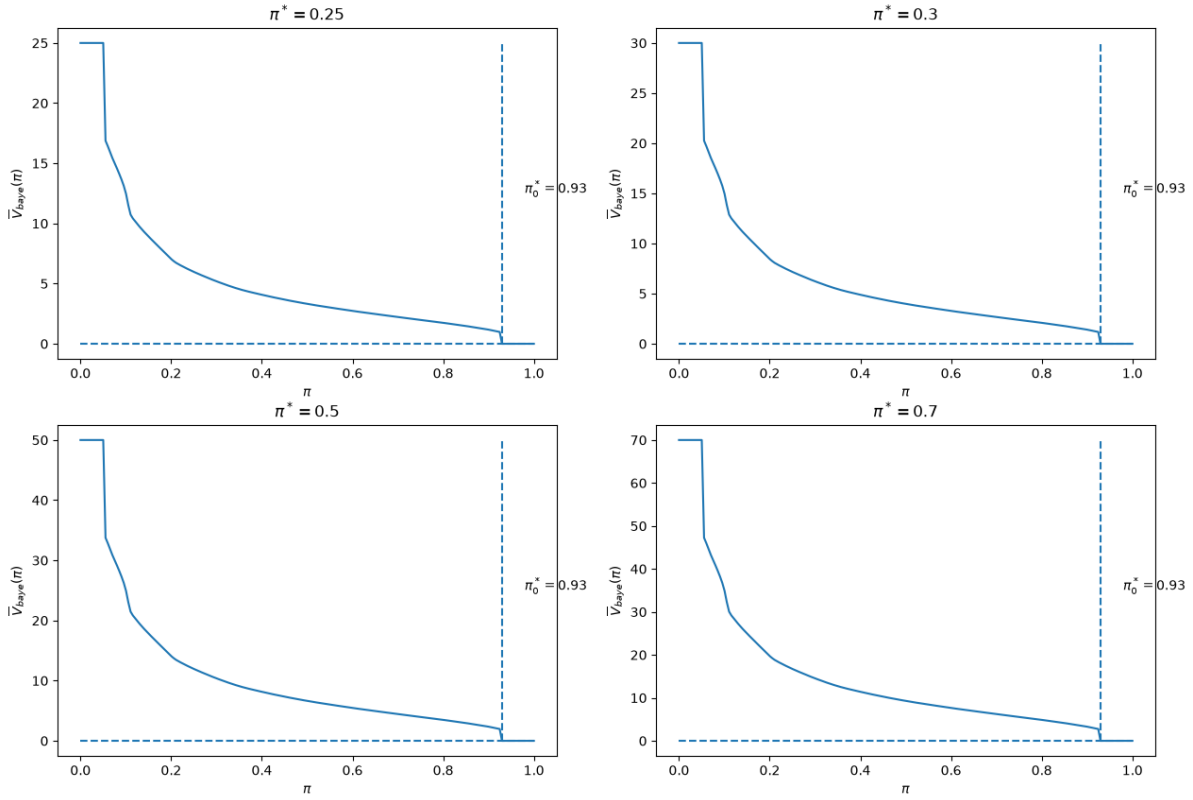
The figure below plots four cases corresponding to $\pi^* = 0.25, 0.3, 0.5, 0.7$.

We observe that in each case π_0^* equals π^*.

```
def compute_V_baye_bar(π_star, V0, V1, wf):

    V_baye = π_star * V0 + (1 - π_star) * V1
    π_idx = np.argmin(V_baye)
    π_optimal = wf.π_grid[π_idx]
    V_baye_bar = V_baye[π_idx]

    return V_baye, π_optimal, V_baye_bar
```

```
π_star_arr = [0.25, 0.3, 0.5, 0.7]

fig, axs = plt.subplots(2, 2, figsize=(15, 10))

for i, π_star in enumerate(π_star_arr):
    row_i = i // 2
    col_i = i % 2

    V_baye, π_optimal, V_baye_bar = compute_V_baye_bar(π_star, V0, V1, wf)

    axs[row_i, col_i].plot(wf.π_grid, V_baye)
    axs[row_i, col_i].hlines(V_baye_bar, 0, 1, linestyle='--')
    axs[row_i, col_i].vlines(π_optimal, V_baye_bar, V_baye.max(), linestyle='--')
    axs[row_i, col_i].text(π_optimal+0.05, (V_baye_bar + V_baye.max()) / 2,
                           r'${\pi_0^*}=$'+f'{π_optimal:0.2f}')
    axs[row_i, col_i].set_xlabel(r'$\pi$')
    axs[row_i, col_i].set_ylabel(r'$\overline{V}_{baye}(\pi)$')
    axs[row_i, col_i].set_title(r'$\pi^*=$' + f'{π_star}')

fig.suptitle(r'$\overline{V}_{baye}(\pi)=\pi^*V^0(\pi) + (1-\pi^*)V^1(\pi)$',␣
 ↪fontsize=16)
plt.show()
```

$$\overline{V}_{baye}(\pi) = \pi^* V^0(\pi) + (1 - \pi^*) V^1(\pi)$$



This pattern of outcomes holds more generally.

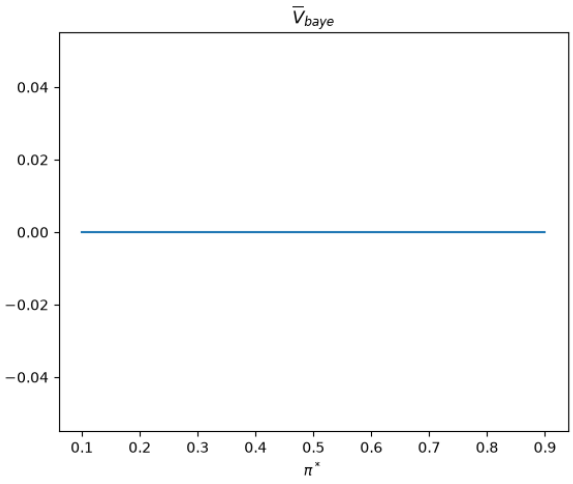
Thus, the following Python code generates the associated graph that verifies the equality of π_0^* to π^* holds for all π^*.

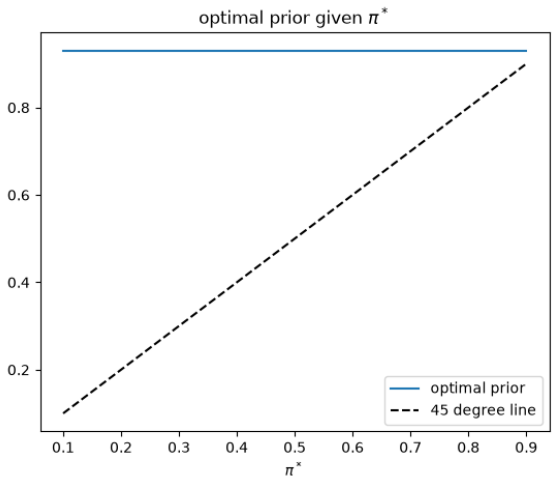
```
π_star_arr = np.linspace(0.1, 0.9, n_π)
V_baye_bar_arr = np.empty_like(π_star_arr)
π_optimal_arr = np.empty_like(π_star_arr)

for i, π_star in enumerate(π_star_arr):

    V_baye, π_optimal, V_baye_bar = compute_V_baye_bar(π_star, V0, V1, wf)

    V_baye_bar_arr[i] = V_baye_bar
    π_optimal_arr[i] = π_optimal

fig, axs = plt.subplots(1, 2, figsize=(14, 5))

axs[0].plot(π_star_arr, V_baye_bar_arr)
axs[0].set_xlabel(r'$\pi^*$')
axs[0].set_title(r'$\overline{V}_{baye}$')

axs[1].plot(π_star_arr, π_optimal_arr, label='optimal prior')
axs[1].plot([π_star_arr.min(), π_star_arr.max()],
            [π_star_arr.min(), π_star_arr.max()],
            c='k', linestyle='--', label='45 degree line')
axs[1].set_xlabel(r'$\pi^*$')
axs[1].set_title(r'optimal prior given $\pi^*$')
axs[1].legend()
```

(continues on next page)

(continued from previous page)

```
plt.show()
```





33.5 Was the Navy Captain's Hunch Correct?

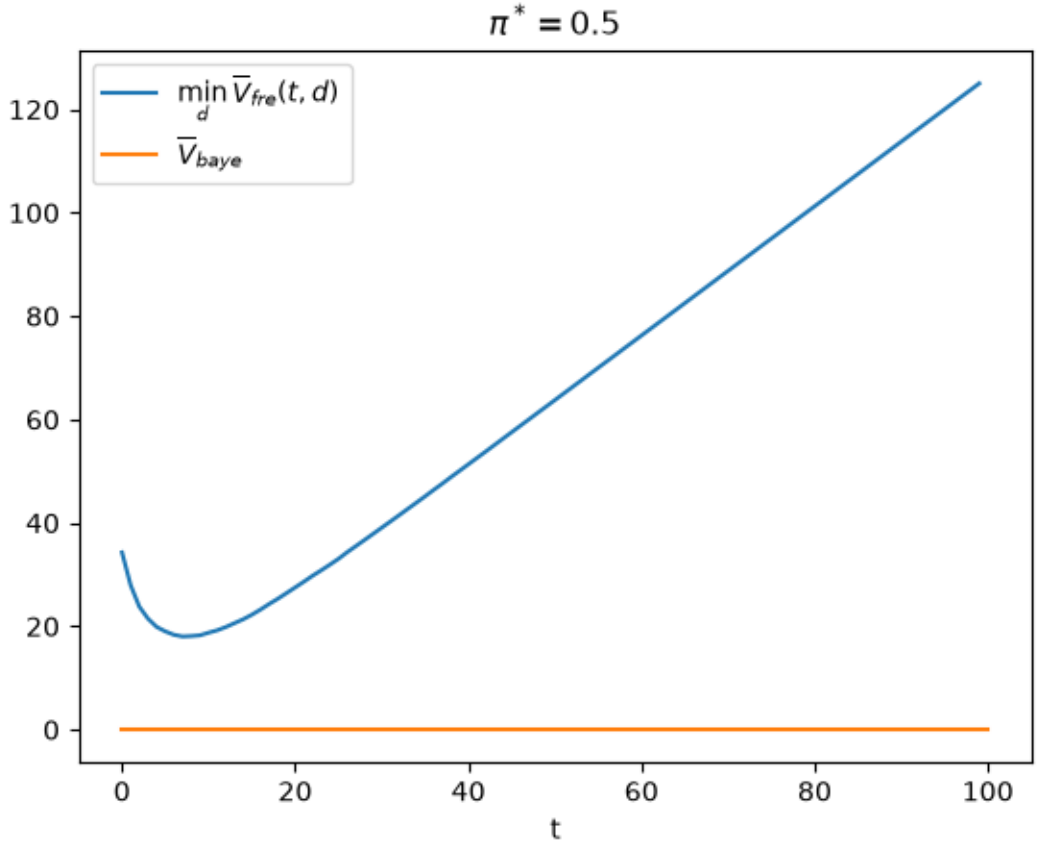
We now compare average losses obtained by our frequentist Neyman-Pearson and Bayesian decision rules.

As a starting point, let's compare average loss functions when $\pi^* = 0.5$.

```
π_star = 0.5
```

```
# frequentist
V_fre_arr, PFA_arr, PD_arr = compute_V_fre(L0_arr, L1_arr, π_star, wf)

# bayesian
V_baye = π_star * V0 + (1 - π_star) * V1
V_baye_bar = V_baye.min()
```

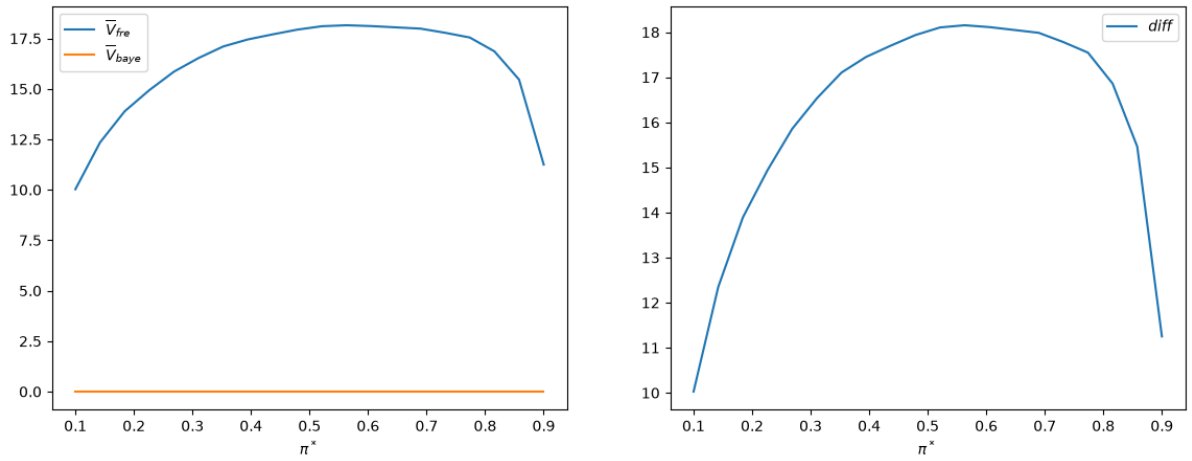
```
plt.plot(range(T), V_fre_arr, label=r'$\min_{d} \overline{V}_{fre}(t,d)$')
plt.plot([0, T], [V_baye_bar, V_baye_bar], label=r'$\overline{V}_{baye}$')
plt.xlabel('t')
plt.title(r'$\pi^*=0.5$')
plt.legend()
plt.show()
```



Evidently, there is no sample size t at which the Neyman-Pearson decision rule attains a lower loss function than does the Bayesian rule.

Furthermore, the following graph indicates that the Bayesian decision rule does better on average for all values of π^*.

```
fig, axs = plt.subplots(1, 2, figsize=(14, 5))

axs[0].plot(π_star_arr, V_fre_bar_arr, label=r'$\overline{V}_{fre}$')
axs[0].plot(π_star_arr, V_baye_bar_arr, label=r'$\overline{V}_{baye}$')
axs[0].legend()
axs[0].set_xlabel(r'$\pi^*$')

axs[1].plot(π_star_arr, V_fre_bar_arr - V_baye_bar_arr, label='$diff$')
axs[1].legend()
axs[1].set_xlabel(r'$\pi^*$')

plt.show()
```



The right panel of the above graph plots the difference $\bar{V}_{fre} - \bar{V}_{Bayes}$.

It is always positive.

33.6 More Details

We can provide more insights by focusing on the case in which $\pi^* = 0.5 = \pi_0$.

```
π_star = 0.5
```

Recall that when $\pi^* = 0.5$, the frequentist Neyman-Pearson decision rule sets a sample size `t_optimal` **ex ante**.

For our parameter settings, we can compute its value:

```
t_optimal
```

```
np.int64(8)
```

For convenience, let's define `t_idx` as the Python array index corresponding to `t_optimal` sample size.

```
t_idx = t_optimal - 1
```

33.7 Distribution of Bayesian Decision Rule's Time to Decide

We use simulations to compute the frequency distribution of the time to decide for the Bayesian decision rule and compare that time to the frequentist rule's fixed t.

The following Python code creates a graph that shows the frequency distribution of Bayesian times to decide of Bayesian decision maker, conditional on distribution $q = f_0$ or $q = f_1$ generating the data.

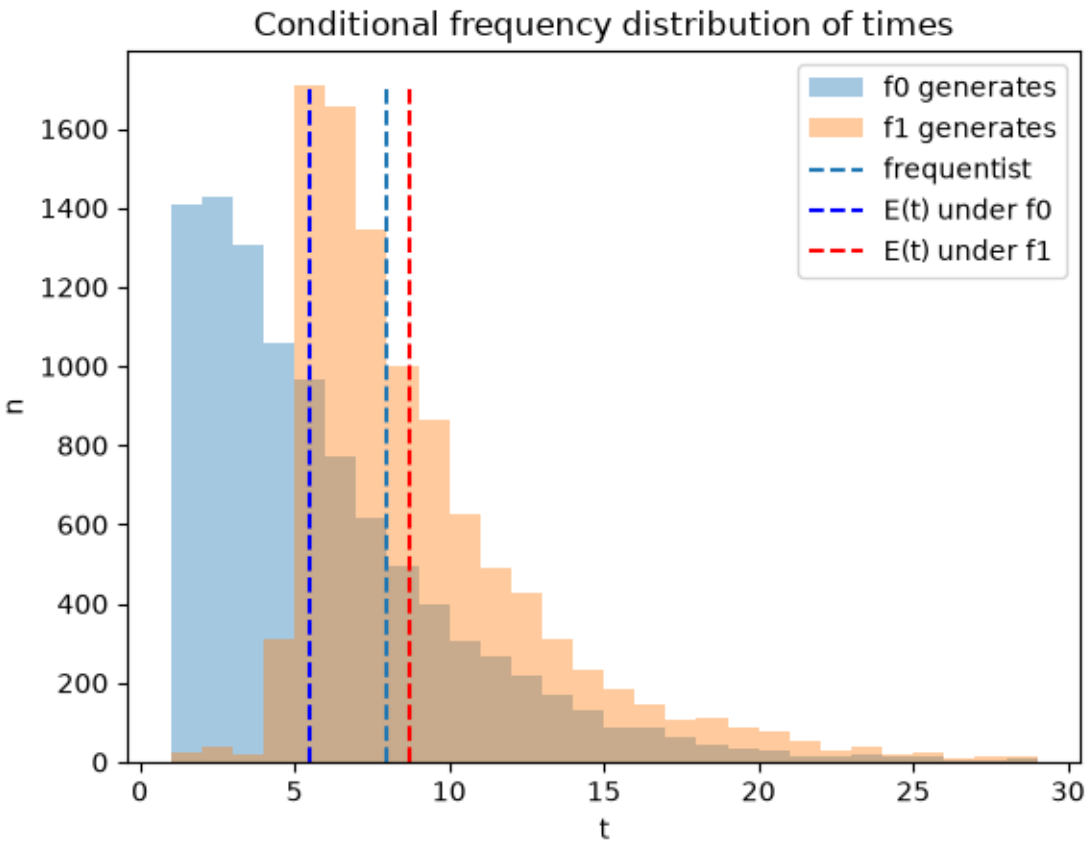
The blue and red dotted lines show averages for the Bayesian decision rule, while the black dotted line shows the frequentist optimal sample size t.

On average the Bayesian rule decides **earlier** than the frequentist rule when $q = f_0$ and **later** when $q = f_1$.

```
@jit(parallel=True)
def check_results(L_arr, A, B, flag, π0):

    N, T = L_arr.shape

    time_arr = np.empty(N)
    correctness = np.empty(N)

    π_arr = π0 * L_arr / (π0 * L_arr + 1 - π0)

    for i in prange(N):
        for t in range(T):
            if (π_arr[i, t] < B) or (π_arr[i, t] > A):
                time_arr[i] = t + 1
                correctness[i] = (flag == 0 and π_arr[i, t] > A) or (flag == 1 and π_
↪arr[i, t] < B)
                break

    return time_arr, correctness
```

```
time_arr0, correctness0 = check_results(L0_arr, A, B, 0, π_star)
time_arr1, correctness1 = check_results(L1_arr, A, B, 1, π_star)

# unconditional distribution
time_arr_u = np.concatenate((time_arr0, time_arr1))
correctness_u = np.concatenate((correctness0, correctness1))
```

```
n1 = plt.hist(time_arr0, bins=range(1, 30), alpha=0.4, label='f0 generates')[0]
n2 = plt.hist(time_arr1, bins=range(1, 30), alpha=0.4, label='f1 generates')[0]
plt.vlines(t_optimal, 0, max(n1.max(), n2.max()), linestyle='--', label='frequentist')
plt.vlines(np.mean(time_arr0), 0, max(n1.max(), n2.max()),
           linestyle='--', color='b', label='E(t) under f0')
plt.vlines(np.mean(time_arr1), 0, max(n1.max(), n2.max()),
           linestyle='--', color='r', label='E(t) under f1')
plt.legend();

plt.xlabel('t')
plt.ylabel('n')
plt.title('Conditional frequency distribution of times')

plt.show()
```



Later we'll figure out how these distributions ultimately affect objective expected values under the Neyman-Pearson and Bayesian decision rules.

To begin, let's look at simulations of the Bayesian's beliefs over time.

We can compute updated beliefs at any time t using the one-to-one mapping from L_t to π_t given π_0 described in this lecture *Likelihood Ratio Processes*.

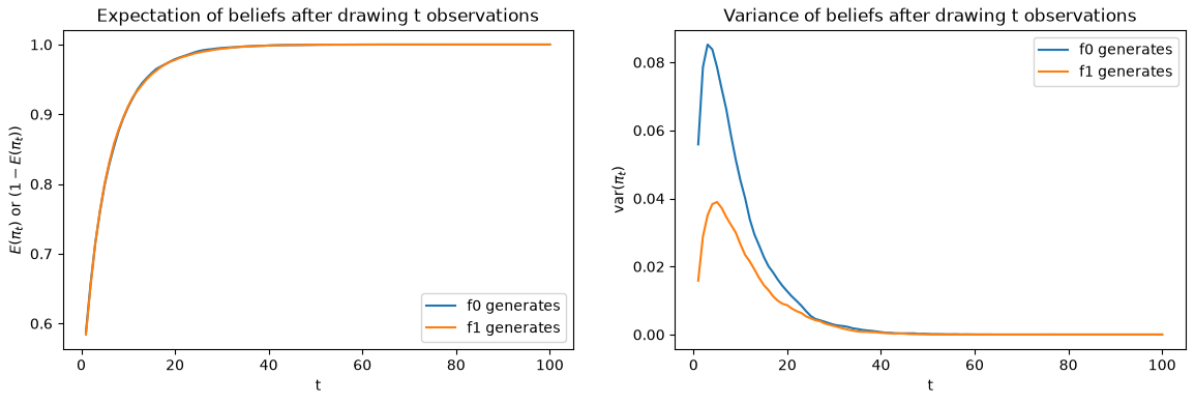
```
π0_arr = π_star * L0_arr / (π_star * L0_arr + 1 - π_star)
π1_arr = π_star * L1_arr / (π_star * L1_arr + 1 - π_star)
```

```
fig, axs = plt.subplots(1, 2, figsize=(14, 4))

axs[0].plot(np.arange(1, π0_arr.shape[1]+1), np.mean(π0_arr, 0), label='f0 generates')
axs[0].plot(np.arange(1, π1_arr.shape[1]+1), 1 - np.mean(π1_arr, 0), label='f1␣
↪generates')
axs[0].set_xlabel('t')
axs[0].set_ylabel(r'$E(\pi_t)$ or ($1 - E(\pi_t)$)')
axs[0].set_title('Expectation of beliefs after drawing t observations')
axs[0].legend()

axs[1].plot(np.arange(1, π0_arr.shape[1]+1), np.var(π0_arr, 0), label='f0 generates')
axs[1].plot(np.arange(1, π1_arr.shape[1]+1), np.var(π1_arr, 0), label='f1 generates')
axs[1].set_xlabel('t')
axs[1].set_ylabel(r'var($\pi_t$)')
axs[1].set_title('Variance of beliefs after drawing t observations')
axs[1].legend()
```

(continues on next page)

(continued from previous page)

```
plt.show()
```



The above figures compare averages and variances of updated Bayesian posteriors after t draws.

The left graph compares $E(\pi_t)$ under f_0 to $1 - E(\pi_t)$ under f_1: they lie on top of each other.

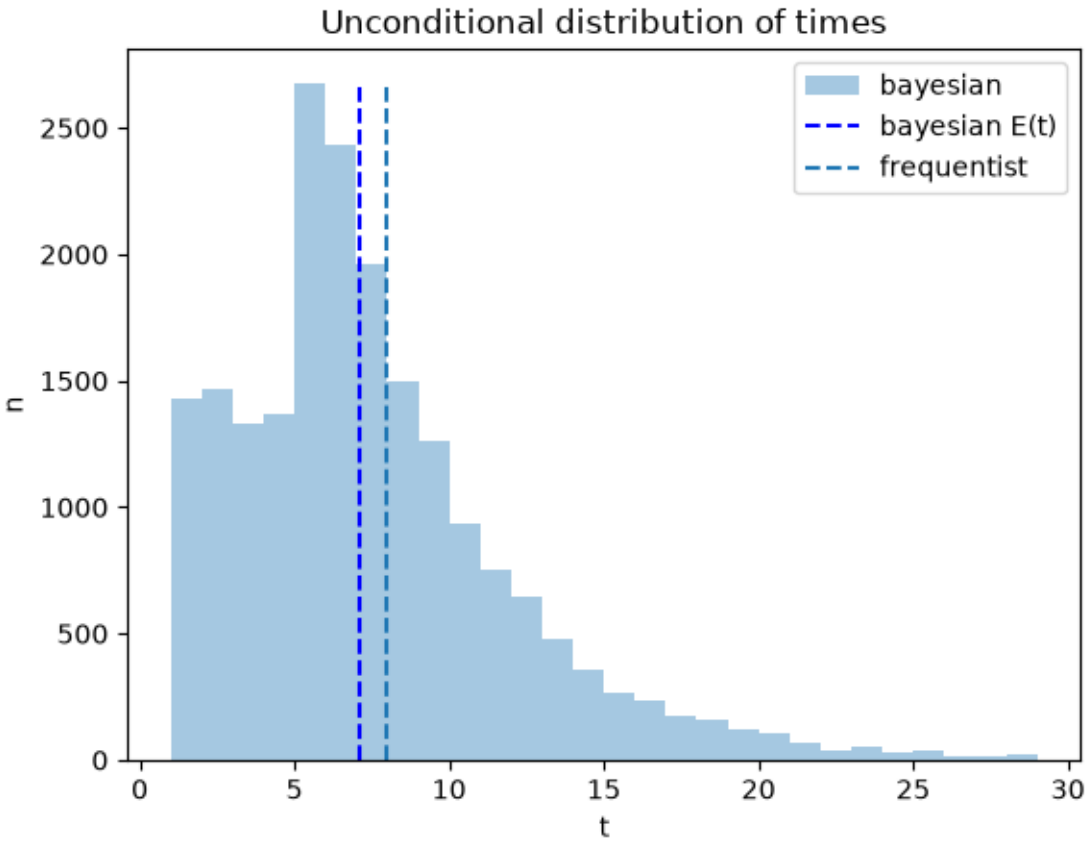
However, as the right hand side graph shows, there is significant difference in variances when t is small: the variance is lower under f_1.

The difference in variances is the reason that the Bayesian decision maker waits longer to decide when f_1 generates the data.

The code below plots outcomes of constructing an unconditional distribution by simply pooling the simulated data across the two possible distributions f_0 and f_1.

The pooled distribution describes a sense in which on average the Bayesian decides earlier, an outcome that seems at least partly to confirm the Navy Captain's hunch.

```
n = plt.hist(time_arr_u, bins=range(1, 30), alpha=0.4, label='bayesian')[0]
plt.vlines(np.mean(time_arr_u), 0, n.max(), linestyle='--',
           color='b', label='bayesian E(t)')
plt.vlines(t_optimal, 0, n.max(), linestyle='--', label='frequentist')
plt.legend()

plt.xlabel('t')
plt.ylabel('n')
plt.title('Unconditional distribution of times')

plt.show()
```



33.8 Probability of Making Correct Decision

Now we use simulations to compute the fractions of samples in which the Bayesian and the frequentist Neyman-Pearson decision rules decide correctly.

For the frequentist Neyman-Pearson rule, the probability of making the correct decision under f_1 is the optimal probability of detection given t that we defined earlier, and similarly it equals 1 minus the optimal probability of a false alarm under f_0.

Below we plot these two probabilities for the frequentist rule, along with the conditional probabilities that the Bayesian rule decides before t *and* that the decision is correct.

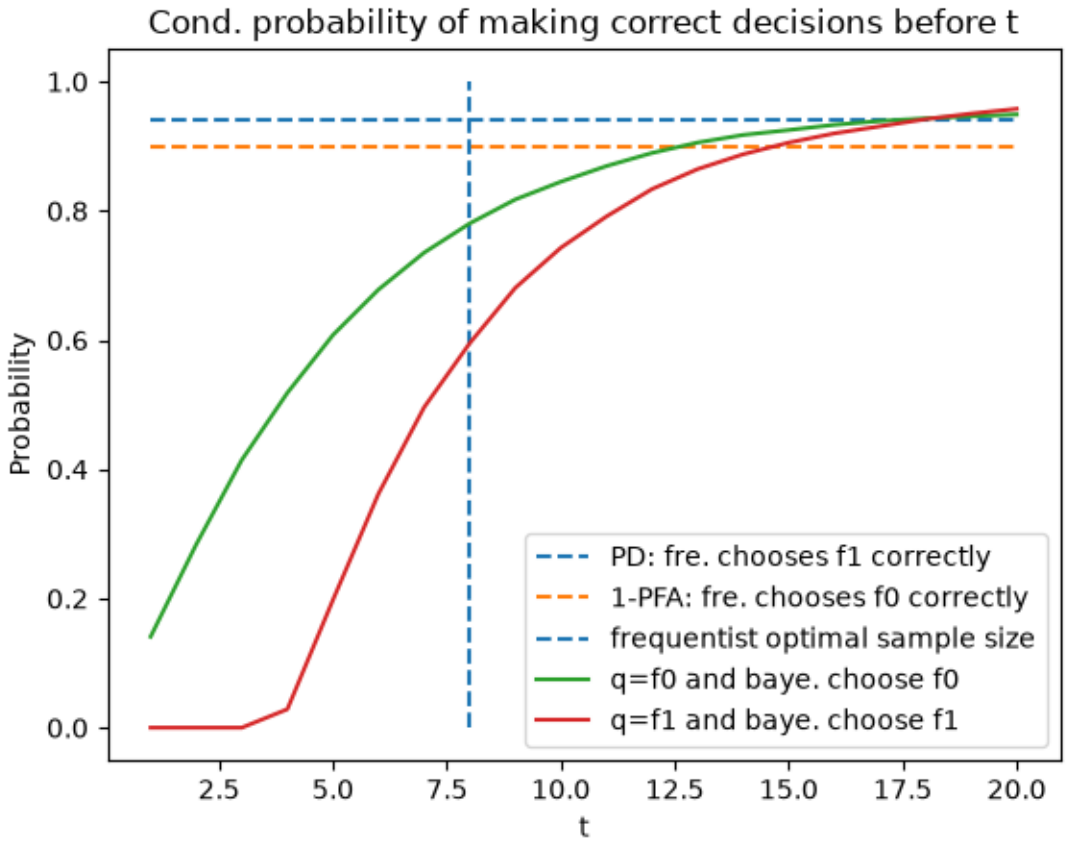
```
# optimal PFA and PD of frequentist with optimal sample size
V, PFA, PD = V_fre_t(t_optimal, L0_arr, L1_arr, π_star, wf)
```

```
plt.plot([1, 20], [PD, PD], linestyle='--', label='PD: fre. chooses f1 correctly')
plt.plot([1, 20], [1-PFA, 1-PFA], linestyle='--', label='1-PFA: fre. chooses f0␣
↪correctly')
plt.vlines(t_optimal, 0, 1, linestyle='--', label='frequentist optimal sample size')

N = time_arr0.size
T_arr = np.arange(1, 21)
plt.plot(T_arr, [np.sum(correctness0[time_arr0 <= t] == 1) / N for t in T_arr],
         label='q=f0 and baye. choose f0')
plt.plot(T_arr, [np.sum(correctness1[time_arr1 <= t] == 1) / N for t in T_arr],
```

(continues on next page)

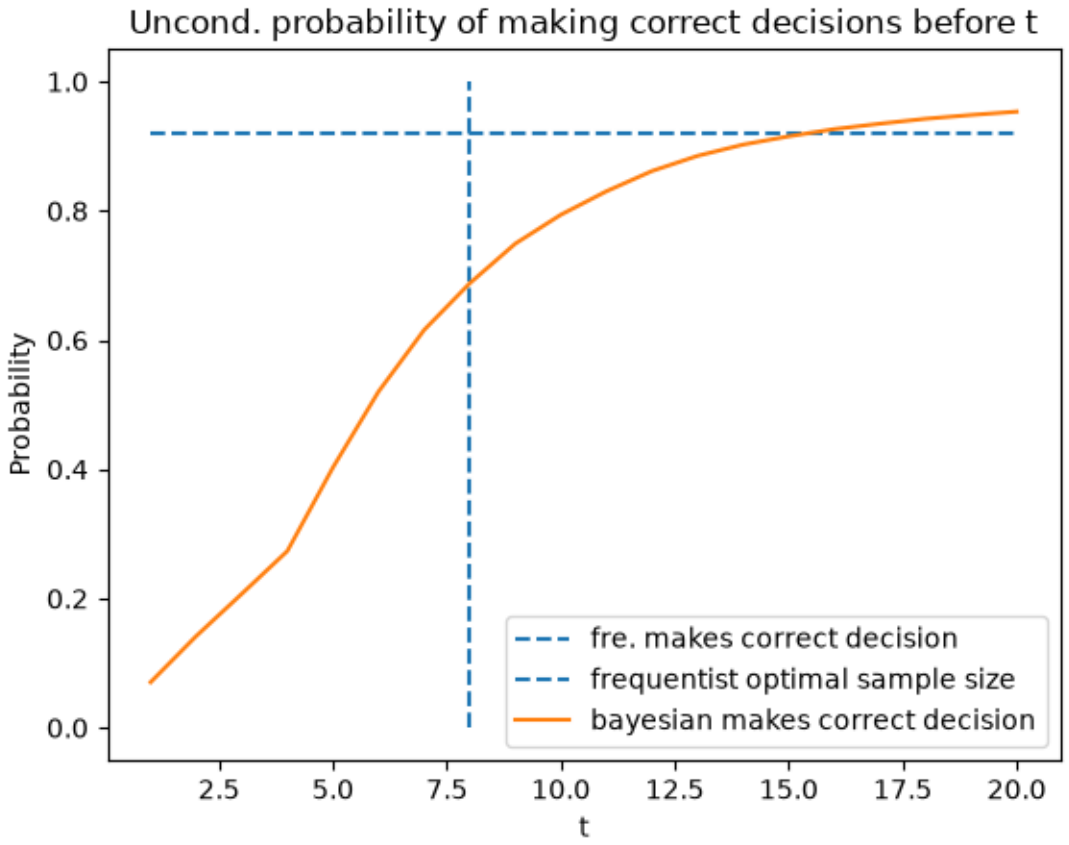
(continued from previous page)

```
        label='q=f1 and baye. choose f1')
plt.legend(loc=4)

plt.xlabel('t')
plt.ylabel('Probability')
plt.title('Cond. probability of making correct decisions before t')

plt.show()
```



By averaging using π^*, we also plot the unconditional distribution.

```
plt.plot([1, 20], [(PD + 1 - PFA) / 2, (PD + 1 - PFA) / 2],
        linestyle='--', label='fre. makes correct decision')
plt.vlines(t_optimal, 0, 1, linestyle='--', label='frequentist optimal sample size')

N = time_arr_u.size
plt.plot(T_arr, [np.sum(correctness_u[time_arr_u <= t] == 1) / N for t in T_arr],
        label="bayesian makes correct decision")
plt.legend()

plt.xlabel('t')
plt.ylabel('Probability')
plt.title('Uncond. probability of making correct decisions before t')

plt.show()
```



33.9 Distribution of Likelihood Ratios at Neyman-Pearson's t

Next we use simulations to construct distributions of likelihood ratios after t draws.

To serve as useful reference points, we also show likelihood ratios that correspond to the Bayesian cutoffs A and B.

In order to exhibit the distribution more clearly, we report logarithms of likelihood ratios.

The graphs below reports two distributions, one conditional on f_0 generating the data, the other conditional on f_1 generating the data.

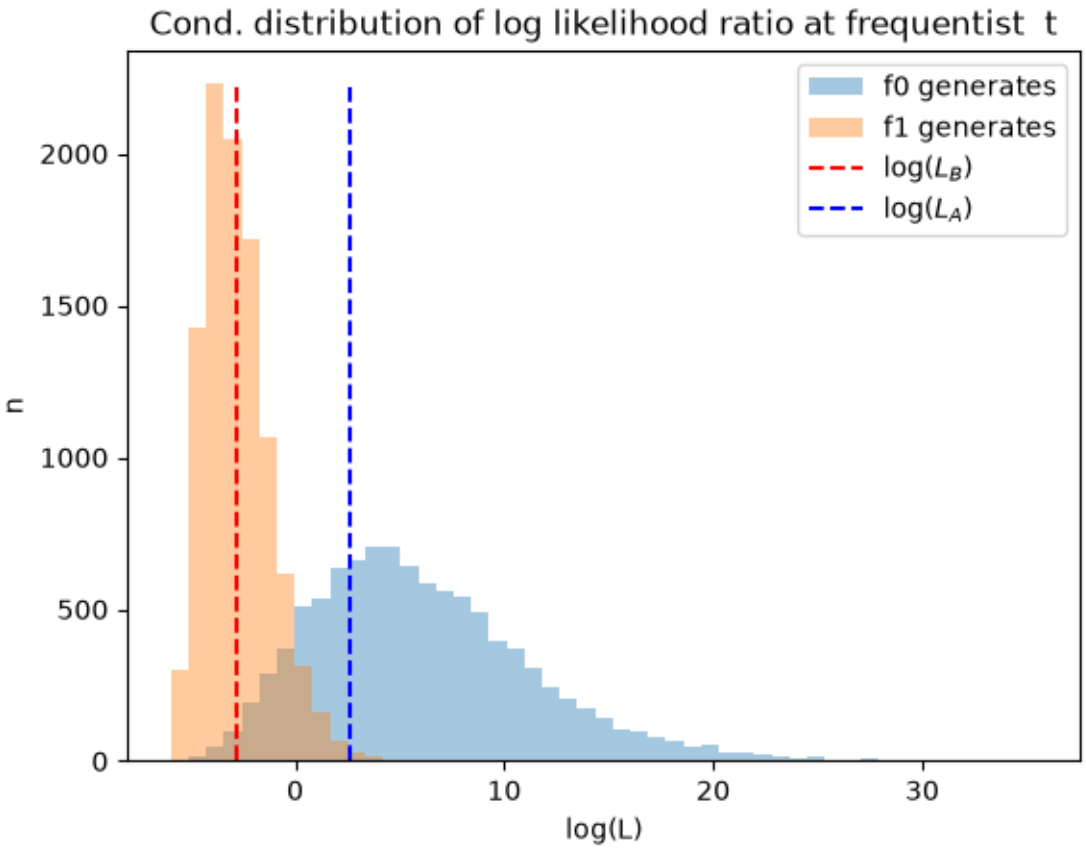
```
LA = (1 - π_star) *  A / (π_star - π_star * A)
LB = (1 - π_star) *  B / (π_star - π_star * B)
```

```
L_min = min(L0_arr[:, t_idx].min(), L1_arr[:, t_idx].min())
L_max = max(L0_arr[:, t_idx].max(), L1_arr[:, t_idx].max())
bin_range = np.linspace(np.log(L_min), np.log(L_max), 50)
n0 = plt.hist(np.log(L0_arr[:, t_idx]), bins=bin_range, alpha=0.4, label='f0 generates
 ↪')[0]
n1 = plt.hist(np.log(L1_arr[:, t_idx]), bins=bin_range, alpha=0.4, label='f1 generates
 ↪')[0]

plt.vlines(np.log(LB), 0, max(n0.max(), n1.max()), linestyle='--', color='r', label=
 ↪'log($L_B$)')
plt.vlines(np.log(LA), 0, max(n0.max(), n1.max()), linestyle='--', color='b', label=
```

(continues on next page)

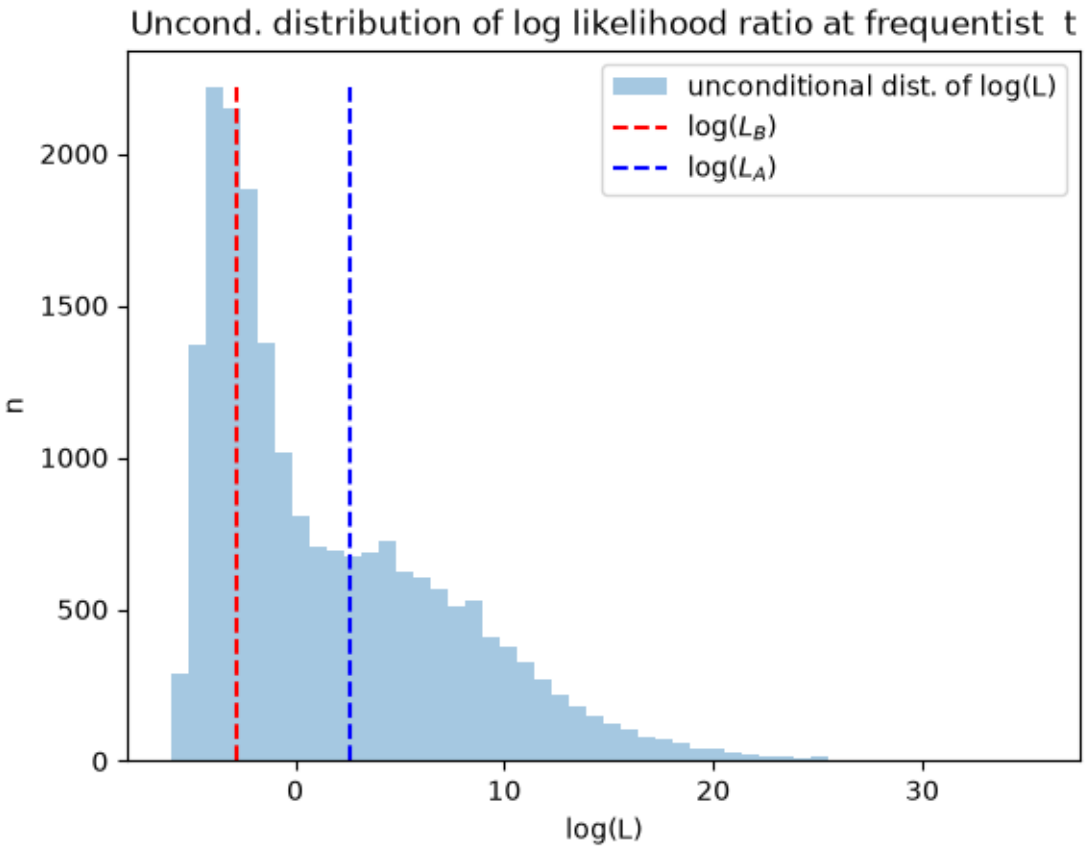
(continued from previous page)

```
 ↪'log($L_A$)')
plt.legend()

plt.xlabel('log(L)')
plt.ylabel('n')
plt.title('Cond. distribution of log likelihood ratio at frequentist  t')

plt.show()
```



The next graph plots the unconditional distribution of Bayesian times to decide, constructed as earlier by pooling the two conditional distributions.

```
plt.hist(np.log(np.concatenate([L0_arr[:, t_idx], L1_arr[:, t_idx]])),
         bins=50, alpha=0.4, label='unconditional dist. of log(L)')
plt.vlines(np.log(LB), 0, max(n0.max(), n1.max()), linestyle='--', color='r', label=
 ↪'log($L_B$)')
plt.vlines(np.log(LA), 0, max(n0.max(), n1.max()), linestyle='--', color='b', label=
 ↪'log($L_A$)')
plt.legend()

plt.xlabel('log(L)')
plt.ylabel('n')
plt.title('Uncond. distribution of log likelihood ratio at frequentist  t')

plt.show()
```


Uncond. distribution of log likelihood ratio at frequentist t
unconditional dist. of log(L)
log(L_B)
log(L_A)
n
2000
1500
1000
500
0
0
10
20
30
log(L)

CHAPTER

THIRTYFOUR

MERGING OF OPINIONS: THE BLACKWELL–DUBINS THEOREM

Contents

- *Merging of Opinions: The Blackwell–Dubins Theorem*
  - *Overview*
  - *Probability measures on sequence spaces*
  - *The likelihood-ratio martingale*
  - *The Blackwell–Dubins theorem*
  - *The Beta–Bernoulli model*
  - *Failure of merging: mutual singularity*
  - *Kakutani's theorem: when does merging hold?*
  - *Extension to continuous time*
  - *Applications*
  - *The rate of merging*
  - *Summary and extensions*

34.1 Overview

This lecture studies the merging-of-opinions theorem of Blackwell and Dubins [1962].

The theorem asks a simple question:

> If two agents hold different prior beliefs about a stochastic process but observe the same stream of data indefinitely, will their probability assessments eventually converge?

The answer is yes under an absolute-continuity condition.

If $Q \ll P$ (that is, P dominates Q), then the conditional distributions under P and Q over the entire future path merge in total variation, Q-almost surely.

If in addition $P \ll Q$ (so that $P \sim Q$), then the same conclusion holds under both agents' probabilities.

This result connects to several other ideas:

- Bayesian consistency: posterior predictions approach the truth when the prior lies in the right absolute-continuity class (*Likelihood Ratio Processes and Bayesian Learning*).
- Agreement results: common data can eliminate disagreement even when agents start from different priors (Aumann [1976]).
- Kakutani's dichotomy: for product measures, equivalence versus singularity can be read from a Hellinger criterion.

We develop the theory in discrete time and then sketch the continuous-time analogue.

Throughout, we use the Beta–Bernoulli model as a running example.

Two agents observe the same stream of coin flips but start from different priors over the coin's bias.

Let us start with some imports.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import beta as beta_dist
from scipy.special import betaln
```

34.2 Probability measures on sequence spaces

34.2.1 The sequence space and its filtration

Let $(S, \mathcal{S})$ be a standard Borel space (i.e., a measurable space isomorphic to a Borel subset of a complete separable metric space), called the signal space.

The standard Borel assumption guarantees the existence of regular conditional distributions, which the theorem requires.

Set $\Omega = S^{\mathbb{N}}$, the set of all infinite sequences $\omega = (x_1, x_2, \ldots)$ with $x_n \in S$, equipped with the product σ-algebra $\mathcal{F} = \mathcal{S}^{\otimes\mathbb{N}}$.

For each $n \geq 1$, define the *finite-horizon* σ-algebra

$$\mathcal{F}_n = \sigma(x_1, \ldots, x_n),$$

so $\mathcal{F}_1 \subseteq \mathcal{F}_2 \subseteq \cdots \subseteq \mathcal{F}$.

Define the **tail σ-algebra** $\mathcal{F}_\infty = \sigma\left(\bigcup_{n\geq 1} \mathcal{F}_n\right)$, which encodes everything that can eventually be learned.

The collection $\{\mathcal{F}_n\}_{n\geq 1}$ is the **natural filtration** generated by the observation process; $\mathcal{F}_n$ encodes everything that can be learned from the first n data points.

Let P and Q denote two probability measures on $(\Omega, \mathcal{F})$.

Write $P_n = P|_{\mathcal{F}_n}$ and $Q_n = Q|_{\mathcal{F}_n}$ for their restrictions to the history up to time n.

34.2.2 Absolute continuity

Definition 34.2.1 (Absolute Continuity)

P is **absolutely continuous** with respect to Q, written $P \ll Q$, if $Q(A) = 0$ implies $P(A) = 0$ for every $A \in \mathcal{F}$.

They are **mutually absolutely continuous**, or **equivalent**, written $P \sim Q$, if both $P \ll Q$ and $Q \ll P$.

P is **locally absolutely continuous** with respect to Q if $P_n \ll Q_n$ for every $n \geq 1$.

Global absolute continuity $P \ll Q$ implies local absolute continuity, but not conversely.

Mutual absolute continuity means the two agents agree on which events are *possible*.

They can disagree about probabilities, but neither agent rules out an event the other deems possible.

34.2.3 Total variation distance

Definition 34.2.2 (Total Variation Distance)

For two probability measures μ and ν on $(E, \mathcal{E})$,

$$\|\mu - \nu\|_{\text{TV}} = \sup_{A \in \mathcal{E}} |\mu(A) - \nu(A)| = \frac{1}{2} \int_E \left| \frac{d\mu}{d\lambda} - \frac{d\nu}{d\lambda} \right| d\lambda,$$

where λ is any **dominating measure**, meaning $\mu \ll \lambda$ and $\nu \ll \lambda$ (for example, $\lambda = \mu + \nu$).

Equivalently, $\|\mu - \nu\|_{\text{TV}} \in [0, 1]$, with 0 meaning $\mu = \nu$ and 1 meaning $\mu \perp \nu$ (mutual singularity).

When $\mu \ll \nu$ with $f = d\mu/d\nu$,

$$\|\mu - \nu\|_{\text{TV}} = \mathbb{E}_\nu[(f-1)^+] = 1 - \mathbb{E}_\nu[\min(f, 1)].$$

Exercise 34.2.1

Show the identity above.

Hint: Start from $\|\mu - \nu\|_{\text{TV}} = \frac{1}{2}\,\mathbb{E}_\nu[|f-1|]$ (which follows from taking ν as the dominating measure) and use the fact that $\mathbb{E}_\nu[f] = 1$.

Solution

Since $\mu \ll \nu$, we can use ν as the dominating measure, so $d\mu/d\nu = f$ and $d\nu/d\nu = 1$, giving

$$\|\mu - \nu\|_{\text{TV}} = \tfrac{1}{2}\,\mathbb{E}_\nu[|f-1|].$$

Write $|f-1| = (f-1)^+ + (1-f)^+$.

Since μ is a probability measure, $\mathbb{E}_\nu[f] = 1$, so the two parts contribute equally: $\mathbb{E}_\nu[(f-1)^+] = \mathbb{E}_\nu[(1-f)^+]$.

Therefore $\frac{1}{2}\,\mathbb{E}_\nu[|f-1|] = \mathbb{E}_\nu[(f-1)^+]$.

Next, note that $(f-1)^+ = f - \min(f, 1)$, so $\mathbb{E}_\nu[(f-1)^+] = \mathbb{E}_\nu[f] - \mathbb{E}_\nu[\min(f,1)] = 1 - \mathbb{E}_\nu[\min(f,1)]$.

Total variation is one of the strongest standard notions of distance between probability measures.

If two measures are close in total variation, then their probabilities of every event are close.

34.2.4 The merging question

The Blackwell–Dubins theorem studies the conditional distribution of the *future* given the *past*.

At time n, after observing $(x_1, \dots, x_n)$, each agent forms a conditional distribution over all future events:

$$P(\cdot \mid \mathcal{F}_n)(\omega), \qquad Q(\cdot \mid \mathcal{F}_n)(\omega).$$

These are probability measures on the whole future path, not just the next observation.

The merging question asks whether

$$d_n \;:=\; \big\| P(\cdot \mid \mathcal{F}_n) - Q(\cdot \mid \mathcal{F}_n) \big\|_{\mathrm{TV}} \;\longrightarrow\; 0$$

almost surely as $n \to \infty$.

34.3 The likelihood-ratio martingale

Our main tool is the Radon–Nikodym derivative process.

34.3.1 The likelihood ratio

Since $Q \ll P$ implies $Q_n \ll P_n$ for every n, the Radon–Nikodym theorem guarantees the existence of the likelihood ratio

$$Z_n = \frac{dQ_n}{dP_n}, \qquad Z_n \geq 0 \;\; P\text{-a.s.}, \qquad \mathbb{E}_P[Z_n] = 1.$$

The key structural property is that global absolute continuity $Q \ll P$ implies the existence of an overall Radon–Nikodym derivative $Z = dQ/dP$ on all of $(\Omega, \mathcal{F})$, and

$$Z_n = \mathbb{E}_P[Z \mid \mathcal{F}_n] \qquad P\text{-a.s.}$$

That is, $\{Z_n, \mathcal{F}_n\}_{n \geq 1}$ is a non-negative, uniformly integrable P-martingale.

Lemma 34.3.1 (Martingale Convergence)

The likelihood-ratio process $\{Z_n\}$ satisfies:

1. $Z_n \to Z_\infty$ P-almost surely as $n \to \infty$.
2. $Z_\infty = \mathbb{E}_P[Z \mid \mathcal{F}_\infty]$ P-a.s.
3. $Z_n \to Z_\infty$ in $L^1(P)$: $\mathbb{E}_P[|Z_n - Z_\infty|] \to 0$.

Proof sketch. Non-negativity and the martingale property give boundedness in $L^1(P)$.

Then almost-sure convergence follows from Doob's martingale convergence theorem Doob [1953].

Uniform integrability (which follows from $Z \in L^1(P)$ via the conditional Jensen inequality) upgrades this to $L^1(P)$ convergence. □

34.3.2 Connecting conditional measures to the likelihood ratio

The following identity connects the likelihood ratio to the conditional distributions.

On the set $\{Z_n > 0\}$, the Radon–Nikodym derivative of $Q(\cdot \mid \mathcal{F}_n)$ with respect to $P(\cdot \mid \mathcal{F}_n)$ is

$$\frac{d\,Q(\cdot \mid \mathcal{F}_n)}{d\,P(\cdot \mid \mathcal{F}_n)} = \frac{Z_\infty}{Z_n} \qquad P\text{-a.s. on } \{Z_n > 0\}.$$

Applying the total-variation formula with $f = Z_\infty / Z_n$ then gives

$$d_n = \mathbb{E}_{P(\cdot|\mathcal{F}_n)}\left[\left(\frac{Z_\infty}{Z_n} - 1\right)^+\right] = 1 - \mathbb{E}_{P(\cdot|\mathcal{F}_n)}\left[\min\left(\frac{Z_\infty}{Z_n},\, 1\right)\right].$$

Multiplying through by Z_n and taking the P-expectation (then using $\mathbb{E}_P[Z_n\, g(\mathcal{F}_n)] = \mathbb{E}_Q[g(\mathcal{F}_n)]$ for $\mathcal{F}_n$-measurable g):

$$2\,\mathbb{E}_Q[d_n] \;=\; \mathbb{E}_P[|Z_\infty - Z_n|],$$

So the $L^1(P)$ convergence of the martingale controls how fast the total variation distance goes to zero.

34.4 The Blackwell–Dubins theorem

Theorem 34.4.1 (Blackwell–Dubins (1962))

Let P and Q be probability measures on $(\Omega, \mathcal{F})$ with $Q \ll P$.

Define

$$d_n = \left\| P(\cdot \mid \mathcal{F}_n) - Q(\cdot \mid \mathcal{F}_n) \right\|_{\mathrm{TV}}.$$

Then $d_n \to 0$ Q-almost surely.

The proof has three steps.

Step 1. Representation of d_n via Z_n.

As shown above, d_n can be written in terms of Z_∞ / Z_n, where $Z_n = \mathbb{E}_P[Z \mid \mathcal{F}_n]$ and $Z = dQ/dP$.

This reduces the problem to a statement about one martingale under P.

Step 2. $\{d_n\}$ is a non-negative supermartingale.

Conditioning on more information reduces distinguishability on average.

Formally, because $P(\cdot \mid \mathcal{F}_n) = \mathbb{E}[P(\cdot \mid \mathcal{F}_{n+1}) \mid \mathcal{F}_n]$ and total variation is convex,

$$\mathbb{E}_Q[d_{n+1} \mid \mathcal{F}_n] \le d_n \qquad Q\text{-a.s.}$$

So $\{d_n, \mathcal{F}_n\}$ is a non-negative Q-supermartingale in $[0, 1]$.

By Doob's theorem, $d_n \to d_\infty$ Q-almost surely for some $[0, 1]$-valued random variable d_∞.

Step 3. The almost-sure limit is zero.

From Step 1 and the L^1 bound:

$$\mathbb{E}_Q[d_n] = \tfrac{1}{2}\,\mathbb{E}_P[|Z_\infty - Z_n|] \to 0.$$

The right-hand side vanishes by $L^1(P)$ convergence of the martingale.

Hence $d_n \to 0$ in $L^1(Q)$ and therefore in Q-probability.

Since d_n already converges Q-almost surely, its limit must satisfy $d_\infty = 0$ Q-a.s. □

Remark 34.4.1 (One-Sided vs. Mutual Absolute Continuity)

The theorem requires only $Q \ll P$, not $P \ll Q$.

One-sided absolute continuity $Q \ll P$ gives merging Q-almost surely.

Since $Q \ll P$ means every P-null set is also Q-null, Q-a.s. convergence does *not* automatically imply P-a.s. convergence.

To conclude that $d_n \to 0$ under *both* agents' measures, one needs mutual absolute continuity $P \sim Q$.

With $P \ll Q$ added, the proof can be run with the roles of P and Q swapped, yielding $d_n \to 0$ P-a.s. as well.

Remark 34.4.2 (Sharpness)

Absolute continuity matters.

When P and Q are singular, merging can fail completely.

The point-mass example below has $d_n = 1$ for every n.

For product measures, Kakutani's theorem later gives a sharp equivalence-versus-singularity dichotomy.

34.5 The Beta–Bernoulli model

Before turning to Python, we introduce the main example used throughout the simulations.

34.5.1 Model

Suppose the data stream $(x_1, x_2, ...)$ consists of IID Bernoulli draws with unknown probability $p^* \in (0, 1)$.

Agent i has a Beta prior:

$$p \sim \text{Beta}(\alpha_i, \beta_i), \qquad i = 1, 2.$$

After observing n draws with k successes, Bayes' rule yields the posterior

$$p \,|\, x^n \;\sim\; \text{Beta}(\alpha_i + k,\; \beta_i + n - k),$$

and the one-step-ahead predictive probability is

$$\hat{p}_i^n = \mathbb{E}[p \,|\, x^n] = \frac{\alpha_i + k}{\alpha_i + \beta_i + n}.$$

By the strong law of large numbers, $k/n \to p^*$ almost surely, so both $\hat{p}_1^n$ and $\hat{p}_2^n$ converge to p^* regardless of the agents' initial priors (α_i, β_i).

34.5.2 The marginal likelihood and likelihood ratio

For each fixed value of $p \in (0,1)$, let P_p denote the IID Bernoulli(p) probability law on infinite sequences.

Agent i does not know p.

Instead, agent i places the prior density π_i on p, which induces a probability measure P_i on data sequences via

$$P_i(A) = \int_0^1 P_p(A)\,\pi_i(p)\,dp \qquad \text{for every event } A.$$

So P_i is the agent's marginal probability measure over histories after averaging over uncertainty about p.

In particular, if x^n is an exact observed history with k successes, then $P_i(x^n)$ means the probability that agent i assigns to that history under this mixture measure.

To compute it, start from the Beta density

$$\pi_i(p) = \frac{p^{\alpha_i-1}(1-p)^{\beta_i-1}}{B(\alpha_i,\beta_i)}, \qquad 0 < p < 1.$$

Given p, the probability of that ordered history is $p^k(1-p)^{n-k}$.

Therefore

$$\begin{aligned} P_i(x^n) &= \int_0^1 p^k(1-p)^{n-k}\pi_i(p)\,dp \\ &= \frac{1}{B(\alpha_i,\beta_i)}\int_0^1 p^{\alpha_i+k-1}(1-p)^{\beta_i+n-k-1}\,dp \\ &= \frac{B(\alpha_i+k,\ \beta_i+n-k)}{B(\alpha_i,\ \beta_i)}. \end{aligned}$$

where $B(a,b) = \Gamma(a)\Gamma(b)/\Gamma(a+b)$ is the beta function.

This expression is the probability of the ordered history x^n.

It depends on the data only through the count k, so histories with the same number of successes receive the same probability.

The likelihood ratio at time n is therefore

$$Z_n = \frac{P_{1,n}(x^n)}{P_{2,n}(x^n)} = \frac{B(\alpha_2,\ \beta_2)}{B(\alpha_1,\ \beta_1)} \cdot \frac{B(\alpha_1+k,\ \beta_1+n-k)}{B(\alpha_2+k,\ \beta_2+n-k)}.$$

This is a martingale under P_2 (agent 2's probability) and converges almost surely to a finite positive limit Z_∞, reflecting the fact that $P_1 \sim P_2$ for any Beta priors with positive parameters.

34.5.3 The exact Blackwell–Dubins distance

For the Beta–Bernoulli model, there is a clean formula for d_n.

By de Finetti's theorem, each agent's conditional distribution of the *future infinite sequence* given the past is a mixture of IID Bernoulli(p) processes, where p is drawn from the posterior Beta distribution.

Since the Bernoulli$(p)^\infty$ measures for different p are mutually singular (the empirical frequency identifies p exactly), the TV distance between the two conditional distributions over the future equals the TV distance between the two posterior distributions over the parameter p.

The TV distance is

$$d_n = \left\| \text{Beta}(\alpha_1 + k_n,\, \beta_1 + n - k_n) - \text{Beta}(\alpha_2 + k_n,\, \beta_2 + n - k_n) \right\|_{\text{TV}}.$$

As $k_n/n \to p^*$ and $n \to \infty$, both posterior Betas concentrate around p^* with variance of order $1/n$, so $d_n \to 0$.

The following code implements the Beta–Bernoulli updating, predictive probabilities, TV distance, and likelihood-ratio computations described above.

```
def beta_bernoulli_update(data, a0, b0):
    """
    Sequential Beta-Bernoulli Bayesian updating.
    """
    n = len(data)
    cum_k = np.concatenate([[0], np.cumsum(data)])    # cumulative successes
    ns    = np.arange(n + 1)                           # 0, 1, ..., n
    a_post = a0 + cum_k
    b_post = b0 + (ns - cum_k)
    return a_post, b_post


def predictive_prob(a_post, b_post):
    """One-step-ahead predictive probability P(X=1 | data)."""
    return a_post / (a_post + b_post)


def tv_distance_beta(a1, b1, a2, b2, n_grid=2000):
    """
    TV distance between Beta(a1,b1) and Beta(a2,b2) via grid quadrature.
    Uses a fine grid on (0,1).
    """
    x  = np.linspace(1e-8, 1 - 1e-8, n_grid)
    dx = x[1] - x[0]
    p1 = beta_dist.pdf(x, a1, b1)
    p2 = beta_dist.pdf(x, a2, b2)
    return 0.5 * np.sum(np.abs(p1 - p2)) * dx


def log_likelihood_ratio(data, a1, b1, a2, b2):
    """
    Log likelihood ratio log Z_n = log P1_n(data) - log P2_n(data)
    for every prefix of `data`.

    Returns an array of length len(data) + 1, starting at 0 (before data).
    """
    a1p, b1p = beta_bernoulli_update(data, a1, b1)
    a2p, b2p = beta_bernoulli_update(data, a2, b2)
    log_P1 = betaln(a1p, b1p) - betaln(a1, b1)
    log_P2 = betaln(a2p, b2p) - betaln(a2, b2)
    return log_P1 - log_P2


def run_simulation(p_true, a1, b1, a2, b2, n_steps, seed=0):
    """
    Simulate one realisation of the merging experiment.

    Returns a dict with arrays of length n_steps + 1 (index 0 = prior).
    """
    rng  = np.random.default_rng(seed)
```

(continues on next page)

(continued from previous page)

```
    data = rng.binomial(1, p_true, n_steps)

    a1p, b1p = beta_bernoulli_update(data, a1, b1)
    a2p, b2p = beta_bernoulli_update(data, a2, b2)

    pred1    = predictive_prob(a1p, b1p)
    pred2    = predictive_prob(a2p, b2p)
    tv_1step = np.abs(pred1 - pred2)

    # TV between posterior Betas; in this model this equals d_n
    tv_beta = np.array([
        tv_distance_beta(a1p[i], b1p[i], a2p[i], b2p[i])
        for i in range(n_steps + 1)
    ])

    log_Z = log_likelihood_ratio(data, a1, b1, a2, b2)

    return dict(data=data, pred1=pred1, pred2=pred2,
                tv_1step=tv_1step, tv_beta=tv_beta, log_Z=log_Z)
```

34.5.4 Simulation

We choose two agents with very different beliefs about the bias of a coin whose true probability of heads is $p^* = 0.65$.

- Agent 1 (skeptic): prior Beta$(1, 8)$, so $\hat{p}_1^0 = 1/9 \approx 0.11$.
- Agent 2 (optimist): prior Beta$(8, 1)$, so $\hat{p}_2^0 = 8/9 \approx 0.89$.

Both priors are supported on all of $(0, 1)$, so $P_1 \sim P_2$.

Blackwell–Dubins guarantees merging.

The figure below shows what that merging looks like.

```
p_true = 0.65
a1, b1 = 1.0, 8.0    # skeptic
a2, b2 = 8.0, 1.0    # optimist
n_steps = 600

sim = run_simulation(p_true, a1, b1, a2, b2, n_steps, seed=7)
steps = np.arange(n_steps + 1)

fig, axes = plt.subplots(2, 2, figsize=(11, 7))
ax = axes[0, 0]
ax.plot(steps, sim['pred1'], color='steelblue', lw=2,
        label=r'Agent 1 $\hat p_1^n$ (prior: skeptic)')
ax.plot(steps, sim['pred2'], color='firebrick', lw=2,
        label=r'Agent 2 $\hat p_2^n$ (prior: optimist)')
ax.axhline(p_true, color='black', lw=1.0, ls='--',
           label=f'Truth $p^*={p_true}$')
ax.set_xlabel('observations $n$')
ax.set_ylabel('predictive probability')
ax.set_title('(a) posterior predictive means')
ax.legend(fontsize=8)
ax.set_ylim(0, 1)

ax = axes[0, 1]
```

(continues on next page)

(continued from previous page)

```
ax.semilogy(steps, sim['tv_beta'] + 1e-10, color='mediumpurple', lw=2)
ax.set_xlabel('observations $n$')
ax.set_ylabel(
    r'$d_n = \|P(\cdot|\mathscr{F}_n)'
    r' - Q(\cdot|\mathscr{F}_n)\|_{\mathrm{TV}}$'
)
ax.set_title(r'(b) total-variation distance $d_n$')
ax.set_ylim(bottom=1e-4)

ax = axes[1, 0]
ax.plot(steps, sim['log_Z'], color='darkorange', lw=2)
ax.axhline(0, color='black', lw=0.8, ls=':')
ax.set_xlabel('observations $n$')
ax.set_ylabel(r'$\log Z_n$')
ax.set_title(r'(c) log likelihood ratio')

ax = axes[1, 1]
xs = np.linspace(0.01, 0.99, 500)
epochs = [0, 20, 100, n_steps]
colors = plt.cm.viridis(np.linspace(0.2, 0.85, len(epochs)))

for epoch, col in zip(epochs, colors):
    k_e = int(np.sum(sim['data'][:epoch]))
    pdf1 = beta_dist.pdf(xs, a1 + k_e, b1 + epoch - k_e)
    pdf2 = beta_dist.pdf(xs, a2 + k_e, b2 + epoch - k_e)
    ax.plot(xs, pdf1, color=col, lw=2, ls='-')
    ax.plot(xs, pdf2, color=col, lw=2, ls='--')

ax.axvline(p_true, color='black', lw=1.0, ls=':', label=f'$p^*={p_true}$')
ax.set_xlabel('$p$')
ax.set_ylabel('posterior density')
ax.set_title('(d) posterior densities')

from matplotlib.lines import Line2D
handles = [
    Line2D([0], [0], color='black', lw=2, label='agent 1'),
    Line2D([0], [0], color='black', lw=2, ls='--', label='agent 2'),
]
for epoch, col in zip(epochs, colors):
    handles.append(Line2D([0], [0], color=col, lw=2, label=f'$n={epoch}$'))
handles.append(
    Line2D([0], [0], color='black', lw=1.0, ls=':', label=f'$p^*={p_true}$')
)
ax.legend(handles=handles, fontsize=8)
ax.set_ylim(bottom=0)

plt.tight_layout()
plt.show()
```

The four panels show:

- Panel (a): Starting from $\hat{p}_1^0 \approx 0.11$ and $\hat{p}_2^0 \approx 0.89$, both agents' predictive probabilities converge to $p^* = 0.65$.
- Panel (b): The total-variation distance d_n decays to zero on a logarithmic scale, consistent with the theorem.
- Panel ©: The log likelihood ratio $\log Z_n$ converges to a finite value, which is consistent with mutual absolute continuity in this example.

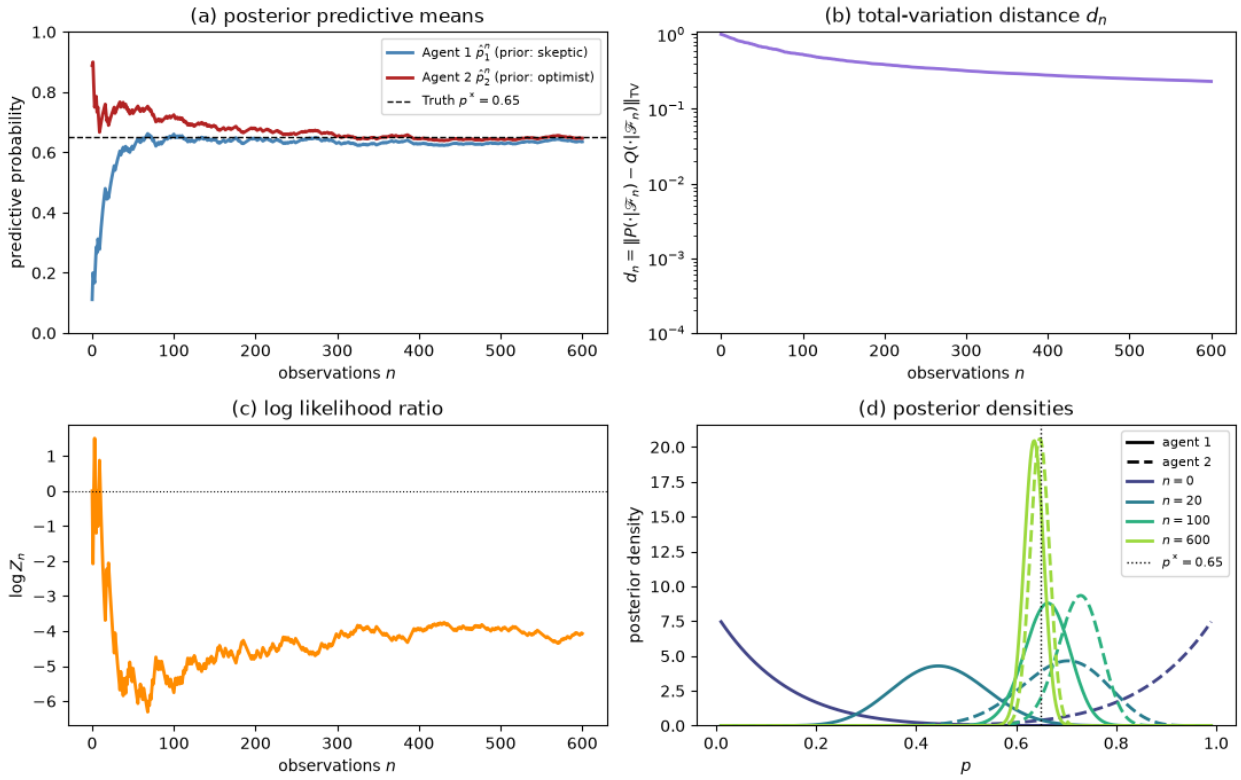


Fig. 34.1: Merging in the Beta–Bernoulli example. The four panels show posterior predictive means, the total-variation distance d_n, the log likelihood ratio $\log Z_n$, and posterior densities at selected horizons.

- Panel (d): The posterior Beta densities for the two agents start far apart (one near 0, one near 1) and progressively concentrate to the same distribution centred on the truth.

34.5.5 Almost-sure convergence across many paths

To illustrate the almost-sure character of the theorem, we run many independent replications.

The theorem concerns almost every path under the reference measure, not just averages across paths.

```
N_paths = 80
n_steps = 500

fig, axes = plt.subplots(1, 2, figsize=(11, 4))

ax_tv  = axes[0]
ax_log = axes[1]

tv_all   = np.empty((N_paths, n_steps + 1))
logZ_all = np.empty((N_paths, n_steps + 1))
steps    = np.arange(n_steps + 1)

for i in range(N_paths):
    s = run_simulation(p_true, a1, b1, a2, b2, n_steps, seed=i)
    tv_all[i]   = s['tv_beta']
    logZ_all[i] = s['log_Z']

for i in range(N_paths):
    ax_tv.semilogy(steps, tv_all[i] + 1e-10, color='steelblue',
                   lw=0.8, alpha=0.3)
ax_tv.semilogy(steps, tv_all.mean(axis=0) + 1e-10,
               color='black', lw=2, label='mean across paths')
ax_tv.set_xlabel('observations $n$')
ax_tv.set_ylabel(r'$d_n$ (log scale)')
ax_tv.legend()

for i in range(N_paths):
    ax_log.plot(steps, logZ_all[i], color='firebrick',
                lw=0.8, alpha=0.3)
ax_log.plot(steps, logZ_all.mean(axis=0),
            color='black', lw=2, label='mean across paths')
ax_log.axhline(0, color='gray', lw=0.8, ls=':')
ax_log.set_xlabel('observations $n$')
ax_log.set_ylabel(r'$\log Z_n$')
ax_log.legend()

plt.tight_layout()
plt.show()

# Finite-horizon summary
frac_below = np.mean(tv_all[:, -1] < 0.30)
mean_final = tv_all[:, -1].mean()
print(f"Fraction of paths with d_n < 0.30 at n = {n_steps}: {frac_below:.2f}")
print(f"Mean distance at n = {n_steps}: {mean_final:.3f}")
```

```
Fraction of paths with d_n < 0.30 at n = 500: 1.00
Mean distance at n = 500: 0.255
```

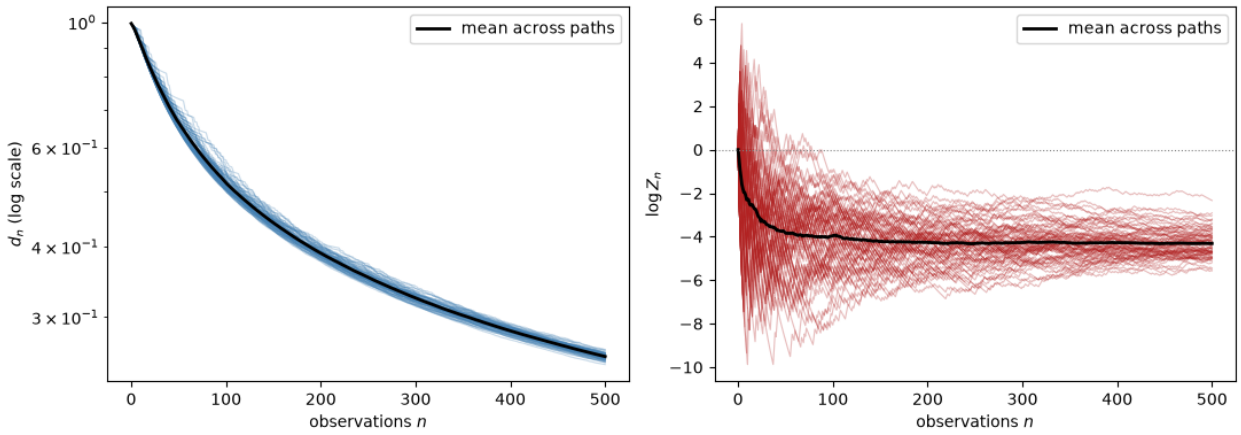


Fig. 34.2: Almost-sure merging across many sample paths. The left panel plots the total-variation distance and the right panel plots the log likelihood ratio $\log Z_n$.

At this finite horizon, the distances have moved down substantially from their initial levels, but they are not yet close to zero.

That is still consistent with the theorem, because almost-sure convergence is an asymptotic statement.

34.5.6 The supermartingale property of d_n

The proof relies on $\{d_n\}$ being a non-negative supermartingale.

We can illustrate this numerically by looking at average increments across many paths.

```
diffs = np.diff(tv_all, axis=1)          # shape (N_paths, n_steps)
mean_diffs = diffs.mean(axis=0)          # average increment at each step
cum_sum    = np.cumsum(mean_diffs)       # cumulative average change

fig, axes = plt.subplots(1, 2, figsize=(10, 4))

ax = axes[0]
ax.plot(mean_diffs[:200], color='purple', lw=2)
ax.axhline(0, color='black', lw=0.8, ls='--')
ax.fill_between(range(200), mean_diffs[:200], 0,
                where=(mean_diffs[:200] < 0), alpha=0.25,
                color='purple', label='negative increments')
ax.fill_between(range(200), mean_diffs[:200], 0,
                where=(mean_diffs[:200] > 0), alpha=0.25,
                color='red', label='positive increments')
ax.set_xlabel('observations $n$')
ax.set_ylabel(r'$\mathbb{E}[d_{n+1} - d_n]$')
ax.legend(fontsize=8)

ax = axes[1]
ax.plot(cum_sum[:200], color='darkorange', lw=2)
ax.axhline(0, color='black', lw=0.8, ls='--')
ax.set_xlabel('observations $n$')
ax.set_ylabel(r'cumulative average change in $d_n$')

plt.tight_layout()
```

(continues on next page)

(continued from previous page)

```
plt.show()

frac_decrease = np.mean(mean_diffs < 0)
print(f"Fraction of steps with average decrement: {frac_decrease:.2%}")
```

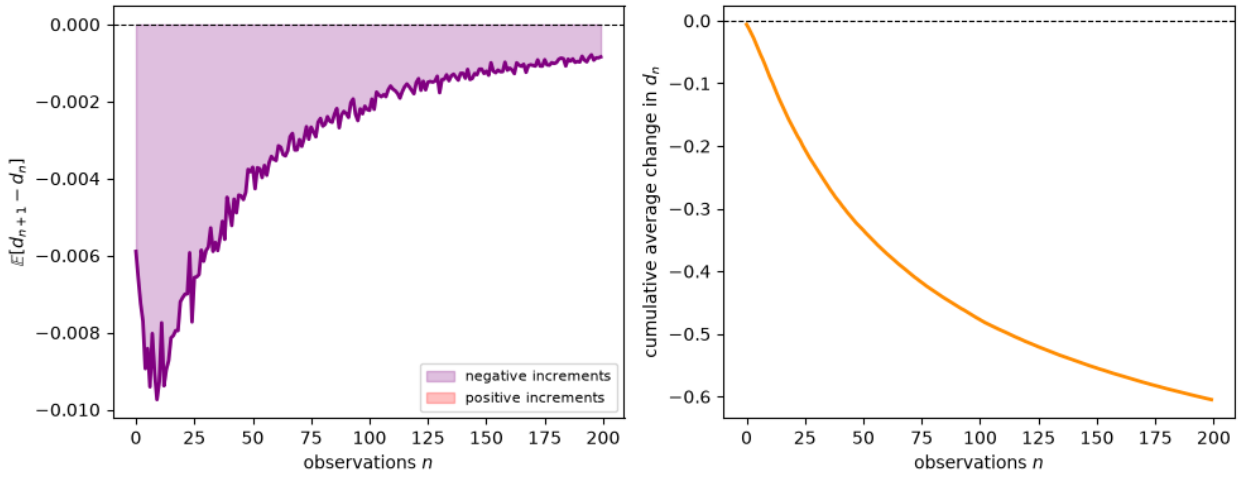


Fig. 34.3: An illustration of the supermartingale property. The plots show average increments of d_n and their cumulative sum across many simulated paths.

```
Fraction of steps with average decrement: 100.00%
```

The average increment is negative at most steps, and the cumulative drift is downward.

This is only an illustration, not a proof, because it uses unconditional averages rather than the full conditional expectation in the theorem.

34.6 Failure of merging: mutual singularity

What happens when the hypothesis $Q \ll P$ fails?

The singular case is the cleanest counterexample.

34.6.1 Point-mass priors

Suppose both agents hold degenerate (point-mass) priors:

- Agent P: certain that $p = p_P = 0.30$.
- Agent Q: certain that $p = p_Q = 0.75$.

Since P charges only sequences whose empirical frequency converges to 0.30, and Q charges only sequences whose empirical frequency converges to 0.75, the two measures are mutually singular: $P \perp Q$.

The conditional distributions do not update, because both agents are already certain of their model.

For the theorem's object, namely the conditional law of the entire future path,

$$\|P(\cdot \mid \mathcal{F}_n) - Q(\cdot \mid \mathcal{F}_n)\|_{\mathrm{TV}} = \|P - Q\|_{\mathrm{TV}} = 1 \quad \text{for all } n.$$

This equality holds because the infinite-product Bernoulli measures with distinct success probabilities are singular.

If we look only one step ahead, the predictive distance is $|p_P - p_Q| = 0.45$.

That is smaller than one, but it is not the quantity that appears in Blackwell–Dubins.

```
p_P = 0.30
p_Q = 0.75
n_steps = 500

tv_singular_full = np.ones(n_steps + 1)
tv_singular_1step = np.full(n_steps + 1, np.abs(p_P - p_Q))

sim_abs_cont = run_simulation(
    p_Q, 1.0, 8.0, 8.0, 1.0, n_steps, seed=1
)

fig, ax = plt.subplots(figsize=(8, 4))
ax.plot(np.arange(n_steps + 1), tv_singular_full,
        color='firebrick', lw=2,
        label=r'singular: full-path $d_n = 1$')
ax.plot(np.arange(n_steps + 1), tv_singular_1step,
        color='gray', lw=2, ls=':',
        label=r'one-step gap $= |p_P - p_Q|$')
ax.plot(np.arange(n_steps + 1),
        sim_abs_cont['tv_beta'],
        color='steelblue', lw=2,
        label=(r'$\mathrm{Beta}(1,8)$ vs'
               r' $\mathrm{Beta}(8,1)$'))
ax.set_xlabel('observations $n$')
ax.set_ylabel(r'$d_n$')
ax.legend(fontsize=8)
ax.set_ylim(0, 1.05)

plt.tight_layout()
plt.show()
```

The contrast is sharp.

With mutually absolutely continuous priors, d_n decays to zero.

With singular point-mass priors, the full future-path distance stays at one forever.

More data does not reconcile the agents, because each rules out paths the other assigns positive probability.

34.7 Kakutani's theorem: when does merging hold?

A natural question is: for which product measures does the Blackwell–Dubins hypothesis $Q \ll P$ hold?

For infinite product measures, the answer is given by a classical result of Kakutani [1948].

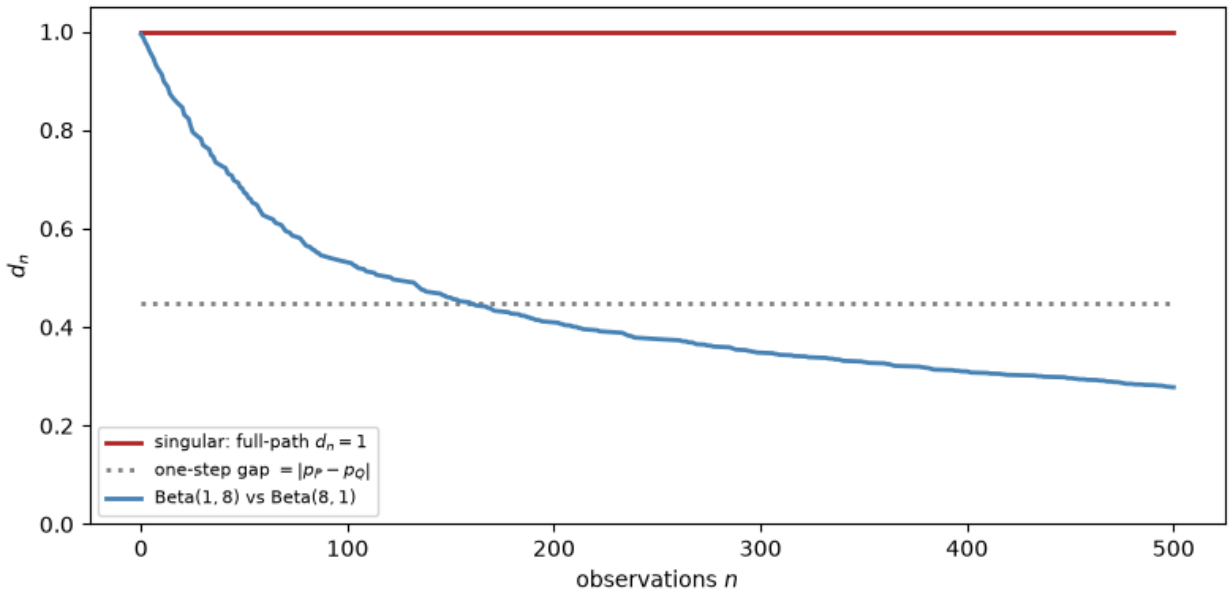


Fig. 34.4: Failure of merging under singular priors. The full future-path distance stays at one, while the one-step predictive gap stays at $|p_P - p_Q|$.

34.7.1 Hellinger affinities

Definition 34.7.1 (Hellinger Affinity)

For probability measures P_n and Q_n on $(S, \mathcal{S})$ with common dominating measure λ, the **Hellinger affinity** is

$$\rho_n = \int_S \sqrt{\frac{dP_n}{d\lambda} \cdot \frac{dQ_n}{d\lambda}}\, d\lambda \;\in\; [0,1].$$

$\rho_n = 1$ if and only if $P_n = Q_n$; $\rho_n = 0$ if and only if $P_n \perp Q_n$.

For two specific one-dimensional families:

- Gaussian: $P_n = \mathcal{N}(\mu_n, 1)$ vs $Q_n = \mathcal{N}(0,1)$:

$$\rho_n^{\text{Gauss}} = \exp\left(-\frac{\mu_n^2}{8}\right).$$

- Bernoulli: $P_n = \text{Bernoulli}(p)$ vs $Q_n = \text{Bernoulli}(q)$:

$$\rho_n^{\text{Bern}} = \sqrt{pq} + \sqrt{(1-p)(1-q)}.$$

34.7.2 Kakutani's dichotomy

Theorem 34.7.1 (Kakutani (1948))

Let $P = \bigotimes_{n=1}^{\infty} P_n$ and $Q = \bigotimes_{n=1}^{\infty} Q_n$ be infinite product measures whose factors are pairwise equivalent: $P_n \sim Q_n$ for every n.

Then either $P \sim Q$ or $P \perp Q$; there is no intermediate case.

Specifically,

$$P \sim Q \quad \Longleftrightarrow \quad \prod_{n=1}^{\infty} \rho_n > 0 \quad \Longleftrightarrow \quad \sum_{n=1}^{\infty} (1 - \rho_n) < \infty.$$

If $\prod_{n=1}^{\infty} \rho_n = 0$, then $P \perp Q$.

Proof idea. A standard proof studies the likelihood-ratio martingale $Z_N = \prod_{n=1}^{N} (dP_n / dQ_n)$ together with the identity $\mathbb{E}_Q[\sqrt{Z_N}] = \prod_{n=1}^{N} \rho_n$.

The product staying positive corresponds to equivalence, while the product collapsing to zero corresponds to singularity.

□

34.7.3 Implication for merging

For IID-type sequences, Kakutani's theorem gives the following picture:

Scenario	$\sum_n (1 - \rho_n)$	Conclusion	Merging?
$P_n = Q_n$ for all n	0	$P = Q$	Trivially yes
$P_n \neq Q_n$ with $\sum_n (1 - \rho_n) < \infty$	Finite	$P \sim Q$	Yes; Blackwell–Dubins applies
$P_n = P \neq Q = Q_n$ fixed, $n \geq 1$	∞	$P \perp Q$	No

The IID case with different fixed marginals is the standard no-merging example.

If two agents assign permanently different distributions to each observation, they end up in disjoint probability worlds.

34.7.4 A Gaussian product-measure example

We illustrate Kakutani's dichotomy with Gaussian product measures.

Take $Q = \mathcal{N}(0, 1)^{\otimes \mathbb{N}}$ as the reference measure and $P = \bigotimes_n \mathcal{N}(\mu_n, 1)$ as the alternative.

Three choices of μ_n:

1. $\mu_n = \mu > 0$ constant ($\sum (1 - \rho_n) = \infty$) $\Rightarrow P \perp Q$.
2. $\mu_n = c/\sqrt{n}$ ($\sum (1 - \rho_n) \approx \sum c^2/(8n) = \infty$) $\Rightarrow P \perp Q$.
3. $\mu_n = c/n$ ($\sum (1 - \rho_n) \approx \sum c^2/(8n^2) < \infty$) $\Rightarrow P \sim Q$.

```
N_max  = 2000
ns     = np.arange(1, N_max + 1)
c      = 2.0
```

(continues on next page)

(continued from previous page)

```
N_plot = 400
rng    = np.random.default_rng(0)

cases = [
    (r'$\mu_n = c$ (constant)', np.full(N_max, c)),
    (r'$\mu_n = c/\sqrt{n}$', c / np.sqrt(ns)),
    (r'$\mu_n = c/n$', c / ns),
]
```

With constant drift, $\log Z_N$ drifts to $-\infty$ under Q, so $Z_N \to 0$ Q-a.s. and $P \perp Q$.

```
label, μ_seq = cases[0]
x = rng.standard_normal(N_plot)
log_Z_inc = μ_seq[:N_plot] * x - μ_seq[:N_plot]**2 / 2
log_Z = np.concatenate([[0], np.cumsum(log_Z_inc)])

fig, ax = plt.subplots(figsize=(8, 3))
ax.plot(np.arange(N_plot + 1), log_Z,
        color='darkorange', lw=2, label=label)
ax.axhline(0, color='black', lw=0.8, ls=':')
ax.set_xlabel('horizon $N$')
ax.set_ylabel(r'$\log Z_N$ under $Q$')
ax.legend(fontsize=8)
plt.tight_layout()
plt.show()
```

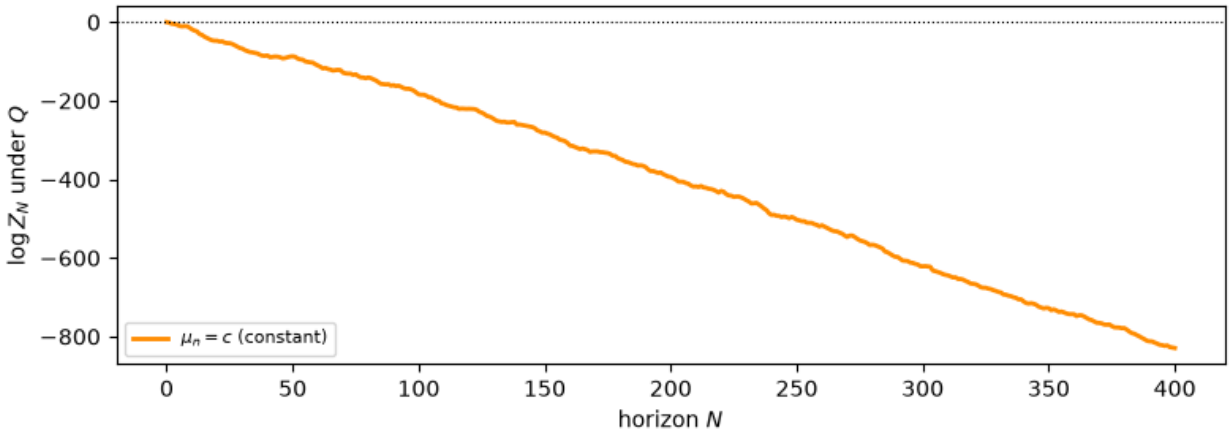


Fig. 34.5: Constant drift $\mu_n = c$: the likelihood ratio collapses ($P \perp Q$).

The $\mu_n = c/\sqrt{n}$ case shows the same qualitative picture: despite the drift vanishing, it does so too slowly.

```
label, μ_seq = cases[1]
x = rng.standard_normal(N_plot)
log_Z_inc = μ_seq[:N_plot] * x - μ_seq[:N_plot]**2 / 2
log_Z = np.concatenate([[0], np.cumsum(log_Z_inc)])

fig, ax = plt.subplots(figsize=(8, 3))
ax.plot(np.arange(N_plot + 1), log_Z,
        color='purple', lw=2, label=label)
```

(continues on next page)

(continued from previous page)

```
ax.axhline(0, color='black', lw=0.8, ls=':')
ax.set_xlabel('horizon $N$')
ax.set_ylabel(r'$\log Z_N$ under $Q$')
ax.legend(fontsize=8)
plt.tight_layout()
plt.show()
```

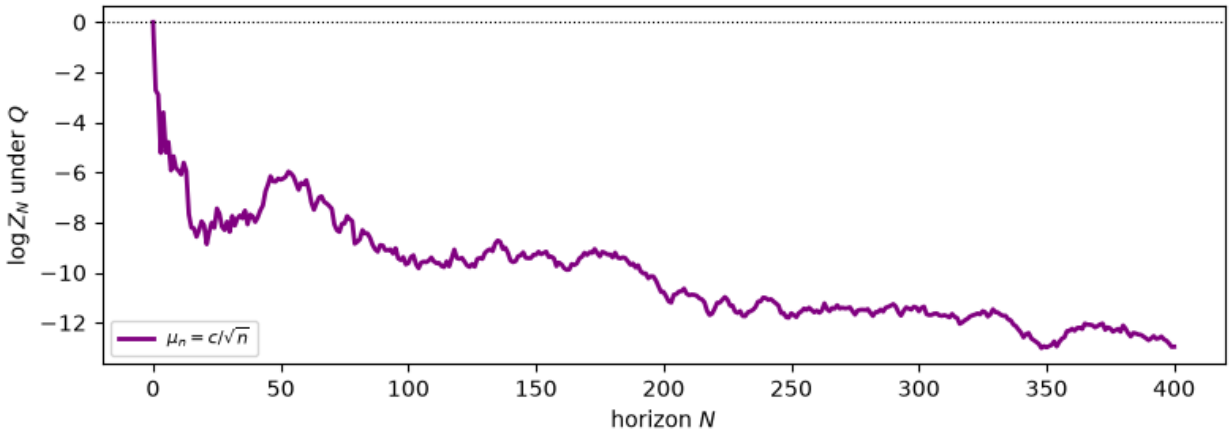


Fig. 34.6: Drift $\mu_n = c/\sqrt{n}$: still singular ($P \perp Q$).

Only with $\mu_n = c/n$ does $\sum(1-\rho_n) < \infty$ hold, so the likelihood ratio remains nondegenerate and $P \sim Q$.

Blackwell–Dubins applies only in this case.

```
label, μ_seq = cases[2]
x = rng.standard_normal(N_plot)
log_Z_inc = μ_seq[:N_plot] * x - μ_seq[:N_plot]**2 / 2
log_Z = np.concatenate([[0], np.cumsum(log_Z_inc)])

fig, ax = plt.subplots(figsize=(8, 3))
ax.plot(np.arange(N_plot + 1), log_Z,
        color='steelblue', lw=2, label=label)
ax.axhline(0, color='black', lw=0.8, ls=':')
ax.set_xlabel('horizon $N$')
ax.set_ylabel(r'$\log Z_N$ under $Q$')
ax.legend(fontsize=8)
plt.tight_layout()
plt.show()
```

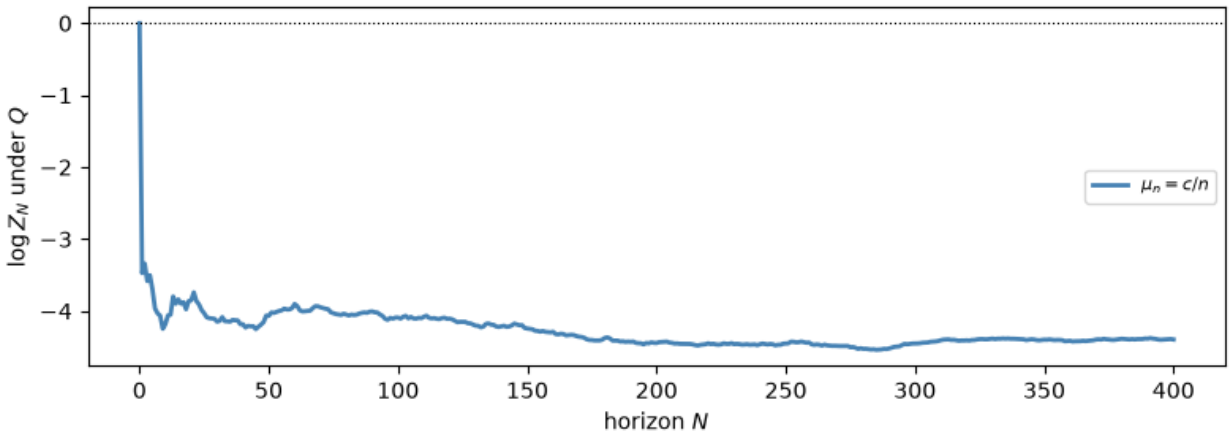


Fig. 34.7: Drift $\mu_n = c/n$: the likelihood ratio stabilises ($P \sim Q$).

34.8 Extension to continuous time

The same logic extends to continuous time.

34.8.1 Girsanov's theorem and the likelihood-ratio process

On the canonical Wiener space with Q the Wiener measure (standard Brownian motion W), suppose agent P believes the process has an additional drift $\theta = \{\theta_s\}_{s \geq 0}$:

$$W_t = \widetilde{W}_t + \int_0^t \theta_s \, ds,$$

where $\widetilde{W}$ is a P-Brownian motion.

The Girsanov–Cameron–Martin theorem [Girsanov, 1960] gives the likelihood-ratio process as the stochastic exponential

$$Z_t = \exp\left(\int_0^t \theta_s \, dW_s - \frac{1}{2} \int_0^t \theta_s^2 \, ds \right).$$

Z_t is always a non-negative Q-local martingale; it is a true martingale if and only if $\mathbb{E}_Q[Z_t] = 1$ for all t.

Novikov's condition [Novikov, 1972], $\mathbb{E}_Q\left[\exp\left(\frac{1}{2}\int_0^T \theta_s^2 \, ds\right)\right] < \infty$ for all T, is sufficient.

34.8.2 The dichotomy at infinity

A key subtlety on $[0, +\infty)$ is that local absolute continuity does *not* imply global absolute continuity on $\mathcal{F}_\infty$.

Remark 34.8.1 (Infinite-Horizon Subtlety)

Suppose Z_t is a true Q-martingale for every finite horizon and let $Z_t \to Z_\infty$ Q-a.s.

If $\{Z_t\}$ is uniformly integrable on $[0, \infty)$, then $P \ll Q$ on $\mathcal{F}_\infty$ with $dP/dQ = Z_\infty$.

For the Blackwell–Dubins conclusion we need $Q \ll P$ on $\mathcal{F}_\infty$ (the reverse direction).

In many standard settings, including deterministic drifts satisfying the energy condition below, the measures are in fact *equivalent* ($P \sim Q$) on $\mathcal{F}_\infty$, so both directions hold.

If uniform integrability fails, then global absolute continuity on $\mathcal{F}_\infty$ can fail.

In many standard examples, including a non-zero constant drift, the measures are in fact singular on $\mathcal{F}_\infty$.

A convenient sufficient condition in deterministic-drift examples is the **energy condition**

$$\int_0^\infty \theta_s^2 \, ds < \infty \quad Q\text{-a.s.}$$

Informally, this says the total amount of information separating the two measures over the infinite horizon is finite.

Under the energy condition, $P \sim Q$ on $\mathcal{F}_\infty$, so Blackwell–Dubins applies and merging holds under both measures.

When θ is a non-zero constant, the condition fails, the measures are singular on $\mathcal{F}_\infty$, and merging does not occur.

Whenever $Q \ll P$ on $\mathcal{F}_\infty$ is established, the proof of the continuous-time Blackwell–Dubins result is identical to the discrete-time proof.

$\{d_t, \mathcal{F}_t\}$ is a non-negative Q-supermartingale in $[0, 1]$, so $d_t \to d_\infty$ Q-a.s.

The L^1 bound $\mathbb{E}_Q[d_t] = \frac{1}{2}\mathbb{E}_P[|Z_t - Z_\infty|] \to 0$ forces $d_\infty = 0$.

34.9 Applications

34.9.1 Bayesian learning

The most direct application is Bayesian inference.

Suppose data $(x_1, x_2, ...)$ are drawn from the true measure Q^*.

An agent holds a prior π over a family $\{Q_\theta : \theta \in \Theta\}$, inducing a marginal $P = \int Q_\theta \, \pi(d\theta)$.

If $Q^* \ll P$ (i.e., the agent's marginal model dominates the truth), then Blackwell–Dubins gives

$$\left\| P(\cdot \,|\, x_1, \ldots, x_n) - Q^*(\cdot \,|\, x_1, \ldots, x_n) \right\|_{\text{TV}} \to 0 \quad Q^*\text{-a.s.}$$

This is a strong form of Bayesian consistency: the agent's predictions merge with the truth under the true measure.

A prior assigning positive mass to a neighbourhood of the true parameter typically guarantees *local* absolute continuity $Q_n^* \ll P_n$ for every finite horizon n, but not the global condition $Q^* \ll P$ on $\mathcal{F}_\infty$ that Blackwell–Dubins requires.

For example, in the Beta–Bernoulli model with a non-atomic prior π, the mixture $P = \int \text{Bernoulli}(p)^\infty \, \pi(dp)$ satisfies $Q_n^* \ll P_n$ for every n, yet $Q^* \not\ll P$ globally because the set $\{\lim k_n/n = p^*\}$ has Q^*-measure one but P-measure zero (different Bernoulli product measures are mutually singular).

Global absolute continuity does hold under additional structure, for instance when the parameter space is finite or the model is sufficiently regular to admit a Doob-consistency argument.

Diaconis and Freedman [1986] study the consistency of Bayes estimates and show, among other results, that the interplay between local and global absolute continuity plays a central role in ensuring posterior convergence.

When $P \perp Q^*$, there are events of probability one under Q^* that have probability zero under P, so the agent's beliefs remain fundamentally misspecified.

34.9.2 Rational expectations and heterogeneous priors

In macroeconomics, rational-expectations models typically impose a common prior.

Blackwell–Dubins gives a dynamic justification for weaker initial agreement.

If two agents start with equivalent priors and observe the same history, their conditional forecasts eventually agree on every event.

Aumann [1976]'s agreement theorem strengthens this: agents with a common prior cannot "agree to disagree" on posterior probabilities.

Blackwell–Dubins complements Aumann by showing that equivalent priors are enough for eventual agreement.

34.9.3 Ergodic Markov chains

For a Markov chain with transition kernel Π and two initial distributions μ and ν, the n-step distributions are $\mu\Pi^n$ and $\nu\Pi^n$.

If Π is ergodic with unique stationary distribution π, both converge to π, so

$$\|\mu\Pi^n - \nu\Pi^n\|_{\mathrm{TV}} \le \|\mu\Pi^n - \pi\|_{\mathrm{TV}} + \|\nu\Pi^n - \pi\|_{\mathrm{TV}} \to 0.$$

This is a special form of merging that does *not* require absolute continuity, because ergodicity already forces both distributions to the same limit.

Blackwell–Dubins is the right analogue for non-ergodic or non-Markovian environments, where no single invariant distribution need exist.

34.10 The rate of merging

Blackwell–Dubins is qualitative.

It tells us that $d_n \to 0$, but not how fast.

The bound

$$\mathbb{E}_Q[d_n] = \tfrac{1}{2}\,\mathbb{E}_P[|Z_n - Z_\infty|]$$

shows that the rate of merging is controlled by the $L^1(P)$ convergence rate of the likelihood-ratio martingale.

In regular parametric examples, one often sees $n^{-1/2}$-type behavior.

The next figure checks that heuristic in the Beta–Bernoulli model.

```
N_paths_rate = 200
n_steps_rate = 800

tv_rate = np.empty((N_paths_rate, n_steps_rate + 1))
for i in range(N_paths_rate):
    s = run_simulation(p_true, a1, b1, a2, b2, n_steps_rate, seed=100 + i)
    tv_rate[i] = s['tv_beta']

ns_rate  = np.arange(1, n_steps_rate + 1)
mean_tv  = tv_rate[:, 1:].mean(axis=0)     # mean d_n, n = 1, ..., n_steps_rate

# Fit a reference line d_n ~ C / sqrt(n) using the later part of the sample
```

(continues on next page)

(continued from previous page)

```
fit_start = 200
log_ns  = np.log(ns_rate[fit_start:])
log_tv  = np.log(mean_tv[fit_start:] + 1e-12)
coeffs  = np.polyfit(log_ns, log_tv, 1)
slope   = coeffs[0]

# Reference curve C/sqrt(n)
C_ref   = np.exp(coeffs[1])
ref_curve = C_ref / np.sqrt(ns_rate)

fig, ax = plt.subplots(figsize=(8, 4))
ax.loglog(ns_rate, mean_tv,  color='steelblue', lw=2,
          label=r'$\mathbb{E}_Q[d_n]$ (Monte Carlo)')
ax.loglog(ns_rate, ref_curve, color='firebrick', lw=2, ls='--',
          label=(rf'Reference $C/\sqrt{{n}}$'
                 rf'  (fitted slope $\approx {slope:.2f}$)'))
ax.set_xlabel('sample size $n$')
ax.set_ylabel(r'$\mathbb{E}_Q[d_n]$')
ax.legend()
plt.tight_layout()
plt.show()

print(f"Fitted log-log slope: {slope:.3f}  (predicted: -0.50)")
```

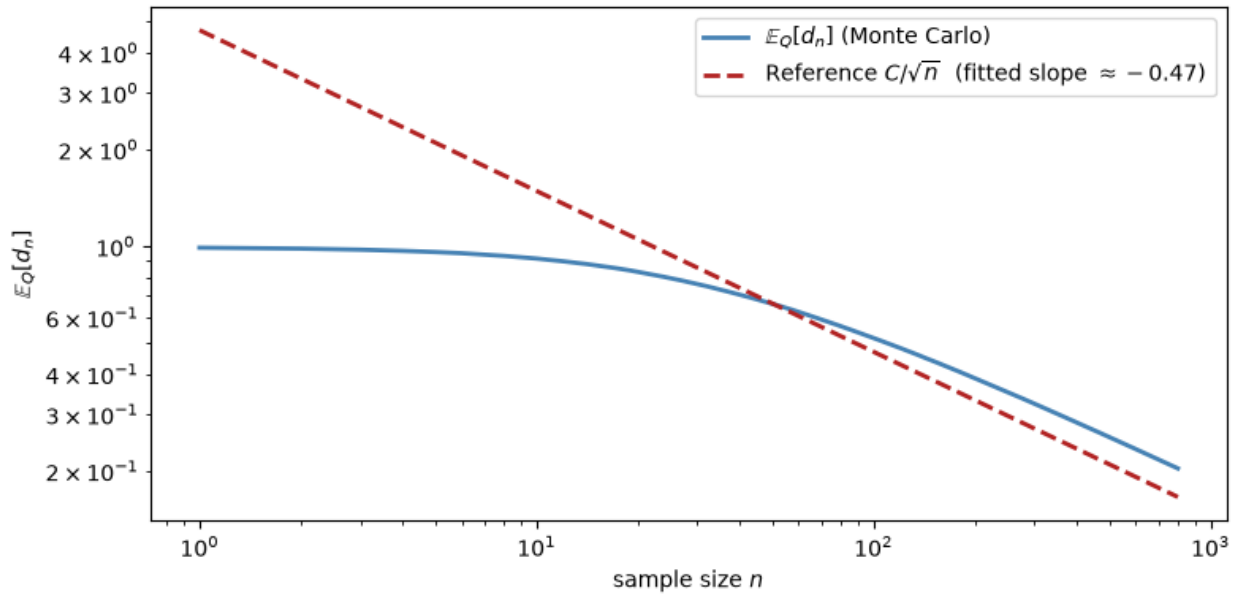


Fig. 34.8: A log-log plot of the average merging distance in the Beta–Bernoulli model. The fitted slope is close to $-1/2$, which is consistent with square-root decay in this experiment.

```
Fitted log-log slope: -0.469  (predicted: -0.50)
```

Fitting the later part of the sample gives a slope close to -0.5.

That is consistent with $n^{-1/2}$ scaling in this simulation.

34.11 Summary and extensions

The logical flow underlying the Blackwell–Dubins theorem is:

$$Q \ll P \implies Z = \frac{dQ}{dP} \in L^1(P) \implies Z_n = \mathbb{E}_P[Z \mid \mathcal{F}_n] \xrightarrow{L^1(P)} Z_\infty \implies d_n \xrightarrow{Q\text{-a.s.}} 0.$$

Takeaways:

1. One-sided absolute continuity $Q \ll P$ gives merging Q-almost surely. For merging under *both* measures, one needs mutual absolute continuity $P \sim Q$.
2. The likelihood-ratio martingale $Z_n = \mathbb{E}_P[Z|\mathcal{F}_n]$ and its $L^1(P)$ convergence drive the result.
3. More data can only reduce (in expectation) the difficulty of distinguishing two hypotheses.
4. For infinite product measures, Kakutani's theorem gives a sharp equivalence-versus-singularity dichotomy: either $P \sim Q$ (when $\sum_n (1 - \rho_n) < \infty$) or $P \perp Q$ (when the sum diverges), with no intermediate case.
5. When $P \sim Q$, Blackwell–Dubins applies and merging occurs under both measures; when $P \perp Q$, disagreement persists forever.

34.11.1 Applications in economics

Some influential applications and extensions are:

- [Kalai and Lehrer, 1993]: repeated-game learning drives play toward Nash behavior when priors are absolutely continuous with respect to the truth.
- [Kalai and Lehrer, 1993]: subjective and objective equilibria coincide asymptotically under the same condition.
- [Kalai and Lehrer, 1994]: weak and strong notions of merging are introduced for environments where full total-variation convergence is too strong.
- [Kalai *et al.*, 1999]: merging is linked to calibrated forecasting.
- [Jackson *et al.*, 1999]: de Finetti-style representations are connected to Bayesian learning and posterior convergence.
- [Jackson and Kalai, 1999]: social learning erodes reputational effects that rely on persistent disagreement across cohorts.
- [Sandroni, 1998]: near-absolute-continuity conditions are shown to suffice for Nash-type convergence in repeated games.
- [Miller and Sanchirico, 1999]: gives an alternative proof and an economic interpretation of persistent disagreement in terms of mutually favorable bets.
- [Lehrer and Smorodinsky, 1996]: studies broader compatibility notions beyond Blackwell–Dubins absolute continuity.
- [Lehrer and Smorodinsky, 1996]: surveys merging and learning in repeated strategic environments.
- [Nyarko, 1994]: relates Bayesian learning under absolute continuity to convergence toward correlated equilibrium.
- [Pomatto *et al.*, 2014]: extends the theorem to finitely additive probabilities and connects merging to test manipulability.
- [Acemoglu *et al.*, 2016]: shows how disagreement can persist when agents are uncertain about the signal structure itself.

34.11.2 A companion result from probability

[Diaconis and Freedman, 1986] study the consistency of Bayes estimates, proving equivalences involving posterior convergence and providing counterexamples that highlight the role of the prior.

Their work is in the same intellectual tradition as Blackwell–Dubins and is routinely co-cited with the merging theorem in the economics learning literature.

CHAPTER

THIRTYFIVE

SURVIVAL AND LONG-RUN DYNAMICS UNDER RECURSIVE PREFERENCES

Contents

- *Survival and Long-Run Dynamics under Recursive Preferences*
  - *Overview*
  - *Environment*
  - *Planner's problem*
  - *Survival conditions*
  - *Wealth dynamics decomposition*
  - *Survival regions*
  - *Three survival channels*
  - *Varying the IES*
  - *Asymptotic results*
  - *The separable case*
  - *Asset pricing implications*
  - *Optimistic and pessimistic distortions*
  - *Long-run consumption distribution*
  - *Summary*

35.1 Overview

This lecture studies the theory of long-run survival in Borovička [2020].

The classical **market selection hypothesis** says that agents with less accurate beliefs are driven out of the market in the long run.

This result was established rigorously by Sandroni [2000] and Blume and Easley [2006] for economies with separable CRRA preferences.

Borovicka shows that the conclusion can fail under Epstein-Zin recursive preferences.

With recursive preferences, agents with distorted beliefs can survive and can even dominate.

The key mechanism is that recursive preferences separate risk aversion from the intertemporal elasticity of substitution.

That separation creates three channels that matter for survival:

1. The *risk premium channel* rewards the more optimistic agent for holding more of the risky asset.
2. The *speculative volatility channel* penalizes aggressive positions through log-return volatility.
3. The *saving channel* changes consumption and saving decisions when the IES differs from one.

Under separable preferences, only the first two channels remain.

Under recursive preferences, the saving channel can overturn market selection.

Note

The paper builds on the continuous-time recursive utility formulation of Duffie and Epstein [1992], using the planner's problem approach of Dumas *et al.* [2000].

Important foundations for the market selection hypothesis were laid by De Long *et al.* [1991] and Blume and Easley [1992].

We start with some imports.

```
import numpy as np
import matplotlib.pyplot as plt
```

35.2 Environment

The economy contains two infinitely lived agents, indexed by $n \in \{1, 2\}$.

The agents have identical recursive preferences but different beliefs about aggregate endowment growth.

We write Borovička's belief distortions u^n as ω^n.

35.2.1 Aggregate endowment

Under the true probability measure P, aggregate endowment satisfies

$$d \log Y_t = \mu_Y dt + \sigma_Y dW_t, \quad Y_0 > 0 \tag{35.1}$$

where W is a standard Brownian motion, μ_Y is the drift, and $\sigma_Y > 0$ is the volatility.

35.2.2 Heterogeneous beliefs

Agent n believes that the drift is $\mu_Y + \omega^n \sigma_Y$ instead of μ_Y.

The parameter ω^n measures optimism when $\omega^n > 0$ and pessimism when $\omega^n < 0$.

Agent n's subjective probability measure Q^n is defined by the Radon–Nikodym derivative

$$M_t^n = \frac{dQ^n}{dP}\bigg|_t = \exp\left(-\frac{1}{2}|\omega^n|^2 t + \omega^n W_t\right) \tag{35.2}$$

Under Q^n, the process $W_t^n = W_t - \omega^n t$ is a Brownian motion, and agent n perceives

$$d \log Y_t = (\mu_Y + \omega^n \sigma_Y) dt + \sigma_Y dW_t^n.$$

An agent with $\omega^n > 0$ is optimistic about endowment growth, while an agent with $\omega^n < 0$ is pessimistic.

35.2.3 Recursive preferences

Both agents have Epstein-Zin recursive preferences.

We use $\gamma > 0$ for relative risk aversion, $\rho > 0$ for the inverse of the IES, and $\beta > 0$ for the time-preference rate.

The Duffie-Epstein-Zin felicity function is

$$F(C, \nu) = \beta \frac{C^{1-\gamma}}{1-\gamma} \cdot \left(\frac{(1-\gamma) - (1-\rho)\nu/\beta}{\rho - \gamma} \right)^{(\gamma-\rho)/(1-\rho)} \tag{35.3}$$

where ν is the endogenous discount rate.

Note

In discrete time, Epstein-Zin preferences aggregate current consumption with a certainty equivalent of future utility via a CES aggregator (see Doubts or Variability?).

In continuous time there is no "next-period V_{t+1}," so Duffie and Epstein [1992] recast the recursion as a felicity function $F(C, \nu)$ that depends on the agent's own continuation-value rate ν.

The two formulations encode the same separation of risk aversion γ from the inverse IES ρ.

When $\gamma = \rho$, preferences reduce to the standard separable CRRA case.

35.3 Planner's problem

Following Dumas *et al.* [2000], we study equilibrium allocations through a social planner's problem.

The planner chooses consumption shares z^1 and $z^2 = 1 - z^1$ and discount-rate processes ν^n for the two agents.

35.3.1 Modified discount factors

It is convenient to absorb belief distortions into the modified discount factors $\tilde{\lambda}^n = \lambda^n M^n$, where M^n is the Radon-Nikodym derivative (35.2).

These processes satisfy

$$d \log \tilde{\lambda}_t^n = -\left(\nu_t^n + \frac{1}{2} (\omega^n)^2 \right) dt + \omega^n dW_t. \tag{35.4}$$

Exercise 35.3.1

Derive (35.4).

Hint: Use $\log \tilde{\lambda}^n = \log \lambda^n + \log M^n$. The Pareto weight λ^n evolves as $d \log \lambda_t^n = -\nu_t^n \, dt$, and $\log M_t^n$ is given by (35.2).

Solution

From the definition $\tilde{\lambda}^n = \lambda^n M^n$, we have

$$\log \tilde{\lambda}_t^n = \log \lambda_t^n + \log M_t^n.$$

The Pareto weight satisfies $d \log \lambda_t^n = -\nu_t^n \, dt$.

From (35.2), $\log M_t^n = -\frac{1}{2}|\omega^n|^2 t + \omega^n W_t$, so

$$d \log M_t^n = -\tfrac{1}{2}(\omega^n)^2 \, dt + \omega^n \, dW_t.$$

Adding the two:

$$d \log \tilde{\lambda}_t^n = -\nu_t^n \, dt - \tfrac{1}{2}(\omega^n)^2 \, dt + \omega^n \, dW_t = -\left(\nu_t^n + \tfrac{1}{2}(\omega^n)^2\right) dt + \omega^n \, dW_t.$$

35.3.2 State variable: Pareto share

The key state variable is the Pareto share of agent 1:

$$\upsilon = \frac{\tilde{\lambda}^1}{\tilde{\lambda}^1 + \tilde{\lambda}^2} \in (0, 1) \tag{35.5}$$

It captures the relative weight of agent 1 in the planner's allocation.

Define the log-odds ratio $\vartheta = \log(\upsilon/(1-\upsilon))$.

Its dynamics are

$$d\vartheta_t = \underbrace{\left[\nu_t^2 + \frac{1}{2}(\omega^2)^2 - \nu_t^1 - \frac{1}{2}(\omega^1)^2\right]}_{m_\vartheta(\upsilon_t)} dt + (\omega^1 - \omega^2) dW_t \tag{35.6}$$

The drift $m_\vartheta(\upsilon)$ determines the long-run behavior of the Pareto share.

Exercise 35.3.2

Derive (35.6) from (35.4) and the definition $\vartheta = \log(\upsilon/(1-\upsilon))$.

Hint: First show that $\vartheta = \log \tilde{\lambda}^1 - \log \tilde{\lambda}^2$, then subtract the two SDEs.

Solution

Since $\upsilon = \tilde{\lambda}^1/(\tilde{\lambda}^1 + \tilde{\lambda}^2)$, we have $1 - \upsilon = \tilde{\lambda}^2/(\tilde{\lambda}^1 + \tilde{\lambda}^2)$, so

$$\vartheta = \log \frac{\upsilon}{1-\upsilon} = \log \tilde{\lambda}^1 - \log \tilde{\lambda}^2.$$

From (35.4), the two log-discount-factor SDEs are

$$d \log \tilde{\lambda}_t^1 = -\left(\nu_t^1 + \tfrac{1}{2}(\omega^1)^2\right) dt + \omega^1 dW_t,$$

$$d \log \tilde{\lambda}_t^2 = -\left(\nu_t^2 + \tfrac{1}{2}(\omega^2)^2\right) dt + \omega^2 dW_t.$$

Subtracting the second from the first:

$$d\vartheta_t = \left[\nu_t^2 + \tfrac{1}{2}(\omega^2)^2 - \nu_t^1 - \tfrac{1}{2}(\omega^1)^2\right] dt + (\omega^1 - \omega^2) dW_t.$$

35.3.3 HJB equation

Homotheticity reduces the planner's problem to a nonlinear ODE in the single state variable υ.

Because each agent's utility is homogeneous of degree $1-\gamma$ in consumption, the planner's value function factors as $J(\upsilon, Y) = \tilde{J}(\upsilon) \cdot Y^{1-\gamma}/(1-\gamma)$, eliminating Y as a state variable.

From discrete to continuous time

In discrete time, a planner maximizes a weighted sum of agents' utilities by choosing allocations at each date.

The Bellman equation is

$$\tilde{J}(\upsilon) = \max_{z^1, z^2} \left\{ \upsilon\, u(z^1) + (1-\upsilon)\, u(z^2) + \beta\, \mathbb{E}\left[\tilde{J}(\upsilon')\right] \right\}.$$

In continuous time, the period length shrinks to dt.

The "flow payoff" over $[t, t+dt)$ becomes $\left[\upsilon F(z^1, \nu^1) + (1-\upsilon) F(z^2, \nu^2)\right] dt$, where F is the Duffie-Epstein-Zin felicity (35.3).

The expected change in the value function over dt is captured by the **infinitesimal generator** $\mathcal{L}$.

For a diffusion $d\upsilon = m\, dt + s\, dW$, Itô's lemma gives

$$\mathcal{L}\tilde{J}(\upsilon) = m(\upsilon)\, \tilde{J}'(\upsilon) + \tfrac{1}{2} s(\upsilon)^2\, \tilde{J}''(\upsilon),$$

where m and s are the drift and diffusion of the Pareto share.

This is the continuous-time analogue of $\beta\, \mathbb{E}[\tilde{J}(\upsilon')] - \tilde{J}(\upsilon)$: it measures how the value function drifts and fluctuates as υ evolves.

Setting flow payoff plus expected capital gain equal to zero gives the schematic HJB equation:

$$0 = \sup_{(z^1, z^2, \nu^1, \nu^2)} \left\{ \upsilon F(z^1, \nu^1) + (1-\upsilon) F(z^2, \nu^2) + \mathcal{L}\tilde{J}(\upsilon) \right\} \tag{35.7}$$

subject to $z^1 + z^2 \leq 1$.

Exact reduced ODE

Proposition 2.3 of Borovička [2020] gives the exact HJB equation after substituting the homogeneity reduction $J(\tilde{\lambda}, Y) = (\tilde{\lambda}^1 + \tilde{\lambda}^2) Y^{1-\gamma} \tilde{J}(\upsilon)$ and the dynamics of υ and Y:

$$\begin{aligned} 0 = \sup_{(z^1, z^2, \nu^1, \nu^2)} & \ \upsilon\, F(z^1, \nu^1) + (1-\upsilon)\, F(z^2, \nu^2) \\ & + \left[-\upsilon\nu^1 - (1-\upsilon)\nu^2 + (\upsilon\omega^1 + (1-\upsilon)\omega^2)(1-\gamma)\sigma_Y + (1-\gamma)\mu_Y + \tfrac{1}{2}(1-\gamma)^2\sigma_Y^2\right] \tilde{J}(\upsilon) \\ & + \upsilon(1-\upsilon)\left[\nu^2 - \nu^1 + (\omega^1 - \omega^2)(1-\gamma)\sigma_Y\right] \tilde{J}'(\upsilon) \\ & + \tfrac{1}{2}\upsilon^2(1-\upsilon)^2(\omega^1 - \omega^2)^2\, \tilde{J}''(\upsilon) \end{aligned} \tag{35.8}$$

subject to $z^1 + z^2 \leq 1$.

The first line is the flow payoff from the two agents' felicity functions.

The second line multiplies $\tilde{J}(\upsilon)$ by a term that combines the agents' discount rates, belief-weighted endowment drift, and a variance correction – these arise from absorbing the $Y^{1-\gamma}$ factor via Itô's lemma.

The third line multiplies $\tilde{J}'(\upsilon)$ by the drift of the Pareto share, which depends on the difference in discount rates and the belief-weighted response to endowment risk.

The fourth line multiplies $\tilde{J}''(\upsilon)$ by the squared diffusion of the Pareto share.

The boundary conditions are $\tilde{J}(0) = \tilde{V}^2$ and $\tilde{J}(1) = \tilde{V}^1$, where $\tilde{V}^n$ is the continuation value in the homogeneous economy populated by agent n alone.

This is the continuous-time counterpart of the discrete-time planner's problem in Blume and Easley [2006] (see also *Heterogeneous Beliefs and Financial Markets*).

35.4 Survival conditions

The central result characterizes survival by the boundary behavior of $m_\vartheta(\upsilon)$.

Proposition 35.4.1

Define the following repelling conditions (i) and (ii) and their attracting counterparts (i') and (ii'):

$$\text{(i)} \lim_{\upsilon \searrow 0} m_\vartheta(\upsilon) > 0, \qquad \text{(i')} \lim_{\upsilon \searrow 0} m_\vartheta(\upsilon) < 0$$

$$\text{(ii)} \lim_{\upsilon \nearrow 1} m_\vartheta(\upsilon) < 0, \qquad \text{(ii')} \lim_{\upsilon \nearrow 1} m_\vartheta(\upsilon) > 0$$

Then:

(a) If (i) and (ii) hold, both agents survive under P.

(b) If (i) and (ii') hold, agent 1 dominates in the long run under P.

© If (i') and (ii) hold, agent 2 dominates in the long run under P.

(d) If (i') and (ii') hold, each agent dominates with strictly positive probability.

The proof uses the Feller classification of boundary behavior for diffusions, as in Karlin and Taylor [1981].

Condition (i) says that when agent 1 is close to extinction, there is a force pushing her share back up.

Condition (ii) says that when agent 1 is close to absorbing the whole economy, there is a force pushing her share back down.

When both forces are present, the Pareto share is recurrent and both agents survive.

35.5 Wealth dynamics decomposition

We now rewrite the survival conditions from *Proposition 35.4.1* in terms of equilibrium wealth dynamics.

Agent 1 survives near extinction if and only if her wealth grows faster than agent 2's when she is negligibly small.

When $\upsilon \searrow 0$, prices are set entirely by agent 2, as if the economy were homogeneous.

Agent 1 is a price-taker in agent 2's economy.

Let $m_A^n(\upsilon)$ denote the expected log growth rate of agent n's wealth.

The difference decomposes into two channels:

$$\lim_{\upsilon \searrow 0}[m_A^1(\upsilon) - m_A^2(\upsilon)] = \underbrace{\lim_{\upsilon \searrow 0}[m_R^1(\upsilon) - m_R^2(\upsilon)]}_{\text{portfolio returns}} + \underbrace{\lim_{\upsilon \searrow 0}[(y^2(\upsilon))^{-1} - (y^1(\upsilon))^{-1}]}_{\text{consumption-wealth ratios}} \tag{35.9}$$

The first term measures how much faster agent 1's portfolio grows.

The second measures how much less agent 1 consumes out of wealth – a lower consumption-wealth ratio means more saving and faster wealth accumulation.

When this total difference is positive, agent 1 survives; when negative, she shrinks toward extinction.

Exercise 35.5.1

Derive (35.9).

Let A^n denote agent n's wealth and C^n her consumption.

The budget constraint is $dA^n = A^n dR^n - C^n dt$, where dR^n is the return on agent n's portfolio.

Define the consumption-wealth ratio $c^n = C^n/A^n = (y^n)^{-1}$.

Show that $d\log A^n = m_R^n\,dt - (y^n)^{-1}dt + \ldots$, so the difference in expected log wealth growth is $m_A^1 - m_A^2 = (m_R^1 - m_R^2) + [(y^2)^{-1} - (y^1)^{-1}]$.

Solution

Dividing the budget constraint by A^n:

$$\frac{dA^n}{A^n} = dR^n - (y^n)^{-1}dt.$$

By Itô's lemma, $d\log A^n = \frac{dA^n}{A^n} - \frac{1}{2}\left(\frac{dA^n}{A^n}\right)^2$.

Write $dR^n = m_R^n\,dt + \sigma_R^n\,dW$ (the portfolio return under P).

Then

$$d\log A^n = \left(m_R^n - (y^n)^{-1} - \tfrac{1}{2}(\sigma_R^n)^2\right)dt + \sigma_R^n\,dW.$$

Taking the difference for agents 1 and 2:

$$m_A^1 - m_A^2 = (m_R^1 - m_R^2) + \left[(y^2)^{-1} - (y^1)^{-1}\right] - \tfrac{1}{2}\left[(\sigma_R^1)^2 - (\sigma_R^2)^2\right].$$

The volatility terms $\frac{1}{2}[(\sigma_R^1)^2 - (\sigma_R^2)^2]$ are absorbed into $m_R^1 - m_R^2$ when we define m_R^n as the expected log portfolio return (i.e., the drift of $\log R^n$ rather than the arithmetic return), giving (35.9).

35.5.1 Portfolio returns

At the boundary $v \searrow 0$, the difference in expected log portfolio returns is

$$\lim_{v \searrow 0} [m_R^1 - m_R^2] = \underbrace{\frac{\omega^1 - \omega^2}{\gamma \sigma_Y}}_{\text{difference in risky shares}} \cdot \underbrace{(\gamma \sigma_Y^2 - \omega^2 \sigma_Y)}_{\text{risk premium}} - \underbrace{\frac{\omega^1 - \omega^2}{\gamma} \left(\sigma_Y + \frac{\omega^1 - \omega^2}{2\gamma} \right)}_{\text{volatility term}} \tag{35.10}$$

An optimistic agent ($\omega^1 > \omega^2$) overweights the risky asset by $(\omega^1 - \omega^2)/(\gamma \sigma_Y)$ relative to agent 2 and earns the equity risk premium on that extra exposure.

The subtracted *volatility penalty* reflects the cost of holding a more extreme portfolio: higher variance of log returns drags down expected log wealth growth.

This term depends on risk aversion γ but not on the IES, because portfolio choice is determined by risk aversion alone.

Exercise 35.5.2

Derive (35.10).

At the boundary $v \searrow 0$, agent n's optimal risky-asset share is $\pi^n = 1 + (\omega^n - \omega^2)/(\gamma \sigma_Y)$ (see (35.14)).

Let $\bar{\mu}_R = \mu_Y + \gamma \sigma_Y^2 - \omega^2 \sigma_Y$ denote the expected return on the risky asset under P, and r the risk-free rate.

The continuously rebalanced portfolio has expected log return $m_R^n = r + \pi^n (\bar{\mu}_R - r) - \frac{1}{2}(\pi^n)^2 \sigma_Y^2$.

Compute $m_R^1 - m_R^2$ and simplify.

Solution

Using $m_R^n = r + \pi^n (\bar{\mu}_R - r) - \frac{1}{2}(\pi^n)^2 \sigma_Y^2$, the difference is

$$m_R^1 - m_R^2 = (\pi^1 - \pi^2)(\bar{\mu}_R - r) - \tfrac{1}{2}[(\pi^1)^2 - (\pi^2)^2]\sigma_Y^2.$$

The difference in risky shares is $\pi^1 - \pi^2 = (\omega^1 - \omega^2)/(\gamma \sigma_Y)$.

The arithmetic equity premium is $\bar{\mu}_R - r = \gamma \sigma_Y^2 - \omega^2 \sigma_Y$, so:

$$(\pi^1 - \pi^2)(\bar{\mu}_R - r) = \frac{\omega^1 - \omega^2}{\gamma \sigma_Y} \cdot (\gamma \sigma_Y^2 - \omega^2 \sigma_Y).$$

For the volatility term, write $(\pi^1)^2 - (\pi^2)^2 = (\pi^1 - \pi^2)(\pi^1 + \pi^2)$ and note $\pi^1 + \pi^2 = 2 + (\omega^1 + \omega^2 - 2\omega^2)/(\gamma \sigma_Y)$.

After simplification:

$$\tfrac{1}{2}[(\pi^1)^2 - (\pi^2)^2]\sigma_Y^2 = \frac{\omega^1 - \omega^2}{\gamma} \left(\sigma_Y + \frac{\omega^1 - \omega^2}{2\gamma} \right).$$

Combining the two pieces gives (35.10).

35.5.2 Consumption-wealth ratios

The difference in consumption-wealth ratios at the boundary is

$$\lim_{v \searrow 0}[(y^2)^{-1} - (y^1)^{-1}] = \frac{1-\rho}{\rho}\left[(\omega^1 - \omega^2)\sigma_Y + \frac{(\omega^1 - \omega^2)^2}{2\gamma}\right] \tag{35.11}$$

The term in brackets is the difference in *subjective* expected portfolio returns – what agent 1 believes she earns relative to agent 2.

The factor $(1-\rho)/\rho$ translates this perceived return advantage into a saving response.

- When IES > 1 ($\rho < 1$), the factor is positive: a higher perceived return makes the agent save more, because the substitution effect dominates the income effect.
- When IES < 1 ($\rho > 1$), the factor is negative: the income effect dominates and the agent saves less, working against survival.
- When IES $= 1$ ($\rho = 1$), the two effects cancel and the saving channel vanishes entirely.

This is the channel through which recursive preferences alter survival outcomes by separating γ from ρ.

Exercise 35.5.3

Derive (35.11).

In the homogeneous economy populated by agent 2, the consumption-wealth ratio is $(y(0))^{-1} = \beta - (1-\rho)\mu_V^2$, where μ_V^2 is agent 2's expected log return on wealth.

Agent 1, as a negligible price-taker, has consumption-wealth ratio $(y^1)^{-1} = \beta - (1-\rho)\mu_V^1$, where μ_V^1 is her own expected log return.

Use $(y^2)^{-1} - (y^1)^{-1} = (1-\rho)(\mu_V^1 - \mu_V^2)$ and express $\mu_V^1 - \mu_V^2$ in terms of agent 1's *subjective* expected excess return.

Hint: Under agent 1's beliefs, her portfolio earns an extra $(\omega^1 - \omega^2)\sigma_Y + (\omega^1 - \omega^2)^2/(2\gamma)$ in expected log returns relative to agent 2's portfolio.

Solution

The consumption-wealth ratio for agent n satisfies $(y^n)^{-1} = \beta - (1-\rho)\mu_V^n$, where μ_V^n is the expected log return on agent n's wealth under her own subjective measure.

Taking the difference:

$$(y^2)^{-1} - (y^1)^{-1} = (1-\rho)(\mu_V^1 - \mu_V^2).$$

Agent 1's subjective expected log portfolio return exceeds agent 2's by the amount she believes she gains from tilting toward the risky asset.

Her extra risky share is $\pi^1 - 1 = (\omega^1 - \omega^2)/(\gamma\sigma_Y)$, and under her subjective measure Q^1 the risky asset's expected excess log return is $(\gamma\sigma_Y^2 + (\omega^1 - \omega^2)\sigma_Y - \omega^2\sigma_Y) - r - \frac{1}{2}\sigma_Y^2$.

After simplification, the subjective expected log return difference is

$$\mu_V^1 - \mu_V^2 = (\omega^1 - \omega^2)\sigma_Y + \frac{(\omega^1 - \omega^2)^2}{2\gamma}.$$

Substituting and dividing through by ρ (from the relationship between $(y^n)^{-1}$ and β):

$$(y^2)^{-1} - (y^1)^{-1} = \frac{1-\rho}{\rho}\left[(\omega^1 - \omega^2)\sigma_Y + \frac{(\omega^1 - \omega^2)^2}{2\gamma}\right].$$

35.5.3 Two comparative statics

Survival depends on γ, ρ, and the signal-to-noise ratios ω^1/σ_Y and ω^2/σ_Y, not on ω^1, ω^2, and σ_Y separately.

The survival conditions do not depend on β or μ_Y, which affect the level of consumption and prices but not relative wealth dynamics at the boundary.

```
def portfolio_return_diff(ω_1, ω_2, γ, σ_y):
    """
    Difference in expected log portfolio returns at the boundary.
    """
    Δω = ω_1 - ω_2
    risky_share_diff = Δω / (γ * σ_y)
    risk_premium = γ * σ_y**2 - ω_2 * σ_y
    volatility_term = (Δω / γ) * (σ_y + 0.5 * Δω / γ)
    return risky_share_diff * risk_premium - volatility_term


def saving_channel(ω_1, ω_2, γ, ρ, σ_y):
    """
    Difference in consumption-wealth ratios at the boundary.
    """
    Δω = ω_1 - ω_2
    subjective_return_diff = Δω * σ_y + Δω**2 / (2 * γ)
    return (1 - ρ) / ρ * subjective_return_diff


def boundary_drift(ω_1, ω_2, γ, ρ, σ_y):
    """
    Boundary drift m_0 when agent 1 becomes negligible.

    Positive drift means agent 1 survives (repelling boundary).
    """
    return γ * (
        portfolio_return_diff(ω_1, ω_2, γ, σ_y)
        + saving_channel(ω_1, ω_2, γ, ρ, σ_y)
    )
```

35.6 Survival regions

A central contribution of Borovička [2020] is the characterization of survival regions in the (γ, ρ) plane.

Under separable preferences, $\gamma = \rho$, the agent with more accurate beliefs always dominates.

Under recursive preferences, all four outcomes in *Proposition 35.4.1* can occur.

Figure 2 in the paper studies the case where agent 2 has correct beliefs, so $\omega^2 = 0$.

The next cell follows that figure.

```
def compute_survival_boundary(ω_1, ω_2, σ_y, γ_grid, boundary="lower"):
    """
    Compute the curve in (γ, ρ) space where the boundary drift is zero.

    For boundary='lower', agent 1 is the small agent.
    For boundary='upper', agent 2 is the small agent.
    """
    ρ_boundary = []

    if boundary == "lower":
        small_agent = (ω_1, ω_2)
    else:
        small_agent = (ω_2, ω_1)

    ω_small, ω_large = small_agent

    for γ in γ_grid:
        pr = portfolio_return_diff(ω_small, ω_large, γ, σ_y)
        Δω = ω_small - ω_large
        subj_ret = Δω * σ_y + Δω**2 / (2 * γ)

        if abs(subj_ret) < 1e-14:
            ρ_boundary.append(np.nan)
            continue

        denom = subj_ret - pr
        if abs(denom) < 1e-14:
            ρ_boundary.append(np.nan)
        else:
            ρ_boundary.append(subj_ret / denom)

    return np.asarray(ρ_boundary)


def compute_limit_boundary(γ_grid, boundary="lower"):
    """
    Boundary curves for the limit |ω_1| / σ_y -> ∞.

    This is equivalent to the constant-endowment case discussed in the paper.
    """
    if boundary == "lower":
        return γ_grid / (1 + γ_grid)

    ρ = np.full_like(γ_grid, np.nan, dtype=float)
    mask = γ_grid < 1
    ρ[mask] = γ_grid[mask] / (1 - γ_grid[mask])
    return ρ
```

Each panel plots two curves in the (γ, ρ) plane for a different value of agent 1's belief distortion ω^1 (agent 2 has correct beliefs, $\omega^2 = 0$).

- The dashed curve (blue) is where the boundary drift at $\upsilon = 0$ equals zero – condition (i) in *Proposition 35.4.1*.
- The solid curve (red) is where the boundary drift at $\upsilon = 1$ equals zero – condition (ii).
- The shaded region between the two curves is where both agents survive.
- The dotted diagonal $\gamma = \rho$ is the separable CRRA case, along which the agent with more accurate beliefs always

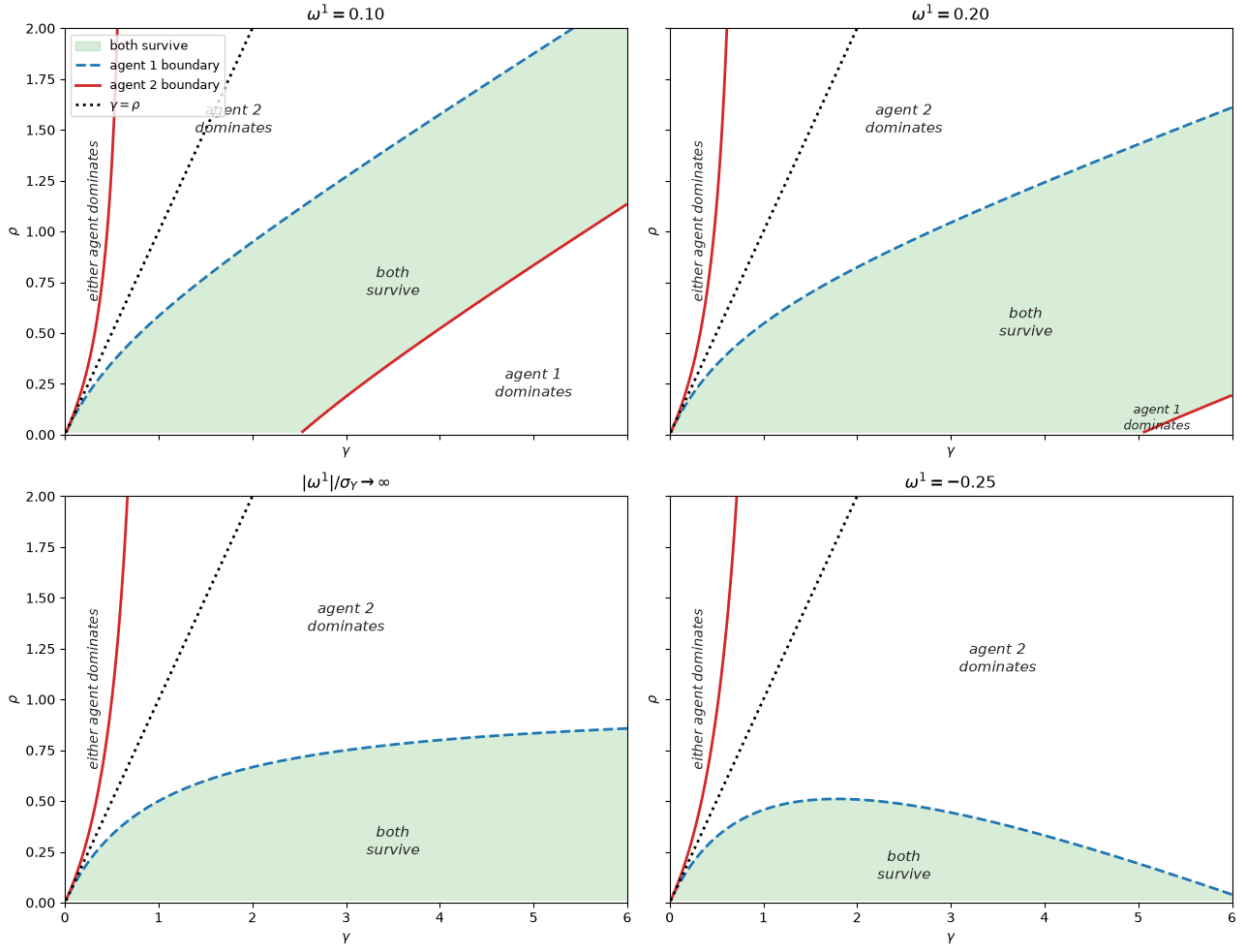


Fig. 35.1: Survival regions corresponding to Figure 2 in Borovicka (2020)

dominates.

Moderate optimism ($\omega^1 = 0.10$) produces a wide coexistence region that extends across most of the γ range.

Stronger optimism ($\omega^1 = 0.20$) narrows the region: the agent 2 boundary shifts out of the plotted range for moderate and large γ, shrinking the set of (γ, ρ) pairs where both agents coexist.

In the limit $|\omega^1|/\sigma_Y \to \infty$ (bottom-left), the boundaries simplify to closed-form expressions.

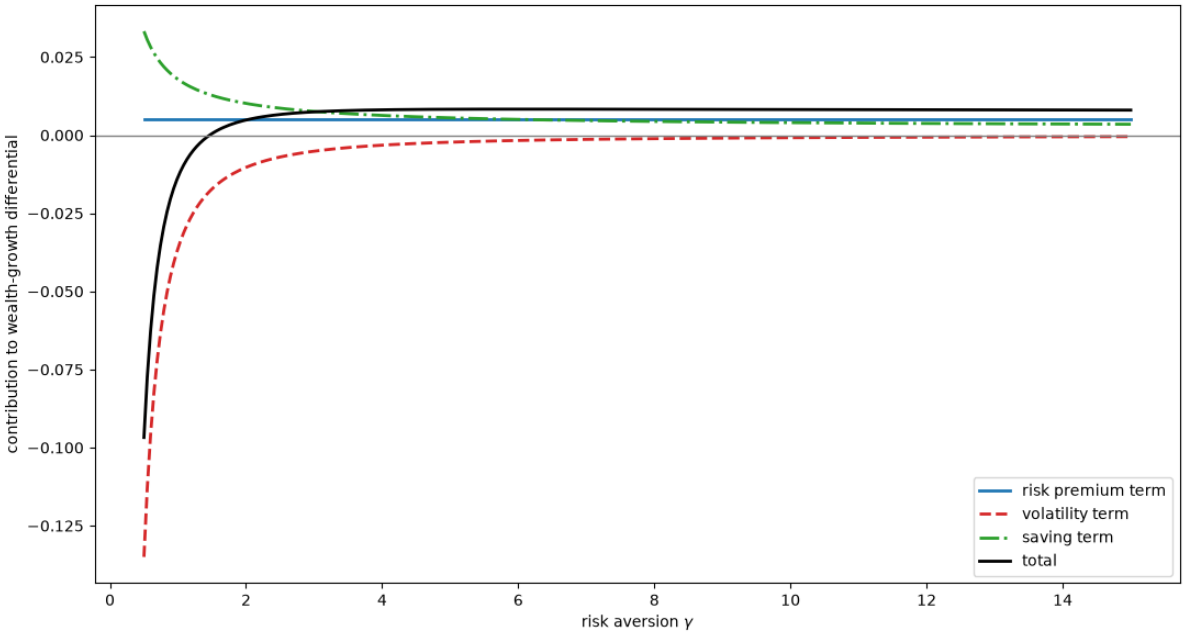
The coexistence region narrows but extends to large γ values below the agent 2 boundary curve.

Pessimistic distortions ($\omega^1 = -0.25$, bottom-right) can also survive, but only in a much narrower part of the parameter space.

35.7 Three survival channels

The decomposition above can be visualized directly.

```
def decompose_survival(ω_1, ω_2, γ_grid, ρ, σ_y):
    """
    Decompose the wealth-growth differential in proposition 3.4.
    """
    Δω = ω_1 - ω_2
    risk_premium_term = Δω * (γ_grid * σ_y - ω_2) / γ_grid
    volatility_term = -(Δω / γ_grid) * (σ_y + 0.5 * Δω / γ_grid)
    saving_term = (1 - ρ) / ρ * (Δω * σ_y + Δω**2 / (2 * γ_grid))
    total = risk_premium_term + volatility_term + saving_term
    return risk_premium_term, volatility_term, saving_term, total


ω_1 = 0.25
ω_2 = 0.0
ρ = 0.67
σ_y = 0.02
γ_grid = np.linspace(0.5, 15.0, 300)

risk_term, vol_term, save_term, total = decompose_survival(
    ω_1, ω_2, γ_grid, ρ, σ_y
)

fig, ax = plt.subplots(figsize=(11, 6))
ax.plot(γ_grid, risk_term, color="C0", lw=2, label="risk premium term")
ax.plot(γ_grid, vol_term, "--", color="C3", lw=2, label="volatility term")
ax.plot(γ_grid, save_term, "-.", color="C2", lw=2, label="saving term")
ax.plot(γ_grid, total, color="black", lw=2, label="total")
ax.axhline(0, color="gray", lw=1)
ax.set_xlabel(r"risk aversion $\gamma$")
ax.set_ylabel("contribution to wealth-growth differential")
ax.legend()
plt.tight_layout()
plt.show()
```



This figure decomposes the boundary drift at $\upsilon = 0$ into three terms for an optimistic agent ($\omega^1 = 0.25$, $\omega^2 = 0$) with IES $= 1/\rho \approx 1.49$ and $\sigma_Y = 0.02$.

- The risk premium term (blue) is positive throughout because the optimistic agent overweights the risky asset and earns the equity premium.
- The volatility term (red dashed) is negative and large at low γ, reflecting the cost of holding a volatile portfolio.
- The saving term (green dash-dot) is positive when IES > 1 because the optimistic agent perceives a high return on wealth and saves more aggressively.
- The total (black) crosses zero at the critical γ below which the volatility penalty dominates and the agent cannot survive.

35.8 Varying the IES

The sign of the saving term is pinned down by the IES.

```
fig, axes = plt.subplots(1, 3, figsize=(16, 4.5), sharey=True)

ω_1 = 0.25
ω_2 = 0.0
σ_y = 0.02
γ_grid = np.linspace(0.5, 25.0, 300)

ies_values = [0.5, 1.0, 1.5]

for idx, ies in enumerate(ies_values):
    ρ = 1.0 / ies
    risk_term, vol_term, save_term, total = decompose_survival(
        ω_1, ω_2, γ_grid, ρ, σ_y
    )

    ax = axes[idx]
```

(continues on next page)

(continued from previous page)

```
    ax.plot(y_grid, risk_term, color="C0", lw=2, label="risk premium")
    ax.plot(y_grid, vol_term, "--", color="C3", lw=2, label="volatility")
    ax.plot(y_grid, save_term, "-.", color="C2", lw=2, label="saving")
    ax.plot(y_grid, total, color="black", lw=2, label="total")
    ax.axhline(0, color="gray", lw=1)
    ax.set_title(f"IES = {ies:.1f}", fontsize=12)
    ax.set_xlabel(r"risk aversion $\gamma$")
    ax.set_ylabel("contribution")

axes[0].legend(fontsize=9)
plt.tight_layout()
plt.show()
```

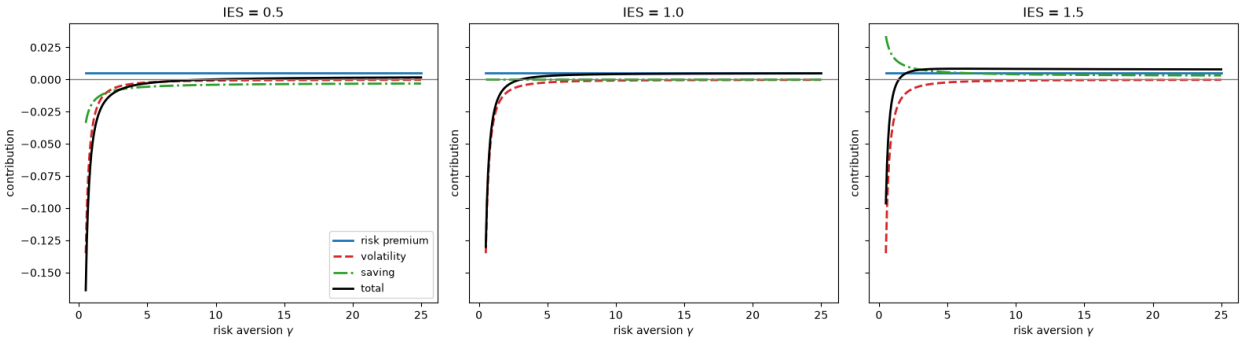


Fig. 35.2: Boundary decomposition for different IES values

Each panel shows the same three-term decomposition as the previous figure, but now for three different values of the IES ($\omega^1 = 0.25$, $\omega^2 = 0$, $\sigma_Y = 0.02$).

- Left panel (IES $= 0.5$): the saving term is negative, so the optimistic agent actually saves less, working against survival.
- Center panel (IES $= 1.0$): the saving term vanishes entirely, so only the portfolio return and volatility channels remain.
  - This eliminates the saving channel but does not by itself reproduce the full separable CRRA benchmark, which requires $\gamma = \rho$ (i.e., IES $= 1/\gamma$), not merely $\rho = 1$.
- Right panel (IES $= 1.5$): the saving term is positive and shifts the total drift upward, expanding the range of γ values for which the optimistic agent survives.

35.9 Asymptotic results

Borovicka derives several useful asymptotic results.

1. As $\gamma \searrow 0$, each agent dominates with strictly positive probability.
2. As $\gamma \nearrow \infty$, the relatively more optimistic agent dominates.
3. As $\rho \searrow 0$, the relatively more optimistic agent always survives.
   - The relatively more pessimistic agent can also survive when risk aversion is sufficiently low.
4. As $\rho \nearrow \infty$, a nondegenerate long-run equilibrium cannot exist.

The next figure illustrates the first result by plotting both boundary drifts as γ becomes small.

```
ω_1 = 0.25
ω_2 = 0.0
ρ = 0.67
σ_y = 0.02
γ_grid = np.linspace(0.05, 5.0, 300)

drift_at_0 = np.array([boundary_drift(ω_1, ω_2, γ, ρ, σ_y) for γ in γ_grid])
drift_at_1 = np.array([-boundary_drift(ω_2, ω_1, γ, ρ, σ_y) for γ in γ_grid])

fig, ax = plt.subplots(figsize=(10, 5))
ax.plot(γ_grid, drift_at_0, color="C0", lw=2, label=r"$\upsilon \to 0$")
ax.plot(γ_grid, drift_at_1, "--", color="C3", lw=2, label=r"$\upsilon \to 1$")
ax.axhline(0, color="gray", lw=1)
ax.set_xlabel(r"risk aversion $\gamma$")
ax.set_ylabel("boundary drift")
ax.legend()
plt.tight_layout()
plt.show()
```

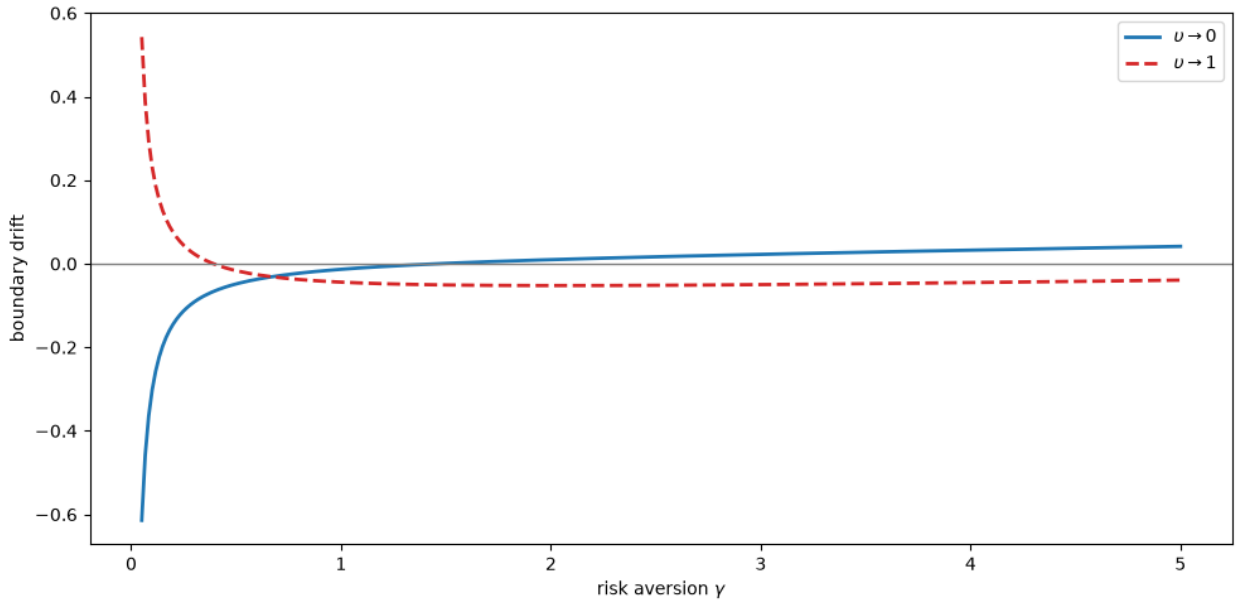


Fig. 35.3: Boundary drifts for small risk aversion

This figure plots the two boundary drifts as a function of γ ($\omega^1 = 0.25$, $\omega^2 = 0$, IES ≈ 1.49).

- The solid blue curve is the drift m_ϑ at $\upsilon \to 0$ (agent 1 near extinction); coexistence requires this to be positive (condition (i)).
- The dashed red curve is the drift m_ϑ at $\upsilon \to 1$ (agent 2 near extinction); coexistence requires this to be negative (condition (ii)).

The figure illustrates asymptotic result 1.

For small γ, the blue curve is negative and the red curve is positive.

Both boundaries are attracting: near $\upsilon = 0$ the negative drift pulls υ toward 0, and near $\upsilon = 1$ the positive drift pushes υ toward 1.

This is outcome (d) in *Proposition 35.4.1*: neither boundary is repelling, so whichever agent happens to get ahead early will dominate, with each agent having strictly positive probability of dominance depending on the realized Brownian path.

As γ increases past roughly 1, the blue curve crosses zero and becomes positive while the red curve stays negative.

Now both boundaries are repelling and we enter the coexistence region – outcome (a).

35.10 The separable case

When $\gamma = \rho$, the model collapses to the separable CRRA benchmark.

In that case, the log-odds process becomes

$$d\vartheta_t = \frac{1}{2}\left[(\omega^2)^2 - (\omega^1)^2\right] dt + (\omega^1 - \omega^2) dW_t.$$

The drift is constant and depends only on the relative entropy of the two belief distortions.

The agent with the smaller $|\omega^n|$ dominates under P.

If the two agents have equal magnitudes of belief distortions, neither becomes extinct almost surely, but no nondegenerate stationary wealth distribution exists.

```
def simulate_crra_pareto(ω_1, ω_2, T, dt, n_paths, seed=42):
    """
    Simulate Pareto-share dynamics in the separable benchmark.
    """
    rng = np.random.default_rng(seed)
    n_steps = int(T / dt)
    t_grid = np.linspace(0, T, n_steps + 1)

    drift = 0.5 * (ω_2**2 - ω_1**2)
    volatility = ω_1 - ω_2

    θ = np.zeros((n_paths, n_steps + 1))
    dW = rng.normal(0.0, np.sqrt(dt), size=(n_paths, n_steps))

    for t in range(n_steps):
        θ[:, t + 1] = θ[:, t] + drift * dt + volatility * dW[:, t]

    υ_paths = 1.0 / (1.0 + np.exp(-θ))
    return t_grid, υ_paths


ω_1 = 0.10
ω_2 = 0.0
t_grid, υ_paths = simulate_crra_pareto(ω_1, ω_2, T=200, dt=0.01, n_paths=50)

fig, ax = plt.subplots(figsize=(11, 5))

for i in range(20):
    ax.plot(t_grid, υ_paths[i], color="C0", alpha=0.25, lw=1)

ax.axhline(0.5, color="gray", linestyle=":", lw=1)
ax.set_xlabel("time")
```

(continues on next page)

(continued from previous page)

```
ax.set_ylabel(r"Pareto share $\upsilon_t$")
ax.set_ylim(0, 1)
plt.tight_layout()
plt.show()
```

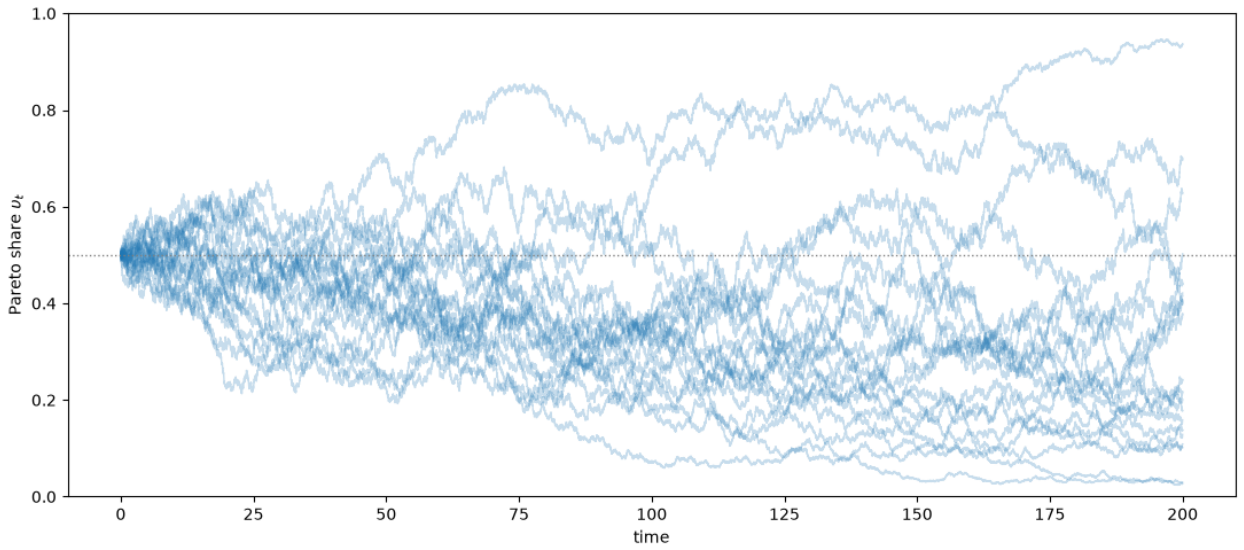


Fig. 35.4: Pareto-share paths in the separable benchmark

This figure simulates 20 sample paths of the Pareto share υ_t under separable CRRA preferences ($\gamma = \rho$) with $\omega^1 = 0.10$ and $\omega^2 = 0$.

Agent 2 has correct beliefs, so the log-odds drift is negative and all paths trend toward $\upsilon = 0$.

Agent 1 is driven to extinction – the classical market-selection result of Blume and Easley [2006].

35.11 Asset pricing implications

As one agent becomes negligible, current prices converge to those of the homogeneous economy populated by the large agent.

When agent 2 is the large agent, Proposition 5.1 in Borovička [2020] implies

$$\lim_{\upsilon \searrow 0} r(\upsilon) = \beta + \rho \left(\mu_Y + \omega^2 \sigma_Y + \frac{1}{2}(1-\gamma)\sigma_Y^2 \right) - \frac{1}{2}\gamma\sigma_Y^2 \tag{35.12}$$

and

$$\lim_{\upsilon \searrow 0} y(\upsilon) = \left[\beta - (1-\rho) \left(\mu_Y + \omega^2 \sigma_Y + \frac{1}{2}(1-\gamma)\sigma_Y^2 \right) \right]^{-1}. \tag{35.13}$$

The aggregate wealth dynamics also converge to those of the homogeneous economy:

$$\lim_{\upsilon \searrow 0} m_A(\upsilon) = \mu_Y, \qquad \lim_{\upsilon \searrow 0} \sigma_A(\upsilon) = \sigma_Y.$$

Proposition 5.3 then gives the negligible agent's own consumption-saving and portfolio choices.

Her consumption-wealth ratio converges to

$$\lim_{v \searrow 0} (y^1(v))^{-1} = (y(0))^{-1} - \frac{1-\rho}{\rho}\left[(\omega^1 - \omega^2)\sigma_Y + \frac{(\omega^1 - \omega^2)^2}{2\gamma}\right].$$

The small agent's risky-asset share converges to

$$\lim_{v \searrow 0} \pi^1(v) = 1 + \frac{\omega^1 - \omega^2}{\gamma \sigma_Y}. \tag{35.14}$$

Hence optimism implies leverage, while sufficiently strong pessimism implies shorting.

```
ω_2 = 0.0
σ_y = 0.02
ω_grid = np.linspace(-0.5, 1.0, 300)

fig, ax = plt.subplots(figsize=(10, 5))

for γ in [2, 5, 10, 20]:
    π_1 = 1 + (ω_grid - ω_2) / (γ * σ_y)
    ax.plot(ω_grid, π_1, lw=2, label=rf"$\gamma = {γ}$")

ax.axhline(1.0, color="gray", linestyle=":", lw=1)
ax.axhline(0.0, color="gray", linestyle=":", lw=1)
ax.axvline(0.0, color="gray", linestyle=":", lw=1)
ax.set_xlabel(r"belief distortion $\omega^1$")
ax.set_ylabel(r"risky share $\pi^1$")
ax.legend()
plt.tight_layout()
plt.show()
```

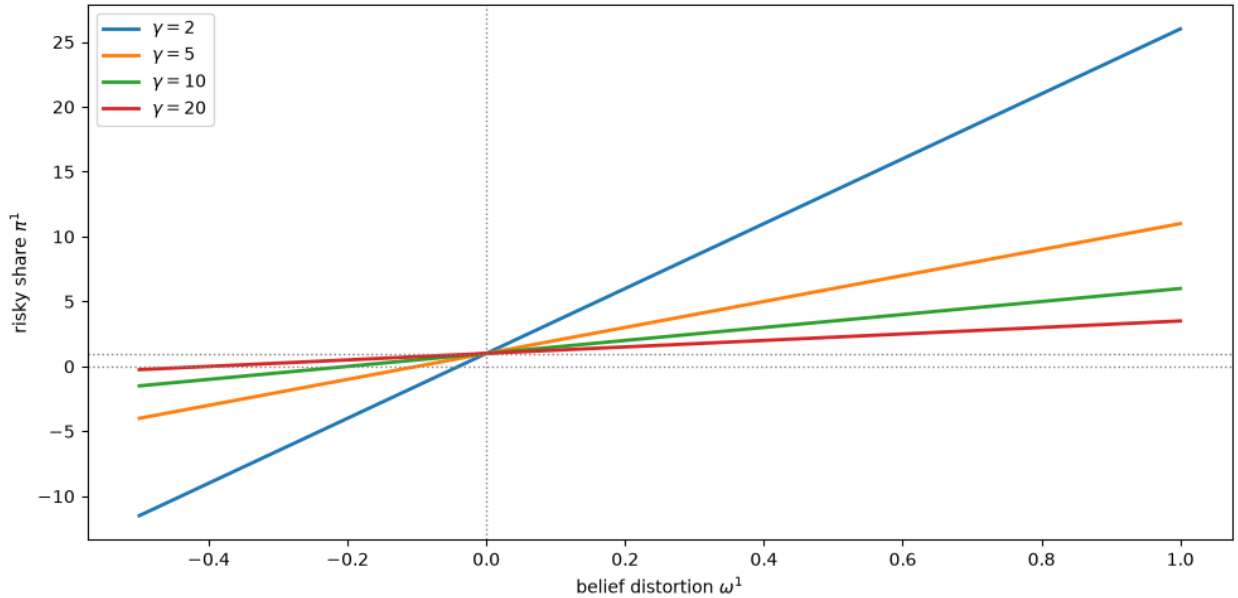


Fig. 35.5: Limiting risky-asset shares of the small agent

This figure plots the limiting risky-asset share π^1 of the negligible agent as a function of her belief distortion ω^1 ($\omega^2 = 0$, $\sigma_Y = 0.02$), for four levels of risk aversion.

At $\omega^1 = 0$ the agent agrees with agent 2 and holds the market portfolio ($\pi^1 = 1$).

Optimism ($\omega^1 > 0$) leads to leverage ($\pi^1 > 1$), while sufficient pessimism ($\omega^1 < 0$) leads to shorting ($\pi^1 < 0$).

Higher risk aversion compresses these deviations toward one.

35.12 Optimistic and pessimistic distortions

Optimistic and pessimistic beliefs affect survival asymmetrically.

An optimistic agent benefits from the risk premium term and, when IES > 1, from the saving term as well.

A pessimistic agent gives up the risk premium and can survive only if the saving effect is strong enough to offset that loss.

```
fig, axes = plt.subplots(1, 2, figsize=(14, 5), sharey=True)

σ_y = 0.02
ω_2 = 0.0
ρ = 0.67
γ_grid = np.linspace(0.5, 25.0, 300)

ax = axes[0]
for ω_1 in [0.1, 0.25, 0.5, 1.0]:
    _, _, _, total = decompose_survival(ω_1, ω_2, γ_grid, ρ, σ_y)
    ax.plot(γ_grid, total, lw=2, label=rf"$\omega^1 = {ω_1}$")
ax.axhline(0, color="gray", lw=1)
ax.set_title("optimistic", fontsize=12)
ax.set_xlabel(r"risk aversion $\gamma$")
ax.set_ylabel("boundary drift")
ax.legend(fontsize=9)

ax = axes[1]
for ω_1 in [-0.1, -0.25, -0.5, -1.0]:
    _, _, _, total = decompose_survival(ω_1, ω_2, γ_grid, ρ, σ_y)
    ax.plot(γ_grid, total, lw=2, label=rf"$\omega^1 = {ω_1}$")
ax.axhline(0, color="gray", lw=1)
ax.set_title("pessimistic", fontsize=12)
ax.set_xlabel(r"risk aversion $\gamma$")
ax.legend(fontsize=9)

plt.tight_layout()
plt.show()
```

Both panels plot the total boundary drift at $\upsilon = 0$ as a function of γ (IES ≈ 1.49, $\omega^2 = 0$).

Where the curve is positive, agent 1 survives near extinction.

- Left panel (optimistic agent): larger ω^1 means a bigger bet on the risky asset, so the volatility penalty dominates at low γ but the drift turns positive once γ is large enough.
- Right panel (pessimistic agent): a pessimistic agent gives up the risk premium by underweighting the risky asset, so the drift is negative for most of the parameter space and survival requires saving motives strong enough to offset the portfolio losses.

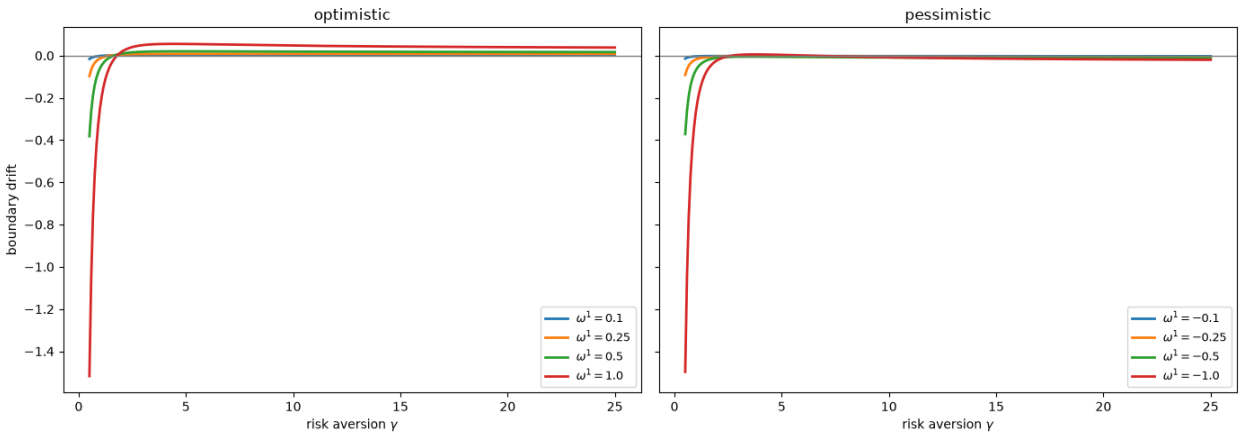


Fig. 35.6: Total boundary drift for optimistic and pessimistic distortions

35.13 Long-run consumption distribution

When both agents survive, the Pareto share keeps moving across the whole interval $(0, 1)$.

The next simulation is only a toy approximation.

It interpolates the drift between its two boundary values, so it illustrates the recurrence logic without solving the full equilibrium ODE.

```
def simulate_pareto_share_toy(ω_1, ω_2, γ, ρ, σ_y, T, dt, n_paths=20, seed=42):
    """
    Simulate a toy Pareto-share process by interpolating boundary drifts.
    """
    rng = np.random.default_rng(seed)
    n_steps = int(T / dt)
    t_grid = np.linspace(0, T, n_steps + 1)

    volatility = ω_1 - ω_2
    m_0 = boundary_drift(ω_1, ω_2, γ, ρ, σ_y)
    m_1 = -boundary_drift(ω_2, ω_1, γ, ρ, σ_y)

    θ = np.zeros((n_paths, n_steps + 1))
    dW = rng.normal(0.0, np.sqrt(dt), size=(n_paths, n_steps))

    for t in range(n_steps):
        υ = 1.0 / (1.0 + np.exp(-θ[:, t]))
        drift = m_0 * (1 - υ) + m_1 * υ
        θ[:, t + 1] = θ[:, t] + drift * dt + volatility * dW[:, t]

    υ_paths = 1.0 / (1.0 + np.exp(-θ))
    return t_grid, υ_paths


ω_1 = 0.25
ω_2 = 0.0
γ = 5.0
ρ = 0.67
σ_y = 0.02
```

(continues on next page)

(continued from previous page)

```
t_grid, υ_paths = simulate_pareto_share_toy(
    ω_1, ω_2, γ, ρ, σ_y, T=500, dt=0.05, n_paths=50, seed=42
)

fig, axes = plt.subplots(1, 2, figsize=(14, 5))

ax = axes[0]
for i in range(20):
    ax.plot(t_grid, υ_paths[i], color="C0", alpha=0.25, lw=1)
ax.axhline(0.5, color="gray", linestyle=":", lw=1)
ax.set_title("sample paths", fontsize=12)
ax.set_xlabel("time")
ax.set_ylabel(r"Pareto share $\upsilon_t$")
ax.set_ylim(0, 1)

ax = axes[1]
_, υ_long = simulate_pareto_share_toy(
    ω_1, ω_2, γ, ρ, σ_y, T=2000, dt=0.05, n_paths=5, seed=123
)
υ_stationary = υ_long[:, υ_long.shape[1] // 2:].ravel()
ax.hist(υ_stationary, bins=80, density=True, color="steelblue",
        edgecolor="white", alpha=0.7)
ax.set_title("approximate stationary density", fontsize=12)
ax.set_xlabel(r"Pareto share $\upsilon$")
ax.set_ylabel("density")
ax.set_xlim(0, 1)

plt.tight_layout()
plt.show()
```

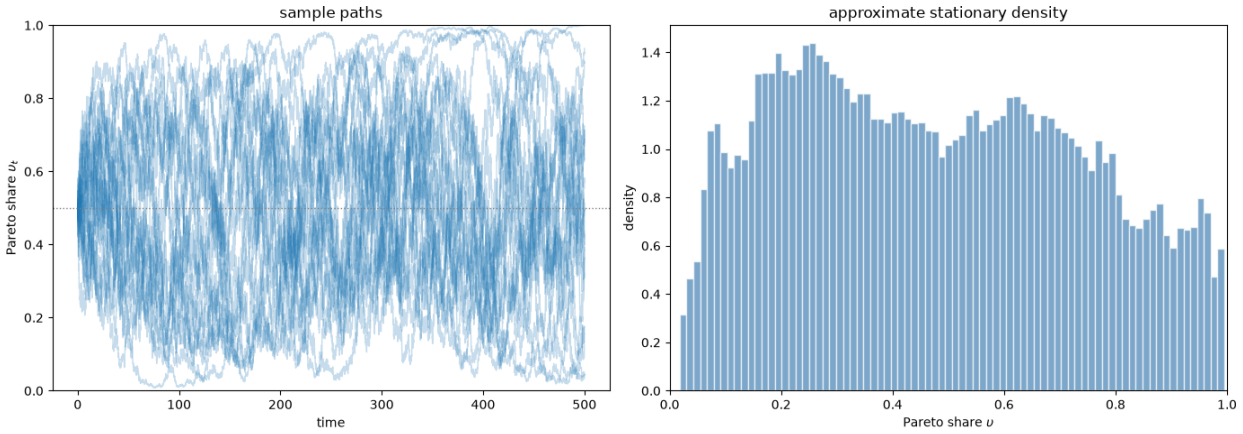


Fig. 35.7: A toy stationary Pareto-share simulation

The left panel shows 20 sample paths of the Pareto share υ_t under parameters inside the coexistence region ($\omega^1 = 0.25$, $\omega^2 = 0$, $\gamma = 5$, IES ≈ 1.49).

Unlike the separable case in Fig. 35.4, the paths do not drift to zero – they repeatedly visit a wide range of values, bouncing between the two repelling boundaries.

The right panel approximates the stationary density by pooling the second half of longer simulations.

The interior mode is consistent with neither agent being driven to extinction.

However, this toy interpolation only illustrates the recurrence logic; it does not reproduce the quantitative stationary consumption-share density in Figure 4 of Borovička [2020], which requires solving the full interior equilibrium ODE.

35.14 Summary

Recursive preferences weaken the classical market-selection result.

The portfolio return channel still rewards more optimistic beliefs.

The volatility channel still penalizes aggressive positions.

But when IES > 1, the saving channel can be strong enough to keep a distorted-belief agent alive.

This is why recursive-preference economies can support stationary long-run wealth distributions with persistent heterogeneity in beliefs and portfolio positions.

Part V

Linear Programming

CHAPTER

THIRTYSIX

OPTIMAL TRANSPORT

36.1 Overview

The **transportation** or **optimal transport** problem is interesting both because of its many applications and because of its important role in the history of economic theory.

In this lecture, we describe the problem, tell how linear programming is a key tool for solving it, and then provide some examples.

We will provide other applications in followup lectures.

The optimal transport problem was studied in early work about linear programming, as summarized for example by [Dorfman *et al.*, 1958]. A modern reference about applications in economics is [Galichon, 2016].

Below, we show how to solve the optimal transport problem using several implementations of linear programming, including, in order,

1. the linprog solver from SciPy,
2. the linprog_simplex solver from QuantEcon and
3. the simplex-based solvers included in the Python Optimal Transport package.

```
!pip install --upgrade quantecon
!pip install --upgrade POT
```

Let's start with some imports.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import linprog
from quantecon.optimize.linprog_simplex import linprog_simplex
import ot
from scipy.stats import betabinom
import networkx as nx
```

36.2 The Optimal Transport Problem

Suppose that m factories produce goods that must be sent to n locations.

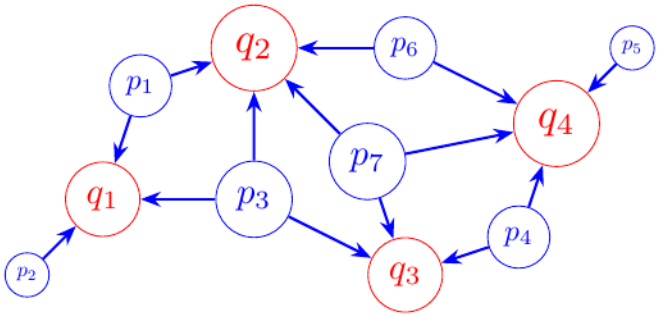
Let

- x_{ij} denote the quantity shipped from factory i to location j
- c_{ij} denote the cost of shipping one unit from factory i to location j
- p_i denote the capacity of factory i and q_j denote the amount required at location j.
- $i = 1, 2, \ldots, m$ and $j = 1, 2, \ldots, n$.

A planner wants to minimize total transportation costs subject to the following constraints:

- The amount shipped **from** each factory must equal its capacity.
- The amount shipped **to** each location must equal the quantity required there.

The figure below shows one visualization of this idea, when factories and target locations are distributed in the plane.



The size of the vertices in the figure are proportional to

- capacity, for the factories, and
- demand (amount required) for the target locations.

The arrows show one possible transport plan, which respects the constraints stated above.

The planner's problem can be expressed as the following constrained minimization problem:

$$\begin{aligned} \min_{x_{ij}} \ & \sum_{i=1}^{m} \sum_{j=1}^{n} c_{ij} x_{ij} \\ \text{subject to} \ & \sum_{j=1}^{n} x_{ij} = p_i, \quad i = 1, 2, \ldots, m \\ & \sum_{i=1}^{m} x_{ij} = q_j, \quad j = 1, 2, \ldots, n \\ & x_{ij} \geq 0 \end{aligned} \tag{36.1}$$

This is an **optimal transport problem** with

- mn decision variables, namely, the entries x_{ij} and
- $m + n$ constraints.

Summing the q_j's across all j's and the p_i's across all i's indicates that the total capacity of all the factories equals total requirements at all locations:

$$\sum_{j=1}^{n} q_j = \sum_{j=1}^{n} \sum_{i=1}^{m} x_{ij} = \sum_{i=1}^{m} \sum_{j=1}^{n} x_{ij} = \sum_{i=1}^{m} p_i \tag{36.2}$$

The presence of the restrictions in (36.2) will be the source of one redundancy in the complete set of restrictions that we describe below.

More about this later.

36.3 The Linear Programming Approach

In this section we discuss using using standard linear programming solvers to tackle the optimal transport problem.

36.3.1 Vectorizing a Matrix of Decision Variables

A *matrix* of decision variables x_{ij} appears in problem (36.1).

The SciPy function `linprog` expects to see a *vector* of decision variables.

This situation impels us to rewrite our problem in terms of a *vector* of decision variables.

Let

- X, C be $m \times n$ matrices with entries x_{ij}, c_{ij},
- p be m-dimensional vector with entries p_i,
- q be n-dimensional vector with entries q_j.

With $\mathbf{1}_n$ denoting the n-dimensional column vector $(1, 1, \ldots, 1)'$, our problem can now be expressed compactly as:

$$\begin{aligned} \min_X \ & \text{tr}(C'X) \\ \text{subject to } & X\,\mathbf{1}_n = p \\ & X'\,\mathbf{1}_m = q \\ & X \geq 0 \end{aligned}$$

We can convert the matrix X into a vector by stacking all of its columns into a column vector.

Doing this is called **vectorization**, an operation that we denote $\text{vec}(X)$.

Similarly, we convert the matrix C into an mn-dimensional vector $\text{vec}(C)$.

The objective function can be expressed as the inner product between $\text{vec}(C)$ and $\text{vec}(X)$:

$$\text{vec}(C)' \cdot \text{vec}(X).$$

To express the constraints in terms of $\text{vec}(X)$, we use a **Kronecker product** denoted by $\otimes$ and defined as follows.

Suppose A is an $m \times s$ matrix with entries (a_{ij}) and that B is an $n \times t$ matrix.

The **Kronecker product** of A and B is defined, in block matrix form, by

$$A \otimes B = \begin{bmatrix} a_{11}B & a_{12}B & \ldots & a_{1s}B \\ a_{21}B & a_{22}B & \ldots & a_{2s}B \\ & & \vdots & \\ a_{m1}B & a_{m2}B & \ldots & a_{ms}B \end{bmatrix}.$$

$A \otimes B$ is an $mn \times st$ matrix.

It has the property that for any $m \times n$ matrix X

$$\text{vec}(A'XB) = (B' \otimes A')\,\text{vec}(X). \tag{36.3}$$

We can now express our constraints in terms of $\text{vec}(X)$.

Let $A = \mathbf{I}'_m, B = \mathbf{1}_n$.

By equation (36.3)

$$X\,\mathbf{1}_n = \text{vec}(X\,\mathbf{1}_n) = \text{vec}(\mathbf{I}_m X\,\mathbf{1}_n) = (\mathbf{1}'_n \otimes \mathbf{I}_m)\,\text{vec}(X).$$

where $\mathbf{I}_m$ denotes the $m \times m$ identity matrix.

Constraint $X\,\mathbf{1}_n = p$ can now be written as:

$$(\mathbf{1}'_n \otimes \mathbf{I}_m)\,\text{vec}(X) = p.$$

Similarly, the constraint $X'\,\mathbf{1}_m = q$ can be rewriten as:

$$(\mathbf{I}_n \otimes \mathbf{1}'_m)\,\text{vec}(X) = q.$$

With $z := \text{vec}(X)$, our problem can now be expressed in terms of an mn-dimensional vector of decision variables:

$$\begin{aligned} \min_z \ & \text{vec}(C)'z \\ \text{subject to } \ & Az = b \\ & z \geq 0 \end{aligned} \tag{36.4}$$

where

$$A = \begin{bmatrix} \mathbf{1}'_n \otimes \mathbf{I}_m \\ \mathbf{I}_n \otimes \mathbf{1}'_m \end{bmatrix} \quad \text{and} \quad b = \begin{bmatrix} p \\ q \end{bmatrix}$$

36.3.2 An Application

We now provide an example that takes the form (36.4) that we'll solve by deploying the function `linprog`.

The table below provides numbers for the requirements vector q, the capacity vector p, and entries c_{ij} of the cost-of-shipping matrix C.

The numbers in the above table tell us to set $m = 3$, $n = 5$, and construct the following objects:

$$p = \begin{bmatrix} 50 \\ 100 \\ 150 \end{bmatrix}, \quad q = \begin{bmatrix} 25 \\ 115 \\ 60 \\ 30 \\ 70 \end{bmatrix} \quad \text{and} \quad C = \begin{bmatrix} 10 & 15 & 20 & 20 & 40 \\ 20 & 40 & 15 & 30 & 30 \\ 30 & 35 & 40 & 55 & 25 \end{bmatrix}.$$

Let's write Python code that sets up the problem and solves it.

```
# Define parameters
m = 3
n = 5

p = np.array([50.0, 100.0, 150.0])
```

(continues on next page)

(continued from previous page)

```
q = np.array([25.0, 115.0, 60.0, 30.0, 70.0])

C = np.array([[10.0, 15.0, 20.0, 20.0, 40.0],
              [20.0, 40.0, 15.0, 30.0, 30.0],
              [30.0, 35.0, 40.0, 55.0, 25.0]])

# Vectorize matrix C
C_vec = C.reshape((m*n, 1), order='F')

# Construct matrix A by Kronecker product
A1 = np.kron(np.ones((1, n)), np.identity(m))
A2 = np.kron(np.identity(n), np.ones((1, m)))
A = np.vstack([A1, A2])

# Construct vector b
b = np.hstack([p, q])

# Solve the primal problem
res = linprog(C_vec, A_eq=A, b_eq=b)

# Print results
print("message:", res.message)
print("nit:", res.nit)
print("fun:", res.fun)
print("z:", res.x)
print("X:", res.x.reshape((m,n), order='F'))
```

```
message: Optimization terminated successfully. (HiGHS Status 7: Optimal)
nit: 8
fun: 7225.0
z: [ 0. 10. 15. 50.  0. 65.  0. 60.  0.  0. 30.  0.  0.  0. 70.]
X: [[ 0. 50.  0.  0.  0.]
 [10.  0. 60. 30.  0.]
 [15. 65.  0.  0. 70.]]
```

Notice how, in the line `C_vec = C.reshape((m*n, 1), order='F')`, we are careful to vectorize using the flag `order='F'`.

This is consistent with converting C into a vector by stacking all of its columns into a column vector.

Here `'F'` stands for "Fortran", and we are using Fortran style column-major order.

(For an alternative approach, using Python's default row-major ordering, see this lecture by Alfred Galichon.)

Interpreting the solver behavior:

Looking at matrix A, we can see that it is rank deficient.

```
np.linalg.matrix_rank(A) < min(A.shape)
```

```
np.True_
```

This indicates that the linear program has been set up to include one or more redundant constraints.

Here, the source of the redundancy is the structure of restrictions (36.2).

Let's explore this further by printing out A and staring at it.

```
A
```

```
array([[1., 0., 0., 1., 0., 0., 1., 0., 0., 1., 0., 0., 1., 0., 0.],
       [0., 1., 0., 0., 1., 0., 0., 1., 0., 0., 1., 0., 0., 1., 0.],
       [0., 0., 1., 0., 0., 1., 0., 0., 1., 0., 0., 1., 0., 0., 1.],
       [1., 1., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 1., 1., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 1., 1., 1., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 1., 1., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 1., 1.]])
```

The singularity of A reflects that the first three constraints and the last five constraints both require that "total requirements equal total capacities" expressed in (36.2).

One equality constraint here is redundant.

Fortunately, SciPy's `linprog` function handles the redundant constraints automatically without explicitly warning about rank deficiency.

But we can drop one of the equality constraints, and use only 7 of them.

After doing this, we attain the same minimized cost.

However, we find a different transportation plan.

Though it is a different plan, it attains the same cost!

```
linprog(C_vec, A_eq=A[:-1], b_eq=b[:-1])
```

```
        message: Optimization terminated successfully. (HiGHS Status 7: Optimal)
        success: True
         status: 0
            fun: 7225.0
              x: [ 0.000e+00  1.000e+01 ...  0.000e+00  7.000e+01]
            nit: 8
          lower:  residual: [ 0.000e+00  1.000e+01 ...  0.000e+00
                              7.000e+01]
                 marginals: [ 0.000e+00  0.000e+00 ...  1.500e+01
                              0.000e+00]
          upper:  residual: [       inf        inf ...        inf
                                    inf]
                 marginals: [ 0.000e+00  0.000e+00 ...  0.000e+00
                              0.000e+00]
          eqlin:  residual: [ 0.000e+00  0.000e+00  0.000e+00  0.000e+00
                              0.000e+00  0.000e+00  0.000e+00]
                 marginals: [ 5.000e+00  1.500e+01  2.500e+01  5.000e+00
                              1.000e+01 -0.000e+00  1.500e+01]
        ineqlin:  residual: []
                 marginals: []
 mip_node_count: 0
 mip_dual_bound: 0.0
        mip_gap: 0.0
```

```
%time linprog(C_vec, A_eq=A[:-1], b_eq=b[:-1])
```

```
CPU times: user 2.28 ms, sys: 0 ns, total: 2.28 ms
Wall time: 2.09 ms
```

```
        message: Optimization terminated successfully. (HiGHS Status 7: Optimal)
        success: True
         status: 0
            fun: 7225.0
              x: [ 0.000e+00  1.000e+01 ...  0.000e+00  7.000e+01]
            nit: 8
          lower:  residual: [ 0.000e+00  1.000e+01 ...  0.000e+00
                              7.000e+01]
                 marginals: [ 0.000e+00  0.000e+00 ...  1.500e+01
                              0.000e+00]
          upper:  residual: [       inf        inf ...        inf
                                    inf]
                 marginals: [ 0.000e+00  0.000e+00 ...  0.000e+00
                              0.000e+00]
          eqlin:  residual: [ 0.000e+00  0.000e+00  0.000e+00  0.000e+00
                              0.000e+00  0.000e+00  0.000e+00]
                 marginals: [ 5.000e+00  1.500e+01  2.500e+01  5.000e+00
                              1.000e+01 -0.000e+00  1.500e+01]
        ineqlin:  residual: []
                 marginals: []
 mip_node_count: 0
 mip_dual_bound: 0.0
        mip_gap: 0.0
```

```
%time linprog(C_vec, A_eq=A, b_eq=b)
```

```
CPU times: user 2.32 ms, sys: 38 µs, total: 2.36 ms
Wall time: 2.18 ms
```

```
        message: Optimization terminated successfully. (HiGHS Status 7: Optimal)
        success: True
         status: 0
            fun: 7225.0
              x: [ 0.000e+00  1.000e+01 ...  0.000e+00  7.000e+01]
            nit: 8
          lower:  residual: [ 0.000e+00  1.000e+01 ...  0.000e+00
                              7.000e+01]
                 marginals: [ 0.000e+00  0.000e+00 ...  1.500e+01
                              0.000e+00]
          upper:  residual: [       inf        inf ...        inf
                                    inf]
                 marginals: [ 0.000e+00  0.000e+00 ...  0.000e+00
                              0.000e+00]
          eqlin:  residual: [ 0.000e+00  0.000e+00  0.000e+00  0.000e+00
                              0.000e+00  0.000e+00  0.000e+00  0.000e+00]
                 marginals: [ 1.000e+01  2.000e+01  3.000e+01 -0.000e+00
                              5.000e+00 -5.000e+00  1.000e+01 -5.000e+00]
        ineqlin:  residual: []
                 marginals: []
 mip_node_count: 0
 mip_dual_bound: 0.0
        mip_gap: 0.0
```

Evidently, it is slightly quicker to work with the system that removed a redundant constraint.

Let's drill down and do some more calculations to help us understand whether or not our finding **two** different optimal transport plans reflects our having dropped a redundant equality constraint.

Hint

It will turn out that dropping a redundant equality isn't really what mattered.

To verify our hint, we shall simply use **all** of the original equality constraints (including a redundant one), but we'll just shuffle the order of the constraints.

```
arr = np.arange(m+n)
```

```
sol_found = []
cost = []
rng = np.random.default_rng()

# simulate 1000 times
for i in range(1000):

    rng.shuffle(arr)
    res_shuffle = linprog(C_vec, A_eq=A[arr], b_eq=b[arr])

    # if find a new solution
    sol = tuple(res_shuffle.x)
    if sol not in sol_found:
        sol_found.append(sol)
        cost.append(res_shuffle.fun)
```

```
for i in range(len(sol_found)):
    print(f"transportation plan {i}: ", sol_found[i])
    print(f"     minimized cost {i}: ", cost[i])
```

```
transportation plan 0:  (np.float64(0.0), np.float64(10.0), np.float64(15.0), np.
↪float64(50.0), np.float64(0.0), np.float64(65.0), np.float64(0.0), np.float64(60.
↪0), np.float64(0.0), np.float64(0.0), np.float64(30.0), np.float64(0.0), np.
↪float64(0.0), np.float64(0.0), np.float64(70.0))
    minimized cost 0:  7225.0
```

Ah hah! As you can see, putting constraints in different orders in this case uncovers two optimal transportation plans that achieve the same minimized cost.

These are the same two plans computed earlier.

Next, we show that leaving out the first constraint "accidentally" leads to the initial plan that we computed.

```
linprog(C_vec, A_eq=A[1:], b_eq=b[1:])
```

```
        message: Optimization terminated successfully. (HiGHS Status 7: Optimal)
        success: True
         status: 0
            fun: 7225.0
              x: [ 0.000e+00  1.000e+01 ...  0.000e+00  7.000e+01]
            nit: 8
          lower:  residual: [ 0.000e+00  1.000e+01 ...  0.000e+00
                              7.000e+01]
                 marginals: [ 0.000e+00  0.000e+00 ...  1.500e+01
                              0.000e+00]
          upper:  residual: [       inf        inf ...        inf
```

(continues on next page)

(continued from previous page)

```
                                  inf]
                marginals: [ 0.000e+00  0.000e+00 ...  0.000e+00
                             0.000e+00]
          eqlin:  residual: [ 0.000e+00  0.000e+00  0.000e+00  0.000e+00
                             0.000e+00  0.000e+00  0.000e+00]
                marginals: [ 1.000e+01  2.000e+01  1.000e+01  1.500e+01
                             5.000e+00  2.000e+01  5.000e+00]
        ineqlin:  residual: []
                marginals: []
 mip_node_count: 0
 mip_dual_bound: 0.0
        mip_gap: 0.0
```

Let's compare this transport plan with

```
res.x
```

```
array([ 0., 10., 15., 50.,  0., 65.,  0., 60.,  0.,  0., 30.,  0.,  0.,
        0., 70.])
```

Here the matrix X contains entries x_{ij} that tell amounts shipped **from** factor $i = 1, 2, 3$ **to** location $j = 1, 2, \ldots, 5$.

The vector z evidently equals vec(X).

The minimized cost from the optimal transport plan is given by the fun variable.

36.3.3 Using a Just-in-Time Compiler

We can also solve optimal transportation problems using a powerful tool from QuantEcon, namely, `quantecon.optimize.linprog_simplex`.

While `scipy.optimize.linprog` uses the HiGHS solver by default, `quantecon.optimize.linprog_simplex` implements the simplex algorithm accelerated by using a just-in-time compiler shipped in the `numba` library.

As you will see very soon, by using `quantecon.optimize.linprog_simplex` the time required to solve an optimal transportation problem can be reduced significantly.

```
# construct matrices/vectors for linprog_simplex
c = C.flatten()

# Equality constraints
A_eq = np.zeros((m+n, m*n))
for i in range(m):
    for j in range(n):
        A_eq[i, i*n+j] = 1
        A_eq[m+j, i*n+j] = 1

b_eq = np.hstack([p, q])
```

Since `quantecon.optimize.linprog_simplex` does maximization instead of minimization, we need to put a negative sign before vector `c`.

```
res_qe = linprog_simplex(-c, A_eq=A_eq, b_eq=b_eq)
```

While the two LP solvers use different algorithms (HiGHS vs. simplex), both should find optimal solutions.

The solutions differs since there are multiple optimal solutions, but the objective values are the same

```
np.allclose(-res_qe.fun, res.fun)
```

```
True
```

```
res_qe.x.reshape((m, n), order='C')
```

```
array([[15., 35.,  0.,  0.,  0.],
       [10.,  0., 60., 30.,  0.],
       [ 0., 80.,  0.,  0., 70.]])
```

```
res.x.reshape((m, n), order='F')
```

```
array([[ 0., 50.,  0.,  0.,  0.],
       [10.,  0., 60., 30.,  0.],
       [15., 65.,  0.,  0., 70.]])
```

Let's do a speed comparison between `scipy.optimize.linprog` and `quantecon.optimize.linprog_simplex`.

```
# scipy.optimize.linprog
%time res = linprog(C_vec, A_eq=A[:-1, :], b_eq=b[:-1])
```

```
CPU times: user 2.36 ms, sys: 0 ns, total: 2.36 ms
Wall time: 2.17 ms
```

```
# quantecon.optimize.linprog_simplex
%time out = linprog_simplex(-c, A_eq=A_eq, b_eq=b_eq)
```

```
CPU times: user 58 µs, sys: 0 ns, total: 58 µs
Wall time: 60.8 µs
```

As you can see, the `quantecon.optimize.linprog_simplex` is much faster.

(Note however, that the SciPy version is probably more stable than the QuantEcon version, having been tested more extensively over a longer period of time.)

36.4 The Dual Problem

Let u, v denotes vectors of dual decision variables with entries $(u_i), (v_j)$.

The **dual** to **minimization** problem (36.1) is the **maximization** problem:

$$\max_{u_i, v_j} \sum_{i=1}^{m} p_i u_i + \sum_{j=1}^{n} q_j v_j$$
$$\text{subject to } u_i + v_j \leq c_{ij},\ i = 1, 2, \dots, m;\ j = 1, 2, \dots, n \tag{36.5}$$

The dual problem is also a linear programming problem.

It has $m + n$ dual variables and mn constraints.

Vectors u and v of **values** are attached to the first and the second sets of primal constraits, respectively.

Thus, u is attached to the constraints

- $(\mathbf{1}'_n \otimes \mathbf{I}_m)\,\text{vec}(X) = p$

and v is attached to constraints

- $(\mathbf{I}_n \otimes \mathbf{1}'_m)\,\text{vec}(X) = q$.

Components of the vectors u and v of per unit **values** are **shadow prices** of the quantities appearing on the right sides of those constraints.

We can write the dual problem as

$$\begin{aligned} &\max_{u_i, v_j} pu + qv \\ &\text{subject to } \; A' \begin{bmatrix} u \\ v \end{bmatrix} = \text{vec}(C) \end{aligned} \tag{36.6}$$

For the same numerical example described above, let's solve the dual problem.

```
# Solve the dual problem
res_dual = linprog(-b, A_ub=A.T, b_ub=C_vec,
                   bounds=[(None, None)]*(m+n))

#Print results
print("message:", res_dual.message)
print("nit:", res_dual.nit)
print("fun:", res_dual.fun)
print("u:", res_dual.x[:m])
print("v:", res_dual.x[-n:])
```

```
message: Optimization terminated successfully. (HiGHS Status 7: Optimal)
nit: 9
fun: -7225.0
u: [-20. -10.   0.]
v: [30. 35. 25. 40. 25.]
```

`quantecon.optimize.linprog_simplex` computes and returns the dual variables alongside the primal solution.

The dual variables (shadow prices) can be extracted directly from the primal solution:

```
# The dual variables are returned by linprog_simplex
print("Dual variables from linprog_simplex:")
print("u:", -res_qe.lambd[:m])
print("v:", -res_qe.lambd[m:])
```

```
Dual variables from linprog_simplex:
u: [-20. -10.  -0.]
v: [30. 35. 25. 40. 25.]
```

We can verify these match the dual solution from SciPy:

```
print("Dual variables from SciPy linprog:")
print("u:", res_dual.x[:m])
print("v:", res_dual.x[-n:])
```

```
Dual variables from SciPy linprog:
u: [-20. -10.   0.]
v: [30. 35. 25. 40. 25.]
```

36.4.1 Interpretation of dual problem

By **strong duality** (please see this lecture Linear Programming), we know that:

$$\sum_{i=1}^{m}\sum_{j=1}^{n} c_{ij}x_{ij} = \sum_{i=1}^{m} p_i u_i + \sum_{j=1}^{n} q_j v_j$$

One unit more capacity in factory i, i.e. p_i, results in u_i more transportation costs.

Thus, u_i describes the cost of shipping one unit **from** factory i.

Call this the ship-out cost of one unit shipped from factory i.

Similarly, v_j is the cost of shipping one unit **to** location j.

Call this the ship-in cost of one unit to location j.

Strong duality implies that total transprotation costs equals total ship-out costs **plus** total ship-in costs.

It is reasonable that, for one unit of a product, ship-out cost u_i **plus** ship-in cost v_j should equal transportation cost c_{ij}.

This equality is assured by **complementary slackness** conditions that state that whenever $x_{ij} > 0$, meaning that there are positive shipments from factory i to location j, it must be true that $u_i + v_j = c_{ij}$.

36.5 The Python Optimal Transport Package

There is an excellent Python package for optimal transport that simplifies some of the steps we took above.

In particular, the package takes care of the vectorization steps before passing the data out to a linear programming routine.

(That said, the discussion provided above on vectorization remains important, since we want to understand what happens under the hood.)

36.5.1 Replicating Previous Results

The following line of code solves the example application discussed above using linear programming.

```
X = ot.emd(p, q, C)
X
```

```
array([[15., 35.,  0.,  0.,  0.],
       [10.,  0., 60., 30.,  0.],
       [ 0., 80.,  0.,  0., 70.]])
```

Sure enough, we have the same solution and the same cost

```
total_cost = np.vdot(X, C)
total_cost
```

```
np.float64(7225.0)
```

Here we use np.vdot for the trace inner product of X and C

36.5.2 A Larger Application

Now let's try using the same package on a slightly larger application.

The application has the same interpretation as above but we will also give each node (i.e., vertex) a location in the plane.

This will allow us to plot the resulting transport plan as edges in a graph.

The following class defines a node by

- its location $(x, y) \in \mathbb{R}^2$,
- its group (factory or location, denoted by `p` or `q`) and
- its mass (e.g., p_i or q_j).

```
class Node:

    def __init__(self, x, y, mass, group, name):

        self.x, self.y = x, y
        self.mass, self.group = mass, group
        self.name = name
```

Next we write a function that repeatedly calls the class above to build instances.

It allocates to the nodes it creates their location, mass, and group.

Locations are assigned randomly.

```
def build_nodes_of_one_type(group='p', n=100, seed=123):

    nodes = []
    rng = np.random.default_rng(seed)

    for i in range(n):

        if group == 'p':
            m = 1/n
            x = rng.uniform(-2, 2)
            y = rng.uniform(-2, 2)
        else:
            m = betabinom.pmf(i, n-1, 2, 2)
            x = 0.6 * rng.uniform(-1.5, 1.5)
            y = 0.6 * rng.uniform(-1.5, 1.5)

        name = group + str(i)
        nodes.append(Node(x, y, m, group, name))

    return nodes
```

Now we build two lists of nodes, each one containing one type (factories or locations)

```
n_p = 32
n_q = 32
p_list = build_nodes_of_one_type(group='p', n=n_p)
q_list = build_nodes_of_one_type(group='q', n=n_q)

p_probs = [p.mass for p in p_list]
q_probs = [q.mass for q in q_list]
```

For the cost matrix C, we use the Euclidean distance between each factory and location.

```
c = np.empty((n_p, n_q))
for i in range(n_p):
    for j in range(n_q):
        x0, y0 = p_list[i].x, p_list[i].y
        x1, y1 = q_list[j].x, q_list[j].y
        c[i, j] = np.sqrt((x0-x1)**2 + (y0-y1)**2)
```

Now we are ready to apply the solver

```
%time pi = ot.emd(p_probs, q_probs, c)
```

```
CPU times: user 592 µs, sys: 30 µs, total: 622 µs
Wall time: 421 µs
```

Finally, let's plot the results using `networkx`.

In the plot below,

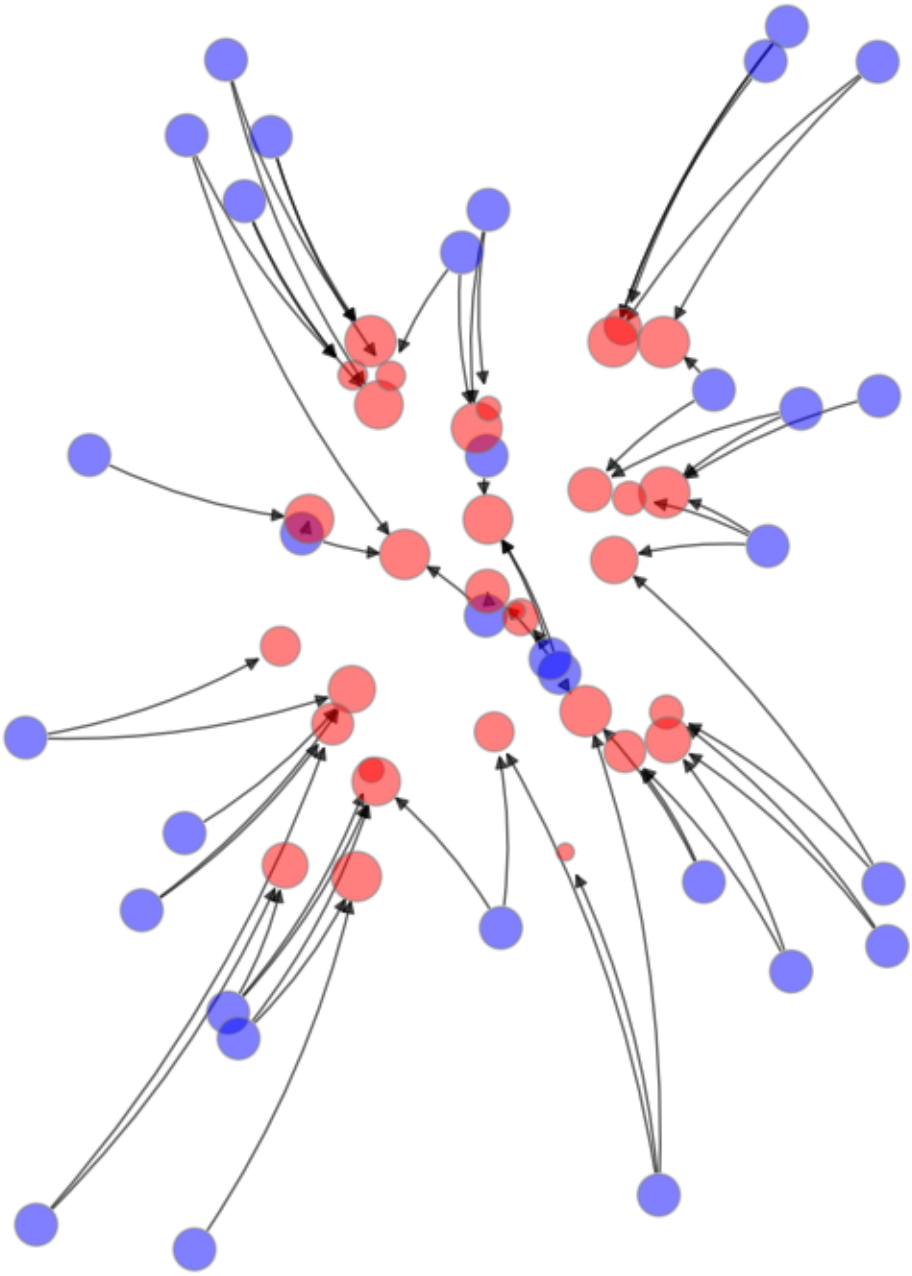
- node size is proportional to probability mass
- an edge (arrow) from i to j is drawn when a positive transfer is made from i to j under the optimal transport plan.

```
g = nx.DiGraph()
g.add_nodes_from([p.name for p in p_list])
g.add_nodes_from([q.name for q in q_list])

for i in range(n_p):
    for j in range(n_q):
        if pi[i, j] > 0:
            g.add_edge(p_list[i].name, q_list[j].name, weight=pi[i, j])

node_pos_dict={}
for p in p_list:
    node_pos_dict[p.name] = (p.x, p.y)

for q in q_list:
    node_pos_dict[q.name] = (q.x, q.y)

node_color_list = []
node_size_list = []
scale = 8_000
for p in p_list:
    node_color_list.append('blue')
    node_size_list.append(p.mass * scale)
for q in q_list:
    node_color_list.append('red')
    node_size_list.append(q.mass * scale)


fig, ax = plt.subplots(figsize=(7, 10))
plt.axis('off')

nx.draw_networkx_nodes(g,
                       node_pos_dict,
                       node_color=node_color_list,
                       node_size=node_size_list,
                       edgecolors='grey',
```

(continues on next page)

(continued from previous page)

```
                        linewidths=1,
                        alpha=0.5,
                        ax=ax)

nx.draw_networkx_edges(g,
                        node_pos_dict,
                        arrows=True,
                        connectionstyle='arc3,rad=0.1',
                        alpha=0.6)
plt.show()
```



CHAPTER

THIRTYSEVEN

VON NEUMANN GROWTH MODEL (AND A GENERALIZATION)

Contents

- *Von Neumann Growth Model (and a Generalization)*
  - *Notation*
  - *Model Ingredients and Assumptions*
  - *Dynamic Interpretation*
  - *Duality*
  - *Interpretation as Two-player Zero-sum Game*

This lecture uses the class `Neumann` to calculate key objects of a linear growth model of John von Neumann [von Neumann, 1937] that was generalized by Kemeny, Morgenstern and Thompson [Kemeny *et al.*, 1956].

Objects of interest are the maximal expansion rate (α), the interest factor (β), the optimal intensities (x), and prices (p).

In addition to watching how the towering mind of John von Neumann formulated an equilibrium model of price and quantity vectors in balanced growth, this lecture shows how fruitfully to employ the following important tools:

- a zero-sum two-player game
- linear programming
- the Perron-Frobenius theorem

We'll begin with some imports:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import fsolve, linprog
from textwrap import dedent

np.set_printoptions(precision=2)
```

The code below provides the `Neumann` class

```
class Neumann:

    """
    This class describes the Generalized von Neumann growth model as it was
    discussed in Kemeny et al. (1956, ECTA) and Gale (1960, Chapter 9.5):
```

(continues on next page)

(continued from previous page)

```
    Let:
    n ... number of goods
    m ... number of activities
    A ... input matrix is m-by-n
        a_{i,j} - amount of good j consumed by activity i
    B ... output matrix is m-by-n
        b_{i,j} - amount of good j produced by activity i

    x ... intensity vector (m-vector) with non-negative entries
        x'B - the vector of goods produced
        x'A - the vector of goods consumed
    p ... price vector (n-vector) with non-negative entries
        Bp - the revenue vector for every activity
        Ap - the cost of each activity

    Both A and B have non-negative entries. Moreover, we assume that
    (1) Assumption I (every good which is consumed is also produced):
        for all j, b_{.,j} > 0, i.e. at least one entry is strictly positive
    (2) Assumption II (no free lunch):
        for all i, a_{i,.} > 0, i.e. at least one entry is strictly positive

    Parameters
    ----------
    A : array_like or scalar(float)
        Part of the state transition equation.  It should be `n x n`
    B : array_like or scalar(float)
        Part of the state transition equation.  It should be `n x k`
    """

    def __init__(self, A, B):

        self.A, self.B = list(map(self.convert, (A, B)))
        self.m, self.n = self.A.shape

        # Check if (A, B) satisfy the basic assumptions
        assert self.A.shape == self.B.shape, 'The input and output matrices \
              must have the same dimensions!'
        assert (self.A >= 0).all() and (self.B >= 0).all(), 'The input and \
              output matrices must have only non-negative entries!'

        # (1) Check whether Assumption I is satisfied:
        if (np.sum(B, 0) <= 0).any():
            self.AI = False
        else:
            self.AI = True

        # (2) Check whether Assumption II is satisfied:
        if (np.sum(A, 1) <= 0).any():
            self.AII = False
        else:
            self.AII = True

    def __repr__(self):
        return self.__str__()

    def __str__(self):
```

(continues on next page)

(continued from previous page)

```
        me = """
        Generalized von Neumann expanding model:
          - number of goods          : {n}
          - number of activities     : {m}

        Assumptions:
          - AI:  every column of B has a positive entry    : {AI}
          - AII: every row of A has a positive entry       : {AII}

        """
        # Irreducible                                       : {irr}
        return dedent(me.format(n=self.n, m=self.m,
                                AI=self.AI, AII=self.AII))

    def convert(self, x):
        """
        Convert array_like objects (lists of lists, floats, etc.) into
        well-formed 2D NumPy arrays
        """
        return np.atleast_2d(np.asarray(x))


    def bounds(self):
        """
        Calculate the trivial upper and lower bounds for alpha (expansion rate)
        and beta (interest factor). See the proof of Theorem 9.8 in Gale (1960)
        """

        n, m = self.n, self.m
        A, B = self.A, self.B

        f = lambda α: ((B - α * A) @ np.ones((n, 1))).max()
        g = lambda β: (np.ones((1, m)) @ (B - β * A)).min()

        UB = fsolve(f, 1).item()  # Upper bound for α, β
        LB = fsolve(g, 2).item()  # Lower bound for α, β

        return LB, UB


    def zerosum(self, γ, dual=False):
        """
        Given gamma, calculate the value and optimal strategies of a
        two-player zero-sum game given by the matrix

                M(gamma) = B - gamma * A

        Row player maximizing, column player minimizing

        Zero-sum game as an LP (primal --> α)

            max (0', 1) @ (x', v)
            subject to
            [-M', ones(n, 1)] @ (x', v)' <= 0
            (x', v) @ (ones(m, 1), 0) = 1
            (x', v) >= (0', -inf)
```

(continues on next page)

(continued from previous page)

```
        Zero-sum game as an LP (dual --> beta)

            min (0', 1) @ (p', u)
            subject to
            [M, -ones(m, 1)] @ (p', u)' <= 0
            (p', u) @ (ones(n, 1), 0) = 1
            (p', u) >= (0', -inf)

        Outputs:
        --------
        value: scalar
            value of the zero-sum game

        strategy: vector
            if dual = False, it is the intensity vector,
            if dual = True, it is the price vector
        """

        A, B, n, m = self.A, self.B, self.n, self.m
        M = B - y * A

        if dual == False:
            # Solve the primal LP (for details see the description)
            # (1) Define the problem for v as a maximization (linprog minimizes)
            c = np.hstack([np.zeros(m), -1])

            # (2) Add constraints :
            # ... non-negativity constraints
            bounds = tuple(m * [(0, None)] + [(None, None)])
            # ... inequality constraints
            A_iq = np.hstack([-M.T, np.ones((n, 1))])
            b_iq = np.zeros((n, 1))
            # ... normalization
            A_eq = np.hstack([np.ones(m), 0]).reshape(1, m + 1)
            b_eq = 1

            res = linprog(c, A_ub=A_iq, b_ub=b_iq, A_eq=A_eq, b_eq=b_eq,
                          bounds=bounds)

        else:
            # Solve the dual LP (for details see the description)
            # (1) Define the problem for v as a maximization (linprog minimizes)
            c = np.hstack([np.zeros(n), 1])

            # (2) Add constraints :
            # ... non-negativity constraints
            bounds = tuple(n * [(0, None)] + [(None, None)])
            # ... inequality constraints
            A_iq = np.hstack([M, -np.ones((m, 1))])
            b_iq = np.zeros((m, 1))
            # ... normalization
            A_eq = np.hstack([np.ones(n), 0]).reshape(1, n + 1)
            b_eq = 1

            res = linprog(c, A_ub=A_iq, b_ub=b_iq, A_eq=A_eq, b_eq=b_eq,
                          bounds=bounds)
```

(continues on next page)

(continued from previous page)

```
        if res.status != 0:
            print(res.message)

        # Pull out the required quantities
        value = res.x[-1]
        strategy = res.x[:-1]

        return value, strategy


    def expansion(self, tol=1e-8, maxit=1000):
        """
        The algorithm used here is described in Hamburger-Thompson-Weil
        (1967, ECTA). It is based on a simple bisection argument and utilizes
        the idea that for a given γ (= α or β), the matrix "M = B - γ * A"
        defines a two-player zero-sum game, where the optimal strategies are
        the (normalized) intensity and price vector.

        Outputs:
        --------
        alpha: scalar
            optimal expansion rate
        """

        LB, UB = self.bounds()

        for iter in range(maxit):

            γ = (LB + UB) / 2
            ZS = self.zerosum(γ=γ)
            V = ZS[0]      # value of the game with γ

            if V >= 0:
                LB = γ
            else:
                UB = γ

            if abs(UB - LB) < tol:
                γ = (UB + LB) / 2
                x = self.zerosum(γ=γ)[1]
                p = self.zerosum(γ=γ, dual=True)[1]
                break

        return γ, x, p

    def interest(self, tol=1e-8, maxit=1000):
        """
        The algorithm used here is described in Hamburger-Thompson-Weil
        (1967, ECTA). It is based on a simple bisection argument and utilizes
        the idea that for a given gamma (= alpha or beta),
        the matrix "M = B - γ * A" defines a two-player zero-sum game,
        where the optimal strategies are the (normalized) intensity and price
        vector

        Outputs:
        --------
        beta: scalar
```

(continues on next page)

(continued from previous page)

```
            optimal interest rate
        """

        LB, UB = self.bounds()

        for iter in range(maxit):
            y = (LB + UB) / 2
            ZS = self.zerosum(y=y, dual=True)
            V = ZS[0]

            if V > 0:
                LB = y
            else:
                UB = y

            if abs(UB - LB) < tol:
                y = (UB + LB) / 2
                p = self.zerosum(y=y, dual=True)[1]
                x = self.zerosum(y=y)[1]
                break

        return y, x, p
```

37.1 Notation

We use the following notation.

0 denotes a vector of zeros.

We call an n-vector **positive** and write $x \gg \mathbf{0}$ if $x_i > 0$ for all $i = 1, 2, \dots, n$.

We call a vector **non-negative** and write $x \geq \mathbf{0}$ if $x_i \geq 0$ for all $i = 1, 2, \dots, n$.

We call a vector **semi-positive** written $x > \mathbf{0}$ if $x \geq \mathbf{0}$ and $x \neq \mathbf{0}$.

For two conformable vectors x and y, $x \gg y$, $x \geq y$ and $x > y$ mean $x - y \gg \mathbf{0}$, $x - y \geq \mathbf{0}$, and $x - y > \mathbf{0}$, respectively.

We let all vectors in this lecture be column vectors; $x^\top$ denotes the transpose of x (i.e., a row vector).

Let ι_n denote a column vector composed of n ones, i.e. $\iota_n = (1, 1, \dots, 1)^\top$.

Let e^i denote a vector (of arbitrary size) containing zeros except for the i th position where it is one.

We denote matrices by capital letters. For an arbitrary matrix A, $a_{i,j}$ represents the entry in its i th row and j th column. $a_{\cdot j}$ and $a_{i \cdot}$ denote the j th column and i th row of A, respectively.

37.2 Model Ingredients and Assumptions

A pair (A, B) of $m \times n$ non-negative matrices defines an economy.

- m is the number of **activities** (or sectors)
- n is the number of **goods** (produced and/or consumed)
- A is called the **input matrix**; $a_{i,j}$ denotes the amount of good j consumed by activity i
- B is called the **output matrix**; $b_{i,j}$ represents the amount of good j produced by activity i

Two key assumptions restrict economy (A, B):

Assumption 37.2.1 (every good that is consumed is also produced)

$$b_{.,j} > \mathbf{0} \quad \forall j = 1, 2, \ldots, n$$

Assumption 37.2.2 (no free lunch)

$$a_{i,.} > \mathbf{0} \quad \forall i = 1, 2, \ldots, m$$

A semi-positive **intensity** m-vector x denotes levels at which activities are operated.

Therefore,

- vector $x^\top A$ gives the total amount of **goods used in production**
- vector $x^\top B$ gives **total outputs**

An economy (A, B) is said to be **productive**, if there exists a non-negative intensity vector $x \geq 0$ such that $x^\top B > x^\top A$.

The semi-positive n-vector p contains prices assigned to the n goods.

The p vector implies **cost** and **revenue** vectors

- the vector Ap tells **costs** of the vector of activities
- the vector Bp tells **revenues** from the vector of activities

Satisfaction of a property of an input-output pair (A, B) called **irreducibility** (or indecomposability) determines whether an economy can be decomposed into multiple "sub-economies".

Definition 37.2.1

For an economy (A, B), the set of goods $S \subset \{1, 2, \ldots, n\}$ is called an **independent subset** if it is possible to produce every good in S without consuming goods from outside S.

Formally, the set S is independent if $\exists T \subset \{1, 2, \ldots, m\}$ (a subset of activities) such that $a_{i,j} = 0$, $\forall i \in T$ and $j \in S^c$ and for all $j \in S$, $\exists i \in T$ for which $b_{i,j} > 0$.

The economy is **irreducible** if there are no proper independent subsets.

We study two examples, both in Chapter 9.6 of Gale [Gale, 1989]

```
# (1) Irreducible (A, B) example: α_0 = β_0
A1 = np.array([[0, 1, 0, 0],
               [1, 0, 0, 1],
               [0, 0, 1, 0]])

B1 = np.array([[1, 0, 0, 0],
               [0, 0, 2, 0],
               [0, 1, 0, 1]])

# (2) Reducible (A, B) example: β_0 < α_0
A2 = np.array([[0, 1, 0, 0, 0, 0],
               [1, 0, 1, 0, 0, 0],
               [0, 0, 0, 1, 0, 0],
               [0, 0, 1, 0, 0, 1],
               [0, 0, 0, 0, 1, 0]])

B2 = np.array([[1, 0, 0, 1, 0, 0],
               [0, 1, 0, 0, 0, 0],
               [0, 0, 1, 0, 0, 0],
               [0, 0, 0, 0, 2, 0],
               [0, 0, 0, 1, 0, 1]])
```

The following code sets up our first Neumann economy or `Neumann` instance

```
n1 = Neumann(A1, B1)
n1
```

```
Generalized von Neumann expanding model:
  - number of goods          : 4
  - number of activities     : 3

Assumptions:
  - AI:  every column of B has a positive entry    : True
  - AII: every row of A has a positive entry       : True
```

Here is a second instance of a Neumann economy

```
n2 = Neumann(A2, B2)
n2
```

```
Generalized von Neumann expanding model:
  - number of goods          : 6
  - number of activities     : 5

Assumptions:
  - AI:  every column of B has a positive entry    : True
  - AII: every row of A has a positive entry       : True
```

37.3 Dynamic Interpretation

Attach a time index t to the preceding objects, regard an economy as a dynamic system, and study sequences

$$\{(A_t, B_t)\}_{t\geq 0}, \qquad \{x_t\}_{t\geq 0}, \qquad \{p_t\}_{t\geq 0}$$

An interesting special case holds the technology process constant and investigates the dynamics of quantities and prices only.

Accordingly, in the rest of this lecture, we assume that $(A_t, B_t) = (A, B)$ for all $t \geq 0$.

A crucial element of the dynamic interpretation involves the timing of production.

We assume that production (consumption of inputs) takes place in period t, while the consequent output materializes in period $t + 1$, i.e., consumption of $x_t^\top A$ in period t results in $x_t^\top B$ amounts of output in period $t + 1$.

These timing conventions imply the following feasibility condition:

$$x_t^\top B \geq x_{t+1}^\top A \qquad \forall t \geq 1$$

which asserts that no more goods can be used today than were produced yesterday.

Accordingly, Ap_t tells the costs of production in period t and Bp_t tells revenues in period $t + 1$.

37.3.1 Balanced Growth

We follow John von Neumann in studying "balanced growth".

Let $./$ denote an elementwise division of one vector by another and let $\alpha > 0$ be a scalar.

Then **balanced growth** is a situation in which

$$x_{t+1}./x_t = \alpha, \quad \forall t \geq 0$$

With balanced growth, the law of motion of x is evidently $x_{t+1} = \alpha x_t$ and so we can rewrite the feasibility constraint as

$$x_t^\top B \geq \alpha x_t^\top A \qquad \forall t$$

In the same spirit, define $\beta \in \mathbb{R}$ as the **interest factor** per unit of time.

We assume that it is always possible to earn a gross return equal to the constant interest factor β by investing "outside the model".

Under this assumption about outside investment opportunities, a no-arbitrage condition gives rise to the following (no profit) restriction on the price sequence:

$$\beta A p_t \geq B p_t \qquad \forall t$$

This says that production cannot yield a return greater than that offered by the outside investment opportunity (here we compare values in period $t + 1$).

The balanced growth assumption allows us to drop time subscripts and conduct an analysis purely in terms of a time-invariant growth rate α and interest factor β.

37.4 Duality

Two problems are connected by a remarkable dual relationship between technological and valuation characteristics of the economy:

Definition 37.4.1

The **technological expansion problem** (TEP) for the economy (A, B) is to find a semi-positive m-vector $x > 0$ and a number $\alpha \in \mathbb{R}$ that satisfy

$$\begin{aligned} \max_{\alpha} \quad & \alpha \\ \text{s.t.} \quad & x^\top B \geq \alpha x^\top A \end{aligned}$$

Theorem 9.3 of David Gale's book [Gale, 1989] asserts that if *Assumption 37.2.1* and *Assumption 37.2.2* are both satisfied, then a maximum value of α exists and that it is positive.

The maximal value is called the *technological expansion rate* and is denoted by α_0. The associated intensity vector x_0 is the *optimal intensity vector*.

Definition 37.4.2

The economic expansion problem (EEP) for (A, B) is to find a semi-positive n-vector $p > 0$ and a number $\beta \in \mathbb{R}$ that satisfy

$$\begin{aligned} \min_{\beta} \quad & \beta \\ \text{s.t.} \quad & Bp \leq \beta Ap \end{aligned}$$

Assumption 37.2.1 and *Assumption 37.2.2* imply existence of a minimum value $\beta_0 > 0$ called the *economic expansion rate*.

The corresponding price vector p_0 is the *optimal price vector*.

Because the criterion functions in the *technological expansion* problem and the *economical expansion problem* are both linearly homogeneous, the optimality of x_0 and p_0 are defined only up to a positive scale factor.

For convenience (and to emphasize a close connection to zero-sum games), we normalize both vectors x_0 and p_0 to have unit length.

A standard duality argument (see Lemma 9.4. in (Gale, 1960) [Gale, 1989]) implies that under *Assumption 37.2.1* and *Assumption 37.2.2*, $\beta_0 \leq \alpha_0$.

But to deduce that $\beta_0 \geq \alpha_0$, *Assumption 37.2.1* and *Assumption 37.2.2* are not sufficient.

Therefore, von Neumann [von Neumann, 1937] went on to prove the following remarkable "duality" result that connects TEP and EEP.

Theorem 37.4.1 (von Neumann)

If the economy (A, B) satisfies *Assumption 37.2.1* and *Assumption 37.2.2*, then there exist (γ^*, x_0, p_0), where $\gamma^* \in [\beta_0, \alpha_0] \subset \mathbb{R}$, $x_0 > 0$ is an m-vector, $p_0 > 0$ is an n-vector, and the following arbitrage conditions hold

$$\begin{aligned} x_0^\top B &\geq \gamma^* x_0^\top A \\ Bp_0 &\leq \gamma^* Ap_0 \\ x_0^\top (B - \gamma^* A)\, p_0 &= 0 \end{aligned}$$

Proof. (Sketch)

Assumption 37.2.1 and *Assumption 37.2.2* imply that there exist (α_0, x_0) and (β_0, p_0) that solve the TEP and EEP, respectively.

If $\gamma^* > \alpha_0$, then by definition of α_0, there cannot exist a semi-positive x that satisfies $x^\top B \geq \gamma^* x^\top A$.

Similarly, if $\gamma^* < \beta_0$, there is no semi-positive p for which $Bp \leq \gamma^* Ap$. Let $\gamma^* \in [\beta_0, \alpha_0]$, then $x_0^\top B \geq \alpha_0 x_0^\top A \geq \gamma^* x_0^\top A$.

Moreover, $Bp_0 \leq \beta_0 Ap_0 \leq \gamma^* Ap_0$. These two inequalities imply $x_0 \left(B - \gamma^* A\right) p_0 = 0$.

Here the constant γ^* is both an expansion factor and an interest factor (not necessarily optimal).

We have already encountered and discussed the first two inequalities that represent feasibility and no-profit conditions.

Moreover, the equality $x_0^\top \left(B - \gamma^* A\right) p_0 = 0$ concisely expresses the requirements that if any good grows at a rate larger than γ^* (i.e., if it is *oversupplied*), then its price must be zero; and that if any activity provides negative profit, it must be unused.

Therefore, the conditions stated in *Theorem 37.4.1* encode all equilibrium conditions.

So *Theorem 37.4.1* essentially states that under *Assumption 37.2.1* and *Assumption 37.2.2* there always exists an equilibrium (γ^*, x_0, p_0) with balanced growth.

Note that *Theorem 37.4.1* is silent about uniqueness of the equilibrium. In fact, it does not rule out (trivial) cases with $x_0^\top B p_0 = 0$ so that nothing of value is produced.

To exclude such uninteresting cases, Kemeny, Morgenstern and Thompson [Kemeny *et al.*, 1956] add an extra requirement

$$x_0^\top B p_0 > 0$$

and call the associated equilibria *economic solutions*.

They show that this extra condition does not affect the existence result, while it significantly reduces the number of (relevant) solutions.

37.5 Interpretation as Two-player Zero-sum Game

To compute the equilibrium (γ^*, x_0, p_0), we follow the algorithm proposed by Hamburger, Thompson and Weil (1967), building on the key insight that an equilibrium (with balanced growth) can be solved as a particular two-player zero-sum game.

First, we introduce some notation.

Consider the $m \times n$ matrix C as a payoff matrix, with the entries representing payoffs from the **minimizing** column player to the **maximizing** row player and assume that the players can use mixed strategies. Thus,

- the row player chooses the m-vector $x > \mathbf{0}$ subject to $\iota_m^\top x = 1$
- the column player chooses the n-vector $p > \mathbf{0}$ subject to $\iota_n^\top p = 1$.

Definition 37.5.1

The $m \times n$ matrix game C has the *solution* $(x^*, p^*, V(C))$ in mixed strategies if

$$(x^*)^\top C e^j \geq V(C) \quad \forall j \in \{1, \dots, n\} \qquad \text{and} \qquad (e^i)^\top C p^* \leq V(C) \quad \forall i \in \{1, \dots, m\}$$

The number $V(C)$ is called the **value** of the game.

From the above definition, it is clear that the value $V(C)$ has two alternative interpretations:

- by playing the appropriate mixed strategy, the maximizing player can assure himself at least $V(C)$ (no matter what the column player chooses)
- by playing the appropriate mixed strategy, the minimizing player can make sure that the maximizing player will not get more than $V(C)$ (irrespective of what is the maximizing player's choice)

A famous theorem of Nash (1951) tells us that there always exists a mixed strategy Nash equilibrium for any *finite* two-player zero-sum game.

Moreover, von Neumann's Minmax Theorem [von Neumann, 1928] implies that

$$V(C) = \max_x \min_p \; x^\top C p = \min_p \max_x \; x^\top C p = (x^*)^\top C p^*$$

37.5.1 Connection with Linear Programming (LP)

Nash equilibria of a finite two-player zero-sum game solve a linear programming problem.

To see this, we introduce the following notation

- For a fixed x, let v be the value of the minimization problem: $v \equiv \min_p x^\top C p = \min_j x^\top C e^j$
- For a fixed p, let u be the value of the maximization problem: $u \equiv \max_x x^\top C p = \max_i (e^i)^\top C p$

Then the *max-min problem* (the game from the maximizing player's point of view) can be written as the *primal* LP

$$\begin{aligned} V(C) = \max \quad & v \\ \text{s.t.} \quad & v \iota_n^\top \leq x^\top C \\ & x \geq \mathbf{0} \\ & \iota_n^\top x = 1 \end{aligned}$$

while the *min-max problem* (the game from the minimizing player's point of view) is the *dual* LP

$$\begin{aligned} V(C) = \min \quad & u \\ \text{s.t.} \quad & u \iota_m \geq C p \\ & p \geq \mathbf{0} \\ & \iota_m^\top p = 1 \end{aligned}$$

Hamburger, Thompson and Weil [Hamburger *et al.*, 1967] view the input-output pair of the economy as payoff matrices of two-player zero-sum games.

Using this interpretation, they restate *Assumption 37.2.1* and *Assumption 37.2.2* as follows

$$V(-A) < 0 \qquad \text{and} \qquad V(B) > 0$$

Proof. (Sketch)

- $\Rightarrow V(B) > 0$ implies $x_0^\top B \gg \mathbf{0}$, where x_0 is a maximizing vector. Since B is non-negative, this requires that each column of B has at least one positive entry, which is *Assumption 37.2.1*.
- $\Leftarrow$ From *Assumption 37.2.1* and the fact that $p > \mathbf{0}$, it follows that $Bp > \mathbf{0}$. This implies that the maximizing player can always choose x so that $x^\top Bp > 0$ so that it must be the case that $V(B) > 0$.

In order to (re)state *Theorem 37.4.1* in terms of a particular two-player zero-sum game, we define a matrix for $\gamma \in \mathbb{R}$

$$M(\gamma) \equiv B - \gamma A$$

For fixed γ, treating $M(\gamma)$ as a matrix game, calculating the solution of the game implies

- If $\gamma > \alpha_0$, then for all $x > 0$, there $\exists j \in \{1, \dots, n\}$, s.t. $[x^\top M(\gamma)]_j < 0$ implying that $V(M(\gamma)) < 0$.
- If $\gamma < \beta_0$, then for all $p > 0$, there $\exists i \in \{1, \dots, m\}$, s.t. $[M(\gamma)p]_i > 0$ implying that $V(M(\gamma)) > 0$.
- If $\gamma \in \{\beta_0, \alpha_0\}$, then (by *Theorem 37.4.1*) the optimal intensity and price vectors x_0 and p_0 satisfy

$$x_0^\top M(\gamma) \geq \mathbf{0}^\top \qquad \text{and} \qquad M(\gamma)p_0 \leq \mathbf{0}$$

That is, $(x_0, p_0, 0)$ is a solution of the game $M(\gamma)$ so that $V\left(M(\beta_0)\right) = V\left(M(\alpha_0)\right) = 0$.

- If $\beta_0 < \alpha_0$ and $\gamma \in (\beta_0, \alpha_0)$, then $V(M(\gamma)) = 0$.

Moreover, if x' is optimal for the maximizing player in $M(\gamma')$ for $\gamma' \in (\beta_0, \alpha_0)$ and p'' is optimal for the minimizing player in $M(\gamma'')$ where $\gamma'' \in (\beta_0, \gamma')$, then $(x', p'', 0)$ is a solution for $M(\gamma)$ $\forall \gamma \in (\gamma'', \gamma')$.

Proof. (Sketch) If x' is optimal for a maximizing player in game $M(\gamma')$, then $(x')^\top M(\gamma') \geq \mathbf{0}^\top$ and so for all $\gamma < \gamma'$.

$$(x')^\top M(\gamma) = (x')^\top M(\gamma') + (x')^\top (\gamma' - \gamma) A \geq \mathbf{0}^\top$$

hence $V(M(\gamma)) \geq 0$. If p'' is optimal for a minimizing player in game $M(\gamma'')$, then $M(\gamma'')p'' \leq \mathbf{0}$ and so for all $\gamma'' < \gamma$

$$M(\gamma)p'' = M(\gamma'') + (\gamma'' - \gamma) A p'' \leq \mathbf{0}$$

hence $V(M(\gamma)) \leq 0$.

It is clear from the above argument that β_0, α_0 are the minimal and maximal γ for which $V(M(\gamma)) = 0$.

Furthermore, Hamburger et al. [Hamburger *et al.*, 1967] show that the function $\gamma \mapsto V(M(\gamma))$ is continuous and nonincreasing in γ.

This suggests an algorithm to compute (α_0, x_0) and (β_0, p_0) for a given input-output pair (A, B).

37.5.2 Algorithm

Hamburger, Thompson and Weil [Hamburger *et al.*, 1967] propose a simple bisection algorithm to find the minimal and maximal roots (i.e. β_0 and α_0) of the function $\gamma \mapsto V(M(\gamma))$.

Step 1

First, notice that we can easily find trivial upper and lower bounds for α_0 and β_0.

- TEP requires that $x^\top(B - \alpha A) \geq \mathbf{0}^\top$ and $x > \mathbf{0}$, so if α is so large that $\max_i\{[(B - \alpha A)\iota_n]_i\} < 0$, then TEP ceases to have a solution.

Accordingly, let **UB** be the α^* that solves $\max_i\{[(B - \alpha^* A)\iota_n]_i\} = 0$.

- Similar to the upper bound, if β is so low that $\min_j\{[\iota_m^\top(B - \beta A)]_j\} > 0$, then the EEP has no solution and so we can define **LB** as the β^* that solves $\min_j\{[\iota_m^\top(B - \beta^* A)]_j\} = 0$.

The *bounds* method calculates these trivial bounds for us

```
n1.bounds()
```

```
(1.0, 2.0)
```

Step 2

Compute α_0 and β_0

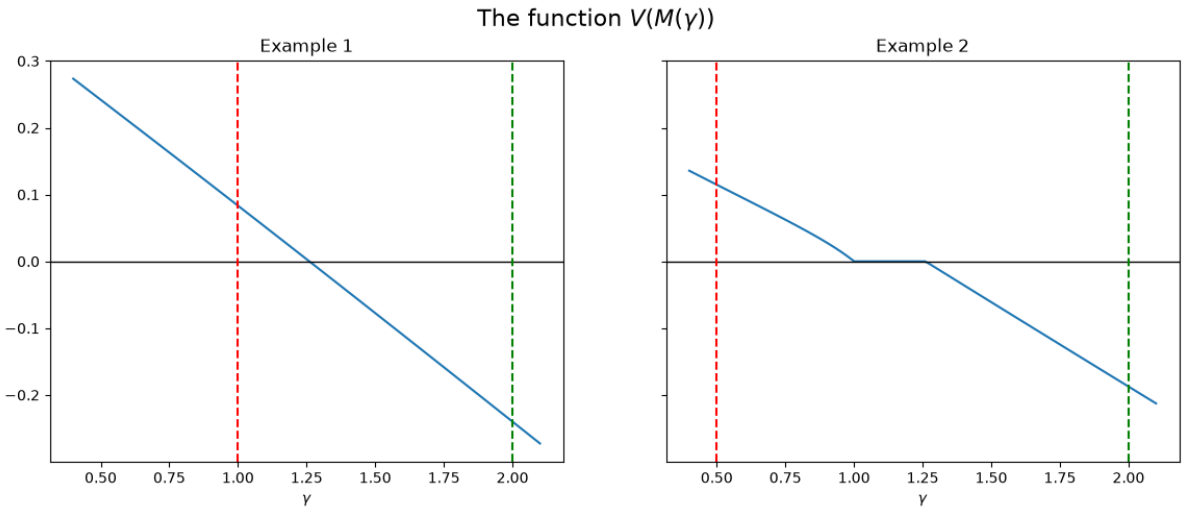
- Finding α_0
  1. Fix $\gamma = \frac{UB+LB}{2}$ and compute the solution of the two-player zero-sum game associated with $M(\gamma)$. We can use either the primal or the dual LP problem.
  2. If $V(M(\gamma)) \geq 0$, then set $LB = \gamma$, otherwise let $UB = \gamma$.
  3. Iterate on 1. and 2. until $|UB - LB| < \epsilon$.
- Finding β_0
  1. Fix $\gamma = \frac{UB+LB}{2}$ and compute the solution of the two-player zero-sum game associated with $M(\gamma)$. We can use either the primal or the dual LP problem.
  2. If $V(M(\gamma)) > 0$, then set $LB = \gamma$, otherwise let $UB = \gamma$.
  3. Iterate on 1. and 2. until $|UB - LB| < \epsilon$.
- *Existence*: Since $V(M(LB)) > 0$ and $V(M(UB)) < 0$ and $V(M(\cdot))$ is a continuous, nonincreasing function, there is at least one $\gamma \in [LB, UB]$, s.t. $V(M(\gamma)) = 0$.

The *zerosum* method calculates the value and optimal strategies associated with a given γ.

```
y = 2

print(f'Value of the game with y = {y}')
print(n1.zerosum(y=y)[0])
print('Intensity vector (from the primal)')
print(n1.zerosum(y=y)[1])
print('Price vector (from the dual)')
print(n1.zerosum(y=y, dual=True)[1])
```

```
Value of the game with γ = 2
-0.24
Intensity vector (from the primal)
[0.32 0.28 0.4 ]
Price vector (from the dual)
[0.4  0.32 0.28 0.  ]
```

```
numb_grid = 100
γ_grid = np.linspace(0.4, 2.1, numb_grid)

value_ex1_grid = np.asarray([n1.zerosum(γ=γ_grid[i])[0]
                             for i in range(numb_grid)])
value_ex2_grid = np.asarray([n2.zerosum(γ=γ_grid[i])[0]
                             for i in range(numb_grid)])

fig, axes = plt.subplots(1, 2, figsize=(14, 5), sharey=True)
fig.suptitle(r'The function $V(M(\gamma))$', fontsize=16)

for ax, grid, N, i in zip(axes, (value_ex1_grid, value_ex2_grid),
                          (n1, n2), (1, 2)):
    ax.plot(γ_grid, grid)
    ax.set(title=f'Example {i}', xlabel=r'$\gamma$')
    ax.axhline(0, c='k', lw=1)
    ax.axvline(N.bounds()[0], c='r', ls='--', label='lower bound')
    ax.axvline(N.bounds()[1], c='g', ls='--', label='upper bound')

plt.show()
```



The *expansion* method implements the bisection algorithm for α_0 (and uses the primal LP problem for x_0)

```
α_0, x, p = n1.expansion()
print(f'α_0 = {α_0}')
print(f'x_0 = {x}')
print(f'The corresponding p from the dual = {p}')
```

```
α_0 = 1.2599210478365421
x_0 = [0.33 0.26 0.41]
The corresponding p from the dual = [0.41 0.33 0.26 0.  ]
```

The *interest* method implements the bisection algorithm for β_0 (and uses the dual LP problem for p_0)

```
β_0, x, p = n1.interest()
print(f'β_0 = {β_0}')
print(f'p_0 = {p}')
print(f'The corresponding x from the primal = {x}')
```

```
β_0 = 1.2599210478365421
p_0 = [0.41 0.33 0.26 0.  ]
The corresponding x from the primal = [0.33 0.26 0.41]
```

Of course, when γ^* is unique, it is irrelevant which one of the two methods we use – both work.

In particular, as will be shown below, in case of an irreducible (A, B) (like in Example 1), the maximal and minimal roots of $V(M(\gamma))$ necessarily coincide implying a "full duality" result, i.e. $\alpha_0 = \beta_0 = \gamma^*$ so that the expansion (and interest) rate γ^* is unique.

37.5.3 Uniqueness and Irreducibility

As an illustration, compute first the maximal and minimal roots of $V(M(\cdot))$ for our Example 2 that has a reducible input-output pair (A, B)

```
α_0, x, p = n2.expansion()
print(f'α_0 = {α_0}')
print(f'x_0 = {x}')
print(f'The corresponding p from the dual = {p}')
```

```
α_0 = 1.259921052493155
x_0 = [5.27e-10 0.00e+00 3.27e-01 2.60e-01 4.13e-01]
The corresponding p from the dual = [0.   0.21 0.33 0.26 0.21 0.  ]
```

```
β_0, x, p = n2.interest()
print(f'β_0 = {β_0}')
print(f'p_0 = {p}')
print(f'The corresponding x from the primal = {x}')
```

```
β_0 = 1.0000000009313226
p_0 = [ 5.00e-01  5.00e-01 -1.55e-09 -1.24e-09 -9.31e-10  0.00e+00]
The corresponding x from the primal = [-0.    0.    0.25  0.25  0.5 ]
```

As we can see, with a reducible (A, B), the roots found by the bisection algorithms might differ, so there might be multiple γ^* that make the value of the game with $M(\gamma^*)$ zero. (see the figure above).

Indeed, although the von Neumann theorem assures existence of the equilibrium, *Assumption 37.2.1* and *Assumption 37.2.2* are not sufficient for uniqueness. Nonetheless, Kemeny et al. (1967) show that there are at most finitely many economic solutions, meaning that there are only finitely many γ^* that satisfy $V(M(\gamma^*)) = 0$ and $x_0^\top B p_0 > 0$ and that for each such γ_i^*, there is a self-contained part of the economy (a sub-economy) that in equilibrium can expand independently with the expansion coefficient γ_i^*.

The following theorem (see Theorem 9.10. in Gale [Gale, 1989]) asserts that imposing irreducibility is sufficient for uniqueness of (γ^*, x_0, p_0).

Theorem 37.5.1

Adopt the conditions of *Theorem 37.4.1*. If the economy (A, B) is irreducible, then $\gamma^* = \alpha_0 = \beta_0$.

37.5.4 A Special Case

There is a special (A, B) that allows us to simplify the solution method significantly by invoking the powerful Perron-Frobenius theorem for non-negative matrices.

Definition 37.5.2

We call an economy *simple* if it satisfies

- $n = m$
- Each activity produces exactly one good
- Each good is produced by one and only one activity.

These assumptions imply that $B = I_n$, i.e., that B can be written as an identity matrix (possibly after reshuffling its rows and columns).

The simple model has the following special property (Theorem 9.11. in Gale [Gale, 1989]): if x_0 and $\alpha_0 > 0$ solve the TEP with (A, I_n), then

$$x_0^\top = \alpha_0 x_0^\top A \qquad \Leftrightarrow \qquad x_0^\top A = \left(\frac{1}{\alpha_0}\right) x_0^\top$$

The latter shows that $1/\alpha_0$ is a positive eigenvalue of A and x_0 is the corresponding non-negative left eigenvector.

The classic result of *Perron and Frobenius* implies that a non-negative matrix has a non-negative eigenvalue-eigenvector pair.

Moreover, if A is irreducible, then the optimal intensity vector x_0 is positive and *unique* up to multiplication by a positive scalar.

Suppose that A is reducible with k irreducible subsets $S_1, \ldots, S_k$. Let A_i be the submatrix corresponding to S_i and let α_i and β_i be the associated expansion and interest factors, respectively. Then we have

$$\alpha_0 = \max_i \{\alpha_i\} \qquad \text{and} \qquad \beta_0 = \min_i \{\beta_i\}$$

Part VI

Introduction to Dynamics

CHAPTER

THIRTYEIGHT

FINITE MARKOV CHAINS

Contents

- *Finite Markov Chains*
  - *Overview*
  - *Definitions*
  - *Simulation*
  - *Marginal Distributions*
  - *Irreducibility and Aperiodicity*
  - *Stationary Distributions*
  - *Ergodicity*
  - *Computing Expectations*
  - *Exercises*

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install quantecon
```

38.1 Overview

Markov chains are one of the most useful classes of stochastic processes, being

- simple, flexible and supported by many elegant theoretical results
- valuable for building intuition about random dynamic models
- central to quantitative modeling in their own right

You will find them in many of the workhorse models of economics and finance.

In this lecture, we review some of the theory of Markov chains.

We will also introduce some of the high-quality routines for working with Markov chains available in QuantEcon.py.

Prerequisite knowledge is basic probability and linear algebra.

Let's start with some standard imports:

```
import matplotlib.pyplot as plt
import quantecon as qe
import numpy as np
from mpl_toolkits.mplot3d import Axes3D
```

38.2 Definitions

The following concepts are fundamental.

38.2.1 Stochastic Matrices

A **stochastic matrix** (or **Markov matrix**) is an $n \times n$ square matrix P such that

1. each element of P is nonnegative, and
2. each row of P sums to one

Each row of P can be regarded as a probability mass function over n possible outcomes.

It is too not difficult to check[1] that if P is a stochastic matrix, then so is the k-th power P^k for all $k \in \mathbb{N}$.

38.2.2 Markov Chains

There is a close connection between stochastic matrices and Markov chains.

To begin, let S be a finite set with n elements $\{x_1, \ldots, x_n\}$.

The set S is called the **state space** and $x_1, \ldots, x_n$ are the **state values**.

A **Markov chain** $\{X_t\}$ on S is a sequence of random variables on S that have the **Markov property**.

This means that, for any date t and any state $y \in S$,

$$\mathbb{P}\{X_{t+1} = y \,|\, X_t\} = \mathbb{P}\{X_{t+1} = y \,|\, X_t, X_{t-1}, \ldots\} \tag{38.1}$$

In other words, knowing the current state is enough to know probabilities for future states.

In particular, the dynamics of a Markov chain are fully determined by the set of values

$$P(x, y) := \mathbb{P}\{X_{t+1} = y \,|\, X_t = x\} \qquad (x, y \in S) \tag{38.2}$$

By construction,

- $P(x, y)$ is the probability of going from x to y in one unit of time (one step)
- $P(x, \cdot)$ is the conditional distribution of X_{t+1} given $X_t = x$

We can view P as a stochastic matrix where

$$P_{ij} = P(x_i, x_j) \qquad 1 \leq i, j \leq n$$

Going the other way, if we take a stochastic matrix P, we can generate a Markov chain $\{X_t\}$ as follows:

- draw X_0 from a marginal distribution ψ
- for each $t = 0, 1, \ldots$, draw X_{t+1} from $P(X_t, \cdot)$

By construction, the resulting process satisfies (38.2).

[1] Hint: First show that if P and Q are stochastic matrices then so is their product — to check the row sums, try post multiplying by a column vector of ones. Finally, argue that P^n is a stochastic matrix using induction.

38.2.3 Example 1

Consider a worker who, at any given time t, is either unemployed (state 0) or employed (state 1).

Suppose that, over a one month period,

1. An unemployed worker finds a job with probability $\alpha \in (0, 1)$.
2. An employed worker loses her job and becomes unemployed with probability $\beta \in (0, 1)$.

In terms of a Markov model, we have

- $S = \{0, 1\}$
- $P(0, 1) = \alpha$ and $P(1, 0) = \beta$

We can write out the transition probabilities in matrix form as

$$P = \begin{pmatrix} 1 - \alpha & \alpha \\ \beta & 1 - \beta \end{pmatrix} \tag{38.3}$$

Once we have the values α and β, we can address a range of questions, such as

- What is the average duration of unemployment?
- Over the long-run, what fraction of time does a worker find herself unemployed?
- Conditional on employment, what is the probability of becoming unemployed at least once over the next 12 months?

We'll cover such applications below.

38.2.4 Example 2

From US unemployment data, Hamilton [Hamilton, 2005] estimated the stochastic matrix

$$P = \begin{pmatrix} 0.971 & 0.029 & 0 \\ 0.145 & 0.778 & 0.077 \\ 0 & 0.508 & 0.492 \end{pmatrix}$$

where

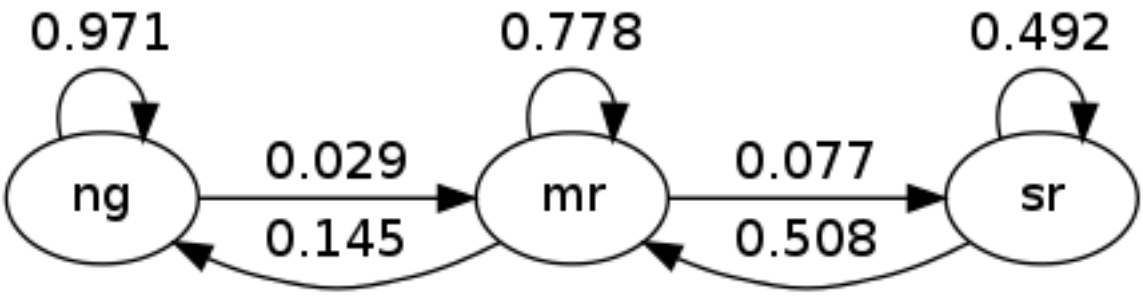
- the frequency is monthly
- the first state represents "normal growth"
- the second state represents "mild recession"
- the third state represents "severe recession"

For example, the matrix tells us that when the state is normal growth, the state will again be normal growth next month with probability 0.97.

In general, large values on the main diagonal indicate persistence in the process $\{X_t\}$.

This Markov process can also be represented as a directed graph, with edges labeled by transition probabilities

Here "ng" is normal growth, "mr" is mild recession, etc.



38.3 Simulation

One natural way to answer questions about Markov chains is to simulate them.

(To approximate the probability of event E, we can simulate many times and count the fraction of times that E occurs).

Nice functionality for simulating Markov chains exists in QuantEcon.py.

- Efficient, bundled with lots of other useful routines for handling Markov chains.

However, it's also a good exercise to roll our own routines — let's do that first and then come back to the methods in QuantEcon.py.

In these exercises, we'll take the state space to be $S = 0, \ldots, n-1$.

38.3.1 Rolling Our Own

To simulate a Markov chain, we need its stochastic matrix P and a marginal probability distribution ψ from which to draw a realization of X_0.

The Markov chain is then constructed as discussed above. To repeat:

1. At time $t = 0$, draw a realization of X_0 from ψ.
2. At each subsequent time t, draw a realization of the new state X_{t+1} from $P(X_t, \cdot)$.

To implement this simulation procedure, we need a method for generating draws from a discrete distribution.

For this task, we'll use `random.draw` from QuantEcon, which works as follows:

```
ψ = (0.3, 0.7)           # probabilities over {0, 1}
cdf = np.cumsum(ψ)       # convert into cummulative distribution
qe.random.draw(cdf, 5)   # generate 5 independent draws from ψ
```

```
array([0, 1, 1, 1, 1])
```

We'll write our code as a function that accepts the following three arguments

- A stochastic matrix `P`
- An initial state `init`
- A positive integer `sample_size` representing the length of the time series the function should return

```
def mc_sample_path(P, ψ_0=None, sample_size=1_000):

    # set up
```

(continues on next page)

(continued from previous page)

```
    P = np.asarray(P)
    X = np.empty(sample_size, dtype=int)

    # Convert each row of P into a cdf
    n = len(P)
    P_dist = [np.cumsum(P[i, :]) for i in range(n)]

    # draw initial state, defaulting to 0
    if ψ_0 is not None:
        X_0 = qe.random.draw(np.cumsum(ψ_0))
    else:
        X_0 = 0

    # simulate
    X[0] = X_0
    for t in range(sample_size - 1):
        X[t+1] = qe.random.draw(P_dist[X[t]])

    return X
```

Let's see how it works using the small matrix

```
P = [[0.4, 0.6],
     [0.2, 0.8]]
```

As we'll see later, for a long series drawn from `P`, the fraction of the sample that takes value 0 will be about 0.25.

Moreover, this is true, regardless of the initial distribution from which X_0 is drawn.

The following code illustrates this

```
X = mc_sample_path(P, ψ_0=[0.1, 0.9], sample_size=100_000)
np.mean(X == 0)
```

```
np.float64(0.24966)
```

You can try changing the initial distribution to confirm that the output is always close to 0.25, at least for the `P` matrix above.

38.3.2 Using QuantEcon's Routines

As discussed above, QuantEcon.py has routines for handling Markov chains, including simulation.

Here's an illustration using the same P as the preceding example

```
from quantecon import MarkovChain

mc = qe.MarkovChain(P)
X = mc.simulate(ts_length=1_000_000)
np.mean(X == 0)
```

```
np.float64(0.249594)
```

The QuantEcon.py routine is JIT compiled and much faster.

```
%time mc_sample_path(P, sample_size=1_000_000) # Our homemade code version
```

```
CPU times: user 2.36 s, sys: 992 µs, total: 2.37 s
Wall time: 2.37 s
```

```
array([0, 1, 1, ..., 1, 1, 1], shape=(1000000,))
```

```
%time mc.simulate(ts_length=1_000_000) # qe code version
```

```
CPU times: user 16.2 ms, sys: 3.01 ms, total: 19.3 ms
Wall time: 19 ms
```

```
array([0, 0, 1, ..., 1, 0, 0], shape=(1000000,))
```

Adding State Values and Initial Conditions

If we wish to, we can provide a specification of state values to `MarkovChain`.

These state values can be integers, floats, or even strings.

The following code illustrates

```
mc = qe.MarkovChain(P, state_values=('unemployed', 'employed'))
mc.simulate(ts_length=4, init='employed')
```

```
array(['employed', 'employed', 'unemployed', 'employed'], dtype='<U10')
```

```
mc.simulate(ts_length=4, init='unemployed')
```

```
array(['unemployed', 'employed', 'employed', 'employed'], dtype='<U10')
```

```
mc.simulate(ts_length=4)  # Start at randomly chosen initial state
```

```
array(['unemployed', 'employed', 'employed', 'employed'], dtype='<U10')
```

If we want to see indices rather than state values as outputs as we can use

```
mc.simulate_indices(ts_length=4)
```

```
array([1, 1, 1, 0])
```

38.4 Marginal Distributions

Suppose that

1. $\{X_t\}$ is a Markov chain with stochastic matrix P
2. the marginal distribution of X_t is known to be ψ_t

What then is the marginal distribution of X_{t+1}, or, more generally, of X_{t+m}?

To answer this, we let ψ_t be the marginal distribution of X_t for $t = 0, 1, 2, ...$

Our first aim is to find ψ_{t+1} given ψ_t and P.

To begin, pick any $y \in S$.

Using the law of total probability, we can decompose the probability that $X_{t+1} = y$ as follows:

$$\mathbb{P}\{X_{t+1} = y\} = \sum_{x \in S} \mathbb{P}\{X_{t+1} = y \,|\, X_t = x\} \cdot \mathbb{P}\{X_t = x\}$$

In words, to get the probability of being at y tomorrow, we account for all ways this can happen and sum their probabilities.

Rewriting this statement in terms of marginal and conditional probabilities gives

$$\psi_{t+1}(y) = \sum_{x \in S} P(x, y)\psi_t(x)$$

There are n such equations, one for each $y \in S$.

If we think of ψ_{t+1} and ψ_t as *row vectors*, these n equations are summarized by the matrix expression

$$\psi_{t+1} = \psi_t P \tag{38.4}$$

Thus, to move a marginal distribution forward one unit of time, we postmultiply by P.

By postmultiplying m times, we move a marginal distribution forward m steps into the future.

Hence, iterating on (38.4), the expression $\psi_{t+m} = \psi_t P^m$ is also valid — here P^m is the m-th power of P.

As a special case, we see that if ψ_0 is the initial distribution from which X_0 is drawn, then $\psi_0 P^m$ is the distribution of X_m.

This is very important, so let's repeat it

$$X_0 \sim \psi_0 \quad \Longrightarrow \quad X_m \sim \psi_0 P^m \tag{38.5}$$

and, more generally,

$$X_t \sim \psi_t \quad \Longrightarrow \quad X_{t+m} \sim \psi_t P^m \tag{38.6}$$

38.4.1 Multiple Step Transition Probabilities

We know that the probability of transitioning from x to y in one step is $P(x, y)$.

It turns out that the probability of transitioning from x to y in m steps is $P^m(x, y)$, the (x, y)-th element of the m-th power of P.

To see why, consider again (38.6), but now with a ψ_t that puts all probability on state x so that the transition probabilities are

- 1 in the x-th position and zero elsewhere

Inserting this into (38.6), we see that, conditional on $X_t = x$, the distribution of X_{t+m} is the x-th row of P^m.

In particular

$$\mathbb{P}\{X_{t+m} = y \,|\, X_t = x\} = P^m(x, y) = (x, y)\text{-th element of } P^m$$

38.4.2 Example: Probability of Recession

Recall the stochastic matrix P for recession and growth *considered above*.

Suppose that the current state is unknown — perhaps statistics are available only at the *end* of the current month.

We guess that the probability that the economy is in state x is $\psi(x)$.

The probability of being in recession (either mild or severe) in 6 months time is given by the inner product

$$\psi P^6 \cdot \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}$$

38.4.3 Example 2: Cross-Sectional Distributions

The marginal distributions we have been studying can be viewed either as probabilities or as cross-sectional frequencies that a Law of Large Numbers leads us to anticipate for large samples.

To illustrate, recall our model of employment/unemployment dynamics for a given worker *discussed above*.

Consider a large population of workers, each of whose lifetime experience is described by the specified dynamics, with each worker's outcomes being realizations of processes that are statistically independent of all other workers' processes.

Let ψ be the current *cross-sectional* distribution over $\{0, 1\}$.

The cross-sectional distribution records fractions of workers employed and unemployed at a given moment.

- For example, $\psi(0)$ is the unemployment rate.

What will the cross-sectional distribution be in 10 periods hence?

The answer is ψP^{10}, where P is the stochastic matrix in (38.3).

This is because each worker's state evolves according to P, so ψP^{10} is a marginal distibution for a single randomly selected worker.

But when the sample is large, outcomes and probabilities are roughly equal (by an application of the Law of Large Numbers).

So for a very large (tending to infinite) population, ψP^{10} also represents fractions of workers in each state.

This is exactly the cross-sectional distribution.

38.5 Irreducibility and Aperiodicity

Irreducibility and aperiodicity are central concepts of modern Markov chain theory.

Let's see what they're about.

38.5.1 Irreducibility

Let P be a fixed stochastic matrix.

Two states x and y are said to **communicate** with each other if there exist positive integers j and k such that

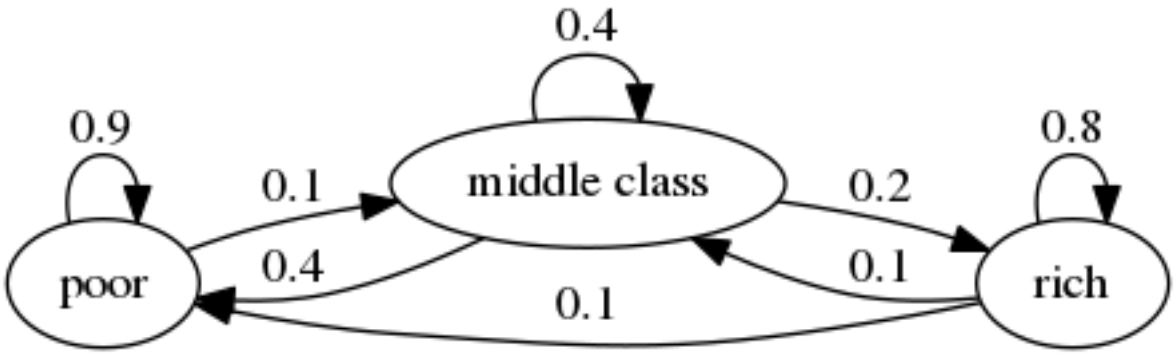
$$P^j(x, y) > 0 \quad \text{and} \quad P^k(y, x) > 0$$

In view of our discussion *above*, this means precisely that

- state x can eventually be reached from state y, and
- state y can eventually be reached from state x

The stochastic matrix P is called **irreducible** if all states communicate; that is, if x and y communicate for all (x, y) in $S \times S$.

For example, consider the following transition probabilities for wealth of a fictitious set of households



We can translate this into a stochastic matrix, putting zeros where there's no edge between nodes

$$P := \begin{pmatrix} 0.9 & 0.1 & 0 \\ 0.4 & 0.4 & 0.2 \\ 0.1 & 0.1 & 0.8 \end{pmatrix}$$

It's clear from the graph that this stochastic matrix is irreducible: we can eventually reach any state from any other state.

We can also test this using QuantEcon.py's MarkovChain class

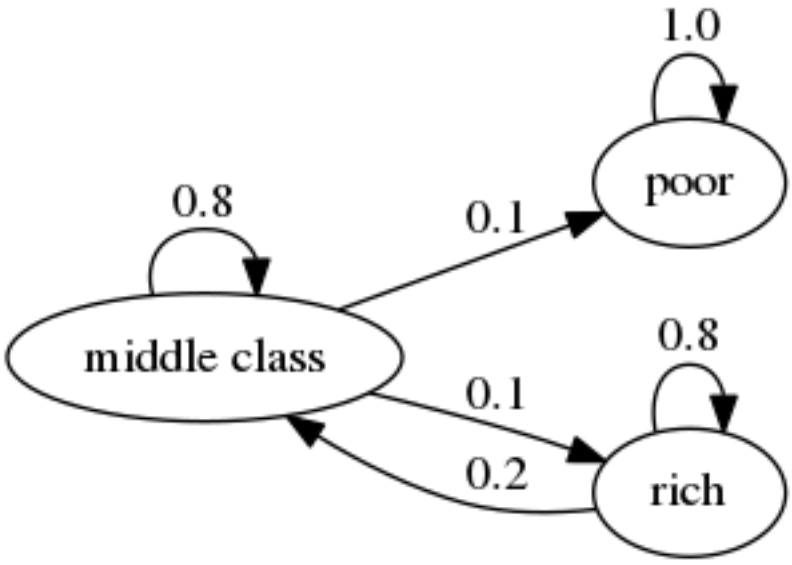
```
P = [[0.9, 0.1, 0.0],
     [0.4, 0.4, 0.2],
     [0.1, 0.1, 0.8]]

mc = qe.MarkovChain(P, ('poor', 'middle', 'rich'))
mc.is_irreducible
```

```
True
```

Here's a more pessimistic scenario in which poor people remain poor forever

This stochastic matrix is not irreducible, since, for example, rich is not accessible from poor.

Let's confirm this




```
P = [[1.0, 0.0, 0.0],
     [0.1, 0.8, 0.1],
     [0.0, 0.2, 0.8]]

mc = qe.MarkovChain(P, ('poor', 'middle', 'rich'))
mc.is_irreducible
```

```
False
```

We can also determine the "communication classes"

```
mc.communication_classes
```

```
[array(['poor'], dtype='<U6'), array(['middle', 'rich'], dtype='<U6')]
```

It might be clear to you already that irreducibility is going to be important in terms of long run outcomes.

For example, poverty is a life sentence in the second graph but not the first.

We'll come back to this a bit later.

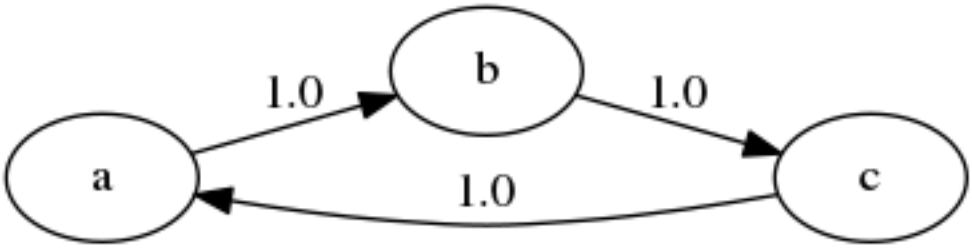
38.5.2 Aperiodicity

Loosely speaking, a Markov chain is called **periodic** if it cycles in a predictable way, and **aperiodic** otherwise.

Here's a trivial example with three states

The chain cycles with period 3:

```
P = [[0, 1, 0],
     [0, 0, 1],
     [1, 0, 0]]

mc = qe.MarkovChain(P)
mc.period
```




```
3
```

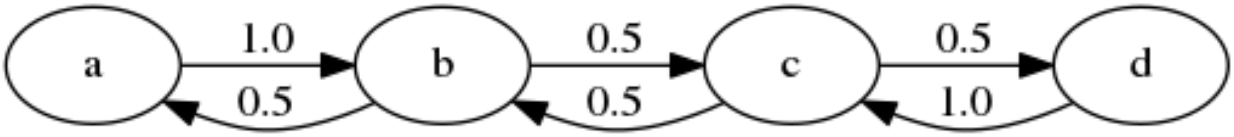
More formally, the **period** of a state x is the largest common divisor of a set of integers

$$D(x) := \{j \geq 1 : P^j(x, x) > 0\}$$

In the last example, $D(x) = \{3, 6, 9, ...\}$ for every state x, so the period is 3.

A stochastic matrix is called **aperiodic** if the period of every state is 1, and **periodic** otherwise.

For example, the stochastic matrix associated with the transition probabilities below is periodic because, for example, state a has period 2



We can confirm that the stochastic matrix is periodic with the following code

```
P = [[0.0, 1.0, 0.0, 0.0],
     [0.5, 0.0, 0.5, 0.0],
     [0.0, 0.5, 0.0, 0.5],
     [0.0, 0.0, 1.0, 0.0]]

mc = qe.MarkovChain(P)
mc.period
```

```
2
```

```
mc.is_aperiodic
```

```
False
```

38.6 Stationary Distributions

As seen in (38.4), we can shift a marginal distribution forward one unit of time via postmultiplication by P.

Some distributions are invariant under this updating process — for example,

```
P = np.array([[0.4, 0.6],
              [0.2, 0.8]])
```

(continues on next page)

(continued from previous page)

```
ψ = (0.25, 0.75)
ψ @ P
```

```
array([0.25, 0.75])
```

Such distributions are called **stationary** or **invariant**.

Formally, a marginal distribution ψ^* on S is called **stationary** for P if $\psi^* = \psi^* P$.

(This is the same notion of stationarity that we learned about in the lecture on AR(1) processes applied to a different setting.)

From this equality, we immediately get $\psi^* = \psi^* P^t$ for all t.

This tells us an important fact: If the distribution of X_0 is a stationary distribution, then X_t will have this same distribution for all t.

Hence stationary distributions have a natural interpretation as **stochastic steady states** — we'll discuss this more soon.

Mathematically, a stationary distribution is a fixed point of P when P is thought of as the map $\psi \mapsto \psi P$ from (row) vectors to (row) vectors.

Theorem. Every stochastic matrix P has at least one stationary distribution.

(We are assuming here that the state space S is finite; if not more assumptions are required)

For proof of this result, you can apply Brouwer's fixed point theorem, or see EDTC, theorem 4.3.5.

There can be many stationary distributions corresponding to a given stochastic matrix P.

- For example, if P is the identity matrix, then all marginal distributions are stationary.

To get uniqueness an invariant distribution, the transition matrix P must have the property that no nontrivial subsets of the state space are **infinitely persistent**.

A subset of the state space is infinitely persistent if other parts of the state space cannot be accessed from it.

Thus, infinite persistence of a non-trivial subset is the opposite of irreducibility.

This gives some intuition for the following fundamental theorem.

Theorem. If P is both aperiodic and irreducible, then

1. P has exactly one stationary distribution ψ^*.
2. For any initial marginal distribution ψ_0, we have $\|\psi_0 P^t - \psi^*\| \to 0$ as $t \to \infty$.

For a proof, see, for example, theorem 5.2 of [Häggström, 2002].

(Note that part 1 of the theorem only requires irreducibility, whereas part 2 requires both irreducibility and aperiodicity)

A stochastic matrix that satisfies the conditions of the theorem is sometimes called **uniformly ergodic**.

A sufficient condition for aperiodicity and irreducibility is that every element of P is strictly positive.

- Try to convince yourself of this.

38.6.1 Example

Recall our model of the employment/unemployment dynamics of a particular worker *discussed above*.

Assuming $\alpha \in (0, 1)$ and $\beta \in (0, 1)$, the uniform ergodicity condition is satisfied.

Let $\psi^* = (p, 1 - p)$ be the stationary distribution, so that p corresponds to unemployment (state 0).

Using $\psi^* = \psi^* P$ and a bit of algebra yields

$$p = \frac{\beta}{\alpha + \beta}$$

This is, in some sense, a steady state probability of unemployment — more about the interpretation of this below.

Not surprisingly it tends to zero as $\beta \to 0$, and to one as $\alpha \to 0$.

38.6.2 Calculating Stationary Distributions

As discussed above, a particular Markov matrix P can have many stationary distributions.

That is, there can be many row vectors ψ such that $\psi = \psi P$.

In fact if P has two distinct stationary distributions ψ_1, ψ_2 then it has infinitely many, since in this case, as you can verify, for any $\lambda \in [0, 1]$

$$\psi_3 := \lambda \psi_1 + (1 - \lambda) \psi_2$$

is a stationary distribution for P.

If we restrict attention to the case in which only one stationary distribution exists, one way to finding it is to solve the system

$$\psi (I_n - P) = 0 \tag{38.7}$$

for ψ, where I_n is the $n \times n$ identity.

But the zero vector solves system (38.7), so we must proceed cautiously.

We want to impose the restriction that ψ is a probability distribution.

There are various ways to do this.

One option is to regard solving system (38.7) as an eigenvector problem: a vector ψ such that $\psi = \psi P$ is a left eigenvector associated with the unit eigenvalue $\lambda = 1$.

A stable and sophisticated algorithm specialized for stochastic matrices is implemented in QuantEcon.py.

This is the one we recommend:

```
P = [[0.4, 0.6],
     [0.2, 0.8]]

mc = qe.MarkovChain(P)
mc.stationary_distributions  # Show all stationary distributions
```

```
array([[0.25, 0.75]])
```

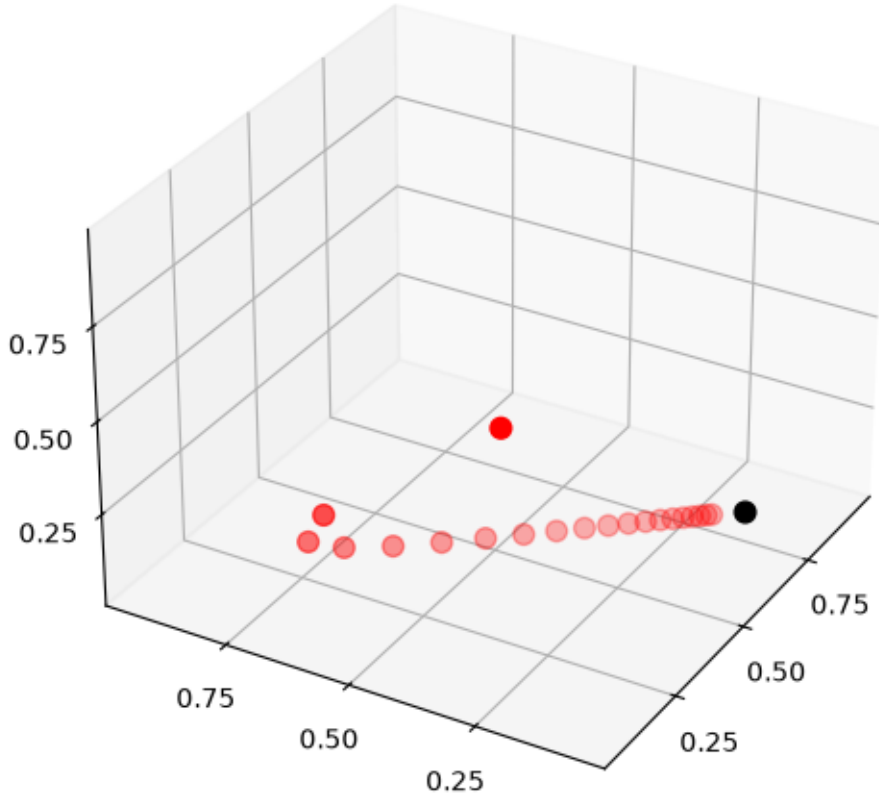
38.6.3 Convergence to Stationarity

Part 2 of the Markov chain convergence theorem *stated above* tells us that the marginal distribution of X_t converges to the stationary distribution regardless of where we begin.

This adds considerable authority to our interpretation of ψ^* as a stochastic steady state.

The convergence in the theorem is illustrated in the next figure

```
P = ((0.971, 0.029, 0.000),
     (0.145, 0.778, 0.077),
     (0.000, 0.508, 0.492))
P = np.array(P)

ψ = (0.0, 0.2, 0.8)        # Initial condition

fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111, projection='3d')

ax.set(xlim=(0, 1), ylim=(0, 1), zlim=(0, 1),
       xticks=(0.25, 0.5, 0.75),
       yticks=(0.25, 0.5, 0.75),
       zticks=(0.25, 0.5, 0.75))

x_vals, y_vals, z_vals = [], [], []
for t in range(20):
    x_vals.append(ψ[0])
    y_vals.append(ψ[1])
    z_vals.append(ψ[2])
    ψ = ψ @ P

ax.scatter(x_vals, y_vals, z_vals, c='r', s=60)
ax.view_init(30, 210)

mc = qe.MarkovChain(P)
ψ_star = mc.stationary_distributions[0]
ax.scatter(ψ_star[0], ψ_star[1], ψ_star[2], c='k', s=60)

plt.show()
```



Here

- P is the stochastic matrix for recession and growth *considered above*.
- The highest red dot is an arbitrarily chosen initial marginal probability distribution ψ, represented as a vector in $\mathbb{R}^3$.
- The other red dots are the marginal distributions ψP^t for $t = 1, 2, ...$
- The black dot is ψ^*.

You might like to try experimenting with different initial conditions.

38.7 Ergodicity

Under irreducibility, yet another important result obtains: for all $x \in S$,

$$\frac{1}{m} \sum_{t=1}^{m} \mathbf{1}\{X_t = x\} \to \psi^*(x) \quad \text{as } m \to \infty \tag{38.8}$$

Here

- $\mathbf{1}\{X_t = x\} = 1$ if $X_t = x$ and zero otherwise
- convergence is with probability one
- the result does not depend on the marginal distribution of X_0

The result tells us that the fraction of time the chain spends at state x converges to $\psi^*(x)$ as time goes to infinity.

This gives us another way to interpret the stationary distribution — provided that the convergence result in (38.8) is valid.

The convergence asserted in (38.8) is a special case of a law of large numbers result for Markov chains — see EDTC, section 4.3.4 for some additional information.

38.7.1 Example

Recall our cross-sectional interpretation of the employment/unemployment model *discussed above*.

Assume that $\alpha \in (0, 1)$ and $\beta \in (0, 1)$, so that irreducibility and aperiodicity both hold.

We saw that the stationary distribution is $(p, 1 - p)$, where

$$p = \frac{\beta}{\alpha + \beta}$$

In the cross-sectional interpretation, this is the fraction of people unemployed.

In view of our latest (ergodicity) result, it is also the fraction of time that a single worker can expect to spend unemployed.

Thus, in the long-run, cross-sectional averages for a population and time-series averages for a given person coincide.

This is one aspect of the concept of ergodicity.

38.8 Computing Expectations

We sometimes want to compute mathematical expectations of functions of X_t of the form

$$\mathbb{E}[h(X_t)] \tag{38.9}$$

and conditional expectations such as

$$\mathbb{E}[h(X_{t+k}) \mid X_t = x] \tag{38.10}$$

where

- $\{X_t\}$ is a Markov chain generated by $n \times n$ stochastic matrix P
- h is a given function, which, in terms of matrix algebra, we'll think of as the column vector

$$h = \begin{pmatrix} h(x_1) \\ \vdots \\ h(x_n) \end{pmatrix}$$

Computing the unconditional expectation (38.9) is easy.

We just sum over the marginal distribution of X_t to get

$$\mathbb{E}[h(X_t)] = \sum_{x \in S} (\psi P^t)(x) h(x)$$

Here ψ is the distribution of X_0.

Since ψ and hence ψP^t are row vectors, we can also write this as

$$\mathbb{E}[h(X_t)] = \psi P^t h$$

For the conditional expectation (38.10), we need to sum over the conditional distribution of X_{t+k} given $X_t = x$.

We already know that this is $P^k(x, \cdot)$, so

$$\mathbb{E}[h(X_{t+k}) \mid X_t = x] = (P^k h)(x) \tag{38.11}$$

The vector $P^k h$ stores the conditional expectation $\mathbb{E}[h(X_{t+k}) \mid X_t = x]$ over all x.

38.8.1 Iterated Expectations

The **law of iterated expectations** states that

$$\mathbb{E}\left[\mathbb{E}[h(X_{t+k}) \mid X_t = x]\right] = \mathbb{E}[h(X_{t+k})]$$

where the outer $\mathbb{E}$ on the left side is an unconditional distribution taken with respect to the marginal distribution ψ_t of X_t (again see equation (38.6)).

To verify the law of iterated expectations, use equation (38.11) to substitute $(P^k h)(x)$ for $E[h(X_{t+k}) \mid X_t = x]$, write

$$\mathbb{E}\left[\mathbb{E}[h(X_{t+k}) \mid X_t = x]\right] = \psi_t P^k h,$$

and note $\psi_t P^k h = \psi_{t+k} h = \mathbb{E}[h(X_{t+k})]$.

38.8.2 Expectations of Geometric Sums

Sometimes we want to compute the mathematical expectation of a geometric sum, such as $\sum_t \beta^t h(X_t)$.

In view of the preceding discussion, this is

$$\mathbb{E}\Big[\sum_{j=0}^{\infty} \beta^j h(X_{t+j}) \mid X_t = x\Big] = [(I - \beta P)^{-1} h](x)$$

where

$$(I - \beta P)^{-1} = I + \beta P + \beta^2 P^2 + \cdots$$

Premultiplication by $(I - \beta P)^{-1}$ amounts to "applying the **resolvent operator**".

38.9 Exercises

Exercise 38.9.1

According to the discussion *above*, if a worker's employment dynamics obey the stochastic matrix

$$P = \begin{pmatrix} 1 - \alpha & \alpha \\ \beta & 1 - \beta \end{pmatrix}$$

with $\alpha \in (0, 1)$ and $\beta \in (0, 1)$, then, in the long-run, the fraction of time spent unemployed will be

$$p := \frac{\beta}{\alpha + \beta}$$

In other words, if $\{X_t\}$ represents the Markov chain for employment, then $\bar{X}_m \to p$ as $m \to \infty$, where

$$\bar{X}_m := \frac{1}{m} \sum_{t=1}^{m} \mathbf{1}\{X_t = 0\}$$

This exercise asks you to illustrate convergence by computing $\bar{X}_m$ for large m and checking that it is close to p.

You will see that this statement is true regardless of the choice of initial condition or the values of α, β, provided both lie in $(0, 1)$.

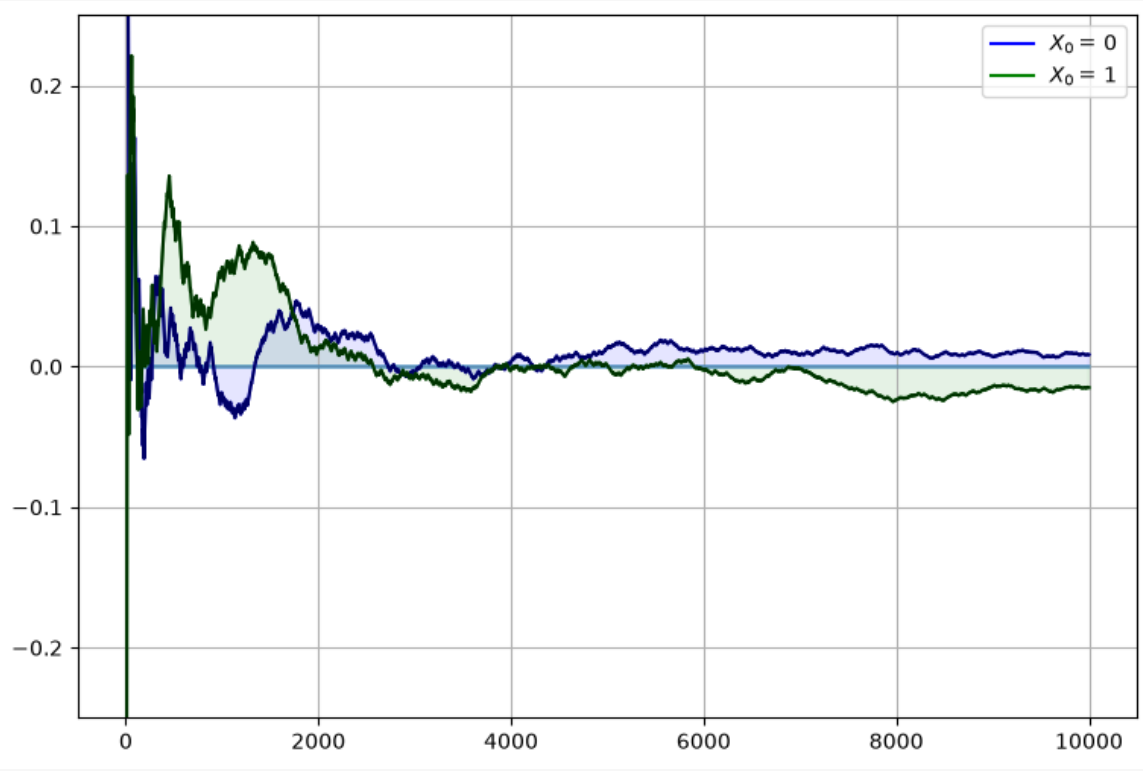
Solution

We will address this exercise graphically.

The plots show the time series of $\bar{X}_m - p$ for two initial conditions.

As m gets large, both series converge to zero.

```
α = β = 0.1
N = 10000
p = β / (α + β)

P = ((1 - α,        α),               # Careful: P and p are distinct
     (    β,    1 - β))
mc = MarkovChain(P)

fig, ax = plt.subplots(figsize=(9, 6))
ax.set_ylim(-0.25, 0.25)
ax.grid()
ax.hlines(0, 0, N, lw=2, alpha=0.6)   # Horizonal line at zero

for x0, col in ((0, 'blue'), (1, 'green')):
    # Generate time series for worker that starts at x0
    X = mc.simulate(N, init=x0)
    # Compute fraction of time spent unemployed, for each n
    X_bar = (X == 0).cumsum() / (1 + np.arange(N, dtype=float))
    # Plot
    ax.fill_between(range(N), np.zeros(N), X_bar - p, color=col, alpha=0.1)
    ax.plot(X_bar - p, color=col, label=fr'$X_0 = \, {x0} $')
    # Overlay in black--make lines clearer
    ax.plot(X_bar - p, 'k-', alpha=0.6)

ax.legend(loc='upper right')
plt.show()
```



Exercise 38.9.2

A topic of interest for economics and many other disciplines is *ranking*.

Let's now consider one of the most practical and important ranking problems — the rank assigned to web pages by search engines.

(Although the problem is motivated from outside of economics, there is in fact a deep connection between search ranking systems and prices in certain competitive equilibria — see [Du *et al.*, 2013].)

To understand the issue, consider the set of results returned by a query to a web search engine.

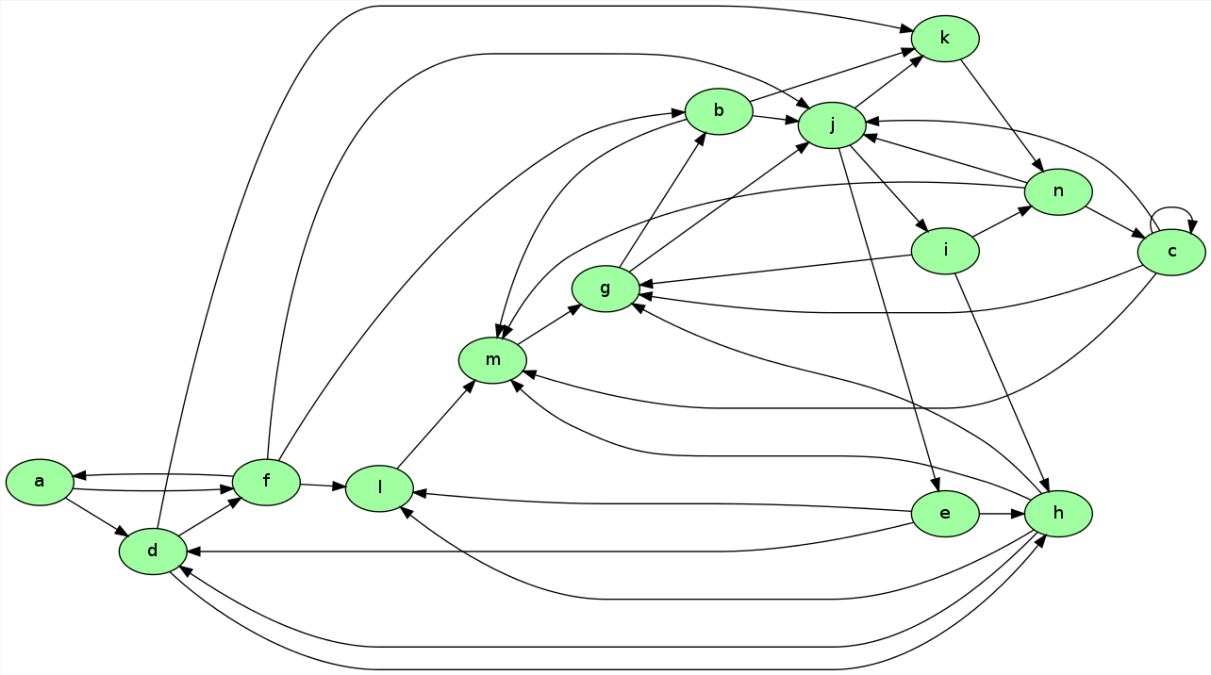
For the user, it is desirable to

1. receive a large set of accurate matches
2. have the matches returned in order, where the order corresponds to some measure of "importance"

Ranking according to a measure of importance is the problem we now consider.

The methodology developed to solve this problem by Google founders Larry Page and Sergey Brin is known as PageRank.

To illustrate the idea, consider the following diagram



Imagine that this is a miniature version of the WWW, with

- each node representing a web page
- each arrow representing the existence of a link from one page to another

Now let's think about which pages are likely to be important, in the sense of being valuable to a search engine user.

One possible criterion for the importance of a page is the number of inbound links — an indication of popularity.

By this measure, `m` and `j` are the most important pages, with 5 inbound links each.

However, what if the pages linking to `m`, say, are not themselves important?

Thinking this way, it seems appropriate to weight the inbound nodes by relative importance.

The PageRank algorithm does precisely this.

A slightly simplified presentation that captures the basic idea is as follows.

Letting j be (the integer index of) a typical page and r_j be its ranking, we set

$$r_j = \sum_{i \in L_j} \frac{r_i}{\ell_i}$$

where

- ℓ_i is the total number of outbound links from i
- L_j is the set of all pages i such that i has a link to j

This is a measure of the number of inbound links, weighted by their own ranking (and normalized by $1/\ell_i$).

There is, however, another interpretation, and it brings us back to Markov chains.

Let P be the matrix given by $P(i, j) = \mathbf{1}\{i \to j\}/\ell_i$ where $\mathbf{1}\{i \to j\} = 1$ if i has a link to j and zero otherwise.

The matrix P is a stochastic matrix provided that each page has at least one link.

With this definition of P we have

$$r_j = \sum_{i \in L_j} \frac{r_i}{\ell_i} = \sum_{\text{all } i} \mathbf{1}\{i \to j\} \frac{r_i}{\ell_i} = \sum_{\text{all } i} P(i,j) r_i$$

Writing r for the row vector of rankings, this becomes $r = rP$.

Hence r is the stationary distribution of the stochastic matrix P.

Let's think of $P(i,j)$ as the probability of "moving" from page i to page j.

The value $P(i,j)$ has the interpretation

- $P(i,j) = 1/k$ if i has k outbound links and j is one of them
- $P(i,j) = 0$ if i has no direct link to j

Thus, motion from page to page is that of a web surfer who moves from one page to another by randomly clicking on one of the links on that page.

Here "random" means that each link is selected with equal probability.

Since r is the stationary distribution of P, assuming that the uniform ergodicity condition is valid, we *can interpret* r_j as the fraction of time that a (very persistent) random surfer spends at page j.

Your exercise is to apply this ranking algorithm to the graph pictured above and return the list of pages ordered by rank.

There is a total of 14 nodes (i.e., web pages), the first named `a` and the last named `n`.

A typical line from the file has the form

```
d -> h;
```

This should be interpreted as meaning that there exists a link from `d` to `h`.

The data for this graph is shown below, and read into a file called `web_graph_data.txt` when the cell is executed.

```
%%file web_graph_data.txt
a -> d;
a -> f;
b -> j;
b -> k;
b -> m;
c -> c;
c -> g;
c -> j;
c -> m;
d -> f;
d -> h;
d -> k;
e -> d;
e -> h;
e -> l;
f -> a;
f -> b;
f -> j;
f -> l;
g -> b;
g -> j;
h -> d;
h -> g;
h -> l;
```

```
h -> m;
i -> g;
i -> h;
i -> n;
j -> e;
j -> i;
j -> k;
k -> n;
l -> m;
m -> g;
n -> c;
n -> j;
n -> m;
```

```
Overwriting web_graph_data.txt
```

To parse this file and extract the relevant information, you can use regular expressions.

The following code snippet provides a hint as to how you can go about this

```
import re
re.findall(r'\w', 'x +++ y ****** z')  # \w matches alphanumerics
```

```
['x', 'y', 'z']
```

```
re.findall(r'\w', 'a ^^ b &&& $$ c')
```

```
['a', 'b', 'c']
```

When you solve for the ranking, you will find that the highest ranked node is in fact `g`, while the lowest is `a`.

Solution

Here is one solution:

```
"""
Return list of pages, ordered by rank
"""
import re
from operator import itemgetter

infile = 'web_graph_data.txt'
alphabet = 'abcdefghijklmnopqrstuvwxyz'

n = 14 # Total number of web pages (nodes)

# Create a matrix Q indicating existence of links
#  * Q[i, j] = 1 if there is a link from i to j
#  * Q[i, j] = 0 otherwise
Q = np.zeros((n, n), dtype=int)
with open(infile) as f:
    edges = f.readlines()
for edge in edges:
    from_node, to_node = re.findall(r'\w', edge)
    i, j = alphabet.index(from_node), alphabet.index(to_node)
    Q[i, j] = 1
# Create the corresponding Markov matrix P
P = np.empty((n, n))
```

```
for i in range(n):
    P[i, :] = Q[i, :] / Q[i, :].sum()
mc = MarkovChain(P)
# Compute the stationary distribution r
r = mc.stationary_distributions[0]
ranked_pages = {alphabet[i] : r[i] for i in range(n)}
# Print solution, sorted from highest to lowest rank
print('Rankings\n ***')
for name, rank in sorted(ranked_pages.items(), key=itemgetter(1), reverse=1):
    print(f'{name}: {rank:.4}')
```

```
Rankings
 ***
g: 0.1607
j: 0.1594
m: 0.1195
n: 0.1088
k: 0.09106
b: 0.08326
e: 0.05312
i: 0.05312
c: 0.04834
h: 0.0456
l: 0.03202
d: 0.03056
f: 0.01164
a: 0.002911
```

Exercise 38.9.3

In numerical work, it is sometimes convenient to replace a continuous model with a discrete one.

In particular, Markov chains are routinely generated as discrete approximations to AR(1) processes of the form

$$y_{t+1} = \rho y_t + u_{t+1}$$

Here u_t is assumed to be IID and $N(0, \sigma_u^2)$.

The variance of the stationary probability distribution of $\{y_t\}$ is

$$\sigma_y^2 := \frac{\sigma_u^2}{1 - \rho^2}$$

Tauchen's method [Tauchen, 1986] is the most common method for approximating this continuous state process with a finite state Markov chain.

A routine for this already exists in QuantEcon.py but let's write our own version as an exercise.

As a first step, we choose

- n, the number of states for the discrete approximation
- m, an integer that parameterizes the width of the state space

Next, we create a state space $\{x_0, \ldots, x_{n-1}\} \subset \mathbb{R}$ and a stochastic $n \times n$ matrix P such that

- $x_0 = -m\,\sigma_y$
- $x_{n-1} = m\,\sigma_y$

- $x_{i+1} = x_i + s$ where $s = (x_{n-1} - x_0)/(n-1)$

Let F be the cumulative distribution function of the normal distribution $N(0, \sigma_u^2)$.

The values $P(x_i, x_j)$ are computed to approximate the AR(1) process — omitting the derivation, the rules are as follows:

1. If $j = 0$, then set

$$P(x_i, x_j) = P(x_i, x_0) = F(x_0 - \rho x_i + s/2)$$

2. If $j = n - 1$, then set

$$P(x_i, x_j) = P(x_i, x_{n-1}) = 1 - F(x_{n-1} - \rho x_i - s/2)$$

3. Otherwise, set

$$P(x_i, x_j) = F(x_j - \rho x_i + s/2) - F(x_j - \rho x_i - s/2)$$

The exercise is to write a function `approx_markov(rho, sigma_u, m=3, n=7)` that returns $\{x_0, \ldots, x_{n-1}\} \subset \mathbb{R}$ and $n \times n$ matrix P as described above.

- Even better, write a function that returns an instance of QuantEcon.py's MarkovChain class.

Solution

A solution from the QuantEcon.py library can be found here.

CHAPTER

THIRTYNINE

INVENTORY DYNAMICS

Contents

- *Inventory Dynamics*
  - *Overview*
  - *Sample paths*
  - *Marginal distributions*
  - *Exercises*

39.1 Overview

In this lecture, we will study the time path of inventories for firms that follow so-called s-S inventory dynamics.

Such firms

1. wait until inventory falls below some level s and then
2. order sufficient quantities to bring their inventory back up to capacity S.

These kinds of policies are common in practice and are also optimal in certain circumstances.

A review of early literature and some macroeconomic implications can be found in [Caplin, 1985].

Here, our main aim is to learn more about simulation, time series, and Markov dynamics.

While our Markov environment and many of the concepts we consider are related to those found in our *lecture on finite Markov chains*, the state space is a continuum in the current application.

Let's start with some imports

```
from functools import partial
from typing import NamedTuple
import jax
import jax.numpy as jnp
from jax import random
import matplotlib.pyplot as plt
from sklearn.neighbors import KernelDensity
```

39.2 Sample paths

Consider a firm with inventory X_t.

The firm waits until $X_t \leq s$ and then restocks up to S units.

It faces stochastic demand $\{D_t\}$, which we assume is IID.

With notation $a^+ := \max\{a, 0\}$, inventory dynamics can be written as

$$X_{t+1} = \begin{cases} (S - D_{t+1})^+ & \text{if } X_t \leq s \\ (X_t - D_{t+1})^+ & \text{if } X_t > s \end{cases}$$

In what follows, we will assume that each D_t is lognormal, so that

$$D_t = \exp(\mu + \sigma Z_t)$$

where μ and σ are parameters and $\{Z_t\}$ is IID and standard normal.

Here's a class that stores parameters and generates time paths for inventory.

```
class Firm(NamedTuple):
    s: int    # restock trigger level
    S: int    # capacity
    μ: float  # shock location parameter
    σ: float  # shock scale parameter
```

```
@partial(jax.jit, static_argnames="sim_length")
def sim_inventory_path(firm, x_init, key, sim_length):
    """
    Simulate an inventory path of length sim_length from a single key.

    A fresh shock is generated each period by folding the period index
    into key, so callers pass one key per path rather than a whole array.

    Args:
        firm: Firm object
        x_init: Initial inventory level
        key: A single JAX random key
        sim_length: Number of periods to simulate

    Returns:
        Array of inventory levels [X_0, X_1, ..., X_{sim_length-1}]
    """

    def update(t, X):
        x = X[t - 1]
        Z = random.normal(random.fold_in(key, t))   # fresh shock for period t
        D = jnp.exp(firm.μ + firm.σ * Z)
        x_new = jnp.where(
            x <= firm.s,
            jnp.maximum(firm.S - D, 0.0),   # restock to S, then meet demand
            jnp.maximum(x - D, 0.0),        # just meet demand
        )
        return X.at[t].set(x_new)

    X = jnp.zeros(sim_length).at[0].set(x_init)
    return jax.lax.fori_loop(1, sim_length, update, X)
```

Note

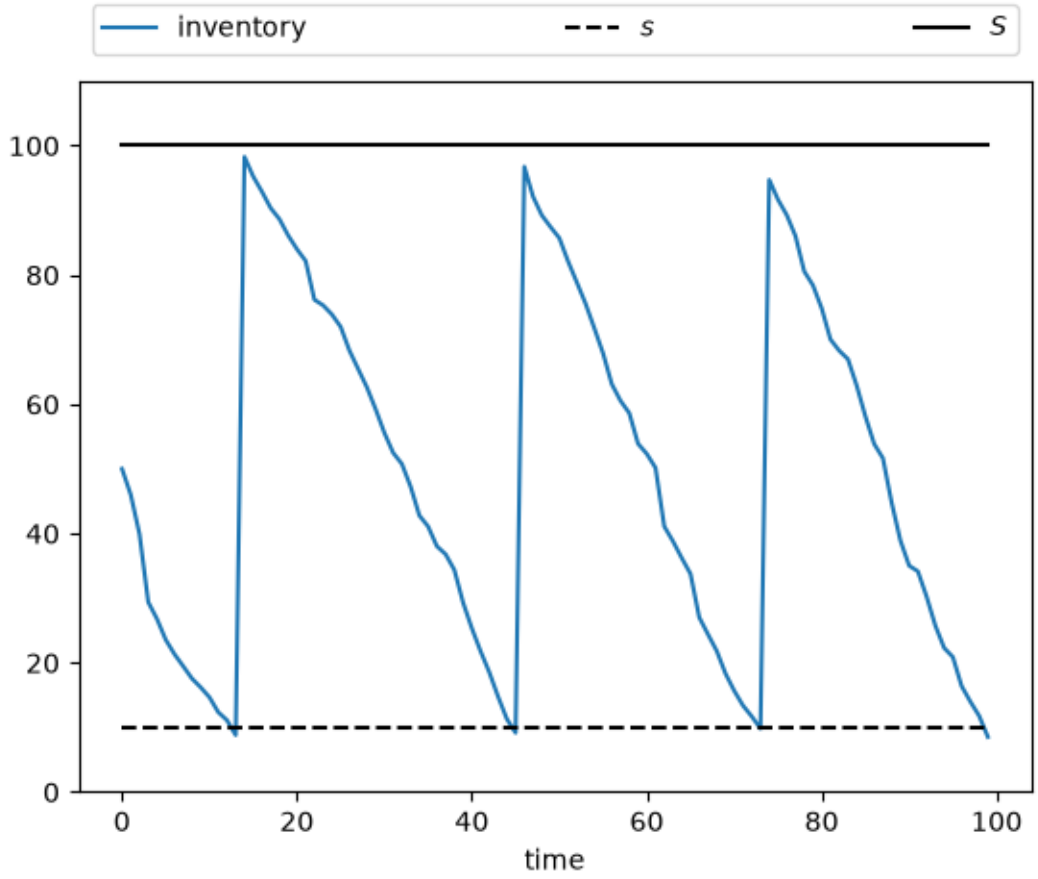
Writing `X.at[t].set(x_new)` returns a *new* array rather than mutating `X` in place, in keeping with JAX's functional style. This is not as wasteful as it looks: inside a `jax.jit`-compiled function the XLA compiler sees that the old array is no longer needed and performs the update in place, so no fresh array is allocated each period.

```
firm = Firm(s=10, S=100, μ=1.0, σ=0.5)
```

```
sim_length = 100
x_init = 50
X = sim_inventory_path(firm, x_init, random.key(21), sim_length)
```

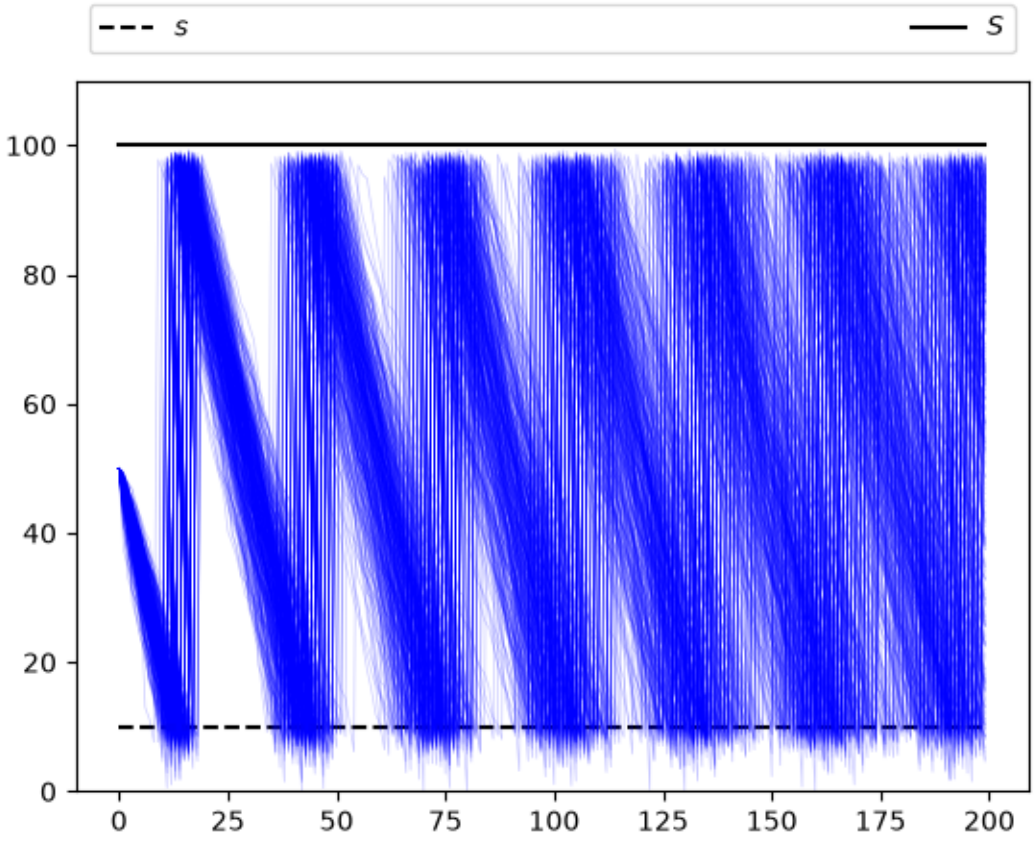
Let's run a first simulation, of a single path:

```
s, S = firm.s, firm.S

fig, ax = plt.subplots()
bbox = (0.0, 1.02, 1.0, 0.102)
legend_args = {"ncol": 3, "bbox_to_anchor": bbox, "loc": 3, "mode": "expand"}

ax.plot(X, label="inventory")
ax.plot(jnp.full(sim_length, s), "k--", label="$s$")
ax.plot(jnp.full(sim_length, S), "k-", label="$S$")
ax.set_ylim(0, S + 10)
ax.set_xlabel("time")
ax.legend(**legend_args)

plt.show()
```



Now let's simulate multiple paths in order to build a more complete picture of the probabilities of different outcomes:

```
sim_length = 200
fig, ax = plt.subplots()

ax.plot(jnp.full(sim_length, s), "k--", label="$s$")
ax.plot(jnp.full(sim_length, S), "k-", label="$S$")
ax.set_ylim(0, S + 10)
ax.legend(**legend_args)

for i in range(400):
    X = sim_inventory_path(firm, x_init, random.key(i), sim_length)
    ax.plot(X, "b", alpha=0.2, lw=0.5)

plt.show()
```



39.3 Marginal distributions

Now let's look at the marginal distribution ψ_T of X_T for some fixed T.

We will do this by generating many draws of X_T given initial condition X_0.

With these draws of X_T we can build up a picture of its distribution ψ_T.

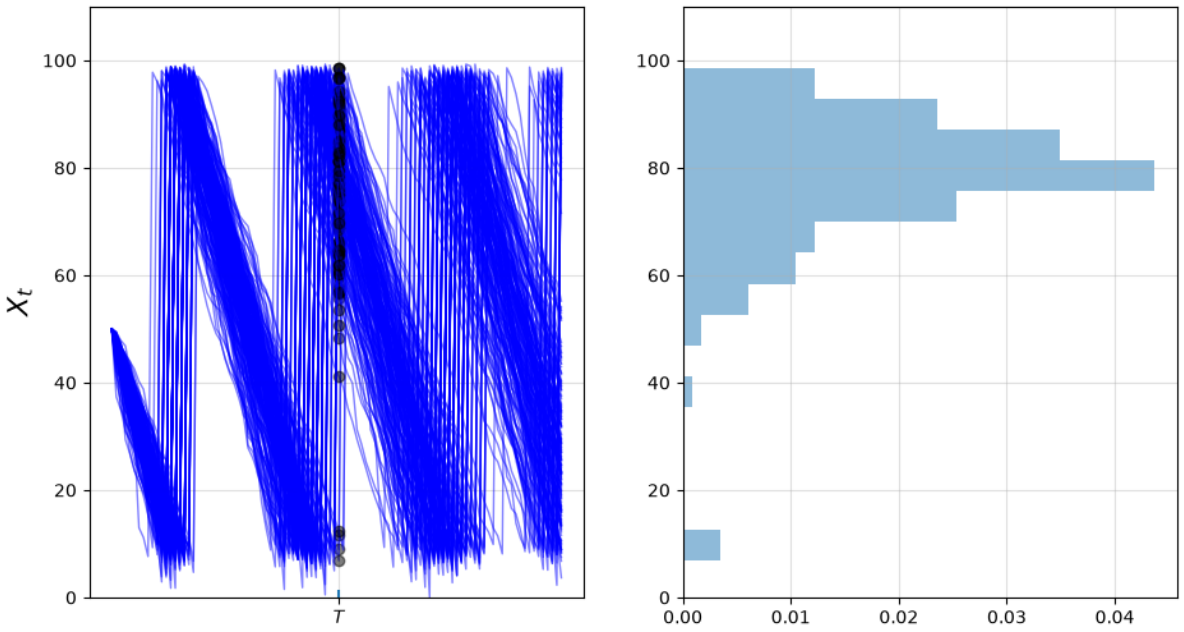
Here's one visualization, with $T = 50$.

```
T = 50
M = 200  # Number of draws

ymin, ymax = 0, S + 10

fig, axes = plt.subplots(1, 2, figsize=(11, 6))

for ax in axes:
    ax.grid(alpha=0.4)

ax = axes[0]

ax.set_ylim(ymin, ymax)
ax.set_ylabel("$X_t$", fontsize=16)
ax.vlines((T,), -1.5, 1.5)
```

(continues on next page)

(continued from previous page)

```
ax.set_xticks((T,))
ax.set_xticklabels((r"$T$",))

sample = []
for m in range(M):
    X = sim_inventory_path(firm, x_init, random.key(m), 2 * T)
    ax.plot(X, "b-", lw=1, alpha=0.5)
    ax.plot((T,), (X[T],), "ko", alpha=0.5)
    sample.append(X[T])

axes[1].set_ylim(ymin, ymax)

axes[1].hist(
    sample,
    bins=16,
    density=True,
    orientation="horizontal",
    histtype="bar",
    alpha=0.5,
)

plt.show()
```



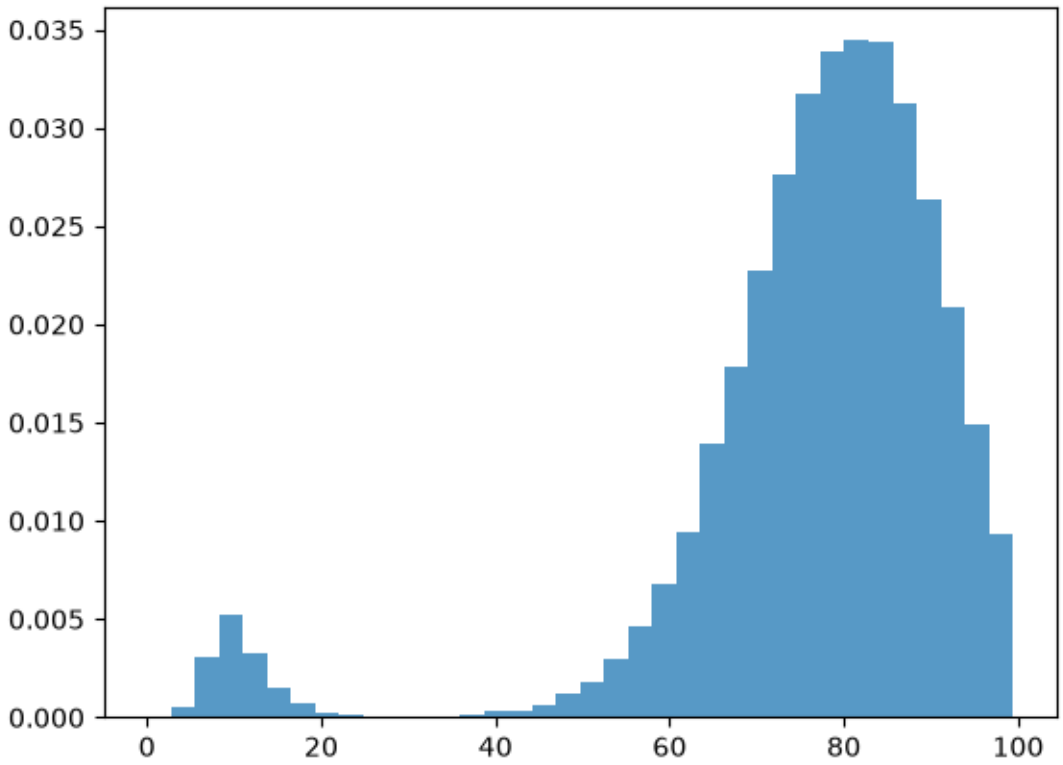
We can build up a clearer picture by drawing more samples

```
T = 50
M = 50_000

fig, ax = plt.subplots()

# Draw all M paths at once by vectorizing over one key per path
keys = random.split(random.key(0), M)
paths = jax.vmap(
    sim_inventory_path, in_axes=(None, None, 0, None)
```

(continues on next page)

(continued from previous page)

```
)(firm, x_init, keys, T + 1)
sample = paths[:, T]

ax.hist(sample, bins=36, density=True, histtype="bar", alpha=0.75)

plt.show()
```



Note that the distribution is bimodal

- Most firms have restocked twice but a few have restocked only once (see figure with paths above).
- Firms in the second category have lower inventory.

We can also approximate the distribution using a kernel density estimator.

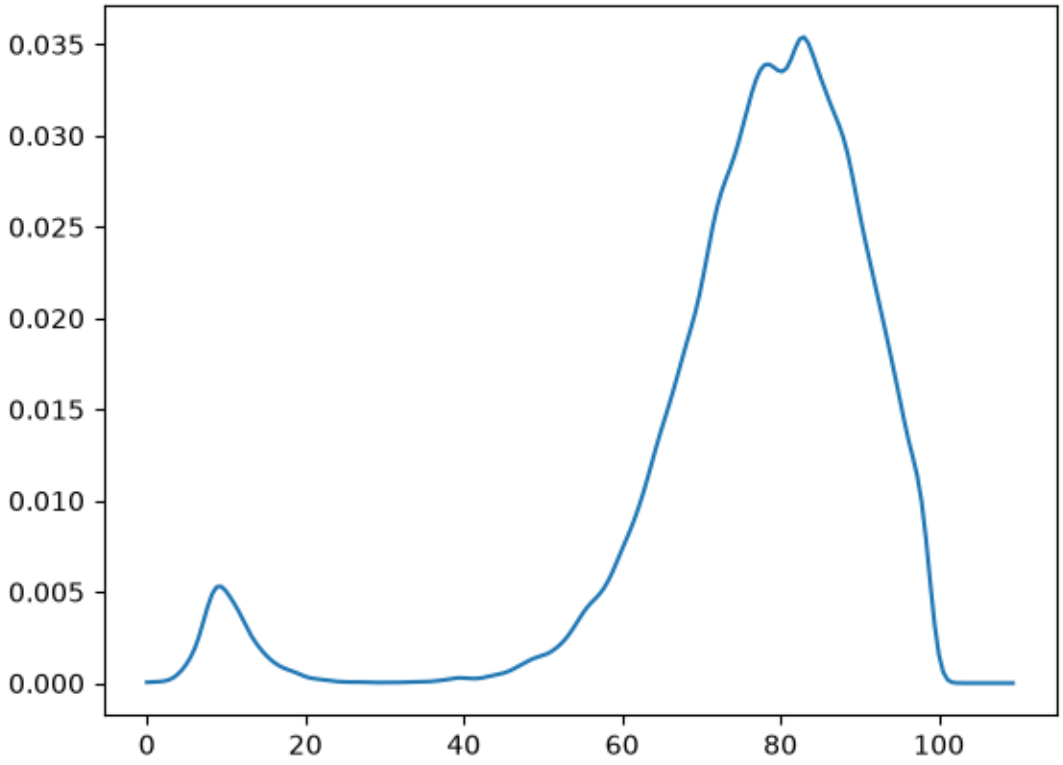
Kernel density estimators can be thought of as smoothed histograms.

They are preferable to histograms when the distribution being estimated is likely to be smooth.

We will use a kernel density estimator from scikit-learn

```
def plot_kde(sample, ax, label=""):
    xmin, xmax = 0.9 * sample.min(), 1.1 * sample.max()
    xgrid = jnp.linspace(xmin, xmax, 200)
    kde = KernelDensity(kernel="gaussian").fit(sample[:, None])
    log_dens = kde.score_samples(xgrid[:, None])

    ax.plot(xgrid, jnp.exp(log_dens), label=label)
```

```
fig, ax = plt.subplots()
plot_kde(sample, ax)
plt.show()
```



The allocation of probability density is similar to what was shown by the histogram just above.

39.4 Exercises

Exercise 39.4.1

This model is asymptotically stationary, with a unique stationary distribution.

(See the discussion of stationarity in our lecture on AR(1) processes for background — the fundamental concepts are the same.)

In particular, the sequence of marginal distributions $\{\psi_t\}$ is converging to a unique limiting distribution that does not depend on initial conditions.

Although we will not prove this here, we can investigate it using simulation.

Your task is to generate and plot the sequence $\{\psi_t\}$ at times $t = 10, 50, 250, 500, 750$ based on the discussion above.

(The kernel density estimator is probably the best way to present each distribution.)

You should see convergence, in the sense that differences between successive distributions are getting smaller.

Try different initial conditions to verify that, in the long run, the distribution is invariant across initial conditions.

Solution

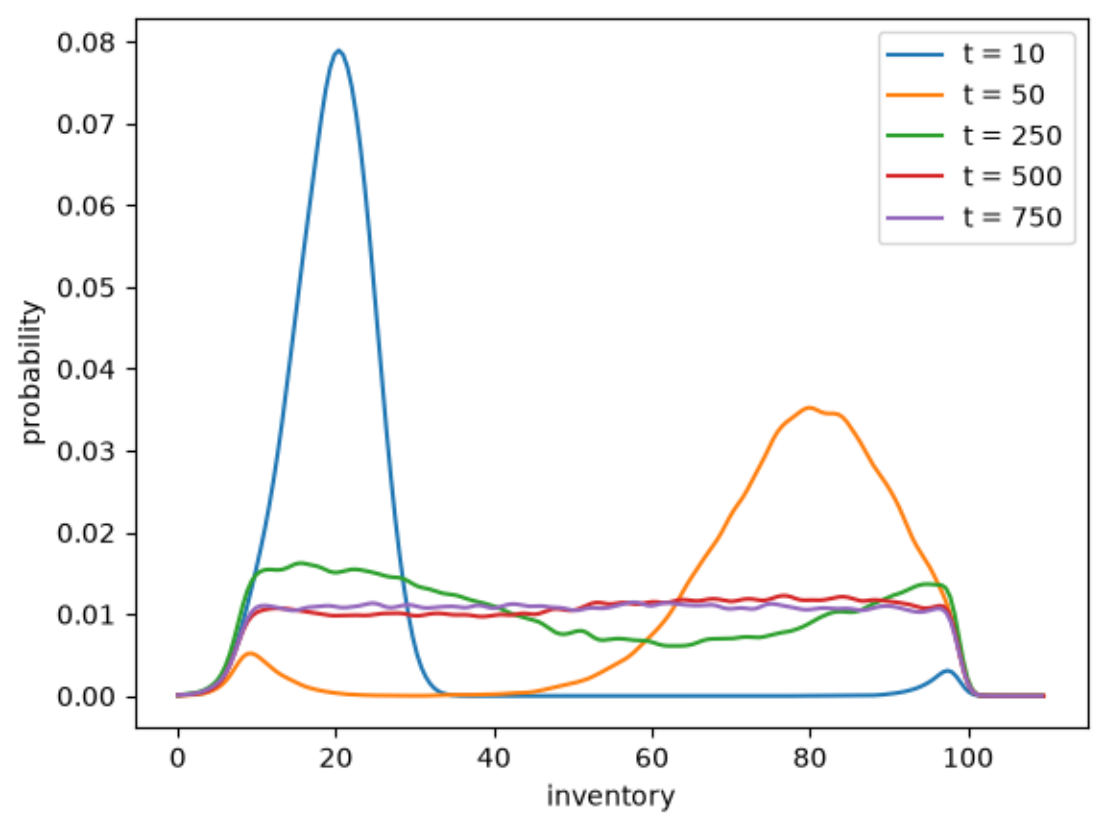
Below is one possible solution:

The computations involve a lot of CPU cycles so we have tried to write the code efficiently using `jax.jit` and `jax.vmap` to run on CPU/GPU.

```
@jax.jit
def simulate_firm_forward(firm, x_init, key, num_periods):
    """
    Simulate a single firm num_periods steps forward and return its
    final inventory level. A fresh shock is drawn each period from key.
    """

    def update(t, x):
        Z = random.normal(random.fold_in(key, t))
        D = jnp.exp(firm.μ + firm.σ * Z)
        return jnp.where(
            x <= firm.s,
            jnp.maximum(firm.S - D, 0.0),
            jnp.maximum(x - D, 0.0),
        )

    return jax.lax.fori_loop(0, num_periods, update, x_init * 1.0)


# Vectorize over firms: each has its own initial level and its own key
simulate_firms_forward = jax.vmap(
    simulate_firm_forward, in_axes=(None, 0, 0, None)
)
```

```
def shift_firms_forward(firm, current_inventory_levels, num_periods, key):
    """
    Shift multiple firms forward by num_periods using JAX vectorization.
    Returns:
        Array of new inventory levels after num_periods
    """

    # Generate one independent random key per firm
    num_firms = len(current_inventory_levels)
    firm_keys = random.split(key, num_firms)
    # Run simulation for all firms in parallel
    new_inventory_levels = simulate_firms_forward(
        firm, current_inventory_levels, firm_keys, num_periods
    )

    return new_inventory_levels
```

```
x_init = 50
num_firms = 50_000

sample_dates = 0, 10, 50, 250, 500, 750

first_diffs = jnp.diff(jnp.array(sample_dates))

fig, ax = plt.subplots()

X = jnp.full(num_firms, x_init)

current_date = 0
for d in first_diffs:
    X = shift_firms_forward(firm, X, d, random.key(current_date + 1))
    current_date += d
    plot_kde(X, ax, label=f"t = {current_date}")

ax.set_xlabel("inventory")
```



Notice that by $t = 500$ or $t = 750$ the densities are barely changing.

We have reached a reasonable approximation of the stationary density.

You can convince yourself that initial conditions don't matter by testing a few of them.

For example, try rerunning the code above with all firms starting at $X_0 = 20$ or $X_0 = 80$.

Exercise 39.4.2

Using simulation, calculate the probability that firms that start with $X_0 = 70$ need to order twice or more in the first 50 periods.

You will need a large sample size to get an accurate reading.

Solution

Here is one solution.

Again, the computations are relatively intensive so we have written a specialized JAX-jitted function and using `jax.vmap` to use parallelization across firms.

Note the time the routine takes to run, as well as the output.

```
@jax.jit
def simulate_firm_restocks(firm, x_init, key, num_periods):
    """
```

```
    Simulate a single firm for num_periods and report whether it
    restocked more than once. A fresh shock is drawn each period from key.

    Returns:
        1 if the firm restocks > 1 times, 0 otherwise
    """

    def update(t, carry):
        x, restock_count = carry
        Z = random.normal(random.fold_in(key, t))
        D = jnp.exp(firm.μ + firm.σ * Z)
        restock = x <= firm.s
        x_new = jnp.where(
            restock,
            jnp.maximum(firm.S - D, 0.0),
            jnp.maximum(x - D, 0.0),
        )
        return x_new, restock_count + restock

    # Carry the inventory level and a running restock count
    _, total_restocks = jax.lax.fori_loop(
        0, num_periods, update, (x_init * 1.0, 0)
    )
    return (total_restocks > 1).astype(jnp.int32)


# Vectorize the simulation across all firms (one key each)
simulate_firms_restocks = jax.vmap(
    simulate_firm_restocks, in_axes=(None, None, 0, None)
)



def compute_freq(
    firm, x_init=70, sim_length=50, num_firms=1_000_000, key=random.key(2)
):
    """
    Compute the frequency of firms that restock 2 or more times using JAX.

    Args:
        firm: Firm dataclass
        x_init: Initial inventory level for all firms
        sim_length: Length of simulation for each firm
        num_firms: Number of firms to simulate
        key: JAX random key

    Returns:
        Fraction of firms that restock 2 or more times
    """
    # Generate one independent random key per firm
    firm_keys = random.split(key, num_firms)
    # Run simulation for all firms
    restock_indicators = simulate_firms_restocks(
        firm, x_init, firm_keys, sim_length
    )
    # Compute frequency (fraction of firms that restocked > 1 times)
    frequency = jnp.mean(restock_indicators)
    return frequency
```

```
%%time

freq = compute_freq(firm)
print(f"Frequency of at least two stock outs = {freq}")
```

```
Frequency of at least two stock outs = 0.44668900966644287
CPU times: user 1.03 s, sys: 25.5 ms, total: 1.05 s
Wall time: 732 ms
```

CHAPTER

FORTY

LINEAR STATE SPACE MODELS

Contents

- *Linear State Space Models*
  - *Overview*
  - *The Linear State Space Model*
  - *Distributions and Moments*
  - *Stationarity and Ergodicity*
  - *Noisy Observations*
  - *Prediction*
  - *Code*
  - *Exercises*

> "We may regard the present state of the universe as the effect of its past and the cause of its future" – Marquis de Laplace

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install quantecon
```

40.1 Overview

This lecture introduces the **linear state space** dynamic system.

The linear state space system is a generalization of the scalar AR(1) process we studied before.

This model is a workhorse that carries a powerful theory of prediction.

Its many applications include:

- representing dynamics of higher-order linear systems
- predicting the position of a system j steps into the future
- predicting a geometric sum of future values of a variable like
  - non-financial income

  - dividends on a stock
  - the money supply
  - a government deficit or surplus, etc.
- key ingredient of useful models
  - Friedman's permanent income model of consumption smoothing.
  - Barro's model of smoothing total tax collections.
  - Rational expectations version of Cagan's model of hyperinflation.
  - Sargent and Wallace's "unpleasant monetarist arithmetic," etc.

Let's start with some imports:

```
import matplotlib.pyplot as plt
import numpy as np
from quantecon import LinearStateSpace
from scipy.stats import norm
import random
```

40.2 The Linear State Space Model

The objects in play are:

- An $n \times 1$ vector x_t denoting the **state** at time $t = 0, 1, 2, \ldots$.
- An IID sequence of $m \times 1$ random vectors $w_t \sim N(0, I)$.
- A $k \times 1$ vector y_t of **observations** at time $t = 0, 1, 2, \ldots$.
- An $n \times n$ matrix A called the **transition matrix**.
- An $n \times m$ matrix C called the **volatility matrix**.
- A $k \times n$ matrix G sometimes called the **output matrix**.

Here is the linear state-space system

$$
\begin{aligned}
x_{t+1} &= Ax_t + Cw_{t+1} \\
y_t &= Gx_t \\
x_0 &\sim N(\mu_0, \Sigma_0)
\end{aligned}
$$

40.2.1 Primitives

The primitives of the model are

1. the matrices A, C, G
2. shock distribution, which we have specialized to $N(0, I)$
3. the distribution of the initial condition x_0, which we have set to $N(\mu_0, \Sigma_0)$

Given A, C, G and draws of x_0 and $w_1, w_2, \ldots$, the model (40.1) pins down the values of the sequences $\{x_t\}$ and $\{y_t\}$.

Even without these draws, the primitives 1–3 pin down the **probability distributions** of $\{x_t\}$ and $\{y_t\}$.

Later we'll see how to compute these distributions and their moments.

Martingale Difference Shocks

We've made the common assumption that the shocks are independent standardized normal vectors.

But some of what we say will be valid under the assumption that $\{w_{t+1}\}$ is a **martingale difference sequence**.

A martingale difference sequence is a sequence that is zero mean when conditioned on past information.

In the present case, since $\{x_t\}$ is our state sequence, this means that it satisfies

$$\mathbb{E}[w_{t+1}|x_t, x_{t-1}, ...] = 0$$

This is a weaker condition than that $\{w_t\}$ is IID with $w_{t+1} \sim N(0, I)$.

40.2.2 Examples

By appropriate choice of the primitives, a variety of dynamics can be represented in terms of the linear state space model.

The following examples help to highlight this point.

They also illustrate the wise dictum *finding the state is an art*.

Second-order Difference Equation

Let $\{y_t\}$ be a deterministic sequence that satisfies

$$y_{t+1} = \phi_0 + \phi_1 y_t + \phi_2 y_{t-1} \quad \text{s.t.} \quad y_0, y_{-1} \text{ given} \tag{40.1}$$

To map (40.1) into our state space system (40.1), we set

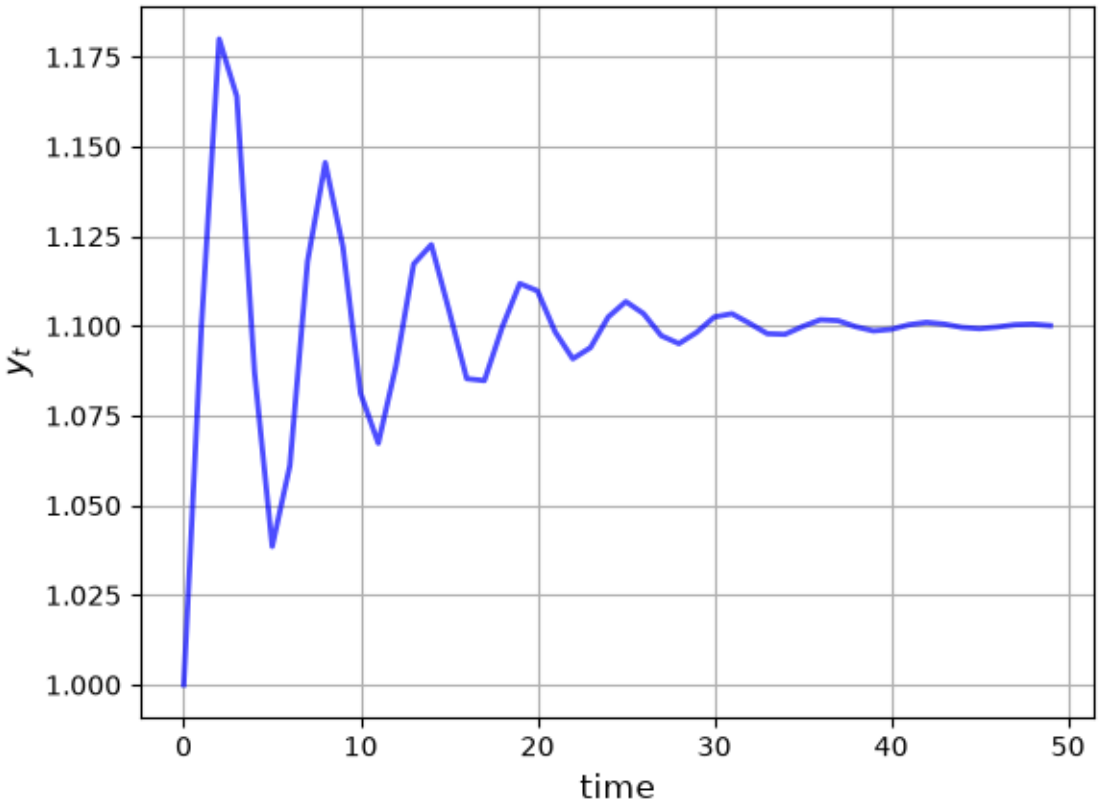
$$x_t = \begin{bmatrix} 1 \\ y_t \\ y_{t-1} \end{bmatrix} \qquad A = \begin{bmatrix} 1 & 0 & 0 \\ \phi_0 & \phi_1 & \phi_2 \\ 0 & 1 & 0 \end{bmatrix} \qquad C = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \qquad G = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}$$

You can confirm that under these definitions, (40.1) and (40.1) agree.

The next figure shows the dynamics of this process when $\phi_0 = 1.1, \phi_1 = 0.8, \phi_2 = -0.8, y_0 = y_{-1} = 1$.

```
def plot_lss(A,
             C,
             G,
             n=3,
             ts_length=50):

    ar = LinearStateSpace(A, C, G, mu_0=np.ones(n))
    x, y = ar.simulate(ts_length)

    fig, ax = plt.subplots()
    y = y.flatten()
    ax.plot(y, 'b-', lw=2, alpha=0.7)
    ax.grid()
    ax.set_xlabel('time', fontsize=12)
    ax.set_ylabel('$y_t$', fontsize=12)
    plt.show()
```

```
ϕ_0, ϕ_1, ϕ_2 = 1.1, 0.8, -0.8

A = [[1,     0,     0  ],
     [ϕ_0,   ϕ_1,   ϕ_2],
     [0,     1,     0  ]]

C = np.zeros((3, 1))
G = [0, 1, 0]

plot_lss(A, C, G)
```



Later you'll be asked to recreate this figure.

Univariate Autoregressive Processes

We can use (40.1) to represent the model

$$y_{t+1} = \phi_1 y_t + \phi_2 y_{t-1} + \phi_3 y_{t-2} + \phi_4 y_{t-3} + \sigma w_{t+1} \tag{40.2}$$

where $\{w_t\}$ is IID and standard normal.

To put this in the linear state space format we take $x_t = \begin{bmatrix} y_t & y_{t-1} & y_{t-2} & y_{t-3} \end{bmatrix}'$ and

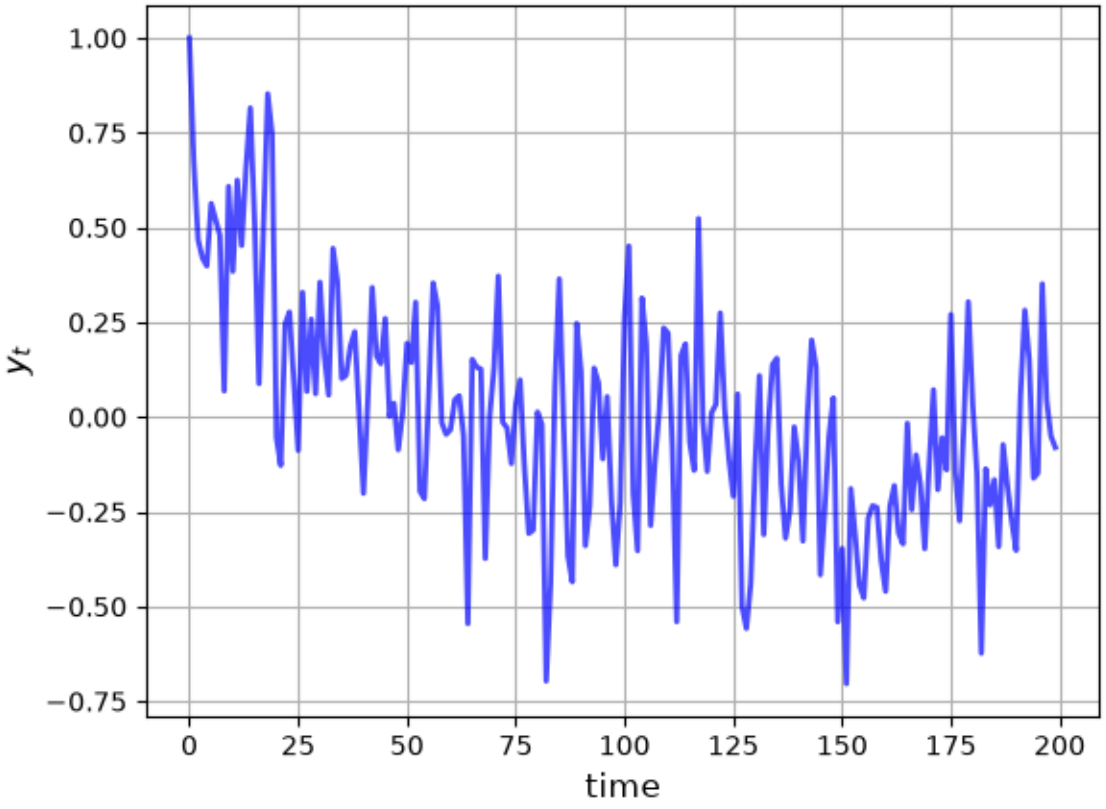
$$A = \begin{bmatrix} \phi_1 & \phi_2 & \phi_3 & \phi_4 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \qquad C = \begin{bmatrix} \sigma \\ 0 \\ 0 \\ 0 \end{bmatrix} \qquad G = \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix}$$

The matrix A has the form of the **companion matrix** to the vector $\begin{bmatrix}\phi_1 & \phi_2 & \phi_3 & \phi_4\end{bmatrix}$.

The next figure shows the dynamics of this process when

$$\phi_1 = 0.5, \phi_2 = -0.2, \phi_3 = 0, \phi_4 = 0.5, \sigma = 0.2, y_0 = y_{-1} = y_{-2} = y_{-3} = 1$$

```
ϕ_1, ϕ_2, ϕ_3, ϕ_4 = 0.5, -0.2, 0, 0.5
σ = 0.2

A_1 = [[ϕ_1,   ϕ_2,   ϕ_3,   ϕ_4],
       [1,     0,     0,     0  ],
       [0,     1,     0,     0  ],
       [0,     0,     1,     0  ]]

C_1 = [[σ],
       [0],
       [0],
       [0]]

G_1 = [1, 0, 0, 0]

plot_lss(A_1, C_1, G_1, n=4, ts_length=200)
```



Vector Autoregressions

Now suppose that

- y_t is a $k \times 1$ vector
- ϕ_j is a $k \times k$ matrix and
- w_t is $k \times 1$

Then (40.2) is termed a **vector autoregression**.

To map this into (40.1), we set

$$x_t = \begin{bmatrix} y_t \\ y_{t-1} \\ y_{t-2} \\ y_{t-3} \end{bmatrix} \quad A = \begin{bmatrix} \phi_1 & \phi_2 & \phi_3 & \phi_4 \\ I & 0 & 0 & 0 \\ 0 & I & 0 & 0 \\ 0 & 0 & I & 0 \end{bmatrix} \quad C = \begin{bmatrix} \sigma \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad G = \begin{bmatrix} I & 0 & 0 & 0 \end{bmatrix}$$

where I is the $k \times k$ identity matrix and σ is a $k \times k$ matrix.

Seasonals

We can use (40.1) to represent

1. the **deterministic seasonal** $y_t = y_{t-4}$
2. the **indeterministic seasonal** $y_t = \phi_4 y_{t-4} + w_t$

In fact, both are special cases of (40.2).

With the deterministic seasonal, the transition matrix becomes

$$A = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

It is easy to check that $A^4 = I$, which implies that x_t is strictly periodic with period 4:[1]

$$x_{t+4} = x_t$$

Such an x_t process can be used to model deterministic seasonals in quarterly time series.

The *indeterministic* seasonal produces recurrent, but aperiodic, seasonal fluctuations.

Time Trends

The model $y_t = at + b$ is known as a **linear time trend**.

We can represent this model in the linear state space form by taking

$$A = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \qquad C = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \qquad G = \begin{bmatrix} a & b \end{bmatrix} \tag{40.3}$$

and starting at initial condition $x_0 = \begin{bmatrix} 0 & 1 \end{bmatrix}'$.

In fact, it's possible to use the state-space system to represent polynomial trends of any order.

[1] The eigenvalues of A are $(1, -1, i, -i)$.

For instance, we can represent the model $y_t = at^2 + bt + c$ in the linear state space form by taking

$$A = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \qquad C = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \qquad G = \begin{bmatrix} 2a & a+b & c \end{bmatrix}$$

and starting at initial condition $x_0 = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}'$.

It follows that

$$A^t = \begin{bmatrix} 1 & t & t(t-1)/2 \\ 0 & 1 & t \\ 0 & 0 & 1 \end{bmatrix}$$

Then $x_t' = \begin{bmatrix} t(t-1)/2 & t & 1 \end{bmatrix}$. You can now confirm that $y_t = Gx_t$ has the correct form.

40.2.3 Moving Average Representations

A nonrecursive expression for x_t as a function of $x_0, w_1, w_2, \ldots, w_t$ can be found by using (40.1) repeatedly to obtain

$$\begin{aligned} x_t &= Ax_{t-1} + Cw_t \\ &= A^2 x_{t-2} + ACw_{t-1} + Cw_t \\ &\quad \vdots \\ &= \sum_{j=0}^{t-1} A^j C w_{t-j} + A^t x_0 \end{aligned}$$

Representation (40.4) is a **moving average** representation.

It expresses $\{x_t\}$ as a linear function of

1. current and past values of the process $\{w_t\}$ and
2. the initial condition x_0

As an example of a moving average representation, let the model be

$$A = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \qquad C = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

You will be able to show that $A^t = \begin{bmatrix} 1 & t \\ 0 & 1 \end{bmatrix}$ and $A^j C = \begin{bmatrix} 1 & 0 \end{bmatrix}'$.

Substituting into the moving average representation (40.4), we obtain

$$x_{1t} = \sum_{j=0}^{t-1} w_{t-j} + \begin{bmatrix} 1 & t \end{bmatrix} x_0$$

where x_{1t} is the first entry of x_t.

The first term on the right is a cumulated sum of martingale differences and is therefore a martingale.

The second term is a translated linear function of time.

For this reason, x_{1t} is called a **martingale with drift**.

40.3 Distributions and Moments

40.3.1 Unconditional Moments

Using (40.1), it's easy to obtain expressions for the (unconditional) means of x_t and y_t.

We'll explain what *unconditional* and *conditional* mean soon.

Letting $\mu_t := \mathbb{E}[x_t]$ and using linearity of expectations, we find that

$$\mu_{t+1} = A\mu_t \quad \text{with} \quad \mu_0 \text{ given} \tag{40.4}$$

Here μ_0 is a primitive given in (40.1).

The variance-covariance matrix of x_t is $\Sigma_t := \mathbb{E}[(x_t - \mu_t)(x_t - \mu_t)']$.

Using $x_{t+1} - \mu_{t+1} = A(x_t - \mu_t) + Cw_{t+1}$, we can determine this matrix recursively via

$$\Sigma_{t+1} = A\Sigma_t A' + CC' \quad \text{with} \quad \Sigma_0 \text{ given} \tag{40.5}$$

As with μ_0, the matrix Σ_0 is a primitive given in (40.1).

As a matter of terminology, we will sometimes call

- μ_t the **unconditional mean** of x_t
- Σ_t the **unconditional variance-covariance matrix** of x_t

This is to distinguish μ_t and Σ_t from related objects that use conditioning information, to be defined below.

However, you should be aware that these "unconditional" moments do depend on the initial distribution $N(\mu_0, \Sigma_0)$.

Moments of the Observables

Using linearity of expectations again we have

$$\mathbb{E}[y_t] = \mathbb{E}[Gx_t] = G\mu_t \tag{40.6}$$

The variance-covariance matrix of y_t is easily shown to be

$$\text{Var}[y_t] = \text{Var}[Gx_t] = G\Sigma_t G' \tag{40.7}$$

40.3.2 Distributions

In general, knowing the mean and variance-covariance matrix of a random vector is not quite as good as knowing the full distribution.

However, there are some situations where these moments alone tell us all we need to know.

These are situations in which the mean vector and covariance matrix are all of the **parameters** that pin down the population distribution.

One such situation is when the vector in question is Gaussian (i.e., normally distributed).

This is the case here, given

1. our Gaussian assumptions on the primitives
2. the fact that normality is preserved under linear operations

In fact, it's well-known that

$$u \sim N(\bar{u}, S) \quad \text{and} \quad v = a + Bu \implies v \sim N(a + B\bar{u}, BSB') \tag{40.8}$$

In particular, given our Gaussian assumptions on the primitives and the linearity of (40.1) we can see immediately that both x_t and y_t are Gaussian for all $t \geq 0$[2].

Since x_t is Gaussian, to find the distribution, all we need to do is find its mean and variance-covariance matrix.

But in fact we've already done this, in (40.4) and (40.5).

Letting μ_t and Σ_t be as defined by these equations, we have

$$x_t \sim N(\mu_t, \Sigma_t) \tag{40.9}$$

By similar reasoning combined with (40.6) and (40.7),

$$y_t \sim N(G\mu_t, G\Sigma_t G') \tag{40.10}$$

40.3.3 Ensemble Interpretations

How should we interpret the distributions defined by (40.9)–(40.10)?

Intuitively, the probabilities in a distribution correspond to relative frequencies in a large population drawn from that distribution.

Let's apply this idea to our setting, focusing on the distribution of y_T for fixed T.

We can generate independent draws of y_T by repeatedly simulating the evolution of the system up to time T, using an independent set of shocks each time.

The next figure shows 20 simulations, producing 20 time series for $\{y_t\}$, and hence 20 draws of y_T.

The system in question is the univariate autoregressive model (40.2).

The values of y_T are represented by black dots in the left-hand figure

```
def cross_section_plot(A,
                       C,
                       G,
                       T=20,                 # Set the time
                       ymin=-0.8,
                       ymax=1.25,
                       sample_size = 20,     # 20 observations/simulations
                       n=4):                 # The number of dimensions for the initial x0

    ar = LinearStateSpace(A, C, G, mu_0=np.ones(n))

    fig, axes = plt.subplots(1, 2, figsize=(16, 5))

    for ax in axes:
        ax.grid(alpha=0.4)
        ax.set_ylim(ymin, ymax)

    ax = axes[0]
    ax.set_ylim(ymin, ymax)
    ax.set_ylabel('$y_t$', fontsize=12)
```

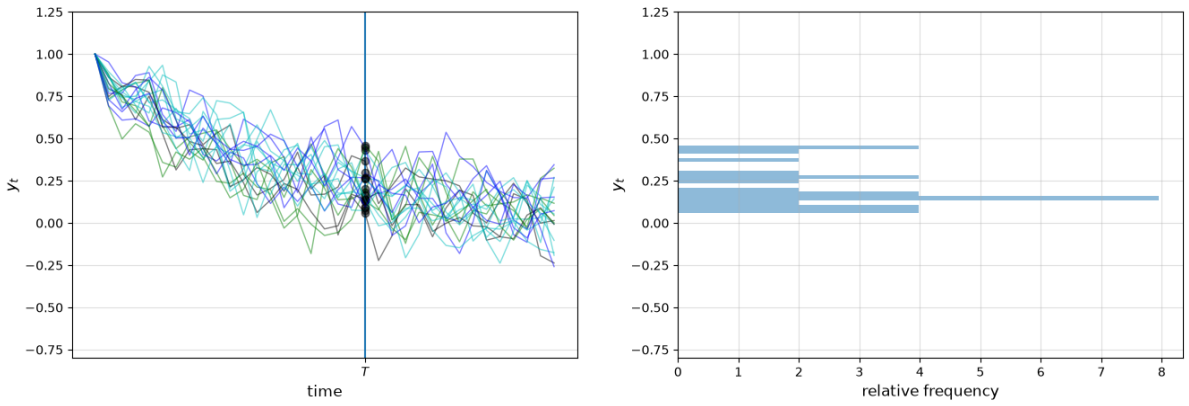
(continues on next page)

[2] The correct way to argue this is by induction. Suppose that x_t is Gaussian. Then (40.1) and (40.8) imply that x_{t+1} is Gaussian. Since x_0 is assumed to be Gaussian, it follows that every x_t is Gaussian. Evidently, this implies that each y_t is Gaussian.

(continued from previous page)

```
    ax.set_xlabel('time', fontsize=12)
    ax.vlines((T,), -1.5, 1.5)

    ax.set_xticks((T,))
    ax.set_xticklabels(('$T$',))

    sample = []
    for i in range(sample_size):
        rcolor = random.choice(('c', 'g', 'b', 'k'))
        x, y = ar.simulate(ts_length=T+15)
        y = y.flatten()
        ax.plot(y, color=rcolor, lw=1, alpha=0.5)
        ax.plot((T,), (y[T],), 'ko', alpha=0.5)
        sample.append(y[T])

    y = y.flatten()
    axes[1].set_ylim(ymin, ymax)
    axes[1].set_ylabel('$y_t$', fontsize=12)
    axes[1].set_xlabel('relative frequency', fontsize=12)
    axes[1].hist(sample, bins=16, density=True, orientation='horizontal', alpha=0.5)
    plt.show()
```

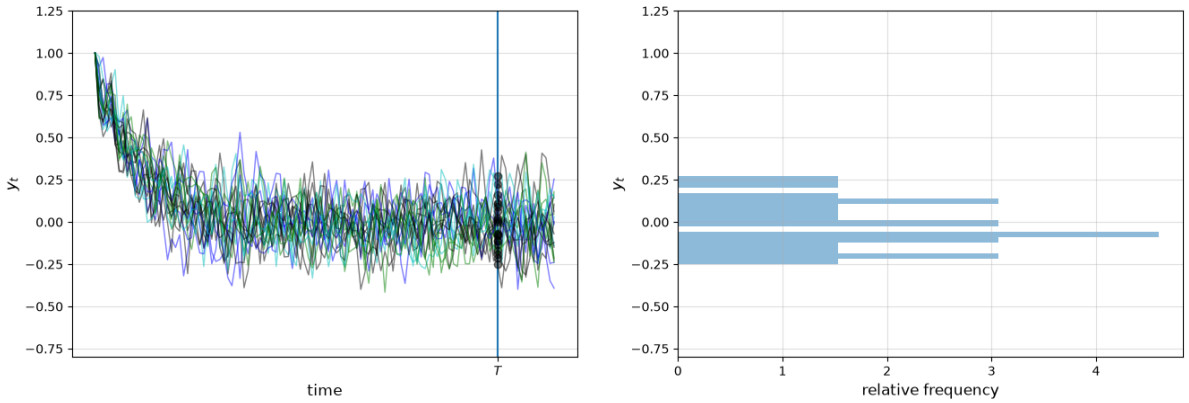
```
ϕ_1, ϕ_2, ϕ_3, ϕ_4 = 0.5, -0.2, 0, 0.5
σ = 0.1

A_2 = [[ϕ_1, ϕ_2, ϕ_3, ϕ_4],
       [1,     0,     0,     0],
       [0,     1,     0,     0],
       [0,     0,     1,     0]]

C_2 = [[σ], [0], [0], [0]]

G_2 = [1, 0, 0, 0]

cross_section_plot(A_2, C_2, G_2)
```



In the right-hand figure, these values are converted into a rotated histogram that shows relative frequencies from our sample of 20 y_T's.

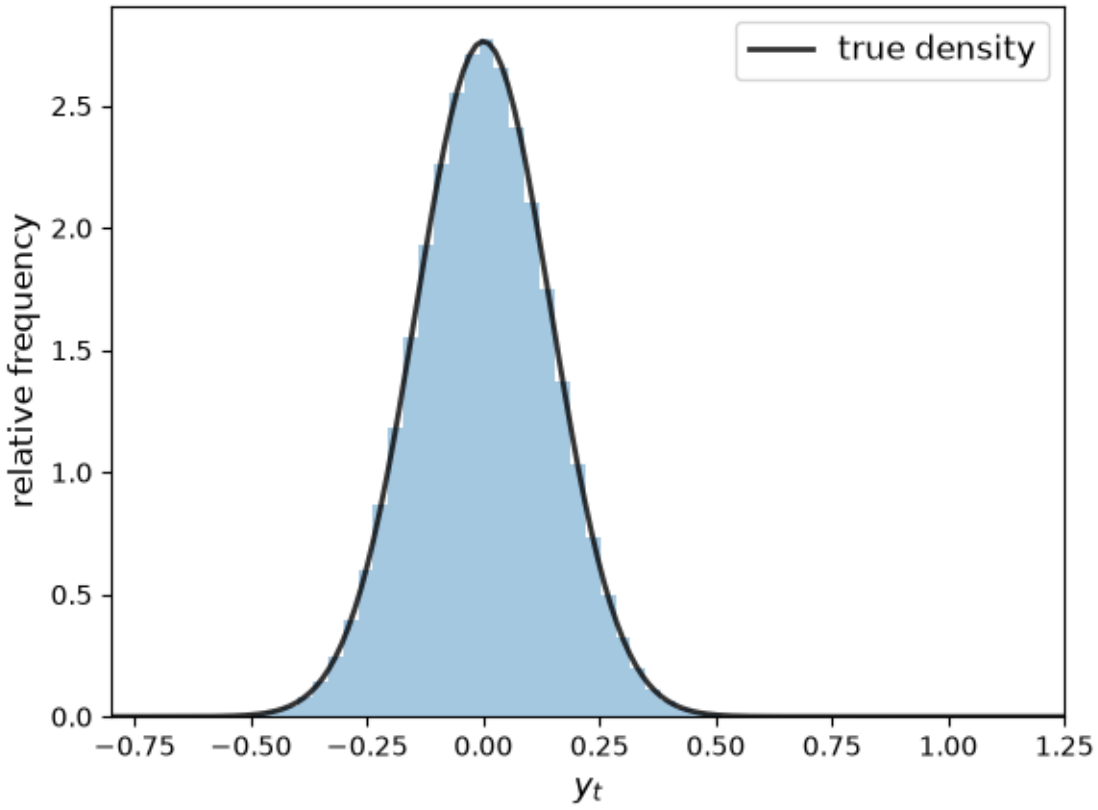
Here is another figure, this time with 100 observations

```
t = 100
cross_section_plot(A_2, C_2, G_2, T=t)
```



Let's now try with 500,000 observations, showing only the histogram (without rotation)

```
T = 100
ymin=-0.8
ymax=1.25
sample_size = 500_000

ar = LinearStateSpace(A_2, C_2, G_2, mu_0=np.ones(4))
fig, ax = plt.subplots()
x, y = ar.simulate(sample_size)
mu_x, mu_y, Sigma_x, Sigma_y, Sigma_yx = ar.stationary_distributions()
f_y = norm(loc=float(mu_y.item()), scale=float(np.sqrt(Sigma_y.item())))
y = y.flatten()
ygrid = np.linspace(ymin, ymax, 150)

ax.hist(y, bins=50, density=True, alpha=0.4)
ax.plot(ygrid, f_y.pdf(ygrid), 'k-', lw=2, alpha=0.8, label='true density')
ax.set_xlim(ymin, ymax)
ax.set_xlabel('$y_t$', fontsize=12)
ax.set_ylabel('relative frequency', fontsize=12)
ax.legend(fontsize=12)
plt.show()
```



The black line is the population density of y_T calculated from (40.10).

The histogram and population distribution are close, as expected.

By looking at the figures and experimenting with parameters, you will gain a feel for how the population distribution depends on the model primitives *listed above*, as intermediated by the distribution's parameters.

Ensemble Means

In the preceding figure, we approximated the population distribution of y_T by

1. generating I sample paths (i.e., time series) where I is a large number
2. recording each observation y_T^i
3. histogramming this sample

Just as the histogram approximates the population distribution, the **ensemble** or **cross-sectional average**

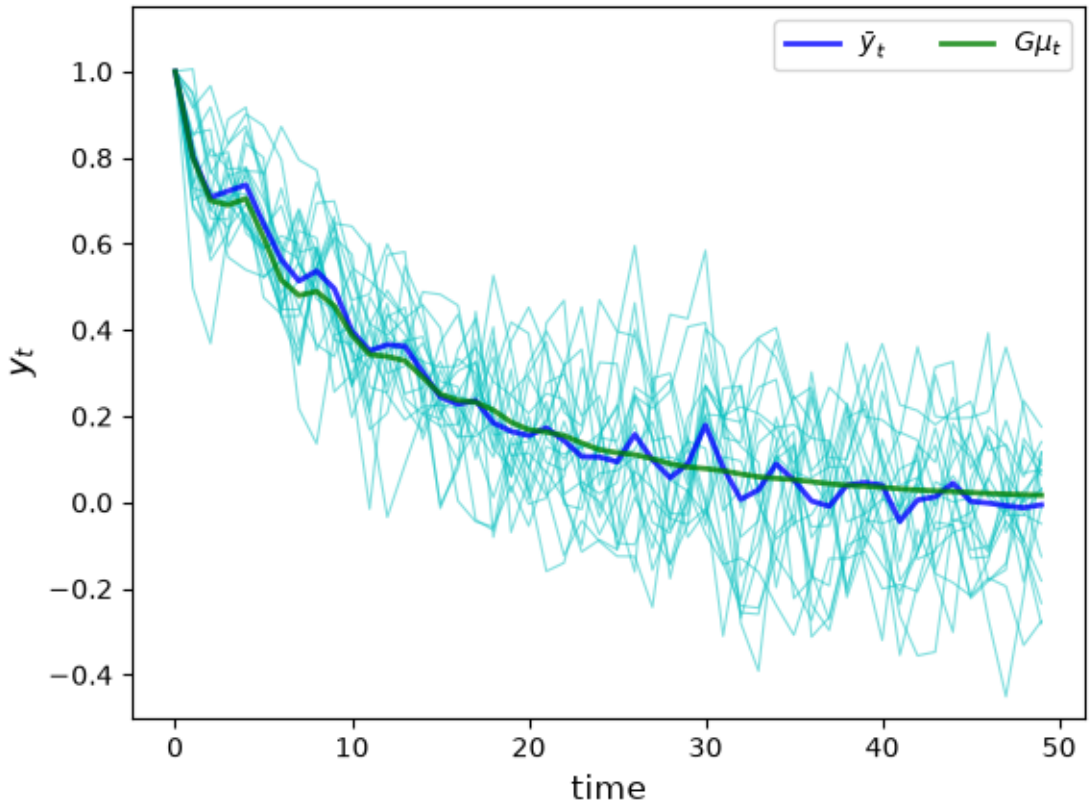
$$\bar{y}_T := \frac{1}{I} \sum_{i=1}^{I} y_T^i$$

approximates the expectation $\mathbb{E}[y_T] = G\mu_T$ (as implied by the law of large numbers).

Here's a simulation comparing the ensemble averages and population means at time points $t = 0, \ldots, 50$.

The parameters are the same as for the preceding figures, and the sample size is relatively small ($I = 20$).

```
I = 20
T = 50
ymin = -0.5
ymax = 1.15

ar = LinearStateSpace(A_2, C_2, G_2, mu_0=np.ones(4))

fig, ax = plt.subplots()

ensemble_mean = np.zeros(T)
for i in range(I):
    x, y = ar.simulate(ts_length=T)
    y = y.flatten()
    ax.plot(y, 'c-', lw=0.8, alpha=0.5)
    ensemble_mean = ensemble_mean + y

ensemble_mean = ensemble_mean / I
ax.plot(ensemble_mean, color='b', lw=2, alpha=0.8, label='$\\bar y_t$')
m = ar.moment_sequence()

population_means = []
for t in range(T):
    μ_x, μ_y, Σ_x, Σ_y = next(m)
    population_means.append(float(μ_y.item()))

ax.plot(population_means, color='g', lw=2, alpha=0.8, label=r'$G\mu_t$')
ax.set_ylim(ymin, ymax)
ax.set_xlabel('time', fontsize=12)
ax.set_ylabel('$y_t$', fontsize=12)
ax.legend(ncol=2)
plt.show()
```



The ensemble mean for x_t is

$$\bar{x}_T := \frac{1}{I}\sum_{i=1}^{I} x_T^i \to \mu_T \qquad (I \to \infty)$$

The limit μ_T is a "long-run average".

(By *long-run average* we mean the average for an infinite ($I = \infty$) number of sample x_T's)

Another application of the law of large numbers assures us that

$$\frac{1}{I}\sum_{i=1}^{I}(x_T^i - \bar{x}_T)(x_T^i - \bar{x}_T)' \to \Sigma_T \qquad (I \to \infty)$$

40.3.4 Joint Distributions

In the preceding discussion, we looked at the distributions of x_t and y_t in isolation.

This gives us useful information but doesn't allow us to answer questions like

- what's the probability that $x_t \geq 0$ for all t?
- what's the probability that the process $\{y_t\}$ exceeds some value a before falling below b?
- etc., etc.

Such questions concern the *joint distributions* of these sequences.

To compute the joint distribution of $x_0, x_1, \ldots, x_T$, recall that joint and conditional densities are linked by the rule

$$p(x, y) = p(y \,|\, x)p(x) \qquad \text{(joint} = \text{conditional} \times \text{marginal)}$$

From this rule we get $p(x_0, x_1) = p(x_1 \mid x_0)p(x_0)$.

The Markov property $p(x_t \mid x_{t-1}, \ldots, x_0) = p(x_t \mid x_{t-1})$ and repeated applications of the preceding rule lead us to

$$p(x_0, x_1, \ldots, x_T) = p(x_0) \prod_{t=0}^{T-1} p(x_{t+1} \mid x_t)$$

The marginal $p(x_0)$ is just the primitive $N(\mu_0, \Sigma_0)$.

In view of (40.1), the conditional densities are

$$p(x_{t+1} \mid x_t) = N(Ax_t, CC')$$

Autocovariance Functions

An important object related to the joint distribution is the **autocovariance function**

$$\Sigma_{t+j,t} := \mathbb{E}[(x_{t+j} - \mu_{t+j})(x_t - \mu_t)'] \tag{40.11}$$

Elementary calculations show that

$$\Sigma_{t+j,t} = A^j \Sigma_t \tag{40.12}$$

Notice that $\Sigma_{t+j,t}$ in general depends on both j, the gap between the two dates, and t, the earlier date.

40.4 Stationarity and Ergodicity

Stationarity and ergodicity are two properties that, when they hold, greatly aid analysis of linear state space models.

Let's start with the intuition.

40.4.1 Visualizing Stability

Let's look at some more time series from the same model that we analyzed above.

This picture shows cross-sectional distributions for y at times T, T', T''

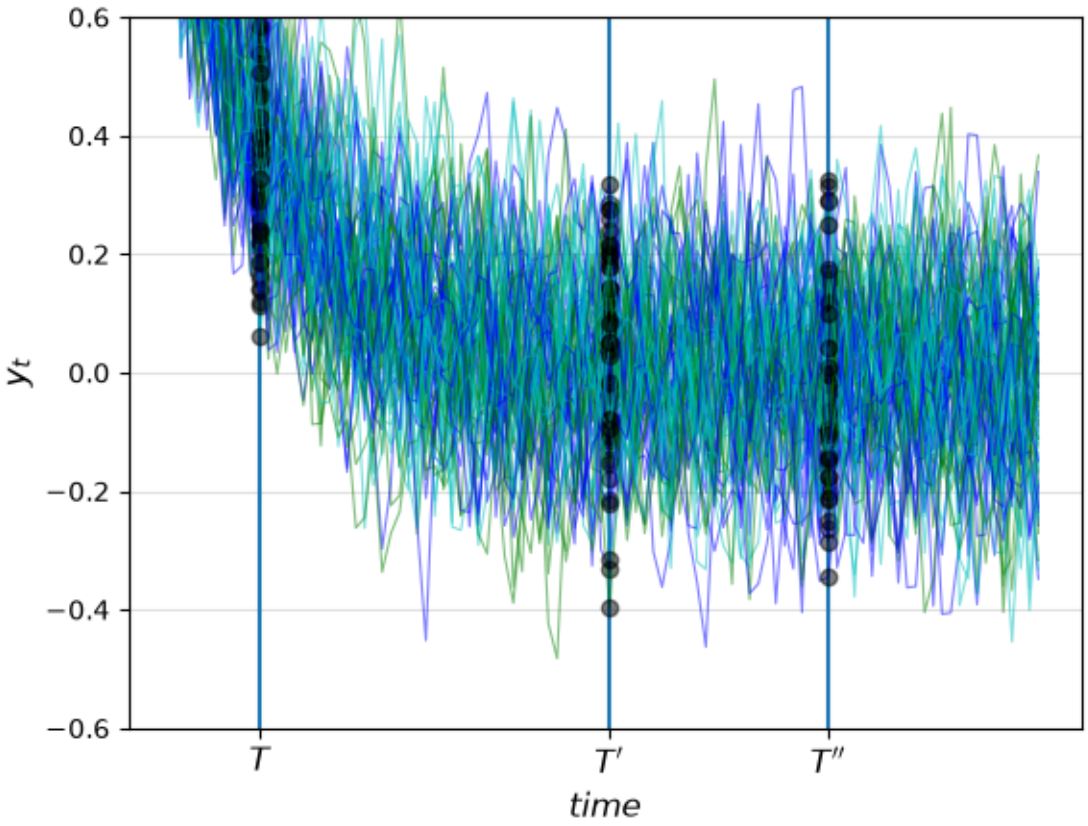
```
def cross_plot(A,
               C,
               G,
               steady_state='False',
               T0 = 10,
               T1 = 50,
               T2 = 75,
               T4 = 100):

    ar = LinearStateSpace(A, C, G, mu_0=np.ones(4))

    if steady_state == 'True':
        μ_x, μ_y, Σ_x, Σ_y, Σ_yx = ar.stationary_distributions()
        ar_state = LinearStateSpace(A, C, G, mu_0=μ_x, Sigma_0=Σ_x)

    ymin, ymax = -0.6, 0.6
    fig, ax = plt.subplots()
```

(continues on next page)

(continued from previous page)

```
    ax.grid(alpha=0.4)
    ax.set_ylim(ymin, ymax)
    ax.set_ylabel('$y_t$', fontsize=12)
    ax.set_xlabel('$time$', fontsize=12)

    ax.vlines((T0, T1, T2), -1.5, 1.5)
    ax.set_xticks((T0, T1, T2))
    ax.set_xticklabels(("$T$", "$T'$", "$T''$"), fontsize=12)
    for i in range(80):
        rcolor = random.choice(('c', 'g', 'b'))

        if steady_state == 'True':
            x, y = ar_state.simulate(ts_length=T4)
        else:
            x, y = ar.simulate(ts_length=T4)

        y = y.flatten()
        ax.plot(y, color=rcolor, lw=0.8, alpha=0.5)
        ax.plot((T0, T1, T2), (y[T0], y[T1], y[T2],), 'ko', alpha=0.5)
    plt.show()
```

```
cross_plot(A_2, C_2, G_2)
```



Note how the time series "settle down" in the sense that the distributions at T' and T'' are relatively similar to each other — but unlike the distribution at T.

Apparently, the distributions of y_t converge to a fixed long-run distribution as $t \to \infty$.

When such a distribution exists it is called a **stationary distribution**.

40.4.2 Stationary Distributions

In our setting, a distribution ψ_∞ is said to be **stationary** for x_t if

$$x_t \sim \psi_\infty \quad \text{and} \quad x_{t+1} = Ax_t + Cw_{t+1} \quad \Longrightarrow \quad x_{t+1} \sim \psi_\infty$$

Since

1. in the present case, all distributions are Gaussian
2. a Gaussian distribution is pinned down by its mean and variance-covariance matrix

we can restate the definition as follows: ψ_∞ is stationary for x_t if

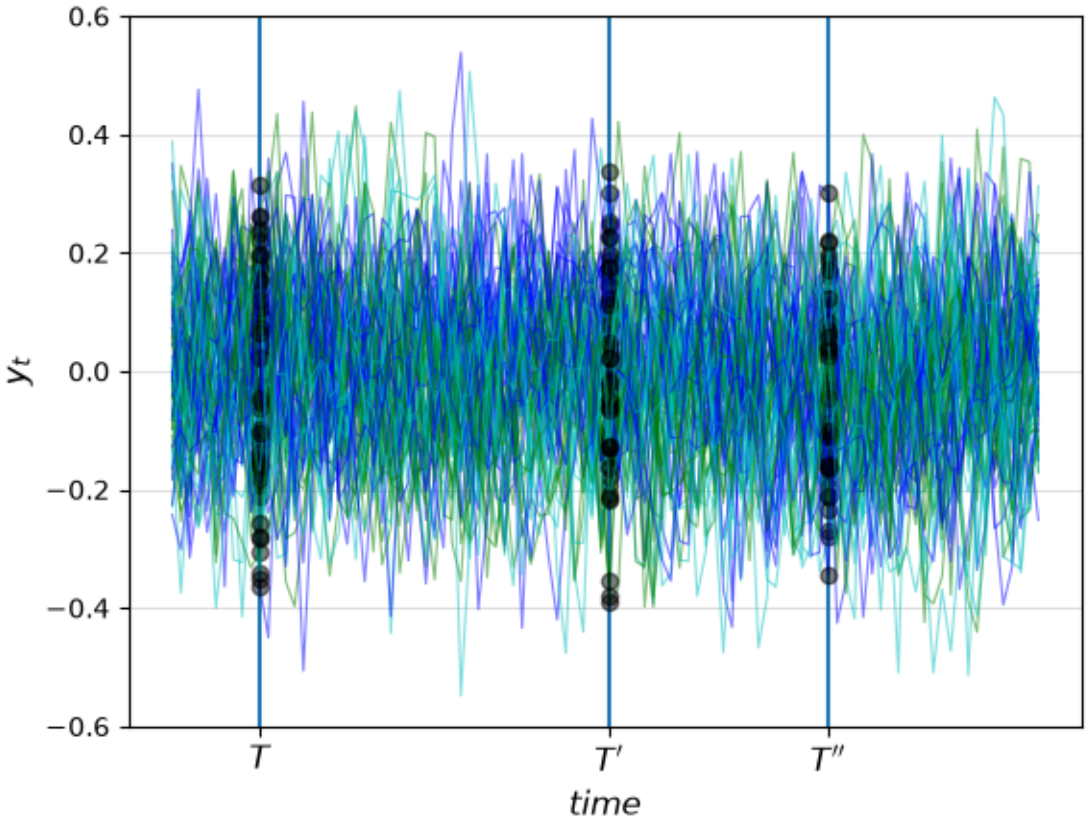
$$\psi_\infty = N(\mu_\infty, \Sigma_\infty)$$

where μ_∞ and Σ_∞ are fixed points of (40.4) and (40.5) respectively.

40.4.3 Covariance Stationary Processes

Let's see what happens to the preceding figure if we start x_0 at the stationary distribution.

```
cross_plot(A_2, C_2, G_2, steady_state='True')
```



Now the differences in the observed distributions at T, T' and T'' come entirely from random fluctuations due to the finite sample size.

By

- our choosing $x_0 \sim N(\mu_\infty, \Sigma_\infty)$
- the definitions of μ_∞ and Σ_∞ as fixed points of (40.4) and (40.5) respectively

we've ensured that

$$\mu_t = \mu_\infty \quad \text{and} \quad \Sigma_t = \Sigma_\infty \quad \text{for all } t$$

Moreover, in view of (40.12), the autocovariance function takes the form $\Sigma_{t+j,t} = A^j \Sigma_\infty$, which depends on j but not on t.

This motivates the following definition.

A process $\{x_t\}$ is said to be **covariance stationary** if

- both μ_t and Σ_t are constant in t
- $\Sigma_{t+j,t}$ depends on the time gap j but not on time t

In our setting, $\{x_t\}$ will be covariance stationary if μ_0, Σ_0, A, C assume values that imply that none of $\mu_t, \Sigma_t, \Sigma_{t+j,t}$ depends on t.

40.4.4 Conditions for Stationarity

The Globally Stable Case

The difference equation $\mu_{t+1} = A\mu_t$ is known to have *unique* fixed point $\mu_\infty = 0$ if all eigenvalues of A have moduli strictly less than unity.

That is, if `(np.absolute(np.linalg.eigvals(A)) < 1).all() == True`.

The difference equation (40.5) also has a unique fixed point in this case, and, moreover

$$\mu_t \to \mu_\infty = 0 \quad \text{and} \quad \Sigma_t \to \Sigma_\infty \quad \text{as} \quad t \to \infty$$

regardless of the initial conditions μ_0 and Σ_0.

This is the *globally stable case* — see these notes for more a theoretical treatment.

However, global stability is more than we need for stationary solutions, and often more than we want.

To illustrate, consider *our second order difference equation example*.

Here the state is $x_t = \begin{bmatrix} 1 & y_t & y_{t-1} \end{bmatrix}'$.

Because of the constant first component in the state vector, we will never have $\mu_t \to 0$.

How can we find stationary solutions that respect a constant state component?

Processes with a Constant State Component

To investigate such a process, suppose that A and C take the form

$$A = \begin{bmatrix} A_1 & a \\ 0 & 1 \end{bmatrix} \qquad C = \begin{bmatrix} C_1 \\ 0 \end{bmatrix}$$

where

- A_1 is an $(n-1) \times (n-1)$ matrix
- a is an $(n-1) \times 1$ column vector

Let $x_t = \begin{bmatrix} x_{1t}' & 1 \end{bmatrix}'$ where x_{1t} is $(n-1) \times 1$.

It follows that

$$x_{1,t+1} = A_1 x_{1t} + a + C_1 w_{t+1}$$

Let $\mu_{1t} = \mathbb{E}[x_{1t}]$ and take expectations on both sides of this expression to get

$$\mu_{1,t+1} = A_1 \mu_{1,t} + a \tag{40.13}$$

Assume now that the moduli of the eigenvalues of A_1 are all strictly less than one.

Then (40.13) has a unique stationary solution, namely,

$$\mu_{1\infty} = (I - A_1)^{-1} a$$

The stationary value of μ_t itself is then $\mu_\infty := \begin{bmatrix} \mu_{1\infty}' & 1 \end{bmatrix}'$.

The stationary values of Σ_t and $\Sigma_{t+j,t}$ satisfy

$$\begin{aligned} \Sigma_\infty &= A \Sigma_\infty A' + CC' \\ \Sigma_{t+j,t} &= A^j \Sigma_\infty \end{aligned}$$

Notice that here $\Sigma_{t+j,t}$ depends on the time gap j but not on calendar time t.

In conclusion, if

- $x_0 \sim N(\mu_\infty, \Sigma_\infty)$ and
- the moduli of the eigenvalues of A_1 are all strictly less than unity

then the $\{x_t\}$ process is covariance stationary, with constant state component.

Note

If the eigenvalues of A_1 are less than unity in modulus, then (a) starting from any initial value, the mean and variance-covariance matrix both converge to their stationary values; and (b) iterations on (40.5) converge to the fixed point of the *discrete Lyapunov equation* in the first line of (40.14).

40.4.5 Ergodicity

Let's suppose that we're working with a covariance stationary process.

In this case, we know that the ensemble mean will converge to μ_∞ as the sample size I approaches infinity.

Averages over Time

Ensemble averages across simulations are interesting theoretically, but in real life, we usually observe only a *single* realization $\{x_t, y_t\}_{t=0}^T$.

So now let's take a single realization and form the time-series averages

$$\bar{x} := \frac{1}{T}\sum_{t=1}^{T} x_t \quad \text{and} \quad \bar{y} := \frac{1}{T}\sum_{t=1}^{T} y_t$$

Do these time series averages converge to something interpretable in terms of our basic state-space representation?

The answer depends on something called *ergodicity*.

Ergodicity is the property that time series and ensemble averages coincide.

More formally, ergodicity implies that time series sample averages converge to their expectation under the stationary distribution.

In particular,

- $\frac{1}{T}\sum_{t=1}^{T} x_t \to \mu_\infty$
- $\frac{1}{T}\sum_{t=1}^{T} (x_t - \bar{x}_T)(x_t - \bar{x}_T)' \to \Sigma_\infty$
- $\frac{1}{T}\sum_{t=1}^{T} (x_{t+j} - \bar{x}_T)(x_t - \bar{x}_T)' \to A^j\Sigma_\infty$

In our linear Gaussian setting, any covariance stationary process is also ergodic.

40.5 Noisy Observations

In some settings, the observation equation $y_t = Gx_t$ is modified to include an error term.

Often this error term represents the idea that the true state can only be observed imperfectly.

To include an error term in the observation we introduce

- An IID sequence of $\ell \times 1$ random vectors $v_t \sim N(0, I)$.
- A $k \times \ell$ matrix H.

and extend the linear state-space system to

$$\begin{aligned} x_{t+1} &= Ax_t + Cw_{t+1} \\ y_t &= Gx_t + Hv_t \\ x_0 &\sim N(\mu_0, \Sigma_0) \end{aligned}$$

The sequence $\{v_t\}$ is assumed to be independent of $\{w_t\}$.

The process $\{x_t\}$ is not modified by noise in the observation equation and its moments, distributions and stability properties remain the same.

The unconditional moments of y_t from (40.6) and (40.7) now become

$$\mathbb{E}[y_t] = \mathbb{E}[Gx_t + Hv_t] = G\mu_t \tag{40.14}$$

The variance-covariance matrix of y_t is easily shown to be

$$\text{Var}[y_t] = \text{Var}[Gx_t + Hv_t] = G\Sigma_t G' + HH' \tag{40.15}$$

The distribution of y_t is therefore

$$y_t \sim N(G\mu_t, G\Sigma_t G' + HH')$$

40.6 Prediction

The theory of prediction for linear state space systems is elegant and simple.

40.6.1 Forecasting Formulas – Conditional Means

The natural way to predict variables is to use conditional distributions.

For example, the optimal forecast of x_{t+1} given information known at time t is

$$\mathbb{E}_t[x_{t+1}] := \mathbb{E}[x_{t+1} \mid x_t, x_{t-1}, \ldots, x_0] = Ax_t$$

The right-hand side follows from $x_{t+1} = Ax_t + Cw_{t+1}$ and the fact that w_{t+1} is zero mean and independent of $x_t, x_{t-1}, \ldots, x_0$.

That $\mathbb{E}_t[x_{t+1}] = \mathbb{E}[x_{t+1} \mid x_t]$ is an implication of $\{x_t\}$ having the **Markov property**.

The one-step-ahead forecast error is

$$x_{t+1} - \mathbb{E}_t[x_{t+1}] = Cw_{t+1}$$

The covariance matrix of the forecast error is

$$\mathbb{E}[(x_{t+1} - \mathbb{E}_t[x_{t+1}])(x_{t+1} - \mathbb{E}_t[x_{t+1}])'] = CC'$$

More generally, we'd like to compute the j-step ahead forecasts $\mathbb{E}_t[x_{t+j}]$ and $\mathbb{E}_t[y_{t+j}]$.

With a bit of algebra, we obtain

$$x_{t+j} = A^j x_t + A^{j-1} Cw_{t+1} + A^{j-2} Cw_{t+2} + \cdots + A^0 Cw_{t+j}$$

In view of the IID property, current and past state values provide no information about future values of the shock.

Hence $\mathbb{E}_t[w_{t+k}] = \mathbb{E}[w_{t+k}] = 0$.

It now follows from linearity of expectations that the j-step ahead forecast of x is

$$\mathbb{E}_t[x_{t+j}] = A^j x_t$$

The j-step ahead forecast of y is therefore

$$\mathbb{E}_t[y_{t+j}] = \mathbb{E}_t[Gx_{t+j} + Hv_{t+j}] = GA^j x_t$$

40.6.2 Covariance of Prediction Errors

It is useful to obtain the covariance matrix of the vector of j-step-ahead prediction errors

$$x_{t+j} - \mathbb{E}_t[x_{t+j}] = \sum_{s=0}^{j-1} A^s C w_{t-s+j} \tag{40.16}$$

Evidently,

$$V_j := \mathbb{E}_t[(x_{t+j} - \mathbb{E}_t[x_{t+j}])(x_{t+j} - \mathbb{E}_t[x_{t+j}])'] = \sum_{k=0}^{j-1} A^k C C' A^{k'} \tag{40.17}$$

V_j defined in (40.17) can be calculated recursively via $V_1 = CC'$ and

$$V_j = CC' + AV_{j-1}A', \quad j \geq 2 \tag{40.18}$$

V_j is the **conditional covariance matrix** of the errors in forecasting x_{t+j}, conditioned on time t information x_t.

Under particular conditions, V_j converges to

$$V_\infty = CC' + AV_\infty A' \tag{40.19}$$

Equation (40.19) is an example of a **discrete Lyapunov** equation in the covariance matrix V_∞.

A sufficient condition for V_j to converge is that the eigenvalues of A be strictly less than one in modulus.

Weaker sufficient conditions for convergence associate eigenvalues equaling or exceeding one in modulus with elements of C that equal 0.

40.7 Code

Our preceding simulations and calculations are based on code in the file lss.py from the QuantEcon.py package.

The code implements a class for handling linear state space models (simulations, calculating moments, etc.).

One Python construct you might not be familiar with is the use of a generator function in the method `moment_sequence()`.

Go back and read the relevant documentation if you've forgotten how generator functions work.

Examples of usage are given in the solutions to the exercises.

40.8 Exercises

Exercise 40.8.1

In several contexts, we want to compute forecasts of geometric sums of future random variables governed by the linear state-space system (40.1).

We want the following objects

- Forecast of a geometric sum of future x's, or $\mathbb{E}_t\left[\sum_{j=0}^{\infty} \beta^j x_{t+j}\right]$.
- Forecast of a geometric sum of future y's, or $\mathbb{E}_t\left[\sum_{j=0}^{\infty} \beta^j y_{t+j}\right]$.

These objects are important components of some famous and interesting dynamic models.

For example,

- if $\{y_t\}$ is a stream of dividends, then $\mathbb{E}\left[\sum_{j=0}^{\infty} \beta^j y_{t+j} | x_t\right]$ is a model of a stock price
- if $\{y_t\}$ is the money supply, then $\mathbb{E}\left[\sum_{j=0}^{\infty} \beta^j y_{t+j} | x_t\right]$ is a model of the price level

Show that:

$$\mathbb{E}_t \left[\sum_{j=0}^{\infty} \beta^j x_{t+j}\right] = [I - \beta A]^{-1} x_t$$

and

$$\mathbb{E}_t \left[\sum_{j=0}^{\infty} \beta^j y_{t+j}\right] = G[I - \beta A]^{-1} x_t$$

what must the modulus for every eigenvalue of A be less than?

Solution

Suppose that every eigenvalue of A has modulus strictly less than $\frac{1}{\beta}$.

It *then follows* that $I + \beta A + \beta^2 A^2 + \cdots = [I - \beta A]^{-1}$.

This leads to our formulas:

- Forecast of a geometric sum of future x's

$$\mathbb{E}_t \left[\sum_{j=0}^{\infty} \beta^j x_{t+j}\right] = [I + \beta A + \beta^2 A^2 + \cdots] x_t = [I - \beta A]^{-1} x_t$$

- Forecast of a geometric sum of future y's

$$\mathbb{E}_t \left[\sum_{j=0}^{\infty} \beta^j y_{t+j}\right] = G[I + \beta A + \beta^2 A^2 + \cdots] x_t = G[I - \beta A]^{-1} x_t$$

CHAPTER

FORTYONE

SAMUELSON MULTIPLIER-ACCELERATOR

Contents

- *Samuelson Multiplier-Accelerator*
  - *Overview*
  - *Details*
  - *Implementation*
  - *Stochastic shocks*
  - *Government spending*
  - *Wrapping everything into a class*
  - *Using the LinearStateSpace class*
  - *Pure multiplier model*
  - *Summary*

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install quantecon
```

41.1 Overview

This lecture creates non-stochastic and stochastic versions of Paul Samuelson's celebrated multiplier accelerator model [Samuelson, 1939].

In doing so, we extend the example of the Solow model class in our second OOP lecture.

Our objectives are to

- provide a more detailed example of OOP and classes
- review a famous model
- review linear difference equations, both deterministic and stochastic

Let's start with some standard imports:

```
import matplotlib.pyplot as plt
import numpy as np
```

We'll also use the following for various tasks described below:

```
from quantecon import LinearStateSpace
import cmath
import math
import sympy
from sympy import Symbol, init_printing
from cmath import sqrt
```

41.1.1 Samuelson's model

Samuelson used a *second-order linear difference equation* to represent a model of national output based on three components:

- a *national output identity* asserting that national output or national income is the sum of consumption plus investment plus government purchases.
- a Keynesian *consumption function* asserting that consumption at time t is equal to a constant times national output at time $t-1$.
- an investment *accelerator* asserting that investment at time t equals a constant called the *accelerator coefficient* times the difference in output between period $t-1$ and $t-2$.

Consumption plus investment plus government purchases constitute *aggregate demand,* which automatically calls forth an equal amount of *aggregate supply*.

(To read about linear difference equations see here or chapter IX of [Sargent, 1987].)

Samuelson used the model to analyze how particular values of the marginal propensity to consume and the accelerator coefficient might give rise to transient **business cycles** in national output.

Possible dynamic properties include

- smooth convergence to a constant level of output
- damped business cycles that eventually converge to a constant level of output
- persistent business cycles that neither dampen nor explode

Later we present an extension that adds a random shock to the right side of the national income identity representing random fluctuations in aggregate demand.

This modification makes national output become governed by a second-order **stochastic linear difference equation** that, with appropriate parameter values, gives rise to recurrent irregular business cycles.

(To read about stochastic linear difference equations see chapter XI of [Sargent, 1987].)

41.2 Details

Let's assume that

- $\{G_t\}$ is a sequence of levels of government expenditures – we'll start by setting $G_t = G$ for all t.
- $\{C_t\}$ is a sequence of levels of aggregate consumption expenditures, a key endogenous variable in the model.
- $\{I_t\}$ is a sequence of rates of investment, another key endogenous variable.
- $\{Y_t\}$ is a sequence of levels of national income, yet another endogenous variable.
- α is the marginal propensity to consume in the Keynesian consumption function $C_t = \alpha Y_{t-1} + \gamma$.
- β is the "accelerator coefficient" in the "investment accelerator" $I_t = \beta(Y_{t-1} - Y_{t-2})$.
- $\{\epsilon_t\}$ is an IID sequence standard normal random variables.
- $\sigma \geq 0$ is a "volatility" parameter — setting $\sigma = 0$ recovers the non-stochastic case that we'll start with.

The model combines the consumption function

$$C_t = \alpha Y_{t-1} + \gamma \tag{41.1}$$

with the investment accelerator

$$I_t = \beta(Y_{t-1} - Y_{t-2}) \tag{41.2}$$

and the national income identity

$$Y_t = C_t + I_t + G_t \tag{41.3}$$

- The parameter α is peoples' **marginal propensity to consume** out of income - equation (41.1) asserts that people consume a fraction of $\alpha \in (0, 1)$ of each additional dollar of income.
- The parameter $\beta > 0$ is the investment accelerator coefficient - equation (41.2) asserts that people invest in physical capital when income is increasing and disinvest when it is decreasing.

Equations (41.1), (41.2), and (41.3) imply the following second-order linear difference equation for national income:

$$Y_t = (\alpha + \beta)Y_{t-1} - \beta Y_{t-2} + (\gamma + G_t)$$

or

$$Y_t = \rho_1 Y_{t-1} + \rho_2 Y_{t-2} + (\gamma + G_t) \tag{41.4}$$

where $\rho_1 = (\alpha + \beta)$ and $\rho_2 = -\beta$.

To complete the model, we require two *initial conditions*.

If the model is to generate time series for $t = 0, \ldots, T$, we require initial values

$$Y_{-1} = \bar{Y}_{-1}, \quad Y_{-2} = \bar{Y}_{-2}$$

We'll ordinarily set the parameters (α, β) so that starting from an arbitrary pair of initial conditions $(\bar{Y}_{-1}, \bar{Y}_{-2})$, national income Y_t converges to a constant value as t becomes large.

We are interested in studying

- the transient fluctuations in Y_t as it converges to its **steady state** level
- the *rate* at which it converges to a steady state level

The deterministic version of the model described so far — meaning that no random shocks hit aggregate demand — has only transient fluctuations.

We can convert the model to one that has persistent irregular fluctuations by adding a random shock to aggregate demand.

41.2.1 Stochastic version of the model

We create a *random* or *stochastic* version of the model by adding a random process of *shocks* or *disturbances* $\{\sigma\epsilon_t\}$ to the right side of equation (41.4), leading to the *second-order scalar linear stochastic difference equation*:

$$Y_t = (\alpha + \beta)Y_{t-1} - \beta Y_{t-2} + (\gamma + G_t) + \sigma\epsilon_t \tag{41.5}$$

41.2.2 Mathematical analysis of the model

To get started, let's set $G_t \equiv 0$, $\sigma = 0$, and $\gamma = 0$.

Then we can write equation (41.5) as

$$Y_t = \rho_1 Y_{t-1} + \rho_2 Y_{t-2}$$

or

$$Y_{t+2} - \rho_1 Y_{t+1} - \rho_2 Y_t = 0 \tag{41.6}$$

To discover the properties of the solution of (41.6), it is useful first to form the **characteristic polynomial** for (41.6):

$$z^2 - \rho_1 z - \rho_2 \tag{41.7}$$

where z is possibly a complex number.

We want to find the two **zeros** (a.k.a. **roots**) – namely λ_1, λ_2 – of the characteristic polynomial.

These are two special values of z, say $z = \lambda_1$ and $z = \lambda_2$, such that if we set z equal to one of these values in expression (41.7), the characteristic polynomial (41.7) equals zero:

$$z^2 - \rho_1 z - \rho_2 = (z - \lambda_1)(z - \lambda_2) = 0 \tag{41.8}$$

Equation (41.8) is said to *factor* the characteristic polynomial.

When the roots are complex, they will occur as a complex conjugate pair.

When the roots are complex, it is convenient to represent them in the polar form

$$\lambda_1 = re^{i\omega}, \ \lambda_2 = re^{-i\omega}$$

where r is the *amplitude* of the complex number and ω is its *angle* or *phase*.

These can also be represented as

$$\lambda_1 = r(cos(\omega) + i \sin(\omega))$$

$$\lambda_2 = r(cos(\omega) - i \sin(\omega))$$

(To read about the polar form, see here)

Given *initial conditions* Y_{-1}, Y_{-2}, we want to generate a *solution* of the difference equation (41.6).

It can be represented as

$$Y_t = \lambda_1^t c_1 + \lambda_2^t c_2$$

where c_1 and c_2 are constants that depend on the two initial conditions and on ρ_1, ρ_2.

When the roots are complex, it is useful to pursue the following calculations.

Notice that

$$\begin{aligned} Y_t &= c_1(re^{i\omega})^t + c_2(re^{-i\omega})^t \\ &= c_1 r^t e^{i\omega t} + c_2 r^t e^{-i\omega t} \\ &= c_1 r^t [\cos(\omega t) + i \sin(\omega t)] + c_2 r^t [\cos(\omega t) - i \sin(\omega t)] \\ &= (c_1 + c_2) r^t \cos(\omega t) + i(c_1 - c_2) r^t \sin(\omega t) \end{aligned}$$

The only way that Y_t can be a real number for each t is if $c_1 + c_2$ is a real number and $c_1 - c_2$ is an imaginary number.

This happens only when c_1 and c_2 are complex conjugates, in which case they can be written in the polar forms

$$c_1 = ve^{i\theta}, \quad c_2 = ve^{-i\theta}$$

So we can write

$$\begin{aligned} Y_t = &\quad ve^{i\theta} r^t e^{i\omega t} + ve^{-i\theta} r^t e^{-i\omega t} \\ = &\quad vr^t [e^{i(\omega t + \theta)} + e^{-i(\omega t + \theta)}] \\ = &\quad 2vr^t \cos(\omega t + \theta) \end{aligned}$$

where v and θ are constants that must be chosen to satisfy initial conditions for Y_{-1}, Y_{-2}.

This formula shows that when the roots are complex, Y_t displays oscillations with *period* $\breve{p} = \frac{2\pi}{\omega}$ and *damping factor* r.

We say that $\breve{p}$ is the *period* because in that amount of time the cosine wave $\cos(\omega t + \theta)$ goes through exactly one complete cycles.

(Draw a cosine function to convince yourself of this please)

Remark: Following [Samuelson, 1939], we want to choose the parameters α, β of the model so that the absolute values (of the possibly complex) roots λ_1, λ_2 of the characteristic polynomial are both strictly less than one:

$$|\lambda_j| < 1 \qquad \text{for } j = 1, 2$$

Remark: When both roots λ_1, λ_2 of the characteristic polynomial have absolute values strictly less than one, the absolute value of the larger one governs the rate of convergence to the steady state of the non stochastic version of the model.

41.2.3 Things this lecture does

We write a function to generate simulations of a $\{Y_t\}$ sequence as a function of time.

The function requires that we put in initial conditions for Y_{-1}, Y_{-2}.

The function checks that α, β are set so that λ_1, λ_2 are less than unity in absolute value (also called "modulus").

The function also tells us whether the roots are complex, and, if they are complex, returns both their real and complex parts.

If the roots are both real, the function returns their values.

We use our function written to simulate paths that are stochastic (when $\sigma > 0$).

We have written the function in a way that allows us to input $\{G_t\}$ paths of a few simple forms, e.g.,

- one time jumps in G at some time
- a permanent jump in G that occurs at some time

We proceed to use the Samuelson multiplier-accelerator model as a laboratory to make a simple OOP example.

The "state" that determines next period's Y_{t+1} is now not just the current value Y_t but also the once lagged value Y_{t-1}.

This involves a little more bookkeeping than is required in the Solow model class definition.

We use the Samuelson multiplier-accelerator model as a vehicle for teaching how we can gradually add more features to the class.

We want to have a method in the class that automatically generates a simulation, either non-stochastic ($\sigma = 0$) or stochastic ($\sigma > 0$).

We also show how to map the Samuelson model into a simple instance of the `LinearStateSpace` class described here.

We can use a `LinearStateSpace` instance to do various things that we did above with our homemade function and class.

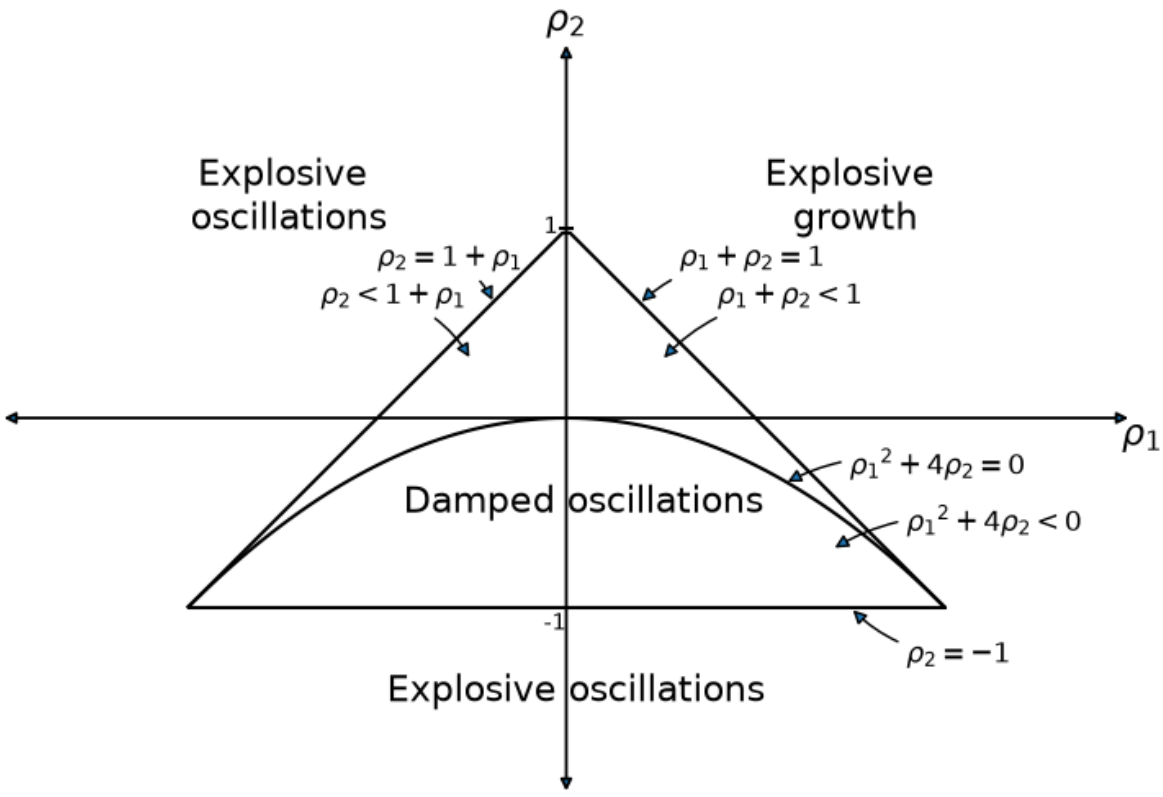
Among other things, we show by example that the eigenvalues of the matrix A that we use to form the instance of the `LinearStateSpace` class for the Samuelson model equal the roots of the characteristic polynomial (41.7) for the Samuelson multiplier accelerator model.

Here is the formula for the matrix A in the linear state space system in the case that government expenditures are a constant G:

$$A = \begin{bmatrix} 1 & 0 & 0 \\ \gamma + G & \rho_1 & \rho_2 \\ 0 & 1 & 0 \end{bmatrix}$$

41.3 Implementation

We'll start by drawing an informative graph from page 189 of [Sargent, 1987]



The graph portrays regions in which the (λ_1, λ_2) root pairs implied by the $(\rho_1 = (\alpha + \beta), \rho_2 = -\beta)$ difference equation parameter pairs in the Samuelson model are such that:

- (λ_1, λ_2) are complex with modulus less than 1 - in this case, the $\{Y_t\}$ sequence displays damped oscillations.
- (λ_1, λ_2) are both real, but one is strictly greater than 1 - this leads to explosive growth.
- (λ_1, λ_2) are both real, but one is strictly less than -1 - this leads to explosive oscillations.
- (λ_1, λ_2) are both real and both are less than 1 in absolute value - in this case, there is smooth convergence to the steady state without damped cycles.

Later we'll present the graph with a red mark showing the particular point implied by the setting of (α, β).

41.3.1 Function to describe implications of characteristic polynomial

```
def categorize_solution(ρ1, ρ2):
    """
    This function takes values of ρ1 and ρ2 and uses them
    to classify the type of solution.
    """
    discriminant = ρ1**2 + 4 * ρ2
    if ρ2 > 1 + ρ1 or ρ2 < -1:
        return "Explosive oscillations"
    elif ρ1 + ρ2 > 1:
        return "Explosive growth"
    elif discriminant < 0:
        return "Damped oscillations"
    else:
        return "Steady state convergence"

def analyze_roots(α, β, verbose=True):
    """
    Unified function to calculate roots and analyze their properties.
    """
    ρ1 = α + β
    ρ2 = -β

    # Compute characteristic polynomial roots
    roots = np.roots([1, -ρ1, -ρ2])

    # Classify solution type
    solution_type = categorize_solution(ρ1, ρ2)

    # Determine root properties
    is_complex = all(isinstance(root, complex) for root in roots)
    is_stable = all(abs(root) < 1 for root in roots)

    if verbose:
        print(f"ρ1 = {ρ1:.2f}, ρ2 = {ρ2:.2f}")
        print(f"Roots: {[f'{root:.2f}' for root in roots]}")
        print(f"Root type: {'Complex' if is_complex else 'Real'}")
        print(f"Stability: {'Stable' if is_stable else 'Unstable'}")
        print(f"Solution type: {solution_type}")

    return {
        'roots': roots,
        'rho1': ρ1,
        'rho2': ρ2,
        'is_complex': is_complex,
        'is_stable': is_stable,
```

(continues on next page)

(continued from previous page)

```
        'solution_type': solution_type
    }
```

We also write a unified simulation function that can handle both non-stochastic and stochastic versions of the model.

It allows for government spending paths of a few simple forms which we specify via a dictionary `g_params`

```
def simulate_samuelson(
    y_0, y_1, α, β, γ=10, n=100, σ=0, g_params=None, seed=0
):
    """
    Unified simulation function for Samuelson model.

    Parameters:
    g_params: dict with keys 'g', 'g_t', 'duration' for government spending
    seed: random seed for reproducible results
    """
    analysis = analyze_roots(α, β, verbose=False)
    ρ1, ρ2 = analysis['rho1'], analysis['rho2']

    # Initialize time series
    y_t = [y_0, y_1]

    # Generate shocks if stochastic
    if σ > 0:
        np.random.seed(seed)
        ϵ = np.random.normal(0, 1, n)

    # Simulate forward
    for t in range(2, n):

        # Determine government spending
        g = 0
        if g_params:
            g_val, g_t_val = g_params.get('g', 0), g_params.get('g_t', 0)
            duration = g_params.get('duration', None)
            if duration == 'permanent' and t >= g_t_val:
                g = g_val
            elif duration == 'one-off' and t == g_t_val:
                g = g_val
            elif duration is None:
                g = g_val

        # Calculate next value
        y_next = ρ1 * y_t[t-1] + ρ2 * y_t[t-2] + γ + g
        if σ > 0:
            y_next += σ * ϵ[t]

        y_t.append(y_next)

    return y_t, analysis
```

We will use this function to run simulations of the model.

But before doing that, let's test the analysis function

```
analysis = analyze_roots(α=1.3, β=0.4)
```

```
ρ1 = 1.70, ρ2 = -0.40
Roots: ['1.42', '0.28']
Root type: Real
Stability: Unstable
Solution type: Explosive growth
```

41.3.2 Function for plotting paths

A useful function for our work below is

```
def plot_y(function=None):
    """Function plots path of Y_t"""

    plt.subplots(figsize=(10, 6))
    plt.plot(function)
    plt.xlabel("$t$")
    plt.ylabel("$Y_t$")
    plt.show()
```

41.3.3 Manual or "by hand" root calculations

The following function calculates roots of the characteristic polynomial using high school algebra.

(We'll calculate the roots in other ways later using `analyze_roots`.)

The function also plots a Y_t starting from initial conditions that we set

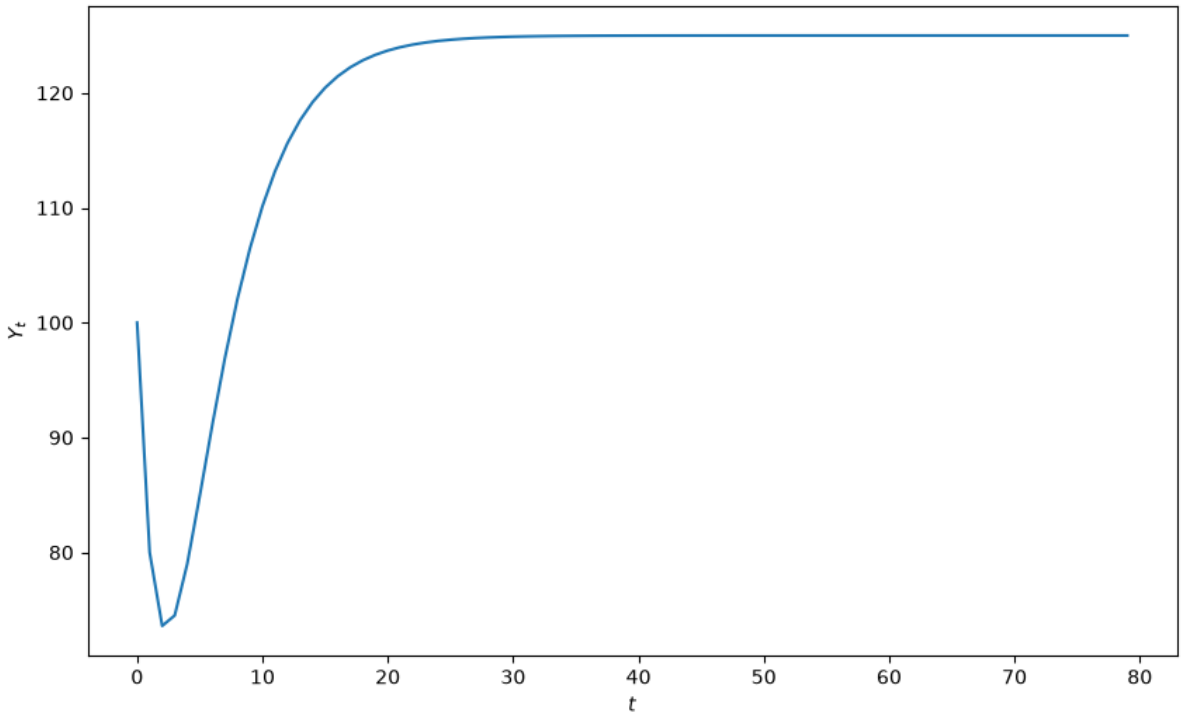
```
def y_nonstochastic(y_0=100, y_1=80, α=0.92, β=0.5, γ=10, n=80):
    """
    This function calculates the roots of the characteristic polynomial
    by hand and returns a path of y_t starting from initial conditions
    """
    roots = []

    ρ1 = α + β
    ρ2 = -β

    print(f"ρ_1 is {ρ1:.2f}")
    print(f"ρ_2 is {ρ2:.2f}")

    discriminant = ρ1**2 + 4 * ρ2

    if discriminant == 0:
        roots.append(-ρ1 / 2)
        print("Single real root: ")
        print("".join(f"{r:.2f}" for r in roots))
    elif discriminant > 0:
        roots.append((-ρ1 + sqrt(discriminant).real) / 2)
        roots.append((-ρ1 - sqrt(discriminant).real) / 2)
        print("Two real roots: ")
        print(" ".join(f"{r:.2f}" for r in roots))
    else:
        roots.append((-ρ1 + sqrt(discriminant)) / 2)
        roots.append((-ρ1 - sqrt(discriminant)) / 2)
        print("Two complex roots: ")
```

(continues on next page)

(continued from previous page)

```
        print(" ".join(f"{r.real:.2f}{r.imag:+.2f}j" for r in roots))

    if all(abs(root) < 1 for root in roots):
        print("Absolute values of roots are less than one")
    else:
        print("Absolute values of roots are not less than one")

    def transition(x, t):
        return ρ1 * x[t - 1] + ρ2 * x[t - 2] + y

    y_t = [y_0, y_1]

    for t in range(2, n):
        y_t.append(transition(y_t, t))

    return y_t


plot_y(y_nonstochastic())
```

```
ρ_1 is 1.42
ρ_2 is -0.50
Two real roots:
-0.65 -0.77
Absolute values of roots are less than one
```



41.3.4 Reverse-engineering parameters to generate damped cycles

The next cell writes code that takes as inputs the modulus r and phase ϕ of a conjugate pair of complex numbers in polar form

$$\lambda_1 = r\exp(i\phi), \quad \lambda_2 = r\exp(-i\phi)$$

- The code assumes that these two complex numbers are the roots of the characteristic polynomial
- It then reverse-engineers (α, β) and (ρ_1, ρ_2), pairs that would generate those roots

```
def f(r, ϕ):
    """
    Takes modulus r and angle ϕ of complex number r exp(j ϕ)
    and creates ρ1 and ρ2 of characteristic polynomial for which
    r exp(j ϕ) and r exp(- j ϕ) are complex roots.

    Returns the multiplier coefficient $\alpha$
    and the accelerator coefficient $\beta$
    that verifies those roots.
    """
    # Create complex conjugate pair from polar coordinates
    g1 = cmath.rect(r, ϕ)
    g2 = cmath.rect(r, -ϕ)

    # Calculate corresponding ρ1, ρ2 parameters
    ρ1 = g1 + g2
    ρ2 = -g1 * g2

    # Derive α and β coefficients from ρ parameters
    β = -ρ2
    α = ρ1 - β
    return ρ1, ρ2, α, β
```

Now let's use the function in an example.

Here are the example parameters

```
r = 0.95

# Cycle period in time units
period = 10
ϕ = 2 * math.pi / period

# Apply the reverse-engineering function
ρ1, ρ2, α, β = f(r, ϕ)

print(f"α, β = {α:.2f}, {β:.2f}")
print(f"ρ1, ρ2 = {ρ1:.2f}, {ρ2:.2f}")
```

```
α, β = 0.63+0.00j, 0.90-0.00j
ρ1, ρ2 = 1.54+0.00j, -0.90+0.00j
```

The real parts of the roots are

```
print(f"ρ1 = {ρ1.real:.2f}, ρ2 = {ρ2.real:.2f}")
```

```
ρ1 = 1.54, ρ2 = -0.90
```

41.3.5 Root finding using numpy

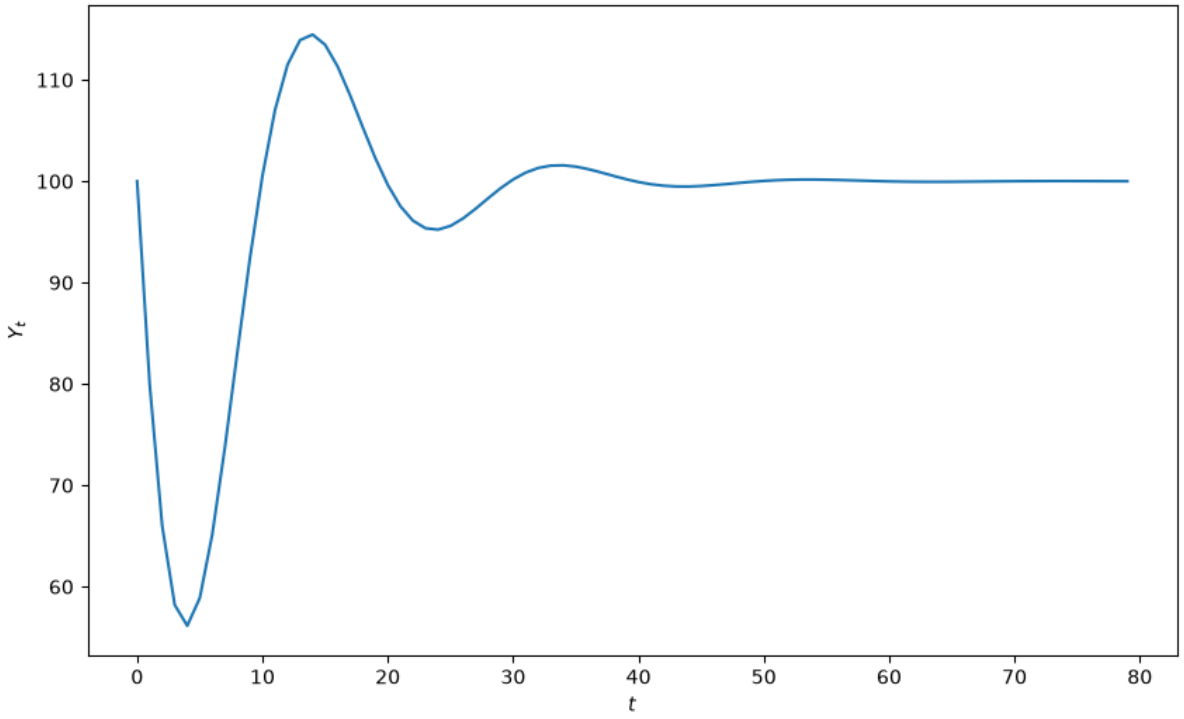
Here we'll use numpy to compute the roots of the characteristic polynomial

```
r1, r2 = np.roots([1, -ρ1, -ρ2])

p1 = cmath.polar(r1)
p2 = cmath.polar(r2)

print(f"r, ϕ = {r:.2f}, {ϕ:.2f}")
print(f"p1, p2 = ({p1[0]:.2f}, {p1[1]:.2f}), ({p2[0]:.2f}, {p2[1]:.2f})")

print(f"α, β = {α:.2f}, {β:.2f}")
print(f"ρ1, ρ2 = {ρ1:.2f}, {ρ2:.2f}")
```

```
r, ϕ = 0.95, 0.63
p1, p2 = (0.95, 0.63), (0.95, -0.63)
α, β = 0.63+0.00j, 0.90-0.00j
ρ1, ρ2 = 1.54+0.00j, -0.90+0.00j
```

```
def y_nonstochastic(y_0=100, y_1=80, α=0.9, β=0.8, γ=10, n=80):
    """
    This function enlists numpy to calculate the roots of the characteristic
    polynomial.
    """

    y_series, analysis = simulate_samuelson(y_0, y_1, α, β, γ, n, 0, None, 42)

    print(f"Solution type: {analysis['solution_type']}")
    print(f"Roots are {analysis['roots']}")
    print(f"Root type: {'Complex' if analysis['is_complex'] else 'Real'}")
    print(f"Stability: {'Stable' if analysis['is_stable'] else 'Unstable'}")

    return y_series


plot_y(y_nonstochastic())
```

```
Solution type: Damped oscillations
Roots are [0.85+0.27838822j 0.85-0.27838822j]
Root type: Complex
Stability: Stable
```



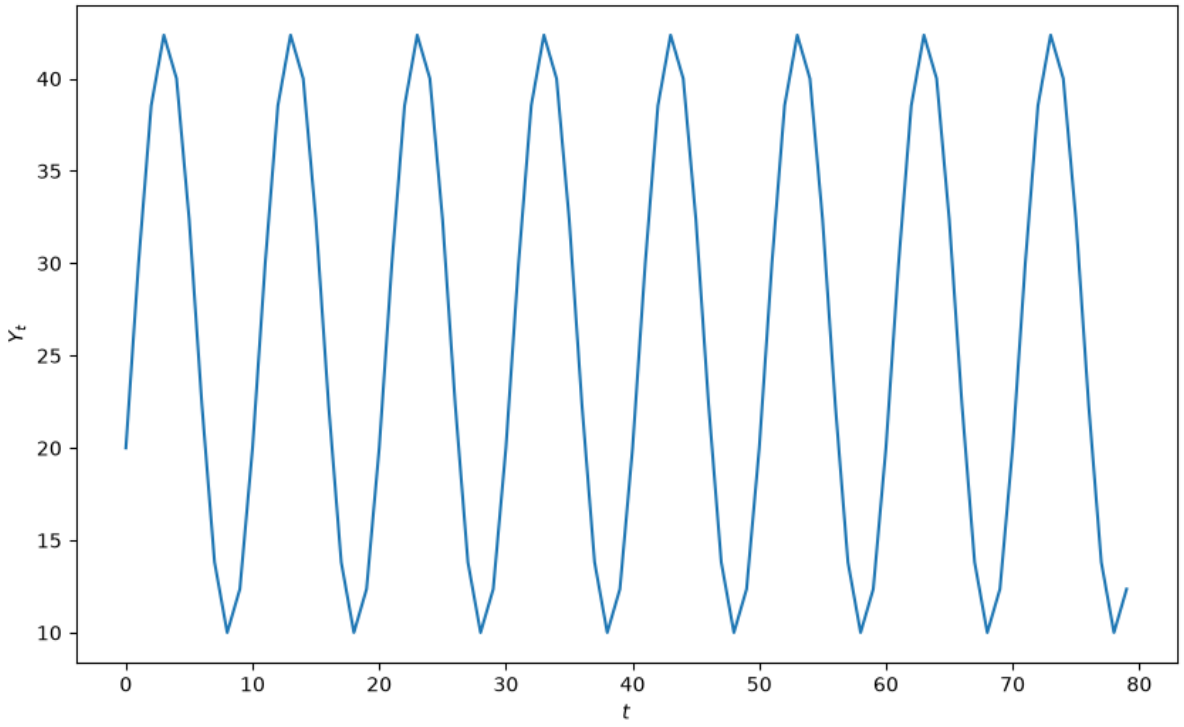
41.3.6 Reverse-engineered complex roots: Example

The next cell studies the implications of reverse-engineered complex roots.

We'll generate an *undamped* cycle of period 10

```
r = 1  # Generates undamped, nonexplosive cycles

period = 10  # Length of cycle in units of time
ϕ = 2 * math.pi / period

# Apply the reverse-engineering function f
ρ1, ρ2, α, β = f(r, ϕ)

# Extract real parts for numerical computation
α = α.real
β = β.real

print(f"α, β = {α:.2f}, {β:.2f}")

ytemp = y_nonstochastic(α=α, β=β, y_0=20, y_1=30)
plot_y(ytemp)
```

```
α, β = 0.62, 1.00
Solution type: Damped oscillations
Roots are [0.80901699+0.58778525j 0.80901699-0.58778525j]
Root type: Complex
Stability: Stable
```



41.3.7 Digression: Using Sympy to find roots

We can also use sympy to compute analytic formulas for the roots

```
init_printing()

r1 = Symbol("ρ_1")
r2 = Symbol("ρ_2")
z = Symbol("z")

sympy.solve(z**2 - r1 * z - r2, z)
```

$$\left[\rho_1/2 - \sqrt{\rho_1^2 + 4\rho_2}/2,\ \rho_1/2 + \sqrt{\rho_1^2 + 4\rho_2}/2\right]$$

```
α = Symbol("α")
β = Symbol("β")
r1 = α + β
r2 = -β

sympy.solve(z**2 - r1 * z - r2, z)
```

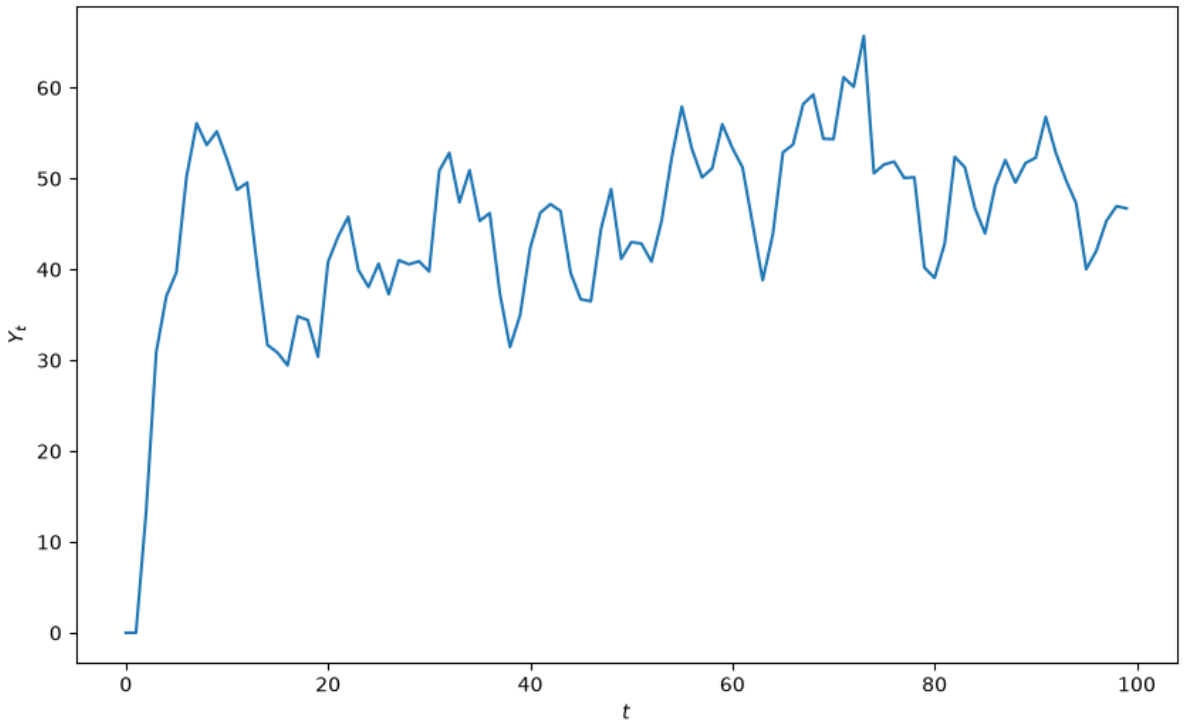
$$\left[\alpha/2 + \beta/2 - \sqrt{\alpha^2 + 2\alpha\beta + \beta^2 - 4\beta}/2,\ \alpha/2 + \beta/2 + \sqrt{\alpha^2 + 2\alpha\beta + \beta^2 - 4\beta}/2\right]$$

41.4 Stochastic shocks

Now we'll construct some code to simulate the stochastic version of the model that emerges when we add a random shock process to aggregate demand

```
def y_stochastic(y_0=0, y_1=0, α=0.8, β=0.2, γ=10, n=100, σ=5):
    """
    This function takes parameters of a stochastic version of
    the model, analyzes the roots of the characteristic
    polynomial and generates a simulation.
    """

    y_series, analysis = simulate_samuelson(y_0, y_1, α, β, γ, n, σ, None, 42)

    print(f"Solution type: {analysis['solution_type']}")
    print(f"Roots are {[f'{root:.2f}' for root in analysis['roots']]}")
    print(f"Root type: {'Complex' if analysis['is_complex'] else 'Real'}")
    print(f"Stability: {'Stable' if analysis['is_stable'] else 'Unstable'}")

    return y_series


plot_y(y_stochastic())
```

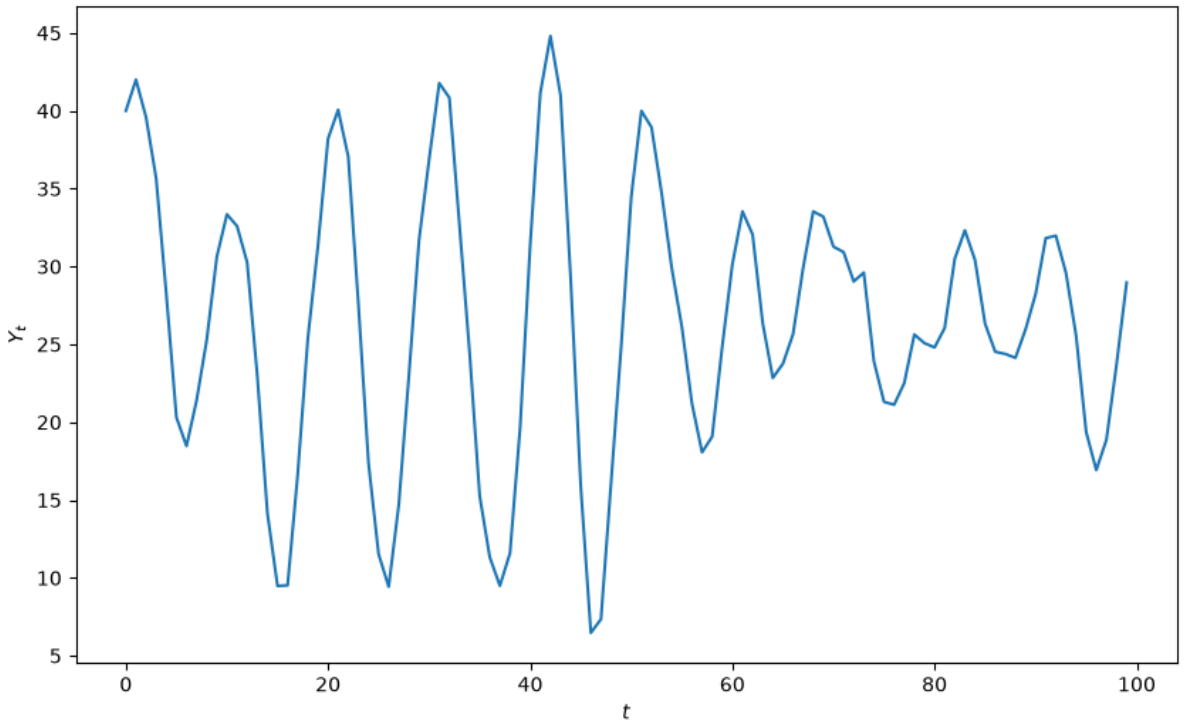
```
Solution type: Steady state convergence
Roots are ['0.72', '0.28']
Root type: Real
Stability: Stable
```



Let's do a simulation in which there are shocks and the characteristic polynomial has complex roots

```
r = 0.97

period = 10  # Length of cycle in units of time
ϕ = 2 * math.pi / period

# Apply the reverse-engineering function f
ρ1, ρ2, α, β = f(r, ϕ)

# Extract real parts for numerical computation
α = α.real
β = β.real

print(f"α, β = {α:.2f}, {β:.2f}")
plot_y(y_stochastic(y_0=40, y_1=42, α=α, β=β, σ=2, n=100))
```

```
α, β = 0.63, 0.94
Solution type: Damped oscillations
Roots are ['0.78+0.57j', '0.78-0.57j']
Root type: Complex
Stability: Stable
```



41.5 Government spending

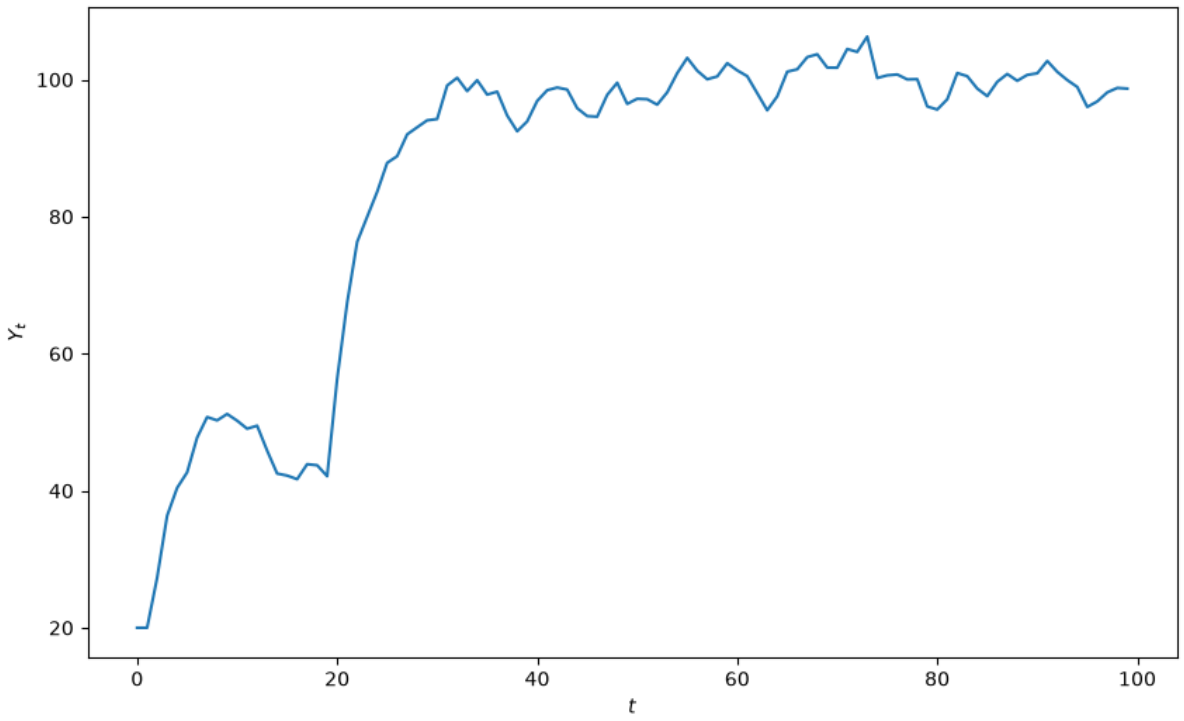
This function computes a response to either a permanent or one-off increase in government expenditures

```
def y_stochastic_g(
    y_0=20, y_1=20, α=0.8, β=0.2, γ=10,
    n=100, σ=2, g=0, g_t=0, duration="permanent"
):
    """
    This program computes a response to a permanent increase
    in government expenditures that occurs at time 20
    """

    g_params = (
        {'g': g, 'g_t': g_t, 'duration': duration} if g != 0 else None
    )
    y_series, analysis = simulate_samuelson(
        y_0, y_1, α, β, γ, n, σ, g_params, 42
    )

    print(f"Solution type: {analysis['solution_type']}")
    print(f"Roots: {analysis['roots']}")
    print(f"Root type: {'Complex' if analysis['is_complex'] else 'Real'}")
    print(f"Stability: {'Stable' if analysis['is_stable'] else 'Unstable'}")

    return y_series
```

A permanent government spending shock can be simulated as follows

```
plot_y(y_stochastic_g(g=10, g_t=20, duration="permanent"))
```

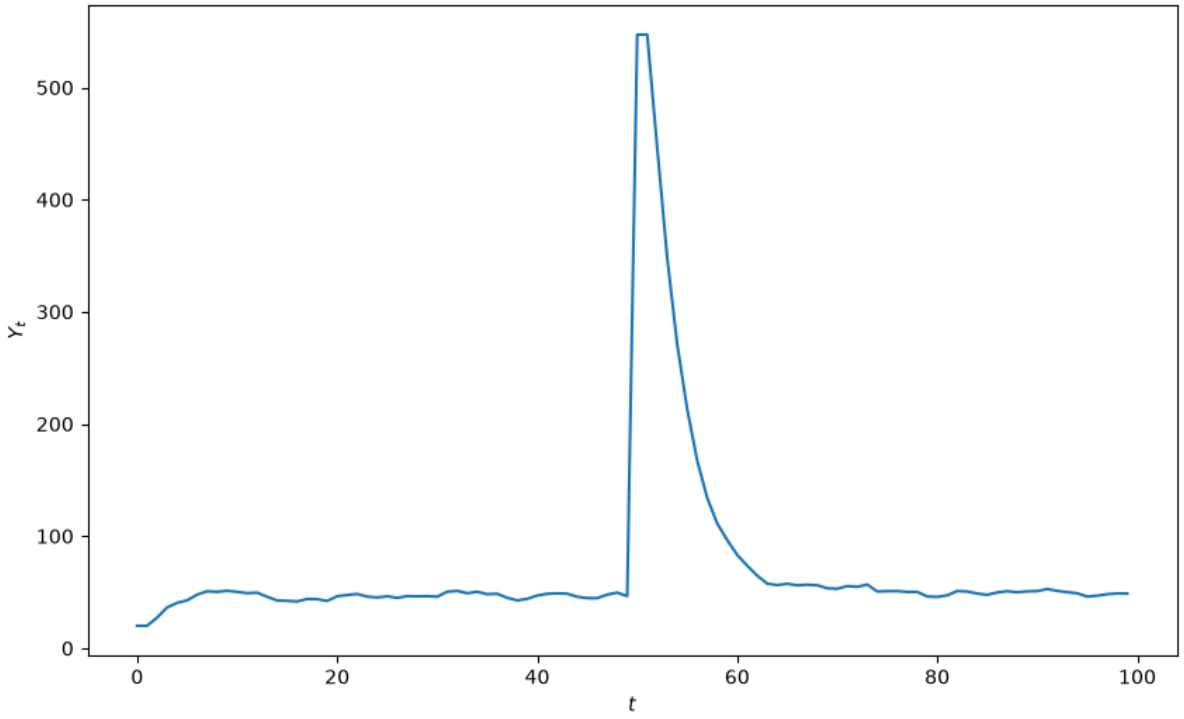
```
Solution type: Steady state convergence
Roots: [0.7236068 0.2763932]
Root type: Real
Stability: Stable
```



We can also see the response to a one time jump in government expenditures

```
plot_y(y_stochastic_g(g=500, g_t=50, duration="one-off"))
```

```
Solution type: Steady state convergence
Roots: [0.7236068 0.2763932]
Root type: Real
Stability: Stable
```



41.6 Wrapping everything into a class

Up to now, we have written functions to do the work.

Now we'll roll up our sleeves and write a Python class called `Samuelson` for the Samuelson model

```
class Samuelson:
    """
    This class represents the Samuelson model, otherwise known as the
    multiplier-accelerator model.
    The model combines the Keynesian multiplier
    with the accelerator theory of investment.

    The path of output is governed by a linear
    second-order difference equation

    .. math::

        Y_t = \alpha (1 + \beta) Y_{t-1} - \alpha \beta Y_{t-2}

    Parameters
    ----------
    y_0 : scalar
        Initial condition for Y_0
    y_1 : scalar
        Initial condition for Y_1
    α : scalar
        Marginal propensity to consume
    β : scalar
        Accelerator coefficient
```

(continues on next page)

(continued from previous page)

```
    n : int
        Number of iterations
    σ : scalar
        Volatility parameter. It must be greater than or equal to 0. Set
        equal to 0 for a non-stochastic model.
    g : scalar
        Government spending shock
    g_t : int
        Time at which government spending shock occurs. Must be specified
        when duration != None.
    duration : {None, 'permanent', 'one-off'}
        Specifies type of government spending shock. If none, government
        spending equal to g for all t.

    """

    def __init__(
        self, y_0=100, y_1=50,
        α=1.3, β=0.2, γ=10, n=100, σ=0, g=0, g_t=0, duration=None
    ):

        self.y_0, self.y_1, self.α, self.β = y_0, y_1, α, β
        self.n, self.g, self.g_t, self.duration = n, g, g_t, duration
        self.γ, self.σ = γ, σ

        # Use unified analysis function
        self.analysis = analyze_roots(α, β, verbose=False)
        self.ρ1, self.ρ2 = self.analysis['rho1'], self.analysis['rho2']
        self.roots = self.analysis['roots']

    def root_type(self):
        return "Complex conjugate" if self.analysis['is_complex'] else "Real"

    def root_less_than_one(self):
        return self.analysis['is_stable']

    def solution_type(self):
        return self.analysis['solution_type']

    def generate_series(self, seed=0):
        g_params = (
            {'g': self.g, 'g_t': self.g_t, 'duration': self.duration}
            if self.g != 0 else None
        )
        y_series, _ = simulate_samuelson(
            self.y_0, self.y_1, self.α, self.β, self.γ,
            self.n, self.σ, g_params, seed
        )
        return y_series

    def summary(self):
        print("Summary\n" + "-" * 50)
        print(f"Root type: {self.root_type()}")
        print(f"Solution type: {self.solution_type()}")
        print(f"Roots: {str(self.roots)}")

        if self.root_less_than_one() == True:
```

(continues on next page)

(continued from previous page)

```
            print("Absolute value of roots is less than one")
        else:
            print("Absolute value of roots is not less than one")

        if self.σ > 0:
            print("Stochastic series with σ = " + str(self.σ))
        else:
            print("Non-stochastic series")

        if self.g != 0:
            print("Government spending equal to " + str(self.g))

        if self.duration != None:
            print(
                self.duration.capitalize()
                + " government spending shock at t = "
                + str(self.g_t)
            )

    def plot(self, seed=0):
        fig, ax = plt.subplots(figsize=(10, 6))
        ax.plot(self.generate_series(seed))
        ax.set(xlabel="iteration", xlim=(0, self.n))
        ax.set_ylabel("$Y_t$", rotation=0)

        # Display model parameters on the plot
        paramstr = (
            f"$α={self.α:.2f}$ \n $β={self.β:.2f}$ \n "
            f"$\\gamma={self.γ:.2f}$ \n $\\sigma={self.σ:.2f}$ \n "
            f"$\\rho_1={self.ρ1:.2f}$ \n $\\rho_2={self.ρ2:.2f}$"
        )
        props = dict(fc="white", pad=10, alpha=0.5)
        ax.text(
            0.87,
            0.05,
            paramstr,
            transform=ax.transAxes,
            fontsize=12,
            bbox=props,
            va="bottom",
        )

        return fig

    def param_plot(self):


        fig = param_plot()
        ax = fig.gca()

        # Display eigenvalues in the legend
        for i, root in enumerate(self.roots):
            if isinstance(root, complex):

                # Handle sign formatting for complex number display
                operator = ["+", ""]
                root_real = self.roots[i].real
```

(continues on next page)

(continued from previous page)

```
                root_imag = self.roots[i].imag
                label = (
                    rf"$\lambda_{i+1} = {root_real:.2f}"
                    rf"{operator[i]} {root_imag:.2f}i$"
                )
            else:
                label = rf"$\lambda_{i+1} = {self.roots[i].real:.2f}$"

            # Add invisible point for legend entry
            ax.scatter(
                0, 0, s=0, label=label
            )

        # Mark current parameter values on the stability diagram
        ax.scatter(
            self.ρ1,
            self.ρ2,
            s=100,
            c="red",
            marker="+",
            label=r"$(\rho_1, \rho_2)$",
            zorder=5,
        )

        plt.legend(fontsize=12, loc=3)

        return fig
```

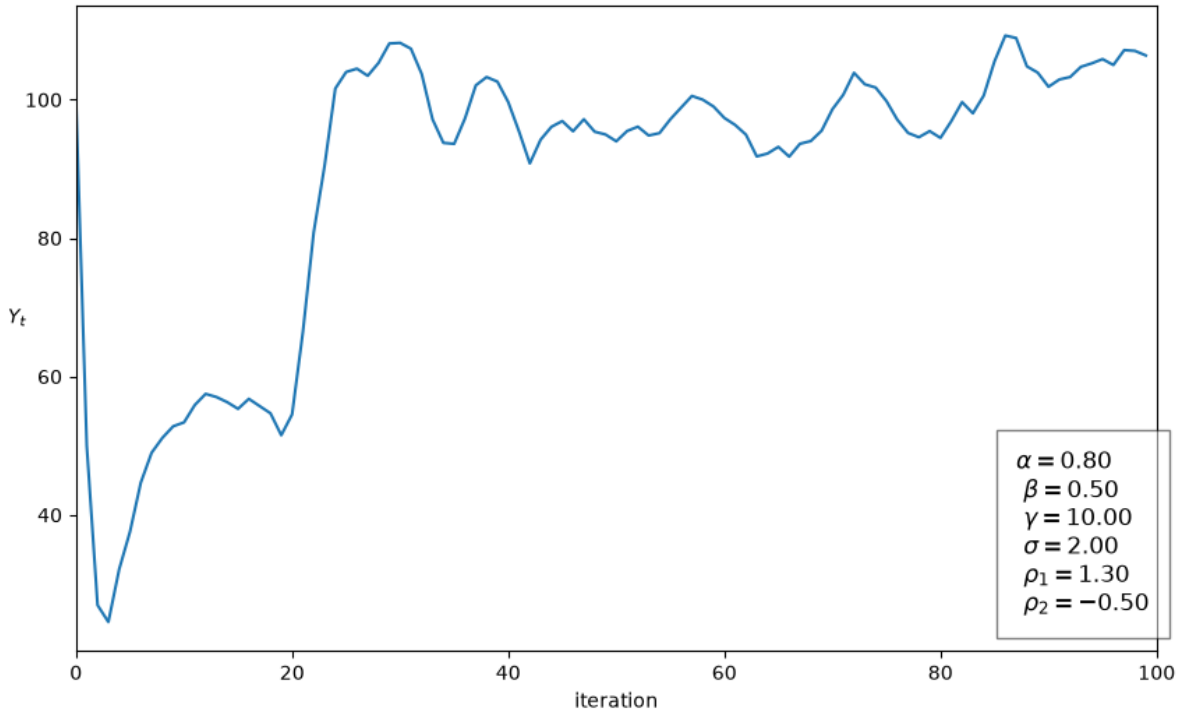
41.6.1 Illustration of Samuelson class

Now we'll put our Samuelson class to work on an example

```
sam = Samuelson(α=0.8, β=0.5, σ=2, g=10, g_t=20, duration="permanent")
sam.summary()
```

```
Summary
-----------------------------------------------------
Root type: Complex conjugate
Solution type: Damped oscillations
Roots: [0.65+0.27838822j 0.65-0.27838822j]
Absolute value of roots is less than one
Stochastic series with σ = 2
Government spending equal to 10
Permanent government spending shock at t = 20
```

```
sam.plot()
plt.show()
```

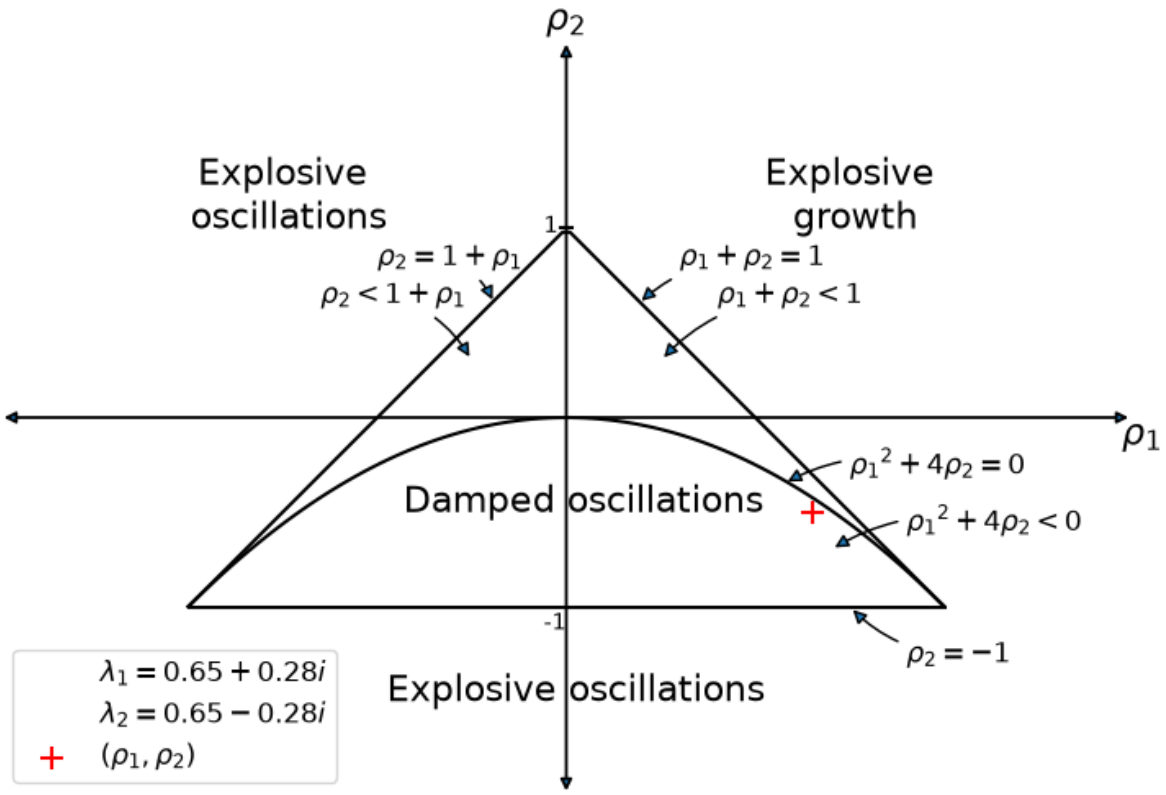


41.6.2 Using the graph

We'll use our graph to show where the roots lie and how their location is consistent with the behavior of the path just graphed.

The red + sign shows the location of the roots

```
sam.param_plot()
plt.show()
```



41.7 Using the LinearStateSpace class

It turns out that we can use the QuantEcon.py LinearStateSpace class to do much of the work that we have done from scratch above.

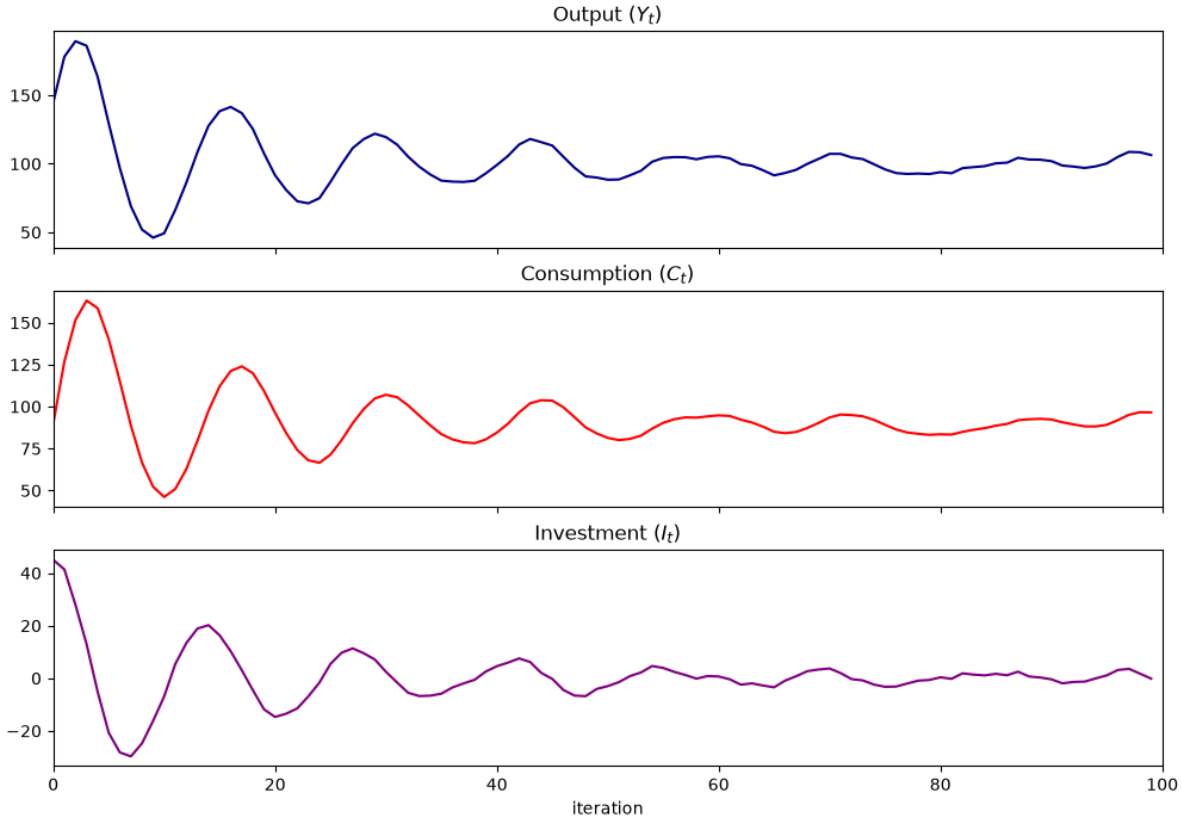
Here is how we map the Samuelson model into an instance of a `LinearStateSpace` class

```
α = 0.8
β = 0.9
ρ1 = α + β
ρ2 = -β
γ = 10
σ = 1
g = 10
n = 100

A = [[1, 0, 0], [γ + g, ρ1, ρ2], [0, 1, 0]]

G = [
    [γ + g, ρ1, ρ2],   # Y_{t+1}
    [γ, α, 0],         # C_{t+1}
    [0, β, -β],        # I_{t+1}
]

μ_0 = [1, 100, 50]
C = np.zeros((3, 1))
C[1] = σ  # Shock variance
```

(continues on next page)

(continued from previous page)

```
sam_t = LinearStateSpace(A, C, G, mu_0=μ_0)

x, y = sam_t.simulate(ts_length=n)

fig, axes = plt.subplots(3, 1, sharex=True, figsize=(12, 8))
titles = ["Output ($Y_t$)", "Consumption ($C_t$)", "Investment ($I_t$)"]
colors = ["darkblue", "red", "purple"]
for ax, series, title, color in zip(axes, y, titles, colors):
    ax.plot(series, color=color)
    ax.set(title=title, xlim=(0, n))

axes[-1].set_xlabel("iteration")

plt.show()
```



41.7.1 Other methods in the `LinearStateSpace` class

Let's plot *impulse response functions* for the instance of the Samuelson model using a method in the `LinearStateSpace` class

```
imres = sam_t.impulse_response()
imres = np.asarray(imres)
y1 = imres[:, :, 0]
y2 = imres[:, :, 1]
y1.shape
```

$$(2,\ 6,\ 1)$$

Now let's compute the zeros of the characteristic polynomial by simply calculating the eigenvalues of A

```
A = np.asarray(A)
w, v = np.linalg.eig(A)
print(np.round(w, 2))
```

```
[0.85+0.42j 0.85-0.42j 1.  +0.j  ]
```

41.7.2 Inheriting methods from `LinearStateSpace`

We could also create a subclass of `LinearStateSpace` (inheriting all its methods and attributes) to add more functions to use

```
class SamuelsonLSS(LinearStateSpace):
    """
    This subclass creates a Samuelson multiplier-accelerator model
    as a linear state space system.
    """

    def __init__(self, y_0=100, y_1=50, α=0.8, β=0.9, γ=10, σ=1, g=10):

        self.α, self.β = α, β
        self.y_0, self.y_1, self.g = y_0, y_1, g
        self.γ, self.σ = γ, σ

        # Set initial state vector
        self.initial_μ = [1, y_0, y_1]

        self.ρ1 = α + β
        self.ρ2 = -β

        # Construct state transition matrix
        self.A = [[1, 0, 0], [γ + g, self.ρ1, self.ρ2], [0, 1, 0]]

        # Construct observation matrix
        self.G = [
            [γ + g, self.ρ1, self.ρ2],  # Y_{t+1}
            [γ, α, 0],                  # C_{t+1}
            [0, β, -β],                 # I_{t+1}
        ]

        self.C = np.zeros((3, 1))
        self.C[1] = σ  # Shock variance

        # Initialize the LinearStateSpace instance
        LinearStateSpace.__init__(
            self, self.A, self.C, self.G, mu_0=self.initial_μ
        )

    # Create unicode aliases for mu_0 and Sigma_0 in the parent class
    @property
    def μ_0(self):
        return self.mu_0
```

(continues on next page)

(continued from previous page)

```
    @μ_0.setter
    def μ_0(self, value):
        self.mu_0 = value

    @property
    def Σ_0(self):
        return self.Sigma_0

    @Σ_0.setter
    def Σ_0(self, value):
        self.Sigma_0 = value

    def plot_simulation(self, ts_length=100, stationary=True, seed=0):

        # Store original distribution parameters
        temp_μ = self.μ_0
        temp_Σ = self.Σ_0

        # Use stationary distribution for simulation
        if stationary == True:
            try:
                (self.μ_x, self.μ_y, self.Σ_x, self.Σ_y, self.Σ_yx
                ) = self.stationary_distributions()
                self.μ_0 = self.μ_x
                self.Σ_0 = self.Σ_x

            # Handle case where stationary distribution doesn't exist
            except ValueError:
                print("Stationary distribution does not exist")

        np.random.seed(seed)
        x, y = self.simulate(ts_length)

        fig, axes = plt.subplots(3, 1, sharex=True, figsize=(12, 8))
        titles = ["Output ($Y_t$)",
                  "Consumption ($C_t$)",
                  "Investment ($I_t$)"]
        colors = ["darkblue", "red", "purple"]
        for ax, series, title, color in zip(axes, y, titles, colors):
            ax.plot(series, color=color)
            ax.set(title=title, xlim=(0, n))

        axes[-1].set_xlabel("iteration")
        plt.show()

        # Restore original distribution parameters
        self.μ_0 = temp_μ
        self.Σ_0 = temp_Σ

    def plot_irf(self, j=5):

        x, y = self.impulse_response(j)

        # Reshape impulse responses for plotting
        yimf = np.array(y).flatten().reshape(j + 1, 3).T

        fig, axes = plt.subplots(3, 1, sharex=True, figsize=(12, 8))
```

(continues on next page)

(continued from previous page)

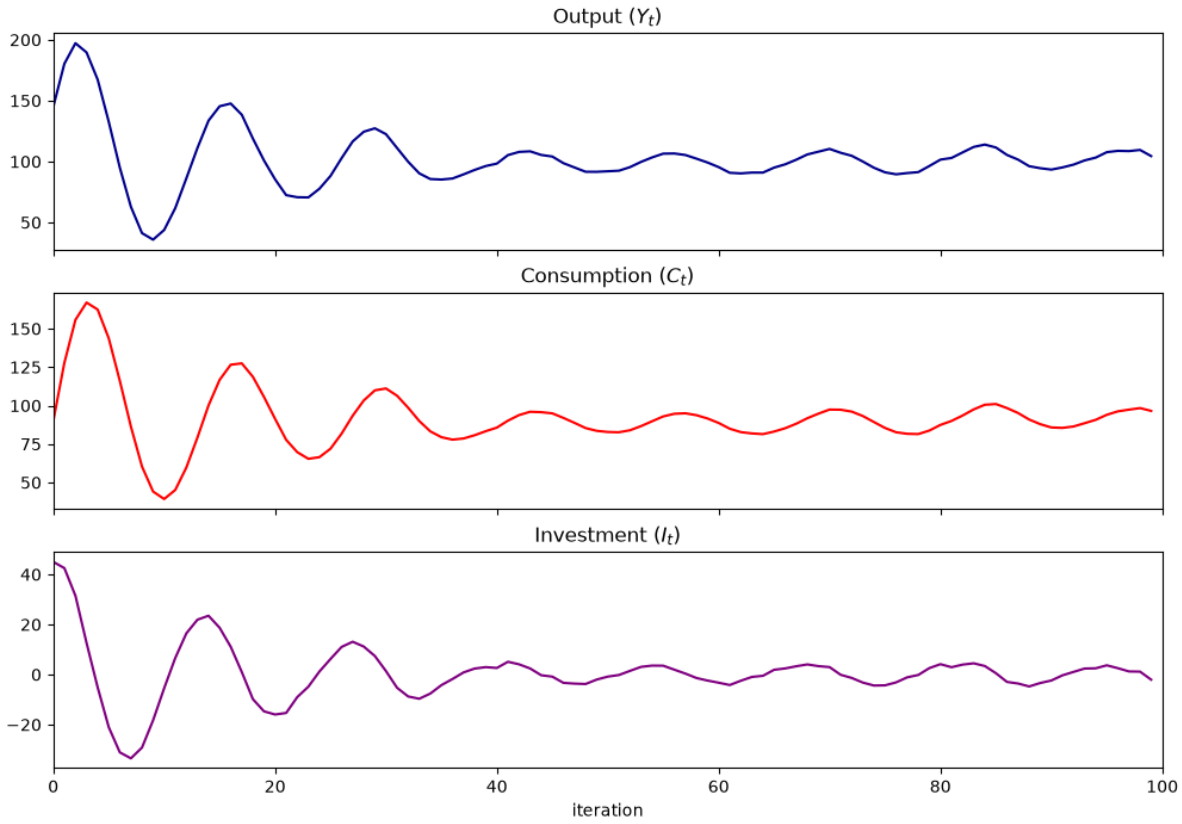
```
        labels = ["$Y_t$", "$C_t$", "$I_t$"]
        colors = ["darkblue", "red", "purple"]
        for ax, series, label, color in zip(axes, yimf, labels, colors):
            ax.plot(series, color=color)
            ax.set(xlim=(0, j))
            ax.set_ylabel(label, rotation=0, fontsize=14, labelpad=10)

        axes[0].set_title("Impulse response functions")
        axes[-1].set_xlabel("iteration")
        plt.show()

    def multipliers(self, j=5):
        x, y = self.impulse_response(j)
        return np.sum(np.array(y).flatten().reshape(j + 1, 3), axis=0)
```

41.7.3 Illustrations

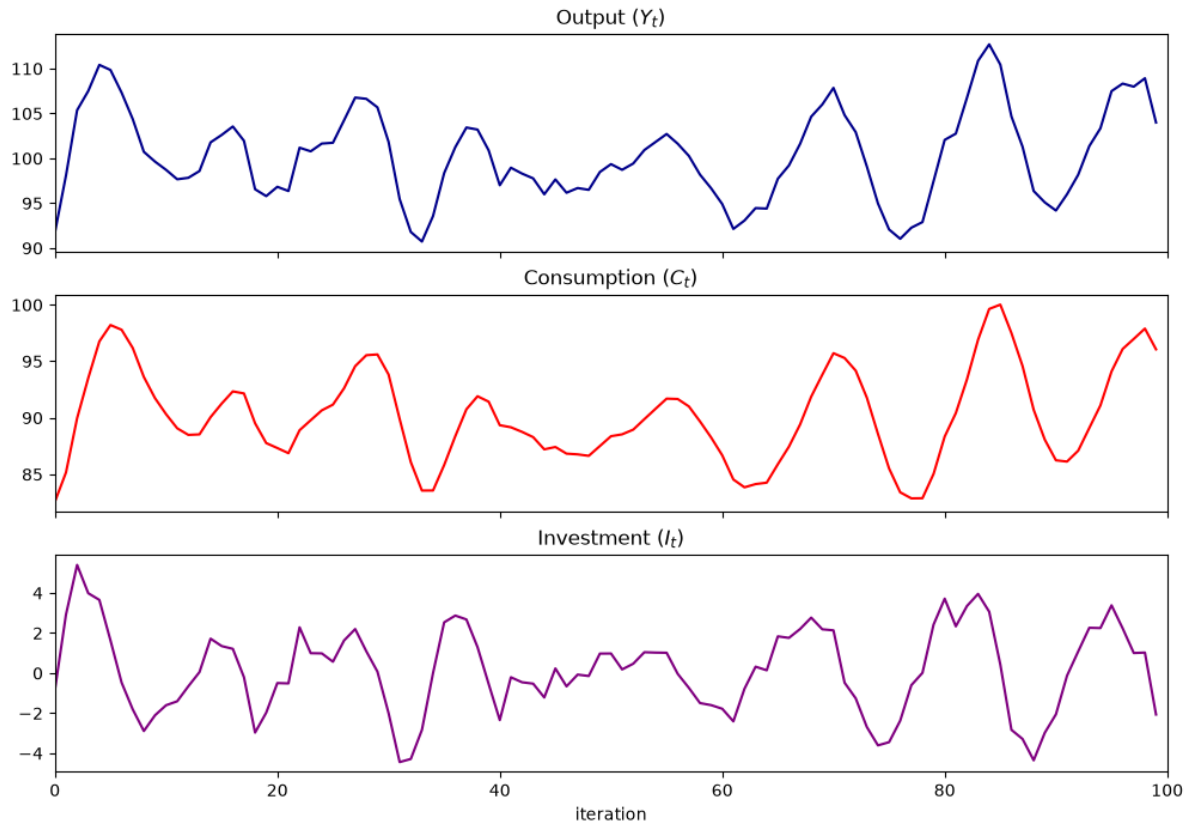
Let's show how we can use the `SamuelsonLSS`

```
samlss = SamuelsonLSS()
```

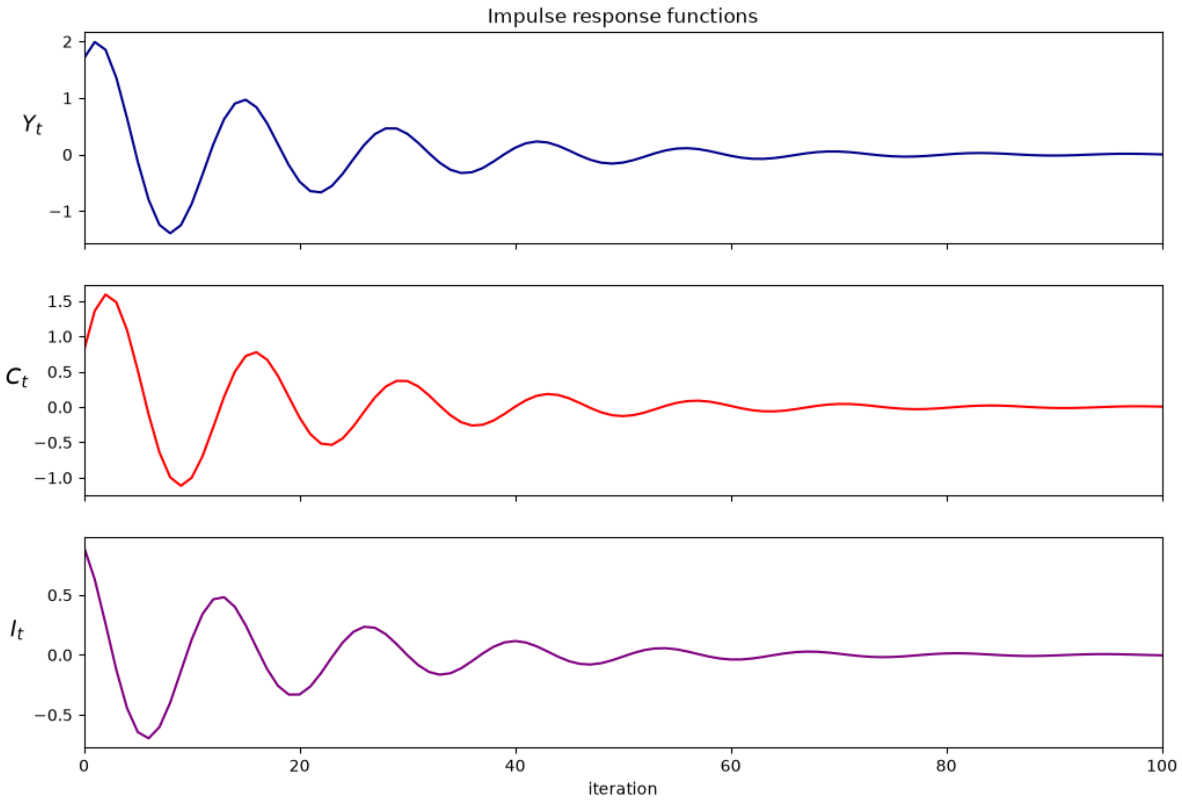
```
samlss.plot_simulation(100, stationary=False)
plt.show()
```



```
samlss.plot_simulation(100, stationary=True)
plt.show()
```




```
samlss.plot_irf(100)
plt.show()
```




```
samlss.multipliers()
```

```
array([7.414389, 6.835896, 0.578493])
```

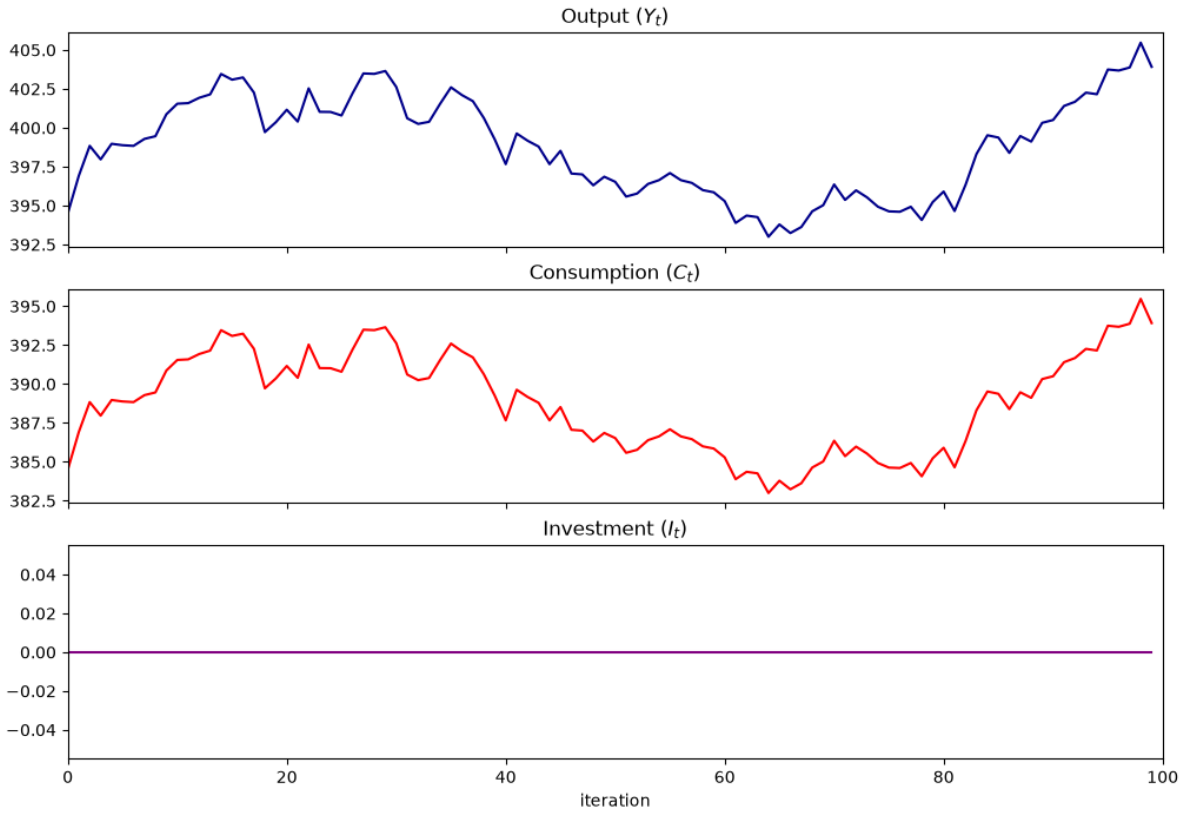
41.8 Pure multiplier model

Let's shut down the accelerator by setting $b = 0$ to get a pure multiplier model

- the absence of cycles gives an idea about why Samuelson included the accelerator

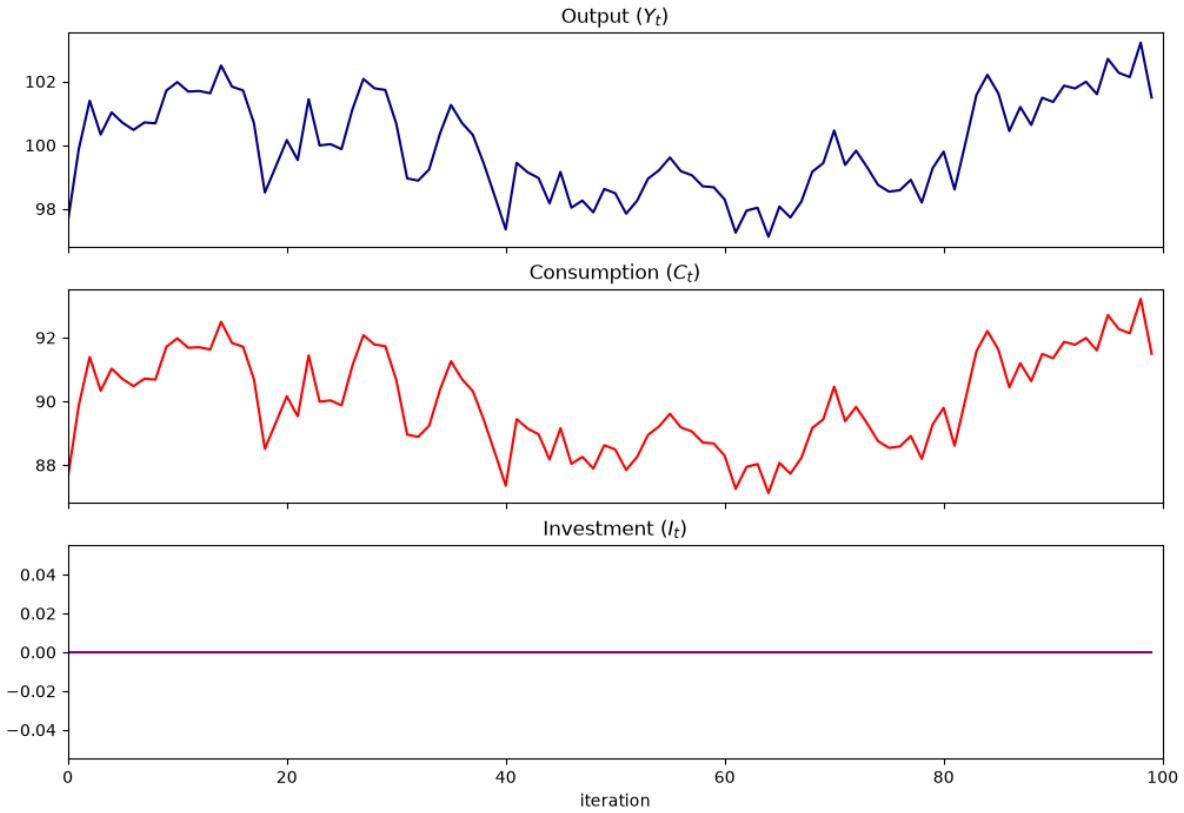
```
pure_multiplier = SamuelsonLSS(α=0.95, β=0)
```

```
pure_multiplier.plot_simulation()
```

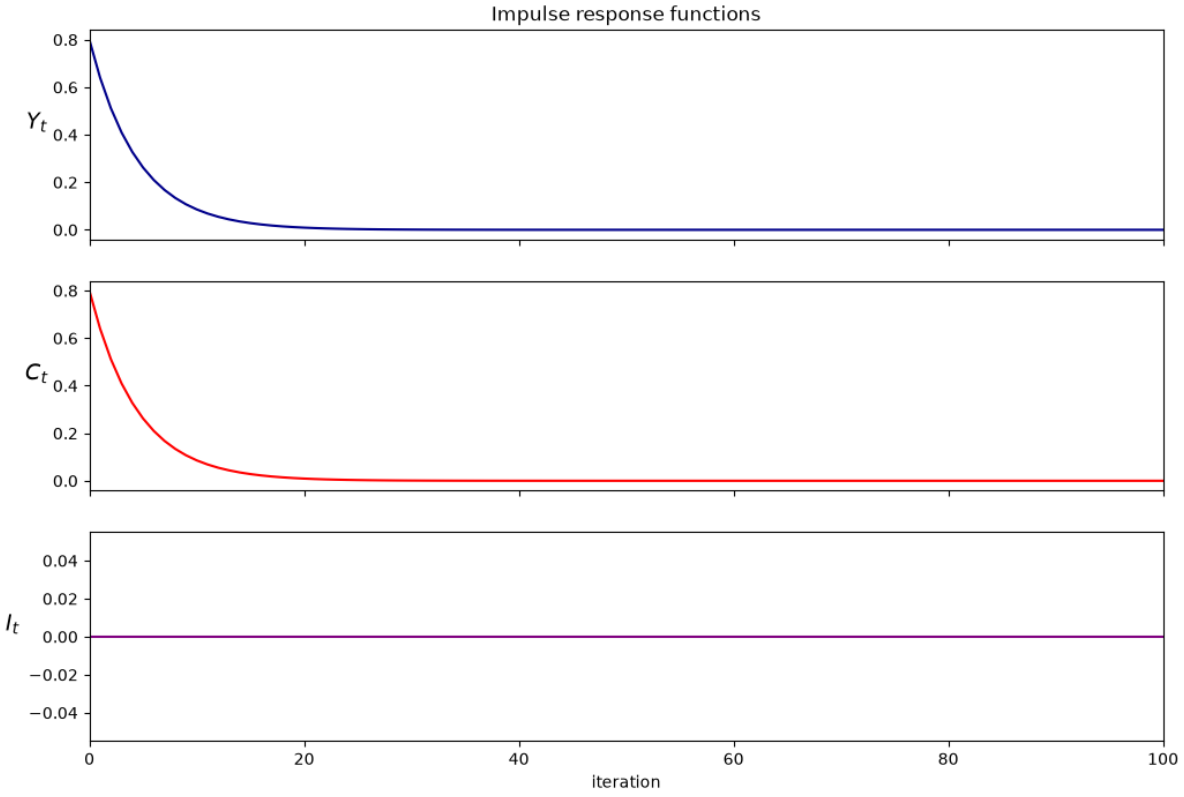



```
pure_multiplier = SamuelsonLSS(α=0.8, β=0)
```

```
pure_multiplier.plot_simulation()
```




```
pure_multiplier.plot_irf(100)
```



41.9 Summary

In this lecture, we wrote functions and classes to represent non-stochastic and stochastic versions of the Samuelson (1939) multiplier-accelerator model, described in [Samuelson, 1939].

We saw that different parameter values led to different output paths, which could either be stationary, explosive, or oscillating.

We also were able to represent the model using the QuantEcon.py LinearStateSpace class.

CHAPTER

FORTYTWO

THE ACCELERATION PRINCIPLE AND THE NATURE OF BUSINESS CYCLES

Contents

- *The Acceleration Principle and the Nature of Business Cycles*
  - *Overview*
  - *Empirical foundation for the acceleration principle*
  - *Acceleration enables oscillations*
  - *A linear system with shocks*
  - *From autocovariances to spectra*
  - *Spectral peaks in the Hansen-Samuelson model*
  - *Real roots can produce peaks in general models*
  - *A calibrated model in the frequency domain*
  - *Summary*
  - *Exercises*

42.1 Overview

This lecture studies two classic papers by Gregory Chow:

- Chow [1968] presents empirical evidence for the acceleration principle, describes how acceleration promotes oscillations, and analyzes conditions for the emergence of spectral peaks in linear difference equation subjected to random shocks
- Chow and Levitan [1969] presents a spectral analysis of a calibrated US macroeconometric model and teaches about spectral gains, coherences, and lead–lag patterns

These papers are related to ideas in the following lectures:

- The multiplier–accelerator mechanism in *Samuelson Multiplier-Accelerator*
- Linear stochastic difference equations and autocovariances in *Linear State Space Models*
- Eigenmodes of multivariate dynamics in *VARs and DMDs*

- Fourier ideas in *Circulant Matrices* (and, for empirical estimation, the advanced lecture Estimation of Spectra)

Chow [1968] builds on earlier empirical work testing the acceleration principle on US investment data.

We start with that empirical evidence before developing the theoretical framework.

We will keep returning to three ideas:

- In deterministic models, oscillations indicate complex eigenvalues of a transition matrix.
- In stochastic models, a "cycle" shows up as a local peak in a (univariate) spectral density.
- Spectral peaks depend on eigenvalues, but also on how shocks enter and on how observables load on eigenmodes.

Let's start with some standard imports:

```
import numpy as np
import matplotlib.pyplot as plt
```

We will use the following helper functions throughout the lecture

```
def spectral_density_var1(A, V, ω_grid):
    """Spectral density matrix for VAR(1): y_t = A y_{t-1} + u_t."""
    A, V = np.asarray(A), np.asarray(V)
    n = A.shape[0]
    I = np.eye(n)
    F = np.empty((len(ω_grid), n, n), dtype=complex)
    for k, ω in enumerate(ω_grid):
        H = np.linalg.inv(I - np.exp(-1j * ω) * A)
        F[k] = (H @ V @ H.conj().T) / (2 * np.pi)
    return F

def spectrum_of_linear_combination(F, b):
    """Spectrum of x_t = b'y_t given the spectral matrix F(ω)."""
    b = np.asarray(b).reshape(-1, 1)
    return np.array([np.real((b.T @ F[k] @ b).item())
                                for k in range(F.shape[0])])

def simulate_var1(A, V, T, burn=200, seed=1234):
    r"""Simulate y_t = A y_{t-1} + u_t with u_t \sim N(0, V)."""
    rng = np.random.default_rng(seed)
    A, V = np.asarray(A), np.asarray(V)
    n = A.shape[0]
    chol = np.linalg.cholesky(V)
    y = np.zeros((T + burn, n))

    for t in range(1, T + burn):
        y[t] = A @ y[t - 1] + chol @ rng.standard_normal(n)

    return y[burn:]

def sample_autocorrelation(x, max_lag):
    """Sample autocorrelation of a 1d array from lag 0 to max_lag."""
    x = np.asarray(x)
    x = x - x.mean()
    denom = np.dot(x, x)
    acf = np.empty(max_lag + 1)
    for k in range(max_lag + 1):
        acf[k] = np.dot(x[:-k] if k else x, x[k:]) / denom
    return acf
```

42.2 Empirical foundation for the acceleration principle

Chow [1968] opens by reviewing empirical evidence for the acceleration principle from earlier macroeconometric work.

Using annual observations for 1931–40 and 1948–63, Chow tested the acceleration equation on three investment categories:

- new construction
- gross private domestic investment in producers' durable equipment combined with change in business inventories
- the last two variables separately

In each case, when the regression included both Y_t and Y_{t-1} (where Y is gross national product minus taxes net of transfers), the coefficient on Y_{t-1} was of *opposite sign* and slightly smaller in absolute value than the coefficient on Y_t.

Equivalently, when expressed in terms of ΔY_t and Y_{t-1}, the coefficient on Y_{t-1} was a small fraction of the coefficient on ΔY_t.

42.2.1 An example: automobile demand

Chow presents a clean illustration using data on net investment in automobiles from his earlier work on automobile demand.

Using annual data for 1922–41 and 1948–57, he estimates by least squares:

$$y_t^n = \underset{(0.0022)}{0.0155} Y_t \; \underset{(0.0020)}{-0.0144}\, Y_{t-1} \; \underset{(0.0056)}{-0.0239}\, p_t \; \underset{(0.0040)}{+0.0199}\, p_{t-1} + \underset{(0.101)}{0.351} y_{t-1}^n + \text{const.} \tag{42.1}$$

where:

- Y_t is real disposable personal income per capita
- p_t is a relative price index for automobiles
- y_t^n is per capita net investment in passenger automobiles
- standard errors appear in parentheses

The key observation: the coefficients on Y_{t-1} and p_{t-1} are *the negatives* of the coefficients on Y_t and p_t.

This pattern is exactly what the acceleration principle predicts.

42.2.2 From stock adjustment to acceleration

The empirical support for acceleration should not be surprising once we accept a stock-adjustment demand equation for capital:

$$s_{it} = a_i Y_t + b_i s_{i,t-1} \tag{42.2}$$

where s_{it} is the stock of capital good i.

The acceleration equation (42.1) is essentially the *first difference* of (42.2).

Net investment is the change in stock, $y_{it}^n = \Delta s_{it}$, and differencing (42.2) gives:

$$y_{it}^n = a_i \Delta Y_t + b_i y_{i,t-1}^n \tag{42.3}$$

The coefficients on Y_t and Y_{t-1} in the level form are a_i and $-a_i(1-b_i)$ respectively.

They are opposite in sign and similar in magnitude when b_i is not too far from unity.

This connection between stock adjustment and acceleration is central to Chow's argument about why acceleration matters for business cycles.

42.3 Acceleration enables oscillations

Having established the empirical evidence for acceleration, we now examine why it matters theoretically for generating oscillations.

Chow [1968] asks a fundamental question: if we build a macro model using only standard demand equations with simple distributed lags, can the system generate sustained oscillations?

He shows that, under natural sign restrictions, the answer is no.

Stock-adjustment demand for durable goods leads to investment equations where the coefficient on Y_{t-1} is negative.

This negative coefficient captures the **acceleration effect**: investment responds not just to the level of income, but to its rate of change.

This negative coefficient is also what makes complex roots possible in the characteristic equation.

Without it, Chow proves that demand systems with only positive coefficients have real positive roots, and hence no oscillatory dynamics.

The *Samuelson Multiplier-Accelerator* lecture explores this mechanism in detail through the Hansen-Samuelson multiplier-accelerator model.

Here we briefly illustrate the effect.

Take the multiplier–accelerator law of motion:

$$Y_t = cY_{t-1} + v(Y_{t-1} - Y_{t-2}),$$

and rewrite it as a first-order system in (Y_t, Y_{t-1}).

```
def samuelson_transition(c, v):
    return np.array([[c + v, -v], [1.0, 0.0]])

# Compare weak vs strong acceleration
# Weak: c=0.8, v=0.1 gives real roots (discriminant > 0)
# Strong: c=0.6, v=0.8 gives complex roots (discriminant < 0)
cases = [("weak acceleration", 0.8, 0.1),
         ("strong acceleration", 0.6, 0.8)]
A_list = [samuelson_transition(c, v) for _, c, v in cases]

for (label, c, v), A in zip(cases, A_list):
    eig = np.linalg.eigvals(A)
    disc = (c + v)**2 - 4*v
    print(
        f"{label}: c={c}, v={v}, discriminant={disc:.2f}, eigenvalues={eig}")
```

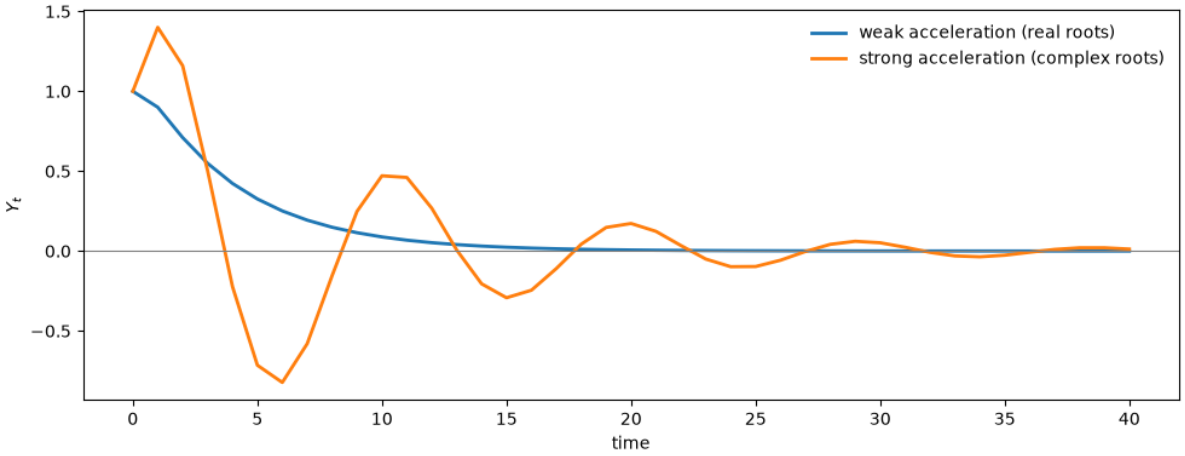
```
weak acceleration: c=0.8, v=0.1, discriminant=0.41, eigenvalues=[0.77015621 0.
↪12984379]
strong acceleration: c=0.6, v=0.8, discriminant=-1.24, eigenvalues=[0.7+0.
↪55677644j 0.7-0.55677644j]
```

With weak acceleration ($v = 0.1$), the discriminant is positive and the roots are real.

With strong acceleration ($v = 0.8$), the discriminant is negative and the roots are complex conjugates that enable oscillatory dynamics.

Now let's see how these different eigenvalue structures affect the impulse responses to a one-time shock in Y

```
T = 40
s0 = np.array([1.0, 0.0])
irfs = []
for A in A_list:
    s = s0.copy()
    path = np.empty(T + 1)
    for t in range(T + 1):
        path[t] = s[0]
        s = A @ s
    irfs.append(path)

fig, ax = plt.subplots(figsize=(10, 4))
ax.plot(range(T + 1), irfs[0], lw=2,
                label="weak acceleration (real roots)")
ax.plot(range(T + 1), irfs[1], lw=2,
                label="strong acceleration (complex roots)")
ax.axhline(0.0, lw=0.8, color='gray')
ax.set_xlabel("time")
ax.set_ylabel(r"$Y_t$")
ax.legend(frameon=False)
plt.tight_layout()
plt.show()
```



With weak acceleration, the impulse response decays monotonically.

With strong acceleration, it oscillates.

We can ask how the eigenvalues change as we increase the accelerator v.

As we increase the accelerator v, the eigenvalues move further from the origin.

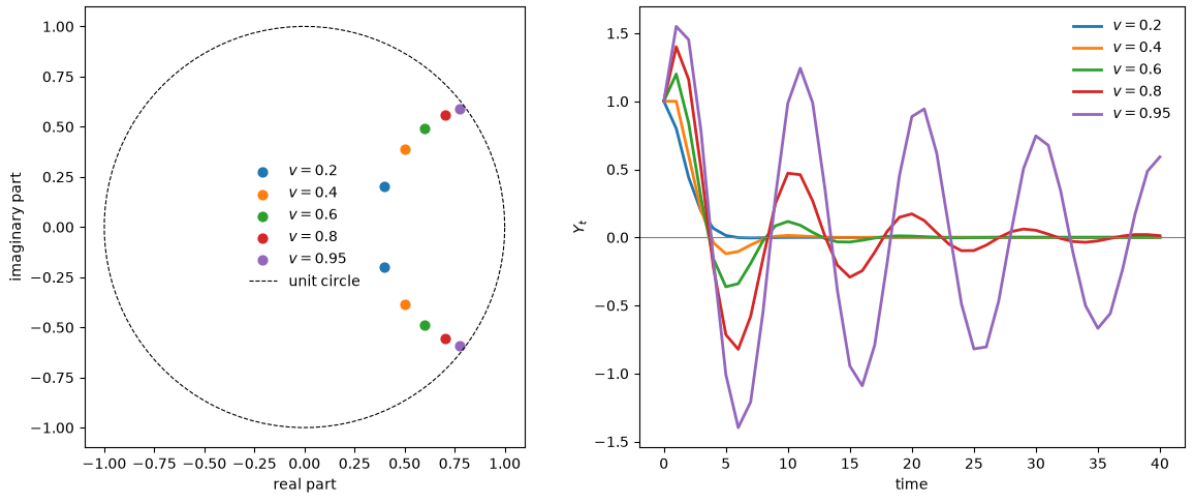
For this model, the eigenvalue modulus is $|\lambda| = \sqrt{v}$, so the stability boundary is $v = 1$.

```
v_grid = [0.2, 0.4, 0.6, 0.8, 0.95]
c = 0.6
T_irf = 40  # periods for impulse response

fig, axes = plt.subplots(1, 2, figsize=(12, 5))

for v in v_grid:
    A = samuelson_transition(c, v)
```

(continues on next page)

(continued from previous page)

```
    eig = np.linalg.eigvals(A)

    # Eigenvalues (left panel)
    axes[0].scatter(eig.real, eig.imag, s=40, label=f'$v={v}$')

    # Impulse response (right panel)
    s = np.array([1.0, 0.0])
    irf = np.empty(T_irf + 1)
    for t in range(T_irf + 1):
        irf[t] = s[0]
        s = A @ s
    axes[1].plot(range(T_irf + 1), irf, lw=2, label=f'$v={v}$')

# Visualize the eigenvalue locations and the unit circle
θ_circle = np.linspace(0, 2*np.pi, 100)
axes[0].plot(np.cos(θ_circle), np.sin(θ_circle),
                'k--', lw=0.8, label='unit circle')
axes[0].set_xlabel('real part')
axes[0].set_ylabel('imaginary part')
axes[0].set_aspect('equal')
axes[0].legend(frameon=False)

# impulse response panel
axes[1].axhline(0, lw=0.8, color='gray')
axes[1].set_xlabel('time')
axes[1].set_ylabel(r'$Y_t$')
axes[1].legend(frameon=False)

plt.tight_layout()
plt.show()
```



As v increases, eigenvalues approach the unit circle and oscillations become more persistent.

This illustrates that acceleration creates complex eigenvalues, which are necessary for oscillatory dynamics in deterministic systems.

But what happens when we add random shocks?

An insight of Ragnar Frisch [Frisch, 1933] was that damped oscillations can be "maintained" when the system is continuously perturbed by random disturbances.

To study this formally, we need to introduce the stochastic framework.

42.4 A linear system with shocks

We analyze a first-order linear stochastic system

$$y_t = Ay_{t-1} + u_t, \qquad \mathbb{E}[u_t] = 0, \qquad \mathbb{E}[u_t u_t^\top] = V, \qquad \mathbb{E}[u_t u_{t-k}^\top] = 0, \quad k \neq 0. \tag{42.4}$$

When the eigenvalues of A are strictly inside the unit circle, the process is covariance stationary and its autocovariances exist.

In the notation of *Linear State Space Models*, this is the same stability condition that guarantees a unique solution to a discrete Lyapunov equation.

Define the lag-k autocovariance matrices

$$\Gamma_k := \mathbb{E}[y_t y_{t-k}^\top]. \tag{42.5}$$

Standard calculations (also derived in [Chow, 1968]) give the recursion

$$\Gamma_k = A\Gamma_{k-1}, \quad k \geq 1, \qquad \text{and} \qquad \Gamma_0 = A\Gamma_0 A^\top + V. \tag{42.6}$$

The second equation is the discrete Lyapunov equation for Γ_0.

Chow [1968] motivates the stochastic analysis with a quote from Ragnar Frisch:

> The examples we have discussed … show that when a [deterministic] economic system gives rise to oscillations, these will most frequently be damped. But in reality the cycles … are generally not damped. How can the maintenance of the swings be explained? … One way which I believe is particularly fruitful and promising is to study what would become of the solution of a determinate dynamic system if it were exposed to a stream of erratic shocks … Thus, by connecting the two ideas: (1) the continuous solution of a determinate dynamic system and (2) the discontinuous shocks intervening and supplying the energy that may maintain the swings–we get a theoretical setup which seems to furnish a rational interpretation of those movements which we have been accustomed to see in our statistical time data.
>
> – Ragnar Frisch (1933) [Frisch, 1933]

Chow's main insight is that oscillations in the deterministic system are *neither necessary nor sufficient* for producing "cycles" in the stochastic system.

We have to bring the stochastic element into the picture.

We will show that even when eigenvalues are real (no deterministic oscillations), the stochastic system can exhibit cyclical patterns in its autocovariances and spectral densities.

42.4.1 Autocovariances in terms of eigenvalues

Let $\lambda_1, \ldots, \lambda_p$ be the distinct, possibly complex, eigenvalues of A, and let B be the matrix whose columns are the corresponding right eigenvectors:

$$AB = BD_\lambda, \quad \text{or equivalently} \quad A = BD_\lambda B^{-1} \tag{42.7}$$

where $D_\lambda = \text{diag}(\lambda_1, \ldots, \lambda_p)$.

Define canonical variables $z_t = B^{-1} y_t$.

These satisfy the decoupled dynamics:

$$z_t = D_\lambda z_{t-1} + \varepsilon_t \tag{42.8}$$

where $\varepsilon_t = B^{-1}u_t$ has covariance matrix $W = B^{-1}V(B^{-1})^\top$.

The autocovariance matrix of the canonical variables, denoted Γ_k^*, satisfies

$$\Gamma_k^* = D_\lambda^k \Gamma_0^*, \quad k = 1, 2, 3, \ldots \tag{42.9}$$

and

$$\Gamma_0^* = \left(\frac{w_{ij}}{1 - \lambda_i \lambda_j} \right) \tag{42.10}$$

where w_{ij} are elements of W.

The autocovariance matrices of the original variables are then

$$\Gamma_k = B\Gamma_k^* B^\top = B D_\lambda^k \Gamma_0^* B^\top, \quad k = 0, 1, 2, \ldots \tag{42.11}$$

The scalar autocovariance $\gamma_{ij,k} = \mathbb{E}[y_{it} y_{j,t-k}]$ is a *linear combination* of powers of the eigenvalues:

$$\gamma_{ij,k} = \sum_m \sum_n b_{im} b_{jn} \gamma^*_{mn,0} \lambda_m^k = \sum_m d_{ij,m} \lambda_m^k \tag{42.12}$$

Compare this to the deterministic time path from initial condition y_0:

$$y_{it} = \sum_j b_{ij} z_{j0} \lambda_j^t \tag{42.13}$$

Both the autocovariance function (42.12) and the deterministic path (42.13) are linear combinations of λ_m^k (or λ_j^t).

42.4.2 Complex roots and damped oscillations

When eigenvalues come in complex conjugate pairs $\lambda = re^{\pm i\theta}$ with $r < 1$, their contribution to the autocovariance function is a **damped cosine**:

$$2sr^k \cos(\theta k + \phi) \tag{42.14}$$

for appropriate amplitude s and phase ϕ determined by the eigenvector loadings.

In the deterministic model, such complex roots generate damped oscillatory time paths.

In the stochastic model, they generate damped oscillatory autocovariance functions.

It is in this sense that deterministic oscillations could be "maintained" in the stochastic model, but as we will see, the connection between eigenvalues and spectral peaks is more subtle than this suggests.

42.5 From autocovariances to spectra

Chow's key step is to translate the autocovariance sequence $\{\Gamma_k\}$ into a frequency-domain object.

The **spectral density matrix** is the Fourier transform of Γ_k:

$$F(\omega) := \frac{1}{2\pi} \sum_{k=-\infty}^{\infty} \Gamma_k e^{-i\omega k}, \qquad \omega \in [0, \pi]. \tag{42.15}$$

For the VAR(1) system (42.4), this sum has a closed form

$$F(\omega) = \frac{1}{2\pi} \left(I - Ae^{-i\omega}\right)^{-1} V \left(I - A^\top e^{i\omega}\right)^{-1}. \tag{42.16}$$

$F(\omega)$ tells us how much variation in y_t is associated with cycles of (angular) frequency ω.

Higher frequencies correspond to rapid oscillations, meaning short cycles where the series completes many up-and-down movements per unit of time.

Lower frequencies correspond to slower oscillations, meaning long cycles that unfold over extended periods.

The corresponding cycle length (or period) is

$$T(\omega) = \frac{2\pi}{\omega}. \tag{42.17}$$

Thus, a frequency of $\omega = \pi$ corresponds to the shortest possible cycle of $T = 2$ periods, while frequencies near zero correspond to very long cycles.

When the spectral density $F(\omega)$ is concentrated at particular frequencies, it indicates that the time series exhibits pronounced cyclical behavior at those frequencies.

The advanced lecture Estimation of Spectra explains how to estimate $F(\omega)$ from data.

Here we focus on the model-implied spectrum.

We saw earlier that acceleration creates complex eigenvalues, which enable oscillatory impulse responses.

But do complex roots guarantee a spectral peak?

Are they necessary for one?

Chow provides precise answers for the Hansen-Samuelson model.

42.6 Spectral peaks in the Hansen-Samuelson model

Chow [1968] provides a detailed spectral analysis of the Hansen-Samuelson multiplier-accelerator model, deriving exact conditions for when spectral peaks occur.

The analysis reveals that in this specific model, complex roots are *necessary* for a peak, but as we will see later, this is not true in general.

42.6.1 The model as a first-order system

The second-order Hansen-Samuelson equation can be written as a first-order system:

$$\begin{bmatrix} y_{1t} \\ y_{2t} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ 1 & 0 \end{bmatrix} \begin{bmatrix} y_{1,t-1} \\ y_{2,t-1} \end{bmatrix} + \begin{bmatrix} u_{1t} \\ 0 \end{bmatrix} \tag{42.18}$$

where $y_{2t} = y_{1,t-1}$ is simply the lagged value of y_{1t}.

This structure implies a special relationship among the autocovariances:

$$\gamma_{11,k} = \gamma_{22,k} = \gamma_{12,k-1} = \gamma_{21,k+1} \tag{42.19}$$

Using the autocovariance recursion, Chow shows that this leads to the condition

$$\gamma_{11,-1} = d_{11,1}\lambda_1^{-1} + d_{11,2}\lambda_2^{-1} = \gamma_{11,1} = d_{11,1}\lambda_1 + d_{11,2}\lambda_2 \tag{42.20}$$

which constrains the spectral density in a useful way.

42.6.2 The spectral density formula

From equations (42.12) and the scalar kernel $g_i(\omega) = (1-\lambda_i^2)/(1+\lambda_i^2-2\lambda_i\cos\omega)$, the spectral density of y_{1t} is:

$$f_{11}(\omega) = d_{11,1}g_1(\omega) + d_{11,2}g_2(\omega) \tag{42.21}$$

which can be written in the combined form:

$$f_{11}(\omega) = \frac{d_{11,1}(1-\lambda_1^2)(1+\lambda_2^2) + d_{11,2}(1-\lambda_2^2)(1+\lambda_1^2) - 2[d_{11,1}(1-\lambda_1^2)\lambda_2 + d_{11,2}(1-\lambda_2^2)\lambda_1]\cos\omega}{(1+\lambda_1^2-2\lambda_1\cos\omega)(1+\lambda_2^2-2\lambda_2\cos\omega)} \tag{42.22}$$

A key observation: due to condition (42.20), the *numerator is not a function of* $\cos\omega$.

Therefore, to find a maximum of $f_{11}(\omega)$, we need only find a minimum of the denominator.

42.6.3 Conditions for a spectral peak

The first derivative of the denominator with respect to ω is:

$$2[(1+\lambda_1^2)\lambda_2 + (1+\lambda_2^2)\lambda_1]\sin\omega - 8\lambda_1\lambda_2\cos\omega\sin\omega \tag{42.23}$$

For $0 < \omega < \pi$, we have $\sin\omega > 0$, so the derivative equals zero if and only if:

$$(1+\lambda_1^2)\lambda_2 + (1+\lambda_2^2)\lambda_1 = 4\lambda_1\lambda_2\cos\omega \tag{42.24}$$

For *complex conjugate roots* $\lambda_1 = re^{i\theta}$, $\lambda_2 = re^{-i\theta}$, substitution into (42.24) gives:

$$\cos\omega = \frac{1+r^2}{2r}\cos\theta \tag{42.25}$$

The second derivative confirms this is a maximum when $\omega < \frac{3\pi}{4}$.

The necessary condition for a valid solution is:

$$-1 < \frac{1+r^2}{2r}\cos\theta < 1 \tag{42.26}$$

We can interpret it as:

- When $r \approx 1$, the factor $(1+r^2)/2r \approx 1$, so $\omega \approx \theta$
- When r is small (e.g., 0.3 or 0.4), condition (42.26) can only be satisfied if $\cos\theta \approx 0$, meaning $\theta \approx \pi/2$ (cycles of approximately 4 periods)

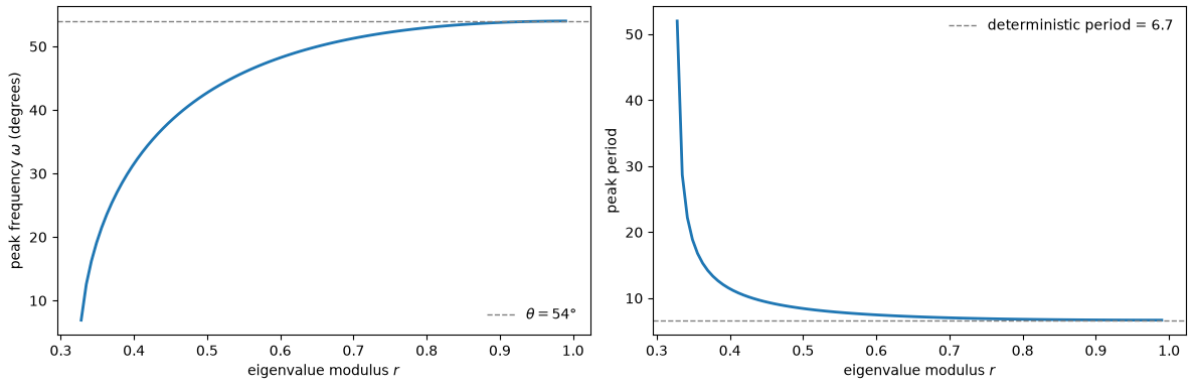
If $\theta = 54^\circ$ (corresponding to cycles of 6.67 periods) and $r = 0.4$, then $(1+r^2)/2r = 1.45$, giving $\cos\omega = 1.45 \times 0.588 = 0.85$, or $\omega = 31.5^\circ$, corresponding to cycles of 11.4 periods, which is much longer than the deterministic cycle.

```
def peak_condition_factor(r):
    """Compute (1 + r^2) / (2r)"""
    return (1 + r**2) / (2 * r)

θ_deg = 54
θ = np.deg2rad(θ_deg)
r_grid = np.linspace(0.3, 0.99, 100)

# For each r, compute the implied peak frequency
ω_peak = []
for r in r_grid:
```

(continues on next page)

(continued from previous page)

```
    factor = peak_condition_factor(r)
    cos_ω = factor * np.cos(θ)
    if -1 < cos_ω < 1:
        ω_peak.append(np.arccos(cos_ω))
    else:
        ω_peak.append(np.nan)

ω_peak = np.array(ω_peak)
period_peak = 2 * np.pi / ω_peak

fig, axes = plt.subplots(1, 2, figsize=(12, 4))

axes[0].plot(r_grid, np.rad2deg(ω_peak), lw=2)
axes[0].axhline(θ_deg, ls='--', lw=1.0, color='gray',
        label=rf'$\theta = {θ_deg}°$')
axes[0].set_xlabel('eigenvalue modulus $r$')
axes[0].set_ylabel(r'peak frequency $\omega$ (degrees)')
axes[0].legend(frameon=False)

axes[1].plot(r_grid, period_peak, lw=2)
axes[1].axhline(360/θ_deg, ls='--', lw=1.0, color='gray',
        label=rf'deterministic period = {360/θ_deg:.1f}')
axes[1].set_xlabel('eigenvalue modulus $r$')
axes[1].set_ylabel('peak period')
axes[1].legend(frameon=False)

plt.tight_layout()
plt.show()

r_example = 0.4
factor = peak_condition_factor(r_example)
cos_ω = factor * np.cos(θ)
ω_example = np.arccos(cos_ω)
print(f"Chow's example: r = {r_example}, θ = {θ_deg}°")
print(f"  cos(ω) = {cos_ω:.3f}")
print(f"  ω = {np.rad2deg(ω_example):.1f}°")
print(f"  Peak period = {360/np.rad2deg(ω_example):.1f}")
```




```
Chow's example: r = 0.4, θ = 54°
  cos(ω) = 0.852
  ω = 31.5°
  Peak period = 11.4
```

As $r \to 1$, the peak frequency converges to θ.

For smaller r, the peak frequency can differ substantially from the deterministic oscillation frequency.

42.6.4 Real positive roots cannot produce peaks

For *real and positive roots* $\lambda_1, \lambda_2 > 0$, the first-order condition (42.24) cannot be satisfied.

To see why, recall that a spectral peak at an interior frequency $\omega \in (0, \pi)$ requires

$$\cos\omega = \frac{(1+\lambda_1^2)\lambda_2 + (1+\lambda_2^2)\lambda_1}{4\lambda_1\lambda_2}.$$

For this to have a solution, we need the right-hand side to lie in $[-1, 1]$.

But for positive λ_1, λ_2, the numerator exceeds $4\lambda_1\lambda_2$:

$$(1+\lambda_1^2)\lambda_2 + (1+\lambda_2^2)\lambda_1 - 4\lambda_1\lambda_2 = \lambda_1(1-\lambda_2)^2 + \lambda_2(1-\lambda_1)^2. \tag{42.27}$$

The right-hand side is a sum of two non-negative terms (each is a positive number times a square).

It equals zero only if both $\lambda_1 = 1$ and $\lambda_2 = 1$, which violates the stability condition $|\lambda_i| < 1$.

For any stable system with real positive roots, this expression is strictly positive, so

$$\cos\omega = \frac{(1+\lambda_1^2)\lambda_2 + (1+\lambda_2^2)\lambda_1}{4\lambda_1\lambda_2} > 1, \tag{42.28}$$

which is impossible.

This is a key result: in the Hansen-Samuelson model, *complex roots are necessary* for a spectral peak at interior frequencies.

The following figure illustrates the difference in spectra between a case with complex roots and a case with real roots

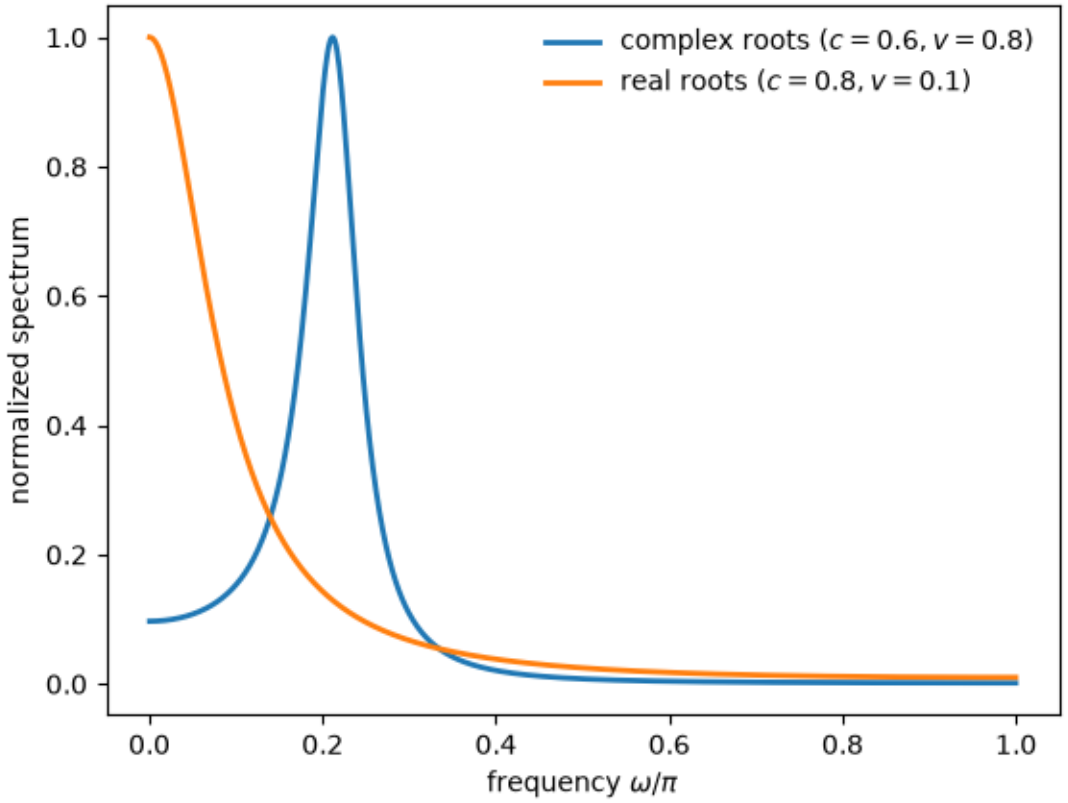
```
ω_grid = np.linspace(1e-3, np.pi - 1e-3, 800)
V_hs = np.array([[1.0, 0.0], [0.0, 0.0]])  # shock only in first equation

# Case 1: Complex roots (c=0.6, v=0.8)
c_complex, v_complex = 0.6, 0.8
A_complex = samuelson_transition(c_complex, v_complex)
eig_complex = np.linalg.eigvals(A_complex)

# Case 2: Real roots (c=0.8, v=0.1)
c_real, v_real = 0.8, 0.1
A_real = samuelson_transition(c_real, v_real)
eig_real = np.linalg.eigvals(A_real)

print(
    f"Complex case (c={c_complex}, v={v_complex}): eigenvalues = {eig_complex}")
print(
    f"Real case (c={c_real}, v={v_real}): eigenvalues = {eig_real}")

F_complex = spectral_density_var1(A_complex, V_hs, ω_grid)
F_real = spectral_density_var1(A_real, V_hs, ω_grid)

f11_complex = np.real(F_complex[:, 0, 0])
f11_real = np.real(F_real[:, 0, 0])

fig, ax = plt.subplots()
```

(continues on next page)

(continued from previous page)

```
ax.plot(ω_grid / np.pi, f11_complex / np.max(f11_complex), lw=2,
        label=fr'complex roots ($c={c_complex}, v={v_complex}$)')
ax.plot(ω_grid / np.pi, f11_real / np.max(f11_real), lw=2,
        label=fr'real roots ($c={c_real}, v={v_real}$)')
ax.set_xlabel(r'frequency $\omega/\pi$')
ax.set_ylabel('normalized spectrum')
ax.legend(frameon=False)
plt.show()
```

```
Complex case (c=0.6, v=0.8): eigenvalues = [0.7+0.55677644j 0.7-0.55677644j]
Real case (c=0.8, v=0.1): eigenvalues = [0.77015621 0.12984379]
```



With complex roots, the spectrum has a clear interior peak.

With real roots, the spectrum is monotonically decreasing and no interior peak is possible.

42.7 Real roots can produce peaks in general models

While real positive roots cannot produce spectral peaks in the Hansen-Samuelson model, Chow [1968] emphasizes that this is *not true in general*.

In multivariate systems, the spectral density of a linear combination of variables can have interior peaks even when all eigenvalues are real and positive.

42.7.1 Example

Chow constructs the following explicit example with two real positive eigenvalues:

$$\lambda_1 = 0.1, \quad \lambda_2 = 0.9 \tag{42.29}$$

$$w_{11} = w_{22} = 1, \quad w_{12} = 0.8 \tag{42.30}$$

$$b_{m1} = 1, \quad b_{m2} = -0.01 \tag{42.31}$$

The spectral density of the linear combination $x_t = b_m^\top y_t$ is:

$$f_{mm}(\omega) = \frac{0.9913}{1.01 - 0.2\cos\omega} - \frac{0.001570}{1.81 - 1.8\cos\omega} \tag{42.32}$$

Chow tabulates the values:

ω	0	$\pi/8$	$2\pi/8$	$3\pi/8$	$4\pi/8$	$5\pi/8$	$6\pi/8$	$7\pi/8$	π
$f_{mm}(\omega)$	1.067	1.183	1.191	1.138	1.061	0.981	0.912	0.860	0.829

The peak at ω slightly below $\pi/8$ (corresponding to periods around 11) is "quite pronounced."

In the following figure, we reproduce this table, but with Python, we can plot a finer grid to find the peak more accurately

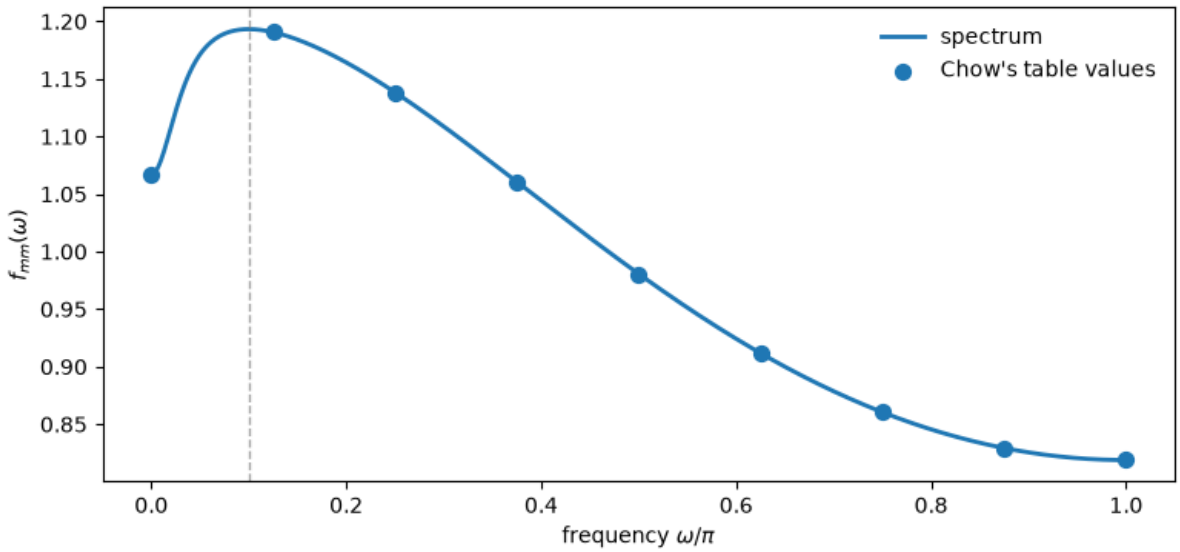
```
λ1, λ2 = 0.1, 0.9
w11, w22, w12 = 1.0, 1.0, 0.8
bm1, bm2 = 1.0, -0.01

# Construct the system
A_chow_ex = np.diag([λ1, λ2])

# W is the canonical shock covariance; we need V = B W B^T
# For diagonal A with distinct eigenvalues, B = I, so V = W
V_chow_ex = np.array([[w11, w12], [w12, w22]])
b_chow_ex = np.array([bm1, bm2])

# Chow's formula
def chow_spectrum_formula(ω):
    term1 = 0.9913 / (1.01 - 0.2 * np.cos(ω))
    term2 = 0.001570 / (1.81 - 1.8 * np.cos(ω))
    return term1 - term2

# Compute via formula and via our general method
ω_table = np.array([0, np.pi/8, 2*np.pi/8, 3*np.pi/8, 4*np.pi/8,
                    5*np.pi/8, 6*np.pi/8, 7*np.pi/8, np.pi])
f_formula = np.array([chow_spectrum_formula(ω) for ω in ω_table])

# General method
ω_grid_fine = np.linspace(1e-4, np.pi, 1000)
F_chow_ex = spectral_density_var1(A_chow_ex, V_chow_ex, ω_grid_fine)
f_general = spectrum_of_linear_combination(F_chow_ex, b_chow_ex)

# Normalize to match Chow's table scale
scale = f_formula[0] / spectrum_of_linear_combination(
    spectral_density_var1(
        A_chow_ex, V_chow_ex, np.array([0.0])), b_chow_ex)[0]
```

(continues on next page)

(continued from previous page)

```
print("Chow's Table (equation 67):")
print("ω/π:        ", "  ".join([f"{ω/np.pi:.3f}" for ω in ω_table]))
print("f_mm(ω):    ", "  ".join([f"{f:.3f}" for f in f_formula]))

fig, ax = plt.subplots(figsize=(9, 4))
ax.plot(ω_grid_fine / np.pi, f_general * scale, lw=2,
            label='spectrum')
ax.scatter(ω_table / np.pi, f_formula, s=50, zorder=3,
            label="Chow's table values")

# Mark the peak
i_peak = np.argmax(f_general)
ω_peak = ω_grid_fine[i_peak]
ax.axvline(ω_peak / np.pi, ls='--', lw=1.0, color='gray', alpha=0.7)
ax.set_xlabel(r'frequency $\omega/\pi$')
ax.set_ylabel(r'$f_{mm}(\omega)$')
ax.legend(frameon=False)
plt.show()

print(f"\nPeak at ω/π ≈ {ω_peak/np.pi:.3f}, period ≈ {2*np.pi/ω_peak:.1f}")
```

```
Chow's Table (equation 67):
ω/π:         0.000  0.125  0.250  0.375  0.500  0.625  0.750  0.875  1.000
f_mm(ω):     1.067  1.191  1.138  1.061  0.981  0.912  0.860  0.829  0.819

Peak at ω/π ≈ 0.100, period ≈ 20.0
```



The peak appears at $\omega/\pi \approx 0.10$, which corresponds to a cycle length of approximately 20 periods, again much longer than the deterministic cycles implied by the eigenvalues.

42.7.2 The Slutsky connection

Chow connects this result to Slutsky's [Slutzky, 1937] finding that moving averages of a random series have recurrent cycles.

The VAR(1) model can be written as an infinite moving average:

$$y_t = u_t + Au_{t-1} + A^2 u_{t-2} + \cdots \tag{42.33}$$

This amounts to taking an infinite moving average of the random vectors u_t with "geometrically declining" weights $A^0, A^1, A^2, ...$

For a scalar process with $0 < \lambda < 1$, no distinct cycles can emerge.

But for a matrix A with real roots between 0 and 1, cycles *can* emerge in linear combinations of the variables.

As Chow puts it: "When neither of two (canonical) variables has distinct cycles… a linear combination can have a peak in its spectral density."

42.7.3 The general lesson

The examples above illustrate the following central points:

1. In the *Hansen-Samuelson model specifically*, complex roots are necessary for a spectral peak
2. But in *general multivariate systems*, complex roots are neither necessary nor sufficient
3. The full spectral shape depends on:
    - The eigenvalues of A
    - The shock covariance structure V
    - How the observable of interest loads on the eigenmodes (the vector b)

42.8 A calibrated model in the frequency domain

Chow and Levitan [1969] use the frequency-domain objects from Chow [1968] to study a calibrated annual macroeconometric model.

They work with five annual aggregates:

- $y_1 = C$ (consumption),
- $y_2 = I_1$ (equipment plus inventories),
- $y_3 = I_2$ (construction),
- $y_4 = R_a$ (long rate),
- $y_5 = Y_1 = C + I_1 + I_2$ (private-domestic GNP),

and add $y_6 = y_{1,t-1}$ to rewrite the original system in first-order form.

Throughout this section, frequency is measured in cycles per year, $f = \omega/2\pi \in [0, 1/2]$.

Following the paper, we normalize each spectrum to have area 1 over $[0, 1/2]$ so plots compare shape rather than scale.

Our goal is to reconstruct the transition matrix A and then compute and interpret the model-implied spectra, gains/coherences, and phase differences.

42.8.1 The cycle subsystem

The paper starts from a reduced form with exogenous inputs,

$$y_t = Ay_{t-1} + Cx_t + u_t. \tag{42.34}$$

To study cycles, they remove the deterministic component attributable to x_t and focus on the zero-mean subsystem

$$y_t = Ay_{t-1} + u_t. \tag{42.35}$$

For second moments, the only additional ingredient is the covariance matrix $V = \mathbb{E}[u_t u_t^\top]$.

Chow and Levitan compute it from structural parameters via

$$V = M^{-1}\Sigma(M^{-1})^\top \tag{42.36}$$

where Σ is the covariance of structural residuals and M is the matrix of contemporaneous structural coefficients.

Here we take A and V as given and ask what they imply for spectra and cross-spectra.

The 6×6 reduced-form shock covariance matrix V (scaled by 10^{-7}) reported by Chow and Levitan is:

$$V = \begin{bmatrix} 8.250 & 7.290 & 2.137 & 2.277 & 17.68 & 0 \\ 7.290 & 7.135 & 1.992 & 2.165 & 16.42 & 0 \\ 2.137 & 1.992 & 0.618 & 0.451 & 4.746 & 0 \\ 2.277 & 2.165 & 0.451 & 1.511 & 4.895 & 0 \\ 17.68 & 16.42 & 4.746 & 4.895 & 38.84 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}. \tag{42.37}$$

The sixth row and column are zeros because y_6 is an identity (lagged y_1).

The transition matrix A has six characteristic roots:

$$\begin{aligned} &\lambda_1 = 0.9999725, \quad \lambda_2 = 0.9999064, \quad \lambda_3 = 0.4838, \\ &\lambda_4 = 0.0761 + 0.1125i, \quad \lambda_5 = 0.0761 - 0.1125i, \quad \lambda_6 = -0.00004142. \end{aligned} \tag{42.38}$$

Two roots are near unity because two structural equations are in first differences.

One root (λ_6) is theoretically zero because of the identity $y_5 = y_1 + y_2 + y_3$.

The complex conjugate pair $\lambda_{4,5}$ has modulus $|\lambda_4| = \sqrt{0.0761^2 + 0.1125^2} \approx 0.136$.

The right eigenvector matrix B (columns are eigenvectors corresponding to $\lambda_1, \ldots, \lambda_6$):

$$B = \begin{bmatrix} -0.008 & 1.143 & 0.320 & 0.283 + 0.581i & 0.283 - 0.581i & 0.000 \\ -0.000 & 0.013 & -0.586 & -2.151 + 0.742i & -2.151 - 0.742i & 2.241 \\ -0.001 & 0.078 & 0.889 & -0.215 + 0.135i & -0.215 - 0.135i & 0.270 \\ 1.024 & 0.271 & 0.069 & -0.231 + 0.163i & -0.231 - 0.163i & 0.307 \\ -0.009 & 1.235 & 0.623 & -2.082 + 1.468i & -2.082 - 1.468i & 2.766 \\ -0.008 & 1.143 & 0.662 & 4.772 + 0.714i & 4.772 - 0.714i & -4.399 \end{bmatrix}. \tag{42.39}$$

Together, V, $\{\lambda_i\}$, and B are sufficient to compute all spectral and cross-spectral densities.

42.8.2 Reconstructing A and computing $F(\omega)$

The paper reports (λ, B, V), which is enough to reconstruct $A = B\,\mathrm{diag}(\lambda_1, \ldots, \lambda_6)\,B^{-1}$ and then compute the model-implied spectral objects.

```
λ = np.array([
    0.9999725, 0.9999064, 0.4838,
    0.0761 + 0.1125j, 0.0761 - 0.1125j, -0.00004142
], dtype=complex)

B = np.array([
    [-0.008, 1.143, 0.320, 0.283+0.581j, 0.283-0.581j, 0.000],
    [-0.000, 0.013, -0.586, -2.151+0.742j, -2.151-0.742j, 2.241],
    [-0.001, 0.078, 0.889, -0.215+0.135j, -0.215-0.135j, 0.270],
    [1.024, 0.271, 0.069, -0.231+0.163j, -0.231-0.163j, 0.307],
    [-0.009, 1.235, 0.623, -2.082+1.468j, -2.082-1.468j, 2.766],
    [-0.008, 1.143, 0.662, 4.772+0.714j, 4.772-0.714j, -4.399]
], dtype=complex)

V = np.array([
    [8.250, 7.290, 2.137, 2.277, 17.68, 0],
    [7.290, 7.135, 1.992, 2.165, 16.42, 0],
    [2.137, 1.992, 0.618, 0.451, 4.746, 0],
    [2.277, 2.165, 0.451, 1.511, 4.895, 0],
    [17.68, 16.42, 4.746, 4.895, 38.84, 0],
    [0, 0, 0, 0, 0, 0]
]) * 1e-7

D_λ = np.diag(λ)
A_chow = B @ D_λ @ np.linalg.inv(B)
A_chow = np.real(A_chow)
print("eigenvalues of reconstructed A:")
print(np.linalg.eigvals(A_chow).round(6))
```

```
eigenvalues of reconstructed A:
[-4.10000e-05+0.j      7.61000e-02+0.1125j  7.61000e-02-0.1125j
  4.83800e-01+0.j      9.99906e-01+0.j      9.99973e-01+0.j    ]
```

42.8.3 Canonical coordinates

Chow and Levitan's canonical transformation uses $z_t = B^{-1} y_t$, giving dynamics $z_t = D_\lambda z_{t-1} + e_t$.

Accordingly, the canonical shock covariance is

$$W = B^{-1} V (B^{-1})^\top.$$

```
B_inv = np.linalg.inv(B)
W = B_inv @ V @ B_inv.T
print("diagonal of W:")
print(np.diag(W).round(10))
```

```
diagonal of W:
[ 8.560e-08-0.00e+00j  3.638e-07-0.00e+00j  1.300e-08-0.00e+00j
 -7.880e-08+6.26e-08j -7.880e-08-6.26e-08j -1.650e-08+0.00e+00j]
```

Chow and Levitan derive the following closed-form formula for the spectral density matrix:

$$F(\omega) = B \left[\frac{w_{ij}}{(1 - \lambda_i e^{-i\omega})(1 - \lambda_j e^{i\omega})} \right] B^\top, \tag{42.40}$$

where w_{ij} are elements of the canonical shock covariance W.

```
def spectral_density_chow(λ, B, W, ω_grid):
    """Spectral density via Chow's eigendecomposition formula."""
    p = len(λ)
    F = np.zeros((len(ω_grid), p, p), dtype=complex)
    for k, ω in enumerate(ω_grid):
        F_star = np.zeros((p, p), dtype=complex)
        for i in range(p):
            for j in range(p):
                denom = (1 - λ[i] * np.exp(-1j * ω)) \
                    * (1 - λ[j] * np.exp(1j * ω))
                F_star[i, j] = W[i, j] / denom
        F[k] = B @ F_star @ B.T
    return F / (2 * np.pi)

freq = np.linspace(1e-4, 0.5, 5000)      # cycles/year in [0, 1/2]
ω_grid = 2 * np.pi * freq                # radians in [0, π]
F_chow = spectral_density_chow(λ, B, W, ω_grid)
```

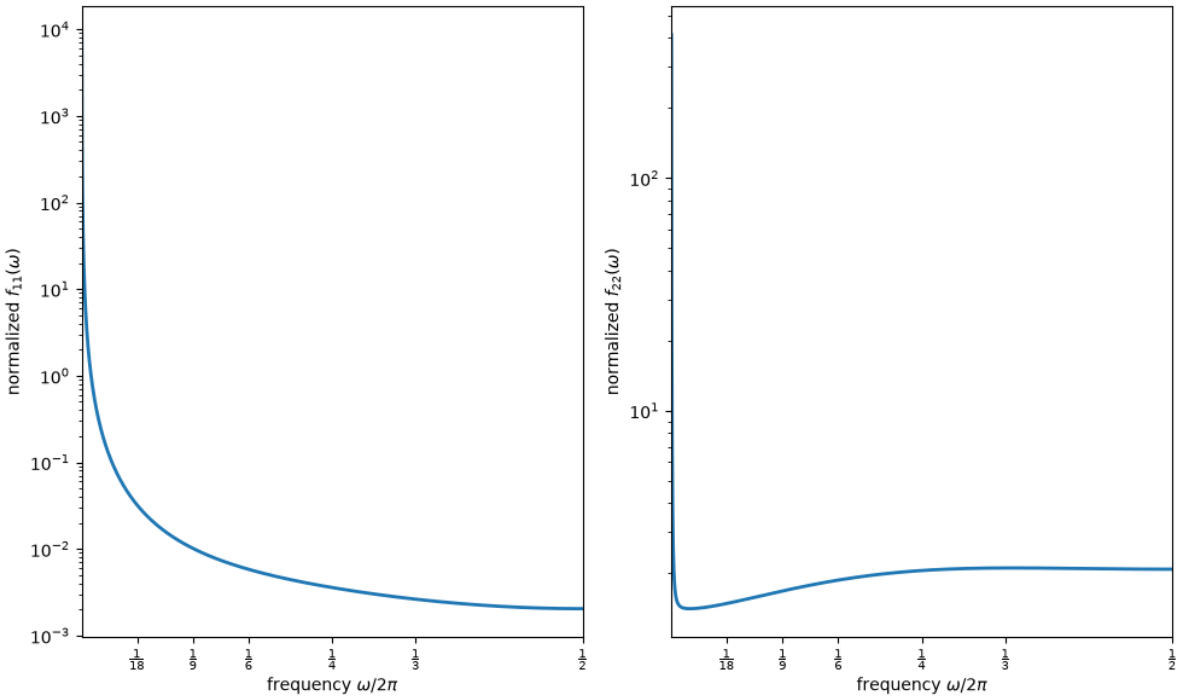
Let's plot the univariate spectra of consumption (y_1) and equipment plus inventories (y_2)

```
variable_names = ['$C$', '$I_1$', '$I_2$', '$R_a$', '$Y_1$']
freq_ticks = [1/18, 1/9, 1/6, 1/4, 1/3, 1/2]
freq_labels = [r'$\frac{1}{18}$', r'$\frac{1}{9}$', r'$\frac{1}{6}$',
               r'$\frac{1}{4}$', r'$\frac{1}{3}$', r'$\frac{1}{2}$']

def paper_frequency_axis(ax):
    ax.set_xlim([0.0, 0.5])
    ax.set_xticks(freq_ticks)
    ax.set_xticklabels(freq_labels)
    ax.set_xlabel(r'frequency $\omega/2\pi$')

# Normalized spectra (areas set to 1)
S = np.real(np.diagonal(F_chow, axis1=1, axis2=2))[:, :5]
df = np.diff(freq)
areas = np.sum(0.5 * (S[1:] + S[:-1]) * df[:, None], axis=0)
S_norm = S / areas
mask = freq >= 0.0

fig, axes = plt.subplots(1, 2, figsize=(10, 6))

# Figure I.1: consumption (log scale)
axes[0].plot(freq[mask], S_norm[mask, 0], lw=2)
axes[0].set_yscale('log')
paper_frequency_axis(axes[0])
axes[0].set_ylabel(r'normalized $f_{11}(\omega)$')

# Figure I.2: equipment + inventories (log scale)
axes[1].plot(freq[mask], S_norm[mask, 1], lw=2)
axes[1].set_yscale('log')
paper_frequency_axis(axes[1])
axes[1].set_ylabel(r'normalized $f_{22}(\omega)$')
```

(continues on next page)

(continued from previous page)

```
plt.tight_layout()
plt.show()

i_peak = np.argmax(S_norm[mask, 1])
f_peak = freq[mask][i_peak]
```



The left panel corresponds to consumption and declines monotonically with frequency.

It illustrates Granger's "typical spectral shape" for macroeconomic time series.

The right panel corresponds to equipment plus inventories and shows the clearest (but still very flat) interior-frequency bump.

Chow and Levitan associate the dominance of very low frequencies in both plots with strong persistence and long-run movements.

Very large low-frequency power can arise from eigenvalues extremely close to one, which occurs mechanically when some equations are written in first differences.

Local peaks are not automatic: complex roots may have small modulus, and multivariate interactions can generate peaks even when all roots are real.

The interior bump in the right panel corresponds to cycles of roughly three years, with the spectrum nearly flat over cycles between about two and four years.

(This discussion follows Section II of [Chow and Levitan, 1969].)

42.8.4 How variables move together across frequencies

Beyond univariate spectra, we can ask how pairs of variables covary at each frequency.

The **cross-spectrum** $f_{ij}(\omega) = c_{ij}(\omega) - i \cdot q_{ij}(\omega)$ decomposes into the cospectrum c_{ij} and the quadrature spectrum q_{ij}.

The **cross-amplitude** is $g_{ij}(\omega) = |f_{ij}(\omega)| = \sqrt{c_{ij}^2 + q_{ij}^2}$.

The **squared coherence** measures linear association at frequency ω:

$$R_{ij}^2(\omega) = \frac{|f_{ij}(\omega)|^2}{f_{ii}(\omega) f_{jj}(\omega)} \in [0, 1]. \tag{42.41}$$

Coherence measures how much of the variance of y_i at frequency ω can be "explained" by y_j at the same frequency.

High coherence means the two series move together tightly at that frequency.

The **gain** is the frequency-response coefficient when regressing y_i on y_j:

$$G_{ij}(\omega) = \frac{|f_{ij}(\omega)|}{f_{jj}(\omega)}. \tag{42.42}$$

It measures how much y_i responds to a unit change in y_j at frequency ω.

For instance, a gain of 0.9 at low frequencies means long-cycle movements in y_j translate almost one-for-one to y_i, and a gain of 0.3 at high frequencies means short-cycle movements are dampened.

The **phase** captures lead-lag relationships (in radians):

$$\Delta_{ij}(\omega) = \tan^{-1}\left(\frac{q_{ij}(\omega)}{c_{ij}(\omega)}\right). \tag{42.43}$$

```
def cross_spectral_measures(F, i, j):
    """Compute coherence, gain (y_i on y_j), and phase between variables i and j."""
    f_ij = F[:, i, j]
    f_ii, f_jj = np.real(F[:, i, i]), np.real(F[:, j, j])
    g_ij = np.abs(f_ij)
    coherence = (g_ij**2) / (f_ii * f_jj)
    gain = g_ij / f_jj
    phase = np.arctan2(-np.imag(f_ij), np.real(f_ij))
    return coherence, gain, phase
```

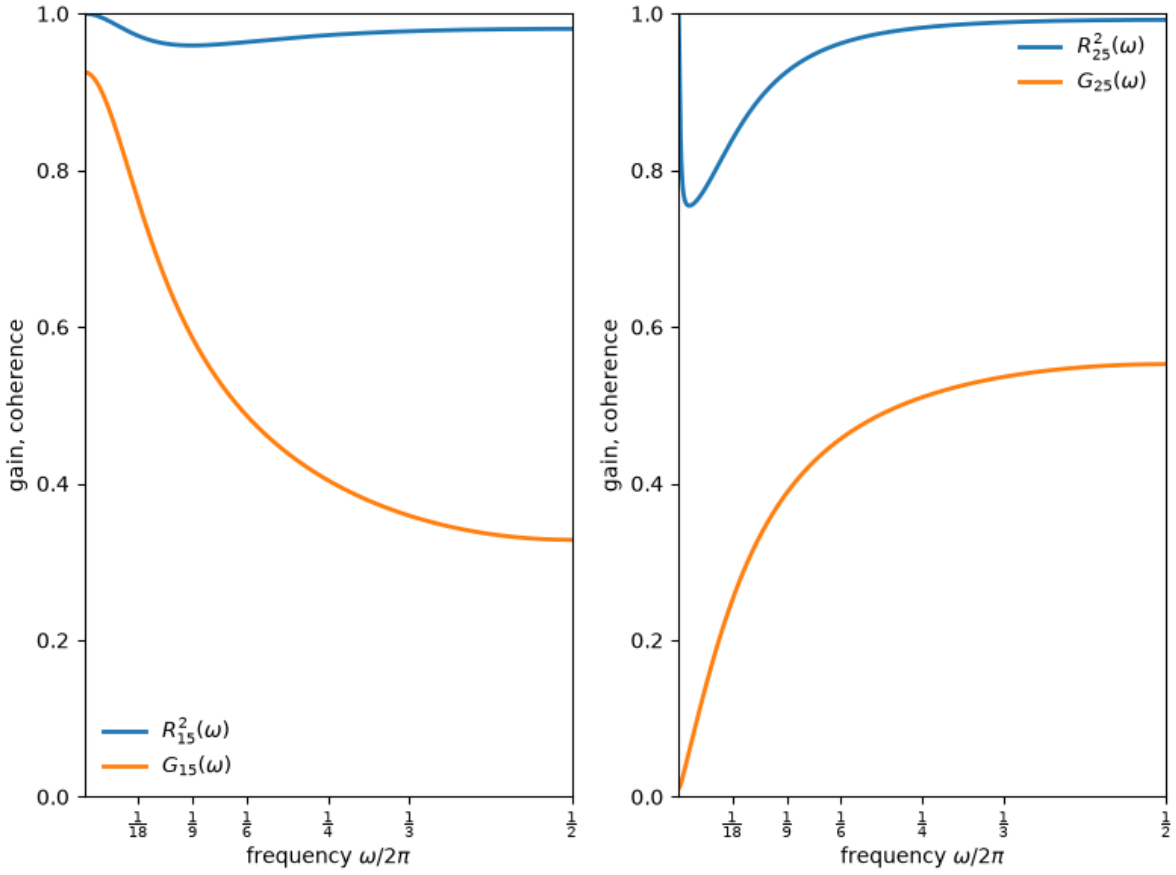
We now plot gain and coherence as in Figures II.1–II.4 of [Chow and Levitan, 1969].

```
gnp_idx = 4

fig, axes = plt.subplots(1, 2, figsize=(8, 6))

for idx, var_idx in enumerate([0, 1]):
    coherence, gain, phase = cross_spectral_measures(F_chow, var_idx, gnp_idx)
    ax = axes[idx]

    ax.plot(freq[mask], coherence[mask],
            lw=2, label=rf'$R^2_{{{var_idx+1}5}}(\omega)$')
    ax.plot(freq[mask], gain[mask],
            lw=2, label=rf'$G_{{{var_idx+1}5}}(\omega)$')
    paper_frequency_axis(ax)
    ax.set_ylim([0, 1.0])
    ax.set_ylabel('gain, coherence')
```

(continues on next page)

(continued from previous page)

```
    ax.legend(frameon=False, loc='best')

plt.tight_layout()
plt.show()
```



The gain and coherence patterns differ across components (Figures II.1–II.2 of [Chow and Levitan, 1969]):

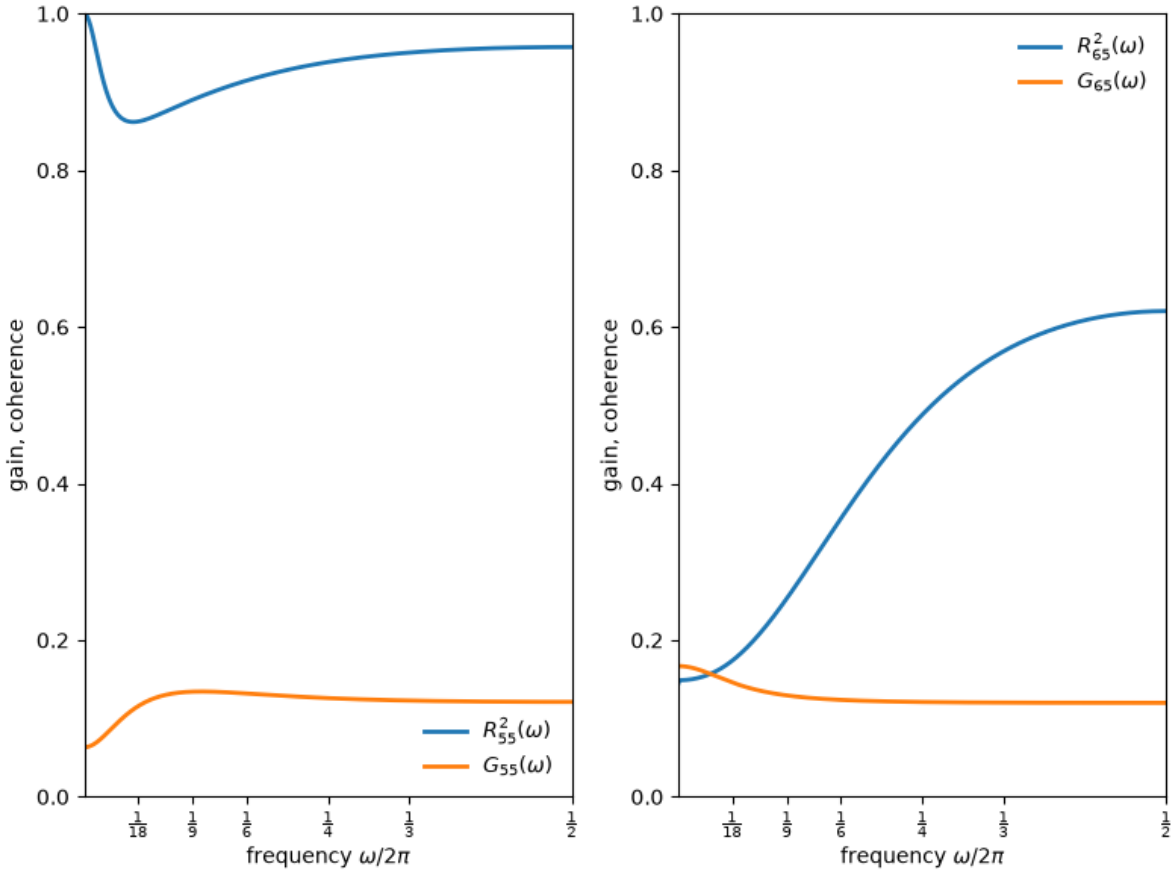
- Consumption vs private-domestic GNP (left panel):
  - Gain is about 0.9 at very low frequencies but falls below 0.4 for cycles shorter than four years.
  - This is evidence that short-cycle income movements translate less into consumption than long-cycle movements, consistent with permanent-income interpretations.
  - Coherence remains high throughout.
- Equipment plus inventories vs private-domestic GNP (right panel):
  - Gain *rises* with frequency, exceeding 0.5 for short cycles.
  - This is the frequency-domain signature of acceleration and volatile short-run inventory movements.

```
fig, axes = plt.subplots(1, 2, figsize=(8, 6))

for idx, var_idx in enumerate([2, 3]):
    coherence, gain, phase = cross_spectral_measures(F_chow, var_idx, gnp_idx)
    ax = axes[idx]
```

(continues on next page)

(continued from previous page)

```
    ax.plot(freq[mask], coherence[mask],
            lw=2, label=rf'$R^2_{{{var_idx+3}5}}(\omega)$')
    ax.plot(freq[mask], gain[mask],
            lw=2, label=rf'$G_{{{var_idx+3}5}}(\omega)$')
    paper_frequency_axis(ax)
    ax.set_ylim([0, 1.0])
    ax.set_ylabel('gain, coherence')
    ax.legend(frameon=False, loc='best')

plt.tight_layout()
plt.show()
```



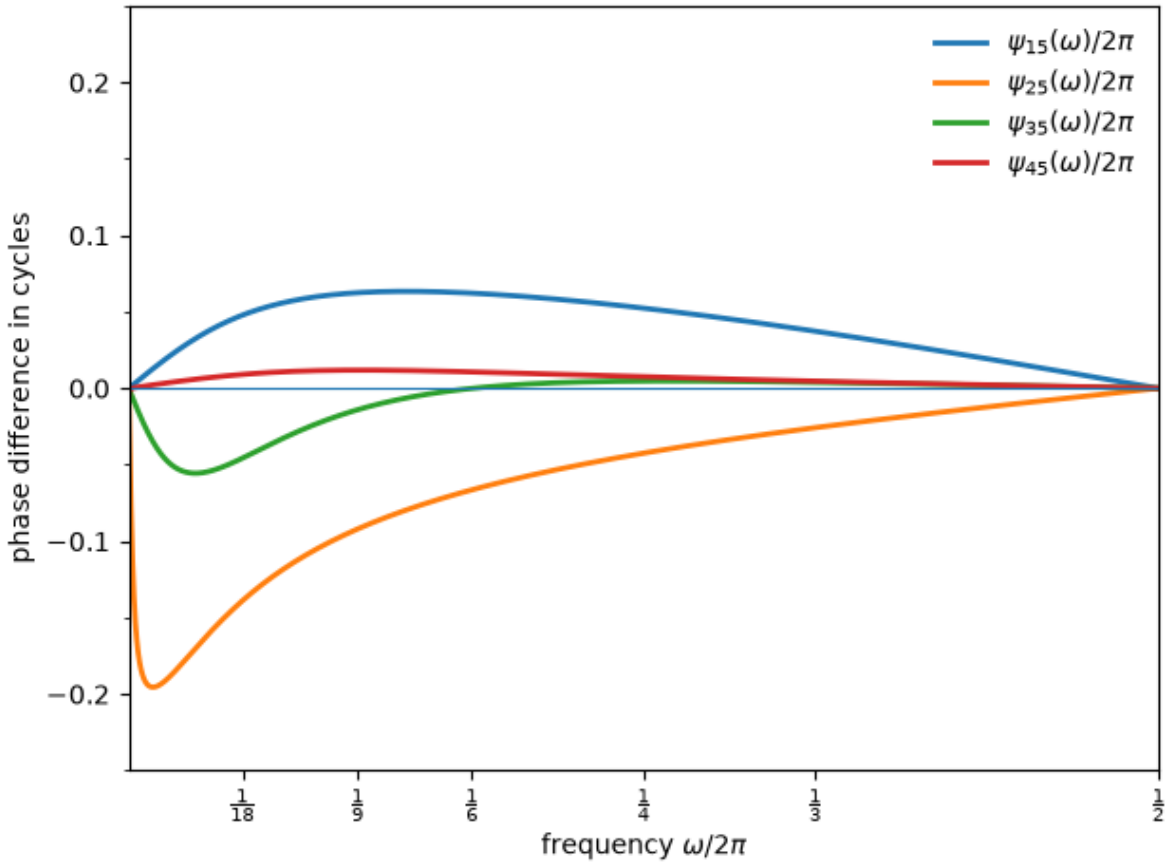
- New construction vs private-domestic GNP (left panel):
  - Gain peaks at medium cycle lengths (around 0.1 for short cycles).
  - Coherence for both investment series stays fairly high across frequencies.
- Long-bond yield vs private-domestic GNP (right panel):
  - Gain varies less across frequencies than real activity series.
  - Coherence with output is comparatively low at business-cycle frequencies, making it hard to explain interest-rate movements by inverting a money-demand equation.

42.8.5 Lead-lag relationships

The phase tells us which variable leads at each frequency.

Positive phase means output leads the component; negative phase means the component leads output.

```
fig, ax = plt.subplots()

labels = [r'$\psi_{15}(\omega)/2\pi$', r'$\psi_{25}(\omega)/2\pi$',
          r'$\psi_{35}(\omega)/2\pi$', r'$\psi_{45}(\omega)/2\pi$']

for var_idx in range(4):
    coherence, gain, phase = cross_spectral_measures(F_chow, var_idx, gnp_idx)
    phase_cycles = phase / (2 * np.pi)
    ax.plot(freq[mask], phase_cycles[mask], lw=2, label=labels[var_idx])

ax.axhline(0, lw=0.8)
paper_frequency_axis(ax)
ax.set_ylabel('phase difference in cycles')
ax.set_ylim([-0.25, 0.25])
ax.set_yticks(np.arange(-0.25, 0.3, 0.05), minor=True)
ax.legend(frameon=False)
plt.tight_layout()
plt.show()
```



The phase relationships reveal that:

- Output leads consumption by a small fraction of a cycle (about 0.06 cycles at a 6-year period, 0.04 cycles at a 3-year

period).

- Equipment plus inventories tends to lead output (by about 0.07 cycles at a 6-year period, 0.03 cycles at a 3-year period).
- New construction leads at low frequencies and is close to coincident at higher frequencies.
- The bond yield lags output slightly, remaining close to coincident in timing.

These implied leads and lags are broadly consistent with turning-point timing summaries reported elsewhere, and simulations of the same model deliver similar lead–lag ordering at turning points (Figure III of [Chow and Levitan, 1969]).

42.8.6 Building blocks of spectral shape

Each eigenvalue contributes a characteristic spectral shape through the *scalar kernel*

$$g_i(\omega) = \frac{1 - |\lambda_i|^2}{|1 - \lambda_i e^{-i\omega}|^2} = \frac{1 - |\lambda_i|^2}{1 + |\lambda_i|^2 - 2\mathrm{Re}(\lambda_i)\cos\omega + 2\mathrm{Im}(\lambda_i)\sin\omega}. \tag{42.44}$$

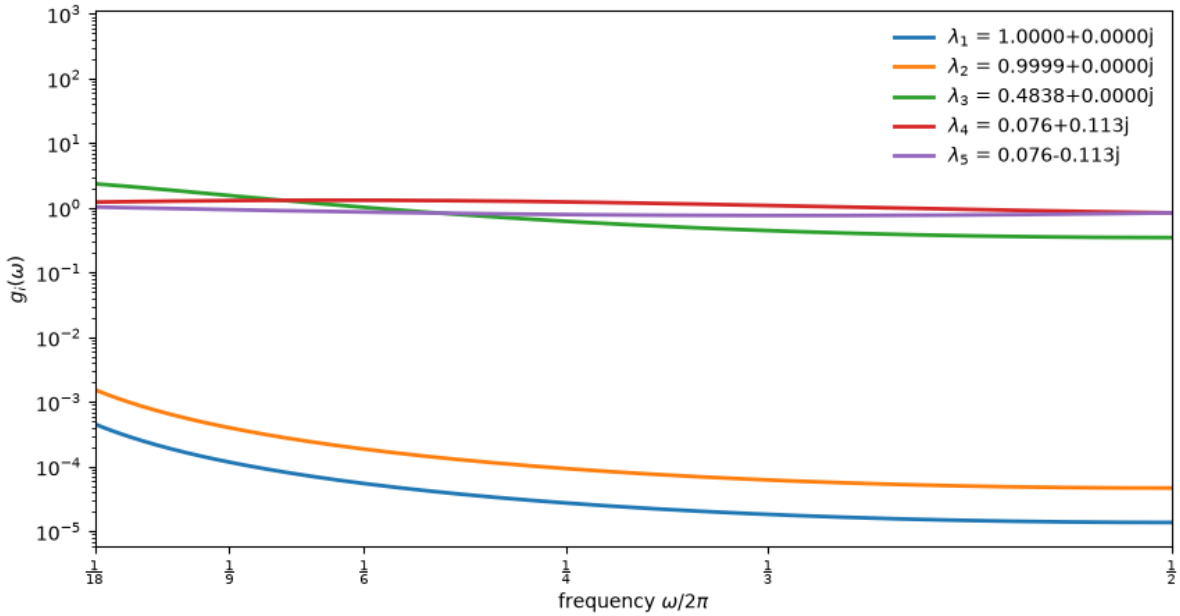
For real λ_i, this simplifies to

$$g_i(\omega) = \frac{1 - \lambda_i^2}{1 + \lambda_i^2 - 2\lambda_i \cos\omega}.$$

Each observable spectral density is a linear combination of these kernels (plus cross-terms).

Below, we plot the scalar kernels for each eigenvalue to see how they shape the overall spectra

```
def scalar_kernel(λ_i, ω_grid):
    """scalar spectral kernel g_i(ω)."""
    λ_i = complex(λ_i)
    mod_sq = np.abs(λ_i)**2
    return np.array(
        [(1 - mod_sq) / np.abs(1 - λ_i * np.exp(-1j * ω))**2
        for ω in ω_grid])

fig, ax = plt.subplots(figsize=(10, 5))
for i, λ_i in enumerate(λ):
    if np.abs(λ_i) > 0.01:
        g_i = scalar_kernel(λ_i, ω_grid)
        label = f'$\\lambda_{i+1}$ = {λ_i:.4f}' \
        if np.isreal(λ_i) else f'$\\lambda_{i+1}$ = {λ_i:.3f}'
        ax.semilogy(freq, g_i, label=label, lw=2)

ax.set_xlabel(r'frequency $\omega/2\pi$')
ax.set_ylabel('$g_i(\\omega)$')
ax.set_xlim([1/18, 0.5])
ax.set_xticks(freq_ticks)
ax.set_xticklabels(freq_labels)
ax.legend(frameon=False)
plt.show()
```



The figure reveals how eigenvalue magnitude shapes spectral contributions:

- *Near-unit eigenvalues* ($\lambda_1, \lambda_2 \approx 1$) produce kernels sharply peaked at low frequencies as these drive the strong low-frequency power seen in the spectra above.
- *The moderate eigenvalue* ($\lambda_3 \approx 0.48$) contributes a flatter component that spreads power more evenly across frequencies.
- *The complex pair* ($\lambda_{4,5}$) has such small modulus ($|\lambda_{4,5}| \approx 0.136$) that its kernel is nearly flat, which is too weak to generate a pronounced interior peak.

This decomposition explains why the spectra look the way they do: the near-unit eigenvalues dominate, concentrating variance at very low frequencies.

The complex pair, despite enabling oscillatory dynamics in principle, has insufficient modulus to produce a visible spectral peak.

42.9 Summary

Chow [1968] draws several conclusions that remain relevant for understanding business cycles.

The acceleration principle receives strong empirical support: the negative coefficient on lagged output in investment equations is a robust finding across datasets.

The relationship between eigenvalues and spectral peaks is more subtle than it first appears:

- Complex roots guarantee oscillatory autocovariances, but they are neither necessary nor sufficient for a pronounced spectral peak.
- In the Hansen–Samuelson model specifically, complex roots *are* necessary for a peak.
- But in general multivariate systems, even real roots can produce peaks through the interaction of shocks and eigenvector loadings.

Chow and Levitan [1969] demonstrate what these objects look like in a calibrated system: strong low-frequency power from near-unit eigenvalues, frequency-dependent gains and coherences, and lead–lag relations that vary with cycle length.

Their results are consistent with Granger's "typical spectral shape" for economic time series.

That is a monotonically decreasing function of frequency, driven by the near-unit eigenvalues that arise when some equations are specified in first differences.

Understanding whether this shape reflects the true data-generating process requires analyzing the spectral densities implied by structural econometric models.

42.10 Exercises

Exercise 42.10.1

Plot impulse responses and spectra side-by-side for several values of the accelerator v in the Hansen-Samuelson model, showing how acceleration strength affects both the time-domain and frequency-domain signatures.

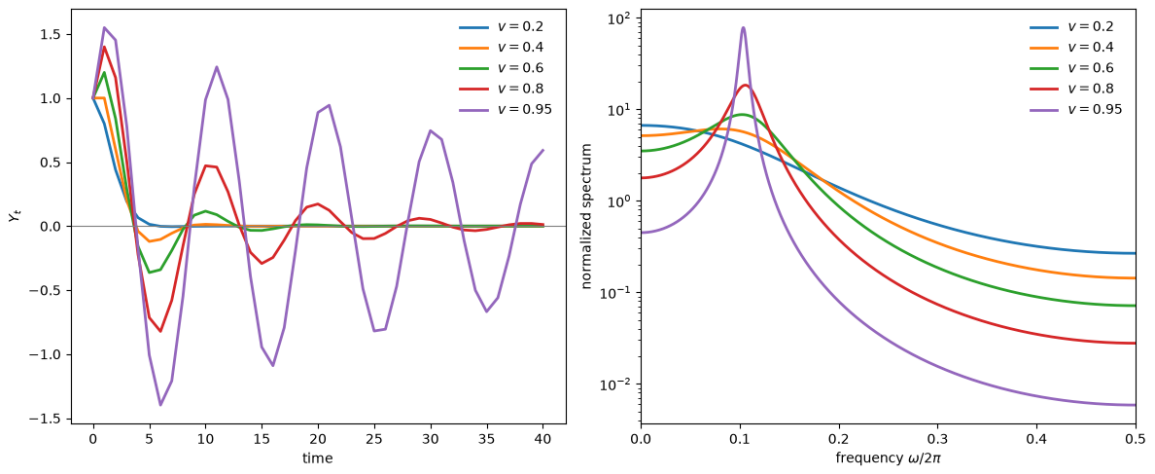
Use the same v values as in the main text: $v \in \{0.2, 0.4, 0.6, 0.8, 0.95\}$ with $c = 0.6$.

Solution

Here is one solution:

```
v_grid_ex1 = [0.2, 0.4, 0.6, 0.8, 0.95]
c_ex1 = 0.6
freq_ex1 = np.linspace(1e-4, 0.5, 2000)
ω_grid_ex1 = 2 * np.pi * freq_ex1
V_ex1 = np.array([[1.0, 0.0], [0.0, 0.0]])
T_irf_ex1 = 40

fig, axes = plt.subplots(1, 2, figsize=(12, 5))

for v in v_grid_ex1:
    A = samuelson_transition(c_ex1, v)

    # impulse response (left panel)
    s = np.array([1.0, 0.0])
    irf = np.empty(T_irf_ex1 + 1)
    for t in range(T_irf_ex1 + 1):
        irf[t] = s[0]
        s = A @ s
    axes[0].plot(range(T_irf_ex1 + 1), irf, lw=2, label=f'$v={v}$')

    # spectrum (right panel)
    F = spectral_density_var1(A, V_ex1, ω_grid_ex1)
    f11 = np.real(F[:, 0, 0])
    df = np.diff(freq_ex1)
    area = np.sum(0.5 * (f11[1:] + f11[:-1]) * df)
    f11_norm = f11 / area
    axes[1].plot(freq_ex1, f11_norm, lw=2, label=f'$v={v}$')

axes[0].axhline(0, lw=0.8, color='gray')
axes[0].set_xlabel('time')
axes[0].set_ylabel(r'$Y_t$')
axes[0].legend(frameon=False)

axes[1].set_xlabel(r'frequency $\omega/2\pi$')
axes[1].set_ylabel('normalized spectrum')
```

```
axes[1].set_xlim([0, 0.5])
axes[1].set_yscale('log')
axes[1].legend(frameon=False)

plt.tight_layout()
plt.show()
```



As v increases, eigenvalues approach the unit circle: oscillations become more persistent in the time domain (left), and the spectral peak becomes sharper in the frequency domain (right).

Complex roots produce a pronounced peak at interior frequencies–the spectral signature of business cycles.

Exercise 42.10.2

Verify spectral peak condition (42.25) numerically for the Hansen-Samuelson model.

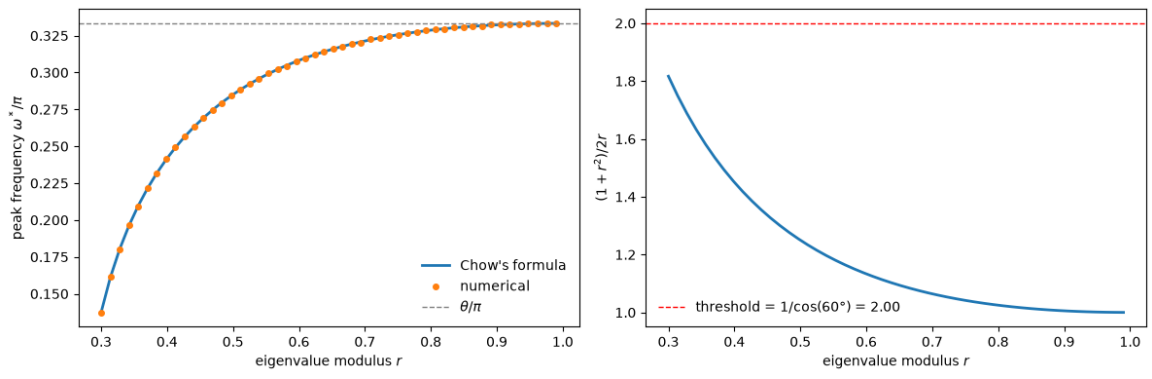
1. For a range of eigenvalue moduli $r \in [0.3, 0.99]$ with fixed $\theta = 60°$, compute:
   - The theoretical peak frequency from formula: $\cos\omega = \frac{1+r^2}{2r}\cos\theta$
   - The actual peak frequency by numerically maximizing the spectral density
2. Plot both on the same graph and verify they match.
3. Identify the range of r for which no valid peak exists (when the condition (42.26) is violated).

Solution

Here is one solution:

```
θ_ex = np.pi / 3  # 60 degrees
r_grid = np.linspace(0.3, 0.99, 50)
ω_grid_ex = np.linspace(1e-3, np.pi - 1e-3, 1000)
V_hs_ex = np.array([[1.0, 0.0], [0.0, 0.0]])

ω_theory = []
ω_numerical = []

for r in r_grid:
    # Theoretical peak
```

```
    factor = (1 + r**2) / (2 * r)
    cos_ω = factor * np.cos(θ_ex)
    if -1 < cos_ω < 1:
        ω_theory.append(np.arccos(cos_ω))
    else:
        ω_theory.append(np.nan)

    # Numerical peak from spectral density
    # Construct Hansen-Samuelson with eigenvalues r*exp(+-iθ)
    # This corresponds to c + v = 2r*cos(θ), v = r^2
    v = r**2
    c = 2 * r * np.cos(θ_ex) - v
    A_ex = samuelson_transition(c, v)
    F_ex = spectral_density_var1(A_ex, V_hs_ex, ω_grid_ex)
    f11 = np.real(F_ex[:, 0, 0])
    i_max = np.argmax(f11)

    # Only count as a peak if it's not at the boundary
    if 5 < i_max < len(ω_grid_ex) - 5:
        ω_numerical.append(ω_grid_ex[i_max])
    else:
        ω_numerical.append(np.nan)

ω_theory = np.array(ω_theory)
ω_numerical = np.array(ω_numerical)

fig, axes = plt.subplots(1, 2, figsize=(12, 4))

# Plot peak frequencies
axes[0].plot(r_grid, ω_theory / np.pi, lw=2, label="Chow's formula")
axes[0].plot(r_grid, ω_numerical / np.pi, 'o', markersize=4, label='numerical')
axes[0].axhline(θ_ex / np.pi, ls='--', lw=1.0, color='gray', label=r'$\theta/\pi$')
axes[0].set_xlabel('eigenvalue modulus $r$')
axes[0].set_ylabel(r'peak frequency $\omega^*/\pi$')
axes[0].legend(frameon=False)

# Plot the factor (1+r^2)/2r to show when peaks are valid
axes[1].plot(r_grid, (1 + r_grid**2) / (2 * r_grid), lw=2)
axes[1].axhline(1 / np.cos(θ_ex), ls='--', lw=1.0, color='red',
                label=f'threshold = 1/cos({np.rad2deg(θ_ex):.0f}°) = {1/np.cos(θ_
↪ex):.2f}')
axes[1].set_xlabel('eigenvalue modulus $r$')
axes[1].set_ylabel(r'$(1+r^2)/2r$')
axes[1].legend(frameon=False)

plt.tight_layout()
plt.show()

# Find threshold r below which no peak exists
valid_mask = ~np.isnan(ω_theory)
if valid_mask.any():
    r_threshold = r_grid[valid_mask][0]
    print(f"Peak exists for r >= {r_threshold:.2f}")
```




```
Peak exists for r >= 0.30
```

The theoretical and numerical peak frequencies match closely.

As $r \to 1$, the peak frequency converges to θ.

For smaller r, the factor $(1 + r^2)/2r$ exceeds the threshold, and no valid peak exists.

Exercise 42.10.3

In the "real roots but a peak" example, hold A fixed and vary the shock correlation (the off-diagonal entry of V) between 0 and 0.99.

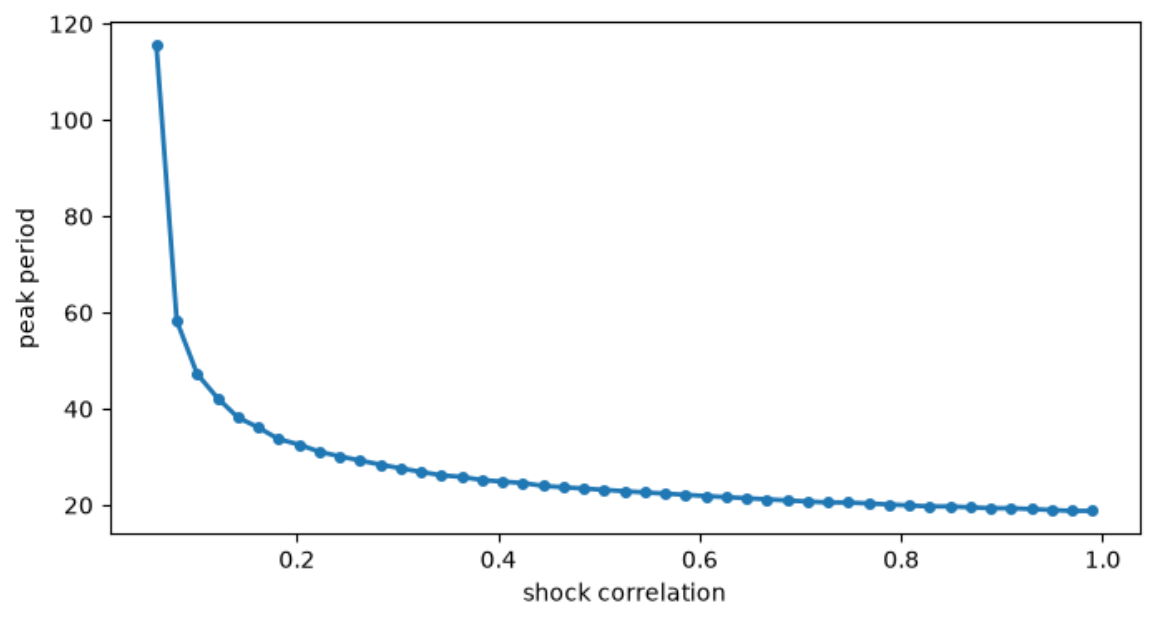
When does the interior-frequency peak appear, and how does its location change?

Solution

Here is one solution:

```
A_ex3 = np.diag([0.1, 0.9])
b_ex3 = np.array([1.0, -0.01])
corr_grid = np.linspace(0, 0.99, 50)
peak_periods = []
for corr in corr_grid:
    V_ex3 = np.array([[1.0, corr], [corr, 1.0]])
    F_ex3 = spectral_density_var1(A_ex3, V_ex3, ω_grid_ex)
    f_x = spectrum_of_linear_combination(F_ex3, b_ex3)
    i_max = np.argmax(f_x)
    if 5 < i_max < len(ω_grid_ex) - 5:
        peak_periods.append(2 * np.pi / ω_grid_ex[i_max])
    else:
        peak_periods.append(np.nan)

fig, ax = plt.subplots(figsize=(8, 4))
ax.plot(corr_grid, peak_periods, marker='o', lw=2, markersize=4)
ax.set_xlabel('shock correlation')
ax.set_ylabel('peak period')
plt.show()

threshold_idx = np.where(~np.isnan(peak_periods))[0]
if len(threshold_idx) > 0:
    print(
```

```
    f"interior peak when correlation >= {corr_grid[threshold_idx[0]]:.2f}")
```




```
interior peak when correlation >= 0.06
```

The interior peak appears only when the shock correlation exceeds a threshold.

This illustrates that spectral peaks depend on the full system structure, not just eigenvalues.

Exercise 42.10.4

Using the calibrated Chow-Levitan parameters, compute the autocovariance matrices $\Gamma_0, \Gamma_1, \ldots, \Gamma_{10}$ using:

1. The recursion $\Gamma_k = A\Gamma_{k-1}$ with Γ_0 from the Lyapunov equation.
2. Chow's eigendecomposition formula $\Gamma_k = BD_\lambda^k \Gamma_0^* B^\top$ where Γ_0^* is the canonical covariance.

Verify that both methods give the same result.

Solution

Here is one solution:

```
from scipy.linalg import solve_discrete_lyapunov

Γ_0_lyap = solve_discrete_lyapunov(A_chow, V)
Γ_recursion = [Γ_0_lyap]
for k in range(1, 11):
    Γ_recursion.append(A_chow @ Γ_recursion[-1])

p = len(λ)
Γ_0_star = np.zeros((p, p), dtype=complex)
for i in range(p):
    for j in range(p):
        Γ_0_star[i, j] = W[i, j] / (1 - λ[i] * λ[j])
```

```
Γ_eigen = []
for k in range(11):
    D_k = np.diag(λ**k)
    Γ_eigen.append(np.real(B @ D_k @ Γ_0_star @ B.T))

print("Comparison of Γ_5 (first 3x3 block):")
print("\nRecursion method:")
print(np.real(Γ_recursion[5][:3, :3]).round(10))
print("\nEigendecomposition method:")
print(Γ_eigen[5][:3, :3].round(10))
print("\nMax absolute difference:",
        np.max(np.abs(np.real(Γ_recursion[5]) - Γ_eigen[5])))
```

```
Comparison of Γ_5 (first 3x3 block):

Recursion method:
[[2.5417901e-03 2.9310700e-05 1.7370210e-04]
 [2.8886900e-05 3.3220000e-07 1.9737000e-06]
 [1.7356020e-04 2.0028000e-06 1.1861700e-05]]

Eigendecomposition method:
[[2.5417901e-03 2.9310700e-05 1.7370210e-04]
 [2.8886900e-05 3.3220000e-07 1.9737000e-06]
 [1.7356020e-04 2.0028000e-06 1.1861700e-05]]

Max absolute difference: 9.500386588534582e-14
```

Both methods produce essentially identical results, up to numerical precision.

Exercise 42.10.5

Modify the Chow-Levitan model by changing λ_3 from 0.4838 to 0.95.

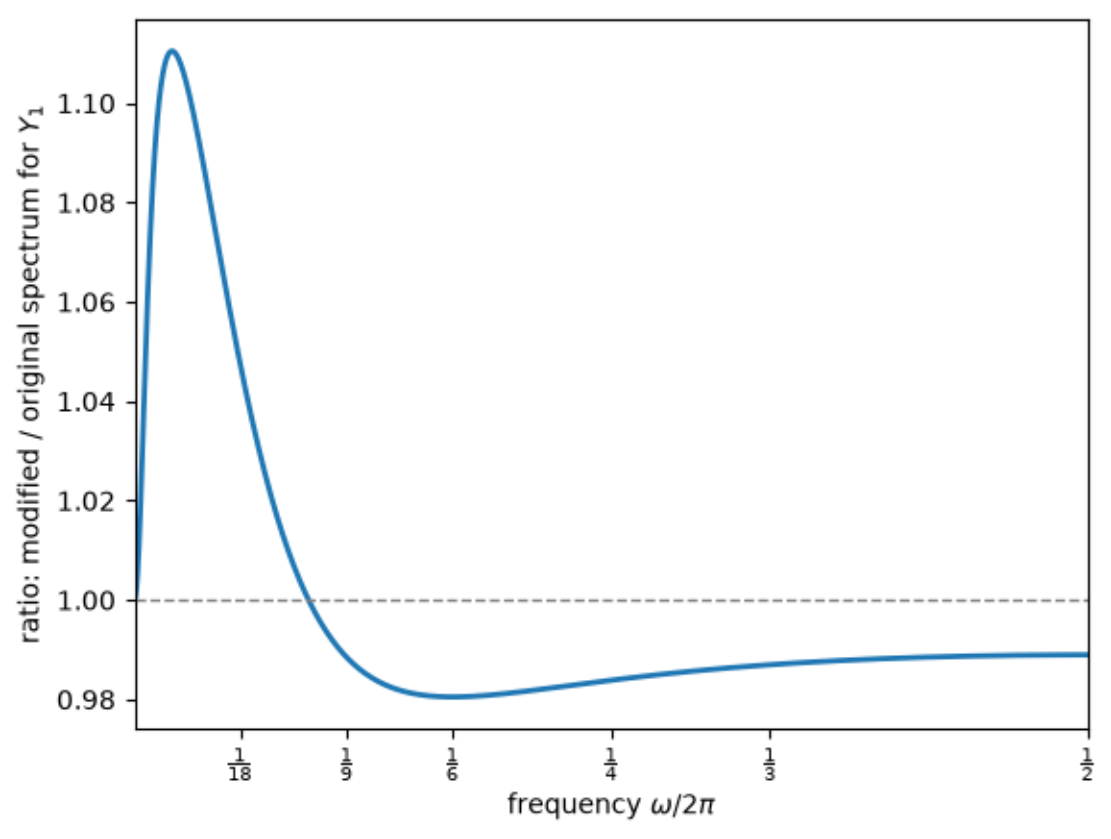
1. Recompute the spectral densities.
2. How does this change affect the spectral shape for each variable?
3. What economic interpretation might correspond to this parameter change?

Solution

Here is one solution:

```
# Modify λ_3 and reconstruct the transition matrix
λ_modified = λ.copy()
λ_modified[2] = 0.95
D_λ_mod = np.diag(λ_modified)
A_mod = np.real(B @ D_λ_mod @ np.linalg.inv(B))

# Compute spectra using the VAR(1) formula with original V
F_mod = spectral_density_var1(A_mod, V, ω_grid)
F_orig = spectral_density_var1(A_chow, V, ω_grid)

# Plot ratio of spectra for output (Y_1)
f_orig = np.real(F_orig[:, 4, 4])
f_mod = np.real(F_mod[:, 4, 4])
```

```
fig, ax = plt.subplots()
ax.plot(freq, f_mod / f_orig, lw=2)
ax.axhline(1.0, ls='--', lw=1, color='gray')
paper_frequency_axis(ax)
ax.set_ylabel(r"ratio: modified / original spectrum for $Y_1$")
plt.show()
```



The near-unit eigenvalues ($\lambda_1, \lambda_2 \approx 0.9999$) dominate the output spectrum so heavily that changing λ_3 from 0.48 to 0.95 produces only a small relative effect.

The ratio plot reveals the change: the modified spectrum has slightly more power at low-to-medium frequencies and slightly less at high frequencies.

Economically, increasing λ_3 adds persistence to the mode it governs.

CHAPTER

FORTYTHREE

KESTEN PROCESSES AND FIRM DYNAMICS

Contents

- *Kesten Processes and Firm Dynamics*
  - *Overview*
  - *Kesten processes*
  - *Heavy tails*
  - *Application: firm dynamics*
  - *Exercises*

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install --upgrade quantecon yfinance
```

43.1 Overview

Previously in AR(1) Processes, we learned about linear scalar-valued stochastic processes (AR(1) models).

Now we generalize these linear models slightly by allowing the multiplicative coefficient to be stochastic.

Such processes are known as Kesten processes after German–American mathematician Harry Kesten (1931–2019).

Although simple to write down, Kesten processes are interesting for at least two reasons:

1. A number of significant economic processes are or can be described as Kesten processes.

2. Kesten processes generate interesting dynamics, including, in some cases, heavy-tailed cross-sectional distributions.

We will discuss these issues as we go along.

Let's start with some imports:

```
import matplotlib.pyplot as plt
import numpy as np
import quantecon as qe
import yfinance as yf
```

Additional technical background related to this lecture can be found in the monograph by [Buraczewski *et al.*, 2016].

43.2 Kesten processes

A **Kesten process** is a stochastic process of the form

$$X_{t+1} = a_{t+1} X_t + \eta_{t+1} \tag{43.1}$$

where $\{a_t\}_{t \geq 1}$ and $\{\eta_t\}_{t \geq 1}$ are IID sequences.

We are interested in the dynamics of $\{X_t\}_{t \geq 0}$ when X_0 is given.

We will focus on the nonnegative scalar case, where X_t takes values in $\mathbb{R}_+$.

In particular, we will assume that

- the initial condition X_0 is nonnegative,
- $\{a_t\}_{t \geq 1}$ is a nonnegative IID stochastic process and
- $\{\eta_t\}_{t \geq 1}$ is another nonnegative IID stochastic process, independent of the first.

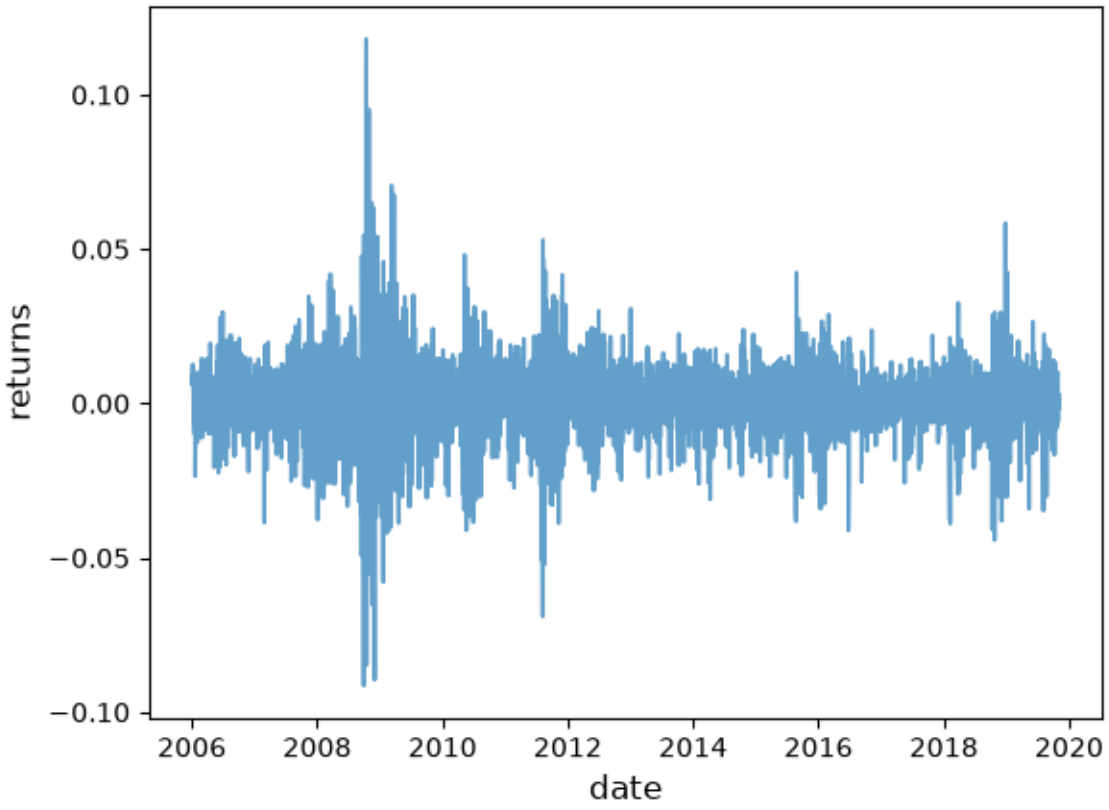
43.2.1 Example: GARCH volatility

The GARCH model is common in financial applications, where time series such as asset returns exhibit time varying volatility.

For example, consider the following plot of daily returns on the Nasdaq Composite Index for the period 1st January 2006 to 1st November 2019.

```
s = yf.download(
    "^IXIC", "2006-1-1", "2019-11-1", auto_adjust=False
)["Adj Close"]

r = s.pct_change()

fig, ax = plt.subplots()
ax.plot(r, alpha=0.7)
ax.set_ylabel("returns", fontsize=12)
ax.set_xlabel("date", fontsize=12)

plt.show()
```

```
[*********************100%***********************]  1 of 1 completed
```



Notice how the series exhibits bursts of volatility (high variance) and then settles down again.

GARCH models can replicate this feature.

The GARCH(1, 1) volatility process takes the form

$$\sigma_{t+1}^2 = \alpha_0 + \sigma_t^2(\alpha_1 \xi_{t+1}^2 + \beta) \tag{43.2}$$

where $\{\xi_t\}$ is IID with $\mathbb{E}\xi_t^2 = 1$ and all parameters are positive.

Returns on a given asset are then modeled as

$$r_t = \sigma_t \zeta_t \tag{43.3}$$

where $\{\zeta_t\}$ is again IID and independent of $\{\xi_t\}$.

The volatility sequence $\{\sigma_t^2\}$, which drives the dynamics of returns, is a Kesten process.

43.2.2 Example: wealth dynamics

Suppose that a given household saves a fixed fraction s of its current wealth in every period.

The household earns labor income y_t at the start of time t.

Wealth then evolves according to

$$w_{t+1} = R_{t+1} s w_t + y_{t+1} \tag{43.4}$$

where $\{R_t\}$ is the gross rate of return on assets.

If $\{R_t\}$ and $\{y_t\}$ are both IID, then (43.4) is a Kesten process.

43.2.3 Stationarity

In earlier lectures, such as the one on AR(1) Processes, we introduced the notion of a stationary distribution.

In the present context, we can define a stationary distribution as follows:

The distribution F^* on $\mathbb{R}$ is called **stationary** for the Kesten process (43.1) if

$$X_t \sim F^* \quad \Longrightarrow \quad a_{t+1} X_t + \eta_{t+1} \sim F^* \tag{43.5}$$

In other words, if the current state X_t has distribution F^*, then so does the next period state X_{t+1}.

We can write this alternatively as

$$F^*(y) = \int \mathbb{P}\{a_{t+1} x + \eta_{t+1} \leq y\} F^*(dx) \quad \text{for all } y \geq 0. \tag{43.6}$$

The left hand side is the distribution of the next period state when the current state is drawn from F^*.

The equality in (43.6) states that this distribution is unchanged.

43.2.4 Cross-sectional interpretation

There is an important cross-sectional interpretation of stationary distributions, discussed previously but worth repeating here.

Suppose, for example, that we are interested in the wealth distribution — that is, the current distribution of wealth across households in a given country.

Suppose further that

- the wealth of each household evolves independently according to (43.4),
- F^* is a stationary distribution for this stochastic process and
- there are many households.

Then F^* is a steady state for the cross-sectional wealth distribution in this country.

In other words, if F^* is the current wealth distribution then it will remain so in subsequent periods, *ceteris paribus*.

To see this, suppose that F^* is the current wealth distribution.

What is the fraction of households with wealth less than y next period?

To obtain this, we sum the probability that wealth is less than y tomorrow, given that current wealth is w, weighted by the fraction of households with wealth w.

Noting that the fraction of households with wealth in interval dw is $F^*(dw)$, we get

$$\int \mathbb{P}\{R_{t+1} sw + y_{t+1} \leq y\} F^*(dw)$$

By the definition of stationarity and the assumption that F^* is stationary for the wealth process, this is just $F^*(y)$.

Hence the fraction of households with wealth in $[0, y]$ is the same next period as it is this period.

Since y was chosen arbitrarily, the distribution is unchanged.

43.2.5 Conditions for stationarity

The Kesten process $X_{t+1} = a_{t+1}X_t + \eta_{t+1}$ does not always have a stationary distribution.

For example, if $a_t \equiv \eta_t \equiv 1$ for all t, then $X_t = X_0 + t$, which diverges to infinity.

To prevent this kind of divergence, we require that $\{a_t\}$ is strictly less than 1 most of the time.

In particular, if

$$\mathbb{E} \ln a_t < 0 \quad \text{and} \quad \mathbb{E}\eta_t < \infty \tag{43.7}$$

then a unique stationary distribution exists on $\mathbb{R}_+$.

- See, for example, theorem 2.1.3 of [Buraczewski *et al.*, 2016], which provides slightly weaker conditions.

As one application of this result, we see that the wealth process (43.4) will have a unique stationary distribution whenever labor income has finite mean and $\mathbb{E} \ln R_t + \ln s < 0$.

43.3 Heavy tails

Under certain conditions, the stationary distribution of a Kesten process has a Pareto tail.

(See our lecture Heavy-Tailed Distributions for background.)

This fact is significant for economics because of the prevalence of Pareto-tailed distributions.

43.3.1 The Kesten–Goldie theorem

To state the conditions under which the stationary distribution of a Kesten process has a Pareto tail, we first recall that a random variable is called **nonarithmetic** if its distribution is not concentrated on $\{\dots, -2t, -t, 0, t, 2t, \dots\}$ for any $t \geq 0$.

For example, any random variable with a density is nonarithmetic.

The famous Kesten–Goldie Theorem (see, e.g., [Buraczewski *et al.*, 2016], theorem 2.4.4) states that if

1. the stationarity conditions in (43.7) hold,
2. the random variable a_t is positive with probability one and nonarithmetic,
3. $\mathbb{P}\{a_t x + \eta_t = x\} < 1$ for all $x \in \mathbb{R}_+$ and
4. there exists a positive constant α such that

$$\mathbb{E}a_t^\alpha = 1, \quad \mathbb{E}\eta_t^\alpha < \infty, \quad \text{and} \quad \mathbb{E}[a_t^{\alpha+1}] < \infty$$

then the stationary distribution of the Kesten process has a Pareto tail with tail index α.

More precisely, if F^* is the unique stationary distribution and $X^* \sim F^*$, then

$$\lim_{x \to \infty} x^\alpha \mathbb{P}\{X^* > x\} = c$$

for some positive constant c.

43.3.2 Intuition

Later we will illustrate the Kesten–Goldie Theorem using rank-size plots.

Prior to doing so, we can give the following intuition for the conditions.

Two important conditions are that $\mathbb{E} \ln a_t < 0$, so the model is stationary, and $\mathbb{E} a_t^\alpha = 1$ for some $\alpha > 0$.

The first condition implies that the distribution of a_t has a large amount of probability mass below 1.

The second condition implies that the distribution of a_t has at least some probability mass at or above 1.

The first condition gives us existence of the stationary condition.

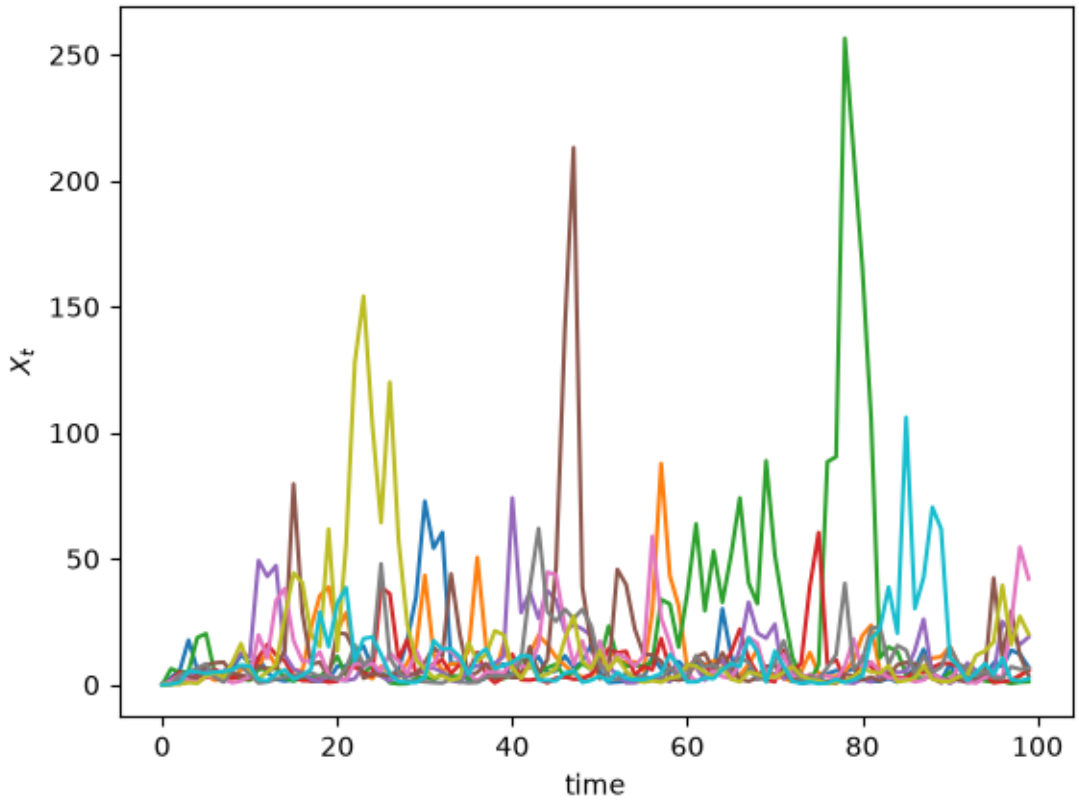
The second condition means that the current state can be expanded by a_t.

If this occurs for several concurrent periods, the effects compound each other, since a_t is multiplicative.

This leads to spikes in the time series, which fill out the extreme right hand tail of the distribution.

The spikes in the time series are visible in the following simulation, which generates 10 paths when a_t and b_t are lognormal.

```
μ = -0.5
σ = 1.0

def kesten_ts(rng, ts_length=100):
    x = np.zeros(ts_length)
    for t in range(ts_length - 1):
        a = np.exp(μ + σ * rng.standard_normal())
        b = np.exp(rng.standard_normal())
        x[t+1] = a * x[t] + b
    return x

fig, ax = plt.subplots()

num_paths = 10
rng = np.random.default_rng(12)

for i in range(num_paths):
    ax.plot(kesten_ts(rng))

ax.set(xlabel="time", ylabel="$X_t$")
plt.show()
```



43.4 Application: firm dynamics

As noted in our lecture Heavy-Tailed Distributions, for common measures of firm size such as revenue or employment, the US firm size distribution exhibits a Pareto tail (see, e.g., [Axtell, 2001], [Gabaix, 2016]).

Let us try to explain this rather striking fact using the Kesten–Goldie Theorem.

43.4.1 Gibrat's law

It was postulated many years ago by Robert Gibrat [Gibrat, 1931] that firm size evolves according to a simple rule whereby size next period is proportional to current size.

This is now known as Gibrat's law of proportional growth.

We can express this idea by stating that a suitably defined measure s_t of firm size obeys

$$\frac{s_{t+1}}{s_t} = a_{t+1} \tag{43.8}$$

for some positive IID sequence $\{a_t\}$.

One implication of Gibrat's law is that the growth rate of individual firms does not depend on their size.

However, over the last few decades, research contradicting Gibrat's law has accumulated in the literature.

For example, it is commonly found that, on average,

1. small firms grow faster than large firms (see, e.g., [Evans, 1987] and [Hall, 1987]) and

2. the growth rate of small firms is more volatile than that of large firms [Dunne *et al.*, 1989].

On the other hand, Gibrat's law is generally found to be a reasonable approximation for large firms [Evans, 1987].

We can accommodate these empirical findings by modifying (43.8) to

$$s_{t+1} = a_{t+1} s_t + b_{t+1} \tag{43.9}$$

where $\{a_t\}$ and $\{b_t\}$ are both IID and independent of each other.

In the exercises you are asked to show that (43.9) is more consistent with the empirical findings presented above than Gibrat's law in (43.8).

43.4.2 Heavy tails

So what has this to do with Pareto tails?

The answer is that (43.9) is a Kesten process.

If the conditions of the Kesten–Goldie Theorem are satisfied, then the firm size distribution is predicted to have heavy tails — which is exactly what we see in the data.

In the exercises below we explore this idea further, generalizing the firm size dynamics and examining the corresponding rank-size plots.

We also try to illustrate why the Pareto tail finding is significant for quantitative analysis.

43.5 Exercises

Exercise 43.5.1

Simulate and plot 15 years of daily returns (consider each year as having 250 working days) using the GARCH(1, 1) process in (43.2)–(43.3).

Take ξ_t and ζ_t to be independent and standard normal.

Set $\alpha_0 = 0.00001, \alpha_1 = 0.1, \beta = 0.9$ and $\sigma_0 = 0$.

Compare visually with the Nasdaq Composite Index returns *shown above*.

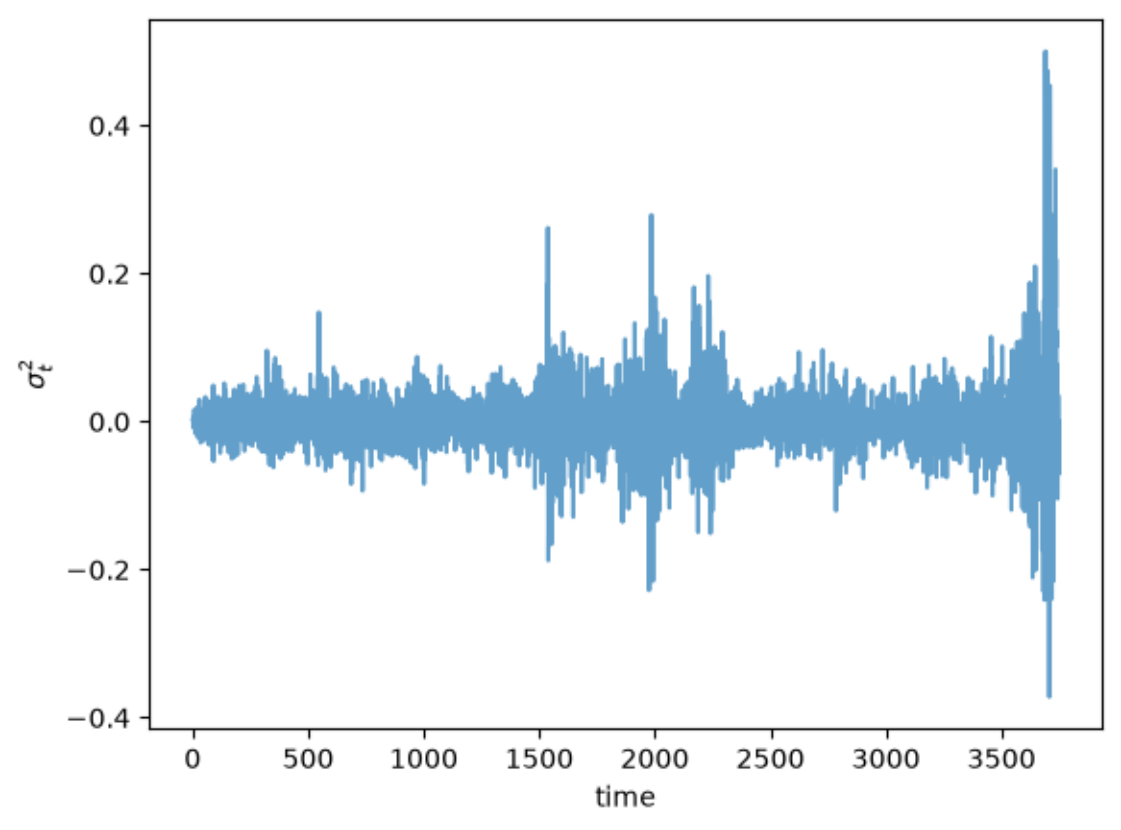
While the time path differs, you should see bursts of high volatility.

Solution

Here is one solution:

```
α_0 = 1e-5
α_1 = 0.1
β = 0.9

years = 15
days = years * 250

def garch_ts(rng, ts_length=days):
    σ2 = 0
    r = np.zeros(ts_length)
```

```
    for t in range(ts_length - 1):
        ξ = rng.standard_normal()
        σ2 = α_0 + σ2 * (α_1 * ξ**2 + β)
        r[t] = np.sqrt(σ2) * rng.standard_normal()
    return r

fig, ax = plt.subplots()

rng = np.random.default_rng(12)

ax.plot(garch_ts(rng), alpha=0.7)

ax.set(xlabel="time", ylabel="$\\sigma_t^2$")
plt.show()
```



Exercise 43.5.2

In our discussion of firm dynamics, it was claimed that (43.9) is more consistent with the empirical literature than Gibrat's law in (43.8).

(The empirical literature was reviewed immediately above (43.9).)

In what sense is this true (or false)?

Solution

The empirical findings are that

1. small firms grow faster than large firms and
2. the growth rate of small firms is more volatile than that of large firms.

Also, Gibrat's law is generally found to be a reasonable approximation for large firms than for small firms

The claim is that the dynamics in (43.9) are more consistent with points 1-2 than Gibrat's law.

To see why, we rewrite (43.9) in terms of growth dynamics:

$$\frac{s_{t+1}}{s_t} = a_{t+1} + \frac{b_{t+1}}{s_t} \tag{43.10}$$

Taking $s_t = s$ as given, the mean and variance of firm growth are

$$\mathbb{E}a + \frac{\mathbb{E}b}{s} \quad \text{and} \quad \mathbb{V}a + \frac{\mathbb{V}b}{s^2}$$

Both of these decline with firm size s, consistent with the data.

Moreover, the law of motion (43.10) clearly approaches Gibrat's law (43.8) as s_t gets large.

Exercise 43.5.3

Consider an arbitrary Kesten process as given in (43.1).

Suppose that $\{a_t\}$ is lognormal with parameters (μ, σ).

In other words, each a_t has the same distribution as $\exp(\mu + \sigma Z)$ when Z is standard normal.

Suppose further that $\mathbb{E}\eta_t^r < \infty$ for every $r > 0$, as would be the case if, say, η_t is also lognormal.

Show that the conditions of the Kesten–Goldie theorem are satisfied if and only if $\mu < 0$.

Obtain the value of α that makes the Kesten–Goldie conditions hold.

Solution

Since a_t has a density it is nonarithmetic.

Since a_t has the same density as $a = \exp(\mu + \sigma Z)$ when Z is standard normal, we have

$$\mathbb{E} \ln a_t = \mathbb{E}(\mu + \sigma Z) = \mu,$$

and since η_t has finite moments of all orders, the stationarity condition holds if and only if $\mu < 0$.

Given the properties of the lognormal distribution (which has finite moments of all orders), the only other condition in doubt is existence of a positive constant α such that $\mathbb{E} a_t^\alpha = 1$.

This is equivalent to the statement

$$\exp\left(\alpha\mu + \frac{\alpha^2\sigma^2}{2}\right) = 1.$$

Solving for α gives $\alpha = -2\mu/\sigma^2$.

Exercise 43.5.4

One unrealistic aspect of the firm dynamics specified in (43.9) is that it ignores entry and exit.

In any given period and in any given market, we observe significant numbers of firms entering and exiting the market.

Empirical discussion of this can be found in a famous paper by Hugo Hopenhayn [Hopenhayn, 1992].

In the same paper, Hopenhayn builds a model of entry and exit that incorporates profit maximization by firms and market clearing quantities, wages and prices.

In his model, a stationary equilibrium occurs when the number of entrants equals the number of exiting firms.

In this setting, firm dynamics can be expressed as

$$s_{t+1} = e_{t+1} \mathbb{1}\{s_t < \bar{s}\} + (a_{t+1} s_t + b_{t+1}) \mathbb{1}\{s_t \geq \bar{s}\} \tag{43.11}$$

Here

- the state variable s_t represents productivity (which is a proxy for output and hence firm size),
- the IID sequence $\{e_t\}$ is thought of as a productivity draw for a new entrant and
- the variable $\bar{s}$ is a threshold value that we take as given, although it is determined endogenously in Hopenhayn's model.

The idea behind (43.11) is that firms stay in the market as long as their productivity s_t remains at or above $\bar{s}$.

- In this case, their productivity updates according to (43.9).

Firms choose to exit when their productivity s_t falls below $\bar{s}$.

- In this case, they are replaced by a new firm with productivity e_{t+1}.

What can we say about dynamics?

Although (43.11) is not a Kesten process, it does update in the same way as a Kesten process when s_t is large.

So perhaps its stationary distribution still has Pareto tails?

Your task is to investigate this question via simulation and rank-size plots.

The approach will be to

1. generate M draws of s_T when M and T are large and
2. plot the largest 1,000 of the resulting draws in a rank-size plot.

(The distribution of s_T will be close to the stationary distribution when T is large.)

In the simulation, assume that

- each of a_t, b_t and e_t is lognormal,
- the parameters are

```
μ_a = -0.5        # location parameter for a
σ_a = 0.1         # scale parameter for a
μ_b = 0.0         # location parameter for b
σ_b = 0.5         # scale parameter for b
μ_e = 0.0         # location parameter for e
σ_e = 0.5         # scale parameter for e
s_bar = 1.0       # threshold
T = 500           # sampling date
M = 1_000_000     # number of firms
s_init = 1.0      # initial condition for each firm
```

Solution

Here's one solution.

We need to simulate a large cross-section of firms, so this is a good use case for JAX.

We store the firm parameters in a `NamedTuple` and write the update rule (43.11) for a single firm as a pure function.

```
import jax
import jax.numpy as jnp
from functools import partial
from typing import NamedTuple


class Firm(NamedTuple):
    μ_a: float
    σ_a: float
    μ_b: float
    σ_b: float
    μ_e: float
    σ_e: float
    s_bar: float


def update(s, key, firm):
    "Update a single firm's size s, given a fresh random key."
    a_key, b_key, e_key = jax.random.split(key, 3)
    a = jnp.exp(firm.μ_a + firm.σ_a * jax.random.normal(a_key))
    b = jnp.exp(firm.μ_b + firm.σ_b * jax.random.normal(b_key))
    e = jnp.exp(firm.μ_e + firm.σ_e * jax.random.normal(e_key))
    return jnp.where(s < firm.s_bar, e, a * s + b)
```

We simulate one firm for `T` periods with `lax.scan`, then use `vmap` to run all `M` firms in parallel.

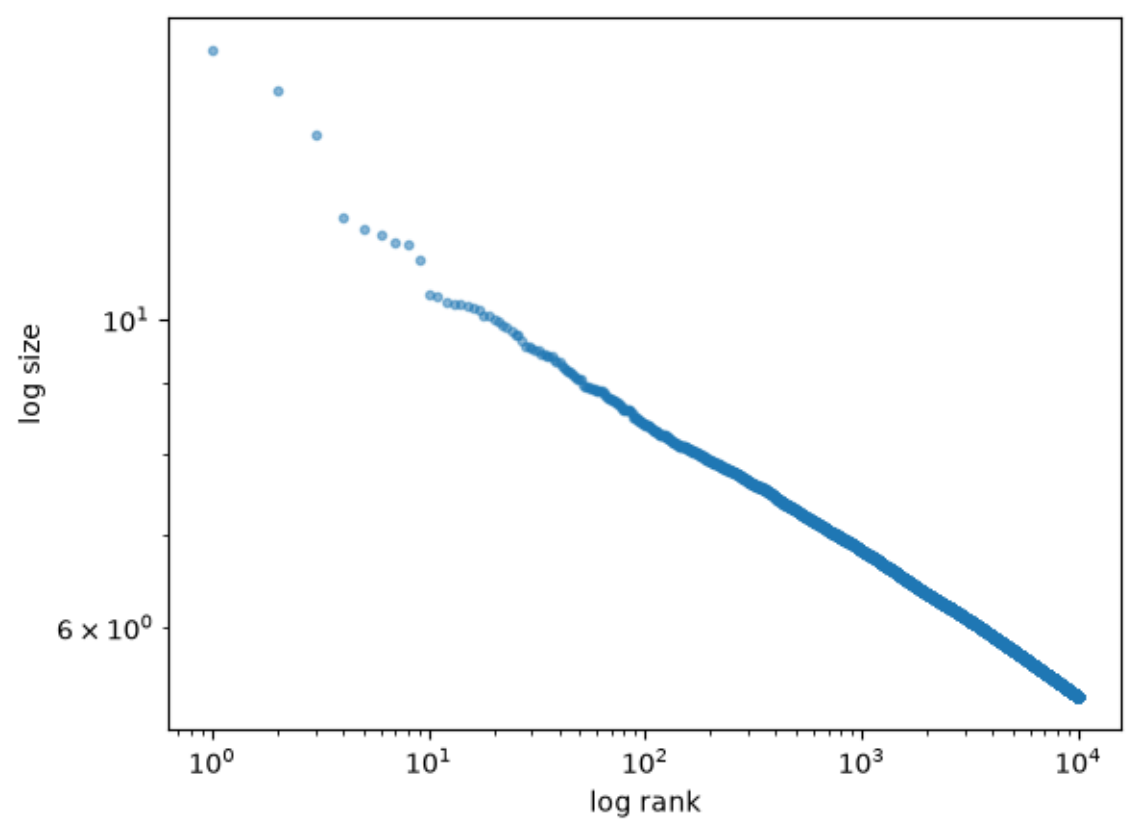
```
@partial(jax.jit, static_argnames=("T", "M"))
def generate_draws(firm, s_init=1.0, T=500, M=1_000_000, seed=0):
    keys = jax.random.split(jax.random.PRNGKey(seed), M)

    def sim_firm(key):
        step_keys = jax.random.split(key, T)
        s_final, _ = jax.lax.scan(
            lambda s, k: (update(s, k, firm), None), s_init, step_keys
        )
        return s_final

    return jax.vmap(sim_firm)(keys)
```

Now we build the firm from the parameters above and generate the draws.

```
firm = Firm(μ_a, σ_a, μ_b, σ_b, μ_e, σ_e, s_bar)
data = generate_draws(firm, s_init=s_init, T=T, M=M)
```

Now we produce the rank-size plot:

```
fig, ax = plt.subplots()

rank_data, size_data = qe.rank_size(data, c=0.01)
ax.loglog(rank_data, size_data, "o", markersize=3.0, alpha=0.5)
ax.set_xlabel("log rank")
ax.set_ylabel("log size")

plt.show()
```



The plot produces a straight line, consistent with a Pareto tail.

CHAPTER

FORTYFOUR

WEALTH DISTRIBUTION DYNAMICS

Contents

- *Wealth Distribution Dynamics*
  - *Overview*
  - *Lorenz Curves and the Gini Coefficient*
  - *A Model of Wealth Dynamics*
  - *Implementation*
  - *Applications*
  - *Exercises*

See also

A version of this lecture using JAX is available here

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install quantecon
```

44.1 Overview

This notebook gives an introduction to wealth distribution dynamics, with a focus on

- modeling and computing the wealth distribution via simulation,
- measures of inequality such as the Lorenz curve and Gini coefficient, and
- how inequality is affected by the properties of wage income and returns on assets.

One interesting property of the wealth distribution we discuss is Pareto tails.

The wealth distribution in many countries exhibits a Pareto tail

- See this lecture for a definition.
- For a review of the empirical evidence, see, for example, [Benhabib and Bisin, 2018].

This is consistent with high concentration of wealth amongst the richest households.

It also gives us a way to quantify such concentration, in terms of the tail index.

One question of interest is whether or not we can replicate Pareto tails from a relatively simple model.

44.1.1 A Note on Assumptions

The evolution of wealth for any given household depends on their savings behavior.

Modeling such behavior will form an important part of this lecture series.

However, in this particular lecture, we will be content with rather ad hoc (but plausible) savings rules.

We do this to more easily explore the implications of different specifications of income dynamics and investment returns.

At the same time, all of the techniques discussed here can be plugged into models that use optimization to obtain savings rules.

We will use the following imports.

```
import matplotlib.pyplot as plt
import numpy as np
import quantecon as qe
from numba import jit, float64, prange
from numba.experimental import jitclass
```

44.2 Lorenz Curves and the Gini Coefficient

Before we investigate wealth dynamics, we briefly review some measures of inequality.

44.2.1 Lorenz Curves

One popular graphical measure of inequality is the Lorenz curve.

The package QuantEcon.py, already imported above, contains a function to compute Lorenz curves.

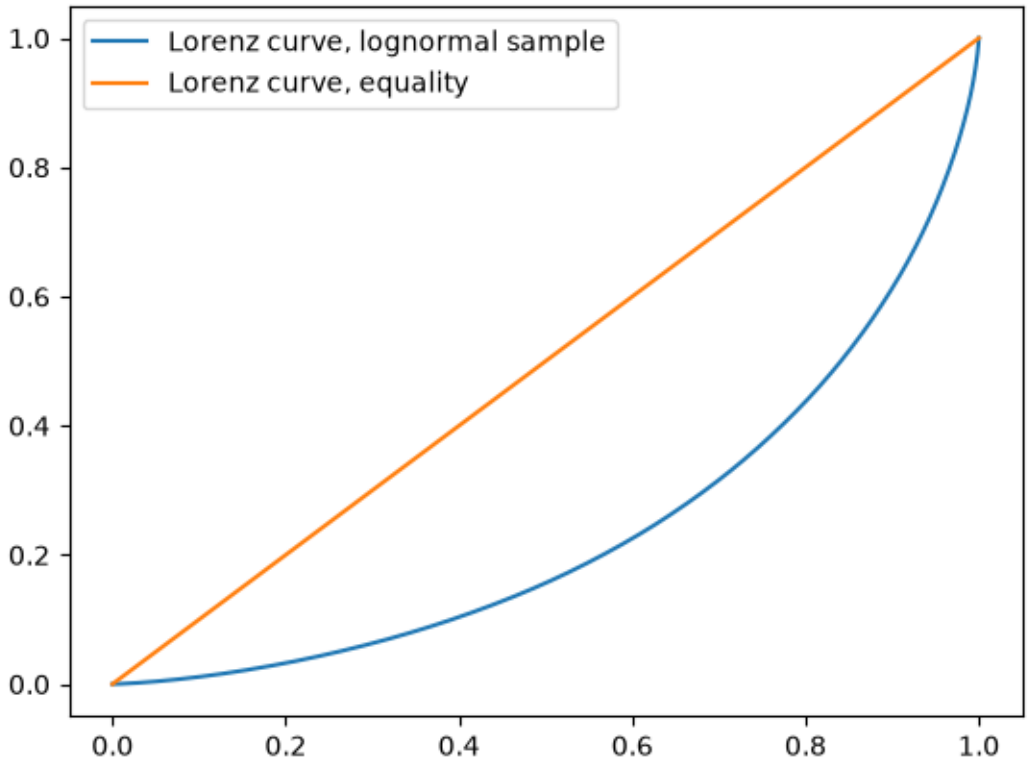
To illustrate, suppose that

```
rng = np.random.default_rng()
n = 10_000                        # size of sample
w = np.exp(rng.standard_normal(n))   # lognormal draws
```

is data representing the wealth of 10,000 households.

We can compute and plot the Lorenz curve as follows:

```
f_vals, l_vals = qe.lorenz_curve(w)

fig, ax = plt.subplots()
ax.plot(f_vals, l_vals, label='Lorenz curve, lognormal sample')
ax.plot(f_vals, f_vals, label='Lorenz curve, equality')
ax.legend()
plt.show()
```



This curve can be understood as follows: if point (x, y) lies on the curve, it means that, collectively, the bottom $(100x)\%$ of the population holds $(100y)\%$ of the wealth.

The "equality" line is the 45 degree line (which might not be exactly 45 degrees in the figure, depending on the aspect ratio).

A sample that produces this line exhibits perfect equality.

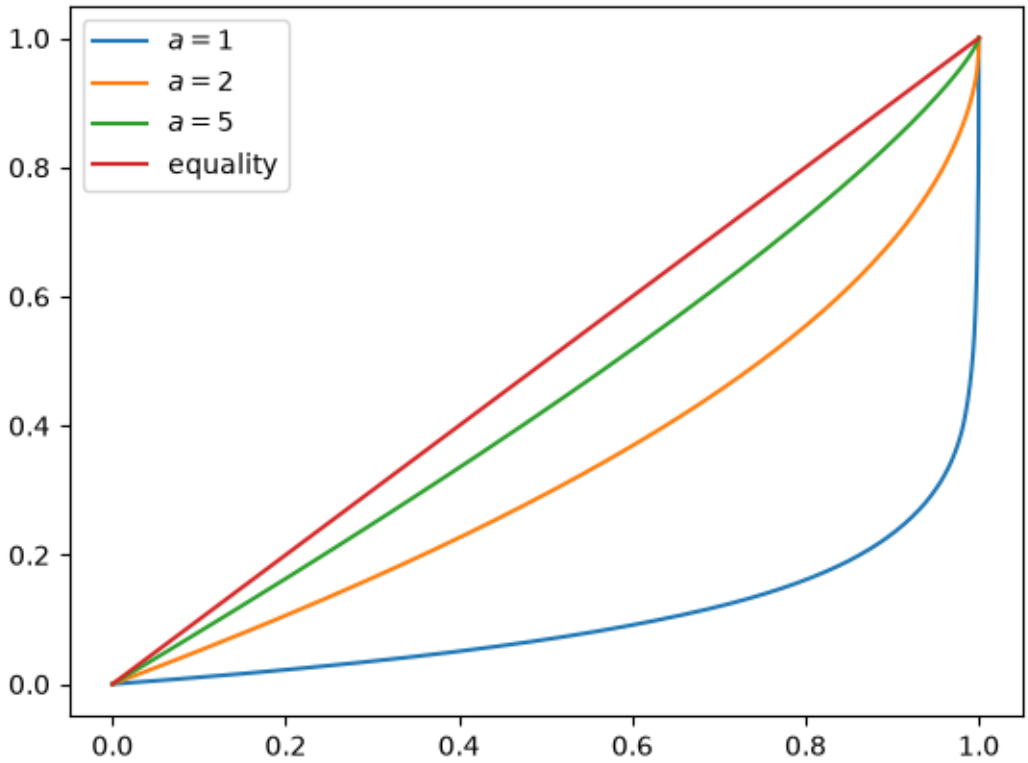
The other line in the figure is the Lorenz curve for the lognormal sample, which deviates significantly from perfect equality.

For example, the bottom 80% of the population holds around 40% of total wealth.

Here is another example, which shows how the Lorenz curve shifts as the underlying distribution changes.

We generate 10,000 observations using the Pareto distribution with a range of parameters, and then compute the Lorenz curve corresponding to each set of observations.

```
a_vals = (1, 2, 5)              # Pareto tail index
n = 10_000                      # size of each sample
fig, ax = plt.subplots()
for a in a_vals:
    u = rng.uniform(size=n)
    y = u**(-1/a)               # distributed as Pareto with tail index a
    f_vals, l_vals = qe.lorenz_curve(y)
    ax.plot(f_vals, l_vals, label=f'$a = {a}$')
ax.plot(f_vals, f_vals, label='equality')
ax.legend()
plt.show()
```



You can see that, as the tail parameter of the Pareto distribution increases, inequality decreases.

This is to be expected, because a higher tail index implies less weight in the tail of the Pareto distribution.

44.2.2 The Gini Coefficient

The definition and interpretation of the Gini coefficient can be found on the corresponding Wikipedia page.

A value of 0 indicates perfect equality (corresponding the case where the Lorenz curve matches the 45 degree line) and a value of 1 indicates complete inequality (all wealth held by the richest household).

The QuantEcon.py library contains a function to calculate the Gini coefficient.

We can test it on the Weibull distribution with parameter a, where the Gini coefficient is known to be

$$G = 1 - 2^{-1/a}$$

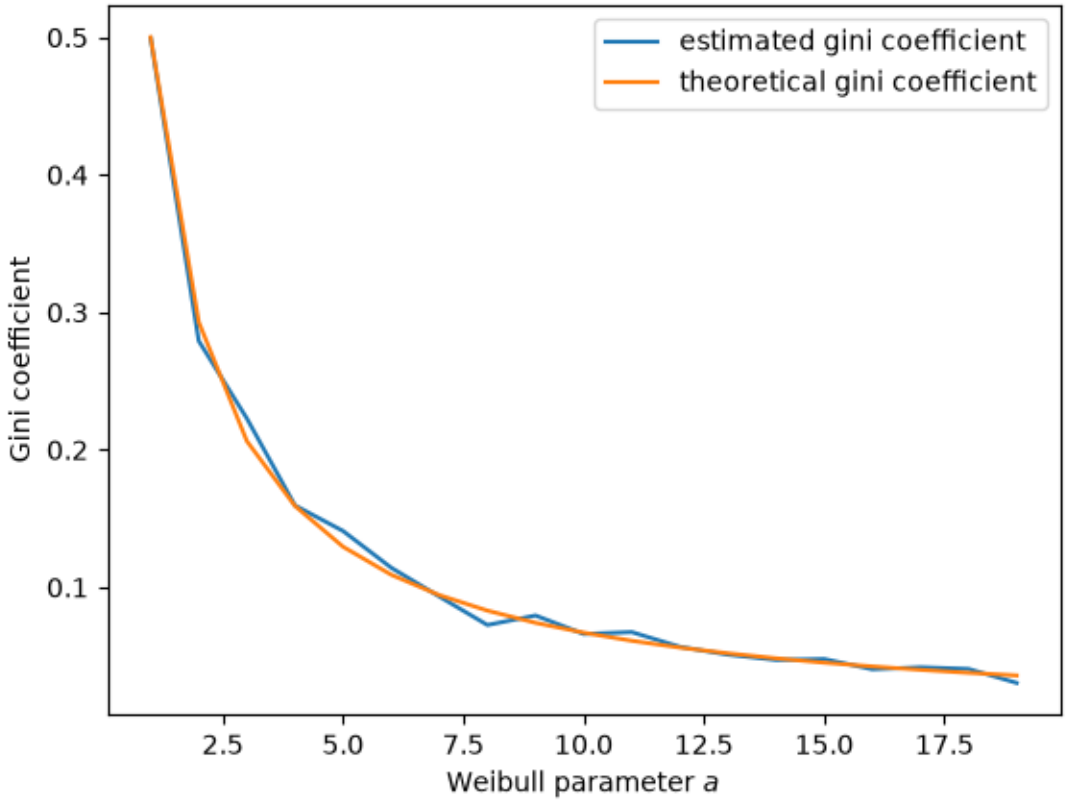
Let's see if the Gini coefficient computed from a simulated sample matches this at each fixed value of a.

```
a_vals = range(1, 20)
ginis = []
ginis_theoretical = []
n = 100

fig, ax = plt.subplots()
for a in a_vals:
    y = rng.weibull(a, size=n)
    ginis.append(qe.gini_coefficient(y))
    ginis_theoretical.append(1 - 2**(-1/a))
ax.plot(a_vals, ginis, label='estimated gini coefficient')
```

(continues on next page)

(continued from previous page)

```
ax.plot(a_vals, ginis_theoretical, label='theoretical gini coefficient')
ax.legend()
ax.set_xlabel("Weibull parameter $a$")
ax.set_ylabel("Gini coefficient")
plt.show()
```



The simulation shows that the fit is good.

44.3 A Model of Wealth Dynamics

Having discussed inequality measures, let us now turn to wealth dynamics.

The model we will study is

$$w_{t+1} = (1 + r_{t+1})s(w_t) + y_{t+1} \tag{44.1}$$

where

- w_t is wealth at time t for a given household,
- r_t is the rate of return of financial assets,
- y_t is current non-financial (e.g., labor) income and
- $s(w_t)$ is current wealth net of consumption

Letting $\{z_t\}$ be a correlated state process of the form

$$z_{t+1} = az_t + b + \sigma_z \epsilon_{t+1}$$

we'll assume that

$$R_t := 1 + r_t = c_r \exp(z_t) + \exp(\mu_r + \sigma_r \xi_t)$$

and

$$y_t = c_y \exp(z_t) + \exp(\mu_y + \sigma_y \zeta_t)$$

Here $\{(\epsilon_t, \xi_t, \zeta_t)\}$ is IID and standard normal in $\mathbb{R}^3$.

The value of c_r should be close to zero, since rates of return on assets do not exhibit large trends.

When we simulate a population of households, we will assume all shocks are idiosyncratic (i.e., specific to individual households and independent across them).

Regarding the savings function s, our default model will be

$$s(w) = s_0 w \cdot \mathbb{1}\{w \geq \hat{w}\} \tag{44.2}$$

where s_0 is a positive constant.

Thus, for $w < \hat{w}$, the household saves nothing. For $w \geq \bar{w}$, the household saves a fraction s_0 of their wealth.

We are using something akin to a fixed savings rate model, while acknowledging that low wealth households tend to save very little.

44.4 Implementation

Here's some type information to help Numba.

```
wealth_dynamics_data = [
    ('w_hat',  float64),    # savings parameter
    ('s_0',    float64),    # savings parameter
    ('c_y',    float64),    # labor income parameter
    ('μ_y',    float64),    # labor income paraemter
    ('σ_y',    float64),    # labor income parameter
    ('c_r',    float64),    # rate of return parameter
    ('μ_r',    float64),    # rate of return parameter
    ('σ_r',    float64),    # rate of return parameter
    ('a',      float64),    # aggregate shock parameter
    ('b',      float64),    # aggregate shock parameter
    ('σ_z',    float64),    # aggregate shock parameter
    ('z_mean', float64),    # mean of z process
    ('z_var', float64),     # variance of z process
    ('y_mean', float64),    # mean of y process
    ('R_mean', float64)     # mean of R process
]
```

Here's a class that stores instance data and implements methods that update the aggregate state and household wealth.

```
@jitclass(wealth_dynamics_data)
class WealthDynamics:

    def __init__(self,
                 w_hat=1.0,
                 s_0=0.75,
                 c_y=1.0,
```

(continues on next page)

(continued from previous page)

```
                μ_y=1.0,
                σ_y=0.2,
                c_r=0.05,
                μ_r=0.1,
                σ_r=0.5,
                a=0.5,
                b=0.0,
                σ_z=0.1):

        self.w_hat, self.s_0 = w_hat, s_0
        self.c_y, self.μ_y, self.σ_y = c_y, μ_y, σ_y
        self.c_r, self.μ_r, self.σ_r = c_r, μ_r, σ_r
        self.a, self.b, self.σ_z = a, b, σ_z

        # Record stationary moments
        self.z_mean = b / (1 - a)
        self.z_var = σ_z**2 / (1 - a**2)
        exp_z_mean = np.exp(self.z_mean + self.z_var / 2)
        self.R_mean = c_r * exp_z_mean + np.exp(μ_r + σ_r**2 / 2)
        self.y_mean = c_y * exp_z_mean + np.exp(μ_y + σ_y**2 / 2)

        # Test a stability condition that ensures wealth does not diverge
        # to infinity.
        α = self.R_mean * self.s_0
        if α >= 1:
            raise ValueError("Stability condition failed.")

    def parameters(self):
        """
        Collect and return parameters.
        """
        parameters = (self.w_hat, self.s_0,
                      self.c_y, self.μ_y, self.σ_y,
                      self.c_r, self.μ_r, self.σ_r,
                      self.a, self.b, self.σ_z)
        return parameters

    def update_states(self, w, z):
        """
        Update one period, given current wealth w and persistent
        state z.
        """

        # Simplify names
        params = self.parameters()
        w_hat, s_0, c_y, μ_y, σ_y, c_r, μ_r, σ_r, a, b, σ_z = params
        zp = a * z + b + σ_z * np.random.randn()

        # Update wealth
        y = c_y * np.exp(zp) + np.exp(μ_y + σ_y * np.random.randn())
        wp = y
        if w >= w_hat:
            R = c_r * np.exp(zp) + np.exp(μ_r + σ_r * np.random.randn())
            wp += R * s_0 * w
        return wp, zp
```

Here's function to simulate the time series of wealth for in individual households.

```
@jit
def wealth_time_series(wdy, w_0, n):
    """
    Generate a single time series of length n for wealth given
    initial value w_0.

    The initial persistent state z_0 for each household is drawn from
    the stationary distribution of the AR(1) process.

        * wdy: an instance of WealthDynamics
        * w_0: scalar
        * n: int


    """
    z = wdy.z_mean + np.sqrt(wdy.z_var) * np.random.randn()
    w = np.empty(n)
    w[0] = w_0
    for t in range(n-1):
        w[t+1], z = wdy.update_states(w[t], z)
    return w
```

Now here's function to simulate a cross section of households forward in time.

Note the use of parallelization to speed up computation.

```
@jit(parallel=True)
def update_cross_section(wdy, w_distribution, shift_length=500):
    """
    Shifts a cross-section of household forward in time

    * wdy: an instance of WealthDynamics
    * w_distribution: array_like, represents current cross-section

    Takes a current distribution of wealth values as w_distribution
    and updates each w_t in w_distribution to w_{t+j}, where
    j = shift_length.

    Returns the new distribution.

    """
    new_distribution = np.empty_like(w_distribution)

    # Update each household
    for i in prange(len(new_distribution)):
        z = wdy.z_mean + np.sqrt(wdy.z_var) * np.random.randn()
        w = w_distribution[i]
        for t in range(shift_length-1):
            w, z = wdy.update_states(w, z)
        new_distribution[i] = w
    return new_distribution
```

Parallelization is very effective in the function above because the time path of each household can be calculated independently once the path for the aggregate state is known.

44.5 Applications

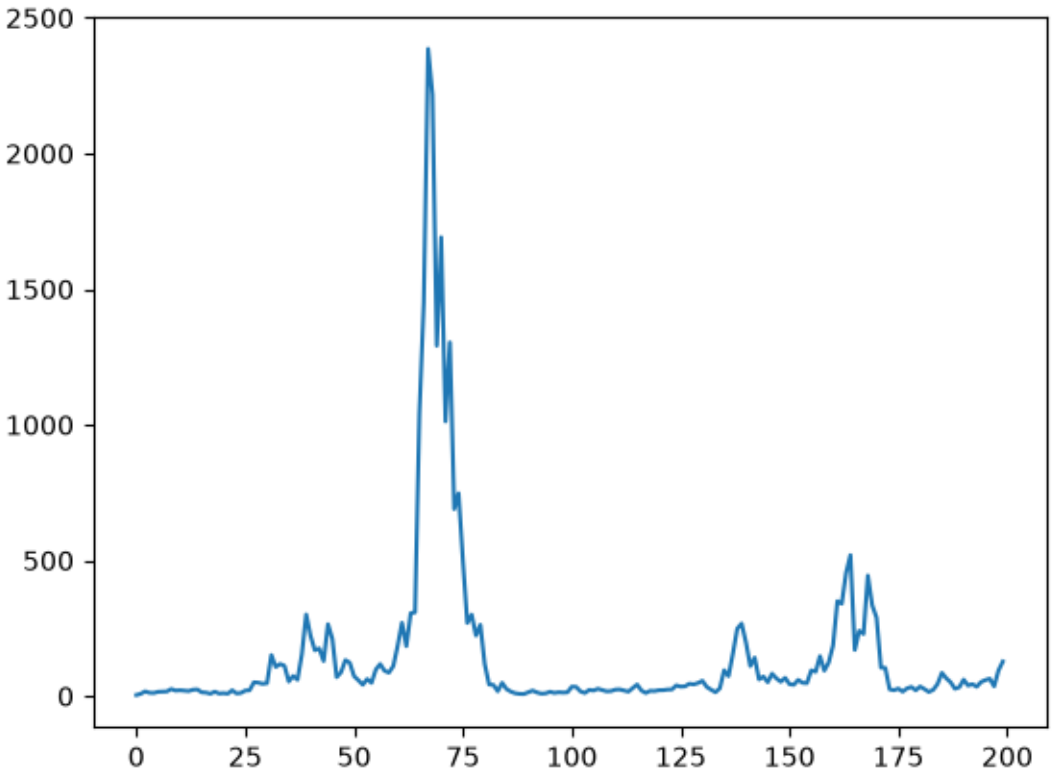
Let's try simulating the model at different parameter values and investigate the implications for the wealth distribution.

44.5.1 Time Series

Let's look at the wealth dynamics of an individual household.

```
wdy = WealthDynamics()
ts_length = 200
w = wealth_time_series(wdy, wdy.y_mean, ts_length)
```

```
fig, ax = plt.subplots()
ax.plot(w)
plt.show()
```



Notice the large spikes in wealth over time.

Such spikes are similar to what we observed in time series when *we studied Kesten processes*.

44.5.2 Inequality Measures

Let's look at how inequality varies with returns on financial assets.

The next function generates a cross section and then computes the Lorenz curve and Gini coefficient.

```
def generate_lorenz_and_gini(wdy, num_households=100_000, T=500):
    """
    Generate the Lorenz curve data and gini coefficient corresponding to a
    WealthDynamics mode by simulating num_households forward to time T.
    """
    ψ_0 = np.full(num_households, wdy.y_mean)
    z_0 = wdy.z_mean

    ψ_star = update_cross_section(wdy, ψ_0, shift_length=T)
    return qe.gini_coefficient(ψ_star), qe.lorenz_curve(ψ_star)
```

Now we investigate how the Lorenz curves associated with the wealth distribution change as return to savings varies.

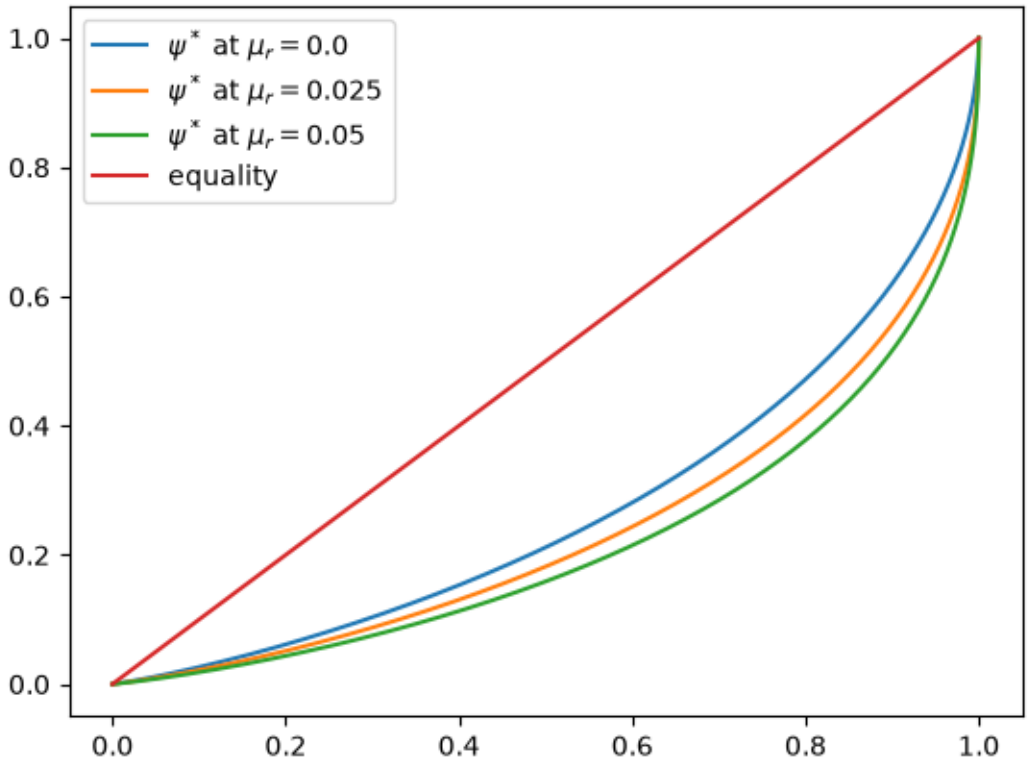
The code below plots Lorenz curves for three different values of μ_r.

If you are running this yourself, note that it will take one or two minutes to execute.

This is unavoidable because we are executing a CPU intensive task.

In fact the code, which is JIT compiled and parallelized, runs extremely fast relative to the number of computations.

```
%%time

fig, ax = plt.subplots()
μ_r_vals = (0.0, 0.025, 0.05)
gini_vals = []

for μ_r in μ_r_vals:
    wdy = WealthDynamics(μ_r=μ_r)
    gv, (f_vals, l_vals) = generate_lorenz_and_gini(wdy)
    ax.plot(f_vals, l_vals, label=fr'$\psi^*$ at $\mu_r = {μ_r:0.2}$')
    gini_vals.append(gv)

ax.plot(f_vals, f_vals, label='equality')
ax.legend(loc="upper left")
plt.show()
```




```
CPU times: user 1min 16s, sys: 56.8 ms, total: 1min 16s
Wall time: 10.3 s
```

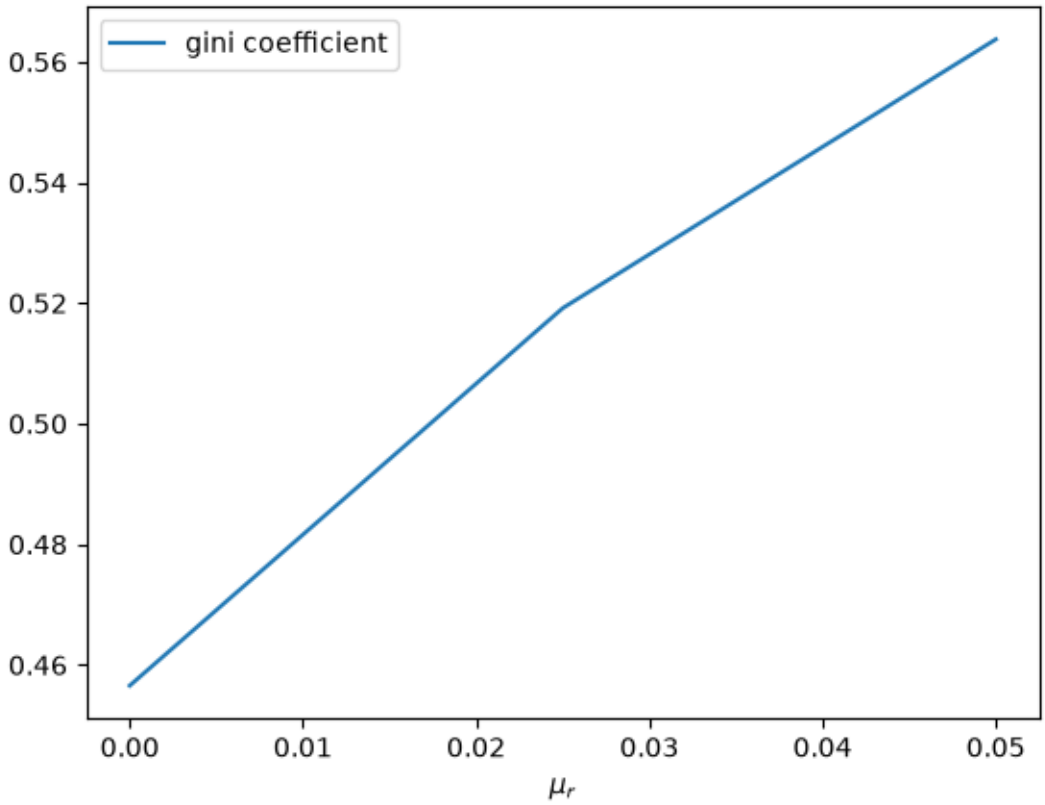
The Lorenz curve shifts downwards as returns on financial income rise, indicating a rise in inequality.

We will look at this again via the Gini coefficient immediately below, but first consider the following image of our system resources when the code above is executing:

Since the code is both efficiently JIT compiled and fully parallelized, it's close to impossible to make this sequence of tasks run faster without changing hardware.

Now let's check the Gini coefficient.

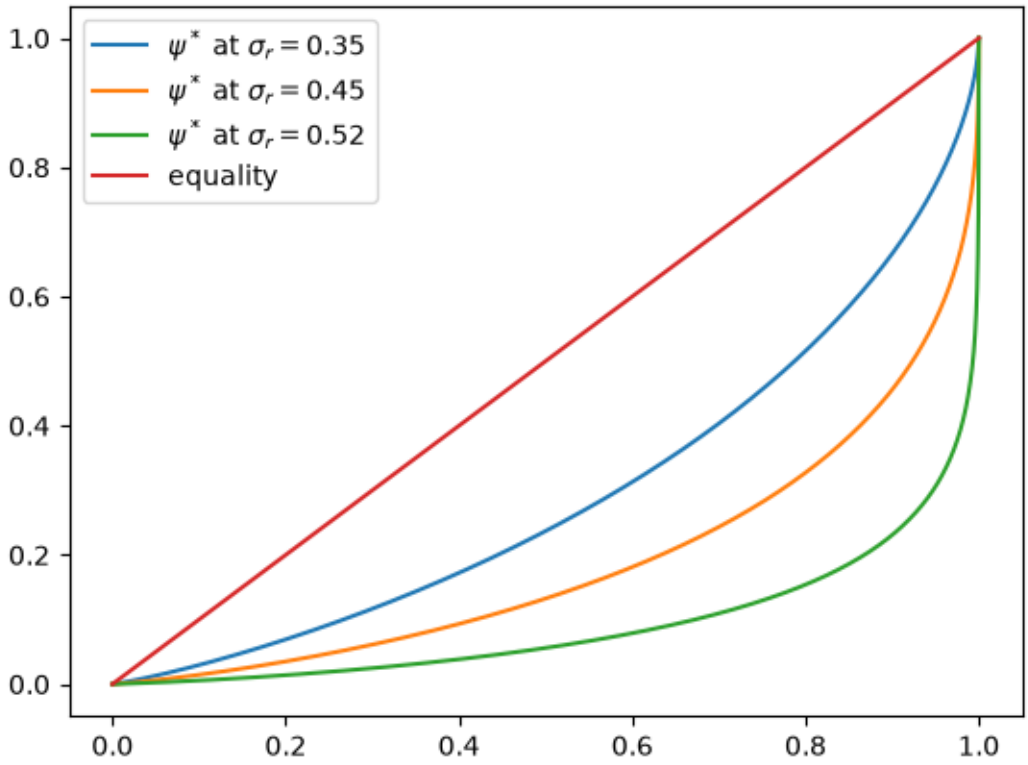
```
fig, ax = plt.subplots()
ax.plot(μ_r_vals, gini_vals, label='gini coefficient')
ax.set_xlabel(r"$\mu_r$")
ax.legend()
plt.show()
```



Once again, we see that inequality increases as returns on financial income rise.

Let's finish this section by investigating what happens when we change the volatility term σ_r in financial returns.

```
%%time

fig, ax = plt.subplots()
σ_r_vals = (0.35, 0.45, 0.52)
gini_vals = []

for σ_r in σ_r_vals:
    wdy = WealthDynamics(σ_r=σ_r)
    gv, (f_vals, l_vals) = generate_lorenz_and_gini(wdy)
    ax.plot(f_vals, l_vals, label=fr'$\psi^*$ at $\sigma_r = {σ_r:0.2}$')
    gini_vals.append(gv)

ax.plot(f_vals, f_vals, label='equality')
ax.legend(loc="upper left")
plt.show()
```




```
CPU times: user 1min 15s, sys: 42 ms, total: 1min 15s
Wall time: 9.65 s
```

We see that greater volatility has the effect of increasing inequality in this model.

44.6 Exercises

Exercise 44.6.1

For a wealth or income distribution with Pareto tail, a higher tail index suggests lower inequality.

Indeed, it is possible to prove that the Gini coefficient of the Pareto distribution with tail index a is $1/(2a-1)$.

To the extent that you can, confirm this by simulation.

In particular, generate a plot of the Gini coefficient against the tail index using both the theoretical value just given and the value computed from a sample via `qe.gini_coefficient`.

For the values of the tail index, use `a_vals = np.linspace(1, 10, 25)`.

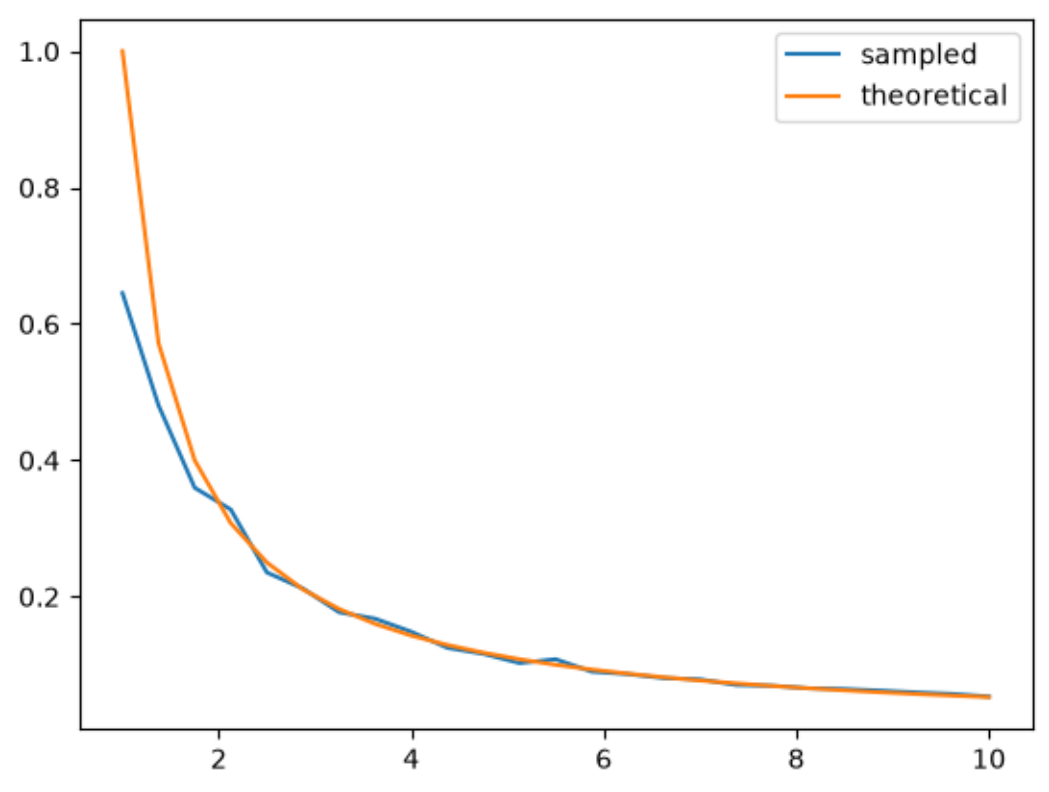
Use sample of size 1,000 for each a and the sampling method for generating Pareto draws employed in the discussion of Lorenz curves for the Pareto distribution.

To the extent that you can, interpret the monotone relationship between the Gini index and a.

Solution

Here is one solution, which produces a good match between theory and simulation.

```
rng = np.random.default_rng()
a_vals = np.linspace(1, 10, 25)  # Pareto tail index
ginis = np.empty_like(a_vals)

n = 1000                         # size of each sample
fig, ax = plt.subplots()
for i, a in enumerate(a_vals):
    y = rng.uniform(size=n)**(-1/a)
    ginis[i] = qe.gini_coefficient(y)
ax.plot(a_vals, ginis, label='sampled')
ax.plot(a_vals, 1/(2*a_vals - 1), label='theoretical')
ax.legend()
plt.show()
```



In general, for a Pareto distribution, a higher tail index implies less weight in the right hand tail.

This means less extreme values for wealth and hence more equality.

More equality translates to a lower Gini index.

Exercise 44.6.2

The wealth process (44.1) is similar to a *Kesten process*.

This is because, according to (44.2), savings is constant for all wealth levels above $\hat{w}$.

When savings is constant, the wealth process has the same quasi-linear structure as a Kesten process, with multiplicative and additive shocks.

The Kesten–Goldie theorem tells us that Kesten processes have Pareto tails under a range of parameterizations.

The theorem does not directly apply here, since savings is not always constant and since the multiplicative and additive terms in (44.1) are not IID.

At the same time, given the similarities, perhaps Pareto tails will arise.

To test this, run a simulation that generates a cross-section of wealth and generate a rank-size plot.

If you like, you can use the function `rank_size` from the `quantecon` library (documentation here).

In viewing the plot, remember that Pareto tails generate a straight line. Is this what you see?

For sample size and initial conditions, use

```
num_households = 250_000
T = 500                                          # shift forward T periods
ψ_0 = np.full(num_households, wdy.y_mean)    # initial distribution
z_0 = wdy.z_mean
```

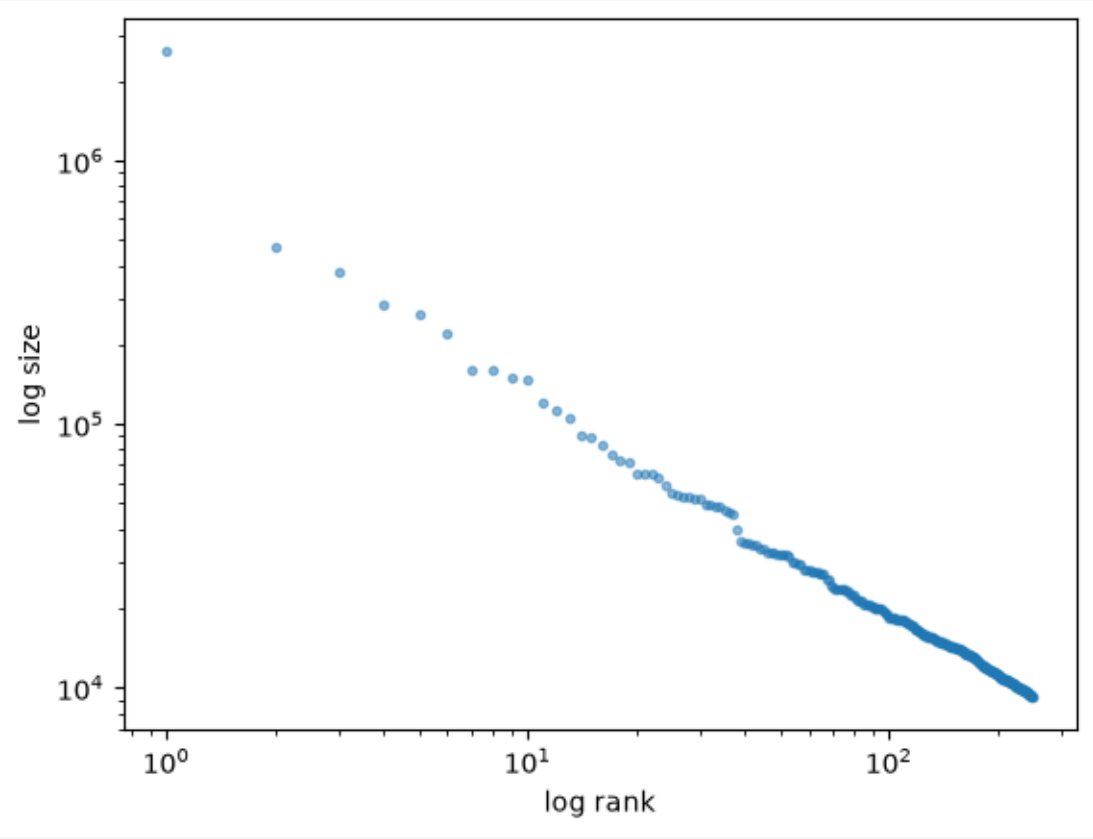
Solution

First let's generate the distribution:

```
num_households = 250_000
T = 500  # how far to shift forward in time
wdy = WealthDynamics()
ψ_0 = np.full(num_households, wdy.y_mean)
z_0 = wdy.z_mean

ψ_star = update_cross_section(wdy, ψ_0, shift_length=T)
```

Now let's see the rank-size plot:

```
fig, ax = plt.subplots()

rank_data, size_data = qe.rank_size(ψ_star, c=0.001)
ax.loglog(rank_data, size_data, 'o', markersize=3.0, alpha=0.5)
ax.set_xlabel("log rank")
ax.set_ylabel("log size")

plt.show()
```


10^6
10^5
10^4
log size
10^0
10^1
10^2
log rank

CHAPTER

FORTYFIVE

A FIRST LOOK AT THE KALMAN FILTER

Contents

- *A First Look at the Kalman Filter*
  - *Overview*
  - *The basic idea*
  - *Convergence*
  - *Implementation*
  - *Exercises*

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install quantecon
```

45.1 Overview

This lecture provides a simple and intuitive introduction to the Kalman filter

It is aimed at readers who either

- have heard of the Kalman filter but don't know how it works, or
- know the Kalman filter equations, but don't know where they come from

Subsequent lectures use the same recursive logic in more applied and more econometric settings.

See *Another Look at the Kalman Filter* for an economic application in which a firm infers a worker's hidden human capital and effort.

See *The Kalman Filter and Vector Autoregressions* for a derivation of the innovations representation and its connection to vector autoregressions.

For additional (more advanced) reading on the Kalman filter, see

- [Ljungqvist and Sargent, 2018], section 2.7
- [Anderson and Moore, 2005]

The second reference presents a comprehensive treatment of the Kalman filter.

Required knowledge: Familiarity with matrix manipulations, multivariate normal distributions, covariance matrices, etc.

We'll need the following imports:

```
import matplotlib.pyplot as plt
from scipy import linalg
import numpy as np
from quantecon import Kalman, LinearStateSpace
from scipy.stats import norm, multivariate_normal
from scipy.integrate import quad
from scipy.linalg import eigvals
```

45.2 The basic idea

The Kalman filter has many applications in economics, but for now let's pretend that we are rocket scientists.

A missile has been launched from a hostile country and our mission is to track it.

Let $X_t \in \mathbb{R}^2$ denote the current location of the missile—a pair indicating latitude-longitude coordinates on a map.

At the present moment in time, the location X_t is unknown, but we do have some beliefs about it.

We could certainly produce a point prediction.

For example, it could mark a point on the globe somewhere in northern Mongolia.

But the fact is that we are uncertain.

And the President wants to know: what is the probability that the missile is within 500km of Manhattan?

A point prediction doesn't address that question.

Hence it's best if we can express our current understanding via a bivariate probability density p.

- Now $\int_E p(x)dx$ indicates the probability that the missile is in region E.

We will call p our **prior** for the random variable X.

To keep things tractable, we assume for now that our prior is Gaussian.

In particular, we take

$$p = N(\mu, \Sigma) \tag{45.1}$$

where μ is the (vector) mean of the distribution—a natural point prediction—and Σ is a 2×2 covariance matrix.

In our simulations, we will suppose that

$$\mu = \begin{pmatrix} 0.2 \\ -0.2 \end{pmatrix}, \qquad \Sigma = \begin{pmatrix} 0.4 & 0.3 \\ 0.3 & 0.45 \end{pmatrix} \tag{45.2}$$

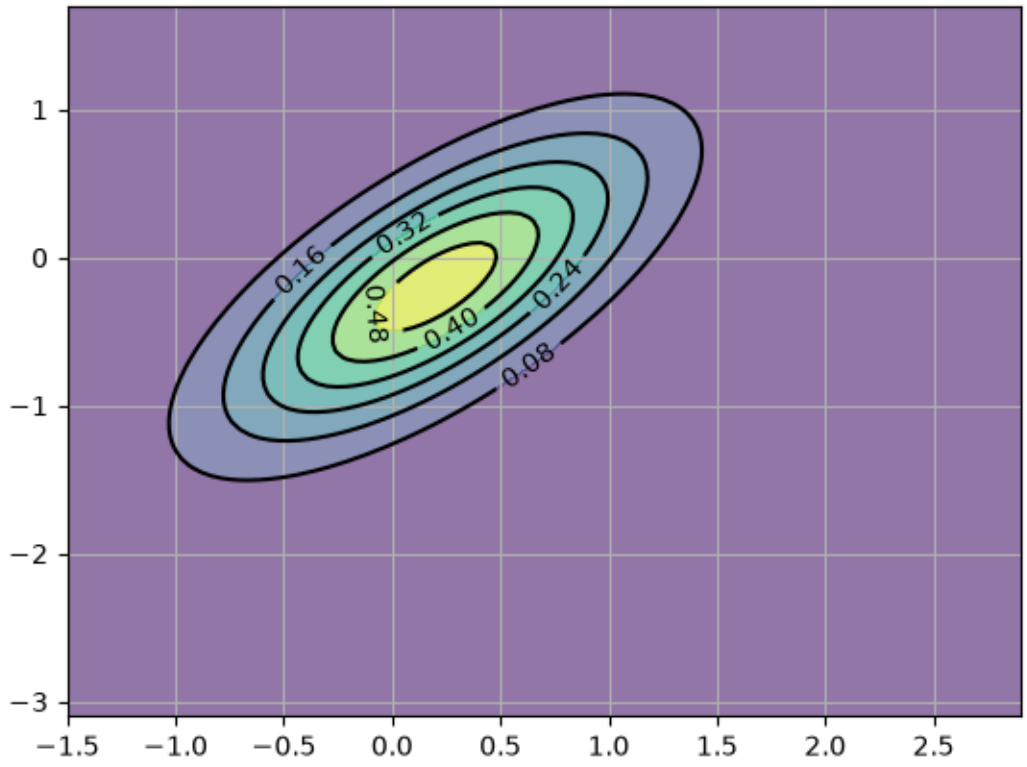
This density p is shown below as a contour map, with the center of the red ellipse being equal to μ.

```
# Set up the Gaussian prior density p
Σ = np.array([[0.4, 0.3],
              [0.3, 0.45]])
μ = np.array([[0.2],
              [-0.2]])
# Define the matrices G and R from the measurement equation Y = G X + v
G = np.array([[1, 0],
              [0, 1]])
R = 0.5 * Σ
```

(continues on next page)

(continued from previous page)

```
# The matrices A and Q
A = np.array([[1.2, 0],
              [0, -0.2]])
Q = 0.3 * Σ
# The observed value of y
y = np.array([[2.3],
              [-1.9]])

# Set up grid for plotting
x_grid = np.linspace(-1.5, 2.9, 100)
y_grid = np.linspace(-3.1, 1.7, 100)
X, Y = np.meshgrid(x_grid, y_grid)

def gen_gaussian_plot_vals(μ, C):
    "Z values for plotting the bivariate Gaussian N(μ, C)"
    pos = np.dstack((X, Y))
    return multivariate_normal(μ.ravel(), C).pdf(pos)

# Plot the figure

fig, ax = plt.subplots()
ax.grid()
Z = gen_gaussian_plot_vals(μ, Σ)
ax.contourf(X, Y, Z, 6, alpha=0.6, cmap="viridis")
cs = ax.contour(X, Y, Z, 6, colors="black")
ax.clabel(cs, inline=1, fontsize=10)
plt.show()
```



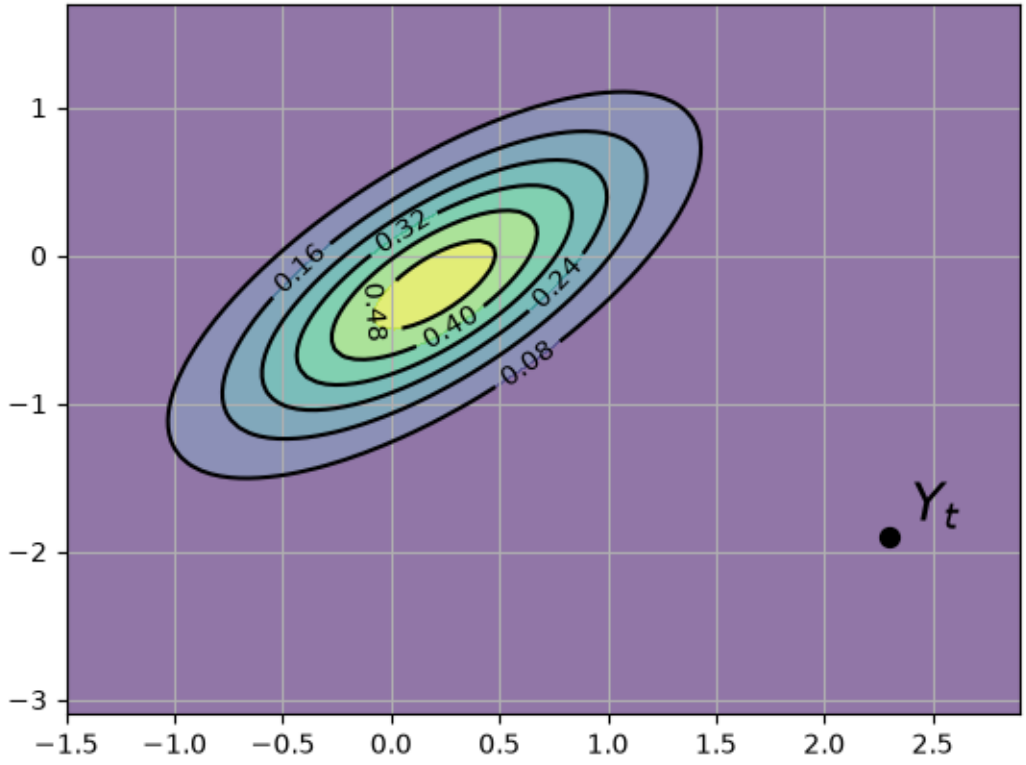
45.2.1 The filtering step

We are now presented with some good news and some bad news.

The good news is that the missile has been located by our sensors, which report that the current location is $Y_t = (2.3, -1.9)$.

The next figure shows the original prior p and the new reported signal Y_t

```
fig, ax = plt.subplots()
ax.grid()
Z = gen_gaussian_plot_vals(μ, Σ)
ax.contourf(X, Y, Z, 6, alpha=0.6, cmap="viridis")
cs = ax.contour(X, Y, Z, 6, colors="black")
ax.clabel(cs, inline=1, fontsize=10)
y_1, y_2 = y[0].item(), y[1].item()
ax.scatter(y_1, y_2, marker="o", s=50, color="black", zorder=3)
ax.text(y_1 + 0.1, y_2 + 0.1, "$Y_t$", fontsize=20, color="black")
plt.show()
```



The bad news is that our sensors are imprecise.

The sensor report is a noisy signal distorted by measurement error.

In particular, we should interpret the output of our sensor not as $Y_t = X_t$, but rather as

$$Y_t = GX_t + v_t, \quad \text{where} \quad v_t \sim N(0, R) \tag{45.3}$$

Here G and R are 2×2 matrices, with R being symmetric and positive definite.

We assume that

- G and R are known

- the noise term v_t is unobservable and independent of X_t

How then should we combine our prior $X_t \sim N(\mu, \Sigma)$ and this new information Y_t to improve our understanding of the location of the missile?

As you may have guessed, the answer is to use Bayes' theorem.

It tells us how to update the prior density $p(x)$ for X_t to the posterior density $p(x \mid y)$ after observing Y_t:

$$p(x \mid Y_t) = \frac{p(Y_t \mid x)\, p(x)}{p(Y_t)}$$

where $p(Y_t) = \int p(Y_t \mid x)\, p(x) dx$.

In solving for $p(x \mid Y_t)$, we observe that

- $p(x)$ is the prior density $N(\mu, \Sigma)$.
- $p(Y_t \mid x)$ is the conditional density of Y_t given $X_t = x$.
- In view of (45.3), this conditional density is $N(Gx, R)$.

Due to our linear Gaussian framework, the updated density turns out to be Gaussian as well.

In particular, the solution is known to be

$$p(x \mid Y_t) = N(\mu^F, \Sigma^F)$$

where

$$\mu^F := \mu + \Sigma G^\top (G \Sigma G^\top + R)^{-1} (y - G\mu) \tag{45.4}$$

and

$$\Sigma^F := \Sigma - \Sigma G^\top (G \Sigma G^\top + R)^{-1} G \Sigma \tag{45.5}$$

Note

A proof can be found in [Bishop, 2006].

To get from his expressions to the ones used above, you will also need to apply the Woodbury matrix identity.

Here $\Sigma G^\top (G \Sigma G^\top + R)^{-1}$ is the matrix of population regression coefficients of the hidden state deviation $X_t - \mu$ on the *signal surprise* $Y_t - G\mu$.

This new density $p(x \mid Y_t) = N(\mu^F, \Sigma^F)$ is shown in the next figure via contour lines and the color map.

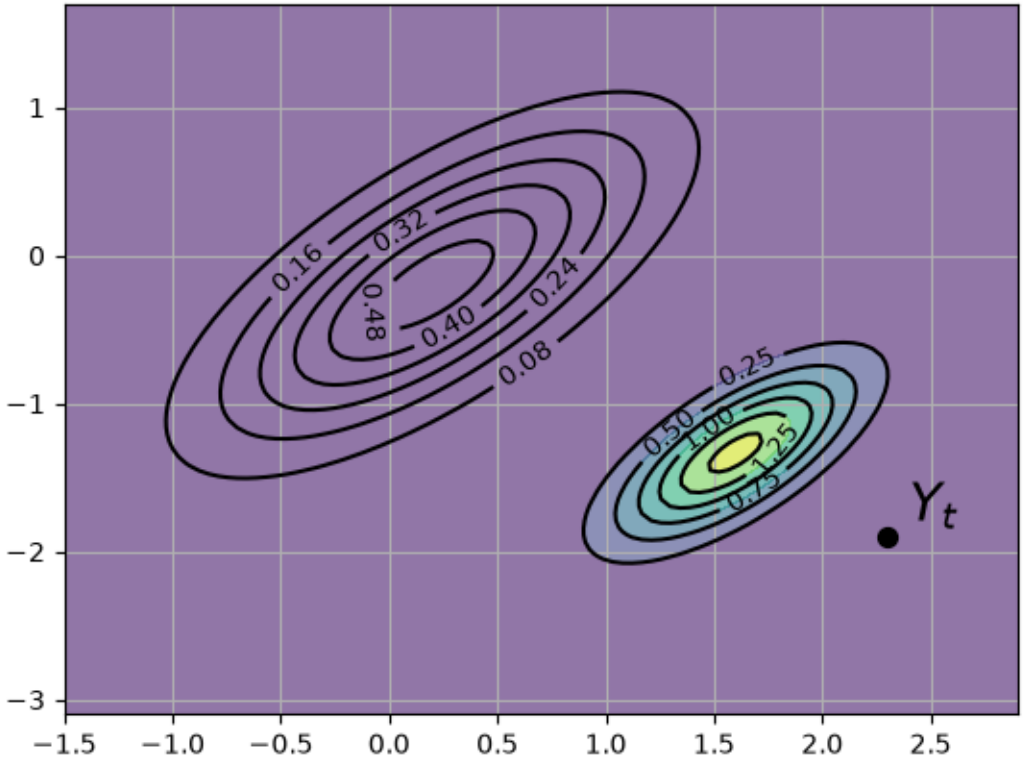
The original density is left in as contour lines for comparison

```
fig, ax = plt.subplots()
ax.grid()

Z = gen_gaussian_plot_vals(μ, Σ)
cs1 = ax.contour(X, Y, Z, 6, colors="black")
ax.clabel(cs1, inline=1, fontsize=10)
M = Σ @ G.T @ linalg.inv(G @ Σ @ G.T + R)
μ_F = μ + M @ (y - G @ μ)
Σ_F = Σ - M @ G @ Σ
new_Z = gen_gaussian_plot_vals(μ_F, Σ_F)
```

(continues on next page)

(continued from previous page)

```
cs2 = ax.contour(X, Y, new_Z, 6, colors="black")
ax.clabel(cs2, inline=1, fontsize=10)
ax.contourf(X, Y, new_Z, 6, alpha=0.6, cmap="viridis")
y_1, y_2 = y[0].item(), y[1].item()
ax.scatter(y_1, y_2, marker="o", s=50, color="black", zorder=3)
ax.text(y_1 + 0.1, y_2 + 0.1, "$Y_t$", fontsize=20, color="black")
plt.show()
```



Our new density twists the prior $p(x)$ in a direction determined by the new information $Y_t - G\mu$.

In generating the figure, we set G to the identity matrix and $R = 0.5\Sigma$ for Σ defined in (45.2).

45.2.2 The forecast step

What have we achieved so far?

We have obtained probabilities for the current location of the state (missile) given prior and current information.

This is called "filtering" rather than forecasting because we are filtering out noise rather than looking into the future.

The posterior $p(x \mid Y_t) = N(\mu^F, \Sigma^F)$ is called the **filtering distribution** for X_t after observing Y_t

But now let's suppose that we are given another task: to predict the location of the missile after one unit of time (whatever that may be) has elapsed.

To do this we need a model of how the state evolves.

Let's suppose that we have one, and that it's linear and Gaussian.

In particular,

$$X_{t+1} = AX_t + W_{t+1}, \quad \text{where} \quad W_t \sim N(0, Q) \tag{45.6}$$

Our aim is to combine this law of motion and our current filtering distribution $N(\mu^F, \Sigma^F)$ to come up with a new **predictive** distribution for the location in one unit of time.

In view of (45.6), all we have to do is introduce a random vector $X^F \sim N(\mu^F, \Sigma^F)$ and work out the distribution of $AX^F + W$ where W is independent of X^F and has distribution $N(0, Q)$.

Since linear combinations of Gaussians are Gaussian, $AX^F + W$ is Gaussian.

Standard calculations and the expressions in (45.4)–(45.5) tell us that

$$\begin{aligned} \mathbb{E}[AX^F + W] &= A\mathbb{E}[X^F] + \mathbb{E}[W] \\ &= A\mu^F \\ &= A\mu + A\Sigma G^\top (G\Sigma G^\top + R)^{-1}(Y_t - G\mu) \end{aligned}$$

and

$$\begin{aligned} \operatorname{Var}[AX^F + W] &= A \operatorname{Var}[X^F] A^\top + Q \\ &= A\Sigma^F A^\top + Q \\ &= A\Sigma A^\top + Q - A\Sigma G^\top (G\Sigma G^\top + R)^{-1} G\Sigma A^\top \end{aligned}$$

The matrix $A\Sigma G^\top (G\Sigma G^\top + R)^{-1}$ is often written as K_Σ and called the **Kalman gain**.

- The subscript Σ has been added to remind us that K_Σ depends on Σ, but not Y_t or μ.

Using this notation, we can summarize our results as follows.

Our updated prediction is the density $N(\mu_{\text{new}}, \Sigma_{\text{new}})$ where

$$\begin{aligned} \mu_{\text{new}} &:= A\mu + K_\Sigma (y - G\mu) \\ \Sigma_{\text{new}} &:= A\Sigma A^\top - K_\Sigma G\Sigma A^\top + Q \end{aligned}$$

- The density $p_{\text{new}}(x) = N(\mu_{\text{new}}, \Sigma_{\text{new}})$ is called the **predictive distribution**

The predictive distribution is the new density shown in the following figure, where the update has used parameters.

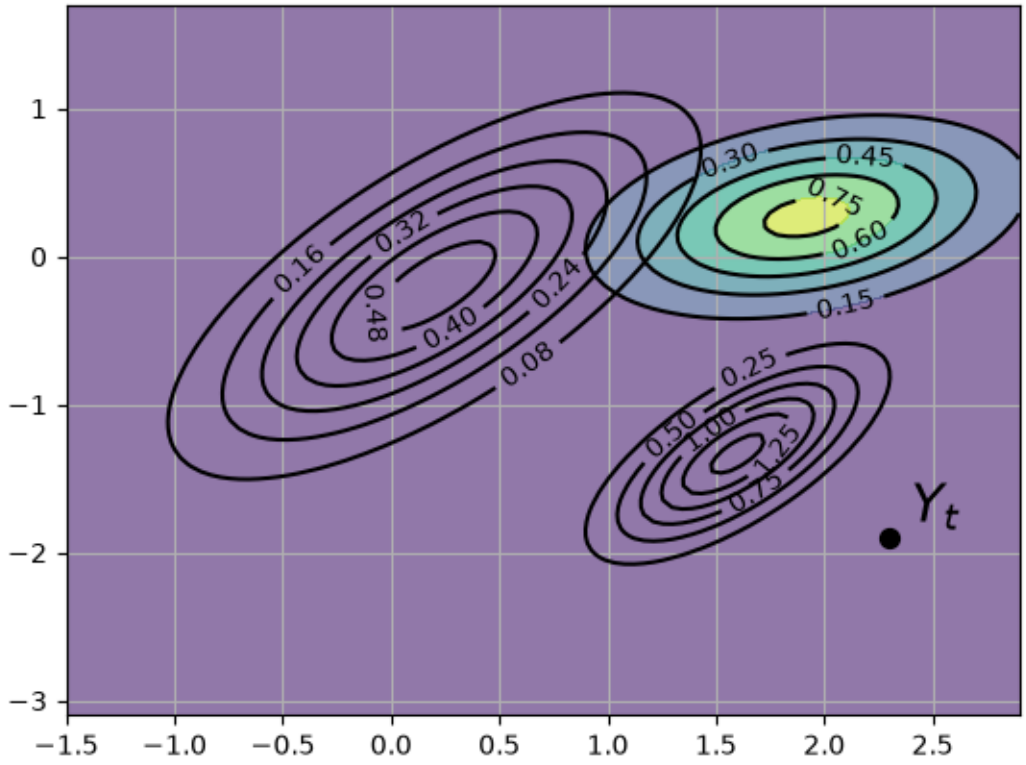
$$A = \begin{pmatrix} 1.2 & 0.0 \\ 0.0 & -0.2 \end{pmatrix}, \qquad Q = 0.3\Sigma$$

```
fig, ax = plt.subplots()
ax.grid()

# Density 1
Z = gen_gaussian_plot_vals(μ, Σ)
cs1 = ax.contour(X, Y, Z, 6, colors="black")
ax.clabel(cs1, inline=1, fontsize=10)

# Density 2
M = Σ @ G.T @ linalg.inv(G @ Σ @ G.T + R)
μ_F = μ + M @ (y - G @ μ)
Σ_F = Σ - M @ G @ Σ
Z_F = gen_gaussian_plot_vals(μ_F, Σ_F)
cs2 = ax.contour(X, Y, Z_F, 6, colors="black")
ax.clabel(cs2, inline=1, fontsize=10)

# Density 3
```

(continues on next page)

(continued from previous page)

```
new_μ = A @ μ_F
new_Σ = A @ Σ_F @ A.T + Q
new_Z = gen_gaussian_plot_vals(new_μ, new_Σ)
cs3 = ax.contour(X, Y, new_Z, 6, colors="black")
ax.clabel(cs3, inline=1, fontsize=10)
ax.contourf(X, Y, new_Z, 6, alpha=0.6, cmap="viridis")
y_1, y_2 = y[0].item(), y[1].item()
ax.scatter(y_1, y_2, marker="o", s=50, color="black", zorder=3)
ax.text(y_1 + 0.1, y_2 + 0.1, "$Y_t$", fontsize=20, color="black")

plt.show()
```



45.2.3 The recursive procedure

Let's look back at what we've done.

We started the current period with a prior density $p_t(x)$ for the hidden state X_t.

We then observed the signal Y_t and updated the prior density to the filtering density $p_t(x \mid Y_t)$.

Finally, we used the law of motion (45.6) for $\{X_t\}$ to update to the predictive density $p_{t+1}(x)$ for X_{t+1}.

If we now step into the next period, we are ready to go round again, taking $p_{t+1}(x)$ as the current prior density and reading in the new observation Y_{t+1}.

Using this time-indexed notation, the full recursive procedure is:

1. Start the current period with prior density $p_t(x) = N(\mu_t, \Sigma_t)$ for X_t.
2. Observe current signal $Y_t = y_t$.

3. Compute the filtering density $p_t(x \mid y_t) = N(\mu_t^F, \Sigma_t^F)$ from $p_t(x)$ and y_t, applying Bayes rule and the conditional distribution (45.3).
4. Compute the predictive density $p_{t+1}(x) = N(\mu_{t+1}, \Sigma_{t+1})$ for X_{t+1} from the filtering density and (45.6).
5. Increment t by one and go to step 1.

Repeating (45.7), the dynamics for μ_t and Σ_t are as follows

$$\mu_{t+1} = A\mu_t + K_{\Sigma_t}(y_t - G\mu_t)$$
$$\Sigma_{t+1} = A\Sigma_t A^\top - K_{\Sigma_t} G\Sigma_t A^\top + Q$$

These are the standard dynamic equations for the Kalman filter (see, for example, [Ljungqvist and Sargent, 2018], page 58).

Note

Here μ_t is the filter's prediction of the hidden state X_t.

In much of the Kalman filter literature it is written $\hat{x}_t$, emphasizing that it is an estimate of X_t.

45.3 Convergence

The matrix Σ_t is a measure of the uncertainty of our prediction μ_t of X_t.

Apart from special cases, this uncertainty will never be fully resolved, regardless of how much time elapses.

One reason is that our prediction μ_t is made based on information available at $t-1$, not t.

Even if we knew the precise realized value $X_{t-1} = x_{t-1}$ (which we don't), the transition equation (45.6) implies that $X_t = Ax_{t-1} + W_t$.

Since the shock W_t is not observable at $t-1$, any time $t-1$ prediction of X_t will incur some error (unless W_t is degenerate).

However, it is certainly possible that Σ_t converges to a constant matrix as $t \to \infty$.

To study this topic, let's expand the second equation in (45.7):

$$\Sigma_{t+1} = A\Sigma_t A^\top - A\Sigma_t G^\top (G\Sigma_t G^\top + R)^{-1} G\Sigma_t A^\top + Q \tag{45.7}$$

This is a nonlinear difference equation in Σ_t.

A fixed point of (45.7) is a constant matrix Σ such that

$$\Sigma = A\Sigma A^\top - A\Sigma G^\top (G\Sigma G^\top + R)^{-1} G\Sigma A^\top + Q \tag{45.8}$$

Equation (45.7) is known as a discrete-time Riccati difference equation.

Equation (45.8) is known as a discrete-time algebraic Riccati equation.

Conditions under which a fixed point exists and the sequence $\{\Sigma_t\}$ converges to it are discussed in [Anderson *et al.*, 1996] and [Anderson and Moore, 2005], chapter 4.

A sufficient (but not necessary) condition is that all the eigenvalues λ_i of A satisfy $|\lambda_i| < 1$.

See, for example, [Anderson and Moore, 2005], p. 77.

(This strong condition assures that the unconditional distribution of X_t converges as $t \to \infty$.)

In this case, for any symmetric nonnegative definite initial choice of Σ_0, the sequence $\{\Sigma_t\}$ in (45.7) converges to a nonnegative symmetric matrix Σ that solves (45.8).

45.4 Implementation

The class `Kalman` from the QuantEcon.py package implements the Kalman filter

- Instance data consists of:
  - the moments (μ_t, Σ_t) of the current prior, stored as the attributes `x_hat` and `Sigma` (the mean μ_t is named `x_hat` because it is also written $\hat{x}_t$ in much of the literature).
  - An instance of the LinearStateSpace class from QuantEcon.py.

The latter represents a linear state space model of the form

$$X_{t+1} = AX_t + Cw_{t+1}$$
$$Y_t = GX_t + Hv_t$$

where X_t and Y_t denote random variables, and the shocks w_t and v_t are IID standard normals.

To connect this with the notation of this lecture we set

$$Q := CC^\top \quad \text{and} \quad R := HH^\top$$

- The class `Kalman` from the QuantEcon.py package has a number of methods, some that we will wait to use until we study more advanced applications in subsequent lectures.
- Methods pertinent for this lecture are:
  - `prior_to_filtered`, which updates (μ_t, Σ_t) to (μ_t^F, Σ_t^F)
  - `filtered_to_forecast`, which updates the filtering distribution to the predictive distribution – which becomes the new prior $(\mu_{t+1}, \Sigma_{t+1})$
  - `update`, which combines the last two methods
  - a `stationary_values`, which computes the solution to (45.8) and the corresponding (stationary) Kalman gain

You can view the program on GitHub.

45.5 Exercises

Exercise 45.5.1

Consider the following simple application of the Kalman filter, loosely based on [Ljungqvist and Sargent, 2018], section 2.9.2.

Suppose that

- all variables are scalars
- the hidden state $\{X_t\}$ is in fact constant, equal to some $\theta \in \mathbb{R}$ unknown to the modeler

State dynamics are therefore given by (45.6) with $A = 1$, $Q = 0$ and $X_0 = \theta$.

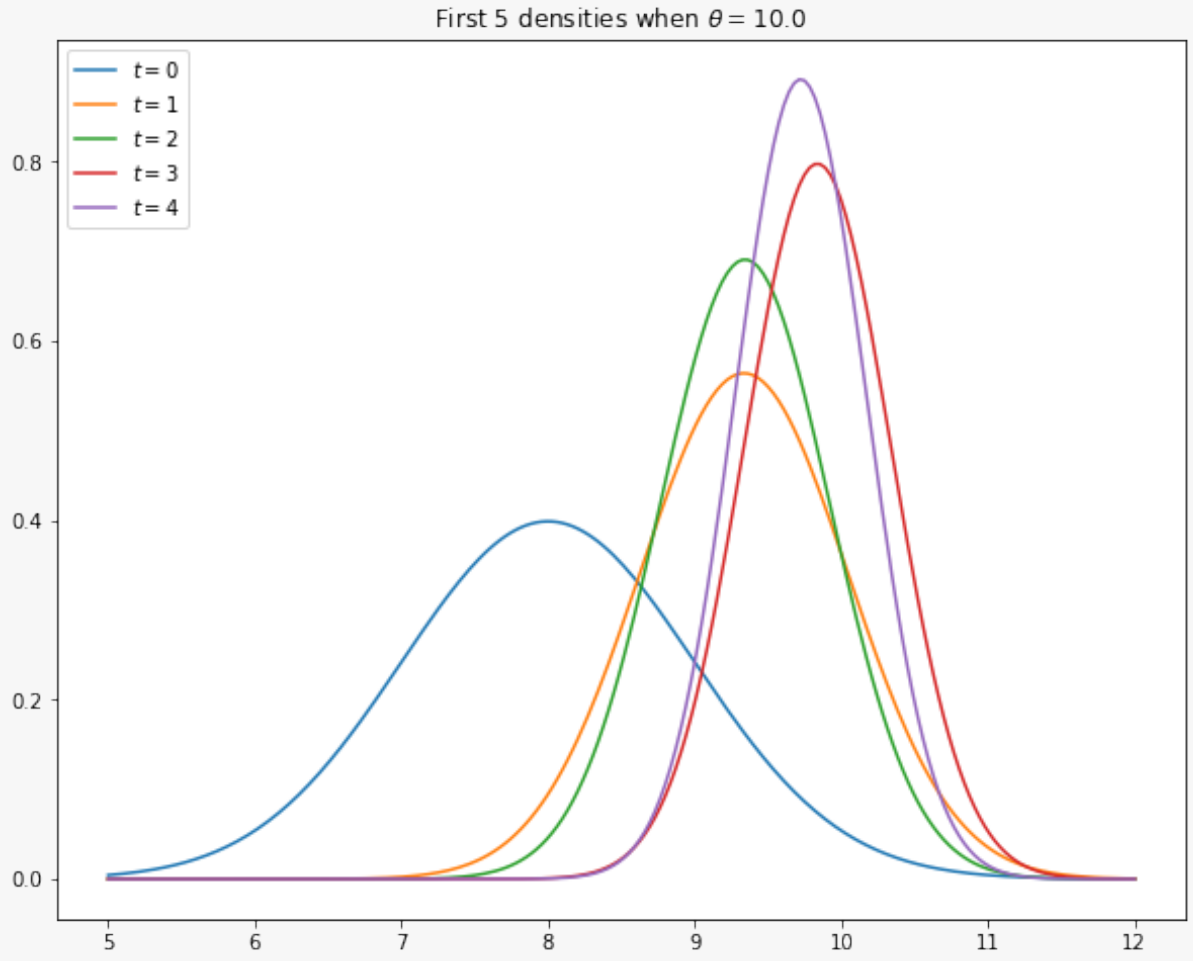
The measurement equation is $Y_t = \theta + v_t$ where v_t is $N(0, 1)$ and IID.

The task of this exercise is to simulate the model and, using the code from `kalman.py`, plot the first five predictive densities $p_t(x) = N(\mu_t, \Sigma_t)$ for X_t.

As shown in [Ljungqvist and Sargent, 2018], sections 2.9.1–2.9.2, these distributions asymptotically put all mass on the unknown value θ.

In the simulation, take $\theta = 10$, $\mu_0 = 8$ and $\Sigma_0 = 1$.

Your figure should – modulo randomness – look something like this

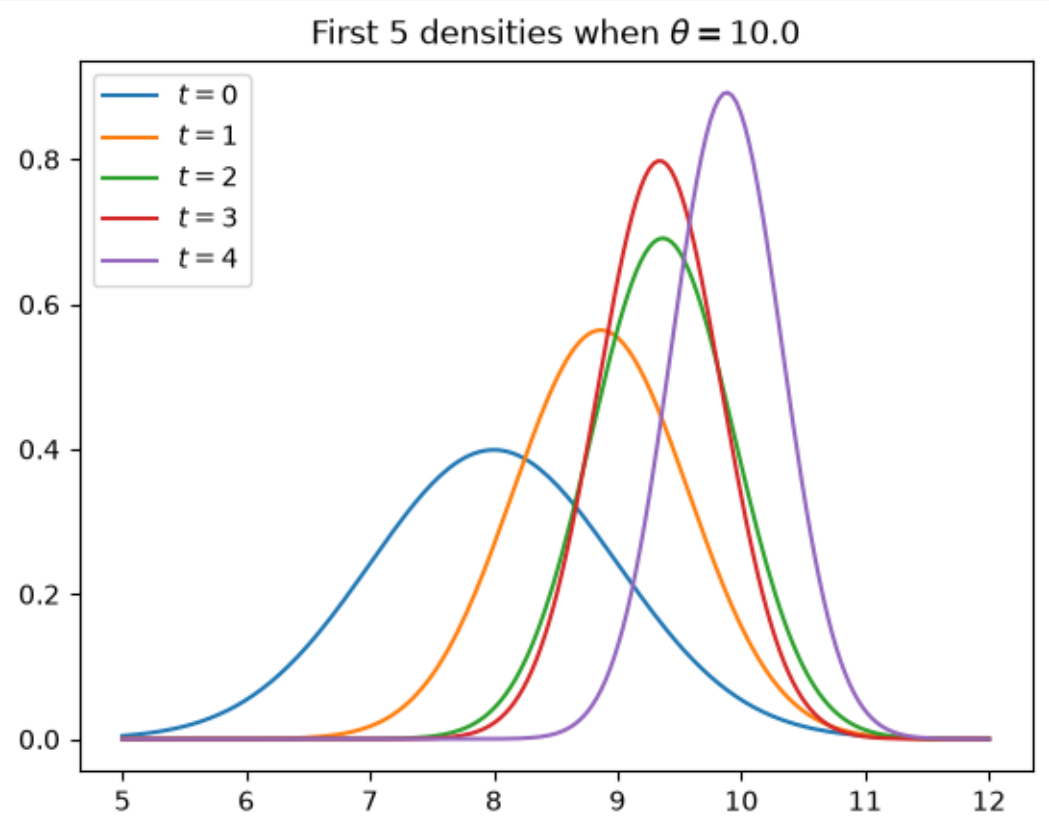


Solution

Here is one solution:

```
# Parameters
θ = 10  # Constant value of state X_t
A, C, G, H = 1, 0, 1, 1
ss = LinearStateSpace(A, C, G, H, mu_0=θ)

# Set prior, initialize kalman filter
μ_0, Σ_0 = 8, 1
kalman = Kalman(ss, μ_0, Σ_0)

# Draw observations of y from state space model
N = 5
x, y = ss.simulate(N)
```

```
y = y.flatten()

# Set up plot
fig, ax = plt.subplots()
xgrid = np.linspace(θ - 5, θ + 2, 200)

for i in range(N):
    # Record the current predicted mean and variance
    m, v = kalman.x_hat.item(), kalman.Sigma.item()
    # Plot, update filter
    ax.plot(xgrid, norm.pdf(xgrid, loc=m, scale=np.sqrt(v)), label=f'$t={i}$')
    kalman.update(y[i])

ax.set_title(f'First {N} densities when $\\theta = {θ:.1f}$')
ax.legend(loc='upper left')
plt.show()
```



Exercise 45.5.2

The preceding figure gives some support to the idea that probability mass converges to θ.

To get a better idea, choose a small $\epsilon > 0$ and calculate the error

$$z_t := 1 - \int_{\theta-\epsilon}^{\theta+\epsilon} p_t(x)dx$$

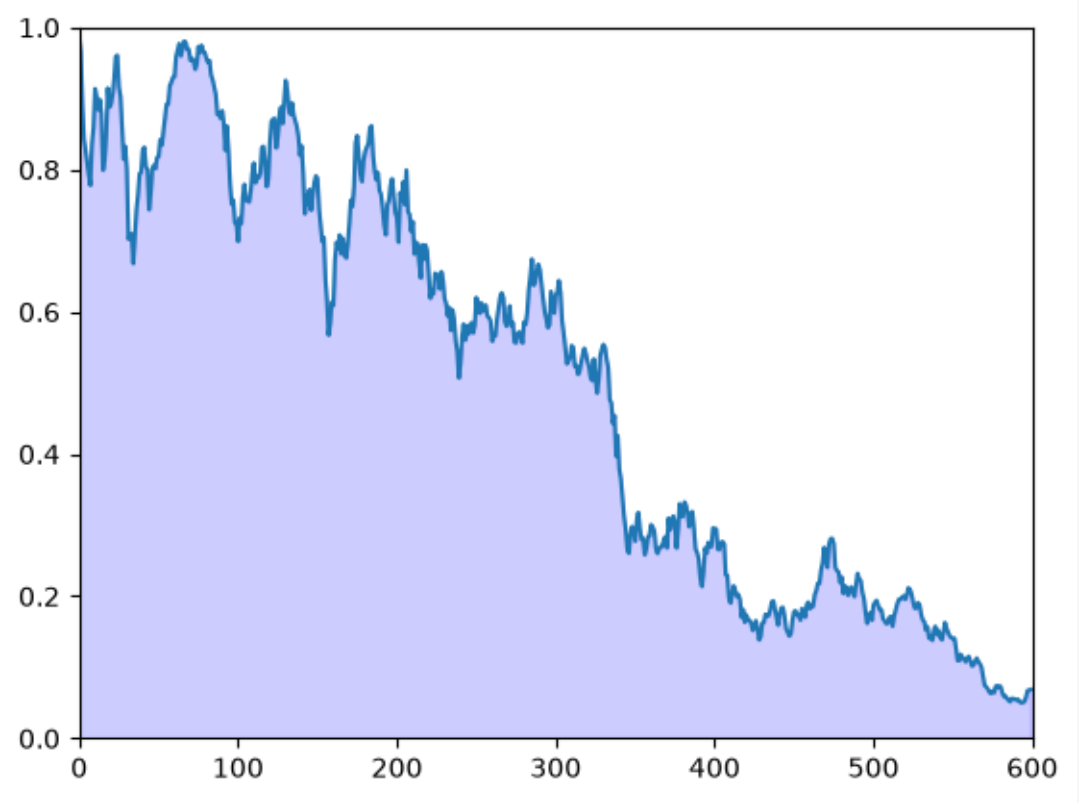
for $t = 0, 1, 2, \ldots, T$.

Plot z_t against t, setting $\epsilon = 0.1$ and $T = 600$.

Solution

Here is one solution:

```
ϵ = 0.1
θ = 10  # Constant value of state X_t
A, C, G, H = 1, 0, 1, 1
ss = LinearStateSpace(A, C, G, H, mu_0=θ)

μ_0, Σ_0 = 8, 1
kalman = Kalman(ss, μ_0, Σ_0)

T = 600
z = np.empty(T)
x, y = ss.simulate(T)
y = y.flatten()

for t in range(T):
    # Record the current predicted mean and variance and plot their densities
    m, v = kalman.x_hat.item(), kalman.Sigma.item()

    f = lambda x: norm.pdf(x, loc=m, scale=np.sqrt(v))
    integral, error = quad(f, θ - ϵ, θ + ϵ)
    z[t] = 1 - integral

    kalman.update(y[t])

fig, ax = plt.subplots()
ax.set_ylim(0, 1)
ax.set_xlim(0, T)
ax.plot(range(T), z)
ax.fill_between(range(T), np.zeros(T), z, color="blue", alpha=0.2)
plt.show()
```



Exercise 45.5.3

As discussed *above*, if the shock sequence $\{W_t\}$ is not degenerate, then it is not in general possible to predict X_t without error at time $t-1$ (and this would be the case even if we could observe X_{t-1}).

Let's now compare the prediction μ_t made by the Kalman filter against a competitor who **is** allowed to observe X_{t-1}.

This competitor will use the conditional expectation $\mathbb{E}[X_t \mid X_{t-1}]$, which in this case is AX_{t-1}.

The conditional expectation is known to be the optimal prediction method in terms of minimizing mean squared error.

(More precisely, the minimizer of $\mathbb{E}\,\|X_t - g(X_{t-1})\|^2$ with respect to g is $g^*(X_{t-1}) := \mathbb{E}[X_t \mid X_{t-1}]$)

Thus we are comparing the Kalman filter against a competitor who has more information (in the sense of being able to observe the latent state) and behaves optimally in terms of minimizing squared error.

Our horse race will be assessed in terms of realized squared error.

In particular, your task is to generate a graph plotting simulated realizations of both $\|X_t - AX_{t-1}\|^2$ and $\|X_t - \mu_t\|^2$ against t for $t = 1, \ldots, 49$.

In the code below, `x[:, t]` is the realized value of X_t along the simulated path.

For the parameters, set $G = I$, $R = 0.5I$ and $Q = 0.3I$, where I is the 2×2 identity.

Set

$$A = \begin{pmatrix} 0.5 & 0.4 \\ 0.6 & 0.3 \end{pmatrix}$$

To initialize the prior density, set

$$\Sigma_0 = \begin{pmatrix} 0.9 & 0.3 \\ 0.3 & 0.9 \end{pmatrix}$$

and $\mu_0 = (8, 8)$.

Finally, set the realized initial state to $x_0 = (0, 0)$.

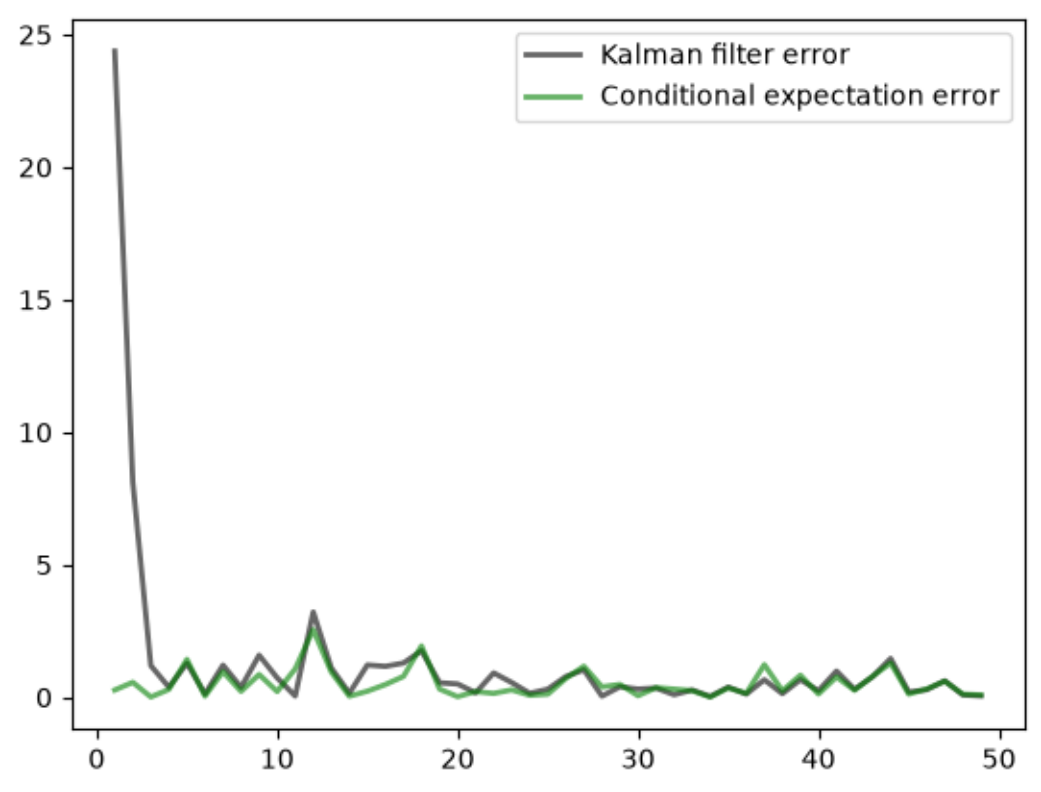
Solution

Here is one solution:

```
# Define A, C, G, H
G = np.identity(2)
H = np.sqrt(0.5) * np.identity(2)

A = [[0.5, 0.4],
     [0.6, 0.3]]
C = np.sqrt(0.3) * np.identity(2)

# Set up state space mode, initial value x_0 set to zero
ss = LinearStateSpace(A, C, G, H, mu_0 = np.zeros(2))

# Define the prior density
Σ = [[0.9, 0.3],
     [0.3, 0.9]]
Σ = np.array(Σ)
μ = np.array([8, 8])

# Initialize the Kalman filter
kn = Kalman(ss, μ, Σ)

# Print eigenvalues of A
print("Eigenvalues of A:")
print(eigvals(A))

# Print stationary Σ
S, K = kn.stationary_values()
print("Stationary prediction error variance:")
print(S)

# Generate the plot
T = 50
x, y = ss.simulate(T)

e1 = np.empty(T-1)
e2 = np.empty(T-1)

for t in range(1, T):
    kn.update(y[:, t-1])
    diff1 = x[:, t] - kn.x_hat.flatten()
    diff2 = x[:, t] - A @ x[:, t-1]
    e1[t-1] = diff1 @ diff1
    e2[t-1] = diff2 @ diff2

fig, ax = plt.subplots()
```

```
ax.plot(range(1, T), e1, 'k-', lw=2, alpha=0.6,
        label='Kalman filter error')
ax.plot(range(1, T), e2, 'g-', lw=2, alpha=0.6,
        label='Conditional expectation error')
ax.legend()
plt.show()
```

```
Eigenvalues of A:
[ 0.9+0.j -0.1+0.j]
Stationary prediction error variance:
[[0.40329108 0.1050718 ]
 [0.1050718  0.41061709]]
```



Observe how, after an initial learning period, the Kalman filter performs quite well, even relative to the competitor who predicts optimally with knowledge of the latent state.

Exercise 45.5.4

Try varying the coefficient 0.3 in $Q = 0.3I$ up and down.

Observe how the diagonal values in the stationary solution Σ (see (45.8)) increase and decrease in line with this coefficient.

The interpretation is that more randomness in the law of motion for X_t causes more (permanent) uncertainty in prediction.

CHAPTER

FORTYSIX

ANOTHER LOOK AT THE KALMAN FILTER

Contents

- *Another Look at the Kalman Filter*
  - *A worker's output*
  - *A firm's wage-setting policy*
  - *A state-space representation*
  - *An innovations representation*
  - *Some computational experiments*
  - *Future extensions*

In *A First Look at the Kalman Filter*, we used a Kalman filter to estimate locations of a rocket.

In this lecture, we'll use the Kalman filter to infer a worker's human capital and the effort that the worker devotes to accumulating human capital, neither of which the firm observes directly.

This lecture is an application of the filtering and forecasting recursion introduced in *A First Look at the Kalman Filter*.

The firm learns about those things only by observing a history of the output that the worker generates for the firm, and from understanding how that output depends on the worker's human capital and how human capital evolves as a function of the worker's effort.

We'll posit a rule that expresses how much the firm pays the worker each period as a function of the firm's information each period.

The lecture *The Kalman Filter and Vector Autoregressions* uses the same recursion to construct innovations, likelihoods, and vector autoregressions.

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install quantecon
```

To conduct simulations, we bring in these imports, as in *A First Look at the Kalman Filter*.

```
import matplotlib.pyplot as plt
import numpy as np
from quantecon import Kalman, LinearStateSpace
from collections import namedtuple
from scipy.stats import multivariate_normal
import matplotlib as mpl
```

(continues on next page)

(continued from previous page)

```
mpl.rcParams['text.usetex'] = True
mpl.rcParams['text.latex.preamble'] = r'\usepackage{amsmath,amsfonts}'
```

46.1 A worker's output

A representative worker is permanently employed at a firm.

A worker's output is described by the following dynamic process:

$$
\begin{aligned}
h_{t+1} &= \alpha h_t + \beta u_t + c\epsilon_{t+1}, \quad \epsilon_{t+1} \sim N(0,1) \\
u_{t+1} &= u_t \\
y_t &= g h_t + v_t, \quad v_t \sim N(0,R)
\end{aligned} \tag{46.1}
$$

Here

- h_t is the logarithm of human capital at time t
- u_t is the logarithm of the worker's effort at accumulating human capital at t
- y_t is the logarithm of the worker's output at time t
- ϵ_{t+1} is an IID standard normal shock to human capital
- $h_0 \sim N(\hat{h}_0, \sigma^2_{h,0})$
- $u_0 \sim N(\hat{u}_0, \sigma^2_{u,0})$

Parameters of the model are $\alpha, \beta, c, R, g, \hat{h}_0, \hat{u}_0, \sigma_{h,0}, \sigma_{u,0}$ where $\sigma_{h,0}$ and $\sigma_{u,0}$ are standard deviations of the firm's initial beliefs about h_0 and u_0.

We assume that h_0, u_0, $\{\epsilon_t\}$, and $\{v_t\}$ are mutually independent.

At time 0, a firm has hired the worker.

The worker is permanently attached to the firm and so works for the same firm at all dates $t = 0, 1, 2, ...$

At the beginning of time 0, the firm observes neither the worker's innate initial human capital h_0 nor its hard-wired permanent effort level u_0.

The firm believes that u_0 for a particular worker is drawn from a Gaussian probability distribution, and so is described by $u_0 \sim N(\hat{u}_0, \sigma^2_{u,0})$.

The h_t part of a worker's "type" moves over time, while the equation $u_{t+1} = u_t$ implies $u_t = u_0$ for all t.

Thus, from the firm's point of view, effort is a fixed, unobserved component of the worker's type that must be inferred from output observations.

At the beginning of time $t \geq 1$, before setting the wage w_t, the firm has observed the history $y^{t-1} = [y_{t-1}, y_{t-2}, \ldots, y_0]$.

The firm does not observe the worker's "type" (h_0, u_0).

After production in period t, the firm observes the worker's output y_t, which it then uses to update its beliefs going into period $t+1$.

46.2 A firm's wage-setting policy

At time $t \geq 1$, before observing current output y_t, the firm sets the worker's log wage using the past output history y^{t-1}:

$$w_t = g\mathbb{E}[h_t | y^{t-1}], \quad t \geq 1$$

and at time 0 pays the worker a log wage equal to the unconditional mean of y_0:

$$w_0 = g\hat{h}_0$$

In using this payment rule, the firm is taking into account that the worker's log output today is partly due to the random component v_t that comes entirely from luck, and that is assumed to be independent of h_t and u_t.

46.3 A state-space representation

Write system *(46.1.1)* in the state-space form

$$\begin{bmatrix} h_{t+1} \\ u_{t+1} \end{bmatrix} = \begin{bmatrix} \alpha & \beta \\ 0 & 1 \end{bmatrix} \begin{bmatrix} h_t \\ u_t \end{bmatrix} + \begin{bmatrix} c \\ 0 \end{bmatrix} \epsilon_{t+1}$$
$$y_t = \begin{bmatrix} g & 0 \end{bmatrix} \begin{bmatrix} h_t \\ u_t \end{bmatrix} + v_t$$

which is equivalent with

$$\begin{aligned} x_{t+1} &= Ax_t + C\epsilon_{t+1} \\ y_t &= Gx_t + v_t \\ x_0 &\sim N(\hat{x}_0, \Sigma_0) \end{aligned} \tag{46.2}$$

where

$$x_t = \begin{bmatrix} h_t \\ u_t \end{bmatrix}, \quad \hat{x}_0 = \begin{bmatrix} \hat{h}_0 \\ \hat{u}_0 \end{bmatrix}, \quad \Sigma_0 = \begin{bmatrix} \sigma_{h,0}^2 & 0 \\ 0 & \sigma_{u,0}^2 \end{bmatrix}$$

To compute the firm's wage setting policy, we first create a `namedtuple` to store the parameters of the model

```
WorkerModel = namedtuple("WorkerModel",
                 ('A', 'C', 'G', 'R', 'xhat_0', 'Σ_0'))

def create_worker(α=.8, β=.2, c=.2,
                  R=.5, g=1.0, hhat_0=4, uhat_0=4,
                  σ_h=2, σ_u=2):

    A = np.array([[α, β],
                  [0, 1]])
    C = np.array([[c],
                  [0]])
    G = np.array([g, 0])

    # Define initial state and covariance matrix
    xhat_0 = np.array([[hhat_0],
                       [uhat_0]])

    # σ_h and σ_u are standard deviations, so Σ_0 holds their squares
    Σ_0 = np.array([[σ_h**2, 0],
                    [0, σ_u**2]])

    return WorkerModel(A=A, C=C, G=G, R=R, xhat_0=xhat_0, Σ_0=Σ_0)
```

Please note how the `WorkerModel` namedtuple creates all of the objects required to compute an associated state-space representation (46.2).

This is handy, because in order to simulate a history $\{y_t, h_t\}$ for a worker, we'll want to form state space system for him/her by using the `LinearStateSpace` class.

```
# Define A, C, G, R, xhat_0, Σ_0
worker = create_worker()
A, C, G, R = worker.A, worker.C, worker.G, worker.R
xhat_0, Σ_0 = worker.xhat_0, worker.Σ_0

# Create a LinearStateSpace object
ss = LinearStateSpace(A, C, G, np.sqrt(R),
        mu_0=xhat_0, Sigma_0=np.zeros((2,2)))

T = 100
x, y = ss.simulate(T)
y = y.flatten()

h_0, u_0 = x[0, 0], x[1, 0]
```

We set `Sigma_0=np.zeros((2,2))` so that the simulation fixes a particular worker's initial state (h_0, u_0), while the firm still enters period 0 with the non-degenerate prior beliefs $\hat{x}_0$ and Σ_0 that drive its Kalman filter.

Next, to compute the firm's policy for setting the log wage based on the information it has about the worker, we use the Kalman filter described in this QuantEcon lecture *A First Look at the Kalman Filter*.

In particular, we want to compute all of the objects in an "innovation representation".

46.4 An innovations representation

We have all the objects in hand required to form an innovations representation for the output process $\{y_t\}_{t=0}^{T-1}$ for a worker.

Let's code that up now.

$$\begin{aligned} \hat{x}_{t+1} &= A\hat{x}_t + K_t a_t \\ y_t &= G\hat{x}_t + a_t \end{aligned}$$

where $\hat{x}_t = \mathbb{E}[x_t | y^{t-1}]$ is the firm's forecast of the state formed before y_t is observed, and K_t is the Kalman gain matrix at time t.

Here $a_t = y_t - G\hat{x}_t$ is the **innovation** at time t, the firm's one-step-ahead error in forecasting output y_t from the history y^{t-1}.

Because $\hat{x}_t$ is conditioned on y^{t-1} rather than y^t, the gain K_t folds together the filtering update that uses the current observation y_t and the one-step-ahead prediction that advances the state to $t+1$.

Writing $\Sigma_t = \mathbb{E}[(x_t - \hat{x}_t)(x_t - \hat{x}_t)' | y^{t-1}]$ for the conditional covariance of the state, the gain is

$$K_t = A\Sigma_t G'(G\Sigma_t G' + R)^{-1} = AL_t,$$

where $L_t = \Sigma_t G'(G\Sigma_t G' + R)^{-1}$ is the filtering gain that updates the firm's beliefs about x_t once y_t is observed.

We accomplish this in the following code that uses the `Kalman` class.

```
kalman = Kalman(ss, xhat_0, Σ_0)
Σ_t = np.zeros((*Σ_0.shape, T))
y_hat_t = np.zeros(T)
x_hat_t = np.zeros((2, T))

for t in range(T):
    # Record the firm's belief about x_t given y^{t-1}, before seeing y_t
    x_hat, Σ = kalman.x_hat, kalman.Sigma
    Σ_t[:, :, t] = Σ
    x_hat_t[:, t] = x_hat.reshape(-1)
    y_hat_t[t] = (worker.G @ x_hat).item()

    # Then incorporate the observation y_t and advance the filter to t+1
    kalman.update(y[t])

u_hat_t = x_hat_t[1, :]
```

For this fixed worker initial state, we plot $\mathbb{E}[y_t|y^{t-1}] = G\hat{x}_t$ where $\hat{x}_t = \mathbb{E}[x_t|y^{t-1}]$.

We also plot $\mathbb{E}[u_0|y^{t-1}]$, which is the firm's inference about a worker's hard-wired "work ethic" u_0, conditioned on information y^{t-1} that it has about him or her coming into period t.

We can watch how the firm updates its inference $\mathbb{E}[u_0|y^{t-1}]$ about the worker's work ethic as more output observations arrive.

```
fig, ax = plt.subplots(1, 2)

ax[0].plot(y_hat_t, label=r'$\mathbb{E}[y_t| y^{t-1}]$')
ax[0].set_xlabel('time')
ax[0].set_ylabel(r'$\mathbb{E}[y_t| y^{t-1}]$')
ax[0].set_title(r'$\mathbb{E}[y_t| y^{t-1}]$ over time')
ax[0].legend()

ax[1].plot(u_hat_t, label=r'$\mathbb{E}[u_0|y^{t-1}]$')
ax[1].axhline(y=u_0, color='grey',
              linestyle='dashed', label=fr'$u_0={u_0:.2f}$')
ax[1].set_xlabel('time')
ax[1].set_ylabel(r'$\mathbb{E}[u_0|y^{t-1}]$')
ax[1].set_title('Inferred work ethic over time')
ax[1].legend()

fig.tight_layout()
plt.show()
```

46.5 Some computational experiments

Let's look at Σ_0 and Σ_{T-1} in order to see how much the firm learns about the hidden state during the horizon we have set.

```
print(Σ_t[:, :, 0])
```

```
[[4. 0.]
 [0. 4.]]
```

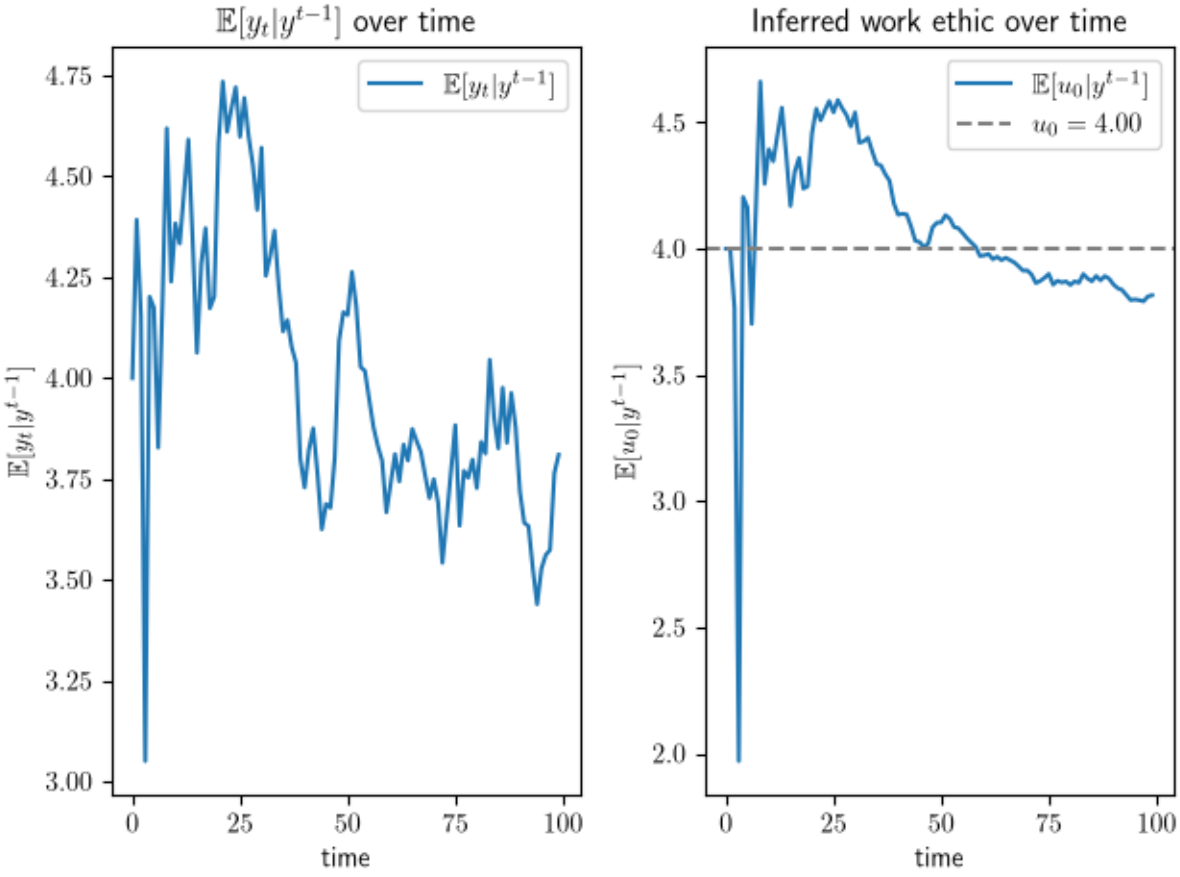


Fig. 46.1: Firm's output forecast and inferred work ethic over time

```
print(Σ_t[:, :, -1])
```

```
[[0.09403859 0.00997018]
 [0.00997018 0.01592704]]
```

Evidently, the conditional variances become smaller over time.

It is enlightening to portray how the firm's conditional beliefs evolve by plotting contours of the conditional bivariate normal density of x_t given y^{t-1} at various t's.

```
# Create a grid of points for contour plotting
h_range = np.linspace(x_hat_t[0, :].min()-0.5*Σ_t[0, 0, 1],
                      x_hat_t[0, :].max()+0.5*Σ_t[0, 0, 1], 100)
u_range = np.linspace(x_hat_t[1, :].min()-0.5*Σ_t[1, 1, 1],
                      x_hat_t[1, :].max()+0.5*Σ_t[1, 1, 1], 100)
h, u = np.meshgrid(h_range, u_range)

# Create a figure with subplots for each time step
fig, axs = plt.subplots(1, 3, figsize=(12, 7))

# Iterate through each time step
for i, t in enumerate(np.linspace(0, T-1, 3, dtype=int)):
    # Create a multivariate normal distribution with x_hat and Σ at time step t
    μ = x_hat_t[:, t]
    cov = Σ_t[:, :, t]
    mvn = multivariate_normal(mean=μ, cov=cov)

    # Evaluate the multivariate normal PDF on the grid
    pdf_values = mvn.pdf(np.dstack((h, u)))

    # Create a contour plot for the PDF
    con = axs[i].contour(h, u, pdf_values, cmap='viridis')
    axs[i].clabel(con, inline=1, fontsize=10)
    axs[i].set_title(f'time step {t}')
    axs[i].set_xlabel(r'$h_{{{}}}$'.format(str(t)))
    axs[i].set_ylabel(r'$u_{{{}}}$'.format(str(t)))

    cov_latex = (
        r'$\Sigma_{{{}}}= \begin{{bmatrix}} {:.2f} & {:.2f} \\ '
        r'{:.2f} & {:.2f} \end{{bmatrix}}$'
    ).format(t, cov[0, 0], cov[0, 1], cov[1, 0], cov[1, 1])
    axs[i].text(0.33, -0.15, cov_latex, transform=axs[i].transAxes)


plt.tight_layout()
plt.show()
```

Note how the accumulation of evidence y^{t-1} affects the shape of the density contours as sample size t grows.

Now let's use our code to set the hidden state x_0 to a particular vector in order to watch how a firm learns starting from some x_0 we are interested in.

For example, let's say $h_0 = 0$ and $u_0 = 4$.

Here is one way to do this.

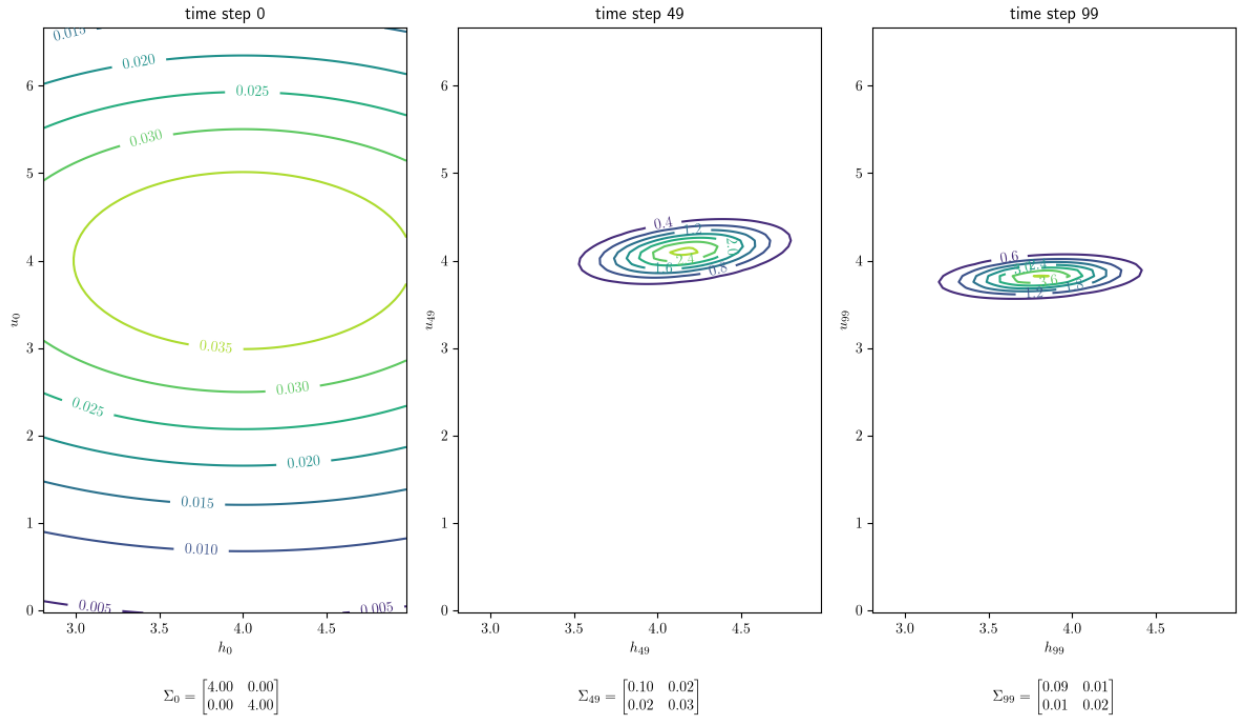


Fig. 46.2: Density contours for the firm's belief about x_t at three dates

```
# For example, we might want h_0 = 0 and u_0 = 4
μ_0 = np.array([[0.0],
                [4.0]])

# Create a LinearStateSpace object with Sigma_0 as a matrix of zeros
ss_example = LinearStateSpace(A, C, G, np.sqrt(R), mu_0=μ_0,
                              # This line forces exact h_0=0 and u_0=4
                              Sigma_0=np.zeros((2, 2))
                             )

T = 100
x, y = ss_example.simulate(T)
y = y.flatten()

# Now h_0=0 and u_0=4
h_0, u_0 = x[0, 0], x[1, 0]
print('h_0 =', h_0)
print('u_0 =', u_0)
```

```
h_0 = 0.0
u_0 = 4.0
```

Another way to accomplish the same goal is to use the following code.

```
# If we want to set the initial
# h_0 = hhat_0 = 0.0 and u_0 = uhat_0 = 4.0:
worker_example = create_worker(hhat_0=0.0, uhat_0=4.0)
```

(continues on next page)

(continued from previous page)

```
# The firm's prior stays at the original xhat_0 and Σ_0
ss_example = LinearStateSpace(A, C, G, np.sqrt(R),
                              # This line takes h_0=hhat_0 and u_0=uhat_0
                              mu_0=worker_example.xhat_0,
                              # This line forces exact h_0=hhat_0 and u_0=uhat_0
                              Sigma_0=np.zeros((2, 2))
                             )

T = 100
x, y = ss_example.simulate(T)
y = y.flatten()

# Now h_0 and u_0 will be exactly hhat_0 and uhat_0
h_0, u_0 = x[0, 0], x[1, 0]
print('h_0 =', h_0)
print('u_0 =', u_0)
```

```
h_0 = 0.0
u_0 = 4.0
```

For this worker, let's generate a plot like the one above.

```
# The firm filters output from ss_example using its prior beliefs
# xhat_0 and Σ_0
kalman = Kalman(ss_example, xhat_0, Σ_0)
Σ_t = []
y_hat_t = np.zeros(T)
u_hat_t = np.zeros(T)

# Then we iteratively update the Kalman filter class using
# observation y based on the linear state model above:
for t in range(T):
    # Record the firm's belief about x_t given y^{t-1}, before seeing y_t
    x_hat, Σ = kalman.x_hat, kalman.Sigma
    Σ_t.append(Σ)
    y_hat_t[t] = (G @ x_hat).item()
    u_hat_t[t] = x_hat[1].item()

    # Then incorporate the observation y_t and advance the filter to t+1
    kalman.update(y[t])


# Generate plots for y_hat_t and u_hat_t
fig, ax = plt.subplots(1, 2)

ax[0].plot(y_hat_t, label=r'$\mathbb{E}[y_t| y^{t-1}]$')
ax[0].set_xlabel('time')
ax[0].set_ylabel(r'$\mathbb{E}[y_t| y^{t-1}]$')
ax[0].set_title(r'$\mathbb{E}[y_t| y^{t-1}]$ over time')
ax[0].legend()

ax[1].plot(u_hat_t, label=r'$\mathbb{E}[u_0|y^{t-1}]$')
ax[1].axhline(y=u_0, color='grey',
              linestyle='dashed', label=fr'$u_0={u_0:.2f}$')
ax[1].set_xlabel('time')
ax[1].set_ylabel(r'$\mathbb{E}[u_0|y^{t-1}]$')
```

(continues on next page)

(continued from previous page)

```
ax[1].set_title('Inferred work ethic over time')
ax[1].legend()

fig.tight_layout()
plt.show()
```

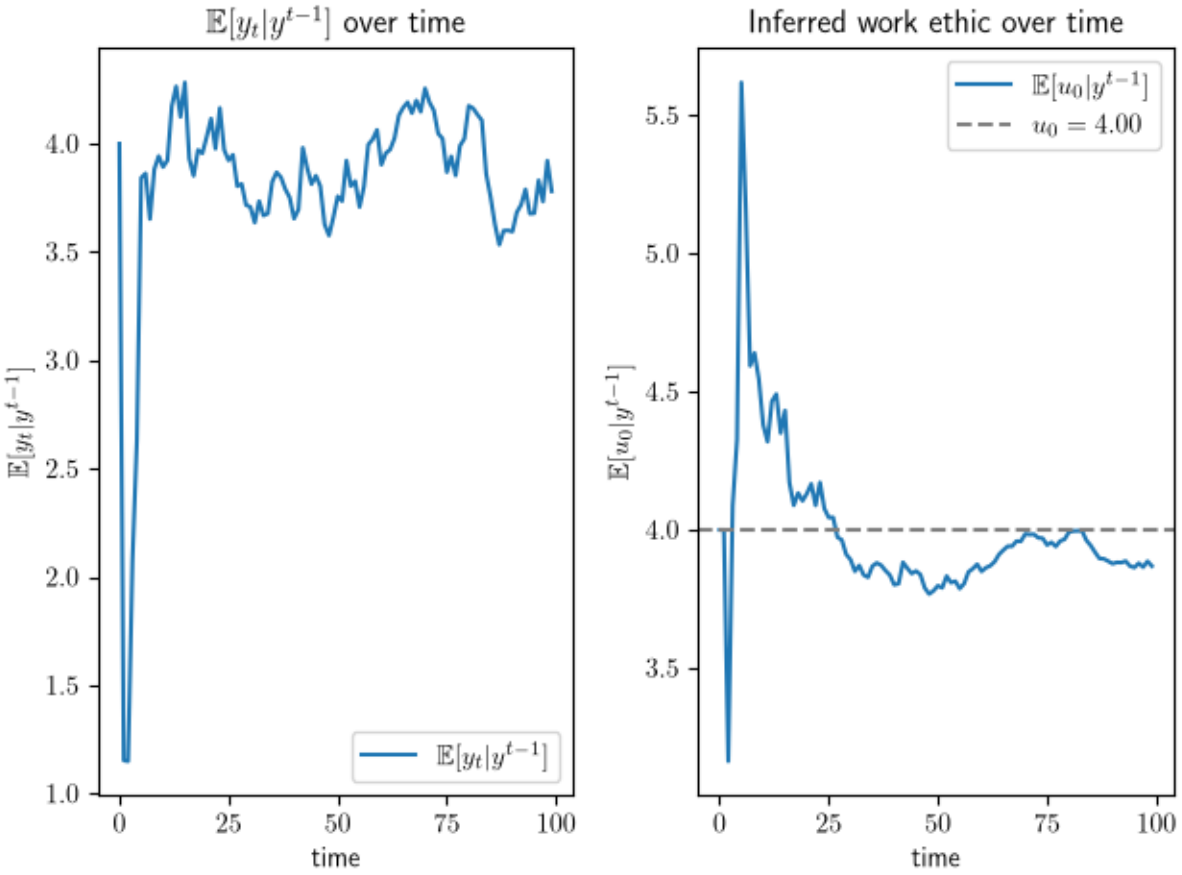


Fig. 46.3: Output forecast and inferred work ethic for the worker with $u_0 = 4$

More generally, we can change some or all of the parameters defining a worker in our `create_worker` namedtuple.

Here is an example.

```
# We can set these parameters when creating a worker, just like classes!
hard_working_worker = create_worker(α=.4, β=.8,
                        hhat_0=7.0, uhat_0=100, σ_h=2.5, σ_u=3.2)

print(hard_working_worker)
```

```
WorkerModel(A=array([[0.4, 0.8],
       [0. , 1. ]]), C=array([[0.2],
       [0. ]]), G=array([1., 0.]), R=0.5, xhat_0=array([[  7.],
       [100.]]), Σ_0=array([[ 6.25,  0.  ],
       [ 0.  , 10.24]]))
```

We can also simulate the system for $T = 100$ periods for different workers.

When effort feeds into output through human capital, so that $g \neq 0$ and $\beta \neq 0$, the firm's uncertainty about u_0 falls over time and the inferred work ethic converges toward the true u_0.

If $\beta = 0$, effort never affects h_t, and if $g = 0$, output carries no information about h_t, so in either case the firm cannot learn u_0 from output alone.

This shows that, under these observability conditions, the filter gradually teaches the firm about the worker's effort.

For three workers, we first plot the gap between the firm's inferred work ethic and the true u_0 over time.

```
num_workers = 3
T = 100
fig, ax = plt.subplots(figsize=(7, 7))

for i in range(num_workers):
    worker = create_worker(uhat_0=4+2*i)
    simulate_workers(worker, T, ax, name=fr'$\hat u_0 = {4+2*i}$',
                     random_state=2 + i)
ax.set_ylim(ymin=-2, ymax=2)
ax.legend()
plt.show()
```

In this simulation the firm's inferred work ethic moves toward the true u_0.

Under the correctly specified observable linear-Gaussian model, the posterior mean of u_0 is consistent, so this gap shrinks as the output history grows.

By setting `diff=False`, we instead plot the level of each worker's inferred work ethic $\mathbb{E}[u_0|y^{t-1}]$, shown together with a dashed line marking the true u_0.

```
fig, ax = plt.subplots(figsize=(7, 7))

uhat_0s = [2, -2, 1]
αs = [0.2, 0.3, 0.5]
βs = [0.2, 0.9, 0.3]

for i, (uhat_0, α, β) in enumerate(zip(uhat_0s, αs, βs)):
    worker = create_worker(uhat_0=uhat_0, α=α, β=β)
    simulate_workers(worker, T, ax, diff=False,
                     name=r'$u_{{{}, 0}}$'.format(i),
                     random_state=3 + i)

ax.legend(bbox_to_anchor=(1, 0.5))
plt.show()
```

These three workers differ in α and β as well as in $\hat{u}_0$, and the speed of learning differs sharply across them.

The worker with the largest β, here $u_{1,0}$ with $\beta = 0.9$, settles onto its dashed true value almost immediately, while the worker with the smallest β, here $u_{0,0}$ with $\beta = 0.2$, converges slower.

The reason is that effort affects output only through human capital, so in these stable examples with $|\alpha| < 1$ its steady-state effect on output is governed by $g\beta/(1-\alpha)$, and a small β leaves the firm with too little signal to pin down u_0 over this horizon.

The speed of learning also reflects the measurement noise R, the shock scale c, and the firm's prior variances.

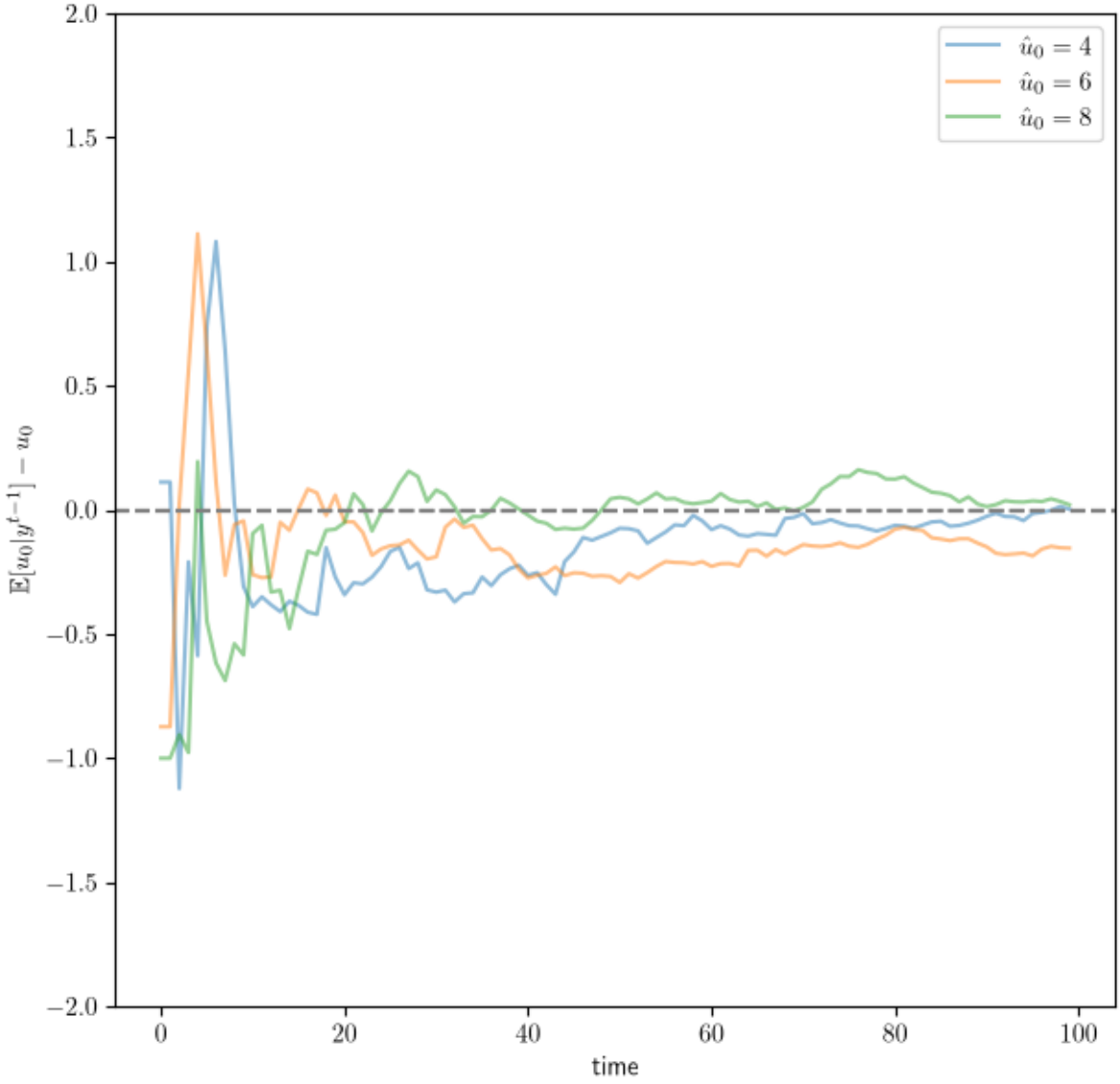


Fig. 46.4: Difference between inferred and true work ethic over time

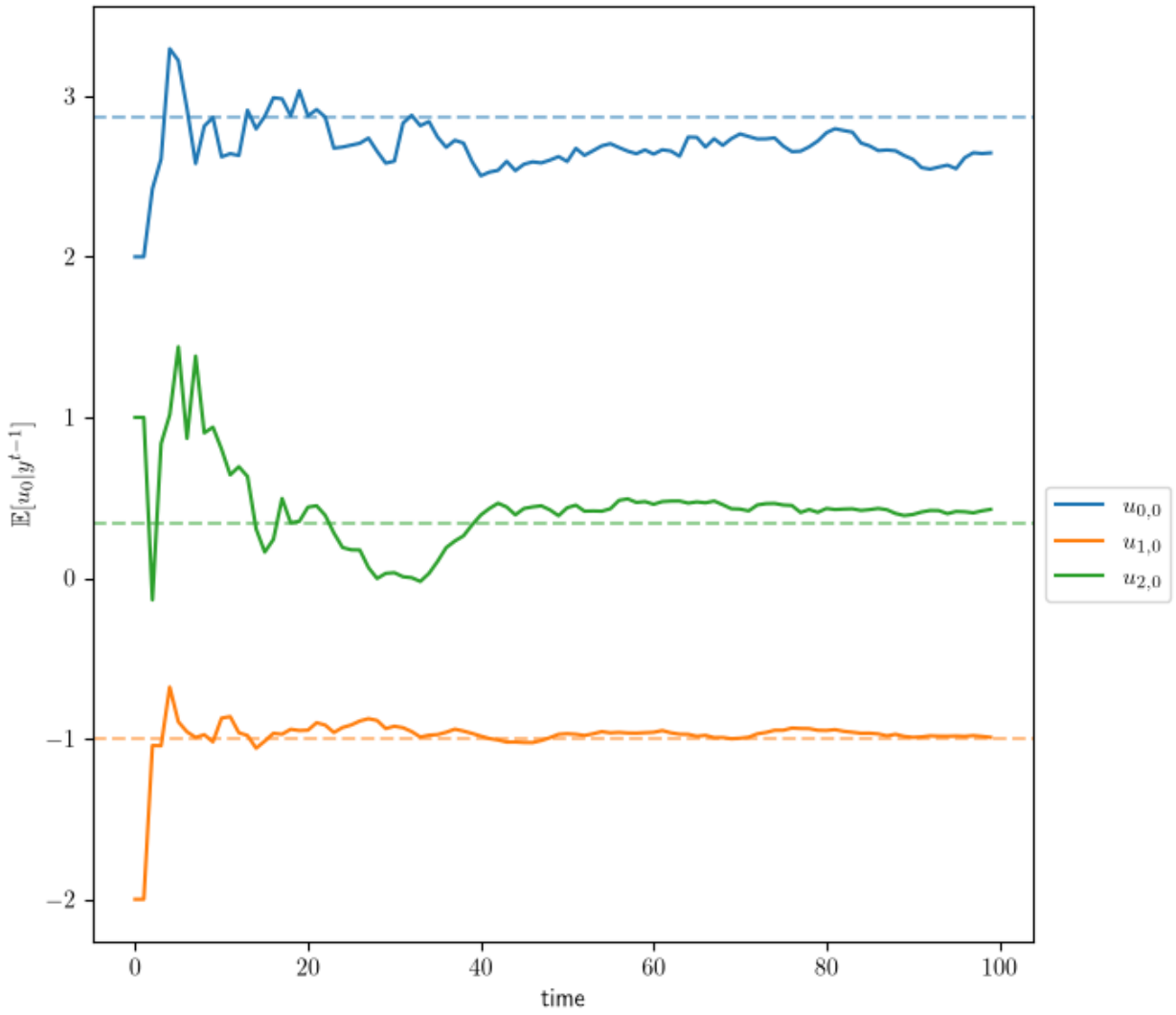


Fig. 46.5: Inferred work ethic over time for three workers

We can also give every worker the same true initial state, here $h_0 = 2$ and $u_0 = 1$, by passing a fixed `μ_sim_0` and a zero `Σ_sim_0` to `simulate_workers`.

```
fig, ax = plt.subplots(figsize=(7, 7))

μ_sim_0 = np.array([[2.0],
                    [1.0]])
Σ_sim_0 = np.zeros((2,2))

uhat_0s = [2, -2, 1]
αs = [0.2, 0.3, 0.5]
βs = [0.2, 0.9, 0.3]

for i, (uhat_0, α, β) in enumerate(zip(uhat_0s, αs, βs)):
    worker = create_worker(uhat_0=uhat_0, α=α, β=β)
    simulate_workers(worker, T, ax, μ_sim_0=μ_sim_0, Σ_sim_0=Σ_sim_0,
                     diff=False, name=r'$u_{{{}, 0}}$'.format(i))

# This controls the boundary of plots
ax.set_ylim(ymin=-3, ymax=3)
ax.legend(bbox_to_anchor=(1, 0.5))
plt.show()
```

Even though the firm begins from different prior means $\hat{u}_0$, all three workers share the true work ethic $u_0 = 1$, and the inferred paths converge to that common dashed line at speeds that again reflect each worker's β.

Finally, we track a single worker type under two different true effort levels, comparing a hard-working worker with $u_0 = 100$ against a normal worker with $u_0 = 30$.

```
T = 50
fig, ax = plt.subplots(figsize=(7, 7))

μ_sim_0_1 = np.array([[1],
                      [100]])
μ_sim_0_2 = np.array([[1],
                      [30]])
Σ_sim_0 = np.zeros((2, 2))

worker = create_worker(uhat_0=1, α=0.5, β=0.3)
simulate_workers(worker, T, ax, μ_sim_0=μ_sim_0_1, Σ_sim_0=Σ_sim_0,
                 diff=False, name=r'Hard-working worker')
simulate_workers(worker, T, ax, μ_sim_0=μ_sim_0_2, Σ_sim_0=Σ_sim_0,
                 diff=False, name=r'Normal worker')
ax.legend(bbox_to_anchor=(1, 0.5))
plt.show()
```

Both inferred paths start from the firm's common prior $\hat{u}_0 = 1$ and climb toward their different true values, showing that the filter corrects the gap between prior and truth as evidence accumulates.

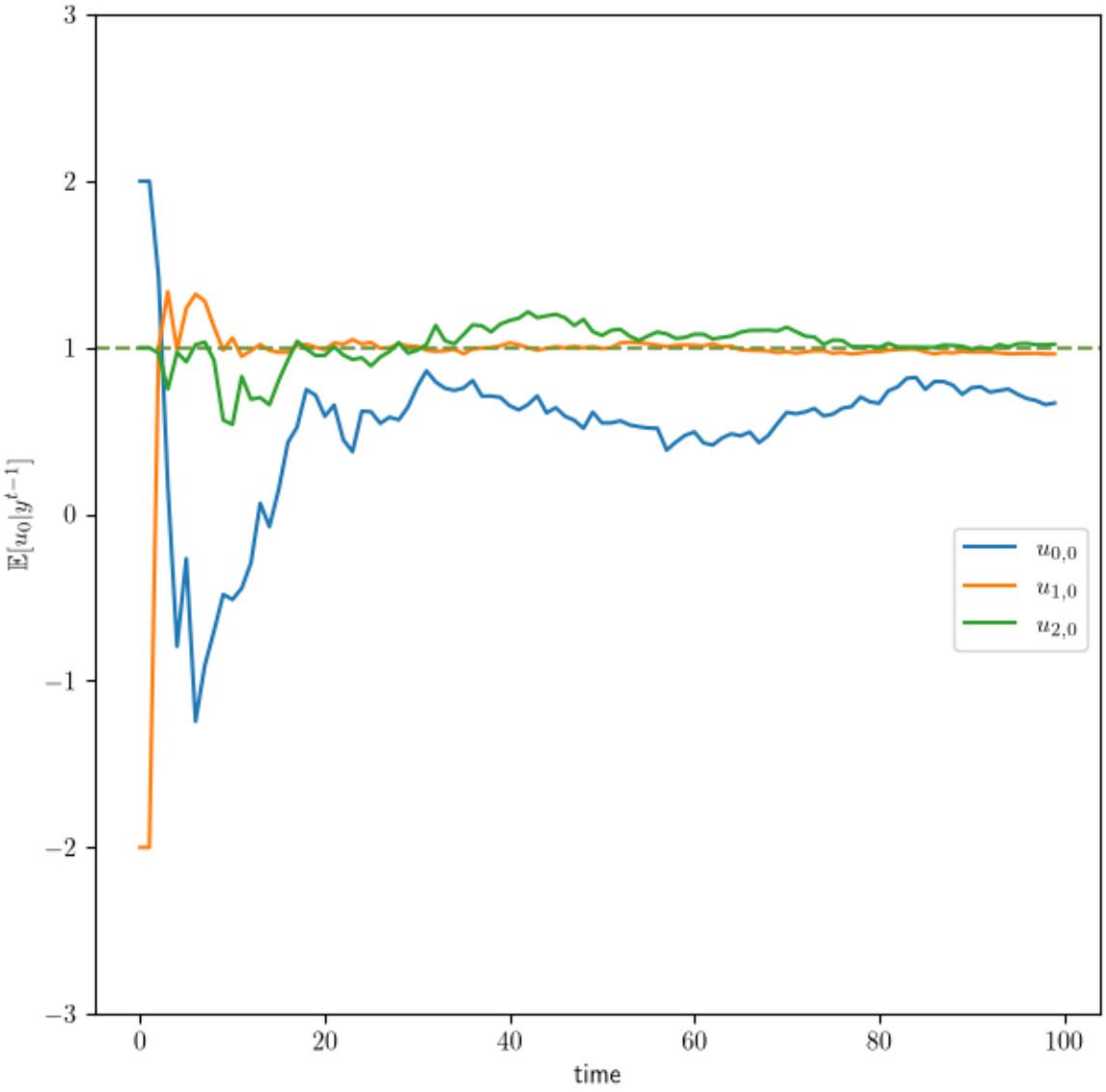


Fig. 46.6: Inferred work ethic when every worker starts at $h_0 = 2$ and $u_0 = 1$

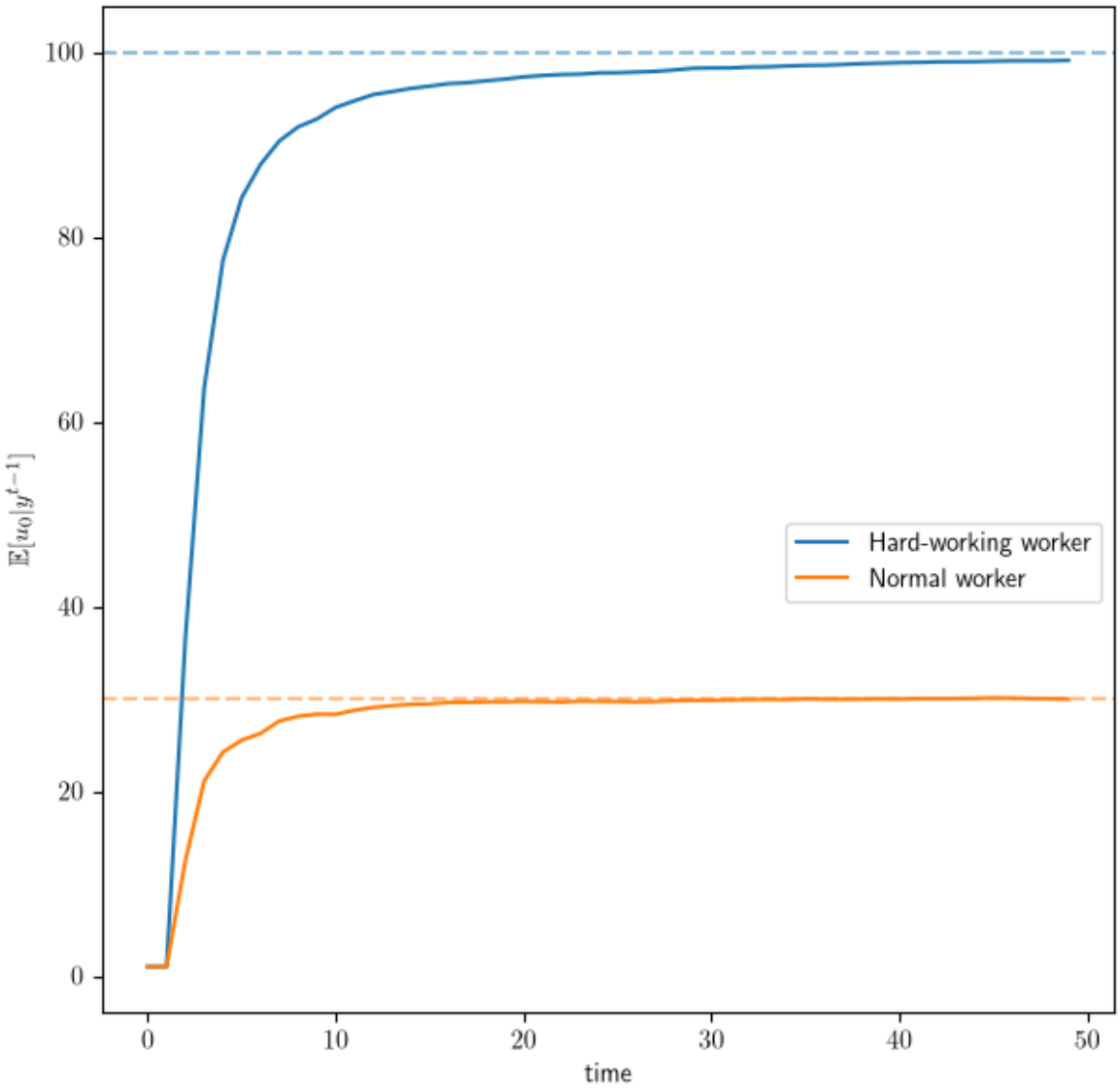


Fig. 46.7: A hard-working worker and a less hard-working worker

46.6 Future extensions

We can do lots of enlightening experiments by creating new types of workers and letting the firm learn about their hidden (to the firm) states by observing just their output histories.

CHAPTER

FORTYSEVEN

THE KALMAN FILTER AND VECTOR AUTOREGRESSIONS

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install quantecon
```

47.1 Overview

This lecture derives the **Kalman filter** for a linear Gaussian state space system and then uses it to construct **vector autoregressions (VARs)**.

It builds on *A First Look at the Kalman Filter*, where the filter is introduced through filtering and forecasting distributions.

It complements *Another Look at the Kalman Filter*, where the same recursion is used in a worker-firm learning problem.

This lecture shifts attention from filtering a hidden state to representing and estimating the observable process generated by that hidden state.

Our approach rests on repeated applications of the **population linear least squares** projection formula, the insight that computing a conditional expectation of a jointly normal random vector is the same as running a population OLS regression.

The lecture covers:

- deriving the Kalman filter recursions from first principles
- the **matrix Riccati difference equation** governing conditional covariance matrices
- the **innovations representation** and the **Gram-Schmidt** whitening property
- the structure of a **hidden Markov model**
- the **likelihood function** for a state space system and its role in maximum likelihood and Bayesian estimation
- how the time-invariant Kalman filter generates a **vector autoregression**
- why the Kalman filter is an essential tool for *interpreting VARs* estimated from economic data

47.2 The state space system

The Kalman filter applies to the **state space system** for $t \geq 0$:

$$\begin{aligned} x_{t+1} &= Ax_t + Cw_{t+1} \\ y_t &= Gx_t + v_t \end{aligned} \tag{47.1}$$

where

- x_t is an $n \times 1$ **state vector** (hidden, unobserved)
- y_t is an $m \times 1$ vector of **signals** on the hidden state (observed)
- w_{t+1} is a $p \times 1$ IID sequence of normal random variables with mean 0 and identity covariance matrix
- v_t is an IID sequence of normal random variables with mean zero and covariance matrix R
- w_{t+1} and v_s are orthogonal for all $t+1$ and $s \geq 0$

The coefficient matrices have the following dimensions: A is $n \times n$, C is $n \times p$, G is $m \times n$, and R is $m \times m$.

The initial state satisfies

$$x_0 \sim N(\hat{x}_0, \Sigma_0) \tag{47.2}$$

We observe $y_t, \dots, y_0$ but *not* $x_t, \dots, x_0$ at time t, and we know all first and second moments implied by (47.1) and (47.2).

47.3 The Kalman filter

47.3.1 Starting distribution

Working forward in time starting at $t = 0$, before observing y_0, the specification (47.1)-(47.2) implies that the marginal distribution of y_0 is

$$y_0 \sim N(G\hat{x}_0,\ G\Sigma_0 G^\top + R) \tag{47.3}$$

For $t \geq 0$ let $y^t = [y_t, y_{t-1}, \dots, y_0]$.

We want a convenient recursive representation of the conditional distribution of y_t given history y^{t-1}.

The Kalman filter attains this by constructing recursive formulas for $\hat{x}_t$ and Σ_t such that the distribution of y_t conditional on y^{t-1} generalises (47.3) to

$$y_t \sim N(G\hat{x}_t,\ G\Sigma_t G^\top + R) \tag{47.4}$$

for $t \geq 1$, where the distribution of x_t conditional on y^{t-1} is $N(\hat{x}_t, \Sigma_t)$.

The objects $\hat{x}_t$ and Σ_t characterise the **population regression**

$$\hat{x}_t = \mathbb{E}[x_t \mid y_{t-1}, \dots, y_0]$$

and the **conditional covariance matrix**

$$\Sigma_t = \mathbb{E}[(x_t - \hat{x}_t)(x_t - \hat{x}_t)^\top \mid y_{t-1}, \dots, y_0]$$

47.3.2 Derivation

At each date, our approach is to *regress what we do not know on what we know*.

Note

Because our assumptions imply that $\{x_t, y_t\}_{t=0}^{\infty}$ is a jointly normal stochastic process, linear least squares regressions equal conditional mathematical expectations.

Each step below is an application of Bayes' law.

Under the weaker assumption that all means and covariances exist without joint normality, the same calculations yield "wide-sense conditional expectations" that coincide with true conditional expectations only when those conditional expectations are linear.

We arrive at $t = 0$ knowing $\hat{x}_0$ and Σ_0.

The information about x_0 in y_0 that is new relative to $(\hat{x}_0, \Sigma_0)$ is the **innovation**

$$a_0 \equiv y_0 - G\hat{x}_0$$

Let L_0 be the population regression coefficient of the hidden-state error $x_0 - \hat{x}_0$ on the signal surprise $y_0 - G\hat{x}_0$.

The conditional mean $\mathbb{E}[x_0 \mid y_0] = \hat{x}_0 + L_0(y_0 - G\hat{x}_0)$ satisfies the population regression formula

$$x_0 - \hat{x}_0 = L_0(y_0 - G\hat{x}_0) + \eta \tag{47.5}$$

where η is the least squares residual.

Orthogonality of η to $(y_0 - G\hat{x}_0)$ pins down L_0 via the normal equations

$$\mathbb{E}(x_0 - \hat{x}_0)(y_0 - G\hat{x}_0)^\top = L_0 \, \mathbb{E}(y_0 - G\hat{x}_0)(y_0 - G\hat{x}_0)^\top$$

Evaluating the moment matrices and solving for L_0 gives

$$L_0 = \Sigma_0 G^\top (G\Sigma_0 G^\top + R)^{-1} \tag{47.6}$$

Thus L_0 updates the estimate of x_0, while $K_0 = AL_0$ updates the forecast of x_1.

To forecast x_1, note that

$$x_1 = A\hat{x}_0 + A(x_0 - \hat{x}_0) + Cw_1 \tag{47.7}$$

Applying (47.5) gives $\mathbb{E}[x_1 \mid y_0] = A\hat{x}_0 + AL_0(y_0 - G\hat{x}_0)$, which we write as

$$\hat{x}_1 = A\hat{x}_0 + K_0(y_0 - G\hat{x}_0) \tag{47.8}$$

where the **Kalman gain** at time 0 is

$$K_0 = A\Sigma_0 G^\top (G\Sigma_0 G^\top + R)^{-1} \tag{47.9}$$

Subtracting (47.8) from (47.7) yields

$$x_1 - \hat{x}_1 = A(x_0 - \hat{x}_0) + Cw_1 - K_0(y_0 - G\hat{x}_0) \tag{47.10}$$

Using (47.10) and $y_0 = Gx_0 + v_0$ to evaluate $\Sigma_1 \equiv \mathbb{E}[(x_1 - \hat{x}_1)(x_1 - \hat{x}_1)^\top \mid y_0]$ gives

$$\Sigma_1 = (A - K_0 G)\Sigma_0 (A - K_0 G)^\top + CC^\top + K_0 R K_0^\top \tag{47.11}$$

Thus $f(x_1 \mid y_0) \sim N(\hat{x}_1, \Sigma_1)$.

Collecting the time-0 equations:

$$\begin{aligned} a_0 &= y_0 - G\hat{x}_0 \\ K_0 &= A\Sigma_0 G^\top (G\Sigma_0 G^\top + R)^{-1} \\ \hat{x}_1 &= A\hat{x}_0 + K_0 a_0 \\ \Sigma_1 &= CC^\top + K_0 R K_0^\top + (A - K_0 G)\Sigma_0 (A - K_0 G)^\top \end{aligned} \tag{47.12}$$

System (47.12) maps a mean-covariance pair $(\hat{x}_0, \Sigma_0)$ into a new pair $(\hat{x}_1, \Sigma_1)$, with auxiliary outputs (a_0, K_0).

Recognising that "we are in the same situation at the start of period 1 as at the start of period 0" activates a recursion, the **Kalman filter**.

47.3.3 The Kalman filter recursions

Iterating system (47.12) yields the Kalman filter for $t \geq 0$:

$$\begin{aligned} a_t &= y_t - G\hat{x}_t \\ K_t &= A\Sigma_t G^\top (G\Sigma_t G^\top + R)^{-1} \\ \hat{x}_{t+1} &= A\hat{x}_t + K_t a_t \\ \Sigma_{t+1} &= CC^\top + K_t R K_t^\top + (A - K_t G)\Sigma_t (A - K_t G)^\top \end{aligned} \tag{47.13}$$

Here K_t is the **Kalman gain** at time t.

47.3.4 The matrix Riccati equation

Substituting the expression for K_t from the second line of (47.13) into the fourth line gives an equivalent update formula:

$$\Sigma_{t+1} = A\Sigma_t A^\top + CC^\top - A\Sigma_t G^\top (G\Sigma_t G^\top + R)^{-1} G\Sigma_t A^\top \tag{47.14}$$

Equation (47.14) is the **matrix Riccati difference equation**.

It governs the sequence of conditional covariance matrices $\{\Sigma_t\}_{t=0}^{\infty}$ without reference to the observations $\{y_t\}$.

47.4 The Gram-Schmidt process

The random vector

$$a_t = y_t - \mathbb{E}[y_t \mid y_{t-1}, \ldots, y_0]$$

is the **innovation** of y_t with respect to y^{t-1}, the part of y_t that cannot be predicted from past observations.

Note that $\mathbb{E} a_t a_t^\top = G\Sigma_t G^\top + R$, the matrix whose inverse appears in the Kalman gain formula (47.13).

A direct calculation using $a_t = G(x_t - \hat{x}_t) + v_t$ shows that $\mathbb{E} a_t a_{t-1}^\top = 0$ and, more generally, $\mathbb{E}[a_t \mid a_{t-1}, \ldots, a_0] = 0$.

Note

An alternative argument from first principles: let $H(y^t)$ denote the closed linear span of y^t.

Since $a_{t+1} = y_{t+1} - \mathbb{E}[y_{t+1} \mid y^t]$ is a least-squares error, $a_{t+1} \perp H(y^t)$, and in particular $a_{t+1} \perp a_t$.

Thus $\{a_t\}$ is a white-noise process of innovations to $\{y_t\}$.

Sometimes (47.13) is called a **whitening filter**: it takes the signal process $\{y_t\}$ as input and produces the white-noise innovation process $\{a_t\}$ as output.

With $H(a^t)$ defined analogously, the linear space $H(a^t)$ is an orthogonal basis for the linear space $H(y^t)$.

Rather than computing $\mathbb{E}[x_t \mid y_{t-1}, \ldots, y_0]$ via one large regression, the Kalman filter performs a sequence of small regressions on successive orthogonal components of the basis $[a_{t-1}, \ldots, a_0]$, an instance of the **Gram-Schmidt procedure**.

47.5 Hidden Markov model

System (47.1)-(47.2) is an example of a **hidden Markov model**.

The observed process $\{y_t\}_{t=0}^{\infty}$ is *not* Markov, but the hidden process $\{x_t\}_{t=0}^{\infty}$ *is* Markov.

So is the process of means and covariances $\{(\hat{x}_t, \Sigma_t)\}$, which are sufficient statistics for the distribution of x_t conditional on $[y_{t-1}, \ldots, y_0]$.

47.6 Estimation

47.6.1 The innovations representation

The **innovations representation** emerging from the Kalman filter is

$$\begin{aligned} \hat{x}_{t+1} &= A\hat{x}_t + K_t a_t \\ y_t &= G\hat{x}_t + a_t \end{aligned} \tag{47.15}$$

where $\hat{x}_t = \mathbb{E}[x_t \mid y^{t-1}]$ for $t \geq 1$ and $\mathbb{E}[a_t a_t^\top \mid y^{t-1}] = G\Sigma_t G^\top + R \equiv \Omega_t$.

For $t \geq 1$, $\mathbb{E}[y_t \mid y^{t-1}] = G\hat{x}_t$ and the conditional distribution of y_t given y^{t-1} is $N(G\hat{x}_t, \Omega_t)$.

The objects $(G\hat{x}_t, \Omega_t)$ emerging from the Kalman filter recursions therefore completely characterise this conditional distribution.

47.6.2 The likelihood function

We can factor the likelihood of a sample $(y_T, y_{T-1}, \ldots, y_0)$ as

$$f(y_T, \ldots, y_0) = f(y_T \mid y^{T-1})\, f(y_{T-1} \mid y^{T-2}) \cdots f(y_1 \mid y_0)\, f(y_0) \tag{47.16}$$

The log conditional density of the $m \times 1$ vector y_t is

$$\log f(y_t \mid y^{t-1}) = -\frac{m}{2}\log(2\pi) - \frac{1}{2}\log\det(\Omega_t) - \frac{1}{2}\, a_t^\top \Omega_t^{-1} a_t \tag{47.17}$$

Using (47.17) and (47.13) together, we can evaluate the likelihood (47.16) recursively for any parameter vector θ that underlies the matrices A, G, C, R.

Such calculations are at the heart of efficient strategies for computing **maximum likelihood estimators** of free parameters.

47.6.3 Bayesian inference

The likelihood function is also central to **Bayesian inference**.

Where θ is the parameter vector, y_0^T the data, and $\tilde{p}(\theta)$ a prior density over θ before seeing y_0^T, Bayes' law gives the **posterior**

$$\tilde{p}(\theta \mid y_0^T) = \frac{f(y_0^T \mid \theta)\,\tilde{p}(\theta)}{\int f(y_0^T \mid \theta)\,\tilde{p}(\theta)\,d\theta}$$

The denominator is the marginal joint density $f(y_0^T)$.

47.7 Vector autoregressions and the Kalman filter

47.7.1 Convergence to a steady state

Under conditions discussed by Anderson *et al.* [1996], iterations on the Riccati equation (47.14) converge to a **time-invariant** matrix Σ from any positive semi-definite starting value Σ_0.

A time-invariant fixed point $\Sigma_t = \Sigma$ of (47.14) is the covariance matrix of x_t around

$$\mathbb{E}[x_t \mid \{y_s\}_{s \le t-1}]$$

where the conditioning extends over the **semi-infinite** past $s \le t-1$.

47.7.2 A time-invariant VAR

If the fixed point Σ exists and we initialise the filter at $\Sigma_0 = \Sigma$, the innovations representation (47.15) becomes time-invariant:

$$\begin{aligned} \hat{x}_{t+1} &= A\hat{x}_t + Ka_t \\ y_t &= G\hat{x}_t + a_t \end{aligned} \tag{47.18}$$

where $\mathbb{E}a_t a_t^\top = G\Sigma G^\top + R$ and the **steady-state Kalman gain** is $K = A\Sigma G^\top (G\Sigma G^\top + R)^{-1}$.

From (47.18) we obtain $\hat{x}_{t+1} = (A - KG)\hat{x}_t + Ky_t$.

If the eigenvalues of $A - KG$ are bounded in modulus strictly below unity, we can solve this equation forward to get

$$\hat{x}_{t+1} = \sum_{j=0}^{\infty} (A - KG)^j K\, y_{t-j} \tag{47.19}$$

Substituting (47.19) into the observation equation of (47.18) gives the **vector autoregression**

$$y_t = G \sum_{j=0}^{\infty} (A - KG)^j K\, y_{t-j-1} + a_t \tag{47.20}$$

where by construction

$$\mathbb{E}\left[a_t\, y_{t-j-1}^\top\right] = 0 \quad \forall\, j \ge 0 \tag{47.21}$$

The orthogonality conditions (47.21) identify (47.20) as a vector autoregression.

Defining the lag operator L by $Lx_{t+1} \equiv x_t$, the **moving average representation** deduced from (47.18) is

$$y_t = \left[I + G(I - AL)^{-1}KL\right] a_t = \left[I + G\sum_{j=0}^{\infty} A^j K L^{j+1}\right] a_t$$

47.7.3 Interpreting VARs

Equilibria of economic models (or their linear or log-linear approximations) typically take the form of state space system (47.1).

This hidden Markov model disturbs the state x_t by the $p \times 1$ shock vector w_{t+1} and perturbs the $m \times 1$ vector of observables y_t by the $m \times 1$ measurement error v_t.

An economic theory typically makes w_{t+1} and v_t directly interpretable as shocks to preferences, technologies, endowments, or information sets.

The state space system (47.1) represents $\{y_t\}$ in terms of these **interpretable shocks**.

However, in the typical situation these shocks *cannot* be recovered directly from the y_t's, even when A, G, C, R are known.

The innovations representation (47.18) represents the *same* stochastic process $\{y_t\}$ in terms of the $m \times 1$ vector a_t of innovations that would be recovered by running an infinite-order population vector autoregression.

Its role in mapping the original representation (47.1) to the VAR (47.20) makes the Kalman filter an indispensable tool for *interpreting vector autoregressions*.

47.8 Spectral factorization identity

Because the original state space system (47.1) and the innovations representation (47.18) describe the *same* stochastic process $\{y_t\}$, they imply two distinct formulas for the **spectral density matrix** of $\{y_t\}$.

Equating those formulas yields the *spectral factorization identity*.

47.8.1 Two representations of the spectral density

Consider first the original state space system.

Writing the first line of (47.1) as $x_t = (zI - A)^{-1} C w_{t+1}$ (using the z-transform convention $z^{-1} x_t = x_{t-1}$), the covariance generating function of $\{x_t\}$ is

$$S_x(z) = (zI - A)^{-1} C C^\top (z^{-1} I - A^\top)^{-1}.$$

Since v_t is orthogonal to x_t, the spectral density of $\{y_t\}$ is

$$S_y(z) = G(zI - A)^{-1} C C^\top (z^{-1} I - A^\top)^{-1} G^\top + R. \tag{47.22}$$

Now consider the innovations representation.

The time-invariant innovations representation (47.18) gives $y_t = [G(zI - A)^{-1} K + I]\, a_t$.

Since a_t is white noise with covariance matrix $G \Sigma G^\top + R$, the spectral density is also

$$S_y(z) = \big[G(zI - A)^{-1} K + I\big](G \Sigma G^\top + R)\big[K^\top (z^{-1} I - A^\top)^{-1} G^\top + I\big]. \tag{47.23}$$

47.8.2 The spectral factorization identity

Equating (47.22) and (47.23) gives the **spectral factorization identity**:

$$G(zI-A)^{-1}CC^\top(z^{-1}I-A^\top)^{-1}G^\top + R = \big[G(zI-A)^{-1}K + I\big](G\Sigma G^\top + R)\big[K^\top(z^{-1}I-A^\top)^{-1}G^\top + I\big]. \tag{47.24}$$

The left side expresses $S_y(z)$ in terms of the **structural shocks** (w_{t+1}, v_t) and the matrices (A, C, G, R).

The right side expresses the same object as a spectral factor built from the innovations a_t and the steady-state Kalman gain K.

47.8.3 Wold and autoregressive representations

Starting from the innovations representation (47.18), we can obtain both a Wold moving average representation and an autoregressive representation.

For the Wold representation, iterate the state equation in (47.18) to express the current observation in terms of current and past innovations.

With L denoting the lag operator, (47.18) implies

$$y_t = \big[I + G(I-AL)^{-1}KL\big]\, a_t \tag{47.25}$$

This is the Wold moving-average representation of $\{y_t\}$ in terms of its one-step forecast errors.

For the autoregressive representation, invert the moving-average operator in (47.25) and solve for a_t in terms of current and past observations.

Using the identity

$$\big[I + G(I-AL)^{-1}KL\big]^{-1} = I - G\big[I-(A-KG)L\big]^{-1}KL$$

gives

$$y_t = G\big[I-(A-KG)L\big]^{-1}K\, y_{t-1} + a_t = \sum_{j=1}^{\infty} G(A-KG)^{j-1}K\, y_{t-j} + a_t, \tag{47.26}$$

which is the vector autoregression already stated in (47.20).

The key analytical fact is that, under mild stability conditions, the zeros of $\det[G(zI-A)^{-1}K + I]$ all lie *inside* the unit circle.

This ensures that the moving-average operator in (47.25) has a causal one-sided inverse.

Hence a_t lies in the closed linear span of current and past observations y^t, so a_t is the population forecast error in the VAR.

47.9 Python implementation

We now illustrate the theory using the `quantecon` library, which provides `LinearStateSpace` and `Kalman` classes that implement everything derived above.

We use the following imports:

```
import numpy as np
import matplotlib.pyplot as plt
import quantecon as qe
```

47.9.1 A scalar hidden AR(1) model

Consider a scalar hidden AR(1) state observed with measurement noise:

$$x_{t+1} = \rho\, x_t + \sigma_w\, w_{t+1}$$
$$y_t = x_t + \sigma_v\, v_t$$

with $w_t, v_t \sim N(0,1)$ IID.

Here ρ is the persistence parameter, while σ_w and σ_v are shock standard deviations.

The `LinearStateSpace` class parameterizes measurement noise by a matrix H such that $R = HH^\top$.

```
# Model parameters
ρ = 0.9
σ_w = 0.5
σ_v = 1.0

# State-space matrices
A = np.array([[ρ]])
C = np.array([[σ_w]])
G = np.array([[1.0]])
R = np.array([[σ_v**2]])

# Build a LinearStateSpace and a Kalman filter object
H = np.array([[σ_v]])   # measurement noise factor: R = H @ H.T
lss = qe.LinearStateSpace(
  A, C, G, H, mu_0=np.zeros(1), Sigma_0=np.eye(1) * 10.0)
kf = qe.Kalman(lss)
kf.set_state(np.zeros(1), np.eye(1) * 10.0)  # diffuse prior
```

We first simulate a sample path of the true hidden state and noisy observations.

```
T = 200
x_path, y_path = lss.simulate(ts_length=T, random_state=42)

# Shapes: x_path is (n, T+1), y_path is (m, T)
x_true = x_path[0, :T]
y_obs = y_path[0, :]
```

We then run the Kalman filter manually, step by step, to collect the filtered estimates.

```
x_hats = np.zeros(T)
Sigmas = np.zeros(T)

for t in range(T):
    kf.update(y_obs[t:t+1])          # one full filter cycle
    x_hats[t] = kf.x_hat.item()
    Sigmas[t] = kf.Sigma.item()
```

```
fig, axes = plt.subplots(3, 1, figsize=(10, 8), sharex=True)
```

(continues on next page)

(continued from previous page)

```
t_range = np.arange(T)

axes[0].plot(t_range, x_true, lw=2,
        label='true state $x_t$')
axes[0].plot(t_range, x_hats, lw=2,
        linestyle='--', label=r'$\hat{x}_t$ (Kalman)')
axes[0].plot(t_range, y_obs, alpha=0.35, lw=2, label='observation $y_t$')
axes[0].set_title('state and observation')
axes[0].legend(fontsize=9)

axes[1].plot(t_range, Sigmas, color='C1', lw=2,
             label=r'conditional variance $\Sigma_t$')
axes[1].axhline(kf.Sigma_infinity[0, 0], ls='--', color='k',
                label=r'steady-state $\Sigma_\infty$')
axes[1].set_title('conditional variance')
axes[1].legend(fontsize=9)

axes[2].plot(t_range, y_obs - x_hats, color='C2', lw=2, alpha=0.7,
             label=r'innovation $a_t = y_t - G\hat{x}_t$')
axes[2].set_title('innovation')
axes[2].set_xlabel('time $t$')
axes[2].legend(fontsize=9)
fig.tight_layout()
plt.show()
```

After a short adjustment, the Kalman estimate follows the hidden state much more closely than the raw noisy observations.

The conditional variance drops quickly from the diffuse prior and then settles onto its steady-state value.

The innovation series fluctuates around zero, as a one-step forecast error should.

47.9.2 Convergence of the Riccati equation

The `Kalman` class computes the steady-state covariance Σ_∞ by solving the discrete algebraic Riccati equation directly.

```
Sigma_inf, K_inf = kf.stationary_values()

print(f"Steady-state covariance  Σ_inf = {Sigma_inf[0, 0]:.6f}")
print(f"Kalman filter converged to Σ_t = {Sigmas[-1]:.6f}")
print(f"Steady-state Kalman gain K  = {K_inf[0, 0]:.6f}")

A_minus_KG = A - K_inf @ G
eigval = np.linalg.eigvals(A_minus_KG)[0]
print(f"\nEigenvalue of (A - KG)      = {eigval:.6f}")
print(f"Stable VAR: {np.abs(eigval) < 1}")
```

```
Steady-state covariance  Σ_inf = 0.530899
Kalman filter converged to Σ_t = 0.530899
Steady-state Kalman gain K  = 0.312110

Eigenvalue of (A - KG)      = 0.587890
Stable VAR: True
```

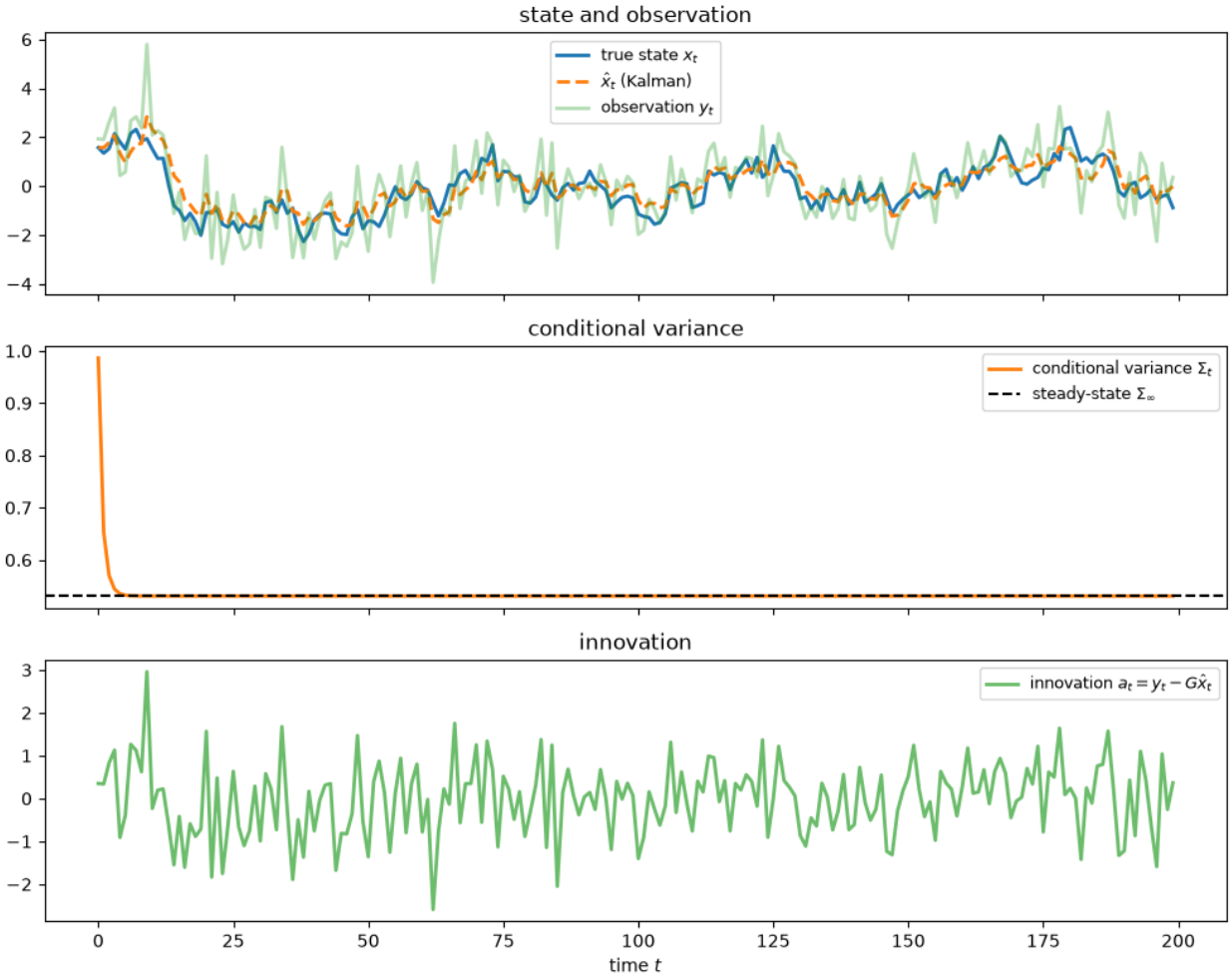


Fig. 47.1: Scalar Kalman filtering

47.9.3 The VAR representation

Using (47.20), the coefficients in the infinite-order VAR representation are $G(A - KG)^j K$ for $j = 0, 1, 2, ...$

We retrieve them via `stationary_coefficients`:

```
J = 30
var_coeffs = kf.stationary_coefficients(J, coeff_type='var')

# Lag j+1 coefficient matrices
lags = np.arange(1, J + 1)
coeff_values = np.array([var_coeffs[j][0, 0] for j in range(J)])

fig, ax = plt.subplots()
ax.stem(lags, coeff_values, basefmt=' ')
ax.set_xlabel('lag $j$')
ax.set_ylabel(r'VAR coefficient $G(A{-}KG)^{j-1}K$')
fig.tight_layout()
plt.show()
```

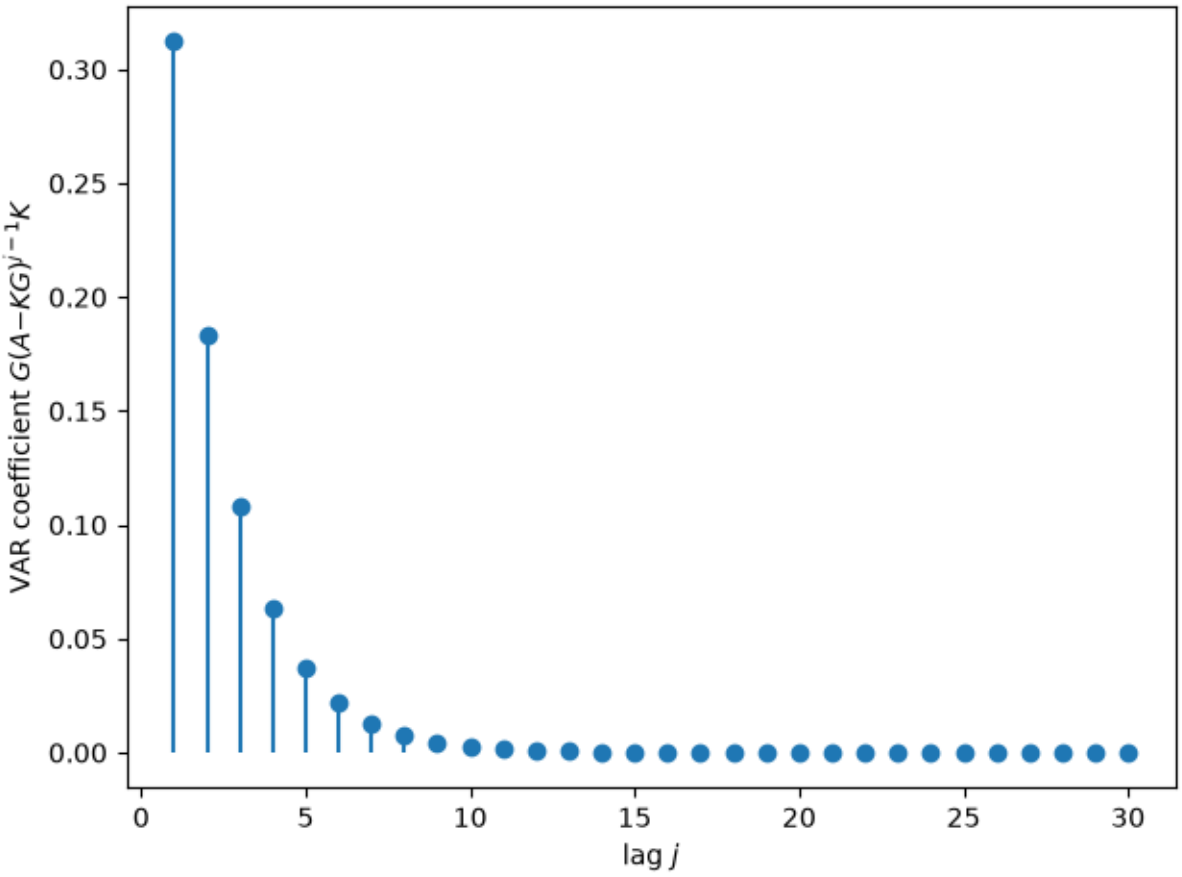


Fig. 47.2: VAR coefficients from the innovations representation

Most of the autoregressive weight is concentrated in the first few lags.

The later coefficients are nearly zero, so a short finite-lag VAR captures most of the infinite-order representation in this

example.

47.9.4 Likelihood evaluation

We use (47.17) to evaluate the log-likelihood of the simulated sample.

```
def log_likelihood(A, C, G, R, y_data, x_hat_0, Sigma_0):
    """Evaluate the log-likelihood using the Kalman filter recursions."""
    H_ = np.linalg.cholesky(R)    # R = H_ @ H_.T
    lss_ = qe.LinearStateSpace(A, C, G, H_, mu_0=x_hat_0, Sigma_0=Sigma_0)
    kf_ = qe.Kalman(lss_)
    kf_.set_state(x_hat_0, Sigma_0)

    T_, m_ = y_data.shape
    loglik = 0.0

    for t in range(T_):
        x_h = kf_.x_hat
        Sig = kf_.Sigma
        Omega = G @ Sig @ G.T + R        # innovation covariance
        a_t = y_data[t] - (G @ x_h).flatten()

        sign, logdet = np.linalg.slogdet(Omega)
        loglik += -0.5 * (m_ * np.log(2 * np.pi) + logdet
                          + float(a_t @ np.linalg.solve(Omega, a_t)))
        kf_.update(y_data[t])

    return loglik


y_data_col = y_obs.reshape(-1, 1)
ll = log_likelihood(A, C, G, R,
                    y_data_col,
                    np.zeros(1), np.eye(1) * 10.0)
print(f"Log-likelihood of sample: {ll:.4f}")
```

```
Log-likelihood of sample: -325.2335
```

47.10 An example

We now work through a structured example that shows how a bivariate VAR(2) fits naturally into the state space framework and how the Kalman filter delivers a Wold (innovations) representation.

47.10.1 A linear state-space system and its filter

The state and observation equations are

$$x_{t+1} = Ax_t + Cw_{t+1} \tag{47.27}$$

$$y_t = Gx_t + v_t \tag{47.28}$$

with initial condition and shock distributions

$$x_0 \sim N(\hat{x}_0, \Sigma_0), \quad w_{t+1} \sim N(0, I), \quad v_t \sim N(0, R).$$

The steady-state error covariance matrix Σ satisfies the Riccati equation

$$\Sigma = A\Sigma A^\top + CC^\top - A\Sigma G^\top \left(G\Sigma G^\top + R\right)^{-1} G\Sigma A^\top \tag{47.29}$$

and the associated steady-state Kalman gain is

$$K = A\Sigma G^\top \left(G\Sigma G^\top + R\right)^{-1} \tag{47.30}$$

Starting from an initial estimate $\hat{x}_0$, the Kalman filter updates the state estimate via

$$\hat{x}_{t+1} = A\hat{x}_t + Ka_t \tag{47.31}$$

where the innovation is

$$a_t = y_t - G\hat{x}_t \tag{47.32}$$

Substituting (47.32) into (47.31) and expanding:

$$\hat{x}_{t+1} = A\hat{x}_t + K(y_t - G\hat{x}_t) = (A - KG)\hat{x}_t + Ky_t = (A - KG)\hat{x}_t + KGx_t + Kv_t \tag{47.33}$$

47.10.2 Impulse responses of y_t to the innovations a_t

It is useful to compute the **ordinary impulse response functions** of the observable vector y_t to its own innovations a_t, the moving-average (Wold) representation that is the mirror image of the VAR (47.20).

From the time-invariant innovations representation (47.18)

$$\hat{x}_{t+1} = A\hat{x}_t + Ka_t, \qquad y_t = G\hat{x}_t + a_t,$$

the moving-average representation (47.25) is

$$y_t = \left[I + G(I - AL)^{-1}KL\right] a_t = a_t + \sum_{h=1}^{\infty} GA^{h-1}K\, a_{t-h}.$$

Hence the impulse response of y_t to a unit innovation a_t is

$$\Psi_0 = I, \qquad \Psi_h = GA^{h-1}K \quad (h \geq 1). \tag{47.34}$$

These coefficients decay at the rate governed by the eigenvalues of A.

We can read the coefficients (47.34) directly off a `quantecon` `LinearStateSpace` object.

We build a state-space system whose state is the filtered estimate $\hat{x}_t$, whose single "shock" is the innovation a_t loaded through $C = K$, and whose observation matrix is G.

The `impulse_response` method of that object returns the sequence GA^jK for $j = 0, 1, 2, \ldots$, which are exactly the Ψ_h for $h \geq 1$; we prepend $\Psi_0 = I$ to capture the contemporaneous feed-through $y_t = G\hat{x}_t + a_t$.

The array returned below has entry `[h, i, j]` equal to the response of observable `i` at horizon `h` to innovation component `j`.

```
def y_to_a_irf(A, K, G, T=40):
    """
    Return Wold IRFs of y_t to its own innovations a_t.
    """
    n, m = A.shape[0], G.shape[0]
    lss = qe.LinearStateSpace(A, K, G, np.zeros((m, m)), mu_0=np.zeros(n))
    _, ycoef = lss.impulse_response(j=T - 2)       # [GK, GAK, GA^2K, ...]
    Psi = np.empty((T, m, m))
    Psi[0] = np.eye(m)                             # contemporaneous response
    for h in range(1, T):
        Psi[h] = ycoef[h - 1]
    return Psi
```

47.10.3 Bivariate VAR(2) in state-space form

Consider two observable series r_t and z_t.

Stack them into the state vector $x_t = (r_t,\ r_{t-1},\ z_t,\ z_{t-1})^\top$.

We posit the VAR(2) state-transition equation:

$$\begin{pmatrix} r_{t+1} \\ r_t \\ z_{t+1} \\ z_t \end{pmatrix} = \begin{pmatrix} d_1 & d_2 & d_3 & d_4 \\ 1 & 0 & 0 & 0 \\ \delta_1 & \delta_2 & \delta_3 & \delta_4 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} r_t \\ r_{t-1} \\ z_t \\ z_{t-1} \end{pmatrix} + \begin{pmatrix} c_{11} & c_{12} \\ 0 & 0 \\ c_{21} & c_{22} \\ 0 & 0 \end{pmatrix} \begin{pmatrix} w_{1,t+1} \\ w_{2,t+1} \end{pmatrix} \tag{47.35}$$

We consider two possible observation equations.

The first is a bivariate observation of r_t and z_t:

$$\begin{pmatrix} r_t \\ z_t \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} r_t \\ r_{t-1} \\ z_t \\ z_{t-1} \end{pmatrix} + \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} v_{1t} \\ v_{2t} \end{pmatrix} \tag{47.36}$$

The second is a univariate observation of r_t:

$$y_t = \begin{pmatrix} 1 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} r_t \\ r_{t-1} \\ z_t \\ z_{t-1} \end{pmatrix} + v_{1t} \tag{47.37}$$

We now compare the Wold impulse responses generated by these two observation systems.

System 1 observes both r_t and z_t, so its innovation a_t is 2×1.

System 2 observes only r_t, so its innovation u_t is scalar.

The transition matrices are the same in the two systems, but the observation matrix changes.

Consequently, the steady-state Kalman gain changes too, and so do the Wold responses of the observables to their own innovations.

47.10.4 Numerical example: impulse responses to innovations

The parameter values are:

$$
\begin{aligned}
d_1 &= 0.80, \quad d_2 = 0.05, \quad d_3 = 0.75, \quad d_4 = -0.72 \\
\delta_1 &= 0.00, \quad \delta_2 = 0.00, \quad \delta_3 = 0.75, \quad \delta_4 = 0.20 \\
c_{11} &= 1.0, \quad c_{12} = 0.0, \quad c_{21} = 0.0, \quad c_{22} = 1.0 \\
R &= 0.0001 \times I_2 \quad \text{(bivariate case)}, \qquad R = 0.0001 \quad \text{(univariate case)}.
\end{aligned}
$$

These give the 4×4 transition matrix and 4×2 shock-loading matrix

$$
A = \begin{pmatrix} 0.80 & 0.05 & 0.75 & -0.72 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0.75 & 0.20 \\ 0 & 0 & 1 & 0 \end{pmatrix}, \qquad C = \begin{pmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix}.
$$

System 1 uses the bivariate observation equation (47.36), so G selects $(r_t, z_t)^\top$ from the state and the innovation a_t is 2×1.

System 2 uses the univariate observation equation (47.37), so the row vector in that equation selects only r_t and the innovation u_t is scalar.

```
# Parameters
d1, d2, d3, d4 = 0.80, 0.05, 0.75, -.72
δ1, δ2, δ3, δ4 = 0.00, 0.00, 0.75, 0.20
c11, c12, c21, c22 = 1.0,  0.0,  0.0,  1.0
σ_v = 0.01  # sqrt(0.0001)

# Shared matrices
A_var = np.array([[d1,     d2,     d3,     d4    ],
                  [1.0,    0.0,    0.0,    0.0   ],
                  [δ1, δ2, δ3, δ4],
                  [0.0,    0.0,    1.0,    0.0   ]])

C_var = np.array([[c11, c12],
                  [0.0, 0.0],
                  [c21, c22],
                  [0.0, 0.0]])

# System 1: bivariate observation
G_biv = np.array([[1.0, 0.0, 0.0, 0.0],
                  [0.0, 0.0, 1.0, 0.0]])
H_biv = σ_v * np.eye(2)           # H @ H.T = 0.0001 * I_2

lss_biv = qe.LinearStateSpace(A_var, C_var, G_biv, H_biv,
                              mu_0=np.zeros(4), Sigma_0=np.eye(4))
kf_biv = qe.Kalman(lss_biv)
_, K_biv = kf_biv.stationary_values()

print("System 1 - steady-state Kalman gain K (4x2):")
print(np.round(K_biv, 5))

# System 2: univariate observation
G_uni = np.array([[1.0, 0.0, 0.0, 0.0]])
H_uni = np.array([[σ_v]])          # H @ H.T = 0.0001

lss_uni = qe.LinearStateSpace(A_var, C_var, G_uni, H_uni,
```

(continues on next page)

(continued from previous page)

```
                                mu_0=np.zeros(4), Sigma_0=np.eye(4))
kf_uni = qe.Kalman(lss_uni)
_, K_uni = kf_uni.stationary_values()

print("\nSystem 2 - steady-state Kalman gain K (4x1):")
print(np.round(K_uni, 5))
```

```
System 1 - steady-state Kalman gain K (4x2):
[[7.9987e-01 7.4987e-01]
 [9.9990e-01 0.0000e+00]
 [1.0000e-05 7.4994e-01]
 [0.0000e+00 9.9990e-01]]

System 2 - steady-state Kalman gain K (4x1):
[[0.72306]
 [0.99994]
 [0.31829]
 [0.30984]]
```

We now apply the helper `y_to_a_irf` defined above to compute the ordinary impulse responses (47.34) of the observable y_t to its own innovations a_t, for both System 1 (bivariate, so a_t is 2×1) and System 2 (univariate, so u_t is scalar).

```
T_irf = 40
horizons = np.arange(T_irf)

Psi_biv = y_to_a_irf(A_var, K_biv, G_biv, T_irf)    # System 1: (T, 2, 2)
Psi_uni = y_to_a_irf(A_var, K_uni, G_uni, T_irf)    # System 2: (T, 1, 1)

obs_labels = [r'$r_t$', r'$z_t$']
innov_labels = [r'$a_{1,t}$', r'$a_{2,t}$']

# System 1 responses
fig, axes = plt.subplots(2, 2, figsize=(10, 6), sharex=True)
for i, obs in enumerate(obs_labels):
    for j, inn in enumerate(innov_labels):
        ax = axes[i, j]
        ax.plot(horizons, Psi_biv[:, i, j], lw=2)
        ax.axhline(0, color='k', lw=0.6, ls='--')
        ax.set_title(fr'{obs} to {inn}', fontsize=9)
        if i == 1:
            ax.set_xlabel('horizon $h$')
        if j == 0:
            ax.set_ylabel('response')
fig.tight_layout()
plt.show()
```

Own innovations move their own observables one-for-one on impact and then fade.

The diagonal panels start at one and the off-diagonal panels start at zero because $\Psi_0 = I$.

The cross response of r_t to $a_{2,t}$ is sizeable at short horizons, while the response of z_t to $a_{1,t}$ is tiny on the displayed scale.

```
fig, ax = plt.subplots()
ax.plot(horizons, Psi_uni[:, 0, 0], lw=2)
```

(continues on next page)

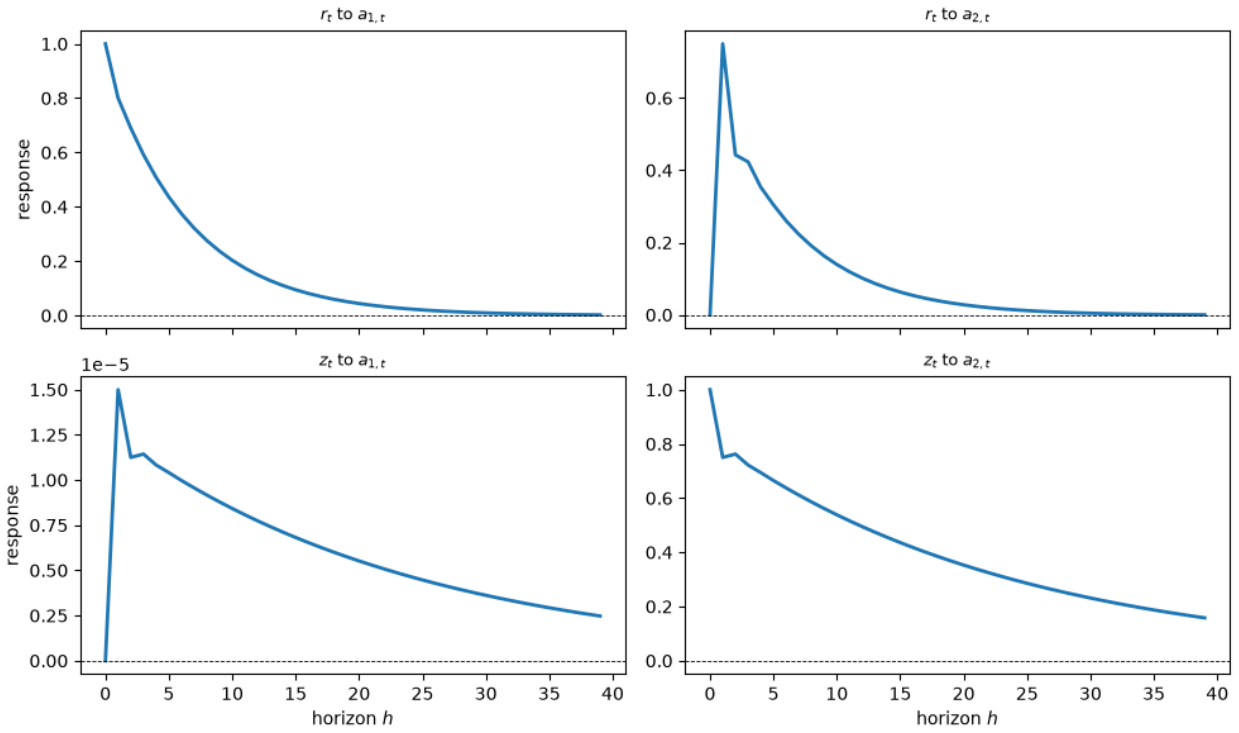


Fig. 47.3: System 1 responses to own innovations

(continued from previous page)

```
ax.axhline(0, color='k', lw=0.6, ls='--')
ax.set_xlabel('horizon $h$')
ax.set_ylabel('response')
fig.tight_layout()
plt.show()
```

With only r_t observed, the single innovation moves r_t one-for-one on impact and then decays monotonically toward zero.

For $h \geq 1$ the responses propagate through the state matrix A and decay geometrically, tracing out the Wold moving-average representation of the bivariate (System 1) and univariate (System 2) processes.

These are forecast-error responses from the Wold representation, not structural shock responses.

With the example in place, we can now pull together the main lessons of the lecture.

47.11 Summary

The Kalman filter recursively updates beliefs about a hidden state by combining a prior forecast with the new information contained in the current observation.

The Riccati equation tracks how the filter's conditional covariance evolves, and its steady state makes the filter time invariant.

At that steady state, the innovations representation writes the observable process in terms of one-step forecast errors that are white noise by construction.

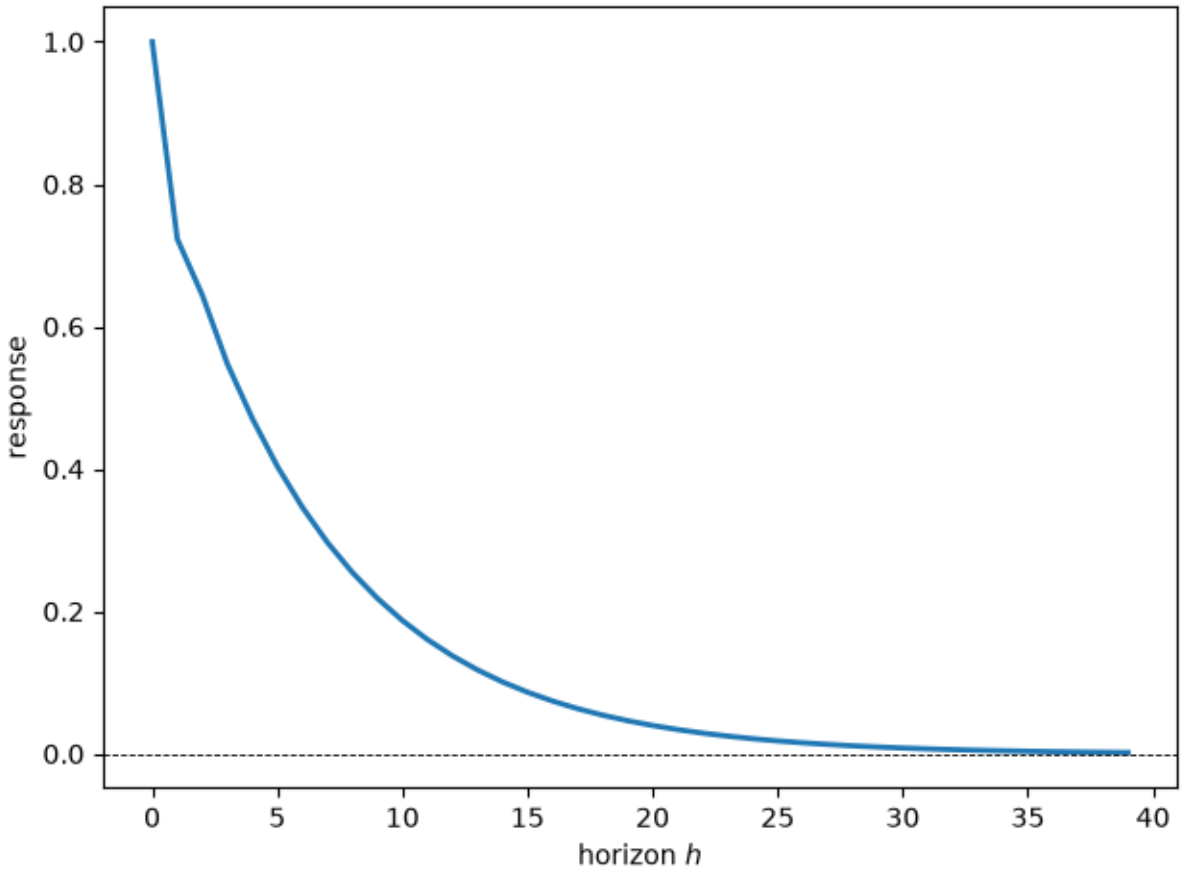


Fig. 47.4: System 2 response to own innovation

Solving the innovations representation backward gives an infinite-order VAR, while solving it forward gives the Wold moving-average representation.

The numerical examples show that changing the observed variables changes the Kalman gain and therefore changes the Wold responses, even when the underlying state dynamics are the same.

47.12 Exercises

Exercise 47.12.1

Consider the scalar AR(1) state space system used above with $\rho = 0.9$, $\sigma_w = 0.5$, $\sigma_v = 1.0$.

Derive an algebraic expression for the **steady-state** conditional variance Σ_∞ by solving the scalar Riccati equation (47.14) at its fixed point $\Sigma_{t+1} = \Sigma_t = \Sigma$.

Show that Σ satisfies a quadratic equation, find its positive root, and verify numerically that your formula matches `kf.Sigma_infinity`.

Solution

Here is one solution:

Setting $\Sigma_{t+1} = \Sigma_t = \Sigma$ in the scalar version of (47.14) with $A = \rho$, $CC^\top = \sigma_w^2$, $GG^\top = 1$, $R = \sigma_v^2$:

$$\Sigma = \rho^2 \Sigma + \sigma_w^2 - \frac{\rho^2 \Sigma^2}{\Sigma + \sigma_v^2}$$

Multiplying through by $\Sigma + \sigma_v^2$ and rearranging:

$$\Sigma^2 + \left[\sigma_v^2(1-\rho^2) - \sigma_w^2\right]\Sigma - \sigma_v^2 \sigma_w^2 = 0$$

Taking the positive root of this quadratic:

$$\Sigma_\infty = \frac{\sigma_w^2 - \sigma_v^2(1-\rho^2) + \sqrt{\left[\sigma_v^2(1-\rho^2) - \sigma_w^2\right]^2 + 4\sigma_v^2\sigma_w^2}}{2}$$

```
ρ_, σ_w_, σ_v_ = 0.9, 0.5, 1.0

b = σ_v_**2 * (1 - ρ_**2) - σ_w_**2
discriminant = b**2 + 4 * σ_v_**2 * σ_w_**2
Sigma_formula = (-b + np.sqrt(discriminant)) / 2

A_ = np.array([[ρ_]])
C_ = np.array([[σ_w_]])
G_ = np.array([[1.0]])
R_ = np.array([[σ_v_**2]])
H_ = np.array([[σ_v_]])   # R_ = H_ @ H_.T
lss_ = qe.LinearStateSpace(A_, C_, G_, H_, mu_0=np.zeros(1), Sigma_0=np.eye(1))
kf_ = qe.Kalman(lss_)

print(f"Analytical Σ_inf   = {Sigma_formula:.8f}")
print(f"Numerical  Σ_inf   = {kf_.Sigma_infinity[0, 0]:.8f}")
```

```
Analytical Σ_inf   = 0.53089919
Numerical  Σ_inf   = 0.53089919
```

Exercise 47.12.2

This exercise considers a two-dimensional state with a one-dimensional observation:

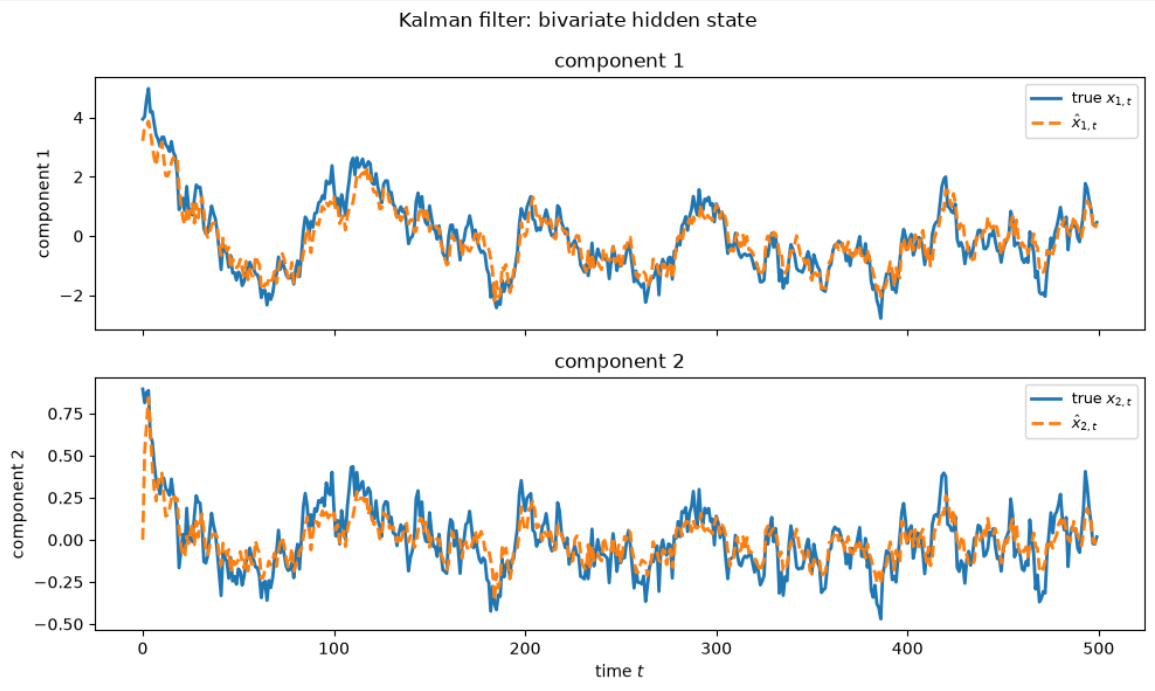
$$A = \begin{pmatrix} 0.9 & 0.1 \\ 0 & 0.8 \end{pmatrix}, \quad C = \begin{pmatrix} 0.4 \\ 0.1 \end{pmatrix}, \quad G = \begin{pmatrix} 1 & 0 \end{pmatrix}, \quad R = [0.5]$$

1. Simulate $T = 500$ observations from this system starting from a diffuse prior.
2. Run the Kalman filter and plot both components of $\hat{x}_t$ against the true hidden state path.
3. Compute and report the steady-state covariance Σ_∞ and Kalman gain K_∞.
4. Check that the eigenvalues of $A - K_\infty G$ lie strictly inside the unit circle, confirming that the VAR representation (47.20) is stable.

Solution

Here is one solution:

```
A2 = np.array([[0.9, 0.1],
               [0.0, 0.8]])
C2 = np.array([[0.4],
               [0.1]])
G2 = np.array([[1.0, 0.0]])
R2 = np.array([[0.5]])
H2 = np.array([[np.sqrt(0.5)]])   # R2 = H2 @ H2.T

lss2 = qe.LinearStateSpace(A2, C2, G2, H2,
                           mu_0=np.zeros(2),
                           Sigma_0=np.eye(2) * 5.0)
kf2 = qe.Kalman(lss2)
kf2.set_state(np.zeros(2), np.eye(2) * 5.0)

T2 = 500
x2_path, y2_path = lss2.simulate(ts_length=T2, random_state=0)

x_hats2 = np.zeros((T2, 2))
for t in range(T2):
    kf2.update(y2_path[:, t])
    x_hats2[t] = kf2.x_hat.ravel()

fig, axes = plt.subplots(2, 1, figsize=(10, 6), sharex=True)
for i, ax in enumerate(axes):
    ax.plot(x2_path[i, :T2], lw=2, label=f'true $x_{{{i+1},t}}$')
    ax.plot(x_hats2[:, i], lw=2, ls='--', label=rf'$\hat{{x}}_{{{i+1},t}}$')
    ax.set_title(f'component {i+1}')
    ax.legend(fontsize=9)
    ax.set_ylabel(f'component {i+1}')
axes[1].set_xlabel('time $t$')
fig.suptitle('Kalman filter: bivariate hidden state')
fig.tight_layout()
plt.show()

# Steady-state values
Sigma2_inf, K2_inf = kf2.stationary_values()
print("Steady-state covariance Σ_inf:")
print(np.round(Sigma2_inf, 5))
print("\nSteady-state Kalman gain K_inf:")
print(np.round(K2_inf, 5))

# Eigenvalues of A - K_inf G
AKG2 = A2 - K2_inf @ G2
eigvals2 = np.linalg.eigvals(AKG2)
print(f"\nEigenvalues of A - K_inf G: {np.round(eigvals2, 5)}")
print(f"Stable VAR: {np.all(np.abs(eigvals2) < 1)}")
```




```
Steady-state covariance Σ_inf:
[[0.32854 0.07219]
 [0.07219 0.0166 ]]

Steady-state Kalman gain K_inf:
[[0.36559]
 [0.06971]]

Eigenvalues of A - K_inf G: [0.56394 0.77047]
Stable VAR: True
```

After the diffuse-prior transient, both Kalman estimates move closely with their corresponding hidden state paths.

The second component is not observed directly, so its tracking comes from the state dynamics and its connection to the observed signal.

The printed eigenvalues then check the separate stability condition for the VAR representation.

Exercise 47.12.3

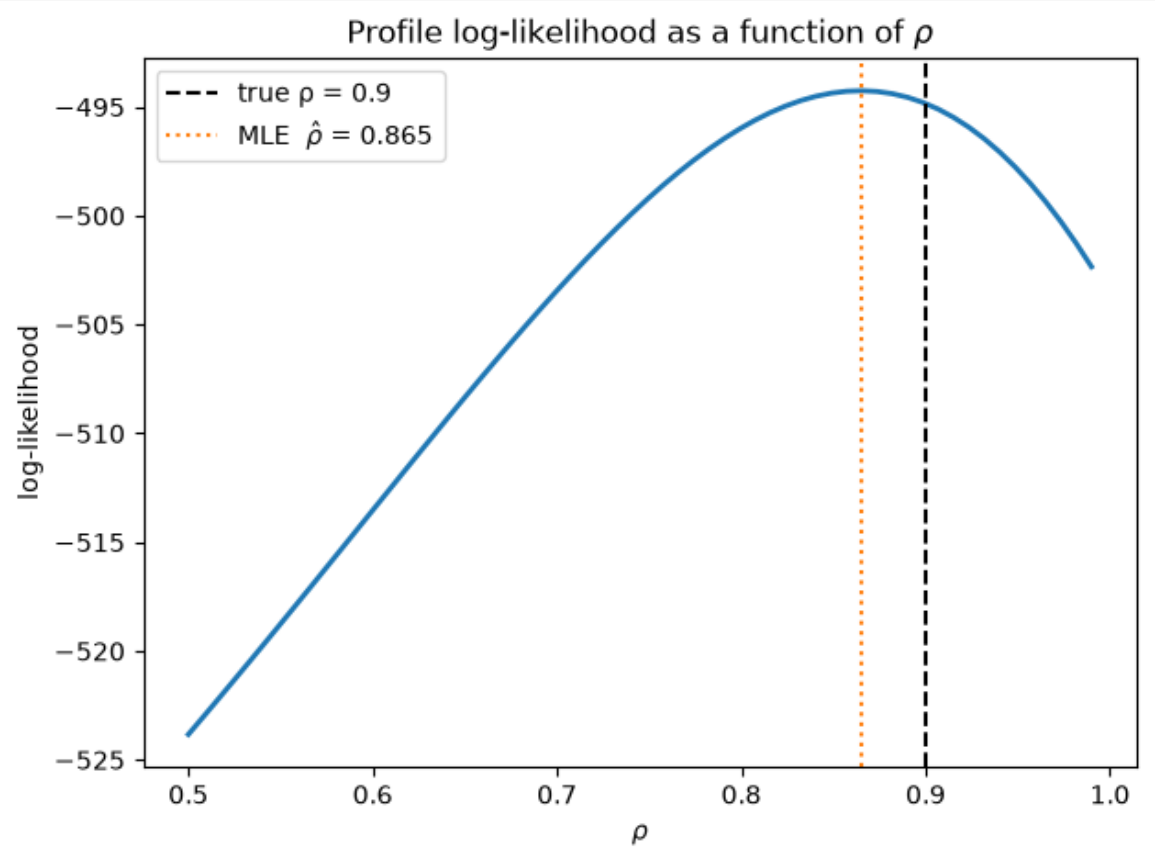
This exercise studies likelihood and parameter estimation using the scalar model from the main text with true parameters $(\rho, \sigma_w, \sigma_v) = (0.9, 0.5, 1.0)$:

1. Simulate $T = 300$ observations.
2. Write a function that evaluates the **log-likelihood** as a function of $\rho \in (0, 1)$, holding $\sigma_w = 0.5$ and $\sigma_v = 1.0$ fixed, and plot the log-likelihood against ρ for a grid of values.
3. Locate the maximum numerically and check that it is close to the true value $\rho = 0.9$.

Solution

Here is one solution:

```
# True parameters
ρ_true, sw_true, sv_true = 0.9, 0.5, 1.0

A_t = np.array([[ρ_true]])
C_t = np.array([[sw_true]])
G_t = np.array([[1.0]])
R_t = np.array([[sv_true**2]])
H_t = np.array([[sv_true]])   # R_t = H_t @ H_t.T

lss_t = qe.LinearStateSpace(A_t, C_t, G_t, H_t,
                            mu_0=np.zeros(1), Sigma_0=np.eye(1))
_, y_sim = lss_t.simulate(ts_length=300, random_state=7)
y_sim = y_sim.T          # shape (300, 1)

def ll_rho(ρ_val):
    A_ = np.array([[ρ_val]])
    C_ = np.array([[sw_true]])
    G_ = np.array([[1.0]])
    R_ = np.array([[sv_true**2]])
    return log_likelihood(A_, C_, G_, R_, y_sim,
                          np.zeros(1), np.eye(1) * 10.0)

ρ_grid = np.linspace(0.5, 0.99, 60)
ll_vals = np.array([ll_rho(r) for r in ρ_grid])

ρ_mle = ρ_grid[np.argmax(ll_vals)]

fig, ax = plt.subplots()
ax.plot(ρ_grid, ll_vals, lw=2)
ax.axvline(ρ_true, color='k',   ls='--', label=f'true ρ = {ρ_true}')
ax.axvline(ρ_mle, color='C1', ls=':', label=f'MLE  $\\hat{{\\rho}}$ = {ρ_mle:.3f}')
ax.set_xlabel(r'$\rho$')
ax.set_ylabel('log-likelihood')
ax.set_title('Profile log-likelihood as a function of $\\rho$')
ax.legend()
fig.tight_layout()
plt.show()

print(f"True ρ = {ρ_true},  MLE ρ_hat = {ρ_mle:.4f}")
```




```
True ρ = 0.9,  MLE ρ_hat = 0.8654
```

The likelihood curve is single-peaked and reaches its maximum close to the true value $\rho = 0.9$.

The small gap between the grid maximizer and the true value comes from finite-sample randomness and the use of a discrete grid.

CHAPTER

FORTYEIGHT

ORGANIZATION CAPITAL

48.1 Overview

This lecture describes a theory of **organization capital** proposed by Prescott and Visscher [1980].

Prescott and Visscher define organization capital as information that a firm accumulates about its employees, teams, and production processes.

This information is an *asset* to the firm because it affects the production possibility set and is produced jointly with output.

Costs of adjusting the stock of organization capital constrain the firm's growth rate, providing an explanation for

1. why firm growth rates are independent of firm size (Gibrat's Law)
2. why adjustment costs for rapid growth arise endogenously rather than being assumed

The paper offers three examples of organization capital:

- *Personnel information*: knowledge about the match between workers and tasks
- *Team information*: knowledge about how well groups of workers mesh
- *Firm-specific human capital*: skills of employees enhanced by on-the-job training

In each case, the investment possibilities lead firms to grow at a common rate, yielding constant returns to scale together with increasing costs of rapid size adjustment.

Note

The theory is related to ideas of Coase [1937] and Williamson [1975] about the nature of the firm.

Prescott and Visscher stress the firm's role as a storehouse of information and argue that incentives within the firm are created for efficient accumulation and use of that information.

Let's start with some imports:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm
from scipy.optimize import brentq
```

48.2 The basic idea

The firm is a storehouse of information.

Within the firm, incentives are created for the efficient accumulation and use of that information.

Prescott and Visscher exploit this concept to explain certain facts about firm growth and size distribution.

The key insight: the process by which information is accumulated naturally leads to

1. *constant returns to scale*, and
2. *increasing costs to rapid firm size adjustment*

Constant returns to scale explain the absence of an observed unique optimum firm size (see Stigler [1958]).

Without costs of adjustment, the pattern of investment by firms in the face of a change in market demand would exhibit discontinuities we do not observe.

Further, without a cost penalty to rapid growth, the first firm to discover a previously untapped market would preempt competition by usurping all profitable investments as they appear, thus implying monopoly more prevalent than it is.

48.3 Personnel information as organization capital

The first example of organization capital is information about the match between workers and tasks.

48.3.1 Setup

Workers have different sets of skills and talents.

A variable θ measures the aptitude of a worker for a particular kind of work.

- Workers with high θ have comparative advantage in tasks requiring repeated attention to detail
- Workers with low θ have comparative advantage in work requiring broadly defined duties

The population distribution of θ is normal with mean zero and precision (inverse of variance) π:

$$\theta \sim N(0, 1/\pi)$$

When a worker is hired from the labor pool, neither the worker nor the employer knows θ.

Both know only the population distribution.

48.3.2 Three tasks

If q units of output are produced, assume:

- $\varphi_1 q$ workers are assigned to *task 1* (screening)
- φq workers are assigned to *task 2*
- the remaining workers are assigned to *task 3*

where $\varphi_1 + 2\varphi = 1$.

> **Note**
>
> The fixed coefficients technology requires a constant ratio between the number of personnel in jobs 2 and 3 and the number assigned to job 1.

For task 1, the screening task, per unit cost of production is *invariant* to the θ-values of the individuals assigned.

However, the larger a worker's θ, the larger is his product in task 2 relative to his product in task 3.

Consequently:

- a worker with a highly positive θ is much better suited for task 2
- a worker with a highly negative θ is much better suited for task 3

48.3.3 Bayesian learning

Performance in tasks 2 or 3 cannot be observed at the individual level.

But information about a worker's θ-value can be obtained from observing performance in task 1, the screening task.

The expert supervising the apprentice determines a value of z each period:

$$z_{it} = \theta_i + \epsilon_{it} \tag{48.1}$$

where $\epsilon_{it} \sim N(0, 1)$ are independently distributed over both workers i and periods t.

After n observations on a worker in the screening job, the *posterior distribution* of θ is normal with

posterior mean:

$$m = \frac{1}{\pi + n} \sum_{k=1}^{n} z_k \tag{48.2}$$

posterior precision:

$$h = \pi + n \tag{48.3}$$

Knowledge of an individual is thus completely characterized by the pair (m, h).

```
def bayesian_update(z_observations, prior_precision):
    """
    Compute posterior mean and precision after observing signals.
    """
    n = len(z_observations)
    h = prior_precision + n
    m = np.sum(z_observations) / h
    return m, h
```

Let's visualize how the posterior evolves as we observe a worker whose true $\theta = 0.8$:

```
np.random.seed(0)

θ_true = 0.8
π = 1.0

T = 20
```

(continues on next page)

(continued from previous page)

```
ε = np.random.randn(T)
z_signals = θ_true + ε

posterior_means = []
posterior_stds = []

for n in range(1, T + 1):
    m, h = bayesian_update(z_signals[:n], π)
    posterior_means.append(m)
    posterior_stds.append(1 / np.sqrt(h))

fig, axes = plt.subplots(1, 2, figsize=(12, 5))

ax = axes[0]
ax.plot(range(1, T + 1), posterior_means, '-o', markersize=4, lw=2,
        label='Posterior mean $m$')
ax.axhline(θ_true, color='r', linestyle='--',
           label=fr'True $\theta = {θ_true}$')
ax.set_xlabel('number of observations $n$')
ax.set_ylabel('posterior mean $m$')
ax.legend()

ax = axes[1]
ax.plot(range(1, T + 1), posterior_stds, '-o', markersize=4, lw=2,
        label=r'Posterior std $1/\sqrt{h}$')
ax.set_xlabel('number of observations $n$')
ax.set_ylabel('posterior standard deviation')
ax.legend()

plt.tight_layout()
plt.show()
```

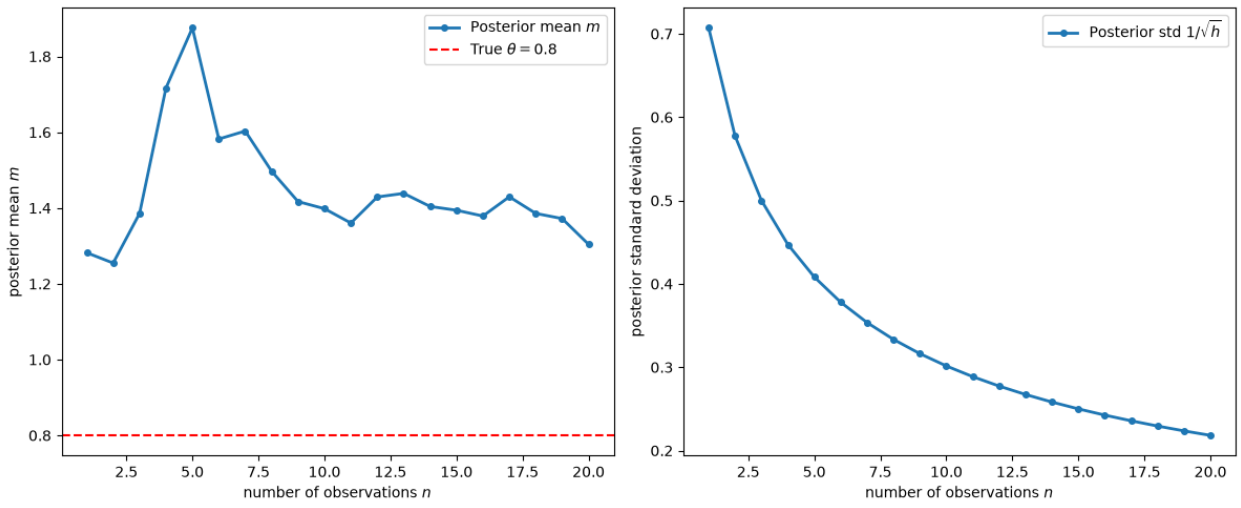


Fig. 48.1: Posterior mean convergence and uncertainty

As the number of screening observations n increases, the posterior mean converges to the true θ, and the posterior uncertainty shrinks at rate $1/\sqrt{n}$.

48.3.4 Per unit costs of production

Under the nonsequential assignment rule, employees with the greatest seniority are assigned to jobs 2 and 3, while newer employees remain in the screening task.

Workers with $m > 0$ are assigned to task 2, and those with $m \leq 0$ to task 3.

Per unit costs of production, assuming this assignment after n screening periods, are:

$$c(n) = c_1 + c_2 + c_3 - E\{\theta \mid m > 0\} + E\{\theta \mid m \leq 0\} \tag{48.4}$$

Because m is normally distributed, evaluation of the conditional expectation in (48.4) yields per unit costs as a function of n:

$$c(n) = c - 0.7978 \frac{n}{\pi(\pi + n)} \tag{48.5}$$

where $c = c_1 + c_2 + c_3$ and $0.7978 = 2 \int_0^\infty \frac{t}{\sqrt{2\pi}} e^{-t^2/2} dt$.

Note

The constant $0.7978 \approx \sqrt{2/\pi}$ is the mean of the standard half-normal distribution.

It arises from computing $E[\theta \mid m > 0] - E[\theta \mid m \leq 0]$ for a normal distribution.

The function $c(n)$ decreases at a *decreasing rate* in n.

More screening observations reduce costs but with diminishing returns.

```
def cost_per_unit(n_vals, π, c_bar=1.0):
    """
    Per unit cost as a function of screening periods n.
    """
    n_vals = np.asarray(n_vals, dtype=float)
    return c_bar - 0.7978 * n_vals / (π * (π + n_vals))


fig, ax = plt.subplots(figsize=(10, 6))

n_vals = np.linspace(0.1, 50, 200)

for π in [0.5, 1.0, 2.0, 5.0]:
    costs = cost_per_unit(n_vals, π)
    ax.plot(n_vals, costs, lw=2, label=fr'$\pi = {π}$')

ax.set_xlabel('screening periods $n$')
ax.set_ylabel('per unit cost $c(n)$')
ax.legend()
ax.set_xlim(0, 50)
plt.tight_layout()
plt.show()
```

The figure shows that:

- costs decrease with more screening time n
- the decrease is at a declining rate (diminishing returns to screening)

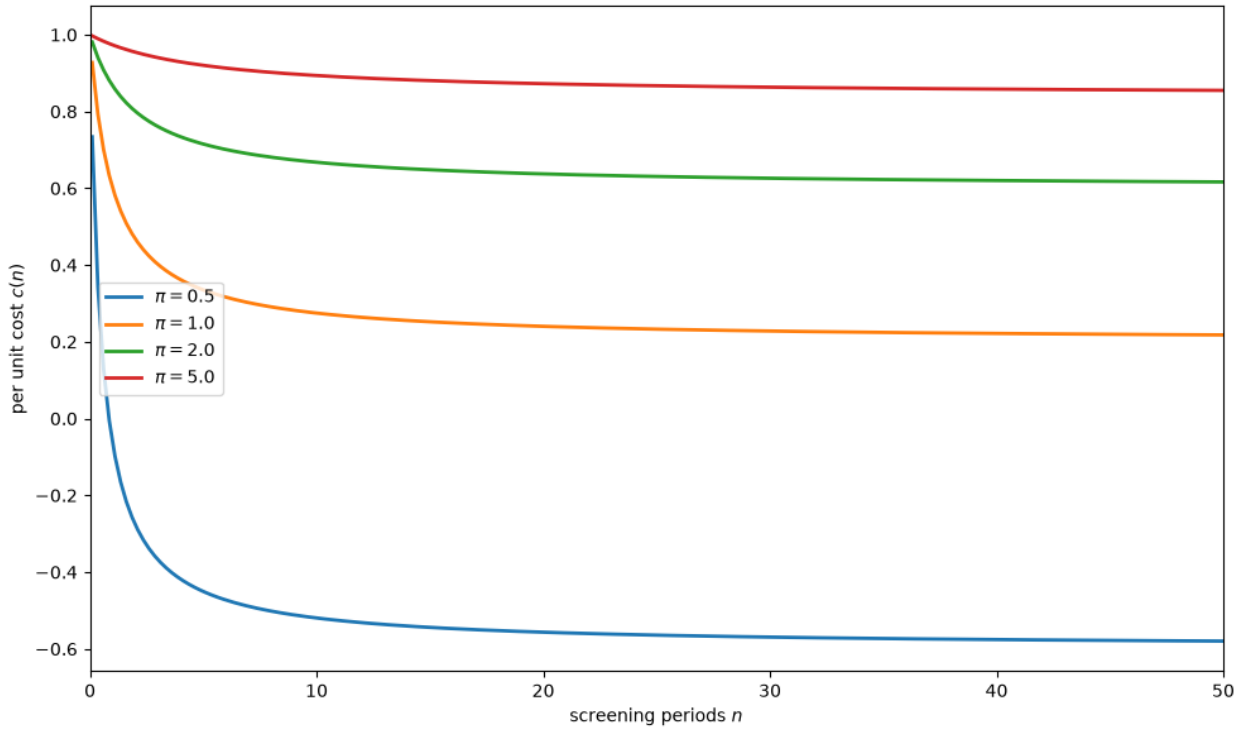


Fig. 48.2: Per unit costs by screening time

- for smaller prior precision π (more initial uncertainty about worker types), the gains from screening are larger

This diminishing-returns structure is the source of the *increasing costs of rapid adjustment*.

48.3.5 Growth rate and screening time

The greater the growth rate, the smaller must be n — the time spent in the screening task before assignment to job 2 or 3.

If γ is the growth rate of output and ρ is the quit rate, and y_i is the current number of vintage i employees, then

$$(1+\gamma)y_{i+1} = (1-\rho)y_i$$

Letting $\xi = (1-\rho)/(1+\gamma)$, from the above $y_i = \xi^i y_0$.

For the fixed coefficients technology, the fraction of present personnel with vintage greater than n must equal $2\varphi/(\varphi_1 + 2\varphi)$, which gives:

$$\xi^{n+1} = \frac{2\varphi}{\varphi_1 + 2\varphi} \tag{48.6}$$

Solving for n as a function of γ:

$$n(\gamma) = \frac{\log(2\varphi) - \log(\varphi_1 + 2\varphi)}{\log(1-\rho) - \log(1+\gamma)} - 1 \quad \text{for } \gamma > -\rho \tag{48.7}$$

```
def screening_time(γ, ρ, φ1, φ):
    """
```

(continues on next page)

(continued from previous page)

```
    Screening time n as a function of growth rate γ.
    """
    γ = np.asarray(γ, dtype=float)
    numerator = np.log(2 * ψ) - np.log(ψ1 + 2 * ψ)
    denominator = np.log(1 - ρ) - np.log(1 + γ)
    return numerator / denominator - 1


ρ = 0.1
ψ1 = 0.5
ψ = 0.25

γ_vals = np.linspace(-0.05, 0.30, 200)

valid = γ_vals > -ρ
γ_valid = γ_vals[valid]
n_vals = screening_time(γ_valid, ρ, ψ1, ψ)
mask = n_vals > 0
γ_plot = γ_valid[mask]
n_plot = n_vals[mask]

fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(γ_plot, n_plot, lw=2)
ax.set_xlabel(r'growth rate $\gamma$')
ax.set_ylabel(r'screening periods $n(\gamma)$')
ax.set_xlim(γ_plot[0], γ_plot[-1])
plt.tight_layout()
plt.show()
```

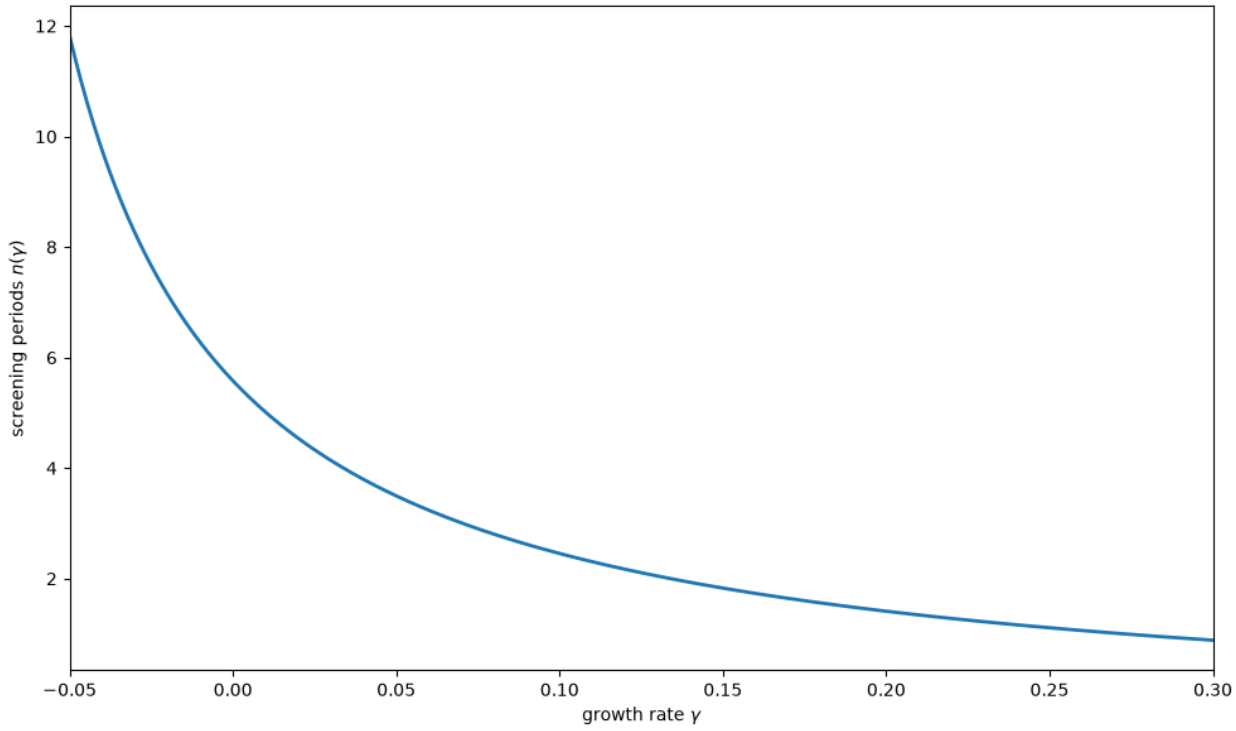


Fig. 48.3: Screening time vs. growth rate

The figure shows the key trade-off: *faster growth forces shorter screening periods*.

When growth is rapid, new workers must be promoted from the screening task to productive tasks more quickly, so less information is gathered about each worker before assignment.

48.3.6 Combined effect: growth rate and per unit costs

Composing the functions $c(n)$ and $n(\gamma)$ reveals how per unit costs depend on the growth rate:

```
fig, ax = plt.subplots(figsize=(10, 6))

π = 1.0
c_bar = 1.0

n_of_γ = screening_time(γ_plot, ρ, ψ1, ψ)
costs_of_γ = cost_per_unit(n_of_γ, π, c_bar)

ax.plot(γ_plot, costs_of_γ, lw=2)
ax.set_xlabel(r'growth rate $\gamma$')
ax.set_ylabel(r'per unit cost $c(n(\gamma))$')
ax.set_xlim(γ_plot[0], γ_plot[-1])
plt.tight_layout()
plt.show()
```

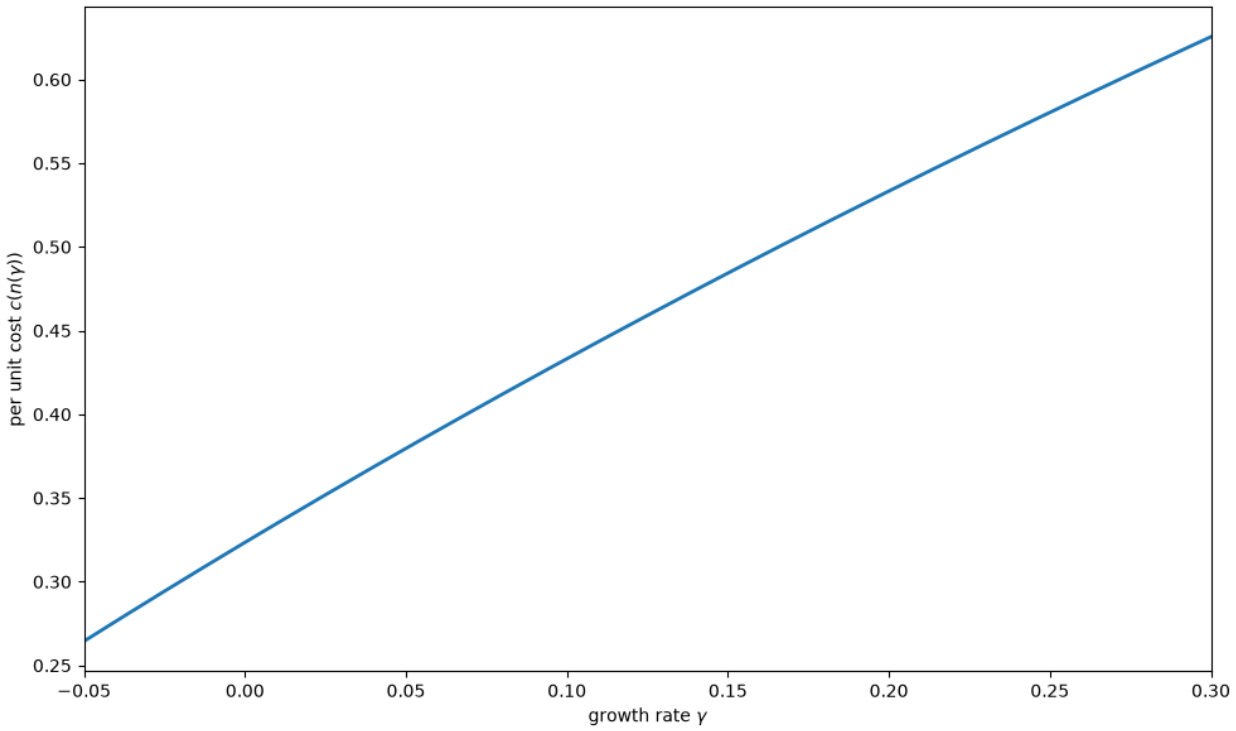


Fig. 48.4: Per unit costs vs. growth rate

This establishes the key result: *increasing costs of rapid adjustment arise endogenously* from the trade-off between screening and growth.

The faster the firm grows, the less time it has to screen workers, the poorer the match between workers and tasks, and the higher the per unit production costs.

48.4 Industry equilibrium

Firm growth rates are independent of firm size in this model because the mathematical structure of the technology constraint is the same as that considered in Lucas Jr [1967], except that the stock of organization capital is a vector rather than a scalar.

The technology set facing price-taking firms is a **convex cone**: there are constant returns to scale.

Constant returns and internal adjustment costs, along with some costs of transferring capital between firms, yield an optimum rate of firm growth *independent of the firm's size* — this is Gibrat's Law.

The bounded, downward-sloping, inverse industry demand function is

$$P_t = p(Q_t, u_t)$$

where Q_t is the sum of output over all firms and u_t is a demand shock subject to a stationary Markov process.

Prescott and Visscher show that a competitive equilibrium exists using the framework of Lucas and Prescott [1971].

The discounted consumer surplus to be maximized is

$$\sum_{t=0}^{\infty} \beta^t \left\{ \int_0^{Q_t} p(y, u_t) dy - Bw - Q_t \sum_i (A_{i2t} + A_{i3t}) c(i) / \sum_i (A_{i2t} + A_{i3t}) \right\} \tag{48.8}$$

where A_{i2t}, A_{i3t}, and B are obtained by summing a_{i2t}, a_{i3t}, and b, respectively, over all firms in the industry.

48.4.1 Key property: growth rates independent of size

If two firms have organization capital vectors $\underline{k}$ that are proportional at a point in time, they will be proportional in all future periods.

That is, *growth rates are independent of firm size*.

```
def simulate_firm_growth(T, γ, ρ, q0, seed=42):
    """
    Simulate firm output growth with stochastic shocks.
    """
    rng = np.random.default_rng(seed)
    output = np.zeros(T)
    output[0] = q0
    for t in range(1, T):
        shock = rng.normal(0, 0.02)
        output[t] = output[t-1] * (1 + γ + shock)
    return output


T = 50
γ_eq = 0.05
ρ = 0.1
```

(continues on next page)

(continued from previous page)

```
fig, axes = plt.subplots(1, 2, figsize=(14, 5))

ax = axes[0]
for q0, label in [(10, 'Small firm'), (50, 'Medium firm'),
                  (200, 'Large firm')]:
    output = simulate_firm_growth(T, y_eq, ρ, q0,
                                  seed=int(q0))
    ax.plot(range(T), output, lw=2, label=f'{label} ($q_0={q0}$)')
ax.set_xlabel('period')
ax.set_ylabel('output $q_t$')
ax.legend()

ax = axes[1]
for q0, label in [(10, 'Small firm'), (50, 'Medium firm'),
                  (200, 'Large firm')]:
    output = simulate_firm_growth(T, y_eq, ρ, q0,
                                  seed=int(q0))
    ax.plot(range(T), np.log(output), lw=2,
            label=f'{label} ($q_0={q0}$)')
ax.set_xlabel('period')
ax.set_ylabel(r'$\log(q_t)$')
ax.legend()

plt.tight_layout()
plt.show()
```

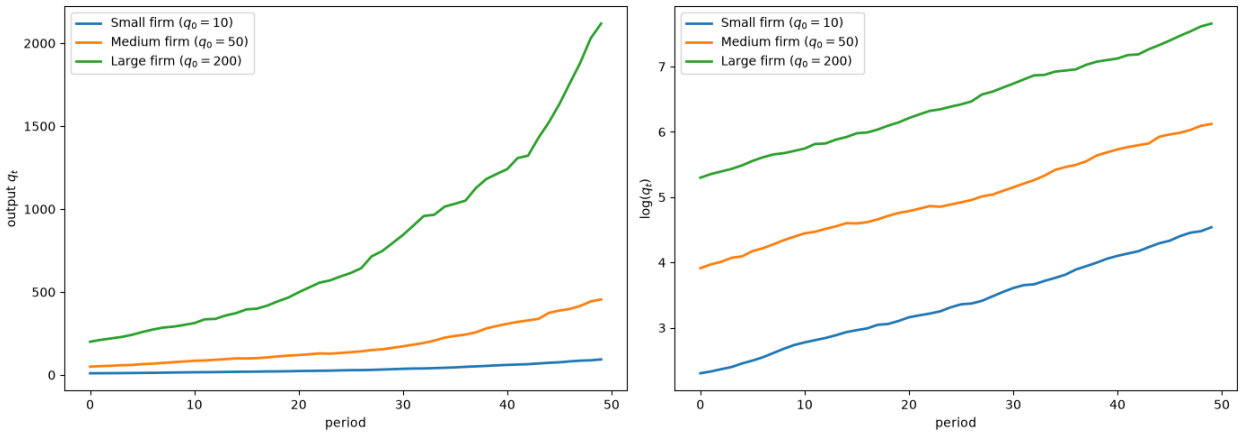


Fig. 48.5: Firm output levels and growth rates

The right panel shows that all firms grow at the same rate regardless of initial size — the log output paths are parallel.

This is **Gibrat's Law**: growth rates are independent of firm size.

48.5 Bayesian screening simulation

Let's simulate the full screening and assignment process for a single firm.

We draw workers from the population, observe their signals in the screening task, and then assign them to the appropriate productive task based on the posterior mean.

```
def simulate_screening(n_workers, n_screen, π, seed=123):
    """
    Simulate screening and assignment of workers.
    """
    rng = np.random.default_rng(seed)

    θ = rng.normal(0, 1/np.sqrt(π), n_workers)
    signals = (θ[:, None]
               + rng.normal(0, 1, (n_workers, n_screen)))
    posterior_means = signals.sum(axis=1) / (π + n_screen)

    assignment = np.where(posterior_means > 0, 2, 3)
    correct_assignment = np.where(θ > 0, 2, 3)
    misassignment_rate = np.mean(assignment != correct_assignment)

    return {
        'theta': θ,
        'posterior_means': posterior_means,
        'assignment': assignment,
        'correct_assignment': correct_assignment,
        'misassignment_rate': misassignment_rate
    }


π = 1.0
n_workers = 5000
screening_periods = [1, 3, 5, 10, 20, 50]

fig, axes = plt.subplots(2, 3, figsize=(15, 10))
axes = axes.flatten()

misassignment_rates = []

for idx, n_screen in enumerate(screening_periods):
    results = simulate_screening(n_workers, n_screen, π)
    misassignment_rates.append(results['misassignment_rate'])

    ax = axes[idx]
    θ = results['theta']
    m = results['posterior_means']

    correct = results['assignment'] == results['correct_assignment']
    ax.scatter(θ[correct], m[correct], alpha=0.1, s=5,
               color='blue', label='Correct')
    ax.scatter(θ[~correct], m[~correct], alpha=0.3, s=5,
               color='red', label='Misassigned')
    ax.axhline(0, color='k', linewidth=0.5)
    ax.axvline(0, color='k', linewidth=0.5)
    mis = results['misassignment_rate']
    ax.set_xlabel(r'true $\theta$')
```

(continues on next page)

(continued from previous page)

```
    ax.set_ylabel('posterior mean $m$')
    if idx == 0:
        ax.legend(markerscale=5, loc='upper left')

plt.tight_layout()
plt.show()
```

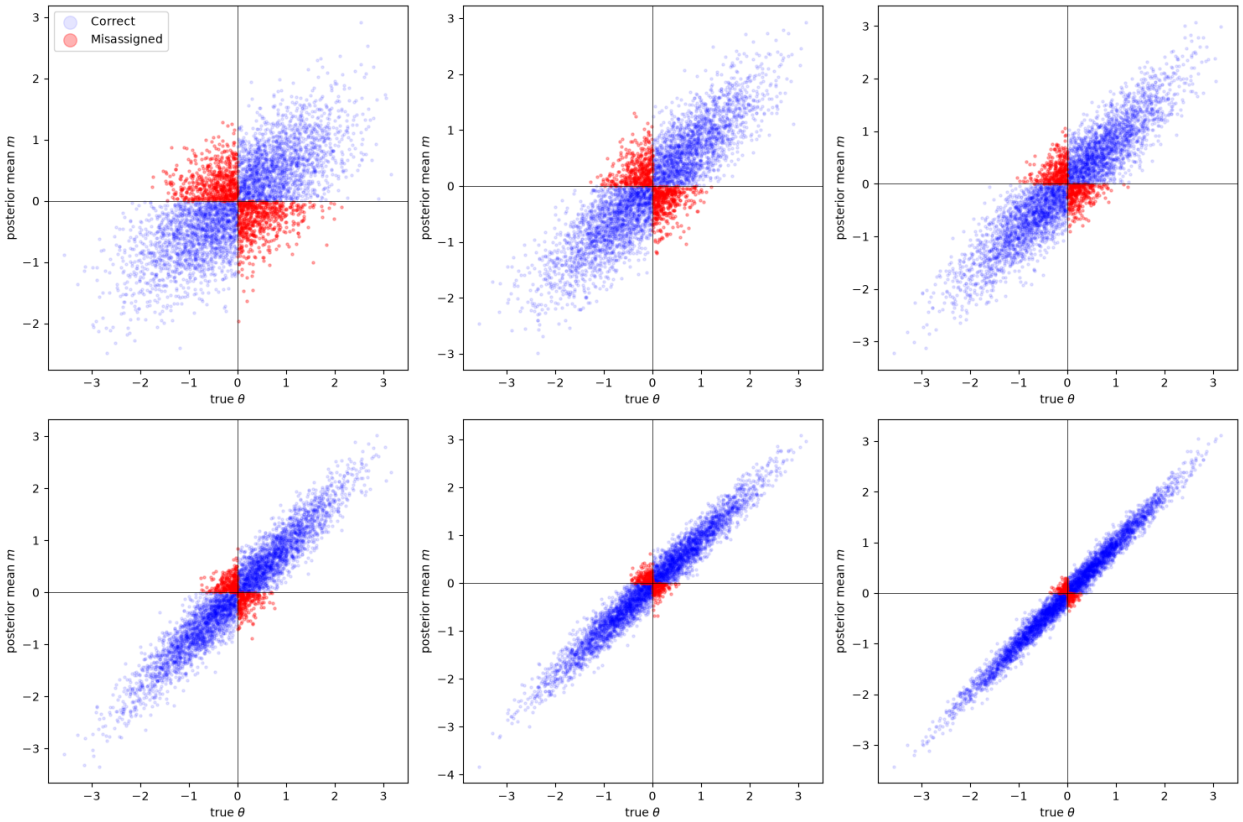


Fig. 48.6: Screening and worker assignment accuracy

Red dots are workers who are *misassigned* — placed in the wrong productive task because the posterior mean had the wrong sign relative to their true θ.

As n increases:

- The posterior mean m becomes more strongly correlated with θ
- Misassignment rates fall

```
fig, ax = plt.subplots(figsize=(10, 6))

n_range = np.arange(1, 51)
mis_rates = []
for n_screen in n_range:
    results = simulate_screening(n_workers, n_screen, π)
    mis_rates.append(results['misassignment_rate'])
```

(continues on next page)

(continued from previous page)

```
ax.plot(n_range, mis_rates, '-o', markersize=3, lw=2)
ax.set_xlabel('screening periods $n$')
ax.set_ylabel('misassignment rate')
plt.tight_layout()
plt.show()
```

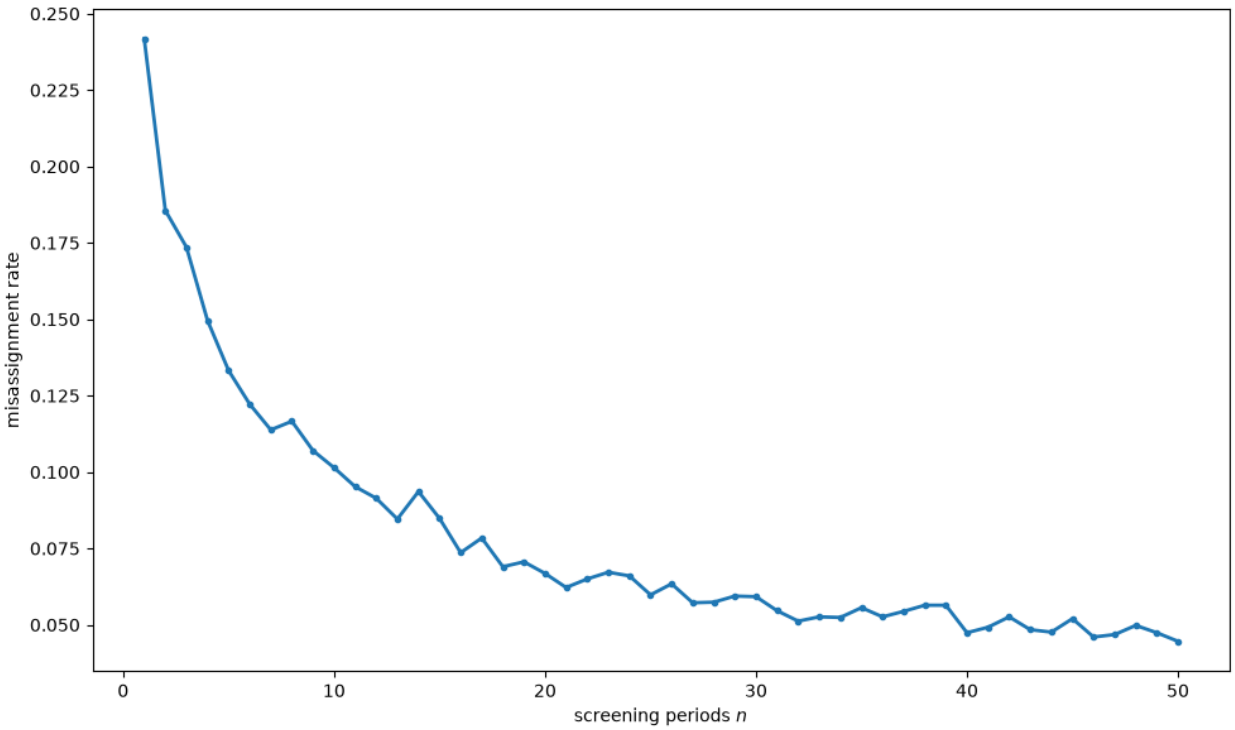


Fig. 48.7: Misassignment rate by screening time

This confirms the theoretical prediction: the cost savings from better assignment exhibit *diminishing returns* in the screening time n.

48.6 Team information

Personnel information need not be valuable only because it facilitates the matching of workers to tasks.

Another equally valuable use of personnel information is in the *matching of workers to workers*.

What is important to performance in many activities within the firm is not just the aptitude of an individual assigned to a task, but also how well the characteristics of the individual mesh with those of others performing related duties.

48.6.1 Structure

Suppose workers are grouped into teams, and team i assigned to a screening task has an observed productivity indicator

$$z_{it} = \theta_i + \epsilon_{it}$$

where:

- θ_i is a deterministic component directly related to how well team workers are paired
- $\epsilon_{it} \sim N(0, 1)$ are i.i.d. stochastic components

The θ from all possible teams are approximately independently and normally distributed $N(\mu, 1/\pi)$.

After n observations on team i, the posterior distribution on θ_i is normal with

$$m = \mu + \frac{1}{\pi + n} \sum_{k=1}^{n} (z_k - \mu)$$

and precision $h = \pi + n$.

If dissolution of a team also dissolves the accrued information, the team information model has the *same mathematical structure* as the personnel information model.

```
def simulate_team_screening(n_teams, n_screen, π, μ=0.5,
                            seed=456):
    """
    Simulate team screening with Bayesian updating.
    """
    rng = np.random.default_rng(seed)

    θ = rng.normal(μ, 1/np.sqrt(π), n_teams)
    signals = (θ[:, None]
               + rng.normal(0, 1, (n_teams, n_screen)))
    z_bar = signals.mean(axis=1)
    post_means = μ + n_screen * (z_bar - μ) / (π + n_screen)
    post_prec = π + n_screen

    return {
        'theta': θ,
        'posterior_means': post_means,
        'posterior_precision': post_prec
    }


fig, axes = plt.subplots(1, 3, figsize=(15, 5))

for idx, n_screen in enumerate([1, 5, 20]):
    results = simulate_team_screening(500, n_screen, π=1.0, μ=0.5)

    ax = axes[idx]
    ax.scatter(results['theta'], results['posterior_means'],
               alpha=0.4, s=10)
    lims = [-1.5, 2.5]
    ax.plot(lims, lims, 'r--', alpha=0.5, lw=2, label='45° line')
    ax.set_xlabel(r'true team quality $\theta$')
    ax.set_ylabel('posterior mean $m$')
    ax.set_xlim(lims)
    ax.set_ylim(lims)
```

(continues on next page)

(continued from previous page)

```
    ax.legend()
    ax.set_aspect('equal')

plt.tight_layout()
plt.show()
```

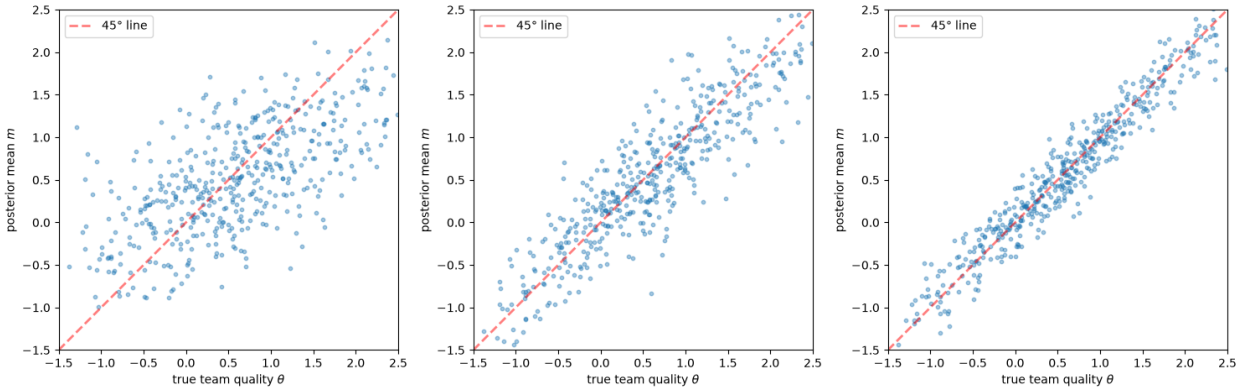


Fig. 48.8: Team quality estimates by screening periods

As with individual screening, more observations improve the precision of team quality estimates.

Rapid growth forces fewer observations before team assignments must be finalized, leading to higher costs.

48.7 Firm-specific human capital

The third example: organization capital consists of the **human capital** of the firm's employees.

The capacity of the organization to function effectively as a production unit is determined largely by the level and meshing of the skills of the employees.

Note

The case for the human capital of employees being part of the capital stock of the firm is well established (see Becker [1975]).

Productivity in the future depends on levels of human capital in the future, but to acquire human capital for the future, a sacrifice in real resources is required in the present.

The key features are:

- Output and skill enhancement are **joint products** resulting from the combination of labor inputs possessing different skill levels
- Experienced and inexperienced workers are combined in one of several available technical processes to generate the firm's product, and in the process, the overall competence of the work force is improved
- The transformation frontier between current output and future human capital is *concave* and linearly homogeneous

This gives the technology set the structure of a closed convex cone with a vertex at the origin — sufficient for optimal proportional growth by firms.

48.7.1 Concave transformation frontier

```
def transformation_frontier(q, α=0.7):
    """
    Concave transformation frontier between output and human capital.
    """
    q = np.asarray(q, dtype=float)
    return (1 - q**α)**(1/α)


fig, ax = plt.subplots(figsize=(8, 8))

q_vals = np.linspace(0, 1, 200)

for α in [0.5, 0.7, 1.0, 1.5]:
    hk = transformation_frontier(q_vals, α)
    ax.plot(q_vals, hk,
            label=fr'$\alpha = {α}$', lw=2)

ax.set_xlabel('current output $q$ (fraction of capacity)')
ax.set_ylabel('future human capital increment $\\Delta h$')
ax.legend()
ax.set_xlim(0, 1.05)
ax.set_ylim(0, 1.05)
ax.set_aspect('equal')
plt.tight_layout()
plt.show()
```

The concavity of the transformation frontier means that moving from an extremely unbalanced bundle of production and learning activity to a more balanced bundle entails little sacrifice.

But a workday consisting primarily of learning also has diminishing returns, creating the cost of rapid adjustment.

48.8 Costs of transferring organization capital

If there were no cost to transferring organization capital from one firm to another, the model would not place constraints on the firm's growth rate.

Firms could then merge, divest, or pirate each other's personnel without a cost penalty and thus produce a pattern of growth not restricted by the model.

Organization capital is *not* costlessly moved, however:

1. *Moving is disruptive*: relocating from one locale to another is disruptive to both employee and family
2. *Information is firm-specific*: the information set that makes a person productive in one organization may not make that person as productive in another, even if both firms produce identical output
    - Facility with a computer system at one firm
    - Knowing whom to ask when problems arise
    - Rapport with buyers or sellers

These are types of organization capital in one firm that *cannot be transferred costlessly* to another.

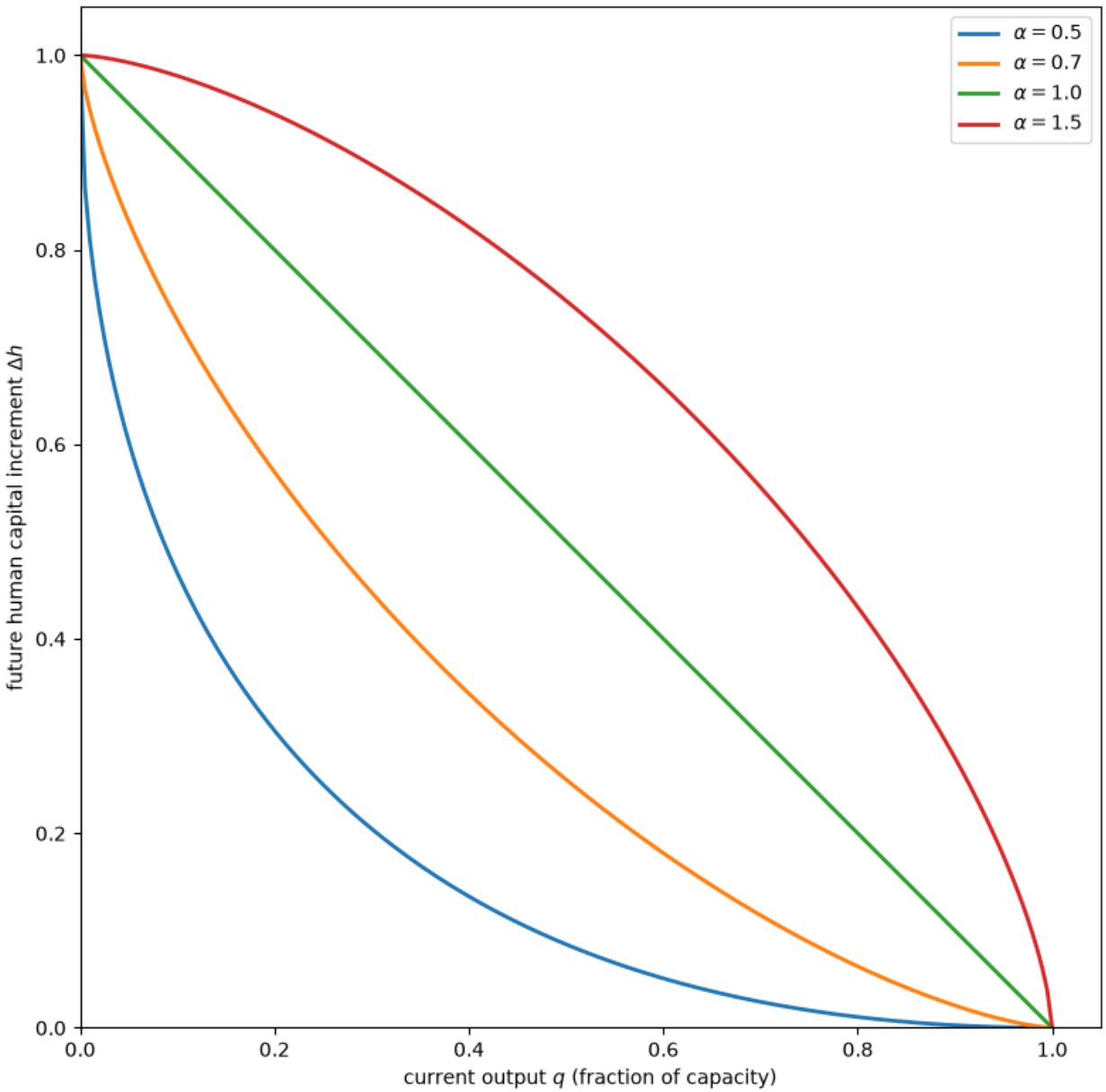


Fig. 48.9: Concave transformation frontier

48.9 Summary and implications

The Prescott-Visscher model provides a unified framework in which:

- The firm exists as an entity because it is an efficient structure for accumulating, storing, and using information
- *Constant returns to scale* arise because once the best combinations of worker types are discovered, nothing prevents the firm from replicating those combinations with proportional gains in product
- *Increasing adjustment costs* arise endogenously from the trade-off between current production and investment in organization capital
- *Gibrat's Law* — growth rates independent of firm size — is a natural implication
- Large firms should have growth rates that display *less variance* than small firms because large firms are essentially portfolios of smaller production units

```
def simulate_growth_rate_distribution(n_firms, n_subunits, γ,
                                      σ, T=100, seed=789):
    """
    Simulate growth rate distributions for firms of different sizes.
    """
    rng = np.random.default_rng(seed)
    subunit_growth = rng.normal(γ, σ,
                                (n_firms, n_subunits, T))
    firm_growth = subunit_growth.mean(axis=1)
    return firm_growth.mean(axis=1)


fig, ax = plt.subplots(figsize=(10, 6))

sizes = {'Small (1 unit)': 1,
         'Medium (5 units)': 5,
         'Large (20 units)': 20}

γ = 0.05
σ = 0.10

for label, n_sub in sizes.items():
    rates = simulate_growth_rate_distribution(
        2000, n_sub, γ, σ)
    ax.hist(rates, bins=50, alpha=0.5, density=True,
            label=f'{label}: std={rates.std():.4f}')

ax.set_xlabel('average growth rate')
ax.set_ylabel('density')
ax.legend()
ax.axvline(γ, color='k', linestyle='--',
           label=r'$\gamma$', alpha=0.5)
plt.tight_layout()
plt.show()
```

The figure shows that although all firms have the *same mean growth rate* (Gibrat's Law), large firms display *less variance* in realized growth rates because they are effectively portfolios of independent subunits.

This is consistent with the empirical findings of Mansfield [1962] and Hymer and Pashigian [1962].

The essence of the Prescott-Visscher theory is that the nature of the firm is tied to *organization capital*.

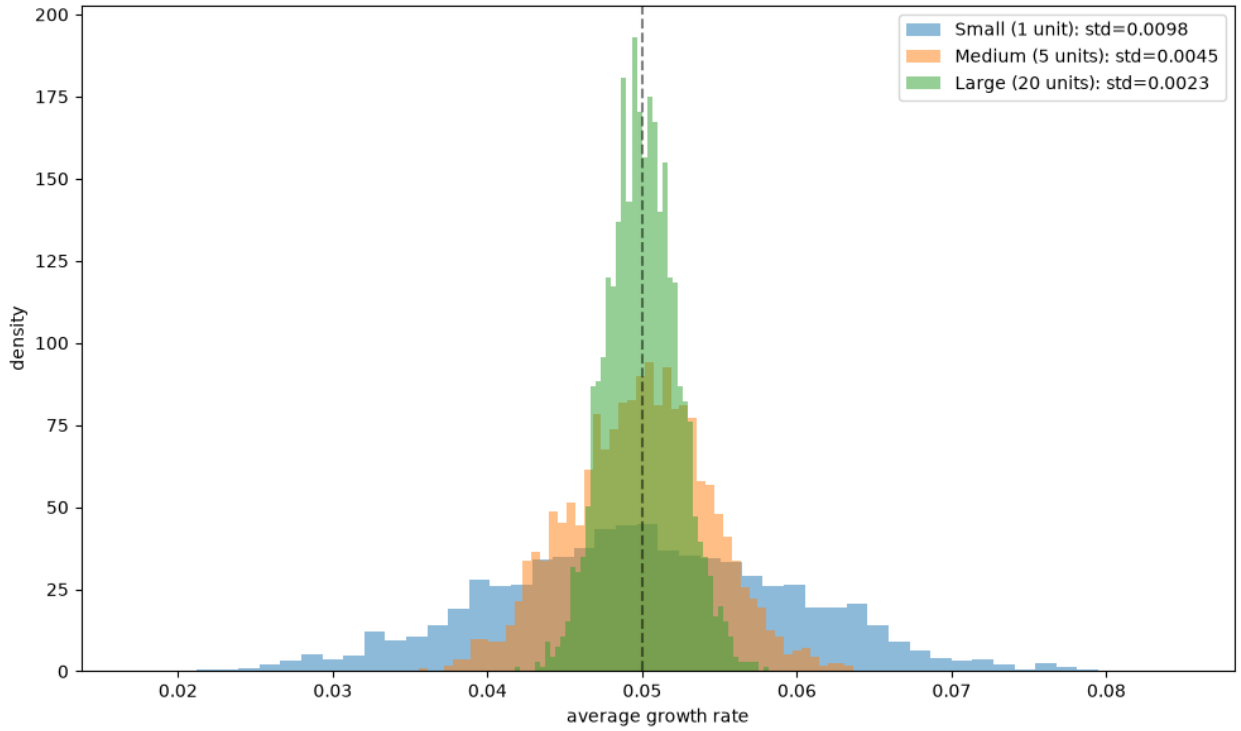


Fig. 48.10: Growth rate distributions by firm size

What distinguishes the firm from other relationships is that it is a structure within which agents have the incentive to acquire and reveal information in a manner that is less costly than in possible alternative institutions.

CHAPTER

FORTYNINE

TWO MODELS OF MEASUREMENTS AND THE INVESTMENT ACCELERATOR

Contents

- *Two Models of Measurements and the Investment Accelerator*
  - *Overview*
  - *The economic model*
  - *Measurement errors*
  - *A classical model of measurements initially collected by an agency*
  - *A model of optimal estimates reported by an agency*
  - *Simulation*
  - *Summary*

49.1 Overview

"Rational expectations econometrics" aims to interpret economic time series in terms of objects that are meaningful to economists, namely, parameters describing preferences, technologies, information sets, endowments, and equilibrium concepts.

When fully worked out, rational expectations models typically deliver a well-defined mapping from these economically interpretable parameters to the moments of the time series determined by the model.

If accurate observations on these time series are available, one can use that mapping to implement parameter estimation methods based either on the likelihood function or on the method of moments.

Note

This is why econometrics estimation is often called an "inverse" problem, while simulating a model for given parameter values is called a "direct problem". The direct problem refers to the mapping we have just described, while the inverse problem involves somehow applying an "inverse" of that mapping to a data set that is treated as if it were one draw from the joint probability distribution described by the mapping.

However, if only error-ridden data exist for the variables of interest, then more steps are needed to extract parameter estimates.

In effect, we require a model of the data reporting agency, one that is workable enough that we can determine the mapping induced jointly by the dynamic economic model and the measurement process to the probability law for the measured data.

The model chosen for the data collection agency is an aspect of an econometric specification that can make big differences in inferences about the economic structure.

Sargent [1989] describes two alternative models of data generation in a *permanent income* economy in which the investment accelerator, the mechanism studied in these two quantecon lectures – *Samuelson Multiplier-Accelerator* and *The Acceleration Principle and the Nature of Business Cycles* – shapes business cycle fluctuations.

- In Model 1, the data collecting agency simply reports the error-ridden data that it collects.
- In Model 2, the data collection agents first collects error-ridden data that satisfy a classical errors-in-variables model, then filters the data, and reports the filtered objects.

Although the two models have the same "deep parameters," they produce quite different sets of restrictions on the data.

In this lecture we follow Sargent [1989] and study how these alternative measurement schemes affect empirical implications.

We start with imports and helper functions to be used throughout this lecture to generate LaTeX output

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy import linalg
from IPython.display import Latex

np.set_printoptions(precision=3, suppress=True)

def df_to_latex_matrix(df, label=''):
    """Convert DataFrame to LaTeX matrix."""
    lines = [r'\begin{bmatrix}']

    for idx, row in df.iterrows():
        row_str = ' & '.join(
          [f'{v:.4f}' if isinstance(v, (int, float))
            else str(v) for v in row]) + r' \\'
        lines.append(row_str)

    lines.append(r'\end{bmatrix}')

    if label:
        return '$' + label + ' = ' + '\n'.join(lines) + '$'
    else:
        return '$' + '\n'.join(lines) + '$'

def df_to_latex_array(df):
    """Convert DataFrame to LaTeX array."""
    n_rows, n_cols = df.shape

    # Build column format (centered columns)
    col_format = 'c' * (n_cols + 1)  # +1 for index

    # Start array
    lines = [r'\begin{array}{' + col_format + '}']
```

(continues on next page)

(continued from previous page)

```
    # Header row
    header = ' & '.join([''] + [str(c) for c in df.columns]) + r' \\'
    lines.append(header)
    lines.append(r'\hline')

    # Data rows
    for idx, row in df.iterrows():
        row_str = str(idx) + ' & ' + ' & '.join(
          [f'{v:.3f}' if isinstance(v, (int, float)) else str(v)
          for v in row]) + r' \\'
        lines.append(row_str)

    lines.append(r'\end{array}')

    return '$' + '\n'.join(lines) + '$'
```

49.2 The economic model

The data are generated by a linear-quadratic version of a stochastic optimal growth model that is an instance of models described in this quantecon lecture: *The Permanent Income Model*.

A social planner chooses a stochastic process for $\{c_t, k_{t+1}\}_{t=0}^{\infty}$ that maximizes

$$E \sum_{t=0}^{\infty} \beta^t \left(u_0 + u_1 c_t - \frac{u_2}{2} c_t^2 \right) \tag{49.1}$$

subject to the restrictions imposed by the technology

$$c_t + k_{t+1} = f k_t + \theta_t, \qquad \beta f^2 > 1. \tag{49.2}$$

Here c_t is consumption, k_t is the capital stock, f is the gross rate of return on capital, and θ_t is an endowment or technology shock following

$$a(L)\, \theta_t = \varepsilon_t, \tag{49.3}$$

where L is the backward shift (or 'lag') operator and $a(z) = 1 - a_1 z - a_2 z^2 - \cdots - a_r z^r$ having all its zeroes outside the unit circle.

49.2.1 Optimal decision rule

The optimal decision rule for c_t is

$$c_t = \frac{-\alpha}{f-1} + \left(1 - \frac{1}{\beta f^2}\right) \frac{L - f^{-1} a(f^{-1})^{-1} a(L)}{L - f^{-1}}\, \theta_t + f k_t, \qquad k_{t+1} = f k_t + \theta_t - c_t, \tag{49.4}$$

where $\alpha = u_1 [1 - (\beta f)^{-1}] / u_2$.

Equations (49.3) and (49.4) exhibit the cross-equation restrictions characteristic of rational expectations models.

49.2.2 Net income and the accelerator

Define net output or national income as

$$y_{nt} = (f-1)k_t + \theta_t. \tag{49.5}$$

Note that (49.2) and (49.5) imply $(k_{t+1} - k_t) + c_t = y_{nt}$.

To obtain both a version of Friedman [1956]'s geometric distributed lag consumption function and a distributed lag accelerator, we impose two assumptions:

1. $a(L) = 1$, so that θ_t is white noise.
2. $\beta f = 1$, so the rate of return on capital equals the rate of time preference.

Assumption 1 is crucial for the strict form of the accelerator.

Relaxing it to allow serially correlated θ_t preserves an accelerator in a broad sense but loses the sharp geometric-lag form of (49.8).

Adding a second shock breaks the one-index structure entirely and can generate nontrivial Granger causality even without measurement error.

Assumption 2 is less important, affecting only various constants.

Under both assumptions, (49.4) simplifies to

$$c_t = (1 - f^{-1})\,\theta_t + (f-1)\,k_t. \tag{49.6}$$

When (49.6), (49.5), and (49.2) are combined, the optimal plan satisfies

$$c_t = \left(\frac{1-\beta}{1-\beta L}\right) y_{nt}, \tag{49.7}$$

$$k_{t+1} - k_t = f^{-1}\left(\frac{1-L}{1-\beta L}\right) y_{nt}, \tag{49.8}$$

$$y_{nt} = \theta_t + (1-\beta)(\theta_{t-1} + \theta_{t-2} + \cdots). \tag{49.9}$$

Equation (49.7) is Friedman's consumption model: consumption is a geometric distributed lag of income, with the decay coefficient β equal to the discount factor.

Equation (49.8) is the distributed lag accelerator: investment is a geometric distributed lag of the first difference of income.

This is the same mechanism that Chow [1968] documented empirically (see *The Acceleration Principle and the Nature of Business Cycles*).

Equation (49.9) states that the first difference of disposable income is a first-order moving average process with innovation equal to the innovation of the endowment shock θ_t.

As Muth [1960] showed, such a process is optimally forecast via a geometric distributed lag or "adaptive expectations" scheme.

49.2.3 The accelerator puzzle

When all variables are measured accurately and are driven by the single shock θ_t, the spectral density matrix of $(c_t,\ k_{t+1}-k_t,\ y_{nt})$ has rank one at all frequencies.

Each variable is an invertible one-sided distributed lag of the same white noise, so no variable Granger-causes any other.

Empirically, however, measures of output Granger-cause investment but not vice versa.

Sargent [1989] shows that measurement error can resolve this puzzle.

To illustrate, suppose first that output y_{nt} is measured perfectly while consumption and capital are each polluted by serially correlated measurement errors v_{ct} and v_{kt} orthogonal to θ_t.

Let $\bar{c}_t$ and $\bar{k}_{t+1} - \bar{k}_t$ denote the measured series. Then

$$\bar{c}_t = \left(\frac{1-\beta}{1-\beta L}\right) y_{nt} + v_{ct}, \tag{49.10}$$

$$\bar{k}_{t+1} - \bar{k}_t = \beta\left(\frac{1-L}{1-\beta L}\right) y_{nt} + (v_{k,t+1} - v_{kt}), \tag{49.11}$$

$$y_{nt} = \theta_t + (1-\beta)(\theta_{t-1} + \theta_{t-2} + \cdots). \tag{49.12}$$

In this case income Granger-causes consumption and investment but is not Granger-caused by them.

When each measured series is corrupted by measurement error, every measured variable will in general Granger-cause every other.

The strength of this Granger causality, as measured by decompositions of j-step-ahead prediction error variances, depends on the relative variances of the measurement errors.

In this case, each observed series mixes the common signal θ_t with idiosyncratic measurement noise.

A series with lower measurement error variance tracks θ_t more closely, so its innovations contain more information about future values of the other series.

Accordingly, in a forecast-error-variance decomposition, shocks to better-measured series account for a larger share of other variables' j-step-ahead prediction errors.

In a one-common-index model like this one (θ_t is the common index), better-measured variables extend more Granger causality to less well measured series than vice versa.

This asymmetry drives the numerical results we observe soon.

49.2.4 State-space formulation

Let's map the economic model and the measurement process into a linear state-space framework.

Set $f = 1.05$ and $\theta_t \sim \mathcal{N}(0,1)$.

Define the state and observation vectors

$$x_t = \begin{bmatrix} k_t \\ \theta_t \end{bmatrix}, \qquad z_t = \begin{bmatrix} y_{nt} \\ c_t \\ \Delta k_t \end{bmatrix},$$

so that the error-free data are described by the state-space system

$$\begin{aligned} x_{t+1} &= Ax_t + \varepsilon_t, \\ z_t &= Cx_t. \end{aligned} \tag{49.13}$$

where $\varepsilon_t = \begin{bmatrix} 0 \\ \theta_t \end{bmatrix}$ has covariance $E\varepsilon_t\varepsilon_t^\top = Q$ and the matrices are

$$A = \begin{bmatrix} 1 & f^{-1} \\ 0 & 0 \end{bmatrix}, \qquad C = \begin{bmatrix} f-1 & 1 \\ f-1 & 1-f^{-1} \\ 0 & f^{-1} \end{bmatrix}, \qquad Q = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}.$$

Q is singular because there is only one source of randomness θ_t; the capital stock k_t evolves deterministically given θ_t.

```
# Baseline structural matrices for the true economy
f = 1.05
β = 1 / f

A = np.array([
    [1.0, 1.0 / f],
    [0.0, 0.0]
])

C = np.array([
    [f - 1.0, 1.0],
    [f - 1.0, 1.0 - 1.0 / f],
    [0.0, 1.0 / f]
])

Q = np.array([
    [0.0, 0.0],
    [0.0, 1.0]
])
```

49.2.5 True impulse responses

Before introducing measurement error, we compute the impulse response of the error-free variables to a unit shock $\theta_0 = 1$.

This benchmark clarifies what changes when we later switch from error-free variables to variables reported by the statistical agency.

The response shows the investment accelerator clearly: the full impact on net income y_n occurs at lag 0, while consumption adjusts by only $1 - f^{-1} \approx 0.048$ and investment absorbs the remainder.

From lag 1 onward the economy is in its new steady state

```
def table2_irf(A, C, n_lags=6):
    x = np.array([0.0, 1.0])  # k_0 = 0, theta_0 = 1
    rows = []
    for j in range(n_lags):
        y_n, c, d_k = C @ x
        rows.append([y_n, c, d_k])
        x = A @ x
    return pd.DataFrame(rows, columns=[r'y_n', r'c', r'\Delta k'],
                        index=pd.Index(range(n_lags), name='lag'))

table2 = table2_irf(A, C, n_lags=6)
display(Latex(df_to_latex_array(table2)))
```

	y_n	c	Δk
0	1.000	0.048	0.952
1	0.048	0.048	0.000
2	0.048	0.048	0.000
3	0.048	0.048	0.000
4	0.048	0.048	0.000
5	0.048	0.048	0.000

49.3 Measurement errors

Let's add the measurement layer that generates reported data.

The econometrician does not observe z_t directly but instead sees $\bar{z}_t = z_t + v_t$, where v_t is a vector of measurement errors.

Measurement errors follow an AR(1) process

$$v_{t+1} = Dv_t + \eta_t, \tag{49.14}$$

where η_t is a vector white noise with $E\eta_t\eta_t^\top = \Sigma_\eta$ and $E\varepsilon_t v_s^\top = 0$ for all t, s.

The parameters are

$$D = \text{diag}(0.6, 0.7, 0.3), \qquad \sigma_\eta = (0.05, 0.035, 0.65),$$

so the unconditional covariance of v_t is

$$R = \text{diag}\left(\frac{\sigma_{\eta,i}^2}{1-\rho_i^2}\right).$$

The innovation variances are smallest for consumption ($\sigma_\eta = 0.035$), next for income ($\sigma_\eta = 0.05$), and largest for investment ($\sigma_\eta = 0.65$).

As in Sargent [1989] and our discussion above, what matters for Granger-causality asymmetries is the overall measurement quality in the full system: output is relatively well measured while investment is relatively poorly measured.

```
ρ = np.array([0.6, 0.7, 0.3])
D = np.diag(ρ)

# Innovation std. devs of η_t
σ_η = np.array([0.05, 0.035, 0.65])
Σ_η = np.diag(σ_η**2)

# Unconditional covariance of measurement errors v_t
R = np.diag((σ_η / np.sqrt(1.0 - ρ**2))**2)

print(f"f = {f},  β = 1/f = {β:.6f}")
print()
display(Latex(df_to_latex_matrix(pd.DataFrame(A), 'A')))
display(Latex(df_to_latex_matrix(pd.DataFrame(C), 'C')))
display(Latex(df_to_latex_matrix(pd.DataFrame(D), 'D')))
```

```
f = 1.05,  β = 1/f = 0.952381
```

$$A = \begin{bmatrix} 1.0000 & 0.9524 \\ 0.0000 & 0.0000 \end{bmatrix}$$

$$C = \begin{bmatrix} 0.0500 & 1.0000 \\ 0.0500 & 0.0476 \\ 0.0000 & 0.9524 \end{bmatrix}$$

$$D = \begin{bmatrix} 0.6000 & 0.0000 & 0.0000 \\ 0.0000 & 0.7000 & 0.0000 \\ 0.0000 & 0.0000 & 0.3000 \end{bmatrix}$$

We will analyze the two reporting schemes separately, but first we need a solver for the steady-state Kalman gain and error covariances.

The function below iterates on the Riccati equation until convergence, returning the Kalman gain K, the state covariance S, and the innovation covariance V

```
def steady_state_kalman(A, C_obs, Q, R, W=None, tol=1e-13, max_iter=200_000):
    """
    Solve steady-state Kalman equations for
        x_{t+1} = A x_t + w_{t+1}
        y_t     = C_obs x_t + v_t
    with cov(w)=Q, cov(v)=R, cov(w,v)=W.
    """
    n = A.shape[0]
    m = C_obs.shape[0]
    if W is None:
        W = np.zeros((n, m))

    S = Q.copy()
    for _ in range(max_iter):
        V = C_obs @ S @ C_obs.T + R
        K = (A @ S @ C_obs.T + W) @ np.linalg.inv(V)
        S_new = Q + A @ S @ A.T - K @ V @ K.T

        if np.max(np.abs(S_new - S)) < tol:
            S = S_new
            break
        S = S_new

    V = C_obs @ S @ C_obs.T + R
    K = (A @ S @ C_obs.T + W) @ np.linalg.inv(V)
    return K, S, V
```

With structural matrices and tools we need in place, we now follow Sargent [1989]'s two reporting schemes in sequence.

49.4 A classical model of measurements initially collected by an agency

A data collecting agency observes a noise-corrupted version of z_t, namely

$$\bar{z}_t = Cx_t + v_t. \tag{49.15}$$

We refer to this as *Model 1*: the agency collects noisy data and reports them without filtering.

To represent the second moments of the $\bar{z}_t$ process, it is convenient to obtain its population vector autoregression.

The error vector in the vector autoregression is the innovation to $\bar{z}_t$ and can be taken to be the white noise in a Wold moving average representation, which can be obtained by "inverting" the autoregressive representation.

The population vector autoregression, and how it depends on the parameters of the state-space system and the measurement error process, carries insights about how to interpret estimated vector autoregressions for $\bar{z}_t$.

Constructing the vector autoregression is also useful as an intermediate step in computing the likelihood of a sample of $\bar{z}_t$'s as a function of the free parameters $\{A, C, D, Q, R\}$.

The particular method that will be used to construct the vector autoregressive representation also proves useful as an intermediate step in constructing a model of an optimal reporting agency.

We use recursive (Kalman filtering) methods to obtain the vector autoregression for $\bar{z}_t$.

49.4.1 Quasi-differencing

Because the measurement errors v_t are serially correlated, the standard Kalman filter with white-noise measurement error cannot be applied directly to $\bar{z}_t = Cx_t + v_t$.

An alternative is to augment the state vector with the measurement-error AR components (see Appendix B of Sargent [1989]).

Here we take the quasi-differencing route described in Sargent [1989], which reduces the system to one with serially uncorrelated observation noise.

Define

$$\tilde{z}_t = \bar{z}_{t+1} - D\bar{z}_t, \qquad \bar{\nu}_t = C\varepsilon_t + \eta_t, \qquad \bar{C} = CA - DC. \tag{49.16}$$

Then the state-space system (49.13), the measurement error process (49.14), and the observation equation (49.15) imply the state-space system

$$\begin{aligned} x_{t+1} &= Ax_t + \varepsilon_t, \\ \tilde{z}_t &= \bar{C}\, x_t + \bar{\nu}_t, \end{aligned} \tag{49.17}$$

where $(\varepsilon_t, \bar{\nu}_t)$ is a white noise process with

$$E\begin{bmatrix}\varepsilon_t\end{bmatrix}\begin{bmatrix}\varepsilon_t^\top & \bar{\nu}_t^\top\end{bmatrix} = \begin{bmatrix} Q & W_1 \\ W_1^\top & R_1 \end{bmatrix}, \qquad R_1 = CQC^\top + R, \quad W_1 = QC^\top. \tag{49.18}$$

System (49.17) with covariances (49.18) is characterized by the five matrices $[A, \bar{C}, Q, R_1, W_1]$.

49.4.2 Innovations representation

Associated with (49.17) and (49.18) is the **innovations representation** for $\tilde{z}_t$,

$$\begin{aligned} \hat{x}_{t+1} &= A\hat{x}_t + K_1 u_t, \\ \tilde{z}_t &= \bar{C}\hat{x}_t + u_t, \end{aligned} \tag{49.19}$$

where

$$\begin{aligned} \hat{x}_t &= E[x_t \mid \tilde{z}_{t-1}, \tilde{z}_{t-2}, \ldots, \hat{x}_0] = E[x_t \mid \bar{z}_t, \bar{z}_{t-1}, \ldots], \\ u_t &= \tilde{z}_t - E[\tilde{z}_t \mid \tilde{z}_{t-1}, \tilde{z}_{t-2}, \ldots] = \bar{z}_{t+1} - E[\bar{z}_{t+1} \mid \bar{z}_t, \bar{z}_{t-1}, \ldots], \end{aligned} \tag{49.20}$$

$[K_1, S_1]$ are computed from the steady-state Kalman filter applied to $[A, \bar{C}, Q, R_1, W_1]$, and

$$S_1 = E[(x_t - \hat{x}_t)(x_t - \hat{x}_t)^\top]. \tag{49.21}$$

From (49.20), u_t is the innovation process for the $\bar{z}_t$ process.

49.4.3 Wold representation

System (49.19) and definition (49.16) can be used to obtain a Wold vector moving average representation for the $\bar{z}_t$ process:

$$\bar{z}_{t+1} = (I - DL)^{-1}\left[\bar{C}(I - AL)^{-1}K_1 L + I\right]u_t, \tag{49.22}$$

where L is the lag operator.

From (49.17) and (49.19) the innovation covariance is

$$V_1 = E\, u_t u_t^\top = \bar{C}\, S_1\, \bar{C}^\top + R_1. \tag{49.23}$$

Below we compute K_1, S_1, and V_1 numerically

```
C_bar = C @ A - D @ C
R1 = C @ Q @ C.T + R
W1 = Q @ C.T

K1, S1, V1 = steady_state_kalman(A, C_bar, Q, R1, W1)
```

49.4.4 Computing coefficients in a Wold moving average representation

To compute the moving average coefficients in (49.22) numerically, define the augmented state

$$r_t = \begin{bmatrix} \hat{x}_{t-1} \\ \bar{z}_{t-1} \end{bmatrix},$$

with dynamics

$$r_{t+1} = F_1 r_t + G_1 u_t, \qquad \bar{z}_t = H_1 r_t + u_t,$$

where

$$F_1 = \begin{bmatrix} A & 0 \\ \bar{C} & D \end{bmatrix}, \quad G_1 = \begin{bmatrix} K_1 \\ I \end{bmatrix}, \quad H_1 = [\bar{C} \;\; D].$$

The moving average coefficients are then $\psi_0 = I$ and $\psi_j = H_1 F_1^{j-1} G_1$ for $j \geq 1$.

```
F1 = np.block([
    [A, np.zeros((2, 3))],
    [C_bar, D]
])
G1 = np.vstack([K1, np.eye(3)])
H1 = np.hstack([C_bar, D])


def measured_wold_coeffs(F, G, H, n_terms=25):
    psi = [np.eye(3)]
    Fpow = np.eye(F.shape[0])
    for _ in range(1, n_terms):
        psi.append(H @ Fpow @ G)
        Fpow = Fpow @ F
    return psi
```

(continues on next page)

(continued from previous page)

```
def fev_contributions(psi, V, n_horizons=20):
    """
    Returns contrib[var, shock, h-1] = contribution at horizon h.
    """
    P = linalg.cholesky(V, lower=True)
    out = np.zeros((3, 3, n_horizons))
    for h in range(1, n_horizons + 1):
        acc = np.zeros((3, 3))
        for j in range(h):
            T = psi[j] @ P
            acc += T**2
        out[:, :, h - 1] = acc
    return out


psi1 = measured_wold_coeffs(F1, G1, H1, n_terms=40)
resp1 = np.array(
      [psi1[j] @ linalg.cholesky(V1, lower=True) for j in range(14)])
decomp1 = fev_contributions(psi1, V1, n_horizons=20)
```

49.4.5 Gaussian likelihood

The Gaussian log-likelihood function for a sample $\{\bar{z}_t,\ t = 0, \dots, T\}$, conditioned on an initial state estimate $\hat{x}_0$, can be represented as

$$\mathcal{L}^* = -T \ln 2\pi - \tfrac{1}{2} T \ln |V_1| - \tfrac{1}{2} \sum_{t=0}^{T-1} u_t^\top V_1^{-1} u_t, \tag{49.24}$$

where u_t is a function of $\{\bar{z}_t\}$ defined by (49.25) below.

To use (49.19) to compute $\{u_t\}$, it is useful to represent it as

$$\begin{aligned} \hat{x}_{t+1} &= (A - K_1 \bar{C})\, \hat{x}_t + K_1 \tilde{z}_t, \\ u_t &= -\bar{C}\, \hat{x}_t + \tilde{z}_t, \end{aligned} \tag{49.25}$$

where $\tilde{z}_t = \bar{z}_{t+1} - D\bar{z}_t$ is the quasi-differenced observation.

Given $\hat{x}_0$, equation (49.25) can be used recursively to compute a $\{u_t\}$ process.

Equations (49.24) and (49.25) give the likelihood function of a sample of error-corrupted data $\{\bar{z}_t\}$.

49.4.6 Forecast-error-variance decomposition

To measure the relative importance of each innovation, we decompose the j-step-ahead forecast-error variance of each measured variable.

Write $\bar{z}_{t+j} - E_t \bar{z}_{t+j} = \sum_{i=0}^{j-1} \psi_i u_{t+j-i}$.

Let P be the lower-triangular Cholesky factor of V_1 so that the orthogonalized innovations are $e_t = P^{-1} u_t$.

Then the contribution of orthogonalized innovation k to the j-step-ahead variance of variable m is $\sum_{i=0}^{j-1} (\psi_i P)_{mk}^2$.

The table below shows the cumulative contribution of each orthogonalized innovation to the forecast-error variance of y_n, c, and Δk at horizons 1 through 20.

Each panel fixes one orthogonalized innovation and reports its cumulative contribution to each variable's forecast-error variance.

Rows are forecast horizons and columns are forecasted variables.

```
horizons = np.arange(1, 21)
labels = [r'y_n', r'c', r'\Delta k']

def fev_table(decomp, shock_idx, horizons):
    return pd.DataFrame(
        np.round(decomp[:, shock_idx, :].T, 4),
        columns=labels,
        index=pd.Index(horizons, name='j')
    )
```

```
shock_titles = [r'\text{A. Innovation in } y_n',
                r'\text{B. Innovation in } c',
                r'\text{C. Innovation in } \Delta k']

parts = []
for i, title in enumerate(shock_titles):
    arr = df_to_latex_array(fev_table(decomp1, i, horizons)).strip('$')
    parts.append(
      r'\begin{array}{c} ' + title + r' \\ ' + arr + r' \end{array}')

display(Latex('$' + r' \quad '.join(parts) + '$'))
```

A. Innovation in y_n

j	y_n	c	Δk
1	1.006	0.002	0.904
2	1.008	0.004	0.904
3	1.011	0.007	0.904
4	1.013	0.009	0.904
5	1.016	0.011	0.904
6	1.018	0.013	0.904
7	1.020	0.016	0.904
8	1.022	0.018	0.904
9	1.025	0.020	0.904
10	1.027	0.023	0.904
11	1.029	0.025	0.904
12	1.031	0.027	0.904
13	1.034	0.029	0.904
14	1.036	0.032	0.904
15	1.038	0.034	0.904
16	1.040	0.036	0.904
17	1.043	0.038	0.904
18	1.045	0.041	0.904
19	1.047	0.043	0.904
20	1.050	0.045	0.904

B. Innovation in c

j	y_n	c	Δk
1	0.000	0.003	0.000
2	0.000	0.004	0.000
3	0.000	0.004	0.000
4	0.000	0.005	0.000
5	0.000	0.005	0.000
6	0.000	0.005	0.000
7	0.000	0.005	0.000
8	0.000	0.005	0.000
9	0.000	0.005	0.000
10	0.000	0.005	0.000
11	0.000	0.005	0.000
12	0.000	0.005	0.000
13	0.000	0.005	0.000
14	0.000	0.005	0.000
15	0.000	0.005	0.000
16	0.000	0.005	0.000
17	0.000	0.005	0.000
18	0.000	0.005	0.000
19	0.000	0.005	0.000
20	0.000	0.005	0.000

C. Innovation in Δk

j	y_n	c	Δk
1	0.000	0.000	0.468
2	0.000	0.000	0.510
3	0.000	0.000	0.513
4	0.000	0.000	0.514
5	0.000	0.000	0.514
6	0.000	0.000	0.514
7	0.000	0.000	0.514
8	0.000	0.000	0.514
9	0.000	0.000	0.514
10	0.000	0.000	0.514
11	0.000	0.000	0.514
12	0.000	0.000	0.514
13	0.000	0.000	0.514
14	0.000	0.000	0.514
15	0.000	0.000	0.514
16	0.000	0.000	0.514
17	0.000	0.000	0.514
18	0.000	0.000	0.514
19	0.000	0.000	0.514
20	0.000	0.000	0.514

The income innovation accounts for substantial proportions of forecast-error variance in all three variables, while the consumption and investment innovations contribute mainly to their own variances.

This is a **Granger causality** pattern: income appears to Granger-cause consumption and investment, but not vice versa.

This matches the paper's message that, in a one-common-index model, the relatively best measured series has the strongest predictive content.

Let's look at the covariance matrix of the innovations

```
print('Covariance matrix of innovations:')
df_v1 = pd.DataFrame(np.round(V1, 4), index=labels, columns=labels)
display(Latex(df_to_latex_matrix(df_v1)))
```

```
Covariance matrix of innovations:
```

$$\begin{bmatrix} 1.0057 & 0.0476 & 0.9533 \\ 0.0476 & 0.0047 & 0.0453 \\ 0.9533 & 0.0453 & 1.3718 \end{bmatrix}$$

The covariance matrix of the innovations is not diagonal, but the eigenvalues are well separated as shown below

```
print('Eigenvalues of covariance matrix:')
print(np.sort(np.linalg.eigvalsh(V1))[::-1].round(4))
```

```
Eigenvalues of covariance matrix:
[2.161 0.218 0.002]
```

The first eigenvalue is much larger than the others, consistent with the presence of a dominant common shock θ_t

49.4.7 Wold impulse responses

Impulse responses in the Wold representation are reported using orthogonalized innovations (Cholesky factorization of V_1 with ordering y_n, c, Δk).

Under this method, lag-0 responses reflect both contemporaneous covariance and the Cholesky ordering.

We first define a helper function to format the response coefficients as a LaTeX array

```
lags = np.arange(14)

def wold_response_table(resp, shock_idx, lags):
    return pd.DataFrame(
        np.round(resp[:, :, shock_idx], 4),
        columns=labels,
        index=pd.Index(lags, name='j')
    )
```

Now we report the impulse responses to each orthogonalized innovation in a single table with three panels

```
wold_titles = [r'\text{A. Response to } y_n \text{ innovation}',
               r'\text{B. Response to } c \text{ innovation}',
               r'\text{C. Response to } \Delta k \text{ innovation}']

parts = []
for i, title in enumerate(wold_titles):
    arr = df_to_latex_array(wold_response_table(resp1, i, lags)).strip('$')
    parts.append(
      r'\begin{array}{c} ' + title + r' \\ ' + arr + r' \end{array}')
```

(continues on next page)

(continued from previous page)

```
display(Latex('$' + r' \quad '.join(parts) + '$'))
```

A. Response to y_n innovation

	y_n	c	Δk
0	1.003	0.048	0.951
1	0.051	0.048	0.000
2	0.050	0.048	0.000
3	0.049	0.048	0.000
4	0.048	0.048	0.000
5	0.048	0.048	0.000
6	0.048	0.048	0.000
7	0.048	0.048	0.000
8	0.048	0.048	0.000
9	0.048	0.048	0.000
10	0.048	0.048	0.000
11	0.048	0.048	0.000
12	0.048	0.048	0.000
13	0.048	0.048	0.000

B. Response to c innovation

	y_n	c	Δk
0	0.000	0.050	0.004
1	−0.002	0.035	−0.000
2	0.000	0.026	−0.000
3	0.001	0.019	−0.000
4	0.002	0.014	−0.000
5	0.002	0.011	−0.000
6	0.002	0.008	−0.000
7	0.003	0.007	−0.000
8	0.003	0.005	−0.000
9	0.003	0.005	−0.000
10	0.003	0.004	−0.000
11	0.003	0.004	−0.000
12	0.003	0.003	−0.000
13	0.003	0.003	−0.000

C. Response to Δk innovation

	y_n	c	Δk
0	0.000	0.000	0.684
1	−0.004	0.000	0.203
2	−0.002	0.000	0.061
3	−0.001	0.000	0.018
4	−0.001	0.000	0.005
5	−0.000	0.000	0.002
6	−0.000	0.000	0.001
7	−0.000	0.000	0.000
8	0.000	0.000	0.000
9	0.000	0.000	0.000
10	0.000	0.000	0.000
11	0.000	0.000	0.000
12	0.000	0.000	0.000
13	0.000	0.000	0.000

At impact, the first orthogonalized innovation loads on all three measured variables.

At subsequent lags the income innovation generates persistent responses in all three variables because, being the best-measured series, its innovation is dominated by the true permanent shock θ_t.

The consumption and investment innovations produce responses that decay according to the AR(1) structure of their respective measurement errors ($\rho_c = 0.7$, $\rho_{\Delta k} = 0.3$), with little spillover to other variables.

49.5 A model of optimal estimates reported by an agency

Suppose that instead of reporting the error-corrupted data $\bar{z}_t$, the data collecting agency reports linear least-squares projections of the true data on a history of the error-corrupted data.

This model provides a possible way of interpreting two features of the data-reporting process.

- *seasonal adjustment*: if the components of v_t have strong seasonals, the optimal filter will assume a shape that can be interpreted partly in terms of a seasonal adjustment filter, one that is one-sided in current and past $\bar{z}_t$'s.
- *data revisions*: if z_t contains current and lagged values of some variable of interest, then the model simultaneously determines "preliminary," "revised," and "final" estimates as successive conditional expectations based on progressively longer histories of error-ridden observations.

To make this operational, we impute to the reporting agency a model of the joint process generating the true data and the measurement errors.

We assume that the reporting agency has "rational expectations": it knows the economic and measurement structure leading to (49.17)–(49.18).

To prepare its estimates, the reporting agency itself computes the Kalman filter to obtain the innovations representation (49.19).

Rather than reporting the error-corrupted data $\bar{z}_t$, the agency reports $\tilde{z}_t = G\hat{x}_t$, where G is a "selection matrix," possibly equal to C, for the data reported by the agency.

The data $G\hat{x}_t = E[Gx_t \mid \bar{z}_t, \bar{z}_{t-1}, \ldots, \hat{x}_0]$.

The state-space representation for the reported data is then

$$\begin{aligned} \hat{x}_{t+1} &= A\hat{x}_t + K_1 u_t, \\ \tilde{z}_t &= G\hat{x}_t, \end{aligned} \tag{49.26}$$

where the first line of (49.26) is from the innovations representation (49.19).

Note that u_t is the innovation to $\bar{z}_{t+1}$ and is *not* the innovation to $\tilde{z}_t$.

To obtain a Wold representation for $\tilde{z}_t$ and the likelihood function for a sample of $\tilde{z}_t$ requires that we obtain an innovations representation for (49.26).

49.5.1 Innovations representation for filtered data

To add a little generality to (49.26) we amend it to the system

$$\begin{aligned} \hat{x}_{t+1} &= A\hat{x}_t + K_1 u_t, \\ \tilde{z}_t &= G\hat{x}_t + \eta_t, \end{aligned} \tag{49.27}$$

where η_t is a type 2 white-noise measurement error process ("typos") with presumably very small covariance matrix R_2.

The covariance matrix of the joint noise is

$$E \begin{bmatrix} K_1 u_t \\ \eta_t \end{bmatrix} \begin{bmatrix} K_1 u_t \\ \eta_t \end{bmatrix}^\top = \begin{bmatrix} Q_2 & 0 \\ 0 & R_2 \end{bmatrix}, \tag{49.28}$$

where $Q_2 = K_1 V_1 K_1^\top$.

If R_2 is singular, it is necessary to adjust the Kalman filtering formulas by using transformations that induce a "reduced order observer."

In practice, we approximate a zero R_2 matrix with the matrix ϵI for a small $\epsilon > 0$ to keep the Kalman filter numerically well-conditioned.

For system (49.27) and (49.28), an innovations representation is

$$\begin{aligned} \check{x}_{t+1} &= A\check{x}_t + K_2 a_t, \\ \tilde{z}_t &= G\check{x}_t + a_t, \end{aligned} \tag{49.29}$$

where

$$\begin{aligned} a_t &= \tilde{z}_t - E[\tilde{z}_t \mid \tilde{z}_{t-1}, \tilde{z}_{t-2}, \ldots], \\ \check{x}_t &= E[\hat{x}_t \mid \tilde{z}_{t-1}, \tilde{z}_{t-2}, \ldots, \check{x}_0], \\ S_2 &= E[(\hat{x}_t - \check{x}_t)(\hat{x}_t - \check{x}_t)^\top], \\ [K_2, S_2] &= \text{kalmanfilter}(A, G, Q_2, R_2, 0). \end{aligned} \tag{49.30}$$

Thus $\{a_t\}$ is the innovation process for the reported data $\tilde{z}_t$, with innovation covariance

$$V_2 = E\, a_t a_t^\top = G\, S_2\, G^\top + R_2. \tag{49.31}$$

49.5.2 Wold representation

A Wold moving average representation for $\tilde{z}_t$ is found from (49.29) to be

$$\tilde{z}_t = [G(I - AL)^{-1} K_2 L + I] a_t, \tag{49.32}$$

with coefficients $\psi_0 = I$ and $\psi_j = GA^{j-1}K_2$ for $j \geq 1$.

Note that this is simpler than the Model 1 Wold representation (49.22) because there is no quasi-differencing to undo.

49.5.3 Gaussian likelihood

When a method analogous to Model 1 is used, a Gaussian log-likelihood for $\tilde{z}_t$ can be computed by first computing an $\{a_t\}$ sequence from observations on $\tilde{z}_t$ by using

$$\begin{aligned} \check{x}_{t+1} &= (A - K_2 G)\,\check{x}_t + K_2 \tilde{z}_t, \\ a_t &= -G\,\check{x}_t + \tilde{z}_t. \end{aligned} \tag{49.33}$$

The likelihood function for a sample of T observations $\{\tilde{z}_t\}$ is then

$$\mathcal{L}^{**} = -T \ln 2\pi - \tfrac{1}{2} T \ln |V_2| - \tfrac{1}{2} \sum_{t=0}^{T-1} a_t^\top V_2^{-1} a_t. \tag{49.34}$$

Note that relative to computing the likelihood function (49.24) for the error-corrupted data, computing the likelihood function for the optimally filtered data requires more calculations.

Both likelihood functions require that the Kalman filter (49.20) be computed, while the likelihood function for the filtered data requires that the Kalman filter (49.30) also be computed.

In effect, in order to interpret and use the filtered data reported by the agency, it is necessary to retrace the steps that the agency used to synthesize those data.

The Kalman filter (49.20) is supposed to be formed by the agency.

The agency need not use Kalman filter (49.30) because it does not need the Wold representation for the filtered data.

In our parameterization $G = C$.

```
Q2 = K1 @ V1 @ K1.T
ε = 1e-6

K2, S2, V2 = steady_state_kalman(A, C, Q2, ε * np.eye(3))


def filtered_wold_coeffs(A, C, K, n_terms=25):
    psi = [np.eye(3)]
    Apow = np.eye(2)
    for _ in range(1, n_terms):
        psi.append(C @ Apow @ K)
        Apow = Apow @ A
    return psi


psi2 = filtered_wold_coeffs(A, C, K2, n_terms=40)
resp2 = np.array(
  [psi2[j] @ linalg.cholesky(V2, lower=True) for j in range(14)])
decomp2 = fev_contributions(psi2, V2, n_horizons=20)
```

49.5.4 Forecast-error-variance decomposition

Because the filtered data are nearly noiseless, the innovation covariance V_2 is close to singular with one dominant eigenvalue.

This means the filtered economy is driven by essentially one shock, just like the true economy

```
parts = []
for i, title in enumerate(shock_titles):
    arr = df_to_latex_array(fev_table(decomp2, i, horizons)).strip('$')
    parts.append(
      r'\begin{array}{c} ' + title + r' \\ ' + arr + r' \end{array}')

display(Latex('$' + r' \quad '.join(parts) + '$'))
```

A. Innovation in y_n

	y_n	c	Δk
1	0.995	0.002	0.902
2	0.997	0.004	0.902
3	0.999	0.007	0.902
4	1.001	0.009	0.902
5	1.004	0.011	0.902
6	1.006	0.014	0.902
7	1.008	0.016	0.902
8	1.010	0.018	0.902
9	1.013	0.020	0.902
10	1.015	0.023	0.902
11	1.017	0.025	0.902
12	1.019	0.027	0.902
13	1.022	0.029	0.902
14	1.024	0.032	0.902
15	1.026	0.034	0.902
16	1.028	0.036	0.902
17	1.031	0.038	0.902
18	1.033	0.041	0.902
19	1.035	0.043	0.902
20	1.038	0.045	0.902

B. Innovation in c

	y_n	c	Δk
1	0.000	0.000	0.000
2	0.000	0.000	0.000
3	0.000	0.000	0.000
4	0.000	0.000	0.000
5	0.000	0.000	0.000
6	0.000	0.000	0.000
7	0.000	0.000	0.000
8	0.000	0.000	0.000
9	0.000	0.000	0.000
10	0.000	0.000	0.000
11	0.000	0.000	0.000
12	0.000	0.000	0.000
13	0.000	0.000	0.000
14	0.000	0.000	0.000
15	0.000	0.000	0.000
16	0.000	0.000	0.000
17	0.000	0.000	0.000
18	0.000	0.000	0.000
19	0.000	0.000	0.000
20	0.000	0.000	0.000

C. Innovation in Δk

	y_n	c	Δk
1	0.000	0.000	0.000
2	0.000	0.000	0.000
3	0.000	0.000	0.000
4	0.000	0.000	0.000
5	0.000	0.000	0.000
6	0.000	0.000	0.000
7	0.000	0.000	0.000
8	0.000	0.000	0.000
9	0.000	0.000	0.000
10	0.000	0.000	0.000
11	0.000	0.000	0.000
12	0.000	0.000	0.000
13	0.000	0.000	0.000
14	0.000	0.000	0.000
15	0.000	0.000	0.000
16	0.000	0.000	0.000
17	0.000	0.000	0.000
18	0.000	0.000	0.000
19	0.000	0.000	0.000
20	0.000	0.000	0.000

In Model 2, the first innovation accounts for virtually all forecast-error variance, just as in the true economy where the single structural shock θ_t drives everything.

The second and third innovations contribute negligibly.

This confirms that filtering strips away the measurement noise that created the appearance of multiple independent sources of variation in Model 1.

The covariance matrix and eigenvalues of the Model 2 innovations are

```
print('Covariance matrix of innovations:')
df_v2 = pd.DataFrame(np.round(V2, 4), index=labels, columns=labels)
display(Latex(df_to_latex_matrix(df_v2)))
```

```
Covariance matrix of innovations:
```

$$\begin{bmatrix} 0.9945 & 0.0474 & 0.9471 \\ 0.0474 & 0.0023 & 0.0452 \\ 0.9471 & 0.0452 & 0.9019 \end{bmatrix}$$

```
print('Eigenvalues of covariance matrix:')
print(np.sort(np.linalg.eigvalsh(V2))[::-1].round(4))
```

```
Eigenvalues of covariance matrix:
[1.899 0.    0.   ]
```

As Sargent [1989] emphasizes, the two models of measurement produce quite different inferences about the economy's dynamics despite sharing identical underlying parameters.

49.5.5 Wold impulse responses

We again use orthogonalized Wold representation impulse responses with a Cholesky decomposition of V_2 ordered as y_n, c, Δk.

```
parts = []
for i, title in enumerate(wold_titles):
    arr = df_to_latex_array(
      wold_response_table(resp2, i, lags)).strip('$')
    parts.append(
      r'\begin{array}{c} ' + title + r' \\ ' + arr + r' \end{array}')

display(Latex('$' + r' \quad '.join(parts) + '$'))
```

A. Response to y_n innovation

	y_n	c	Δk
0	0.997	0.048	0.950
1	0.048	0.048	0.000
2	0.048	0.048	0.000
3	0.048	0.048	0.000
4	0.048	0.048	0.000
5	0.048	0.048	0.000
6	0.048	0.048	0.000
7	0.048	0.048	0.000
8	0.048	0.048	0.000
9	0.048	0.048	0.000
10	0.048	0.048	0.000
11	0.048	0.048	0.000
12	0.048	0.048	0.000
13	0.048	0.048	0.000

B. Response to c innovation

	y_n	c	Δk
0	0.000	0.003	−0.002
1	0.002	0.002	0.000
2	0.002	0.002	0.000
3	0.002	0.002	0.000
4	0.002	0.002	0.000
5	0.002	0.002	0.000
6	0.002	0.002	0.000
7	0.002	0.002	0.000
8	0.002	0.002	0.000
9	0.002	0.002	0.000
10	0.002	0.002	0.000
11	0.002	0.002	0.000
12	0.002	0.002	0.000
13	0.002	0.002	0.000

C. Response to Δk innovation

	y_n	c	Δk
0	0.000	0.000	0.002
1	−0.001	−0.001	0.000
2	−0.001	−0.001	0.000
3	−0.001	−0.001	0.000
4	−0.001	−0.001	0.000
5	−0.001	−0.001	0.000
6	−0.001	−0.001	0.000
7	−0.001	−0.001	0.000
8	−0.001	−0.001	0.000
9	−0.001	−0.001	0.000
10	−0.001	−0.001	0.000
11	−0.001	−0.001	0.000
12	−0.001	−0.001	0.000
13	−0.001	−0.001	0.000

The income innovation in Model 2 produces responses that closely approximate the true impulse response function from the structural shock θ_t.

Readers can compare the left table with the table in the *True impulse responses* section above.

The numbers are essentially the same.

The consumption and investment innovations produce responses that are orders of magnitude smaller, confirming that the filtered data are driven by essentially one shock.

Unlike Model 1, the filtered data from Model 2 *cannot* reproduce the apparent Granger causality pattern that the accelerator literature has documented empirically.

Hence, at the population level, the two measurement models imply different empirical stories even though they share the same structural economy.

- In Model 1 (raw data), measurement noise creates multiple innovations and an apparent Granger-causality pattern.
- In Model 2 (filtered data), innovations collapse back to essentially one dominant shock, mirroring the true one-index economy.

Let's verify these implications in a finite sample simulation.

49.6 Simulation

The tables above characterize population moments of the two models.

Let's simulate 80 periods of true, measured, and filtered data to compare population implications with finite-sample behavior.

First, we define a function to simulate the true economy, generate measured data with AR(1) measurement errors, and apply the Model 1 Kalman filter to produce filtered estimates

```
def simulate_series(seed=7909, T=80, k0=10.0):
    """
    Simulate true, measured, and filtered series.
    """
    rng = np.random.default_rng(seed)

    # True state/observables
    θ = rng.normal(0.0, 1.0, size=T)
    k = np.empty(T + 1)
    k[0] = k0

    y = np.empty(T)
    c = np.empty(T)
    dk = np.empty(T)

    for t in range(T):
        x_t = np.array([k[t], θ[t]])
        y[t], c[t], dk[t] = C @ x_t
        k[t + 1] = k[t] + (1.0 / f) * θ[t]

    # Measured data with AR(1) errors
    v_prev = np.zeros(3)
    v = np.empty((T, 3))
    for t in range(T):
        η_t = rng.multivariate_normal(np.zeros(3), Σ_η)
        v_prev = D @ v_prev + η_t
        v[t] = v_prev

    z_meas = np.column_stack([y, c, dk]) + v

    # Filtered data via Model 1 transformed filter
    xhat_prev = np.array([k0, 0.0])
    z_prev = np.zeros(3)
    z_filt = np.empty((T, 3))
```

(continues on next page)

(continued from previous page)

```
    k_filt = np.empty(T)

    for t in range(T):
        z_bar_t = z_meas[t] - D @ z_prev
        u_t = z_bar_t - C_bar @ xhat_prev
        xhat_t = A @ xhat_prev + K1 @ u_t

        z_filt[t] = C @ xhat_t
        k_filt[t] = xhat_t[0]

        xhat_prev = xhat_t
        z_prev = z_meas[t]

    out = {
        "y_true": y, "c_true": c, "dk_true": dk, "k_true": k[:-1],
        "y_meas": z_meas[:, 0], "c_meas": z_meas[:, 1],
        "dk_meas": z_meas[:, 2],
        "y_filt": z_filt[:, 0], "c_filt": z_filt[:, 1],
        "dk_filt": z_filt[:, 2], "k_filt": k_filt
    }
    return out


sim = simulate_series(seed=7909, T=80, k0=10.0)
```

We use the following helper function to plot the true series against either the measured or filtered series

```
def plot_true_vs_other(t, true_series, other_series,
                                other_label, ylabel=""):
    fig, ax = plt.subplots(figsize=(8, 4))
    ax.plot(t, true_series, lw=2, label="true")
    ax.plot(t, other_series, lw=2, ls="--", label=other_label)
    ax.set_xlabel("time")
    ax.set_ylabel(ylabel)
    ax.legend()
    plt.tight_layout()
    plt.show()


t = np.arange(1, 81)
```

Let's first compare the true series with the measured series to see how measurement errors distort the data

```
plot_true_vs_other(t, sim["c_true"], sim["c_meas"],
                                    "measured", ylabel="consumption")
```

```
plot_true_vs_other(t, sim["dk_true"], sim["dk_meas"],
                                    "measured", ylabel="investment")
```

```
plot_true_vs_other(t, sim["y_true"], sim["y_meas"],
                                    "measured", ylabel="income")
```

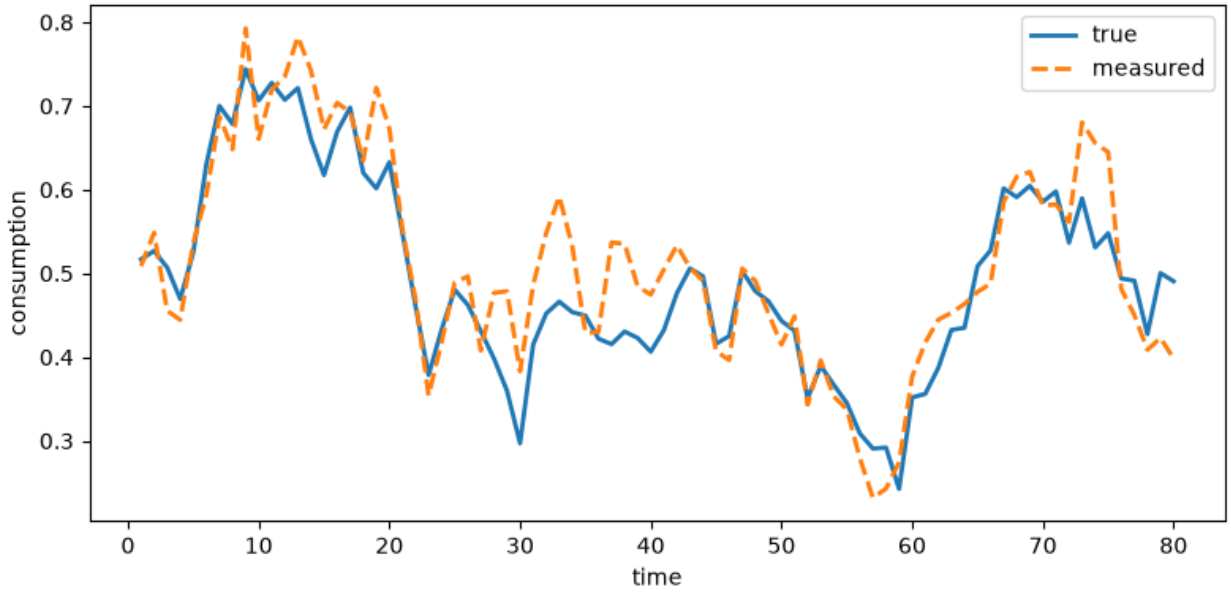


Fig. 49.1: True and measured consumption

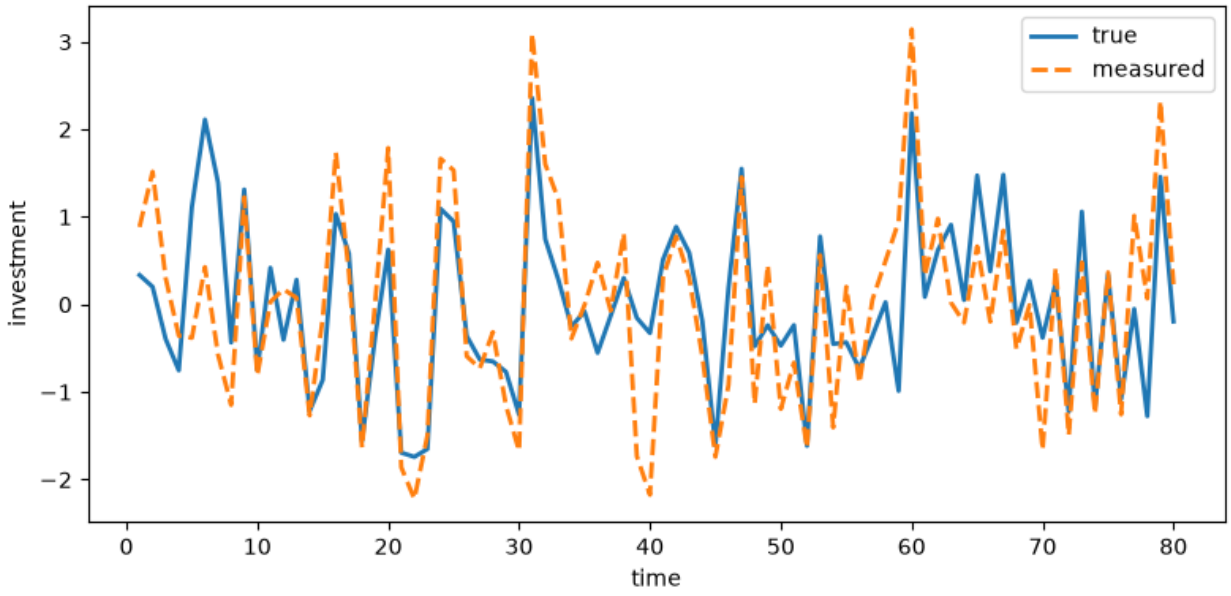


Fig. 49.2: True and measured investment

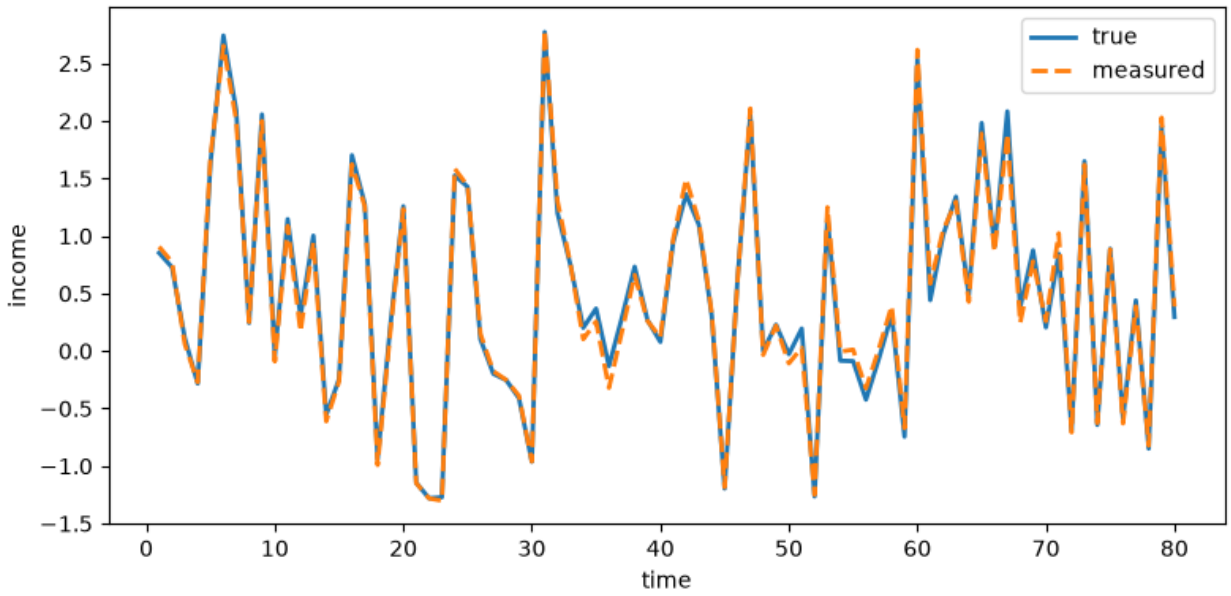


Fig. 49.3: True and measured income

Investment is distorted the most because its measurement error has the largest innovation variance ($\sigma_\eta = 0.65$), while income is distorted the least ($\sigma_\eta = 0.05$).

For the filtered series, we expect the Kalman filter to recover the true series more closely by stripping away measurement noise

```
plot_true_vs_other(t, sim["c_true"], sim["c_filt"],
                                    "filtered", ylabel="consumption")
```

```
plot_true_vs_other(t, sim["dk_true"], sim["dk_filt"],
                                    "filtered", ylabel="investment")
```

```
plot_true_vs_other(t, sim["y_true"], sim["y_filt"],
                                    "filtered", ylabel="income")
```

```
plot_true_vs_other(t, sim["k_true"], sim["k_filt"],
                                    "filtered", ylabel="capital stock")
```

Indeed, Kalman-filtered estimates from Model 1 remove much of the measurement noise and track the truth closely.

In the true model the national income identity $c_t + \Delta k_t = y_{n,t}$ holds exactly.

Independent measurement errors break this accounting identity in the measured data.

The Kalman filter approximately restores it.

The following figure confirms this by showing the residual $c_t + \Delta k_t - y_{n,t}$ for both measured and filtered data

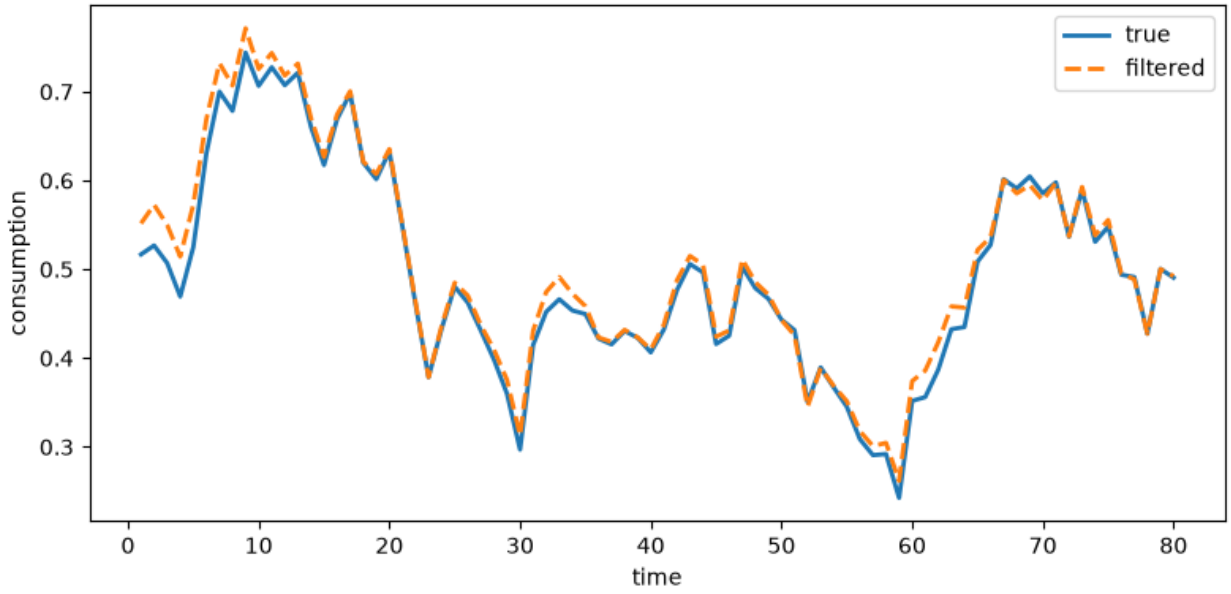


Fig. 49.4: True and filtered consumption

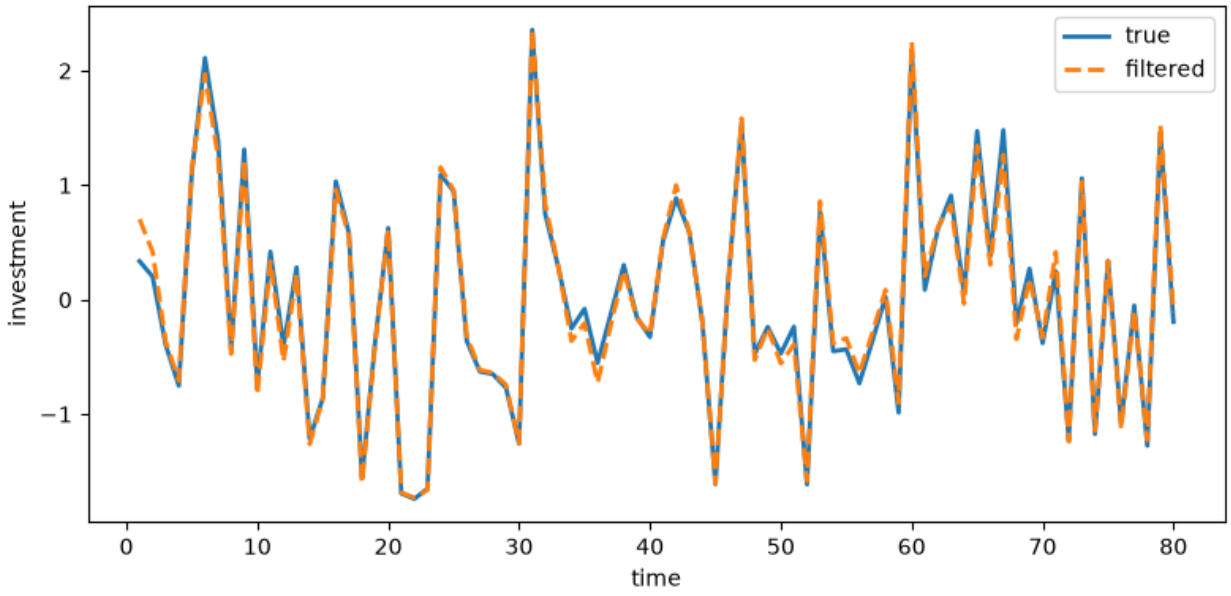


Fig. 49.5: True and filtered investment

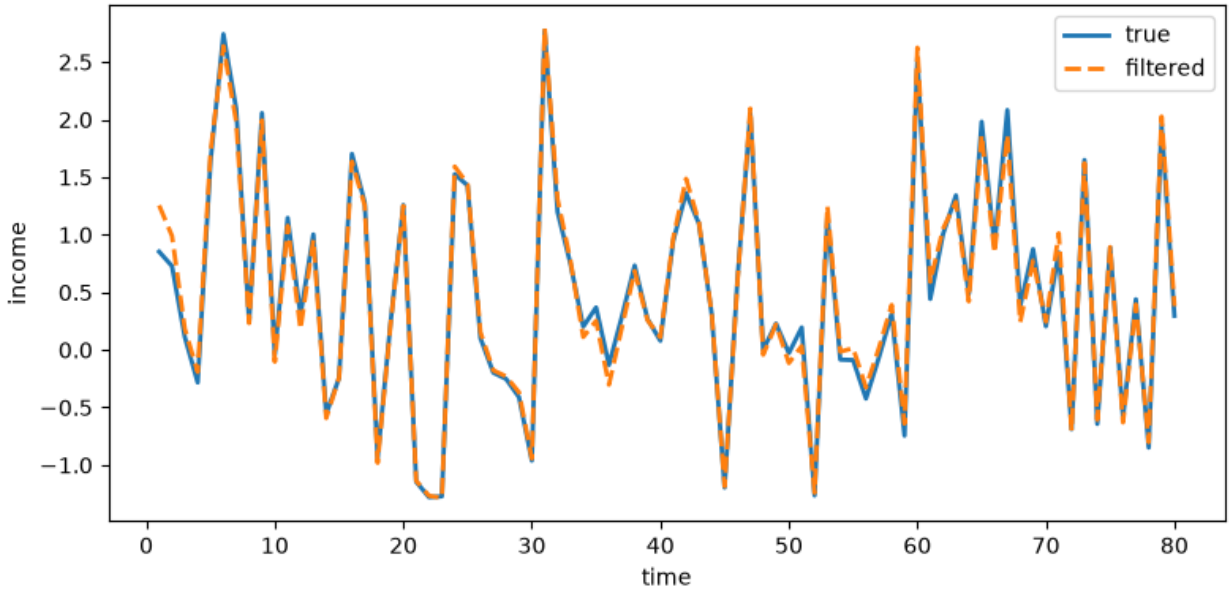


Fig. 49.6: True and filtered income

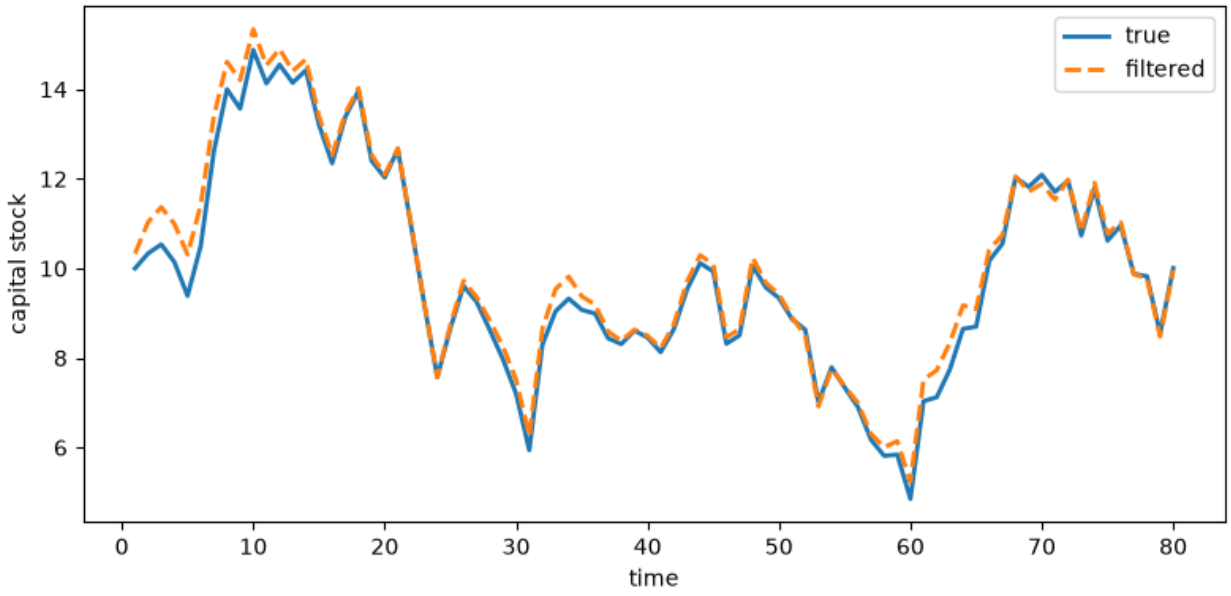


Fig. 49.7: True and filtered capital stock

```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 4))

ax1.plot(t, sim["c_meas"] + sim["dk_meas"] - sim["y_meas"], lw=2)
ax1.axhline(0, color='black', lw=0.8, ls='--', alpha=0.5)
ax1.set_xlabel("time")
ax1.set_ylabel("measured residual")

ax2.plot(t, sim["c_filt"] + sim["dk_filt"] - sim["y_filt"], lw=2)
ax2.axhline(0, color='black', lw=0.8, ls='--', alpha=0.5)
ax2.set_xlabel("time")
ax2.set_ylabel("filtered residual")

plt.tight_layout()
plt.show()
```

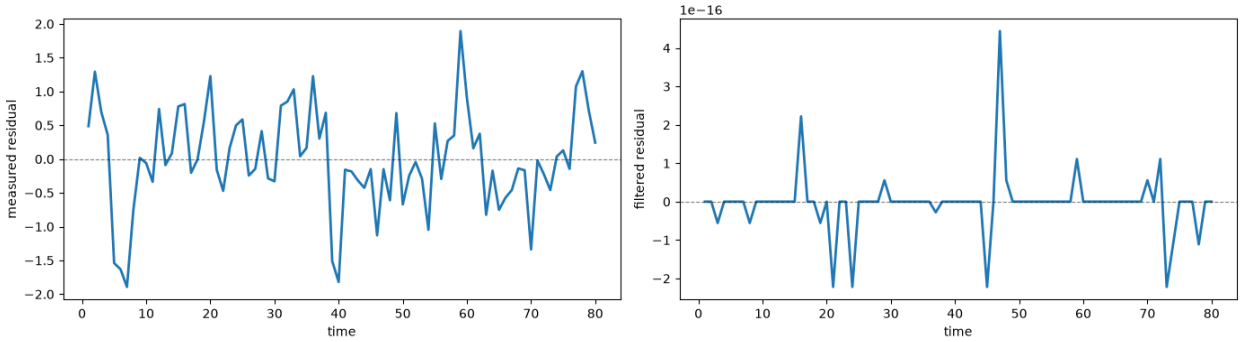


Fig. 49.8: National income identity residual

As we have predicted, the residual for the measured data is large and volatile, while the residual for the filtered data is numerically 0.

49.7 Summary

Sargent [1989] shows how measurement error alters an econometrician's view of a permanent income economy driven by the investment accelerator.

The Wold representations and variance decompositions of Model 1 (raw measurements) and Model 2 (filtered measurements) differ substantially, even though the underlying economy is the same.

Measurement error can reshape inferences about which shocks drive which variables.

Model 1 reproduces the **Granger causality** pattern documented in the empirical accelerator literature: income appears to Granger-cause consumption and investment, a result Sargent [1989] attributes to measurement error and signal extraction in raw reported data.

Model 2, working with filtered data, attributes nearly all variance to the single structural shock θ_t and *cannot* reproduce the Granger causality pattern.

The *Kalman filter* effectively strips measurement noise from the data, so the filtered series track the truth closely.

Raw measurement error breaks the national income accounting identity, but the near-zero residual shows that the filter approximately restores it.

Part VII

Search

CHAPTER

FIFTY

JOB SEARCH I: THE MCCALL SEARCH MODEL

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

Contents

- *Job Search I: The McCall Search Model*
  - *Overview*
  - *The McCall Model*
  - *Computing the Optimal Policy: Take 1*
  - *Computing an Optimal Policy: Take 2*
  - *Continuous Offer Distribution*
  - *Volatility*
  - *Exercises*

> "Questioning a McCall worker is like having a conversation with an out-of-work friend: 'Maybe you are setting your sights too high', or 'Why did you quit your old job before you had a new one lined up?' This is real social science: an attempt to model, to understand, human behavior by visualizing the situation people find themselves in, the options they face and the pros and cons as they themselves see them." – Robert E. Lucas, Jr.

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install quantecon jax
```

50.1 Overview

The McCall search model [McCall, 1970] helped transform economists' way of thinking about labor markets.

To clarify notions such as "involuntary" unemployment, McCall modeled the decision problem of an unemployed worker in terms of factors including

- current and likely future wages
- impatience
- unemployment compensation

To solve the decision problem McCall used dynamic programming.

Here we set up McCall's model and use dynamic programming to analyze it.

As we'll see, McCall's model is not only interesting in its own right but also an excellent vehicle for learning dynamic programming.

Let's start with some imports:

```
import matplotlib.pyplot as plt
import numpy as np
import numba
import jax
import jax.numpy as jnp
from typing import NamedTuple
from functools import partial
import quantecon as qe
from quantecon.distributions import BetaBinomial
```

50.2 The McCall Model

An unemployed agent receives in each period a job offer at wage W_t.

In this lecture, we adopt the following simple environment:

- The offer sequence $\{W_t\}_{t \geq 0}$ is IID, with $q(w)$ being the probability of observing wage w in finite set $\mathbb{W}$.
- The agent observes W_t at the start of t.
- The agent knows that $\{W_t\}$ is IID with common distribution q and can use this when computing expectations.

(In later lectures, we will relax these assumptions.)

At time t, our agent has two choices:

1. Accept the offer and work permanently at constant wage W_t.
2. Reject the offer, receive unemployment compensation c, and reconsider next period.

The agent is infinitely lived and aims to maximize the expected discounted sum of earnings

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t y_t \tag{50.1}$$

The constant β lies in $(0, 1)$ and is called a **discount factor**.

The smaller is β, the more the agent discounts future earnings relative to current earnings.

The variable y_t is income, equal to

- his/her wage W_t when employed
- unemployment compensation c when unemployed

50.2.1 A Trade-Off

The worker faces a trade-off:

- Waiting too long for a good offer is costly, since the future is discounted.
- Accepting too early is costly, since better offers might arrive in the future.

To decide the optimal wait time in the face of this trade-off, we use dynamic programming.

Dynamic programming can be thought of as a two-step procedure that

1. first assigns values to "states" and
2. then deduces optimal actions given those values

We'll go through these steps in turn.

50.2.2 The Value Function

In order to optimally trade-off current and future rewards, we need to think about two things:

1. the current payoffs we get from different choices
2. the different states that those choices will lead to in next period

To weigh these two aspects of the decision problem, we need to assign *values* to states.

To this end, let $v^*(w)$ be the total lifetime value accruing to an unemployed worker who enters the current period unemployed when the wage is $w \in \mathbb{W}$.

(In particular, the agent has wage offer w in hand and can accept or reject it.)

More precisely, $v^*(w)$ denotes the total sum of expected discounted earnings when an agent always behaves in an optimal way. points in time.

Of course $v^*(w)$ is not trivial to calculate because we don't yet know what decisions are optimal and what aren't!

If we don't know what opimal choices are, it feels imposible to calculate $v^*(w)$.

But let's put this aside for now and think of v^* as a function that assigns to each possible wage w the maximal lifetime value $v^*(w)$ that can be obtained with that offer in hand.

A crucial observation is that this function v^* must satisfy

$$v^*(w) = \max \left\{ \frac{w}{1-\beta}, \, c + \beta \sum_{w' \in \mathbb{W}} v^*(w') q(w') \right\} \tag{50.2}$$

for every possible w in $\mathbb{W}$.

This is a version of the **Bellman equation**, which is ubiquitous in economic dynamics and other fields involving planning over time.

The intuition behind it is as follows:

- the first term inside the max operation is the lifetime payoff from accepting current offer, since such a worker works forever at w and values this income stream as

$$\frac{w}{1-\beta} = w + \beta w + \beta^2 w + \cdots$$

- the second term inside the max operation is the continuation value, which is the lifetime payoff from rejecting the current offer and then behaving optimally in all subsequent periods

If we optimize and pick the best of these two options, we obtain maximal lifetime value from today, given current offer w.

But this is precisely $v^*(w)$, which is the left-hand side of (50.2).

Putting this all together, we see that (50.2) is valid for all w.

50.2.3 The Optimal Policy

We still don't know how to compute v^* (although (50.2) gives us hints we'll return to below).

But suppose for now that we do know v^*.

Once we have this function in hand we can easily make optimal choices (i.e., make the right choice between accept and reject given any w).

All we have to do is select the maximal choice on the right-hand side of (50.2).

In other words, we make the best choice between stopping and continuing, given the information provided to us by v^*.

The optimal action is best thought of as a **policy**, which is, in general, a map from states to actions.

Given any w, we can read off the corresponding best choice (accept or reject) by picking the max on the right-hand side of (50.2).

Thus, we have a map from $\mathbb{W}$ to $\{0, 1\}$, with 1 meaning accept and 0 meaning reject.

We can write the policy as follows

$$\sigma(w) := \mathbf{1}\left\{\frac{w}{1-\beta} \geq c + \beta \sum_{w' \in \mathbb{W}} v^*(w')q(w')\right\}$$

Here $\mathbf{1}\{P\} = 1$ if statement P is true and equals 0 otherwise.

We can also write this as

$$\sigma(w) := \mathbf{1}\{w \geq \bar{w}\}$$

where

$$\bar{w} := (1-\beta)\left\{c + \beta \sum_{w'} v^*(w')q(w')\right\} \tag{50.3}$$

Here $\bar{w}$ (called the **reservation wage**) is a constant depending on β, c and the wage distribution.

The agent should accept if and only if the current wage offer exceeds the reservation wage.

In view of (50.3), we can compute this reservation wage if we can compute the value function.

50.3 Computing the Optimal Policy: Take 1

To put the above ideas into action, we need to compute the value function at each $w \in \mathbb{W}$.

To simplify notation, let's set

$$\mathbb{W} := \{w_1, \ldots, w_n\} \quad \text{and} \quad v^*(i) := v^*(w_i)$$

The value function is then represented by the vector $v^* = (v^*(i))_{i=1}^n$.

In view of (50.2), this vector satisfies the nonlinear system of equations

$$v^*(i) = \max\left\{\frac{w(i)}{1-\beta},\, c + \beta \sum_{j=1}^n v^*(j) q(j)\right\} \quad \text{for } i = 1, \ldots, n \tag{50.4}$$

50.3.1 The Algorithm

To compute this vector, we use successive approximations:

Step 1: pick an arbitrary initial guess $v \in \mathbb{R}^n$.

Step 2: compute a new vector $v' \in \mathbb{R}^n$ via

$$v'(i) = \max\left\{\frac{w(i)}{1-\beta},\, c + \beta \sum_{j=1}^n v(j) q(j)\right\} \quad \text{for } i = 1, \ldots, n \tag{50.5}$$

Step 3: calculate a measure of a discrepancy between v and v', such as $\max_i |v(i) - v'(i)|$.

Step 4: if the deviation is larger than some fixed tolerance, set $v = v'$ and go to step 2, else continue.

Step 5: return v.

For a small tolerance, the returned function v is a close approximation to the value function v^*.

The theory below elaborates on this point.

50.3.2 Fixed Point Theory

What's the mathematics behind these ideas?

First, one defines a mapping T from $\mathbb{R}^n$ to itself via

$$(Tv)(i) = \max\left\{\frac{w(i)}{1-\beta},\, c + \beta \sum_{j=1}^n v(j) q(j)\right\} \quad \text{for } i = 1, \ldots, n \tag{50.6}$$

(A new vector Tv is obtained from given vector v by evaluating the r.h.s. at each i.)

The element v_k in the sequence $\{v_k\}$ of successive approximations corresponds to $T^k v$.

- This is T applied k times, starting at the initial guess v

One can show that the conditions of the Banach fixed point theorem are satisfied by T on $\mathbb{R}^n$.

One implication is that T has a unique fixed point in $\mathbb{R}^n$.

- That is, a unique vector $\bar{v}$ such that $T\bar{v} = \bar{v}$.

Moreover, it's immediate from the definition of T that this fixed point is v^*.

A second implication of the Banach contraction mapping theorem is that $\{T^k v\}$ converges to the fixed point v^* regardless of v.

50.3.3 Implementation

Our default for q, the wage offer distribution, will be Beta-binomial.

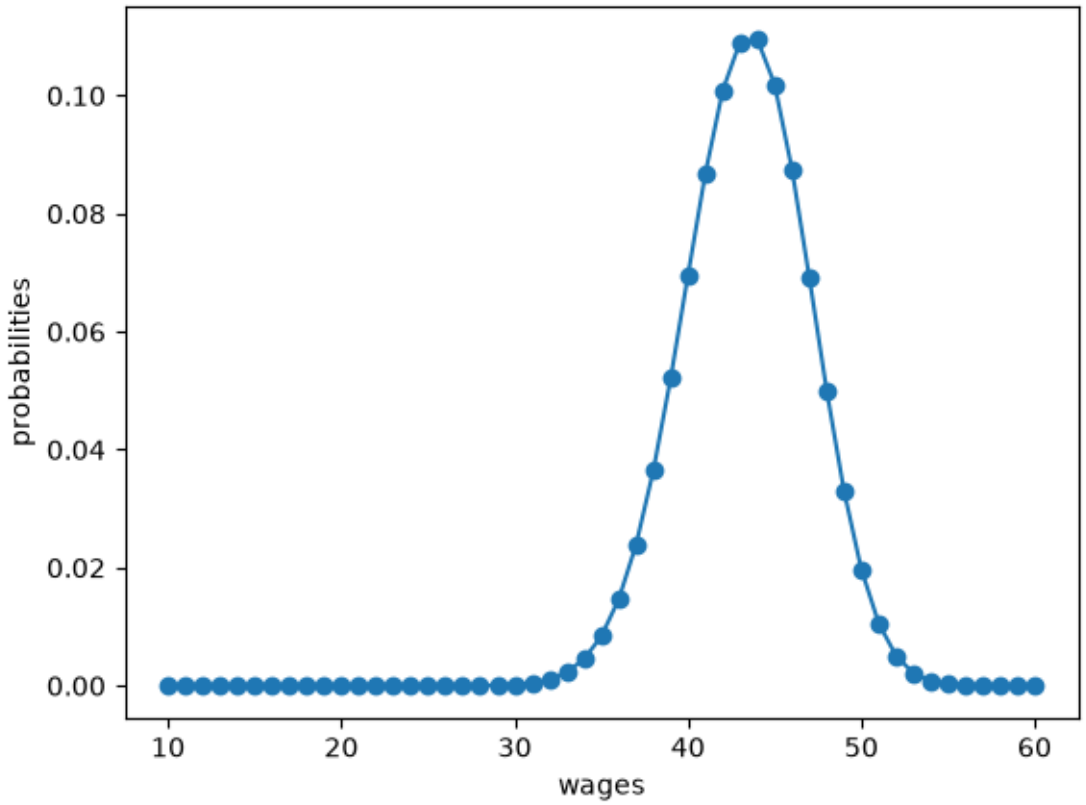
```
n, a, b = 50, 200, 100                          # default parameters
q_default = jnp.array(BetaBinomial(n, a, b).pdf())
```

Our default set of values for wages will be

```
w_min, w_max = 10, 60
w_default = jnp.linspace(w_min, w_max, n+1)
```

Here's a plot of the probabilities of different wage outcomes:

```
fig, ax = plt.subplots()
ax.plot(w_default, q_default, '-o', label='$q(w(i))$')
ax.set_xlabel('wages')
ax.set_ylabel('probabilities')

plt.show()
```



We will use JAX to write our code.

We'll use `NamedTuple` for our model class to maintain immutability, which works well with JAX's functional programming paradigm.

Here's a class that stores the model parameters with default values.

```
class McCallModel(NamedTuple):
    c: float = 25                # unemployment compensation
    β: float = 0.99              # discount factor
    w: jnp.ndarray = w_default   # array of wage values, w[i] = wage at state i
    q: jnp.ndarray = q_default   # array of probabilities
```

We implement the Bellman operator T from (50.6), which we can write in terms of array operations as

$$Tv = \max \left\{ \frac{w}{1-\beta}, \, c + \beta \sum_{j=1}^{n} v(j) q(j) \right\} \tag{50.7}$$

(The first term inside the max is an array and the second is just a number – here we mean that the max comparison against this number is done element-by-element for all elements in the array.)

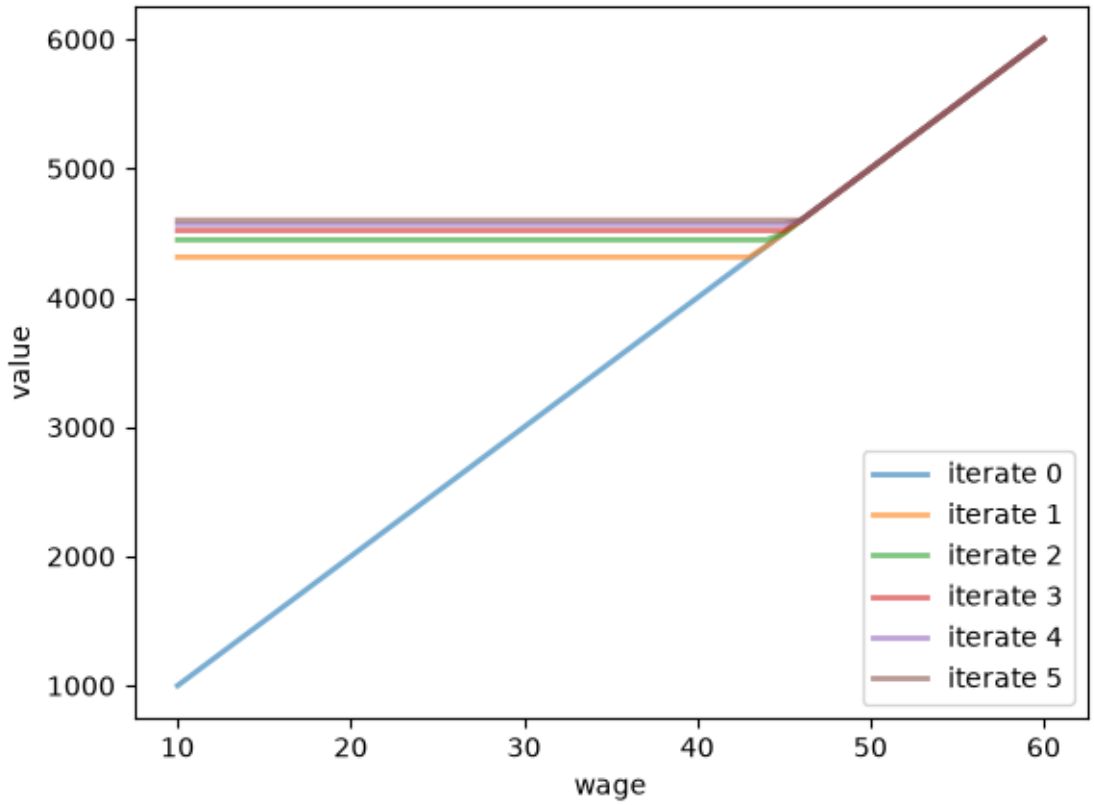
We can code T up as follows.

```
def T(model: McCallModel, v: jnp.ndarray):
    c, β, w, q = model
    accept = w / (1 - β)
    reject = c + β * v @ q
    return jnp.maximum(accept, reject)
```

Based on these defaults, let's try plotting the first few approximate value functions in the sequence $\{T^k v\}$.

We will start from guess v given by $v(i) = w(i)/(1-\beta)$, which is the value of accepting at every given wage.

```
model = McCallModel()
c, β, w, q = model
v = w / (1 - β)    # Initial condition
fig, ax = plt.subplots()

num_plots = 6
for i in range(num_plots):
    ax.plot(w, v, '-', alpha=0.6, lw=2, label=f"iterate {i}")
    v = T(model, v)

ax.legend(loc='lower right')
ax.set_xlabel('wage')
ax.set_ylabel('value')
plt.show()
```



You can see that convergence is occurring: successive iterates are getting closer together.

Here's a more serious iteration effort to compute the limit, which continues until measured deviation between successive iterates is below `tol`.

Once we obtain a good approximation to the limit, we will use it to calculate the reservation wage.

```
def compute_reservation_wage(
        model: McCallModel,    # instance containing default parameters
        v_init: jnp.ndarray,   # initial condition for iteration
        tol: float=1e-6,       # error tolerance
        max_iter: int=500,     # maximum number of iterations for loop
    ):
    "Computes the reservation wage in the McCall job search model."
    c, β, w, q = model
    i = 0
    error = tol + 1
    v = v_init

    while i < max_iter and error > tol:
        v_next = T(model, v)
        error = jnp.max(jnp.abs(v_next - v))
        v = v_next
        i += 1

    w_bar = (1 - β) * (c + β * v @ q)
    return v, w_bar
```

The cell computes the reservation wage at the default parameters

```
model = McCallModel()
c, β, w, q = model
v_init = w / (1 - β)  # initial guess
v, w_bar = compute_reservation_wage(model, v_init)
print(w_bar)
```

```
47.316475
```

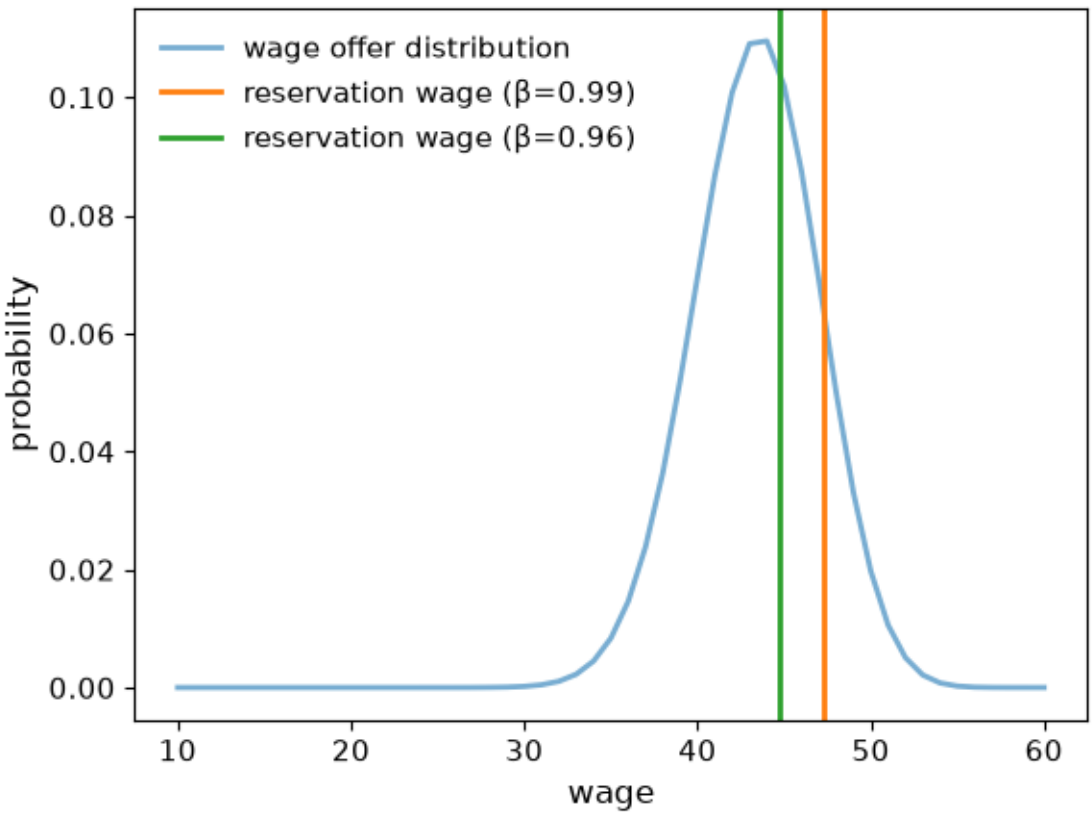
50.3.4 Comparative Statics

Now that we know how to compute the reservation wage, let's see how it varies with parameters.

Here we compare the reservation wage at two values of β.

The reservation wages will be plotted alongside the wage offer distribution, so that we can get a sense of what fraction of offers will be accepted.

```
fig, ax = plt.subplots()

# Get the default color cycle
prop_cycle = plt.rcParams['axes.prop_cycle']
colors = prop_cycle.by_key()['color']

# Plot the wage offer distribution
ax.plot(w, q, '-', alpha=0.6, lw=2,
        label='wage offer distribution',
        color=colors[0])

# Compute reservation wage with default beta
model_default = McCallModel()
c, β, w, q = model_default
v_init = w / (1 - β)
v_default, res_wage_default = compute_reservation_wage(
    model_default, v_init
)

# Compute reservation wage with lower beta
β_new = 0.96
model_low_beta = McCallModel(β=β_new)
c, β_low, w, q = model_low_beta
v_init_low = w / (1 - β_low)
v_low, res_wage_low = compute_reservation_wage(
    model_low_beta, v_init_low
)

# Plot vertical lines for reservation wages
ax.axvline(x=res_wage_default, color=colors[1], lw=2,
           label=f'reservation wage (β={β})')
ax.axvline(x=res_wage_low, color=colors[2], lw=2,
           label=f'reservation wage (β={β_new})')

ax.set_xlabel('wage', fontsize=12)
ax.set_ylabel('probability', fontsize=12)
ax.tick_params(axis='both', which='major', labelsize=11)
ax.legend(loc='upper left', frameon=False, fontsize=11)
plt.show()
```



We see that the reservation wage is higher when β is higher.

This is not surprising, since higher β is associated with more patience.

Now let's look more systematically at what happens when we change β and c.

As a first step, given that we'll use it many times, let's create a more efficient, jit-complied version of the function that computes the reservation wage:

```
@jax.jit
def compute_res_wage_jitted(
        model: McCallModel,    # instance containing default parameters
        v_init: jnp.ndarray,   # initial condition for iteration
        tol: float=1e-6,       # error tolerance
        max_iter: int=500,     # maximum number of iterations for loop
    ):
    c, β, w, q = model
    i = 0
    error = tol + 1
    initial_state = v_init, i, error

    def cond(loop_state):
        v, i, error = loop_state
        return jnp.logical_and(i < max_iter, error > tol)

    def update(loop_state):
        v, i, error = loop_state
        v_next = T(model, v)
        error = jnp.max(jnp.abs(v_next - v))
        i += 1
```

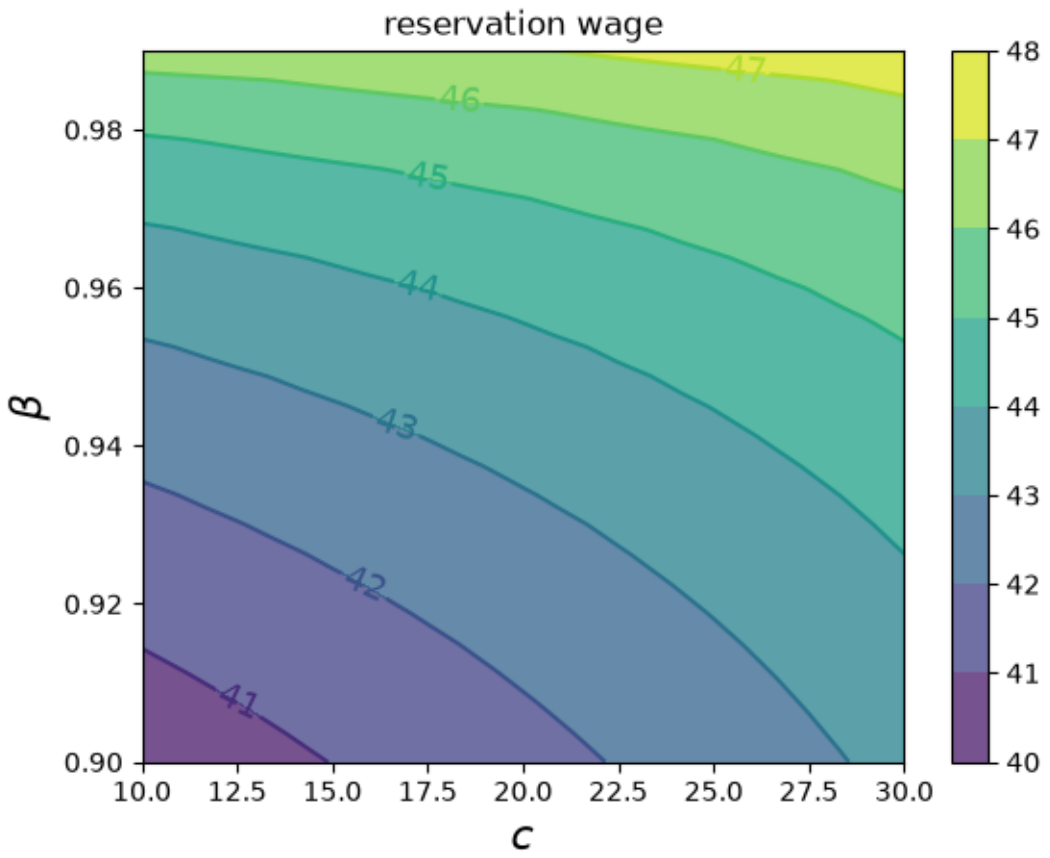
(continues on next page)

(continued from previous page)

```
        new_loop_state = v_next, i, error
        return new_loop_state

    final_state = jax.lax.while_loop(cond, update, initial_state)
    v, i, error = final_state

    w_bar = (1 - β) * (c + β * v @ q)
    return v, w_bar
```

Now we compute the reservation wage at each c, β pair.

```
grid_size = 25
c_vals = jnp.linspace(10.0, 30.0, grid_size)
β_vals = jnp.linspace(0.9, 0.99, grid_size)

res_wage_matrix = np.empty((grid_size, grid_size))
model = McCallModel()
v_init = model.w / (1 - model.β)

for i, c in enumerate(c_vals):
    for j, β in enumerate(β_vals):
        model = McCallModel(c=c, β=β)
        v, w_bar = compute_res_wage_jitted(model, v_init)
        v_init = v
        res_wage_matrix[i, j] = w_bar

fig, ax = plt.subplots()
cs1 = ax.contourf(c_vals, β_vals, res_wage_matrix.T, alpha=0.75)
ctr1 = ax.contour(c_vals, β_vals, res_wage_matrix.T)
plt.clabel(ctr1, inline=1, fontsize=13)
plt.colorbar(cs1, ax=ax)
ax.set_title("reservation wage")
ax.set_xlabel("$c$", fontsize=16)
ax.set_ylabel("$β$", fontsize=16)
ax.ticklabel_format(useOffset=False)
plt.show()
```



As expected, the reservation wage increases with both patience and unemployment compensation.

50.4 Computing an Optimal Policy: Take 2

The approach to dynamic programming just described is standard and broadly applicable.

But for our McCall search model there's also an easier way that circumvents the need to compute the value function.

Let h denote the continuation value:

$$h = c + \beta \sum_{w'} v^*(w') q(w') \tag{50.8}$$

The Bellman equation can now be written as

$$v^*(w') = \max \left\{ \frac{w'}{1-\beta}, \, h \right\} \tag{50.9}$$

Now let's derive a nonlinear equation for h alone.

Starting from (50.9), we multiply both sides by $q(w')$ to get

$$v^*(w') q(w') = \max \left\{ \frac{w'}{1-\beta}, h \right\} q(w')$$

Next, we sum both sides over $w' \in \mathbb{W}$:

$$\sum_{w' \in \mathbb{W}} v^*(w') q(w') = \sum_{w' \in \mathbb{W}} \max \left\{ \frac{w'}{1-\beta}, h \right\} q(w')$$

Now multiply both sides by β:

$$\beta \sum_{w' \in \mathbb{W}} v^*(w') q(w') = \beta \sum_{w' \in \mathbb{W}} \max \left\{ \frac{w'}{1-\beta}, h \right\} q(w')$$

Add c to both sides:

$$c + \beta \sum_{w' \in \mathbb{W}} v^*(w') q(w') = c + \beta \sum_{w' \in \mathbb{W}} \max \left\{ \frac{w'}{1-\beta}, h \right\} q(w')$$

Finally, using the definition of h from (50.8), the left-hand side is just h, giving us

$$h = c + \beta \sum_{w' \in \mathbb{W}} \max \left\{ \frac{w'}{1-\beta}, h \right\} q(w') \tag{50.10}$$

This is a nonlinear equation in the single scalar h that we can solve for h.

As before, we will use successive approximations:

Step 1: pick an initial guess h.

Step 2: compute the update h' via

$$h' = c + \beta \sum_{w' \in \mathbb{W}} \max \left\{ \frac{w'}{1-\beta}, h \right\} q(w') \tag{50.11}$$

Step 3: calculate the deviation $|h - h'|$.

Step 4: if the deviation is larger than some fixed tolerance, set $h = h'$ and go to step 2, else return h.

One can again use the Banach contraction mapping theorem to show that this process always converges.

The big difference here, however, is that we're iterating on a scalar h, rather than an n-vector, $v(i), i = 1, \ldots, n$.

Here's an implementation:

```
def compute_reservation_wage_two(
        model: McCallModel,    # instance containing default parameters
        tol: float=1e-5,       # error tolerance
        max_iter: int=500,     # maximum number of iterations for loop
    ):
    c, β, w, q = model
    h = (w @ q) / (1 - β)  # initial condition
    i = 0
    error = tol + 1
    initial_loop_state = i, h, error

    def cond(loop_state):
        i, h, error = loop_state
        return jnp.logical_and(i < max_iter, error > tol)

    def update(loop_state):
        i, h, error = loop_state
        s = jnp.maximum(w / (1 - β), h)
        h_next = c + β * (s @ q)
        error = jnp.abs(h_next - h)
        i_next = i + 1
        new_loop_state = i_next, h_next, error
        return new_loop_state
```

(continues on next page)

(continued from previous page)

```
    final_state = jax.lax.while_loop(cond, update, initial_loop_state)
    i, h, error = final_state

    # Compute and return the reservation wage
    return (1 - β) * h
```

You can use this code to solve the exercise below.

50.5 Continuous Offer Distribution

The discrete wage offer distribution used above is convenient for theory and computation, but many realistic distributions are continuous (i.e., have a density).

Fortunately, the theory changes little in our simple model when we shift to a continuous offer distribution.

Recall that h in (50.8) denotes the value of not accepting a job in this period but then behaving optimally in all subsequent periods.

To shift to a continuous offer distribution, we can replace (50.8) by

$$h = c + \beta \int v^*(s') q(s') ds'. \tag{50.12}$$

Equation (50.10) becomes

$$h = c + \beta \int \max \left\{ \frac{w(s')}{1-\beta}, h \right\} q(s') ds' \tag{50.13}$$

The aim is to solve this nonlinear equation by iteration, and from it obtain the reservation wage.

50.5.1 Implementation with Lognormal Wages

Let's implement this for the case where

- the state sequence $\{s_t\}$ is IID and standard normal and
- the wage function is $w(s) = \exp(\mu + \sigma s)$.

This gives us a lognormal wage distribution.

We use Monte Carlo integration to evaluate the integral, averaging over a large number of wage draws.

For default parameters, we use `c=25, β=0.99, σ=0.5, μ=2.5`.

```
class McCallModelContinuous(NamedTuple):
    c: float              # unemployment compensation
    β: float              # discount factor
    σ: float              # scale parameter in lognormal distribution
    μ: float              # location parameter in lognormal distribution
    w_draws: jnp.ndarray  # draws of wages for Monte Carlo


def create_mccall_continuous(
        c=25, β=0.99, σ=0.5, μ=2.5, mc_size=1000, seed=1234
    ):
    key = jax.random.PRNGKey(seed)
```

(continues on next page)

(continued from previous page)

```
    s = jax.random.normal(key, (mc_size,))
    w_draws = jnp.exp(μ + σ * s)
    return McCallModelContinuous(c, β, σ, μ, w_draws)


@jax.jit
def compute_reservation_wage_continuous(model, max_iter=500, tol=1e-5):
    c, β, σ, μ, w_draws = model

    h = jnp.mean(w_draws) / (1 - β)  # initial guess

    def update(state):
        h, i, error = state
        integral = jnp.mean(jnp.maximum(w_draws / (1 - β), h))
        h_next = c + β * integral
        error = jnp.abs(h_next - h)
        return h_next, i + 1, error

    def cond(state):
        h, i, error = state
        return jnp.logical_and(i < max_iter, error > tol)

    initial_state = (h, 0, tol + 1)
    final_state = jax.lax.while_loop(cond, update, initial_state)
    h_final, _, _ = final_state

    # Now compute the reservation wage
    return (1 - β) * h_final
```

Now let's investigate how the reservation wage changes with c and β using a contour plot.

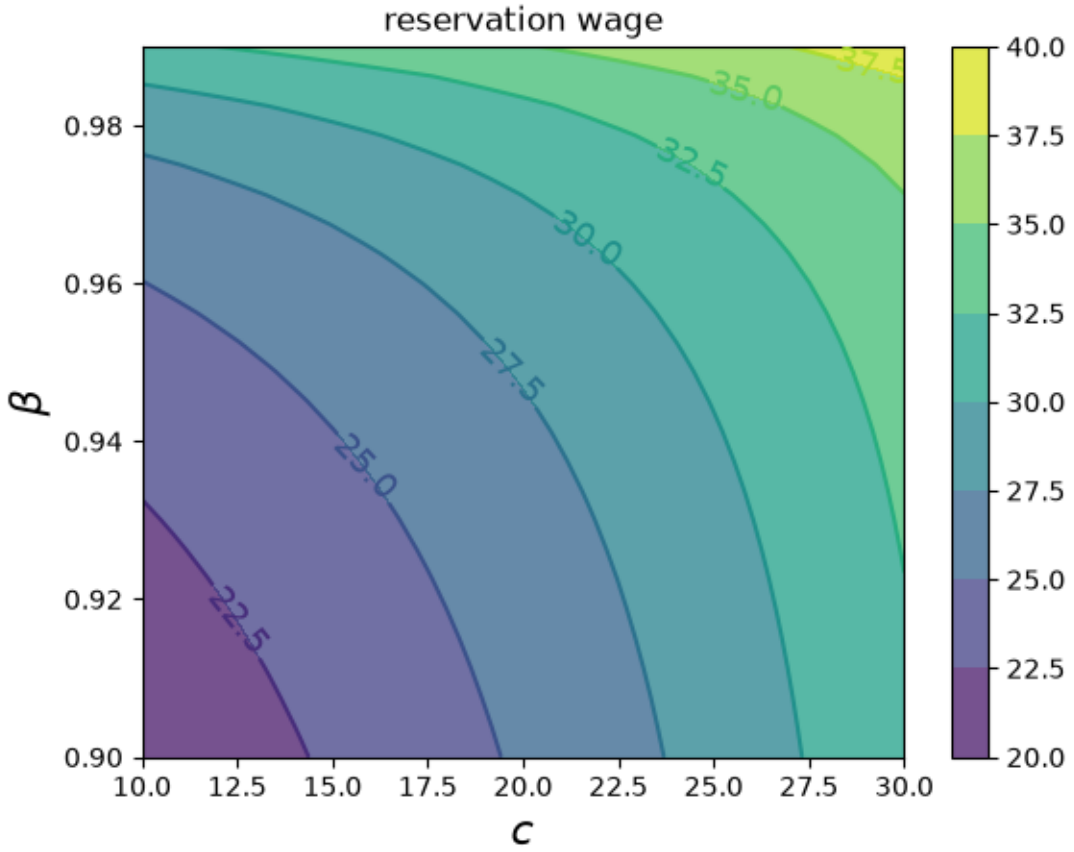
```
grid_size = 25
c_vals = jnp.linspace(10.0, 30.0, grid_size)
β_vals = jnp.linspace(0.9, 0.99, grid_size)

def compute_R_element(c, β):
    model = create_mccall_continuous(c=c, β=β)
    return compute_reservation_wage_continuous(model)

# First, vectorize over β (holding c fixed)
compute_R_over_β = jax.vmap(compute_R_element, in_axes=(None, 0))

# Next, vectorize over c (applying the above function to each c)
compute_R_vectorized = jax.vmap(compute_R_over_β, in_axes=(0, None))

# Apply to compute the full grid
R = compute_R_vectorized(c_vals, β_vals)
```

```
fig, ax = plt.subplots()

cs1 = ax.contourf(c_vals, β_vals, R.T, alpha=0.75)
ctr1 = ax.contour(c_vals, β_vals, R.T)

plt.clabel(ctr1, inline=1, fontsize=13)
plt.colorbar(cs1, ax=ax)
```

(continues on next page)

(continued from previous page)

```
ax.set_title("reservation wage")
ax.set_xlabel("$c$", fontsize=16)
ax.set_ylabel("$β$", fontsize=16)

ax.ticklabel_format(useOffset=False)

plt.show()
```



As with the discrete case, the reservation wage increases with both patience and unemployment compensation.

50.6 Volatility

An interesting feature of the McCall model is that increased volatility in wage offers tends to increase the reservation wage.

The intuition is that volatility is attractive to the worker because they can enjoy the upside (high wage offers) while rejecting the downside (low wage offers).

Hence, with more volatility, workers are more willing to continue searching rather than accept a given offer, which means the reservation wage rises.

To illustrate this phenomenon, we use a mean-preserving spread of the wage distribution.

In particular, we vary the scale parameter σ in the lognormal wage distribution $w(s) = \exp(\mu + \sigma s)$ while adjusting μ to keep the mean constant.

Recall that for a lognormal distribution with parameters μ and σ, the mean is $\exp(\mu + \sigma^2/2)$.

To keep the mean constant at some value m, we need:

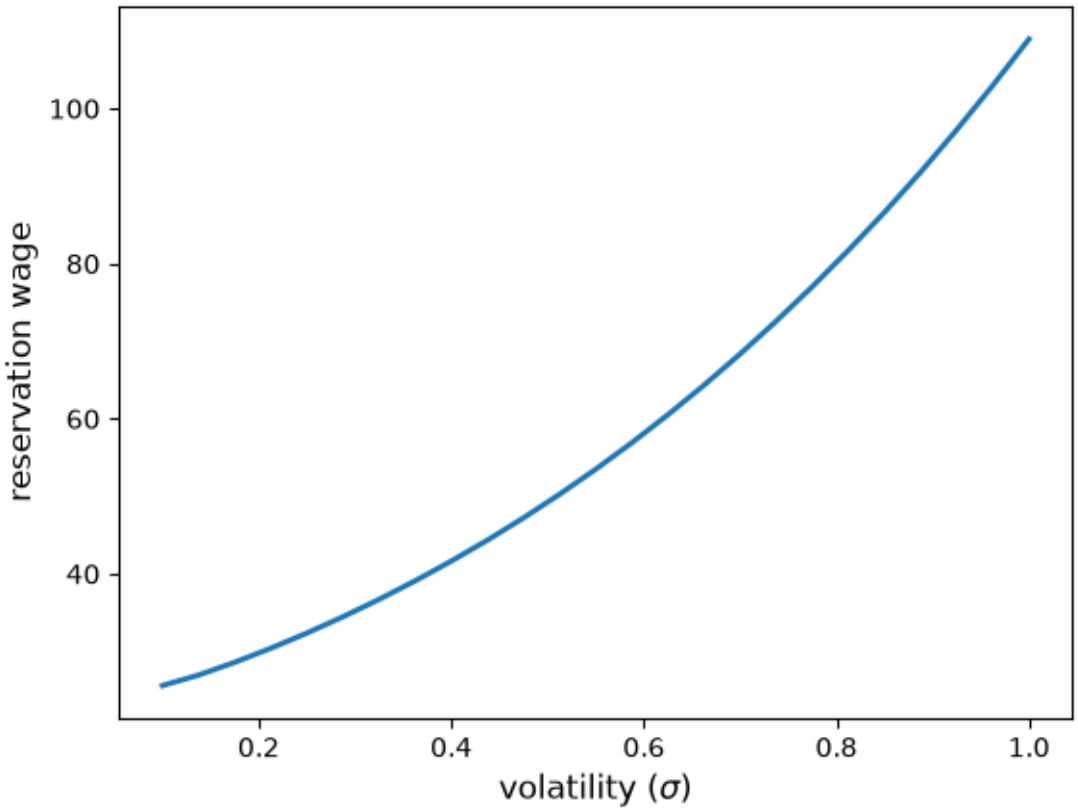
$$\mu = \ln(m) - \frac{\sigma^2}{2}$$

Let's implement this and compute the reservation wage for different values of σ:

```
# Fix the mean wage
mean_wage = 20.0

# Create a range of volatility values
σ_vals = jnp.linspace(0.1, 1.0, 25)

# Given σ, compute μ to maintain constant mean
def compute_μ_for_mean(σ, mean_wage):
    return jnp.log(mean_wage) - (σ**2) / 2

# Compute reservation wage for each volatility level
res_wages_volatility = []

for σ in σ_vals:
    μ = compute_μ_for_mean(σ, mean_wage)
    model = create_mccall_continuous(σ=float(σ), μ=float(μ))
    w_bar = compute_reservation_wage_continuous(model)
    res_wages_volatility.append(w_bar)

res_wages_volatility = jnp.array(res_wages_volatility)
```

Now let's plot the reservation wage as a function of volatility:

```
fig, ax = plt.subplots()
ax.plot(σ_vals, res_wages_volatility, linewidth=2)
ax.set_xlabel(r'volatility ($\sigma$)', fontsize=12)
ax.set_ylabel('reservation wage', fontsize=12)
plt.show()
```



As expected, the reservation wage is increasing in σ.

50.6.1 Lifetime Value and Volatility

We've seen that the reservation wage increases with volatility.

It's also the case that maximal lifetime value increases with volatility.

Higher volatility provides more upside potential, while at the same time workers can protect themselves against downside risk by rejecting low offers.

This option value translates into higher expected lifetime utility.

To demonstrate this, we will:

1. Compute the reservation wage for each volatility level
2. Calculate the expected discounted value of the lifetime income stream associated with that reservation wage, using Monte Carlo.

The simulation works as follows:

1. Compute the present discounted value of one lifetime earnings path, from a given wage path.
2. Average over a large number of such calculations to approximate expected discounted value.

We truncate each path at $T = 100$, which provides sufficient resolution for our purposes.

```
@jax.jit
def simulate_lifetime_value(key, model, w_bar, n_periods=100):
    """
```

(continues on next page)

(continued from previous page)

```
    Simulate one realization of the wage path and compute lifetime value.

    Parameters:
    -----------
    key : jax.random.PRNGKey
        Random key for JAX
    model : McCallModelContinuous
        The model containing parameters
    w_bar : float
        The reservation wage
    n_periods : int
        Number of periods to simulate

    Returns:
    --------
    lifetime_value : float
        Discounted sum of income over n_periods
    """
    c, β, σ, μ, w_draws = model

    # Draw all wage offers upfront
    key, subkey = jax.random.split(key)
    s_vals = jax.random.normal(subkey, (n_periods,))
    wage_offers = jnp.exp(μ + σ * s_vals)

    # Determine which offers are acceptable
    accept = wage_offers >= w_bar

    # Track employment status: employed from first acceptance onward
    employed = jnp.cumsum(accept) > 0

    # Get the accepted wage (first wage where accept is True)
    first_accept_idx = jnp.argmax(accept)
    accepted_wage = wage_offers[first_accept_idx]

    # Earnings at each period: accepted_wage if employed, c if unemployed
    earnings = jnp.where(employed, accepted_wage, c)

    # Compute discounted sum
    periods = jnp.arange(n_periods)
    discount_factors = β ** periods
    lifetime_value = jnp.sum(discount_factors * earnings)

    return lifetime_value


@jax.jit
def compute_mean_lifetime_value(model, w_bar, num_reps=10000, seed=1234):
    """
    Compute mean lifetime value across many simulations.

    """
    key = jax.random.PRNGKey(seed)
    keys = jax.random.split(key, num_reps)

    # Vectorize the simulation across all replications
    simulate_fn = jax.vmap(simulate_lifetime_value, in_axes=(0, None, None))
```

(continues on next page)

(continued from previous page)

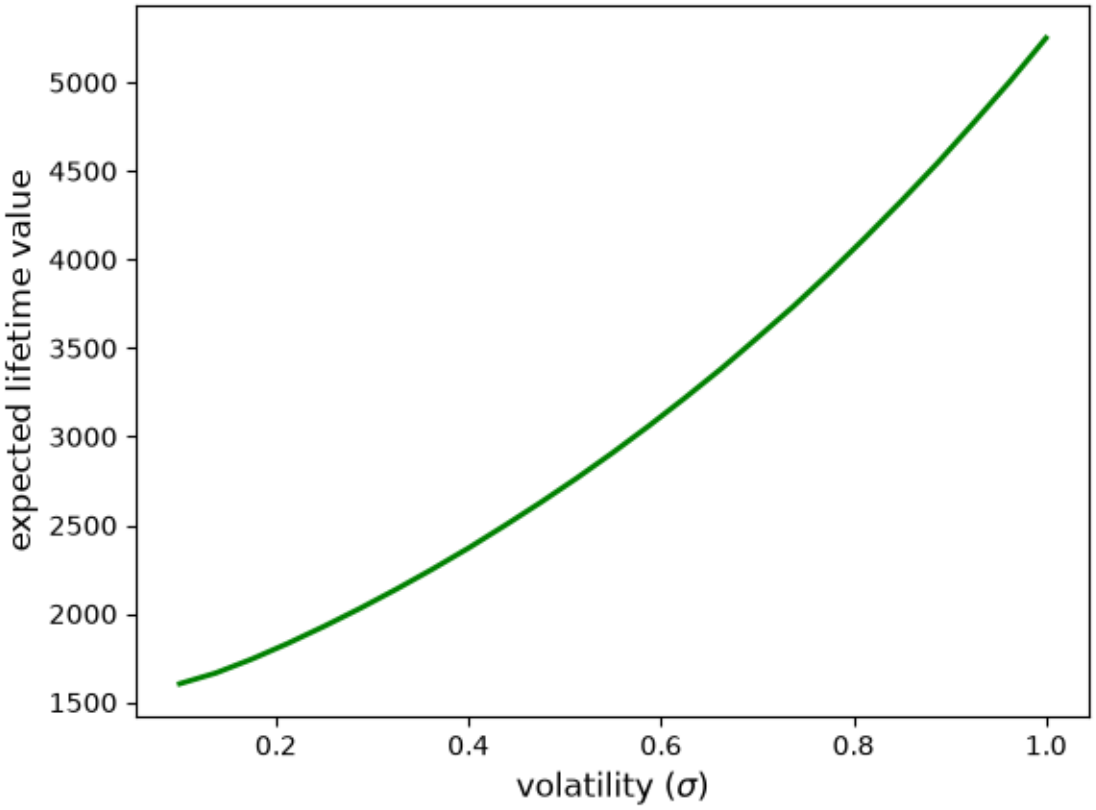
```
    lifetime_values = simulate_fn(keys, model, w_bar)
    return jnp.mean(lifetime_values)
```

Now let's compute the expected lifetime value for each volatility level:

```
# Use the same volatility range and mean wage
σ_vals = jnp.linspace(0.1, 1.0, 25)
mean_wage = 20.0

lifetime_vals = []
for σ in σ_vals:
    μ = compute_μ_for_mean(σ, mean_wage)
    model = create_mccall_continuous(σ=σ, μ=μ)
    w_bar = compute_reservation_wage_continuous(model)
    lv = compute_mean_lifetime_value(model, w_bar)
    lifetime_vals.append(lv)
```

Let's visualize the expected lifetime value as a function of volatility:

```
fig, ax = plt.subplots()
ax.plot(σ_vals, lifetime_vals, linewidth=2, color='green')
ax.set_xlabel(r'volatility ($\sigma$)', fontsize=12)
ax.set_ylabel('expected lifetime value', fontsize=12)
plt.show()
```



The plot confirms that despite workers setting higher reservation wages when facing more volatile wage offers (as shown above), they achieve higher expected lifetime values due to the option value of search.

50.7 Exercises

Exercise 50.7.1

Compute the average duration of unemployment when $\beta = 0.99$ and c takes the following values

```
c_vals = np.linspace(10, 40, 4)
```

That is, start the agent off as unemployed, compute their reservation wage given the parameters, and then simulate to see how long it takes to accept.

Repeat a large number of times and take the average.

Plot mean unemployment duration as a function of c in `c_vals`.

Try to explain what you see.

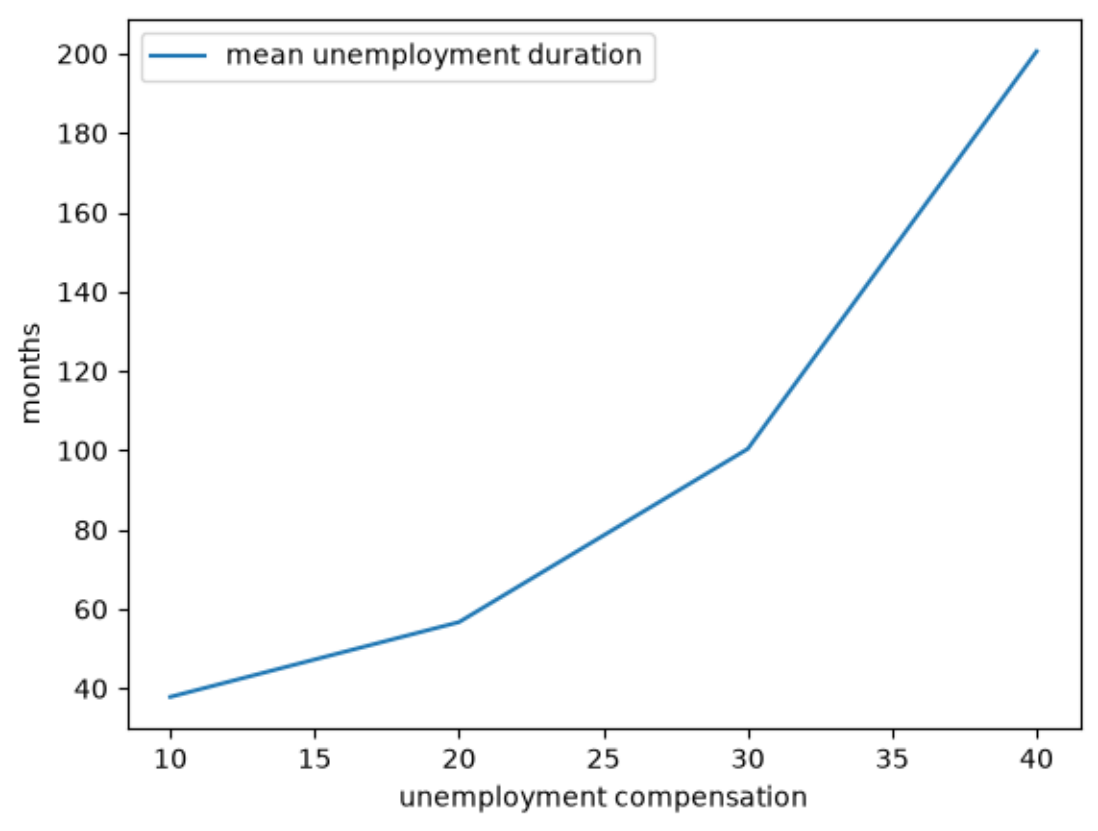
Solution

Here's a solution using the continuous wage offer distribution with JAX.

```
def compute_stopping_time_continuous(w_bar, key, model):
    """
    Compute stopping time by drawing wages from the continuous distribution
    until one exceeds `w_bar`.

    Parameters:
    -----------
    w_bar : float
        The reservation wage
    key : jax.random.PRNGKey
        Random key for JAX
    model : McCallModelContinuous
        The model containing wage draws

    Returns:
    --------
    t_final : int
        The stopping time (number of periods until acceptance)
    """
    c, β, σ, μ, w_draws = model

    def update(loop_state):
        t, key, accept = loop_state
        key, subkey = jax.random.split(key)
        # Draw a standard normal and transform to wage
        s = jax.random.normal(subkey)
        w = jnp.exp(μ + σ * s)
        accept = w >= w_bar
        t = t + 1
        return t, key, accept

    def cond(loop_state):
        _, _, accept = loop_state
        return jnp.logical_not(accept)

    initial_loop_state = (0, key, False)
```

```
    t_final, _, _ = jax.lax.while_loop(cond, update, initial_loop_state)
    return t_final


def compute_mean_stopping_time_continuous(w_bar, model, num_reps=100000,␣
↪seed=1234):
    """
    Generate a mean stopping time over `num_reps` repetitions.

    Parameters:
    -----------
    w_bar : float
        The reservation wage
    model : McCallModelContinuous
        The model containing parameters
    num_reps : int
        Number of simulation replications
    seed : int
        Random seed

    Returns:
    --------
    mean_time : float
        Average stopping time across all replications
    """
    # Generate a key for each MC replication
    key = jax.random.PRNGKey(seed)
    keys = jax.random.split(key, num_reps)

    # Vectorize compute_stopping_time_continuous and evaluate across keys
    compute_fn = jax.vmap(compute_stopping_time_continuous, in_axes=(None, 0,␣
↪None))
    obs = compute_fn(w_bar, keys, model)

    # Return mean stopping time
    return jnp.mean(obs)


# Compute mean stopping time for different values of c
c_vals = jnp.linspace(10, 40, 4)

@jax.jit
def compute_stop_time_for_c_continuous(c):
    """Compute mean stopping time for a given compensation value c."""
    model = create_mccall_continuous(c=c)
    w_bar = compute_reservation_wage_continuous(model)
    return compute_mean_stopping_time_continuous(w_bar, model)

# Vectorize across all c values
compute_stop_time_vectorized = jax.vmap(compute_stop_time_for_c_continuous)
stop_times = compute_stop_time_vectorized(c_vals)

fig, ax = plt.subplots()

ax.plot(c_vals, stop_times, label="mean unemployment duration")
ax.set(xlabel="unemployment compensation", ylabel="months")
ax.legend()
```

```
plt.show()
```


mean unemployment duration
200
180
160
140
120
100
80
60
40
months
10
15
20
25
30
35
40
unemployment compensation

CHAPTER

FIFTYONE

JOB SEARCH II: SEARCH AND SEPARATION

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

Contents

- *Job Search II: Search and Separation*
  - *Overview*
  - *The model*
  - *Solving the model*
  - *Code*
  - *A simplifying transformation*
  - *Implementation*
  - *Impact of parameters*
  - *Exercises*

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install quantecon jax myst-nb
```

51.1 Overview

Previously *we looked* at the McCall job search model [McCall, 1970] as a way of understanding unemployment and worker decisions.

One unrealistic feature of that version of the model was that every job is permanent.

In this lecture, we extend the model by introducing job separation.

Once separation enters the picture, the agent comes to view

- the loss of a job as a capital loss, and
- a spell of unemployment as an *investment* in searching for an acceptable job

The other minor addition is that a utility function will be included to make worker preferences slightly more sophisticated.

We'll need the following imports

```
import matplotlib.pyplot as plt
import numpy as np
import jax
import jax.numpy as jnp
from typing import NamedTuple
from quantecon.distributions import BetaBinomial
from myst_nb import glue
```

51.2 The model

The model is similar to the *baseline McCall job search model*.

It concerns the life of an infinitely lived worker and

- the opportunities he or she (let's say he to save one character) has to work at different wages
- exogenous events that destroy his current job
- his decision making process while unemployed

The worker can be in one of two states: employed or unemployed.

He wants to maximize

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t u(y_t) \tag{51.1}$$

At this stage the only difference from the *baseline model* is that we've added some flexibility to preferences by introducing a utility function u.

It satisfies $u' > 0$ and $u'' < 0$.

Wage offers $\{W_t\}$ are IID with common distribution q.

The set of possible wage values is denoted by $\mathbb{W}$.

51.2.1 Timing and decisions

At the start of each period, the agent can be either

- unemployed or
- employed at some existing wage level w.

If currently employed at wage w, the worker

1. receives utility $u(w)$ from their current wage and
2. is fired with some (small) probability α, becoming unemployed next period.

If currently unemployed, the worker receives random wage offer W_t and either accepts or rejects.

If he accepts, then he begins work immediately at wage W_t.

If he rejects, then he receives unemployment compensation c.

The process then repeats.

Note

We do not allow for job search while employed—this topic is taken up in a *later lecture*.

51.3 Solving the model

We drop time subscripts in what follows and primes denote next period values.

Let

- $v_e(w)$ be maximum lifetime value for a worker who enters the current period employed with wage w
- $v_u(w)$ be maximum lifetime for a worker who who enters the current period unemployed and receives wage offer w.

Here, **maximum lifetime value** means the value of (51.1) when the worker makes optimal decisions at all future points in time.

As we now show, obtaining these functions is key to solving the model.

51.3.1 The Bellman equations

We recall that, in *the original job search model*, the value function (the value of being unemployed with a given wage offer) satisfied a Bellman equation.

Here this function again satisfies a Bellman equation that looks very similar.

$$v_u(w) = \max \left\{ v_e(w),\ u(c) + \beta \sum_{w' \in \mathbb{W}} v_u(w') q(w') \right\} \tag{51.2}$$

The difference is that the value of accepting is $v_e(w)$ rather than $w/(1-\beta)$.

We have to make this change because jobs are not permanent.

Accepting transitions the worker to employment and hence yields reward $v_e(w)$, which we discuss below.

Rejecting leads to unemployment compensation and unemployment tomorrow.

Equation (51.2) expresses the value of being unemployed with offer w in hand as a maximum over the value of two options: accept or reject the current offer.

The function v_e also satisfies a Bellman equation:

$$v_e(w) = u(w) + \beta \left[(1-\alpha) v_e(w) + \alpha \sum_{w' \in \mathbb{W}} v_u(w') q(w') \right] \tag{51.3}$$

Note

This equation differs from a traditional Bellman equation because there is no max.

There is no max because an employed agent has no choices.

Nonetheless, in keeping with most of the literature, we also refer to it as a Bellman equation.

Equation (51.3) expresses the value of being employed at wage w in terms of

- current reward $u(w)$ plus
- discounted expected reward tomorrow, given the α probability of being fired

As we will see, equations (51.3) and (51.2) provide enough information to solve for both v_e and v_u.

Once we have them in hand, we will be able to make optimal choices.

51.3.2 The reservation wage

Let

$$h := u(c) + \beta \sum_{w' \in \mathbb{W}} v_u(w') q(w') \tag{51.4}$$

This is the **continuation value** for an unemployed agent – the value of rejecting the current offer and then making optimal choices.

From (51.2), we see that an unemployed agent accepts current offer w if $v_e(w) \geq h$.

This means precisely that the value of accepting is higher than the value of rejecting.

The function v_e is increasing in w, since an employed agent is never made worse off by a higher current wage.

Hence, we can express the optimal choice as accepting wage offer w if and only if $w \geq \bar{w}$, where the **reservation wage** $\bar{w}$ is the first wage level $w \in \mathbb{W}$ such that

$$v_e(w) \geq h$$

51.4 Code

Let's now implement a solution method based on the two Bellman equations (51.2) and (51.3).

51.4.1 Set up

The default utility function is a CRRA utility function

```
def u(x, γ):
    return (x**(1 - γ) - 1) / (1 - γ)
```

Also, here's a default wage distribution, based around the BetaBinomial distribution:

```
n = 60                                  # n possible outcomes for w
w_default = jnp.linspace(10, 20, n)     # wages between 10 and 20
a, b = 600, 400                         # shape parameters
dist = BetaBinomial(n-1, a, b)          # distribution
q_default = jnp.array(dist.pdf())       # probabilities as a JAX array
```

Here's our model class for the McCall model with separation.

```
class Model(NamedTuple):
    α: float = 0.2              # job separation rate
    β: float = 0.98             # discount factor
    γ: float = 2.0              # utility parameter (CRRA)
    c: float = 6.0              # unemployment compensation
    w: jnp.ndarray = w_default  # wage outcome space
    q: jnp.ndarray = q_default  # probabilities over wage offers
```

51.4.2 Operators

We'll use a similar iterative approach to solving the Bellman equations that we adopted in the *first job search lecture*.

As a first step, to iterate on the Bellman equations, we define two operators, one for each value function.

These operators take the current value functions as inputs and return updated versions.

```
def T_u(model, v_u, v_e):
    """
    Apply the unemployment Bellman update rule and return new guess of v_u.

    """
    α, β, γ, c, w, q = model
    h = u(c, γ) + β * (v_u @ q)
    v_u_new = jnp.maximum(v_e, h)
    return v_u_new
```

```
def T_e(model, v_u, v_e):
    """
    Apply the employment Bellman update rule and return new guess of v_e.

    """
    α, β, γ, c, w, q = model
    v_e_new = u(w, γ) + β * ((1 - α) * v_e + α * (v_u @ q))
    return v_e_new
```

51.4.3 Iteration

Now we write an iteration routine, which updates the pair of arrays v_u, v_e until convergence.

More precisely, we iterate until successive realizations are closer together than some small tolerance level.

```
def solve_full_model(
        model,
        tol: float = 1e-6,
        max_iter: int = 1_000,
    ):
    """
    Solves for both value functions v_u and v_e iteratively.

    """
    α, β, γ, c, w, q = model
    i = 0
    error = tol + 1
    v_e = v_u = w / (1 - β)

    while i < max_iter and error > tol:
        v_u_next = T_u(model, v_u, v_e)
        v_e_next = T_e(model, v_u, v_e)
        error_u = jnp.max(jnp.abs(v_u_next - v_u))
        error_e = jnp.max(jnp.abs(v_e_next - v_e))
        error = jnp.max(jnp.array([error_u, error_e]))
        v_u = v_u_next
        v_e = v_e_next
        i += 1

    return v_u, v_e
```

51.4.4 Computing the reservation wage

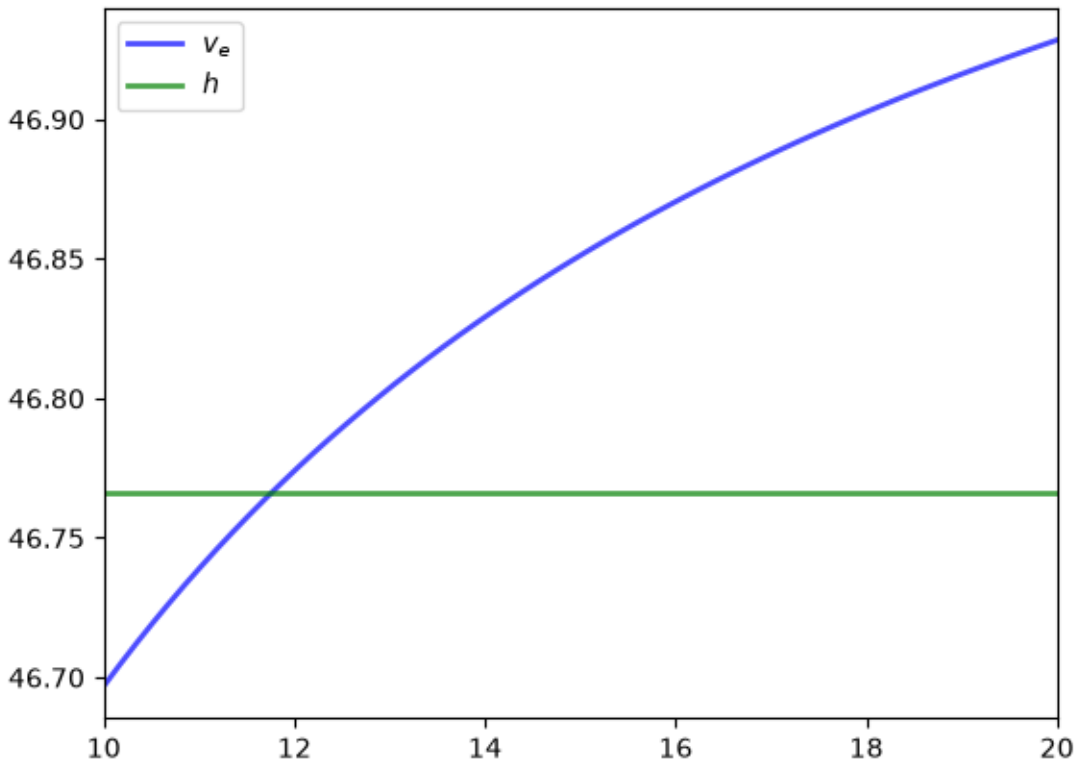
Now that we can solve for both value functions, let's investigate the reservation wage.

Recall from above that the reservation wage $\bar{w}$ is the first $w \in \mathbb{W}$ satisfying $v_e(w) \geq h$, where h is the continuation value defined in (51.4).

Let's compare v_e and h to see what they look like.

We'll use the default parameterizations found in the code above.

```
model = Model()
α, β, γ, c, w, q = model
v_u, v_e = solve_full_model(model)
h = u(c, γ) + β * (v_u @ q)

fig, ax = plt.subplots()
ax.plot(w, v_e, 'b-', lw=2, alpha=0.7, label='$v_e$')
ax.plot(w, [h] * len(w), 'g-', lw=2, alpha=0.7, label='$h$')
ax.set_xlim(min(w), max(w))
ax.legend()
plt.show()
```



The value v_e is increasing because higher w generates a higher wage flow conditional on staying employed.

The reservation wage is the w where these lines meet.

Let's compute this reservation wage explicitly:

```
def compute_reservation_wage_full(model):
    """
    Computes the reservation wage using the full model solution.
    """
    α, β, γ, c, w, q = model
    v_u, v_e = solve_full_model(model)
    h = u(c, γ) + β * (v_u @ q)
    # Find the first w such that v_e(w) >= h, or +inf if none exist
    accept = v_e >= h
    i = jnp.argmax(accept)  # returns first accept index
    w_bar = jnp.where(jnp.any(accept), w[i], jnp.inf)
    return w_bar

w_bar_full = compute_reservation_wage_full(model)
print(f"Reservation wage (full model): {w_bar_full:.4f}")
```

```
Reservation wage (full model): 11.8644
```

This value seems close to where the two lines meet.

51.5 A simplifying transformation

The approach above works, but iterating over two vector-valued functions is computationally expensive.

With some mathematics and some brain power, we can form a solution method that is far more efficient.

(This process will be analogous to our *second pass* at the plain vanilla McCall model, where we reduced the Bellman equation to an equation in an unknown scalar value, rather than an unknown vector.)

First, we use the continuation value h, as defined in (51.4), to write (51.2) as

$$v_u(w) = \max\{v_e(w),\, h\}$$

Taking the expectation of both sides and then discounting, this becomes

$$\beta \sum_{w'} v_u(w') q(w') = \beta \sum_{w'} \max\{v_e(w'),\, h\}\, q(w')$$

Adding $u(c)$ to both sides and using (51.4) again gives

$$h = u(c) + \beta \sum_{w'} \max\{v_e(w'),\, h\}\, q(w') \tag{51.5}$$

This is a nice scalar equation in the continuation value, which is already useful.

But we can go further, but eliminating v_e from the above equation.

51.5.1 Simplifying to a single equation

As a first step, we rearrange the expression defining h (see (51.4)) to obtain

$$\sum_{w'} v_u(w') q(w') = \frac{h - u(c)}{\beta}$$

Using this, the Bellman equation for v_e, as given in (51.3), can now be rewritten as

$$v_e(w) = u(w) + \beta \left[(1-\alpha) v_e(w) + \alpha \frac{h - u(c)}{\beta} \right] \tag{51.6}$$

Our next step is to solve (51.6) for v_e as a function of h.

Rearranging (51.6) gives

$$v_e(w) = u(w) + \beta(1-\alpha) v_e(w) + \alpha(h - u(c))$$

or

$$v_e(w) - \beta(1-\alpha) v_e(w) = u(w) + \alpha(h - u(c))$$

Solving for $v_e(w)$ gives

$$v_e(w) = \frac{u(w) + \alpha(h - u(c))}{1 - \beta(1-\alpha)} \tag{51.7}$$

Substituting this into (51.5) yields

$$h = u(c) + \beta \sum_{w' \in \mathbb{W}} \max\left\{ \frac{u(w') + \alpha(h - u(c))}{1 - \beta(1-\alpha)},\, h \right\} q(w') \tag{51.8}$$

Finally we have a single scalar equation in h!

If we can solve this for h, we can easily recover v_e using (51.7).

Then we have enough information to compute the reservation wage.

51.5.2 Solving the Bellman equations

To solve (51.8), we use the iteration rule

$$h_{n+1} = u(c) + \beta \sum_{w' \in \mathbb{W}} \max \left\{ \frac{u(w') + \alpha(h_n - u(c))}{1 - \beta(1-\alpha)}, \, h_n \right\} q(w') \tag{51.9}$$

starting from some initial condition h_0.

(It is possible to prove that (51.9) converges via the Banach contraction mapping theorem.)

51.6 Implementation

To implement iteration on h, we provide a function that provides one update, from h_n to h_{n+1}

```
def update_h(model, h):
    " One update of the scalar h. "
    α, β, γ, c, w, q = model
    v_e = compute_v_e(model, h)
    h_new = u(c, γ) + β * (jnp.maximum(v_e, h) @ q)
    return h_new
```

Also, we provide a function to compute v_e from (51.7).

```
def compute_v_e(model, h):
    " Compute v_e from h using the closed-form expression. "
    α, β, γ, c, w, q = model
    return (u(w, γ) + α * (h - u(c, γ))) / (1 - β * (1 - α))
```

This function will be applied once convergence is achieved.

Now we can write our model solver.

```
@jax.jit
def solve_model(model, tol=1e-5, max_iter=2000):
    " Iterates to convergence on the Bellman equations. "

    def cond(loop_state):
        h, i, error = loop_state
        return jnp.logical_and(error > tol, i < max_iter)

    def update(loop_state):
        h, i, error = loop_state
        h_new = update_h(model, h)
        error_new = jnp.abs(h_new - h)
        return h_new, i + 1, error_new

    # Initialize
    h_init = u(model.c, model.γ) / (1 - model.β)
    i_init = 0
    error_init = tol + 1
    init_state = (h_init, i_init, error_init)

    final_state = jax.lax.while_loop(cond, update, init_state)
    h_final, _, _ = final_state
```

(continues on next page)

(continued from previous page)

```
    # Compute v_e from the converged h
    v_e_final = compute_v_e(model, h_final)

    return v_e_final, h_final
```

Finally, here's a function `compute_reservation_wage` that uses all the logic above, taking an instance of `Model` and returning the associated reservation wage.

```
def compute_reservation_wage(model):
    """
    Computes the reservation wage of an instance of the McCall model
    by finding the smallest w such that v_e(w) >= h.

    """
    # Find the first i such that v_e(w_i) >= h and return w[i]
    # If no such w exists, then w_bar is set to np.inf
    v_e, h = solve_model(model)
    accept = v_e >= h
    i = jnp.argmax(accept)    # take first accept index
    w_bar = jnp.where(jnp.any(accept), model.w[i], jnp.inf)
    return w_bar
```

Let's verify that this simplified approach gives the same answer as the full model:

```
w_bar_simplified = compute_reservation_wage(model)
print(f"Reservation wage (simplified): {w_bar_simplified:.4f}")
print(f"Reservation wage (full model): {w_bar_full:.4f}")
print(f"Difference: {abs(w_bar_simplified - w_bar_full):.6f}")
```

```
Reservation wage (simplified): 11.8644
Reservation wage (full model): 11.8644
Difference: 0.000000
```

As we can see, both methods produce essentially the same reservation wage.

However, the simplified method is far more efficient.

Next we will investigate how the reservation wage varies with parameters.

51.7 Impact of parameters

In each instance below, we'll show you a figure and then ask you to reproduce it in the exercises.

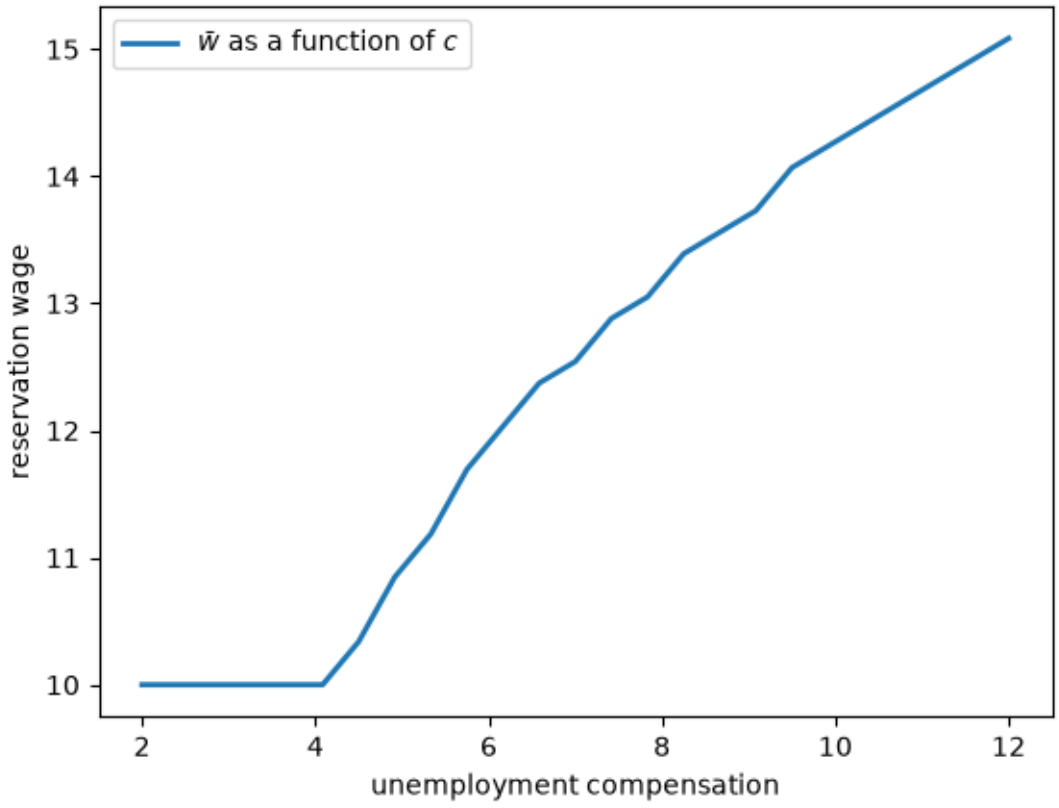
51.7.1 The reservation wage and unemployment compensation

First, let's look at how $\bar{w}$ varies with unemployment compensation.

In the figure below, we use the default parameters in the `Model` class, apart from c (which takes the values given on the horizontal axis)

As expected, higher unemployment compensation causes the worker to hold out for higher wages.

In effect, the cost of continuing job search is reduced.



51.7.2 The reservation wage and discounting

Next, let's investigate how $\bar{w}$ varies with the discount factor.

The next figure plots the reservation wage associated with different values of β

Again, the results are intuitive: More patient workers will hold out for higher wages.

51.7.3 The reservation wage and job destruction

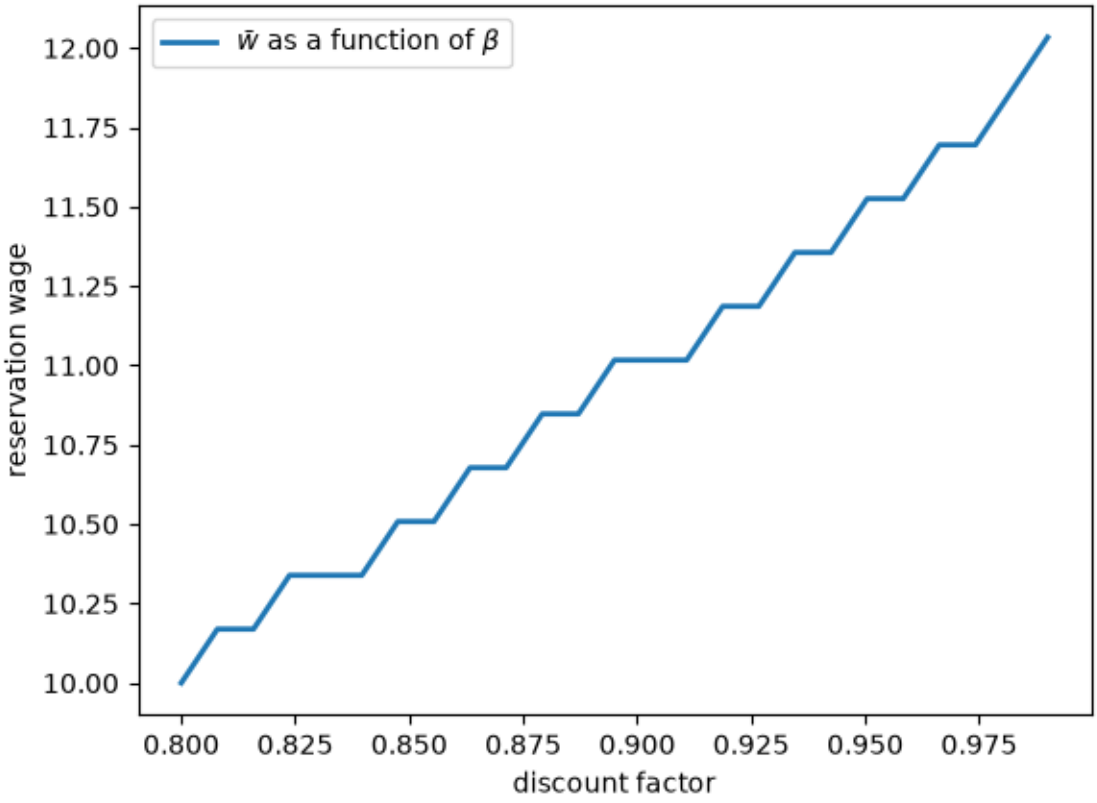
Finally, let's look at how $\bar{w}$ varies with the job separation rate α.

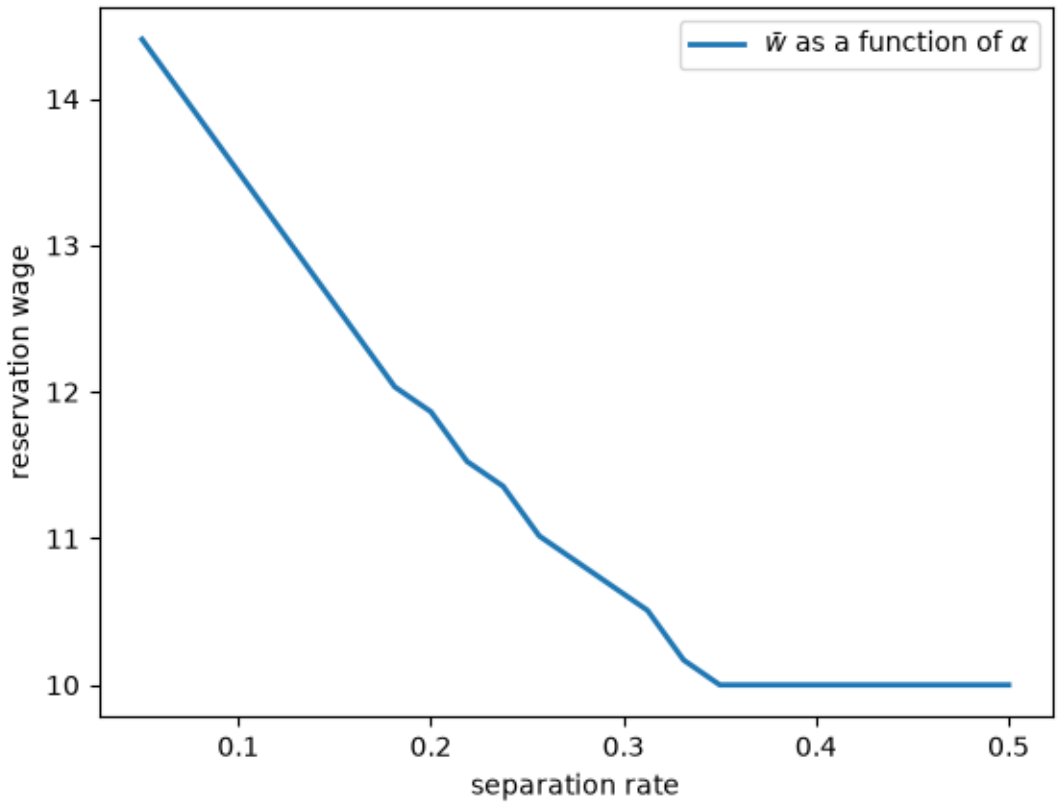
Higher α translates to a greater chance that a worker will face termination in each period once employed.

Once more, the results are in line with our intuition.

If the separation rate is high, then the benefit of holding out for a higher wage falls.

Hence the reservation wage is lower.


$\bar{w}$ as a function of β
12.00
11.75
11.50
11.25
11.00
10.75
10.50
10.25
10.00
reservation wage
0.800
0.825
0.850
0.875
0.900
0.925
0.950
0.975
discount factor


$\bar{w}$ as a function of α
14
13
12
11
10
reservation wage
0.1
0.2
0.3
0.4
0.5
separation rate

51.8 Exercises

Exercise 51.8.1

Reproduce all the reservation wage figures shown above.

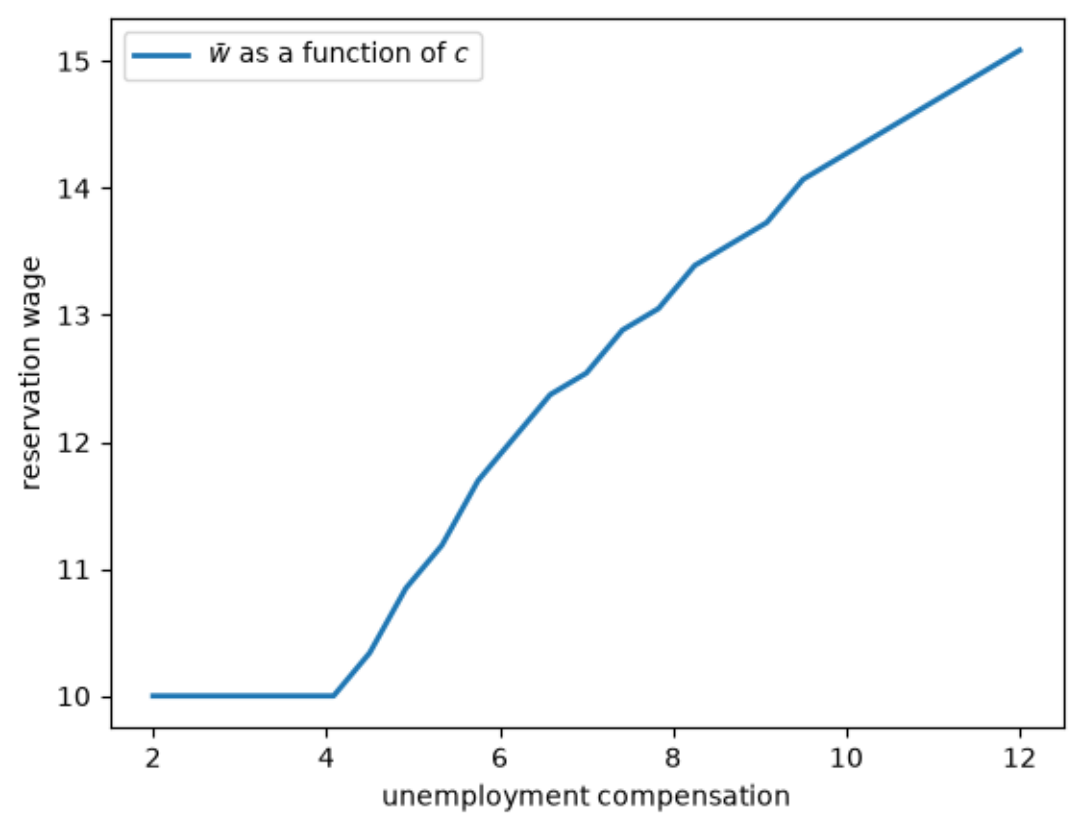
Regarding the values on the horizontal axis, use

```
grid_size = 25
c_vals = jnp.linspace(2, 12, grid_size)          # unemployment compensation
β_vals = jnp.linspace(0.8, 0.99, grid_size)      # discount factors
α_vals = jnp.linspace(0.05, 0.5, grid_size)      # separation rate
```

Solution

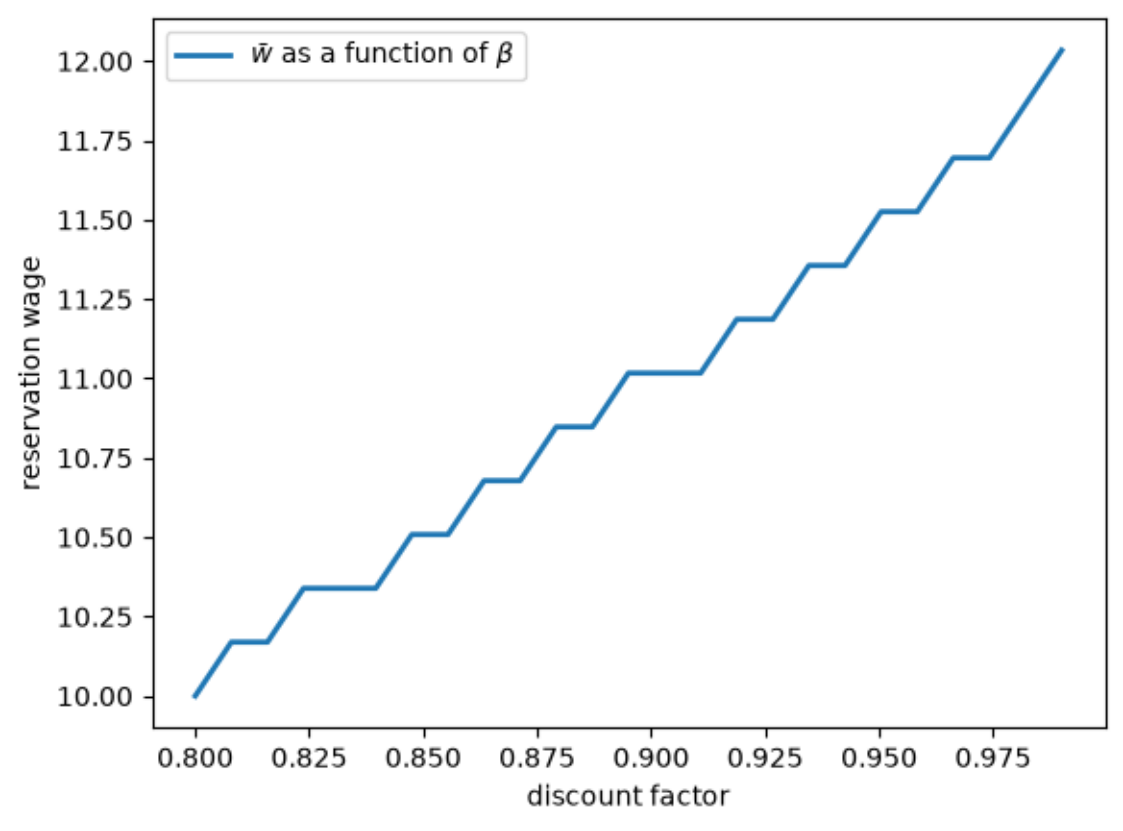
Here's the first figure.

```
def compute_res_wage_given_c(c):
    model = Model(c=c)
    w_bar = compute_reservation_wage(model)
    return w_bar

w_bar_vals = jax.vmap(compute_res_wage_given_c)(c_vals)

fig, ax = plt.subplots()
ax.set(xlabel='unemployment compensation', ylabel='reservation wage')
ax.plot(c_vals, w_bar_vals, lw=2, label=r'$\bar w$ as a function of $c$')
ax.legend()
glue("mccall_resw_c", fig, display=False)
plt.show()
```



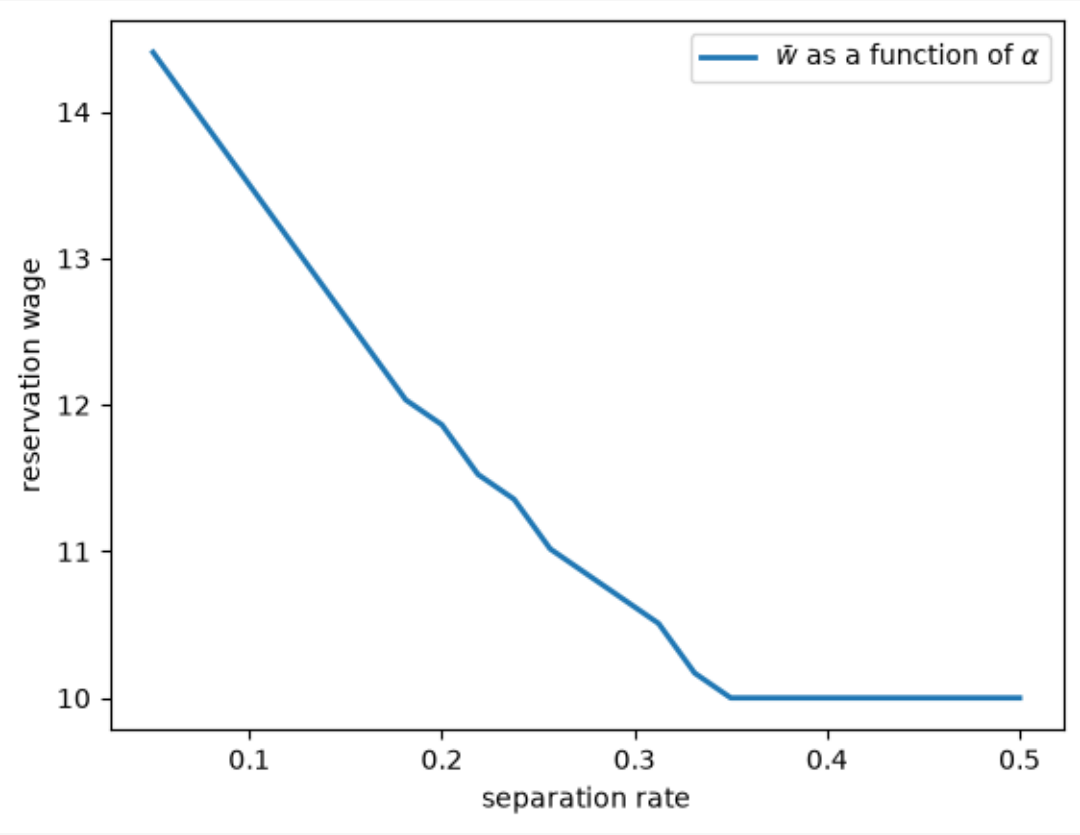
Here's the second one.

```
def compute_res_wage_given_beta(β):
    model = Model(β=β)
    w_bar = compute_reservation_wage(model)
    return w_bar

w_bar_vals = jax.vmap(compute_res_wage_given_beta)(β_vals)

fig, ax = plt.subplots()
ax.set(xlabel='discount factor', ylabel='reservation wage')
ax.plot(β_vals, w_bar_vals, lw=2, label=r'$\bar w$ as a function of $\beta$')
ax.legend()
glue("mccall_resw_beta", fig, display=False)
plt.show()
```



Here's the third.

```
def compute_res_wage_given_alpha(α):
    model = Model(α=α)
    w_bar = compute_reservation_wage(model)
    return w_bar

w_bar_vals = jax.vmap(compute_res_wage_given_alpha)(α_vals)

fig, ax = plt.subplots()
ax.set(xlabel='separation rate', ylabel='reservation wage')
ax.plot(α_vals, w_bar_vals, lw=2, label=r'$\bar w$ as a function of $\alpha$')
ax.legend()
glue("mccall_resw_alpha", fig, display=False)
plt.show()
```


$\bar{w}$ as a function of α
reservation wage
separation rate
14
13
12
11
10
0.1
0.2
0.3
0.4
0.5

CHAPTER

FIFTYTWO

JOB SEARCH III: SEARCH WITH SEPARATION AND MARKOV WAGES

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

Contents

- *Job Search III: Search with Separation and Markov Wages*
  - *Model setup*
  - *Code*
  - *Improving efficiency*
  - *Sensitivity analysis*
  - *Employment simulation*
  - *Ergodic property*
  - *Cross-sectional analysis*
  - *Lower unemployment compensation (c=0.5)*
  - *Exercises*

This lecture builds on the job search model with separation presented in the *previous lecture*.

The key difference is that wage offers now follow a *Markov chain* rather than being IID.

This modification adds persistence to the wage offer process, meaning that today's wage offer provides information about tomorrow's offer.

This feature makes the model more realistic, as labor market conditions tend to exhibit serial correlation over time.

In addition to what's in Anaconda, this lecture will need the following libraries

```
!pip install quantecon jax
```

We use the following imports:

```
from quantecon.markov import tauchen
import jax.numpy as jnp
import jax
from jax import lax
from typing import NamedTuple
import matplotlib.pyplot as plt
from functools import partial
```

52.1 Model setup

The setting is as follows:

- Each unemployed agent receives a wage offer w from a finite set $\mathbb{W}$
- Wage offers follow a Markov chain with transition matrix P
- Jobs terminate with probability α each period (separation rate)
- Unemployed workers receive compensation c per period
- Future payoffs are discounted by factor $\beta \in (0, 1)$

52.1.1 Decision problem

When unemployed and receiving wage offer w, the agent chooses between:

1. Accept offer w: Become employed at wage w
2. Reject offer: Remain unemployed, receive c, get new offer next period

The wage updates are as follows:

- If an unemployed agent rejects offer w, then their next offer is drawn from $P(w, \cdot)$
- If an employed agent loses a job in which they were paid wage w, then their next offer is drawn from $P(w, \cdot)$

52.1.2 The wage offer process

To construct the wage offer process we start with an AR1 process.

$$X_{t+1} = \rho X_t + \nu Z_{t+1}$$

where $\{Z_t\}$ is IID and standard normal.

Below we will always choose $\rho \in (0, 1)$.

This means that the wage process will be positively correlated: the higher the current wage offer, the more likely we are to get a high offer tomorrow.

To go from the AR1 process to the wage offer process, we set $W_t = \exp(X_t)$.

Actually, in practice, we approximate this wage process as follows:

- discretize the AR1 process using *Tauchen's method* and

- take the exponential of the resulting wage offer values.

52.1.3 Value functions

We let

- $v_u(w)$ be the value of being unemployed when current wage offer is w
- $v_e(w)$ be the value of being employed at wage w

The Bellman equations are obvious modifications of the *IID case*.

The only change is that expectations for next period are computed using the transition matrix P conditioned on current wage w, instead of being drawn independently from q.

The unemployed worker's value function satisfies the Bellman equation

$$v_u(w) = \max \left\{ v_e(w), u(c) + \beta \sum_{w'} v_u(w') P(w, w') \right\}$$

The employed worker's value function satisfies the Bellman equation

$$v_e(w) = u(w) + \beta \left[\alpha \sum_{w'} v_u(w') P(w, w') + (1 - \alpha) v_e(w) \right]$$

As a matter of notation, given a function h assigning values to wages, it is common to set

$$(Ph)(w) = \sum_{w'} h(w') P(w, w')$$

(To understand this expression, think of P as a matrix, h as a column vector, and w as a row index.)

With this notation, the Bellman equations become

$$v_u(w) = \max\{v_e(w), u(c) + \beta (P v_u)(w)\}$$

and

$$v_e(w) = u(w) + \beta \left[\alpha (P v_u)(w) + (1 - \alpha) v_e(w)\right]$$

52.1.4 Optimal policy

Once we have the solutions v_e and v_u to these Bellman equations, we can compute the optimal policy: accept at current wage offer w if

$$v_e(w) \geq u(c) + \beta (P v_u)(w)$$

The optimal policy turns out to be a reservation wage strategy: accept all wages above some threshold.

52.2 Code

Let's now implement the model.

52.2.1 Set up

The default utility function is a CRRA utility function

```
def u(x, γ):
    return (x**(1 - γ) - 1) / (1 - γ)
```

Let's set up a `Model` class to store information needed to solve the model.

We include `P_cumsum`, the row-wise cumulative sum of the transition matrix, to optimize simulation – the details are explained below.

```
class Model(NamedTuple):
    n: int
    w_vals: jnp.ndarray
    P: jnp.ndarray
    P_cumsum: jnp.ndarray
    β: float
    c: float
    α: float
    γ: float
```

The next function holds default values and creates a `Model` instance:

```
def create_js_with_sep_model(
        n: int = 200,           # wage grid size
        ρ: float = 0.9,         # wage persistence
        ν: float = 0.2,         # wage volatility
        β: float = 0.96,        # discount factor
        α: float = 0.05,        # separation rate
        c: float = 1.0,         # unemployment compensation
        γ: float = 1.5          # utility parameter
    ) -> Model:
    """
    Creates an instance of the job search model with separation.

    """
    mc = tauchen(n, ρ, ν)
    w_vals, P = jnp.exp(jnp.array(mc.state_values)), jnp.array(mc.P)
    P_cumsum = jnp.cumsum(P, axis=1)
    return Model(n, w_vals, P, P_cumsum, β, c, α, γ)
```

52.2.2 Solution: first pass

Let's put together a (not very efficient) routine for calculating the reservation wage.

(We will think carefully about efficiency below.)

It works by starting with guesses for v_e and v_u and iterating to convergence.

Here are Bellman operators that update v_u and v_e respectively.

```
def T_u(model, v_u, v_e):
    """
    Apply the unemployment Bellman update rule and return new guess of v_u.

    """
```

(continues on next page)

(continued from previous page)

```
    n, w_vals, P, P_cumsum, β, c, α, γ = model
    h = u(c, γ) + β * P @ v_u
    v_u_new = jnp.maximum(v_e, h)
    return v_u_new
```

```
def T_e(model, v_u, v_e):
    """
    Apply the employment Bellman update rule and return new guess of v_e.

    """
    n, w_vals, P, P_cumsum, β, c, α, γ = model
    v_e_new = u(w_vals, γ) + β * ((1 - α) * v_e + α * P @ v_u)
    return v_e_new
```

Here's a routine to iterate to convergence and then compute the reservation wage.

```
def solve_model_first_pass(
        model: Model,           # instance containing default parameters
        v_u_init: jnp.ndarray,  # initial condition for v_u
        v_e_init: jnp.ndarray,  # initial condition for v_e
        tol: float=1e-6,        # error tolerance
        max_iter: int=1_000,    # maximum number of iterations for loop
    ):
    n, w_vals, P, P_cumsum, β, c, α, γ = model
    i = 0
    error = tol + 1
    v_u = v_u_init
    v_e = v_e_init

    while i < max_iter and error > tol:
        v_u_next = T_u(model, v_u, v_e)
        v_e_next = T_e(model, v_u, v_e)
        error_u = jnp.max(jnp.abs(v_u_next - v_u))
        error_e = jnp.max(jnp.abs(v_e_next - v_e))
        error = jnp.maximum(error_u, error_e)
        v_u = v_u_next
        v_e = v_e_next
        i += 1

    # Compute accept and reject values
    continuation_values = u(c, γ) + β * P @ v_u

    # Find where acceptance becomes optimal
    accept_indices = v_e >= continuation_values
    first_accept_idx = jnp.argmax(accept_indices)  # index of first True

    # If no acceptance (all False), return infinity
    # Otherwise return the wage at the first acceptance index
    w_bar = jnp.where(
        jnp.any(accept_indices), w_vals[first_accept_idx], jnp.inf
    )
    return v_u, v_e, w_bar
```

52.2.3 Road test

Let's solve the model:

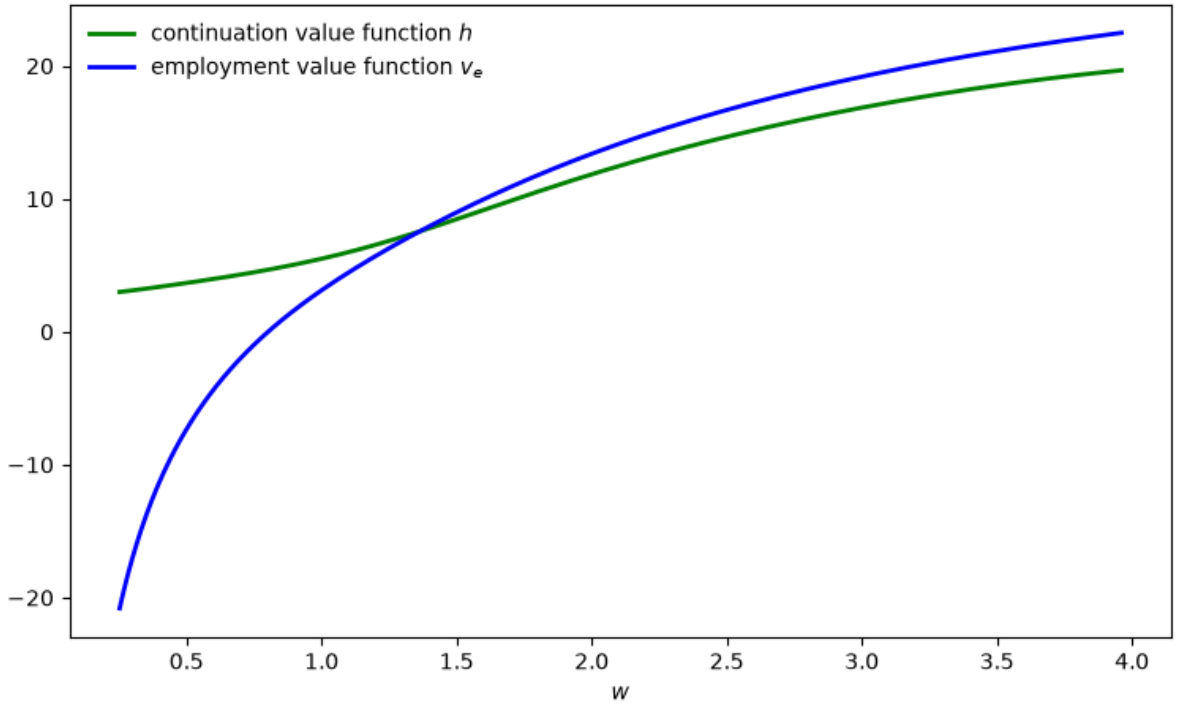
```
model = create_js_with_sep_model()
n, w_vals, P, P_cumsum, β, c, α, γ = model
v_u_init = jnp.zeros(n)
v_e_init = jnp.zeros(n)
v_u, v_e, w_bar_first = solve_model_first_pass(model, v_u_init, v_e_init)
```

Next we compute the continuation values.

```
h = u(c, γ) + β * P @ v_u
```

Let's plot our results.

```
fig, ax = plt.subplots(figsize=(9, 5.2))
ax.plot(w_vals, h, 'g-', linewidth=2,
        label="continuation value function $h$")
ax.plot(w_vals, v_e, 'b-', linewidth=2,
        label="employment value function $v_e$")
ax.legend(frameon=False)
ax.set_xlabel(r"$w$")
plt.show()
```



The reservation wage is at the intersection of v_e, and the continuation value function, which is the value of rejecting.

52.3 Improving efficiency

The solution method described above works fine but we can do much better.

First, we use the employed worker's Bellman equation to express v_e in terms of Pv_u

$$v_e(w) = \frac{1}{1-\beta(1-\alpha)} \cdot (u(w) + \alpha\beta(Pv_u)(w))$$

Next we substitute into the unemployed agent's Bellman equation to get

$$v_u(w) = \max\left\{\frac{1}{1-\beta(1-\alpha)} \cdot (u(w) + \alpha\beta(Pv_u)(w)), u(c) + \beta(Pv_u)(w)\right\}$$

Then we use value function iteration to solve for v_u.

With v_u in hand, we can recover v_e through the equations above and then compute the reservation wage.

Here's the new Bellman operator for the unemployed worker's value function:

```
def T(v: jnp.ndarray, model: Model) -> jnp.ndarray:
    """
    The Bellman operator for v_u.

    """
    n, w_vals, P, P_cumsum, β, c, α, γ = model
    d = 1 / (1 - β * (1 - α))
    v_e = d * (u(w_vals, γ) + α * β * P @ v)
    h = u(c, γ) + β * P @ v
    return jnp.maximum(v_e, h)
```

Here's a routine for value function iteration.

```
@jax.jit
def vfi(
        model: Model,
        tolerance: float = 1e-6,    # Error tolerance
        max_iter: int = 100_000,    # Max iteration bound
    ):

    v_init = jnp.zeros(model.w_vals.shape)

    def cond(loop_state):
        v, error, i = loop_state
        return (error > tolerance) & (i <= max_iter)

    def update(loop_state):
        v, error, i = loop_state
        v_new = T(v, model)
        error = jnp.max(jnp.abs(v_new - v))
        new_loop_state = v_new, error, i + 1
        return new_loop_state

    initial_state = (v_init, tolerance + 1, 1)
    final_loop_state = lax.while_loop(cond, update, initial_state)
    v_final, error, i = final_loop_state

    return v_final
```

Here is a routine that computes the reservation wage from the value function.

```
@jax.jit
def get_reservation_wage(v: jnp.ndarray, model: Model) -> float:
    """
    Calculate the reservation wage from the unemployed agents
    value function v := v_u.

    The reservation wage is the lowest wage w where accepting (v_e(w))
    is at least as good as rejecting (u(c) + β(Pv_u)(w)).

    """
    n, w_vals, P, P_cumsum, β, c, α, γ = model

    # Compute accept and reject values
    d = 1 / (1 - β * (1 - α))
    v_e = d * (u(w_vals, γ) + α * β * P @ v)
    continuation_values = u(c, γ) + β * P @ v

    # Find where acceptance becomes optimal
    accept_indices = v_e >= continuation_values
    first_accept_idx = jnp.argmax(accept_indices)  # index of first True

    # If no acceptance (all False), return infinity
    # Otherwise return the wage at the first acceptance index
    return jnp.where(jnp.any(accept_indices), w_vals[first_accept_idx], jnp.inf)
```

Let's solve the model using our new method:

```
model = create_js_with_sep_model()
n, w_vals, P, P_cumsum, β, c, α, γ = model
v_u = vfi(model)
w_bar = get_reservation_wage(v_u, model)
```

Let's verify that both methods produce the same reservation wage:

```
print(f"Reservation wage (first method):  {w_bar_first:.6f}")
print(f"Reservation wage (second method): {w_bar:.6f}")
print(f"Difference: {abs(w_bar - w_bar_first):.2e}")
```

```
Reservation wage (first method):  1.365155
Reservation wage (second method): 1.365155
Difference: 0.00e+00
```

Next we compute some related quantities for plotting.

```
d = 1 / (1 - β * (1 - α))
v_e = d * (u(w_vals, γ) + α * β * P @ v_u)
h = u(c, γ) + β * P @ v_u
```

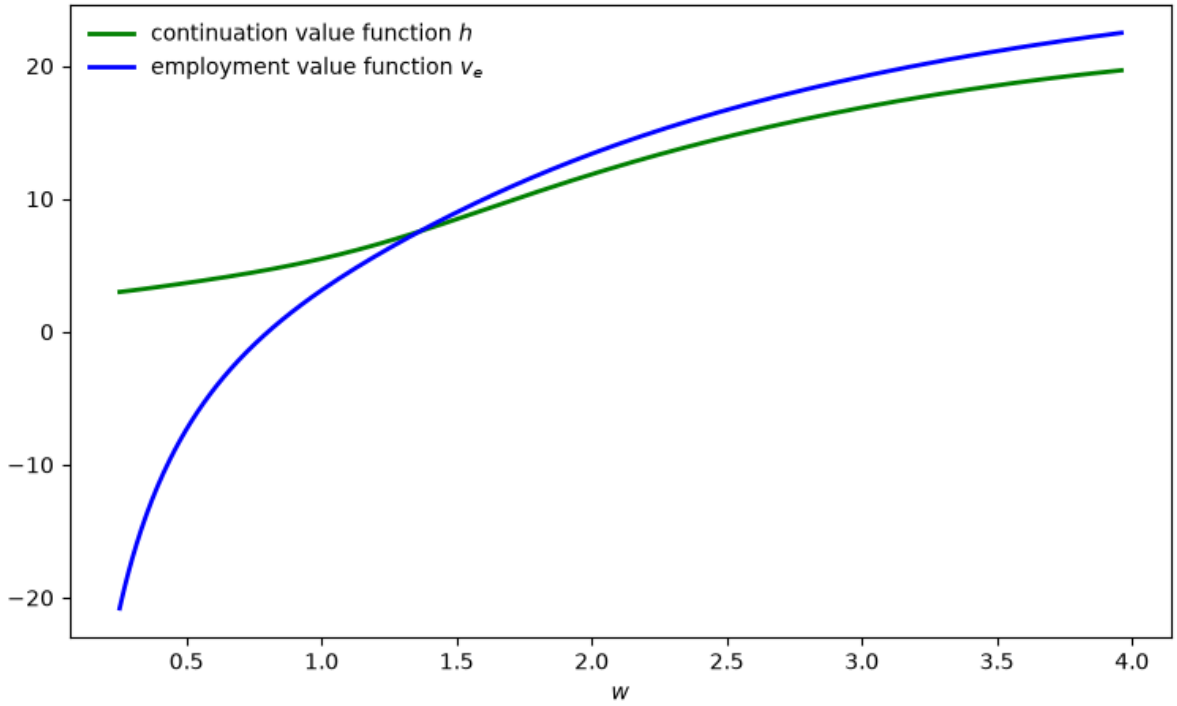
Let's plot our results.

```
fig, ax = plt.subplots(figsize=(9, 5.2))
ax.plot(w_vals, h, 'g-', linewidth=2,
        label="continuation value function $h$")
ax.plot(w_vals, v_e, 'b-', linewidth=2,
        label="employment value function $v_e$")
ax.legend(frameon=False)
```

(continues on next page)

(continued from previous page)

```
ax.set_xlabel(r"$w$")
plt.show()
```

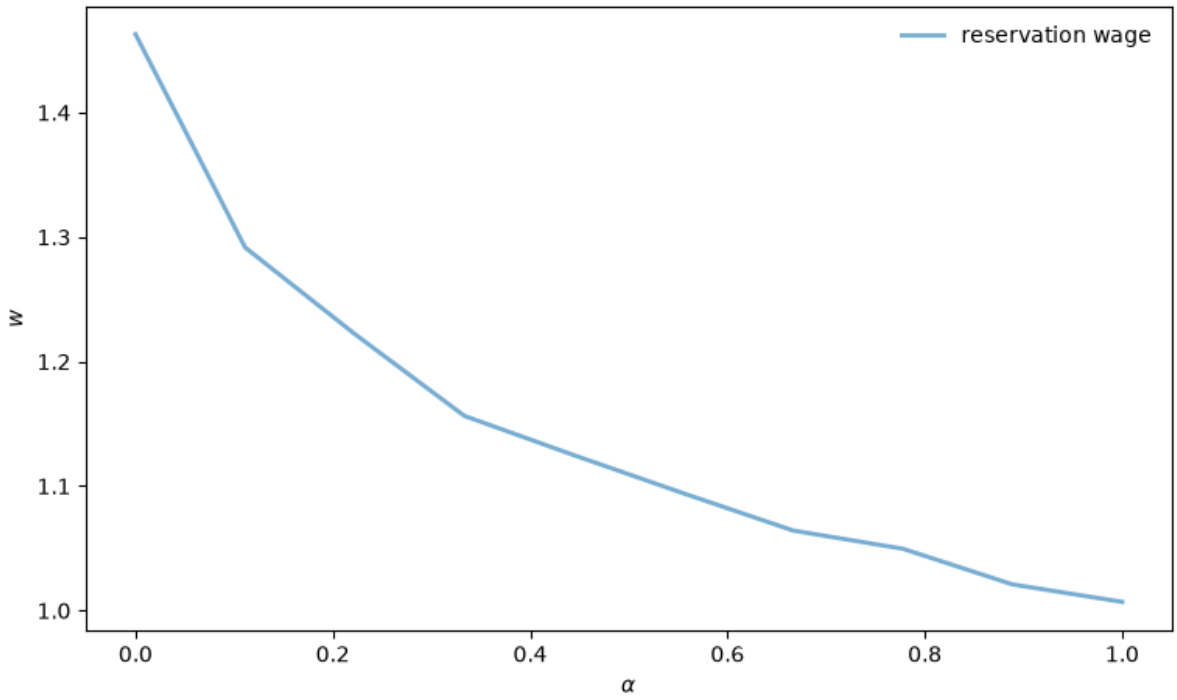


The result is the same as before but we only iterate on one array — and also our JAX code is more efficient.

52.4 Sensitivity analysis

Let's examine how reservation wages change with the separation rate.

```
α_vals: jnp.ndarray = jnp.linspace(0.0, 1.0, 10)

w_bar_vec = []
for α in α_vals:
    model = create_js_with_sep_model(α=α)
    v_u = vfi(model)
    w_bar = get_reservation_wage(v_u, model)
    w_bar_vec.append(w_bar)

fig, ax = plt.subplots(figsize=(9, 5.2))
ax.plot(
    α_vals, w_bar_vec, linewidth=2, alpha=0.6, label="reservation wage"
)
ax.legend(frameon=False)
ax.set_xlabel(r"$\alpha$")
ax.set_ylabel(r"$w$")
plt.show()
```



Can you provide an intuitive economic story behind the outcome that you see in this figure?

52.5 Employment simulation

Now let's simulate the employment dynamics of a single agent under the optimal policy.

Note that, when simulating the Markov chain for wage offers, we need to draw from the distribution in each row of P many times.

To do this, we use the inverse transform method: draw a uniform random variable and find where it falls in the cumulative distribution.

This is implemented via `jnp.searchsorted` on the precomputed cumulative sum `P_cumsum`, which is much faster than recomputing the cumulative sum each time.

The function `update_agent` advances the agent's state by one period.

The agent's state is a pair (S_t, W_t), where S_t is employment status (0 if unemployed, 1 if employed) and W_t is

- their current wage offer, if unemployed, or
- their current wage, if employed.

```
def update_agent(key, status, wage_idx, model, w_bar):
    """
    Updates an agent's employment status and current wage.

    Parameters:
    - key: JAX random key
    - status: Current employment status (0 or 1)
    - wage_idx: Current wage, recorded as an array index
    - model: Model instance
    - w_bar: Reservation wage
```

(continues on next page)

(continued from previous page)

```
    """
    n, w_vals, P, P_cumsum, β, c, α, γ = model

    key1, key2 = jax.random.split(key)
    # Use precomputed cumulative sum for efficient sampling
    # via the inverse transform method.
    new_wage_idx = jnp.searchsorted(
        P_cumsum[wage_idx, :], jax.random.uniform(key1)
    )
    separation_occurs = jax.random.uniform(key2) < α
    # Accept if current wage meets or exceeds reservation wage
    accepts = w_vals[wage_idx] >= w_bar

    # If employed: status = 1 if no separation, 0 if separation
    # If unemployed: status = 1 if accepts, 0 if rejects
    next_status = jnp.where(
        status,
        1 - separation_occurs.astype(jnp.int32),  # employed path
        accepts.astype(jnp.int32)                 # unemployed path
    )

    # If employed: wage = current if no separation, new if separation
    # If unemployed: wage = current if accepts, new if rejects
    next_wage = jnp.where(
        status,
        jnp.where(separation_occurs, new_wage_idx, wage_idx),  # employed path
        jnp.where(accepts, wage_idx, new_wage_idx)             # unemployed path
    )

    return next_status, next_wage
```

Here's a function to simulate the employment path of a single agent.

```
def simulate_employment_path(
        model: Model,      # Model details
        w_bar: float,     # Reservation wage
        T: int = 2_000,    # Simulation length
        seed: int = 42     # Set seed for simulation
    ):
    """
    Simulate employment path for T periods starting from unemployment.

    """
    key = jax.random.PRNGKey(seed)
    # Unpack model
    n, w_vals, P, P_cumsum, β, c, α, γ = model

    # Initial conditions
    status = 0
    wage_idx = 0

    wage_path = []
    status_path = []

    for t in range(T):
        wage_path.append(w_vals[wage_idx])
```

(continues on next page)

(continued from previous page)

```
        status_path.append(status)

        key, subkey = jax.random.split(key)
        status, wage_idx = update_agent(
            subkey, status, wage_idx, model, w_bar
        )

    return jnp.array(wage_path), jnp.array(status_path)
```

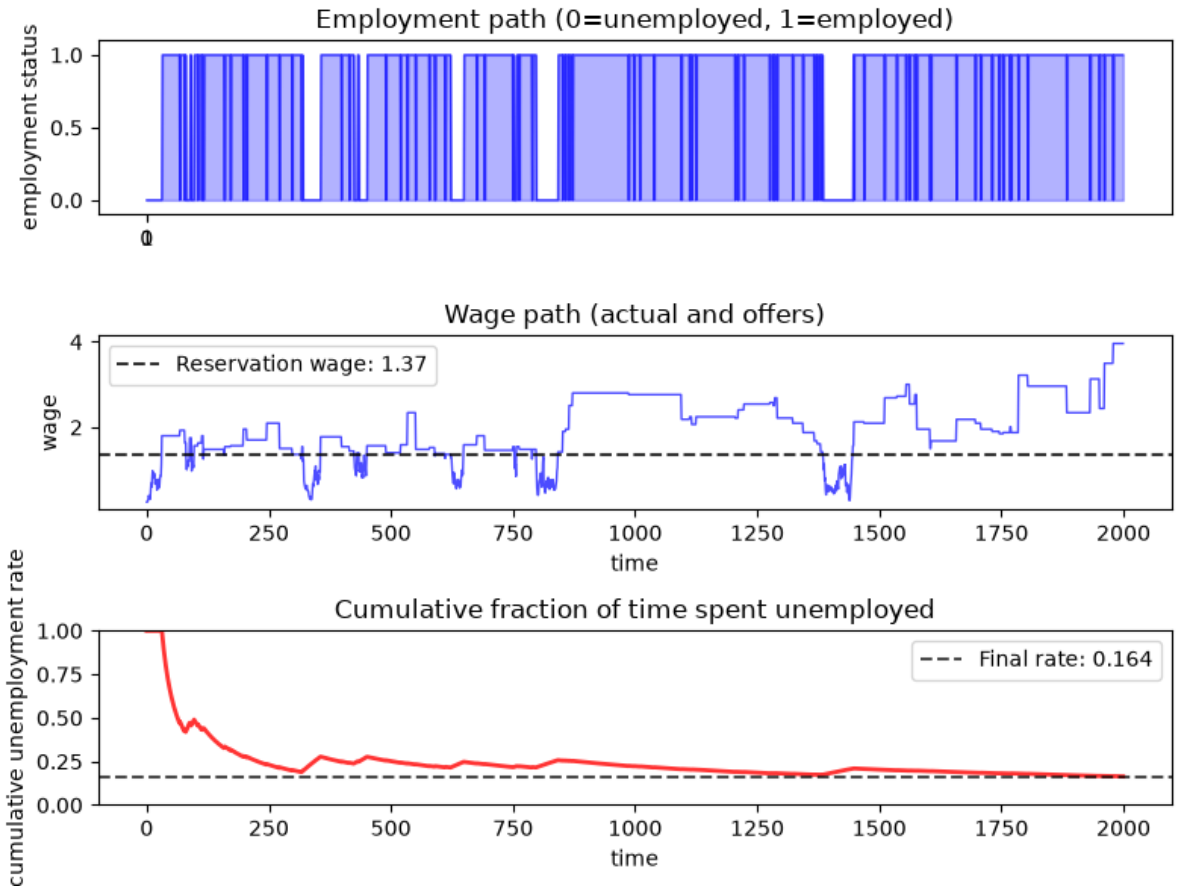
Let's create a comprehensive plot of the employment simulation:

```
model = create_js_with_sep_model()

# Calculate reservation wage for plotting
v_u = vfi(model)
w_bar = get_reservation_wage(v_u, model)

wage_path, employment_status = simulate_employment_path(model, w_bar)

fig, (ax1, ax2, ax3) = plt.subplots(3, 1, figsize=(8, 6))

# Plot employment status
ax1.plot(employment_status, 'b-', alpha=0.7, linewidth=1)
ax1.fill_between(
    range(len(employment_status)), employment_status, alpha=0.3, color='blue'
)
ax1.set_ylabel('employment status')
ax1.set_title('Employment path (0=unemployed, 1=employed)')
ax1.set_xticks((0, 1))
ax1.set_ylim(-0.1, 1.1)

# Plot wage path with employment status coloring
ax2.plot(wage_path, 'b-', alpha=0.7, linewidth=1)
ax2.axhline(y=w_bar, color='black', linestyle='--', alpha=0.8,
           label=f'Reservation wage: {w_bar:.2f}')
ax2.set_xlabel('time')
ax2.set_ylabel('wage')
ax2.set_title('Wage path (actual and offers)')
ax2.legend()

# Plot cumulative fraction of time unemployed
unemployed_indicator = (employment_status == 0).astype(int)
cumulative_unemployment = (
    jnp.cumsum(unemployed_indicator) /
    jnp.arange(1, len(employment_status) + 1)
)

ax3.plot(cumulative_unemployment, 'r-', alpha=0.8, linewidth=2)
ax3.axhline(y=jnp.mean(unemployed_indicator), color='black',
            linestyle='--', alpha=0.7,
            label=f'Final rate: {jnp.mean(unemployed_indicator):.3f}')
ax3.set_xlabel('time')
ax3.set_ylabel('cumulative unemployment rate')
ax3.set_title('Cumulative fraction of time spent unemployed')
ax3.legend()
ax3.set_ylim(0, 1)
```

(continues on next page)

(continued from previous page)

```
plt.tight_layout()
plt.show()
```



The simulation helps to visualize outcomes associated with this model.

The agent follows a reservation wage strategy.

Often the agent loses her job and immediately takes another job at a different wage.

This is because she uses the wage w from her last job to draw a new wage offer via $P(w, \cdot)$, and positive correlation means that a high current w is often leads a high new draw.

52.6 Ergodic property

Below we examine cross-sectional unemployment.

In particular, we will look at the unemployment rate in a cross-sectional simulation and compare it to the time-average unemployment rate, which is the fraction of time an agent spends unemployed over a long time series.

We will see that these two values are approximately equal – in fact they are exactly equal in the limit.

The reason is that the process (S_t, W_t), where

- S_t is the employment status and
- W_t is the wage

is Markovian, since the next pair depends only on the current pair and iid randomness, and ergodic.

Ergodicity holds as a result of irreducibility.

Indeed, from any (status, wage) pair, an agent can eventually reach any other (status, wage) pair.

This holds because:

- Unemployed agents can become employed by accepting offers
- Employed agents can become unemployed through separation (probability α)
- The wage process can transition between all wage states (because P is itself irreducible)

These properties ensure the chain is ergodic with a unique stationary distribution π over states (s, w).

For an ergodic Markov chain, the ergodic theorem guarantees that time averages = cross-sectional averages.

In particular, the fraction of time a single agent spends unemployed (across all wage states) converges to the cross-sectional unemployment rate:

$$\lim_{T\to\infty} \frac{1}{T}\sum_{t=1}^{T} \mathbb{1}\{S_t = \text{unemployed}\} = \sum_{w=1}^{n} \pi(\text{unemployed}, w)$$

This holds regardless of initial conditions – provided that we burn in the cross-sectional distribution (run it forward in time from a given initial cross section in order to remove the influence of that initial condition).

As a result, we can study steady-state unemployment either by:

- Following one agent for a long time (time average), or
- Observing many agents at a single point in time (cross-sectional average)

Often the second approach is better for our purposes, since it's easier to parallelize.

52.7 Cross-sectional analysis

Now let's simulate many agents simultaneously to examine the cross-sectional unemployment rate.

To do this efficiently, we need a different approach than `simulate_employment_path` defined above.

The key differences are:

- `simulate_employment_path` records the entire history (all T periods) for a single agent, which is useful for visualization but memory-intensive
- The new function `sim_agent` below only tracks and returns the final state, which is all we need for cross-sectional statistics
- `sim_agent` uses `lax.fori_loop` instead of a Python loop, making it JIT-compilable and suitable for vectorization across many agents

We first define a function that simulates a single agent forward T time steps:

```
@jax.jit
def sim_agent(key, initial_status, initial_wage_idx, model, w_bar, T):
    """
    Simulate a single agent forward T time steps using lax.fori_loop.

    Uses fold_in to generate a new key at each time step.

    Parameters:
```

(continues on next page)

(continued from previous page)

```
    - key: JAX random key for this agent
    - initial_status: Initial employment status (0 or 1)
    - initial_wage_idx: Initial wage index
    - model: Model instance
    - w_bar: Reservation wage
    - T: Number of time periods to simulate

    Returns:
    - final_status: Employment status after T periods
    - final_wage_idx: Wage index after T periods
    """
    def update(t, loop_state):
        status, wage_idx = loop_state
        step_key = jax.random.fold_in(key, t)
        status, wage_idx = update_agent(step_key, status, wage_idx, model, w_bar)
        return status, wage_idx

    initial_loop_state = (initial_status, initial_wage_idx)
    final_loop_state = lax.fori_loop(0, T, update, initial_loop_state)
    final_status, final_wage_idx = final_loop_state
    return final_status, final_wage_idx


# Create vectorized version of sim_agent to process multiple agents in parallel
sim_agents_vmap = jax.vmap(sim_agent, in_axes=(0, 0, 0, None, None, None))


def simulate_cross_section(
        model: Model,               # Model instance with parameters
        n_agents: int = 100_000,    # Number of agents to simulate
        T: int = 200,               # Length of burn-in
        seed: int = 42              # For reproducibility
    ) -> float:
    """
    Simulate cross-section of agents and return unemployment rate.

    This approach:
    1. Generates n_agents random keys
    2. Calls sim_agent for each agent (vectorized via vmap)
    3. Collects the final states to produce the cross-section

    Returns the cross-sectional unemployment rate.
    """
    key = jax.random.PRNGKey(seed)

    # Solve for optimal reservation wage
    v_u = vfi(model)
    w_bar = get_reservation_wage(v_u, model)

    # Initialize arrays
    initial_wage_indices = jnp.zeros(n_agents, dtype=jnp.int32)
    initial_status_vec = jnp.zeros(n_agents, dtype=jnp.int32)

    # Generate n_agents random keys
    agent_keys = jax.random.split(key, n_agents)

    # Simulate each agent forward T steps (vectorized)
```

(continues on next page)

(continued from previous page)

```
    final_status, final_wage_idx = sim_agents_vmap(
        agent_keys, initial_status_vec, initial_wage_indices, model, w_bar, T
    )

    unemployment_rate = 1 - jnp.mean(final_status)
    return unemployment_rate
```

This function generates a histogram showing the distribution of employment status across many agents:

```
def plot_cross_sectional_unemployment(
        model: Model,
        t_snapshot: int = 200,     # Time of cross-sectional snapshot
        n_agents: int = 20_000     # Number of agents to simulate
    ):
    """
    Generate histogram of cross-sectional unemployment at a specific time.

    """
    # Get final employment state directly
    key = jax.random.PRNGKey(42)
    v_u = vfi(model)
    w_bar = get_reservation_wage(v_u, model)

    # Initialize arrays
    initial_wage_indices = jnp.zeros(n_agents, dtype=jnp.int32)
    initial_status_vec = jnp.zeros(n_agents, dtype=jnp.int32)

    # Generate n_agents random keys
    agent_keys = jax.random.split(key, n_agents)

    # Simulate each agent forward T steps (vectorized)
    final_status, _ = sim_agents_vmap(
        agent_keys, initial_status_vec, initial_wage_indices, model, w_bar, t_snapshot
    )

    # Calculate unemployment rate
    unemployment_rate = 1 - jnp.mean(final_status)

    fig, ax = plt.subplots(figsize=(8, 5))

    # Plot histogram as density (bars sum to 1)
    weights = jnp.ones_like(final_status) / len(final_status)
    ax.hist(final_status, bins=[-0.5, 0.5, 1.5],
            alpha=0.7, color='blue', edgecolor='black',
            density=True, weights=weights)

    ax.set_xlabel('employment status (0=unemployed, 1=employed)')
    ax.set_ylabel('density')
    ax.set_title(f'Cross-sectional distribution at t={t_snapshot}, ' +
                 f'unemployment rate = {unemployment_rate:.3f}')
    ax.set_xticks([0, 1])

    plt.tight_layout()
    plt.show()
```

Now let's compare the time-average unemployment rate (from a single agent's long simulation) with the cross-sectional unemployment rate (from many agents at a single point in time).

We claimed above that these numbers will be approximately equal in large samples, due to ergodicity.

Let's see if that's true.

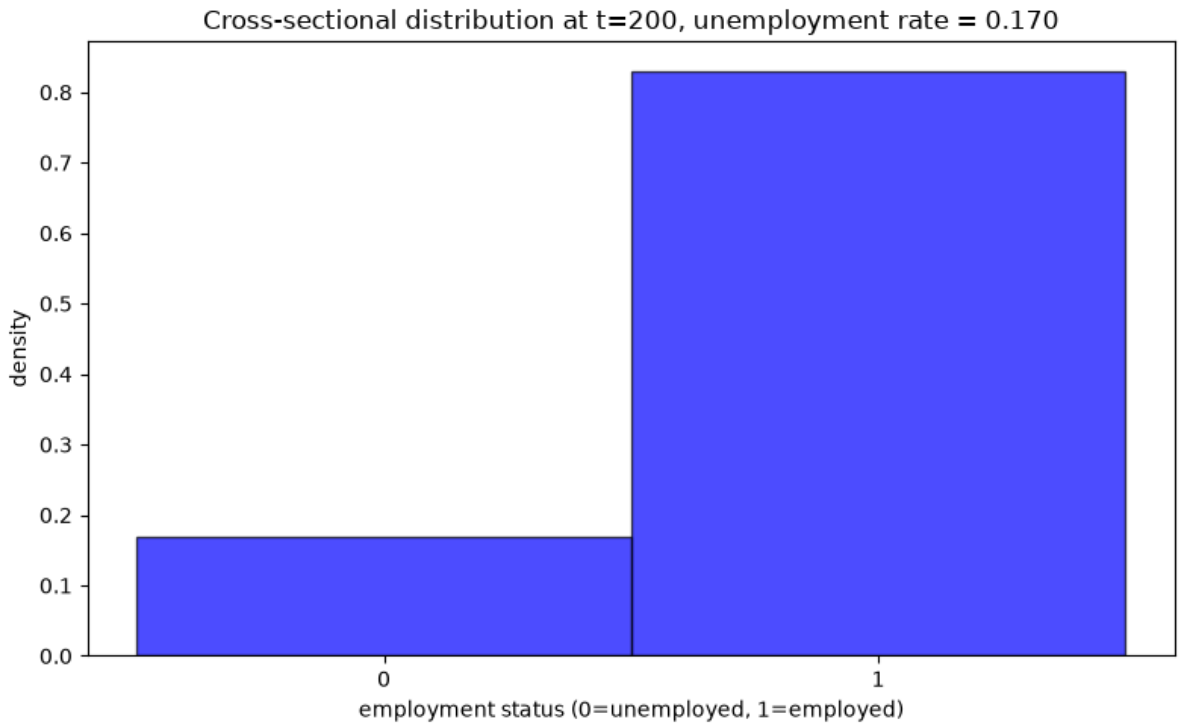
```
model = create_js_with_sep_model()
cross_sectional_unemp = simulate_cross_section(
    model, n_agents=20_000, T=200
)

time_avg_unemp = jnp.mean(unemployed_indicator)
print(f"Time-average unemployment rate (single agent): "
      f"{time_avg_unemp:.4f}")
print(f"Cross-sectional unemployment rate (at t=200): "
      f"{cross_sectional_unemp:.4f}")
print(f"Difference: {abs(time_avg_unemp - cross_sectional_unemp):.4f}")
```

```
Time-average unemployment rate (single agent): 0.1640
Cross-sectional unemployment rate (at t=200): 0.1696
Difference: 0.0056
```

Indeed, they are very close.

Now let's visualize the cross-sectional distribution:

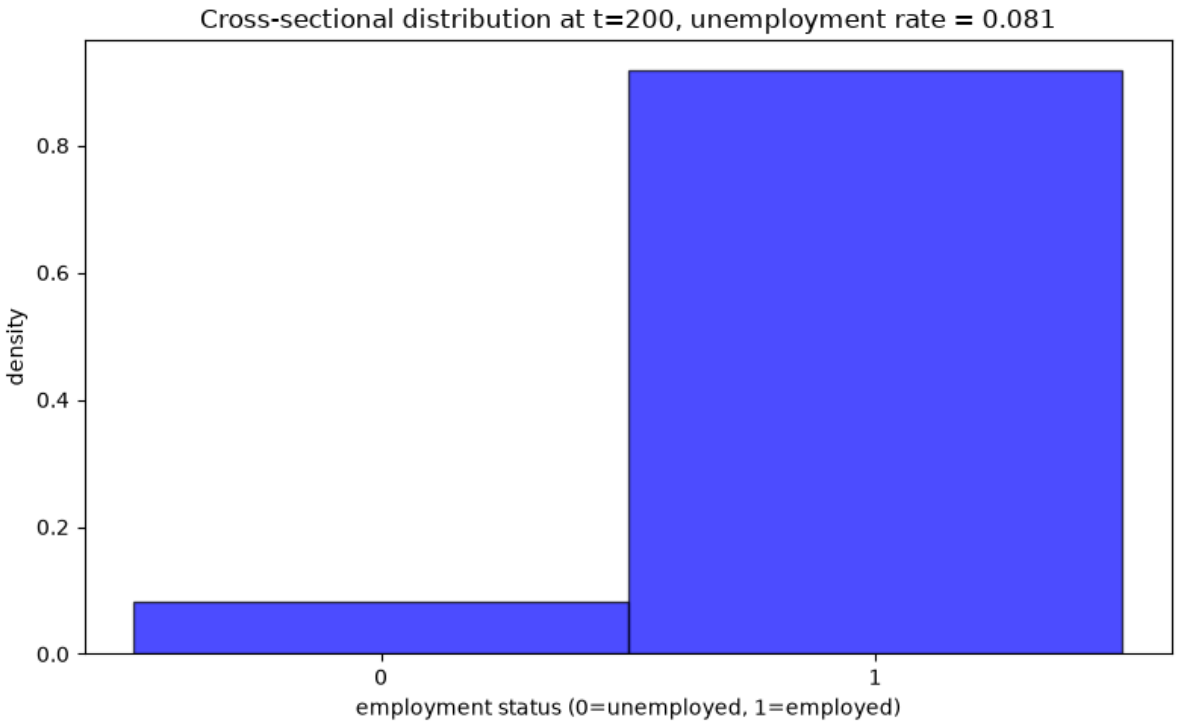
```
plot_cross_sectional_unemployment(model)
```



52.8 Lower unemployment compensation (c=0.5)

What happens to the cross-sectional unemployment rate with lower unemployment compensation?

```
model_low_c = create_js_with_sep_model(c=0.5)
plot_cross_sectional_unemployment(model_low_c)
```



52.9 Exercises

Exercise 52.9.1

Create a plot that investigates more carefully how the steady state cross-sectional unemployment rate changes with unemployment compensation.

Try a range of values for unemployment compensation `c`, such as `c = 0.2, 0.4, 0.6, 0.8, 1.0`. For each value, compute the steady-state cross-sectional unemployment rate and plot it against `c`.

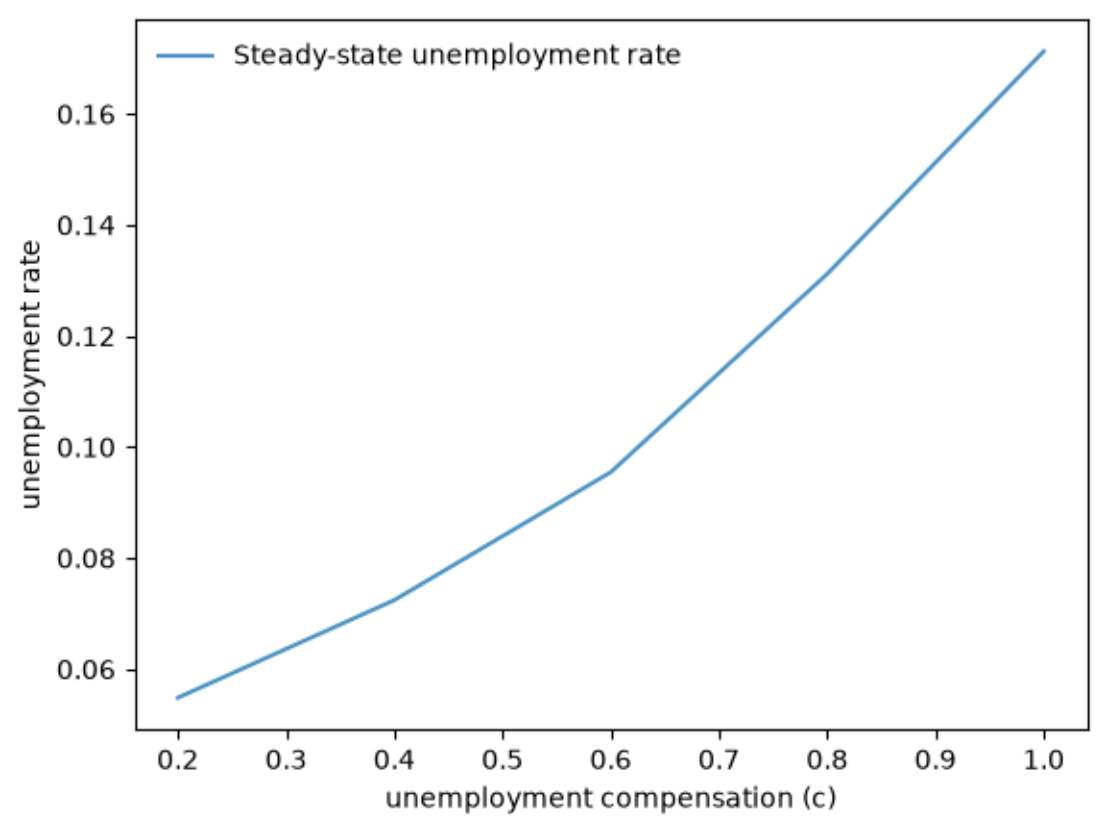
What relationship do you observe between unemployment compensation and the unemployment rate?

Solution

We compute the steady-state unemployment rate for different values of unemployment compensation:

```
c_values = 1.0, 0.8, 0.6, 0.4, 0.2
rates = []
for c in c_values:
    model = create_js_with_sep_model(c=c)
```

```
    unemployment_rate = simulate_cross_section(model)
    rates.append(unemployment_rate)

fig, ax = plt.subplots()
ax.plot(
    c_values, rates, alpha=0.8,
    linewidth=1.5, label='Steady-state unemployment rate'
)
ax.set_xlabel('unemployment compensation (c)')
ax.set_ylabel('unemployment rate')
ax.legend(frameon=False)
plt.show()
```



CHAPTER

FIFTYTHREE

JOB SEARCH IV: FITTED VALUE FUNCTION ITERATION

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

Contents

- *Job Search IV: Fitted Value Function Iteration*
  - *Overview*
  - *Model*
  - *Solution method*
  - *Implementation*
  - *Simulation*
  - *Exercises*

53.1 Overview

This lecture follows on from the job search model with separation presented in the *previous lecture*.

That lecture combined exogenous job separation events and a Markov wage offer process.

In this lecture we continue with this set and, in addition, allow the wage offer process to be continuous rather than discrete.

In particular,

$$W_t = \exp(X_t) \quad \text{where} \quad X_{t+1} = \rho X_t + \nu Z_{t+1}$$

and $\{Z_t\}$ is IID and standard normal.

While we already considered continuous wage distributions briefly in *Job Search I: The McCall Search Model*, the change was relatively trivial in that case.

The reason is that we were able to reduce the problem to solving for a single scalar value (the continuation value).

Here, in our Markov setting, the change is less trivial, since a continuous wage distribution leads to an uncountably infinite state space.

The infinite state space leads to additional challenges, particularly when it comes to applying value function iteration (VFI).

These challenges will lead us to modify VFI by adding an interpolation step.

The combination of VFI and this interpolation step is called **fitted value function iteration** (fitted VFI).

Fitted VFI is very common in practice, so we will take some time to work through the details.

In addition to what's in Anaconda, this lecture will need the following libraries

```
!pip install quantecon jax
```

We will use the following imports:

```
import matplotlib.pyplot as plt
import jax
import jax.numpy as jnp
from jax import lax
from typing import NamedTuple
from functools import partial
import quantecon as qe
```

53.2 Model

Assuming that readers are familiar with the content of *Job Search III: Search with Separation and Markov Wages*, the model can be summarized as follows.

- Wage offers follow a continuous Markov process: $W_t = \exp(X_t)$ where $X_{t+1} = \rho X_t + \nu Z_{t+1}$
- $\{Z_t\}$ is IID and standard normal
- Jobs terminate with probability α each period (separation rate)
- Unemployed workers receive compensation c per period
- Workers have CRRA utility $u(x) = \frac{x^{1-\gamma}-1}{1-\gamma}$
- Future payoffs are discounted by factor $\beta \in (0, 1)$

53.3 Solution method

Let's discuss how we can solve this model.

The only real change from *Job Search III: Search with Separation and Markov Wages* is that we replace sums with integrals.

53.3.1 Value function iteration

In the *discrete case*, we ended up iterating on the Bellman operator

$$(Tv_u)(w) = \max\left\{\frac{1}{1-\beta(1-\alpha)} \cdot (u(w) + \alpha\beta(Pv_u)(w)), u(c) + \beta(Pv_u)(w)\right\} \tag{53.1}$$

where

$$(Pv_u)(w) := \sum_{w'} v_u(w')P(w, w')$$

Here we iterate on the same law after changing the definition of the P operator to

$$(Pv_u)(w) := \int v_u(w')p(w, w')dw'$$

where $p(w, \cdot)$ is the conditional density of w' given w.

Here we are thinking of v_u as a function on all of $\mathbb{R}_+$.

After taking ψ to be the standard normal density, we can write the expression above more explicitly as

$$(Pv_u)(w) := \int v_u(w^\rho \exp(\nu z))\psi(z)dz,$$

To understand this expression, recall that $W_t = \exp(X_t)$ where $X_{t+1} = \rho X_t + \nu Z_{t+1}$.

As a result $W_{t+1} = \exp(X_{t+1}) = \exp(\rho \log(W_t) + \nu Z_{t+1}) = W_t^\rho \exp(\nu Z_{t+1})$.

The integral above regards the current wage W_t as fixed at w and takes the expectation of $v_u(w^\rho \exp(\nu Z_{t+1}))$.

53.3.2 Fitting

In theory, we should now proceed as follows:

1. Begin with a guess v
2. Applying T to obtain the update $v' = Tv$
3. Unless some stopping condition is satisfied, set $v = v'$ and go to step 2.

However, there is a problem we must confront before we implement this procedure: The iterates of the value function can neither be calculated exactly nor stored on a computer.

To see the issue, consider (53.1).

Even if v is a known function, the only way to store its update v' is to record its value $v'(w)$ for every $w \in \mathbb{R}_+$.

Clearly, this is impossible.

53.3.3 Fitted value function iteration

What we will do instead is use **fitted value function iteration**.

The procedure is as follows:

Let a current guess v be given.

Now we record the value of the function v' at only finitely many "grid" points $w_1 < w_2 < \cdots < w_I$ and then reconstruct v' from this information when required.

More precisely, the algorithm will be

1. Begin with an array **v** representing the values of an initial guess of the value function on some grid points $\{w_i\}$.
2. Build a function v on the state space $\mathbb{R}_+$ by interpolation or approximation, based on **v** and $\{w_i\}$.
3. Obtain and record the samples of the updated function $v'(w_i)$ on each grid point w_i.
4. Unless some stopping condition is satisfied, take this as the new array and go to step 1.

How should we go about step 2?

This is a problem of function approximation, and there are many ways to approach it.

What's important here is that the function approximation scheme must not only produce a good approximation to each v, but also that it combines well with the broader iteration algorithm described above.

One good choice from both respects is continuous piecewise linear interpolation.

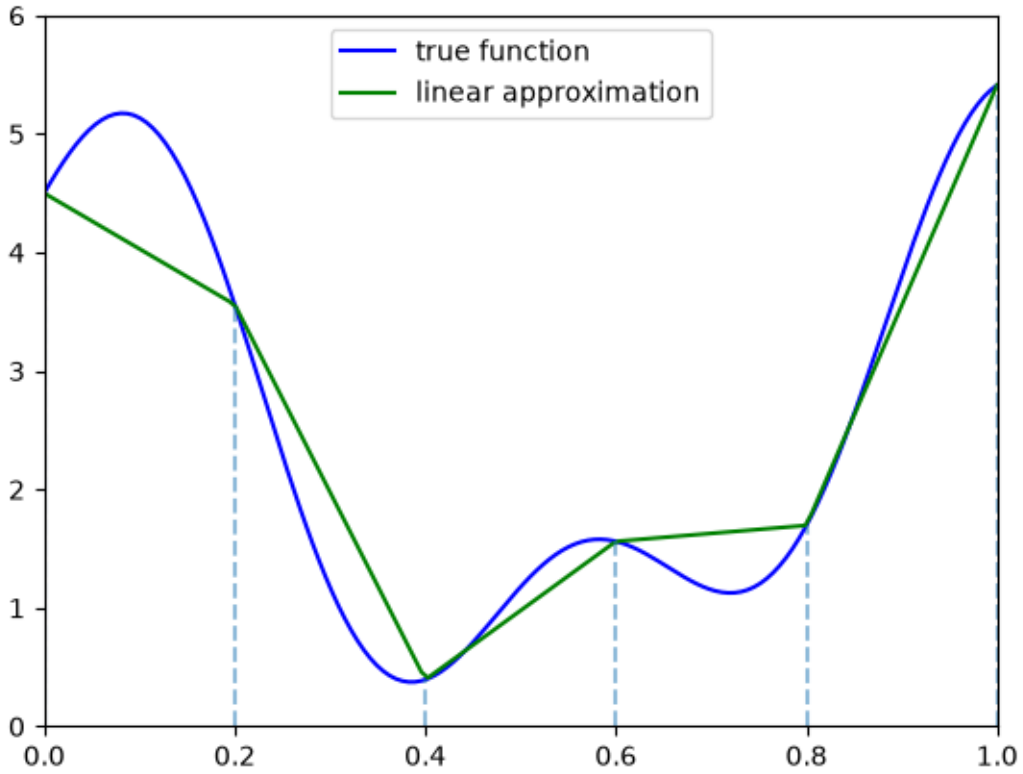
This method

1. combines well with value function iteration (see, e.g., [Gordon, 1995] or [Stachurski, 2008]) and
2. preserves useful shape properties such as monotonicity and concavity/convexity.

Linear interpolation will be implemented using JAX's interpolation function `jnp.interp`.

The next figure illustrates piecewise linear interpolation of an arbitrary function on grid points $0, 0.2, 0.4, 0.6, 0.8, 1$.

```
def f(x):
    y1 = 2 * jnp.cos(6 * x) + jnp.sin(14 * x)
    return y1 + 2.5

c_grid = jnp.linspace(0, 1, 6)
f_grid = jnp.linspace(0, 1, 150)

def Af(x):
    return jnp.interp(x, c_grid, f(c_grid))

fig, ax = plt.subplots()

ax.plot(f_grid, f(f_grid), 'b-', label='true function')
ax.plot(f_grid, Af(f_grid), 'g-', label='linear approximation')
ax.vlines(c_grid, c_grid * 0, f(c_grid), linestyle='dashed', alpha=0.5)

ax.legend(loc="upper center")

ax.set(xlim=(0, 1), ylim=(0, 6))
plt.show()
```



53.4 Implementation

Let's code up and solve the model.

53.4.1 Setup

The first step is to build a JAX-compatible structure for the McCall model with separation and a continuous wage offer distribution.

The key computational challenge is evaluating the conditional expectation $(Pv_u)(w) = \int v_u(w')p(w, w')dw'$ at each wage grid point.

Recall that we have:

$$(Pv_u)(w) = \int v_u(w^\rho \exp(\nu z))\psi(z)dz$$

where ψ is the standard normal density.

We will approximate this integral using Monte Carlo integration with draws $\{Z_i\}$ from the standard normal distribution:

$$(Pv_u)(w) \approx \frac{1}{N}\sum_{i=1}^{N} v_u(w^\rho \exp(\nu Z_i))$$

For this reason, our data structure will include a fixed set of IID $N(0, 1)$ draws $\{Z_i\}$.

```
class Model(NamedTuple):
    c: float              # unemployment compensation
    α: float              # job separation rate
    β: float              # discount factor
    ρ: float              # wage persistence
    ν: float              # wage volatility
    γ: float              # utility parameter
    w_grid: jnp.ndarray   # grid of points for fitted VFI
    z_draws: jnp.ndarray  # draws from the standard normal distribution

def create_mccall_model(
        c: float = 1.0,
        α: float = 0.05,
        β: float = 0.96,
        ρ: float = 0.9,
        ν: float = 0.2,
        γ: float = 1.5,
        grid_size: int = 100,
        mc_size: int = 1000,
        seed: int = 1234
    ):
    """Factory function to create a McCall model instance."""

    key = jax.random.PRNGKey(seed)
    z_draws = jax.random.normal(key, (mc_size,))

    # Discretize just to get a suitable wage grid for interpolation
    mc = qe.markov.tauchen(grid_size, ρ, ν)
    w_grid = jnp.exp(jnp.array(mc.state_values))

    return Model(c, α, β, ρ, ν, γ, w_grid, z_draws)
```

We use the same CRRA utility function as in the discrete case:

```
def u(x, γ):
    return (x**(1 - γ) - 1) / (1 - γ)
```

53.4.2 Iteration

Here is the Bellman operator, where we use Monte Carlo integration to evaluate the expectation.

```
def T(model, v):
    """Update the value function."""

    # Unpack model parameters
    c, α, β, ρ, ν, γ, w_grid, z_draws = model

    # Interpolate array represented value function
    vf = lambda x: jnp.interp(x, w_grid, v)

    def compute_expectation(w):
        # Use Monte Carlo to evaluate integral (P v)(w) = E[v(W' | w)]
        # where W' = w^ρ * exp(ν * Z)
        w_next = w**ρ * jnp.exp(ν * z_draws)
        return jnp.mean(vf(w_next))
```

(continues on next page)

(continued from previous page)

```
    compute_exp_on_grid = jax.vmap(compute_expectation)
    Pv = compute_exp_on_grid(w_grid)

    d = 1 / (1 - β * (1 - α))
    v_e = d * (u(w_grid, γ) + α * β * Pv)
    continuation_values = u(c, γ) + β * Pv
    return jnp.maximum(v_e, continuation_values)
```

Here's the solver, which computes an approximate fixed point v_u of T.

```
@jax.jit
def vfi(
        model: Model,
        tolerance: float = 1e-6,    # Error tolerance
        max_iter: int = 100_000,    # Max iteration bound
    ):
    """
    Compute the fixed point v_u of T.

    """

    v_init = jnp.zeros(model.w_grid.shape)

    def cond(loop_state):
        v, error, i = loop_state
        return (error > tolerance) & (i <= max_iter)

    def update(loop_state):
        v, error, i = loop_state
        v_new = T(model, v)
        error = jnp.max(jnp.abs(v_new - v))
        new_loop_state = v_new, error, i + 1
        return new_loop_state

    initial_state = (v_init, tolerance + 1, 1)
    final_loop_state = lax.while_loop(cond, update, initial_state)
    v_final, error, i = final_loop_state

    return v_final
```

Here's a function that uses a solution v_u to compute the remaining functions of interest: v_e, and the continuation value function h.

We use the same expressions as we did in the *discrete case*, after replacing sums with integrals.

```
def compute_solution_functions(model, v_u):

    # Unpack model parameters
    c, α, β, ρ, ν, γ, w_grid, z_draws = model

    # Interpolate v_u on the wage grid
    vf = lambda x: jnp.interp(x, w_grid, v_u)

    def compute_expectation(w):
        # Use Monte Carlo to evaluate integral (P v)(w)
        # Compute E[v(w' | w)] where w' = w^ρ * exp(ν * z)
        w_next = w**ρ * jnp.exp(ν * z_draws)
```

(continues on next page)

(continued from previous page)

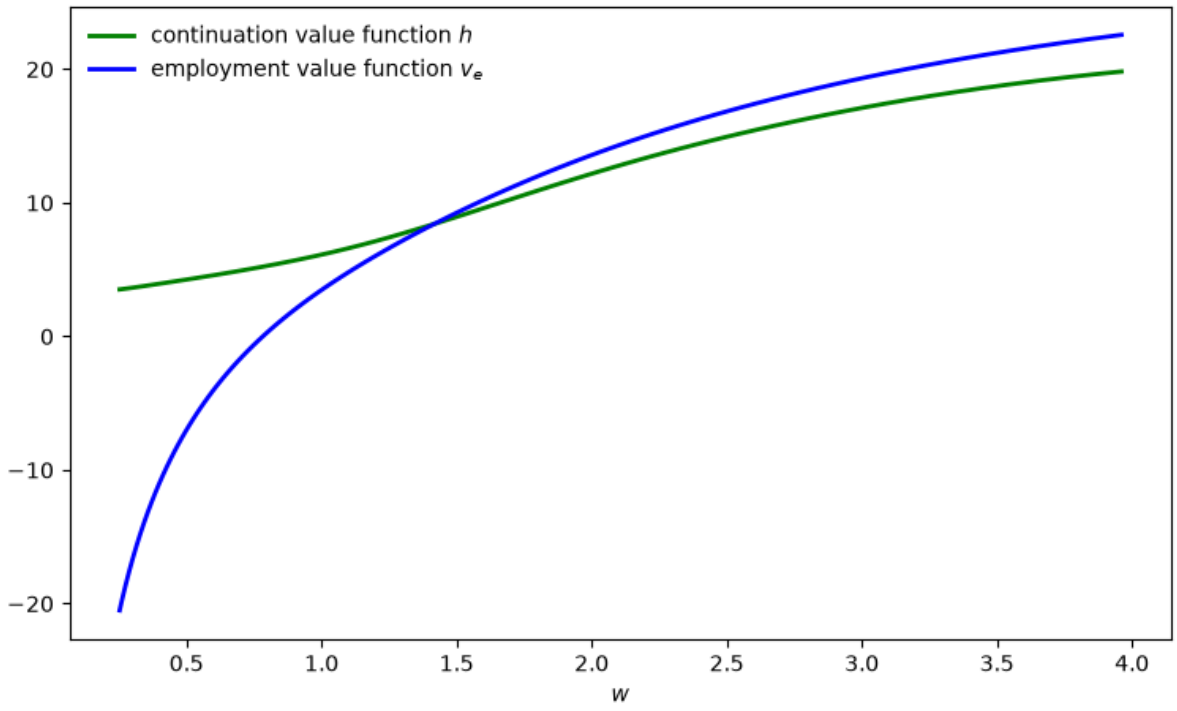
```
        return jnp.mean(vf(w_next))

    compute_exp_on_grid = jax.vmap(compute_expectation)
    Pv = compute_exp_on_grid(w_grid)

    d = 1 / (1 - β * (1 - α))
    v_e = d * (u(w_grid, γ) + α * β * Pv)
    h = u(c, γ) + β * Pv

    return v_e, h
```

Let's try solving the model:

```
model = create_mccall_model()
c, α, β, ρ, ν, γ, w_grid, z_draws = model
v_u = vfi(model)
v_e, h = compute_solution_functions(model, v_u)
```

Let's plot our results.

```
fig, ax = plt.subplots(figsize=(9, 5.2))
ax.plot(w_grid, h, 'g-', linewidth=2,
        label="continuation value function $h$")
ax.plot(w_grid, v_e, 'b-', linewidth=2,
        label="employment value function $v_e$")
ax.legend(frameon=False)
ax.set_xlabel(r"$w$")
plt.show()
```



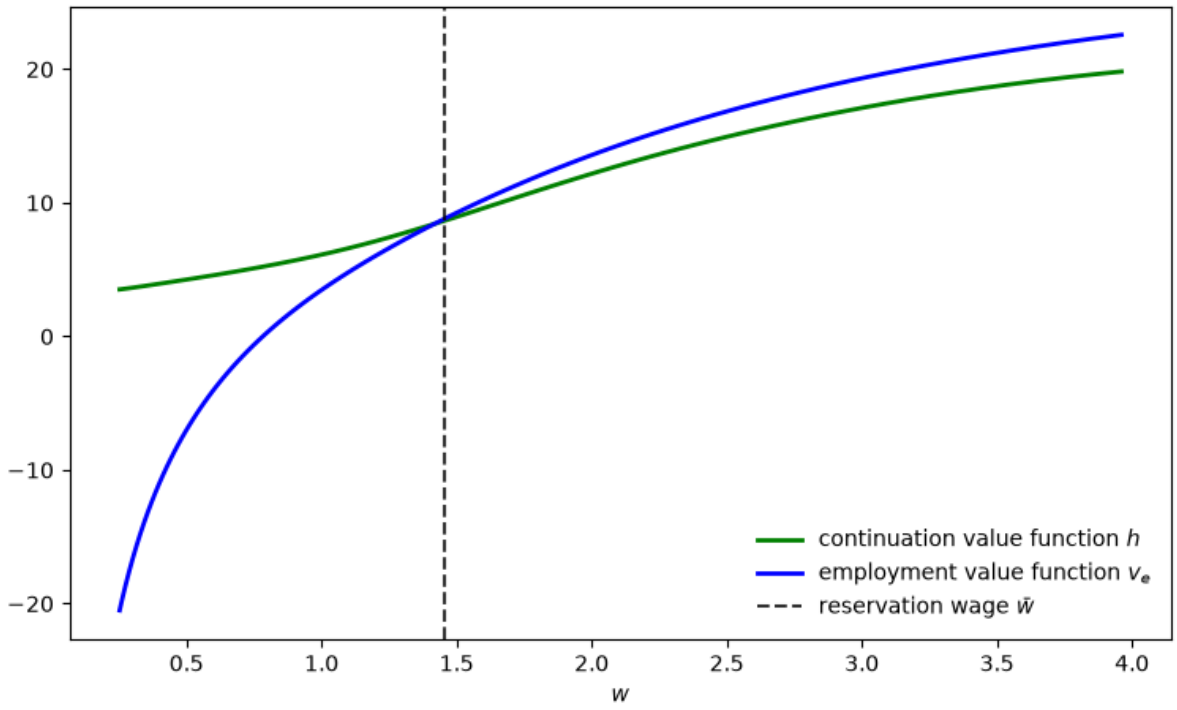
The reservation wage is at the intersection of the employment value function v_e and the continuation value function h.

Here's a function to compute it explicitly.

```
@jax.jit
def get_reservation_wage(model: Model) -> float:
    """
    Calculate the reservation wage for a given model.

    """
    c, α, β, ρ, ν, γ, w_grid, z_draws = model

    v_u = vfi(model)
    v_e, h = compute_solution_functions(model, v_u)

    # Compute optimal policy (acceptance indices)
    σ = v_e >= h

    # Find first index where policy indicates acceptance
    first_accept_idx = jnp.argmax(σ) # returns first True value

    # If no acceptance (all False), return infinity
    # Otherwise return the wage at the first acceptance index
    return jnp.where(jnp.any(σ), w_grid[first_accept_idx], jnp.inf)
```

Let's repeat our plot, but now inserting the reservation wage.

```
w_bar = get_reservation_wage(model)

fig, ax = plt.subplots(figsize=(9, 5.2))
ax.plot(w_grid, h, 'g-', linewidth=2,
        label="continuation value function $h$")
ax.plot(w_grid, v_e, 'b-', linewidth=2,
        label="employment value function $v_e$")
ax.axvline(x=w_bar, color='black', linestyle='--', alpha=0.8,
           label=f'reservation wage $\\bar{{w}}$')
ax.legend(frameon=False)
ax.set_xlabel(r"$w$")
plt.show()
```



53.5 Simulation

Now we run some simulations with a focus on unemployment rate.

53.5.1 Single agent dynamics

Let's simulate the employment path of a single agent under the optimal policy.

We need a function to update the agent's state by one period.

```
def update_agent(key, status, wage, model, w_bar):
    """
    Updates an agent's employment status and current wage by one period.

    Parameters:
    - key: JAX random key
    - status: Current employment status (0 or 1)
    - wage: Current wage if employed, current offer if unemployed
    - model: Model instance
    - w_bar: Reservation wage

    """
    c, α, β, ρ, ν, γ, w_grid, z_draws = model

    # Draw new wage offer based on current wage
    key1, key2 = jax.random.split(key)
    z = jax.random.normal(key1)
    new_wage = wage**ρ * jnp.exp(ν * z)
```

(continues on next page)

(continued from previous page)

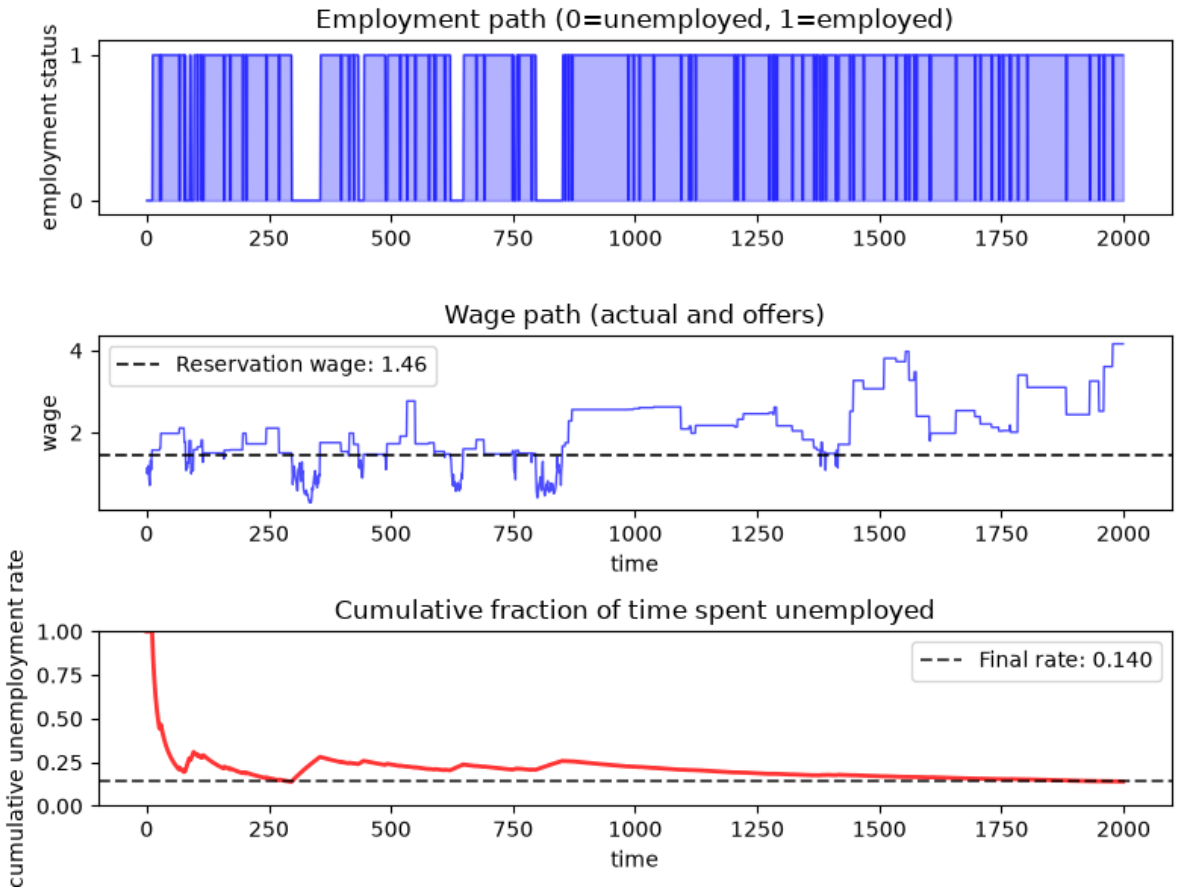
```
    # Check if separation occurs (for employed workers)
    separation_occurs = jax.random.uniform(key2) < α

    # Accept if current wage meets or exceeds reservation wage
    accepts = wage >= w_bar

    # If employed: status = 1 if no separation, 0 if separation
    # If unemployed: status = 1 if accepts, 0 if rejects
    next_status = jnp.where(
        status,
        1 - separation_occurs.astype(jnp.int32),  # employed path
        accepts.astype(jnp.int32)                 # unemployed path
    )

    # If employed: wage = current if no separation, new if separation
    # If unemployed: wage = current if accepts, new if rejects
    next_wage = jnp.where(
        status,
        jnp.where(separation_occurs, new_wage, wage),  # employed path
        jnp.where(accepts, wage, new_wage)             # unemployed path
    )

    return next_status, next_wage
```

Here's a function to simulate the employment path of a single agent.

```
def simulate_employment_path(
        model: Model,      # Model details
        w_bar: float,      # Reservation wage
        T: int = 2_000,    # Simulation length
        seed: int = 42     # Set seed for simulation
    ):
    """
    Simulate employment path for T periods starting from unemployment.

    """
    key = jax.random.PRNGKey(seed)
    c, α, β, ρ, ν, γ, w_grid, z_draws = model

    # Initial conditions: start unemployed with initial wage draw
    status = 0
    key, subkey = jax.random.split(key)
    wage = jnp.exp(jax.random.normal(subkey) * ν)

    wage_path = []
    status_path = []

    for t in range(T):
        wage_path.append(wage)
        status_path.append(status)

        key, subkey = jax.random.split(key)
        status, wage = update_agent(
            subkey, status, wage, model, w_bar
        )

    return jnp.array(wage_path), jnp.array(status_path)
```

Let's create a comprehensive plot of the employment simulation:

```
model = create_mccall_model()
w_bar = get_reservation_wage(model)

wage_path, employment_status = simulate_employment_path(model, w_bar)

fig, (ax1, ax2, ax3) = plt.subplots(3, 1, figsize=(8, 6))

# Plot employment status
ax1.plot(employment_status, 'b-', alpha=0.7, linewidth=1)
ax1.fill_between(
    range(len(employment_status)), employment_status, alpha=0.3, color='blue'
)
ax1.set_ylabel('employment status')
ax1.set_title('Employment path (0=unemployed, 1=employed)')
ax1.set_yticks((0, 1))
ax1.set_ylim(-0.1, 1.1)

# Plot wage path with reservation wage
ax2.plot(wage_path, 'b-', alpha=0.7, linewidth=1)
ax2.axhline(y=w_bar, color='black', linestyle='--', alpha=0.8,
           label=f'Reservation wage: {w_bar:.2f}')
ax2.set_xlabel('time')
ax2.set_ylabel('wage')
ax2.set_title('Wage path (actual and offers)')
ax2.legend()

# Plot cumulative fraction of time unemployed
unemployed_indicator = (employment_status == 0).astype(int)
cumulative_unemployment = (
    jnp.cumsum(unemployed_indicator) /
    jnp.arange(1, len(employment_status) + 1)
)

ax3.plot(cumulative_unemployment, 'r-', alpha=0.8, linewidth=2)
ax3.axhline(y=jnp.mean(unemployed_indicator), color='black',
            linestyle='--', alpha=0.7,
            label=f'Final rate: {jnp.mean(unemployed_indicator):.3f}')
ax3.set_xlabel('time')
ax3.set_ylabel('cumulative unemployment rate')
ax3.set_title('Cumulative fraction of time spent unemployed')
ax3.legend()
ax3.set_ylim(0, 1)

plt.tight_layout()
plt.show()
```



The simulation shows the agent cycling between employment and unemployment.

The agent starts unemployed and receives wage offers according to the Markov process.

When unemployed, the agent accepts offers that exceed the reservation wage.

When employed, the agent faces job separation with probability α each period.

53.5.2 Cross-sectional analysis

Now let's simulate many agents simultaneously to examine the cross-sectional unemployment rate.

To do this efficiently, we need a different approach than `simulate_employment_path` defined above.

The key differences are:

- `simulate_employment_path` records the entire history (all T periods) for a single agent, which is useful for visualization but memory-intensive
- The new function `sim_agent` below only tracks and returns the final state, which is all we need for cross-sectional statistics
- `sim_agent` uses `lax.fori_loop` instead of a Python loop, making it JIT-compilable and suitable for vectorization across many agents

We first define a function that simulates a single agent forward T time steps:

```
@jax.jit
def sim_agent(key, initial_status, initial_wage, model, w_bar, T):
    """
    Simulate a single agent forward T time steps using lax.fori_loop.

    Uses fold_in to generate a new key at each time step.

    Parameters:
    - key: JAX random key for this agent
    - initial_status: Initial employment status (0 or 1)
    - initial_wage: Initial wage
    - model: Model instance
    - w_bar: Reservation wage
    - T: Number of time periods to simulate

    Returns:
    - final_status: Employment status after T periods
    - final_wage: Wage after T periods
    """
    def update(t, loop_state):
        status, wage = loop_state
        step_key = jax.random.fold_in(key, t)
        status, wage = update_agent(step_key, status, wage, model, w_bar)
        return status, wage

    initial_loop_state = (initial_status, initial_wage)
    final_loop_state = lax.fori_loop(0, T, update, initial_loop_state)
    final_status, final_wage = final_loop_state
    return final_status, final_wage


# Create vectorized version of sim_agent to process multiple agents in parallel
sim_agents_vmap = jax.vmap(sim_agent, in_axes=(0, 0, 0, None, None, None))


def simulate_cross_section(
        model: Model,
        n_agents: int = 100_000,
        T: int = 200,
        seed: int = 42
    ) -> float:
    """
    Simulate cross-section of agents and return unemployment rate.

    This approach:
    1. Generates n_agents random keys
    2. Calls sim_agent for each agent (vectorized via vmap)
    3. Collects the final states to produce the cross-section

    Returns the cross-sectional unemployment rate.
    """
    c, α, β, ρ, ν, γ, w_grid, z_draws = model

    key = jax.random.PRNGKey(seed)

    # Solve for optimal reservation wage
    w_bar = get_reservation_wage(model)
```

(continues on next page)

(continued from previous page)

```
    # Initialize arrays
    init_key, subkey = jax.random.split(key)
    initial_wages = jnp.exp(jax.random.normal(subkey, (n_agents,)) * v)
    initial_status_vec = jnp.zeros(n_agents, dtype=jnp.int32)

    # Generate n_agents random keys
    agent_keys = jax.random.split(init_key, n_agents)

    # Simulate each agent forward T steps (vectorized)
    final_status, final_wages = sim_agents_vmap(
        agent_keys, initial_status_vec, initial_wages, model, w_bar, T
    )

    unemployment_rate = 1 - jnp.mean(final_status)
    return unemployment_rate
```

Now let's compare the time-average unemployment rate (from a single agent's long simulation) with the cross-sectional unemployment rate (from many agents at a single point in time).

```
model = create_mccall_model()
cross_sectional_unemp = simulate_cross_section(
    model, n_agents=20_000, T=200
)

time_avg_unemp = jnp.mean(unemployed_indicator)
print(f"Time-average unemployment rate (single agent, T=2000): "
      f"{time_avg_unemp:.4f}")
print(f"Cross-sectional unemployment rate (at t=200): "
      f"{cross_sectional_unemp:.4f}")
print(f"Difference: {abs(time_avg_unemp - cross_sectional_unemp):.4f}")
```

```
Time-average unemployment rate (single agent, T=2000): 0.1395
Cross-sectional unemployment rate (at t=200): 0.2029
Difference: 0.0634
```

The difference above can be further reduced by increasing the simulation length for the single agent.

```
wage_path_long, employment_status_long = simulate_employment_path(model, w_bar, T=10_
↪000)
unemployed_indicator_long = (employment_status_long == 0).astype(int)
time_avg_unemp_long = jnp.mean(unemployed_indicator_long)

print(f"Time-average unemployment rate (single agent, T=10000): "
      f"{time_avg_unemp_long:.4f}")
print(f"Cross-sectional unemployment rate (at t=200): "
      f"{cross_sectional_unemp:.4f}")
print(f"Difference: {abs(time_avg_unemp_long - cross_sectional_unemp):.4f}")
```

```
Time-average unemployment rate (single agent, T=10000): 0.2071
Cross-sectional unemployment rate (at t=200): 0.2029
Difference: 0.0042
```

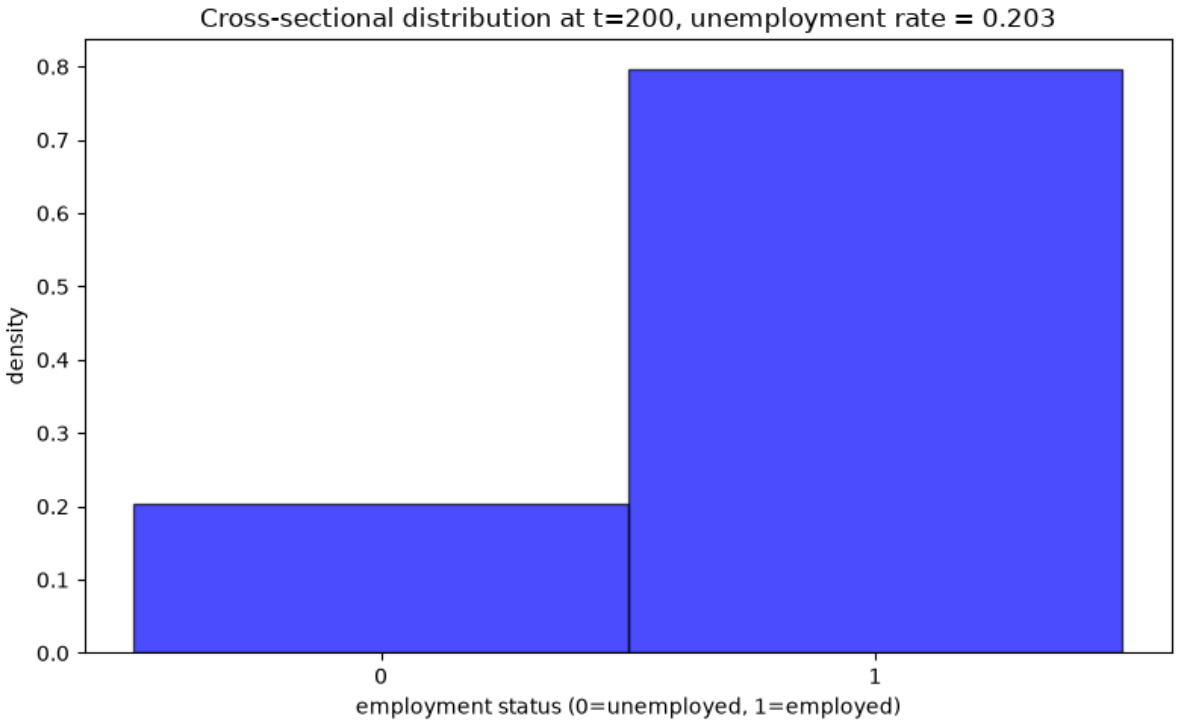
53.5.3 Visualization

This function generates a histogram showing the distribution of employment status across many agents:

```
def plot_cross_sectional_unemployment(
        model: Model,            # Model instance with parameters
        t_snapshot: int = 200,   # Time for cross-sectional snapshot
        n_agents: int = 20_000   # Number of agents to simulate
    ):
    """
    Generate histogram of cross-sectional unemployment at a specific time.

    """
    c, α, β, ρ, ν, γ, w_grid, z_draws = model

    # Get final employment state directly
    key = jax.random.PRNGKey(42)
    w_bar = get_reservation_wage(model)

    # Initialize arrays
    init_key, subkey = jax.random.split(key)
    initial_wages = jnp.exp(jax.random.normal(subkey, (n_agents,)) * ν)
    initial_status_vec = jnp.zeros(n_agents, dtype=jnp.int32)

    # Generate n_agents random keys
    agent_keys = jax.random.split(init_key, n_agents)

    # Simulate each agent forward T steps (vectorized)
    final_status, _ = sim_agents_vmap(
        agent_keys, initial_status_vec, initial_wages, model, w_bar, t_snapshot
    )

    # Calculate unemployment rate
    unemployment_rate = 1 - jnp.mean(final_status)

    fig, ax = plt.subplots(figsize=(8, 5))

    # Plot histogram as density (bars sum to 1)
    weights = jnp.ones_like(final_status) / len(final_status)
    ax.hist(final_status, bins=[-0.5, 0.5, 1.5],
            alpha=0.7, color='blue', edgecolor='black',
            density=True, weights=weights)

    ax.set_xlabel('employment status (0=unemployed, 1=employed)')
    ax.set_ylabel('density')
    ax.set_title(f'Cross-sectional distribution at t={t_snapshot}, ' +
                 f'unemployment rate = {unemployment_rate:.3f}')
    ax.set_xticks([0, 1])

    plt.tight_layout()
    plt.show()
```

Let's plot the cross-sectional distribution:

```
plot_cross_sectional_unemployment(model)
```



53.6 Exercises

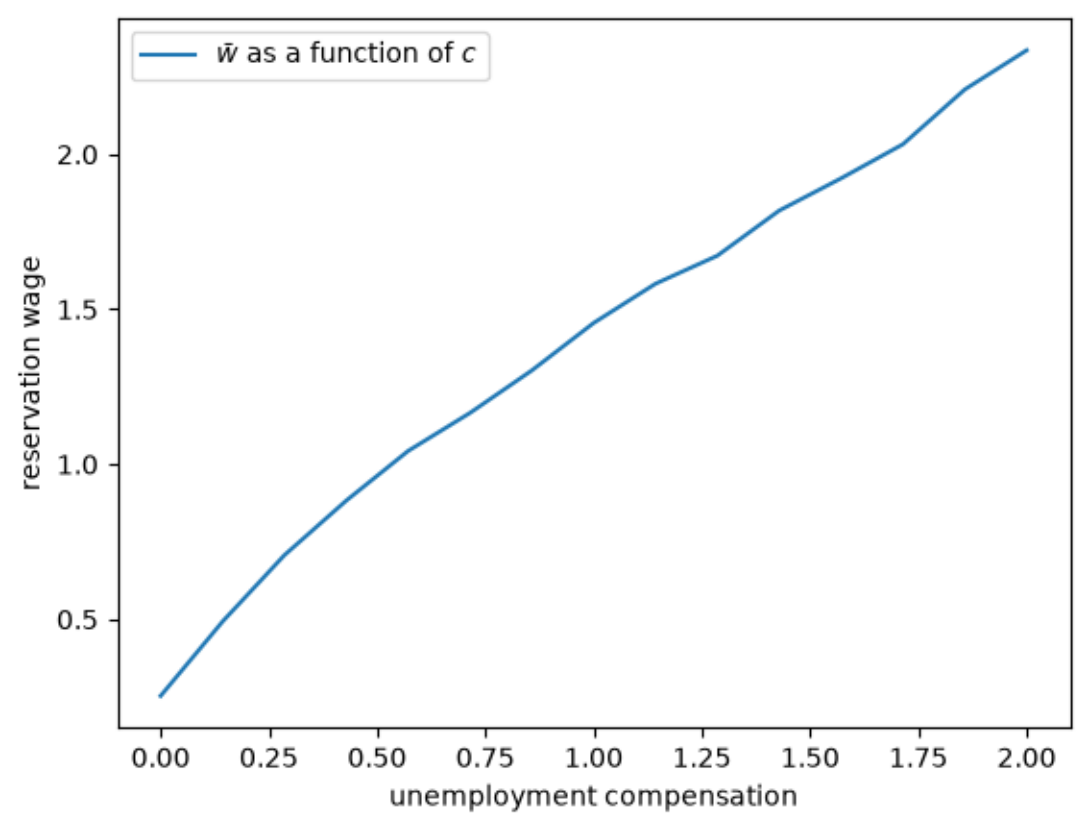
Exercise 53.6.1

Use the code above to explore what happens to the reservation wage when c changes.

Solution

Here is one solution

```
def compute_res_wage_given_c(c):
    model = create_mccall_model(c=c)
    w_bar = get_reservation_wage(model)
    return w_bar

c_vals = jnp.linspace(0.0, 2.0, 15)
w_bar_vals = jax.vmap(compute_res_wage_given_c)(c_vals)

fig, ax = plt.subplots()
ax.set(xlabel='unemployment compensation', ylabel='reservation wage')
ax.plot(c_vals, w_bar_vals, label=r'$\bar w$ as a function of $c$')
ax.legend()
plt.show()
```



As unemployment compensation increases, the reservation wage also increases.

This makes economic sense: when the value of being unemployed rises (through higher c), workers become more selective about which job offers to accept.

Exercise 53.6.2

Create a plot that shows how the reservation wage changes with the risk aversion parameter γ.

Use `γ_vals = jnp.linspace(1.2, 2.5, 15)` and keep all other parameters at their default values.

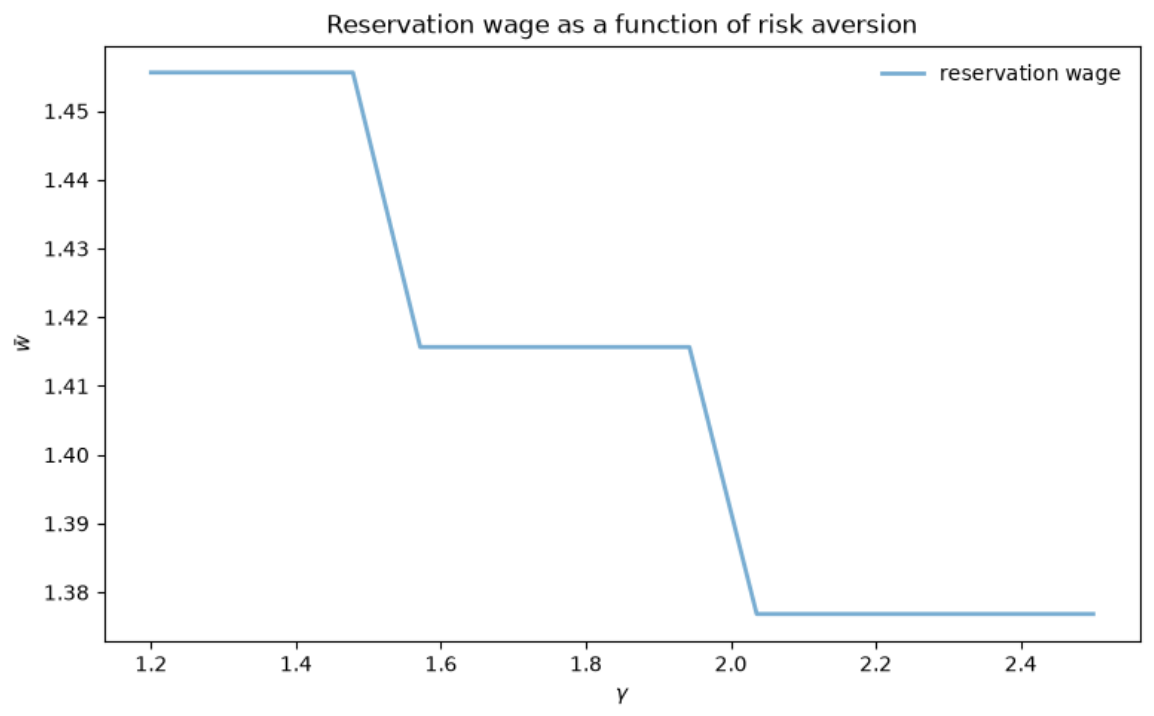
How do you expect the reservation wage to vary with γ? Why?

Solution

We compute the reservation wage for different values of the risk aversion parameter:

```
γ_vals = jnp.linspace(1.2, 2.5, 15)
w_bar_vec = jnp.empty_like(γ_vals)

for i, γ in enumerate(γ_vals):
    model = create_mccall_model(γ=γ)
    w_bar = get_reservation_wage(model)
    w_bar_vec = w_bar_vec.at[i].set(w_bar)

fig, ax = plt.subplots(figsize=(9, 5.2))
```

```
ax.plot(γ_vals, w_bar_vec, linewidth=2, alpha=0.6,
        label='reservation wage')
ax.legend(frameon=False)
ax.set_xlabel(r'$\gamma$')
ax.set_ylabel(r'$\bar{w}$')
ax.set_title('Reservation wage as a function of risk aversion')
plt.show()
```



As risk aversion (γ) increases, the reservation wage decreases.

This occurs because more risk-averse workers place higher value on the security of employment relative to the uncertainty of continued search.

With higher γ, the utility cost of unemployment (foregone consumption) becomes more severe, making workers more willing to accept lower wages rather than continue searching.

CHAPTER

FIFTYFOUR

JOB SEARCH V: PERSISTENT AND TRANSITORY WAGE SHOCKS

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

Contents

- *Job Search V: Persistent and Transitory Wage Shocks*
  - *Overview*
  - *The model*
  - *Implementation*
  - *Unemployment duration*
  - *Exercises*

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install quantecon jax
```

54.1 Overview

In this lecture we extend the *McCall job search model* by decomposing wage offers into **persistent** and **transitory** components.

In the *baseline model*, wage offers are IID over time, which is unrealistic.

In *Job Search III*, we introduced correlated wage draws using a Markov chain, but we also added job separation.

Here we take a different approach: we model wage dynamics through an AR(1) process for the persistent component plus a transitory shock, while returning to the assumption that jobs are permanent (as in the *baseline model*).

This persistent-transitory decomposition is:

- More realistic for modeling actual wage processes
- Commonly used in labor economics (see, e.g., [MaCurdy, 1982], [Meghir and Pistaferri, 2004])
- Simple enough to analyze while capturing key features of wage dynamics

By keeping jobs permanent, we can focus on understanding how persistent and transitory wage shocks affect search behavior and reservation wages.

We will solve the model using fitted value function iteration with linear interpolation, as introduced in *Job Search IV*.

We will use the following imports:

```
import matplotlib.pyplot as plt
import jax
import jax.numpy as jnp
import jax.random
import quantecon as qe
from typing import NamedTuple
```

54.2 The model

Wages at each point in time are given by

$$W_t = \exp(Z_t) + Y_t$$

where

$$Y_t \sim \exp(\mu + s\zeta_t) \quad \text{and} \quad Z_{t+1} = d + \rho Z_t + \sigma \epsilon_{t+1}$$

Here $\{\zeta_t\}$ and $\{\epsilon_t\}$ are both IID and standard normal.

Here $\{Y_t\}$ is a transitory component and $\{Z_t\}$ is persistent.

As before, the worker can either

1. accept an offer and work permanently at that wage, or
2. take unemployment compensation c and wait till next period.

The value function satisfies the Bellman equation

$$v^*(w, z) = \max \left\{ \frac{u(w)}{1-\beta}, u(c) + \beta \, \mathbb{E}_z v^*(w', z') \right\}$$

In this expression, u is a utility function and $\mathbb{E}_z$ is the expectation of next period variables given current z.

The variable z enters as a state in the Bellman equation because its current value helps predict future wages.

54.2.1 A simplification

There is a way that we can reduce dimensionality in this problem, which greatly accelerates computation.

To start, let f^* be the continuation value function, defined by

$$f^*(z) := u(c) + \beta \, \mathbb{E}_z v^*(w', z')$$

The Bellman equation can now be written

$$v^*(w, z) = \max \left\{ \frac{u(w)}{1-\beta}, \, f^*(z) \right\}$$

Combining the last two expressions, we see that the continuation value function satisfies

$$f^*(z) = u(c) + \beta \, \mathbb{E}_z \max \left\{ \frac{u(w')}{1-\beta}, f^*(z') \right\}$$

We'll solve this functional equation for f^* by introducing the operator

$$Qf(z) = u(c) + \beta \, \mathbb{E}_z \max \left\{ \frac{u(w')}{1-\beta}, f(z') \right\}$$

By construction, f^* is a fixed point of Q, in the sense that $Qf^* = f^*$.

Under mild assumptions, it can be shown that Q is a contraction mapping over a suitable space of continuous functions on $\mathbb{R}$.

By Banach's contraction mapping theorem, this means that f^* is the unique fixed point and we can calculate it by iterating with Q from any reasonable initial condition.

Once we have f^*, we can solve the search problem by stopping when the reward for accepting exceeds the continuation value, or

$$\frac{u(w)}{1-\beta} \geq f^*(z)$$

For utility, we take $u(x) = \ln(x)$.

The reservation wage is the wage where equality holds in the last expression.

That is,

$$\bar{w}(z) := \exp(f^*(z)(1-\beta)) \tag{54.1}$$

Our main aim is to solve for the reservation rule and study its properties and implications.

54.3 Implementation

Let f be our initial guess of f^*.

When we iterate, we use the *fitted value function iteration* algorithm.

In particular, f and all subsequent iterates are stored as a vector of values on a grid.

These points are interpolated into a function as required, using piecewise linear interpolation.

The integral in the definition of Qf is calculated by Monte Carlo.

Here's a `NamedTuple` that stores the model parameters and data.

Default parameter values are embedded in the model.

```
class Model(NamedTuple):
    μ: float     # transient shock log mean
    s: float     # transient shock log variance
    d: float     # shift coefficient of persistent state
    ρ: float     # correlation coefficient of persistent state
    σ: float     # state volatility
    β: float     # discount factor
    c: float     # unemployment compensation
    z_grid: jnp.ndarray
    e_draws: jnp.ndarray

def create_job_search_model(μ=0.0, s=1.0, d=0.0, ρ=0.9, σ=0.1, β=0.98, c=5.0,
                            mc_size=1000, grid_size=100, key=jax.random.PRNGKey(1234)):
    """
    Create a Model with computed grid and draws.
    """
    # Set up grid
    z_mean = d / (1 - ρ)
    z_sd = σ / jnp.sqrt(1 - ρ**2)
    k = 3  # std devs from mean
    a, b = z_mean - k * z_sd, z_mean + k * z_sd
    z_grid = jnp.linspace(a, b, grid_size)

    # Draw and store shocks
    e_draws = jax.random.normal(key, (2, mc_size))

    return Model(μ, s, d, ρ, σ, β, c, z_grid, e_draws)
```

Next, we implement the Q operator.

```
def Q(model, f_in):
    """
    Apply the operator Q.

        * model is an instance of Model
        * f_in is an array that represents f
        * returns Qf

    """
    μ, s, d = model.μ, model.s, model.d
    ρ, σ, β, c = model.ρ, model.σ, model.β, model.c
    z_grid, e_draws = model.z_grid, model.e_draws
    M = e_draws.shape[1]

    def compute_expectation(z):
        def evaluate_shock(e):
            e1, e2 = e[0], e[1]
            z_next = d + ρ * z + σ * e1
            go_val = jnp.interp(z_next, z_grid, f_in)  # f(z')
            y_next = jnp.exp(μ + s * e2)               # y' draw
            w_next = jnp.exp(z_next) + y_next          # w' draw
            stop_val = jnp.log(w_next) / (1 - β)
            return jnp.maximum(stop_val, go_val)

        expectations = jax.vmap(evaluate_shock)(e_draws.T)
        return jnp.mean(expectations)
```

(continues on next page)

(continued from previous page)

```
    expectations = jax.vmap(compute_expectation)(z_grid)
    f_out = jnp.log(c) + β * expectations
    return f_out
```

Here's a function to compute an approximation to the fixed point of Q.

```
@jax.jit
def compute_fixed_point(model, tol=1e-4, max_iter=1000):
    """
    Compute an approximation to the fixed point of Q.
    """

    def cond_fun(loop_state):
        f, i, error = loop_state
        return jnp.logical_and(error > tol, i < max_iter)

    def body_fun(loop_state):
        f, i, error = loop_state
        f_new = Q(model, f)
        error_new = jnp.max(jnp.abs(f_new - f))
        return f_new, i + 1, error_new

    # Initial state
    f_init = jnp.full(len(model.z_grid), jnp.log(model.c))
    init_state = (f_init, 0, tol + 1)

    # Run iteration
    f_final, iterations, final_error = jax.lax.while_loop(
        cond_fun, body_fun, init_state
    )

    return f_final
```

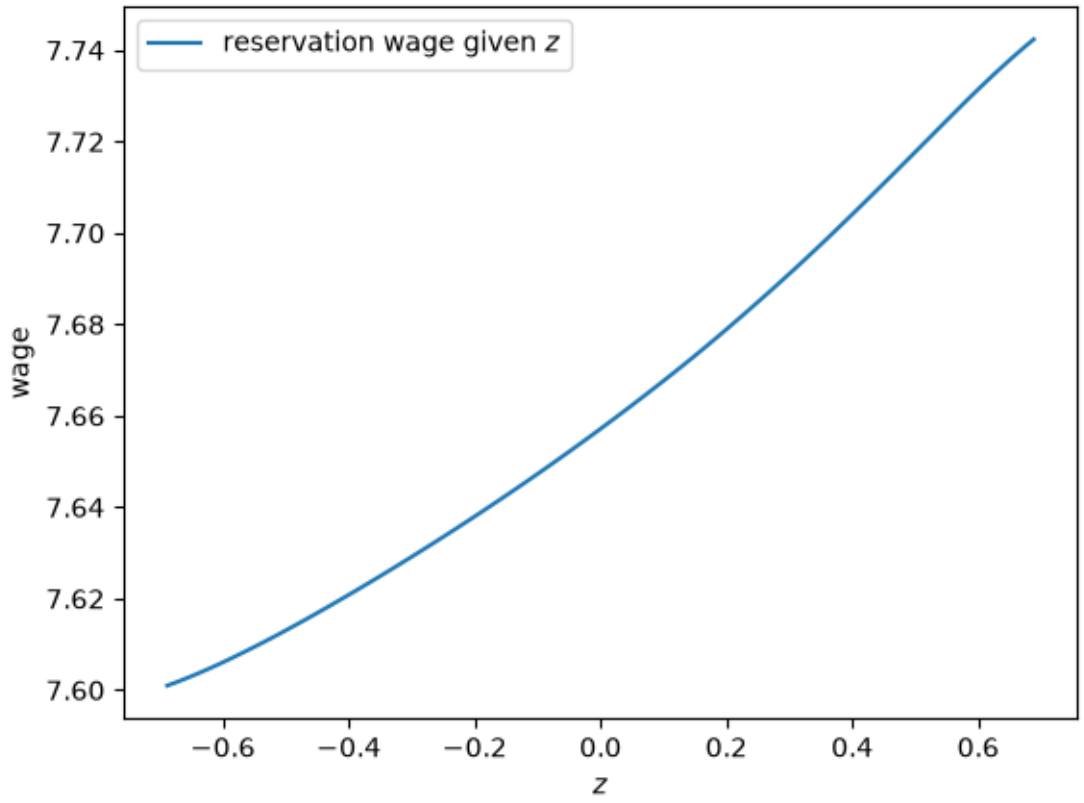
Let's try generating an instance and solving the model.

```
model = create_job_search_model()

with qe.Timer():
    f_star = compute_fixed_point(model).block_until_ready()
```

```
0.4075 seconds elapsed
```

Next, we will compute and plot the reservation wage function defined in (54.1).

```
res_wage_function = jnp.exp(f_star * (1 - model.β))

fig, ax = plt.subplots()
ax.plot(
    model.z_grid, res_wage_function, label="reservation wage given $z$"
)
ax.set(xlabel="$z$", ylabel="wage")
ax.legend()
plt.show()
```

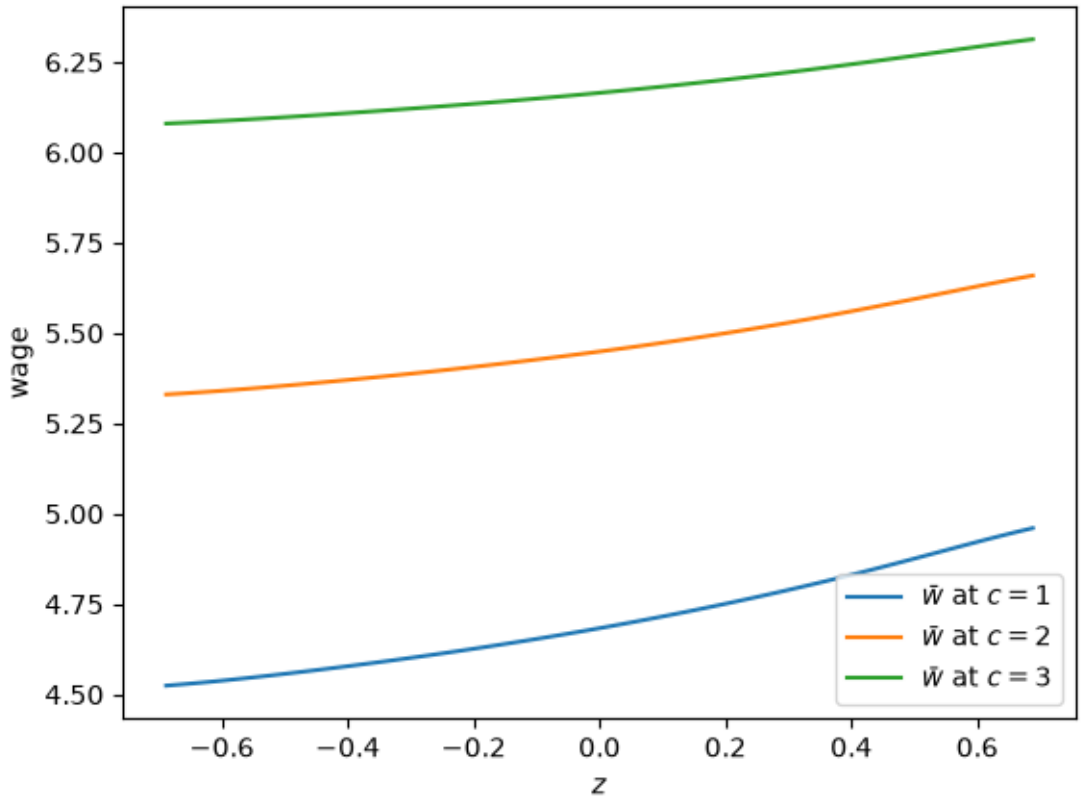


Notice that the reservation wage is increasing in the current state z.

This is because a higher state leads the agent to predict higher future wages, increasing the option value of waiting.

Let's try changing unemployment compensation and looking at its impact on the reservation wage:

```
c_vals = 1, 2, 3

fig, ax = plt.subplots()

for c in c_vals:
    model = create_job_search_model(c=c)
    f_star = compute_fixed_point(model)
    res_wage_function = jnp.exp(f_star * (1 - model.β))
    ax.plot(model.z_grid, res_wage_function,
                label=rf"$\bar w$ at $c = {c}$")

ax.set(xlabel="$z$", ylabel="wage")
ax.legend()
plt.show()
```



As expected, higher unemployment compensation shifts the reservation wage up at all state values.

54.4 Unemployment duration

Next, we study how mean unemployment duration varies with unemployment compensation.

For simplicity, we'll fix the initial state at $Z_0 = 0$.

```
@jax.jit
def draw_duration(key, μ, s, d, ρ, σ, β, z_grid, f_star, t_max=10_000):
    """
    Draw unemployment duration for a single simulation.

    """
    def f_star_function(z):
        return jnp.interp(z, z_grid, f_star)

    def cond_fun(loop_state):
        z, t, unemployed, key = loop_state
        return jnp.logical_and(unemployed, t < t_max)

    def body_fun(loop_state):
        z, t, unemployed, key = loop_state
        key1, key2, key = jax.random.split(key, 3)

        # Draw current wage
        y = jnp.exp(μ + s * jax.random.normal(key1))
```

(continues on next page)

(continued from previous page)

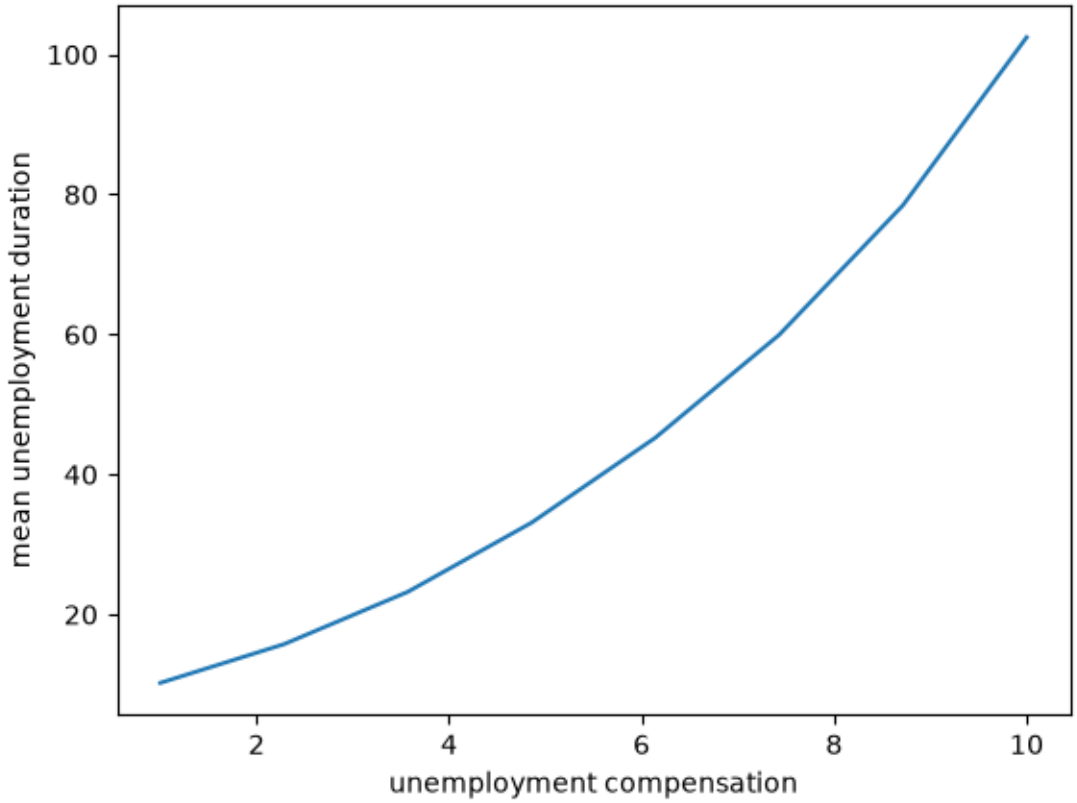
```
        w = jnp.exp(z) + y
        res_wage = jnp.exp(f_star_function(z) * (1 - β))

        # Check if optimal to stop
        accept = w >= res_wage
        τ = jnp.where(accept, t, t_max)

        # Update state if not accepting
        z_new = jnp.where(accept, z,
                          ρ * z + d + σ * jax.random.normal(key2))
        t_new = t + 1
        unemployed_new = jnp.logical_not(accept)

        return z_new, t_new, unemployed_new, key

    # Initial loop_state: (z, t, unemployed, key)
    init_state = (0.0, 0, True, key)
    z_final, t_final, unemployed_final, _ = jax.lax.while_loop(
        cond_fun, body_fun, init_state)

    # Return final time if job found, otherwise t_max
    return jnp.where(unemployed_final, t_max, t_final)


def compute_unemployment_duration(
        model, key=jax.random.PRNGKey(1234), num_reps=100_000
    ):
    """
    Compute expected unemployment duration.

    """
    f_star = compute_fixed_point(model)
    μ, s, d = model.μ, model.s, model.d
    ρ, σ, β = model.ρ, model.σ, model.β
    z_grid = model.z_grid

    # Generate keys for all simulations
    keys = jax.random.split(key, num_reps)

    # Vectorize over simulations
    τ_vals = jax.vmap(
        lambda k: draw_duration(k, μ, s, d, ρ, σ, β, z_grid, f_star)
    )(keys)

    return jnp.mean(τ_vals)
```

Let's test this out with some possible values for unemployment compensation.

```
c_vals = jnp.linspace(1.0, 10.0, 8)
durations = []
for i, c in enumerate(c_vals):
    model = create_job_search_model(c=c)
    τ = compute_unemployment_duration(model, num_reps=10_000)
    durations.append(τ)
durations = jnp.array(durations)
```

Here is a plot of the results.

```
fig, ax = plt.subplots()
ax.plot(c_vals, durations)
ax.set_xlabel("unemployment compensation")
ax.set_ylabel("mean unemployment duration")
plt.show()
```



Not surprisingly, unemployment duration increases when unemployment compensation is higher.

This is because the value of waiting increases with unemployment compensation.

54.5 Exercises

Exercise 54.5.1

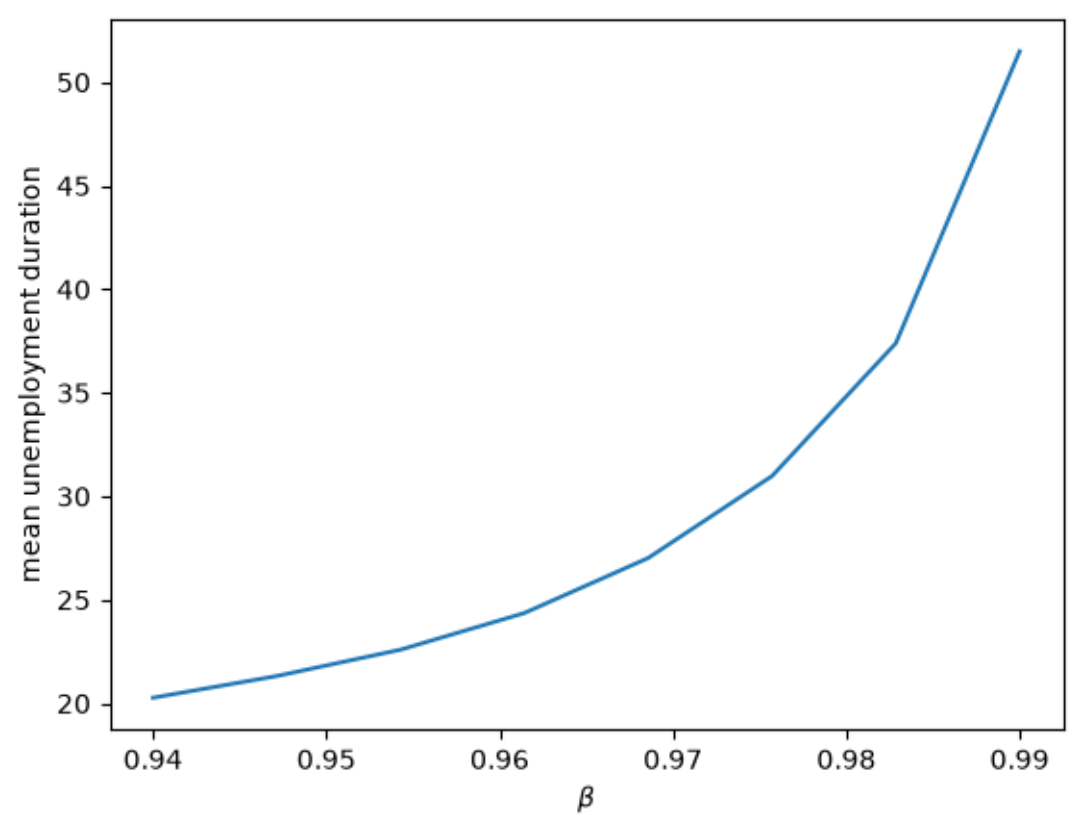
Investigate how mean unemployment duration varies with the discount factor β.

- What is your prior expectation?
- Do your results match up?

Solution

Here is one solution:

```
beta_vals = jnp.linspace(0.94, 0.99, 8)
durations = []
for i, β in enumerate(beta_vals):
    model = create_job_search_model(β=β)
    τ = compute_unemployment_duration(model, num_reps=10_000)
    durations.append(τ)
durations = jnp.array(durations)
fig, ax = plt.subplots()
ax.plot(beta_vals, durations)
ax.set_xlabel(r"$\beta$")
ax.set_ylabel("mean unemployment duration")
plt.show()
```



The figure shows that more patient individuals tend to wait longer before accepting an offer.

CHAPTER

FIFTYFIVE

JOB SEARCH VI: MODELING CAREER CHOICE

Contents

- *Job Search VI: Modeling Career Choice*
  - *Overview*
  - *Model*
  - *Implementation*
  - *Exercises*

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install quantecon
```

55.1 Overview

Next, we study a computational problem concerning career and job choices.

The model is originally due to Derek Neal [Neal, 1999].

This exposition draws on the presentation in [Ljungqvist and Sargent, 2018], section 6.5.

We begin with some imports:

```
from typing import NamedTuple
```

(continues on next page)

(continued from previous page)

```
import matplotlib.pyplot as plt
from matplotlib import cm
from mpl_toolkits.mplot3d.axes3d import Axes3D
import jax
import jax.numpy as jnp
import jax.random as jr
from quantecon.distributions import BetaBinomial
```

55.1.1 Model Features

- Career and job within career both chosen to maximize expected discounted wage flow.
- Infinite horizon dynamic programming with two state variables.

55.2 Model

In what follows we distinguish between a career and a job, where

- a **career** is understood to be a general field encompassing many possible jobs, and
- a **job** is understood to be a position with a particular firm

For workers, wages can be decomposed into the contribution of job and career

- $w_t = \theta_t + \epsilon_t$, where
  - θ_t is the contribution of career at time t
  - ϵ_t is the contribution of the job at time t

At the start of time t, a worker has the following options

- retain a current (career, job) pair (θ_t, ϵ_t) — referred to hereafter as "stay put"
- retain a current career θ_t but redraw a job ϵ_t — referred to hereafter as "new job"
- redraw both a career θ_t and a job ϵ_t — referred to hereafter as "new life"

Draws of θ and ϵ are independent of each other and past values, with

- $\theta_t \sim F$
- $\epsilon_t \sim G$

Notice that the worker does not have the option to retain a job but redraw a career — starting a new career always requires starting a new job.

A young worker aims to maximize the expected sum of discounted wages

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t w_t \tag{55.1}$$

subject to the choice restrictions specified above.

Let $v(\theta, \epsilon)$ denote the value function, which is the maximum of (55.1) overall feasible (career, job) policies, given the initial state (θ, ϵ).

The value function obeys

$$v(\theta, \epsilon) = \max\{I, II, III\}$$

where

$$I = \theta + \epsilon + \beta v(\theta, \epsilon)$$
$$II = \theta + \int \epsilon' G(d\epsilon') + \beta \int v(\theta, \epsilon') G(d\epsilon')$$
$$III = \int \theta' F(d\theta') + \int \epsilon' G(d\epsilon') + \beta \int \int v(\theta', \epsilon') G(d\epsilon') F(d\theta')$$

Evidently I, II and III correspond to "stay put", "new job" and "new life", respectively.

55.2.1 Parameterization

As in [Ljungqvist and Sargent, 2018], section 6.5, we will focus on a discrete version of the model, parameterized as follows:

- both θ and ϵ take values in the set `jnp.linspace(0, B, grid_size)` — an even grid of points between 0 and B inclusive
- `grid_size = 50`
- `B = 5`
- `β = 0.95`

The distributions F and G are discrete distributions generating draws from the grid points `jnp.linspace(0, B, grid_size)`.

A very useful family of discrete distributions is the Beta-binomial family, with probability mass function

$$p(k \mid n, a, b) = \binom{n}{k} \frac{B(k + a, n - k + b)}{B(a, b)}, \qquad k = 0, \ldots, n$$

Interpretation:

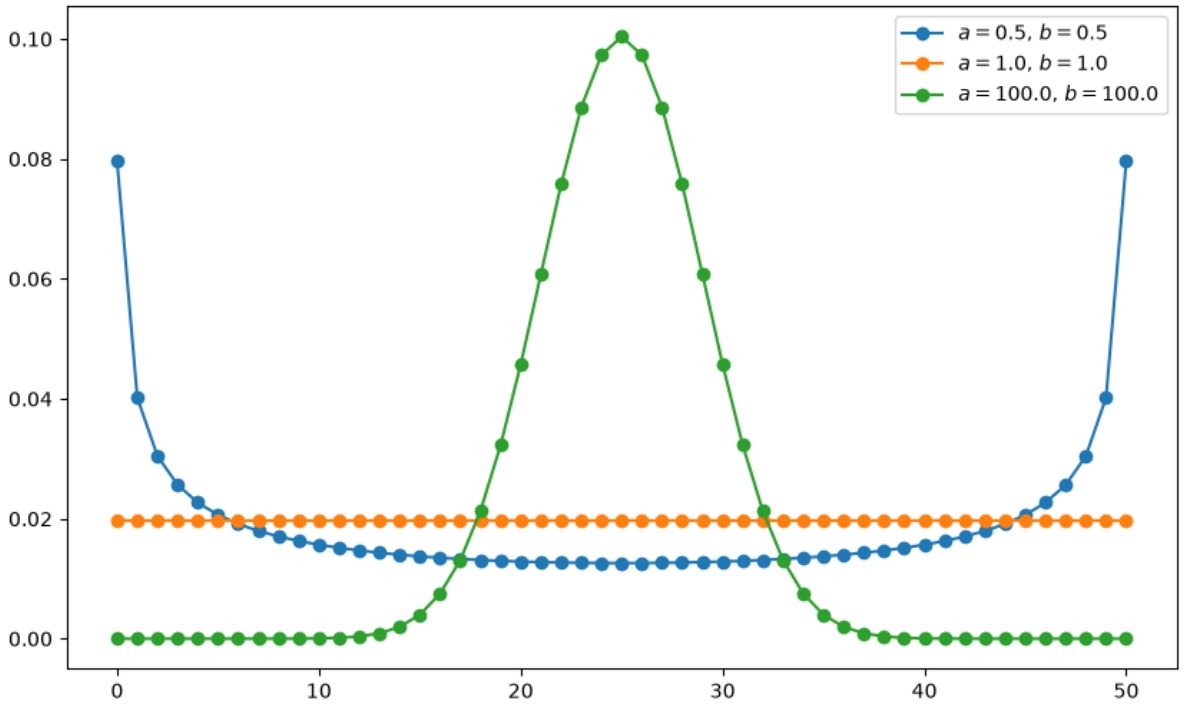
- draw q from a Beta distribution with shape parameters (a, b)
- run n independent binary trials, each with success probability q
- $p(k \mid n, a, b)$ is the probability of k successes in these n trials

Nice properties:

- very flexible class of distributions, including uniform, symmetric unimodal, etc.
- only three parameters

Here's a figure showing the effect on the pmf of different shape parameters when $n = 50$.

```
n = 50
a_vals = [0.5, 1, 100]
b_vals = [0.5, 1, 100]

fig, ax = plt.subplots(figsize=(10, 6))
for a, b in zip(a_vals, b_vals):
    ab_label = f'$a = {a:.1f}$, $b = {b:.1f}$'
    ax.plot(range(n + 1), BetaBinomial(n, a, b).pdf(), '-o', label=ab_label)
ax.legend()
plt.show()
```



55.3 Implementation

We store the model primitives in a `NamedTuple`, built by a factory function.

```
class CareerWorkerProblem(NamedTuple):
    β: float                    # Discount factor
    θ: jnp.ndarray              # Set of θ values (career)
    ϵ: jnp.ndarray              # Set of ϵ values (job)
    F_probs: jnp.ndarray        # Distribution over new career draws
    G_probs: jnp.ndarray        # Distribution over new job draws
    F_mean: float               # Mean of F
    G_mean: float               # Mean of G


def create_career_worker_problem(B=5.0,          # Upper bound
                                 β=0.95,         # Discount factor
                                 grid_size=50,   # Grid size
                                 F_a=1,
                                 F_b=1,
                                 G_a=1,
                                 G_b=1):
    "Create an instance of the career choice model."
    θ = jnp.linspace(0, B, grid_size)
    ϵ = jnp.linspace(0, B, grid_size)

    F_probs = jnp.array(BetaBinomial(grid_size - 1, F_a, F_b).pdf())
    G_probs = jnp.array(BetaBinomial(grid_size - 1, G_a, G_b).pdf())

    return CareerWorkerProblem(β=β, θ=θ, ϵ=ϵ,
                               F_probs=F_probs, G_probs=G_probs,
```

(continues on next page)

(continued from previous page)

```
                                F_mean=θ @ F_probs, G_mean=ϵ @ G_probs)
```

The Bellman operator is $Tv(\theta, \epsilon) = \max\{I, II, III\}$, where I, II and III are as given in (55.2).

We start by writing those three values for a **single** state (θ_i, ϵ_j), so that the code sits close to the equation.

```
def _B(v, cw, i, j):
    """
    The values of the three options available at state (θ_i, ϵ_j), in the
    order they appear in the Bellman equation.
    """
    stay_put = cw.θ[i] + cw.ϵ[j] + cw.β * v[i, j]                          # I
    new_job = cw.θ[i] + cw.G_mean + cw.β * v[i, :] @ cw.G_probs             # II
    new_life = cw.G_mean + cw.F_mean + cw.β * cw.F_probs @ v @ cw.G_probs # III
    return jnp.array([stay_put, new_job, new_life])
```

Now we evaluate `_B` at every state.

Rather than write two nested loops over i and j, we apply `jax.vmap` twice.

In `in_axes`, a `0` marks the argument being mapped over, while `None` holds an argument fixed.

```
# The argument order of _B is  (v,    cw,   i,    j)
_B_j  = jax.vmap(_B,   in_axes=(None, None, None, 0))   # over j
_B_ij = jax.vmap(_B_j, in_axes=(None, None, 0,    None))  # then over i


@jax.jit
def B(v, cw):
    "Value of each option at each state; shape (grid_size, grid_size, 3)."
    n = len(cw.θ)
    return _B_ij(v, cw, jnp.arange(n), jnp.arange(n))
```

The Bellman operator and the greedy policy are now the maximum and the maximizer of the same array.

```
@jax.jit
def T(v, cw):
    "The Bellman operator."
    return jnp.max(B(v, cw), axis=-1)


@jax.jit
def get_greedy(v, cw):
    "The v-greedy policy, coded as 1 = stay put, 2 = new job, 3 = new life."
    return jnp.argmax(B(v, cw), axis=-1) + 1
```

Lastly, `solve_model` iterates the Bellman operator to find the fixed point.

We use `jax.lax.while_loop` so that the whole iteration compiles into a single operation, and bound the number of steps so the loop always terminates.

```
@jax.jit
def solve_model(cw, tol=1e-4, max_iter=1_000):
    """
    Solve the model by value function iteration.

    Returns the value function, the number of iterations taken and the final
    error, so that the caller can check convergence.
```

(continues on next page)

(continued from previous page)

```
    """
    def condition(loop_state):
        i, v, error = loop_state
        return (error > tol) & (i < max_iter)

    def update(loop_state):
        i, v, error = loop_state
        v_new = T(v, cw)
        return i + 1, v_new, jnp.max(jnp.abs(v_new - v))

    n = len(cw.θ)
    v_init = jnp.full((n, n), 100.0)
    i, v, error = jax.lax.while_loop(condition, update, (0, v_init, tol + 1))
    return v, i, error
```

Note

The grid here is small, and this model would also run perfectly well in NumPy.

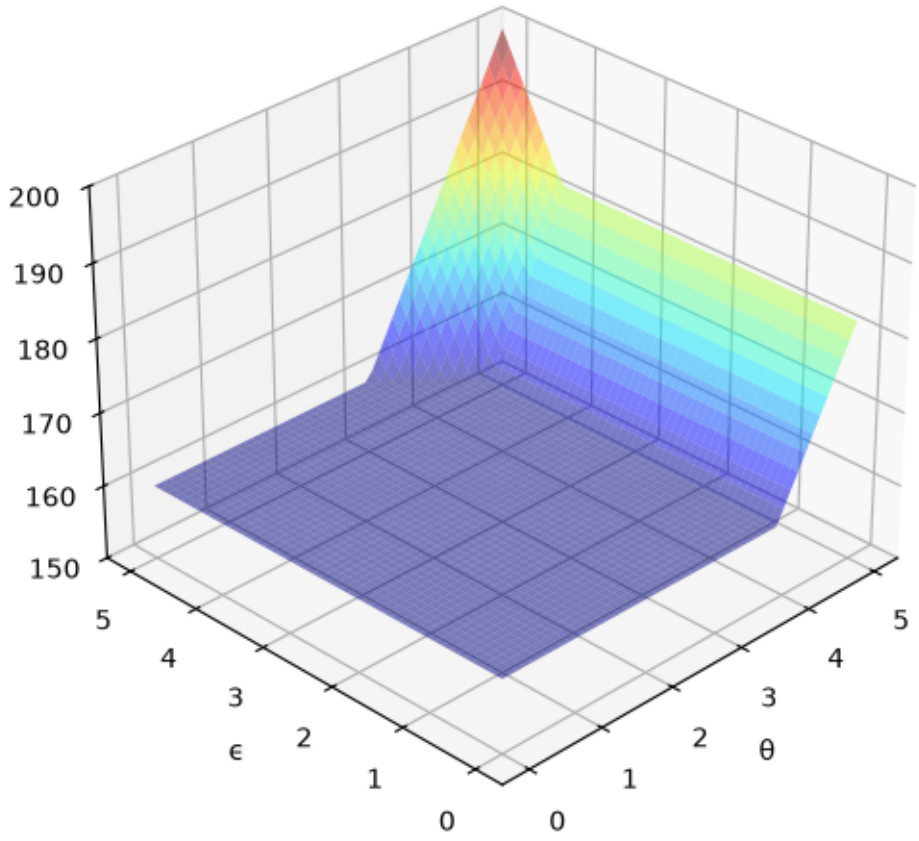
We use JAX because the code is almost as readable as the NumPy equivalent while scaling far better — to finer grids, or to richer versions of the model with more state variables, where the same code will make full use of a GPU.

The gain is already visible in Exercise 55.4.2, where we simulate 25,000 independent careers at once.

Here's the solution to the model – an approximate value function

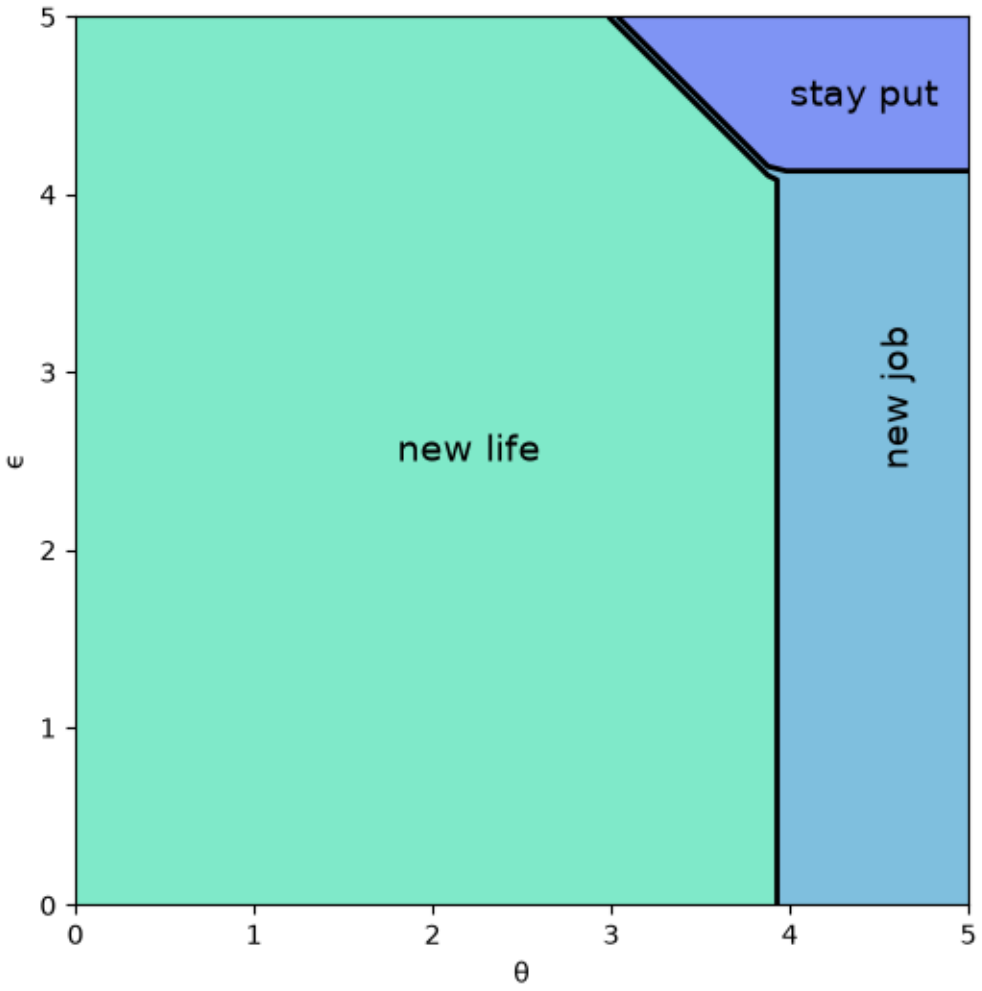
```
cw = create_career_worker_problem()
v_star, num_iter, error = solve_model(cw)
greedy_star = get_greedy(v_star, cw)

print(f"Converged in {num_iter} iterations with error {error:.2e}.")

fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111, projection='3d')
tg, eg = jnp.meshgrid(cw.θ, cw.ϵ)
ax.plot_surface(tg,
                eg,
                v_star.T,
                cmap=cm.jet,
                alpha=0.5,
                linewidth=0.25)
ax.set(xlabel='θ', ylabel='ϵ', zlim=(150, 200))
ax.view_init(ax.elev, 225)
plt.show()
```

```
Converged in 213 iterations with error 9.16e-05.
```



And here is the optimal policy

```
fig, ax = plt.subplots(figsize=(6, 6))
tg, eg = jnp.meshgrid(cw.θ, cw.ϵ)
lvls = (0.5, 1.5, 2.5, 3.5)
ax.contourf(tg, eg, greedy_star.T, levels=lvls, cmap=cm.winter, alpha=0.5)
ax.contour(tg, eg, greedy_star.T, colors='k', levels=lvls, linewidths=2)
ax.set(xlabel='θ', ylabel='ϵ')
ax.text(1.8, 2.5, 'new life', fontsize=14)
ax.text(4.5, 2.5, 'new job', fontsize=14, rotation='vertical')
ax.text(4.0, 4.5, 'stay put', fontsize=14)
plt.show()
```



Interpretation:

- If both job and career are poor or mediocre, the worker will experiment with a new job and new career.
- If career is sufficiently good, the worker will hold it and experiment with new jobs until a sufficiently good one is found.
- If both job and career are good, the worker will stay put.

Notice that the worker will always hold on to a sufficiently good career, but not necessarily hold on to even the best paying job.

The reason is that high lifetime wages require both a good job and a good career, but the worker cannot change careers without changing jobs.

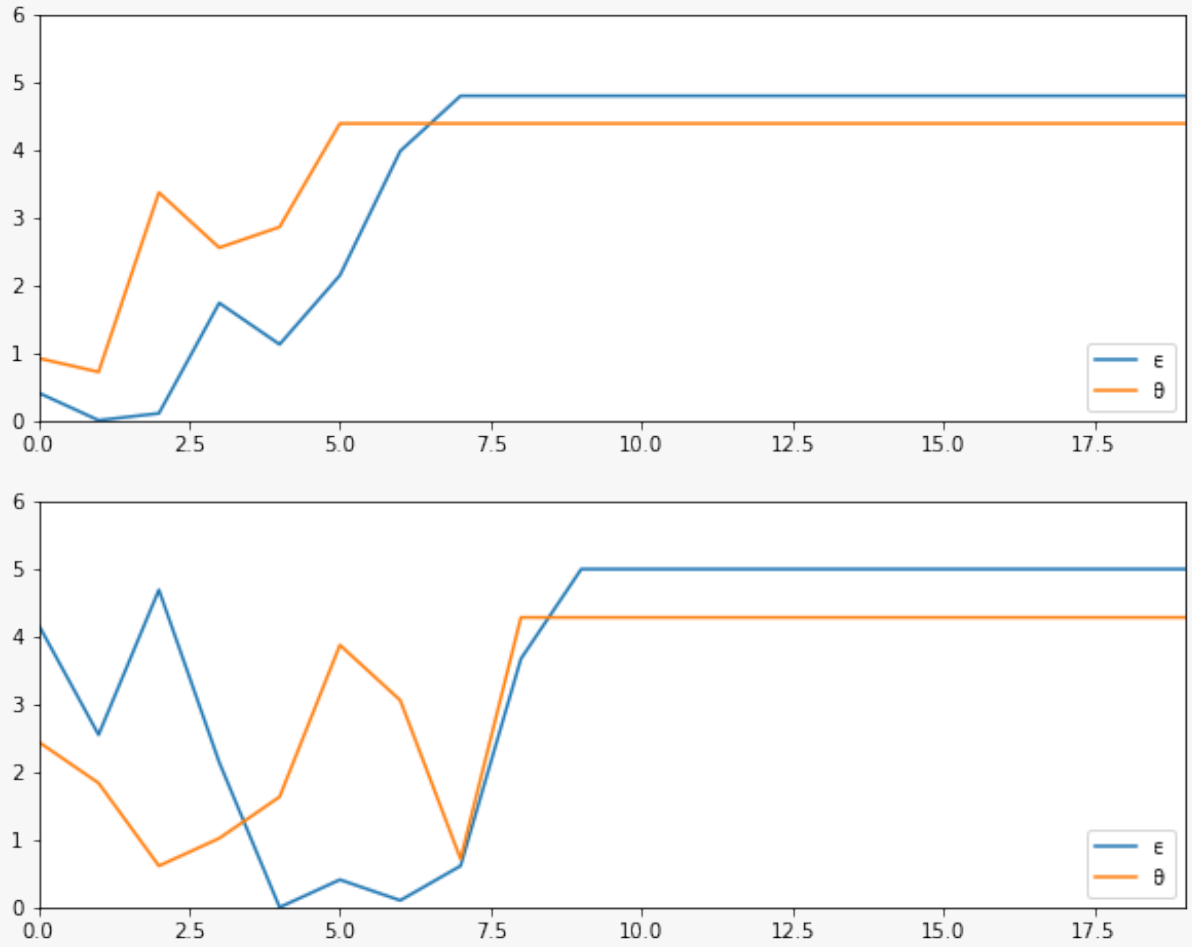
- Sometimes a good job must be sacrificed in order to change to a better career.

55.4 Exercises

Exercise 55.4.1

Using the default parameterization in the function `create_career_worker_problem`, generate and plot typical sample paths for θ and ϵ when the worker follows the optimal policy.

In particular, modulo randomness, reproduce the following figure (where the horizontal axis represents time)



Hint

To draw from F and G, invert their cdfs with `jnp.searchsorted`.

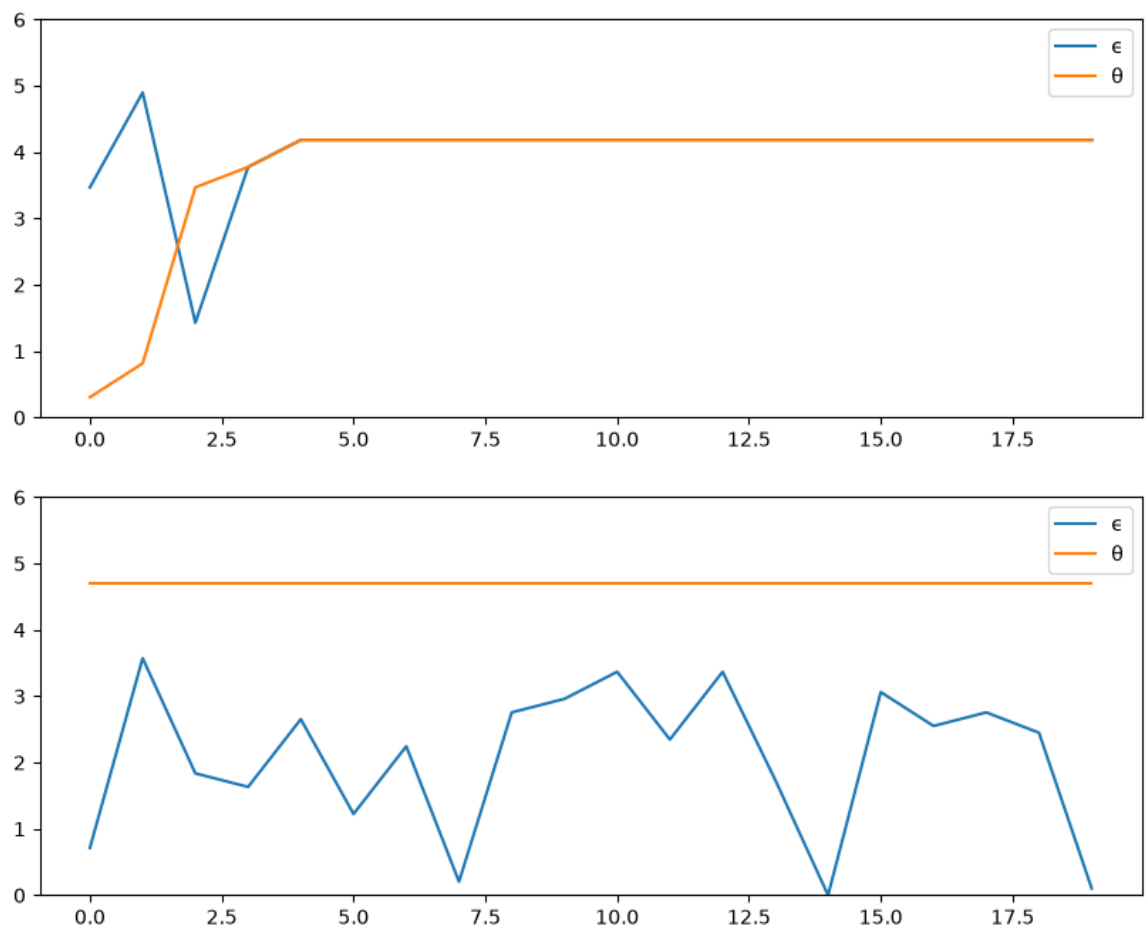
Solution

Simulate job/career paths.

In reading the code, recall that `greedy_star[i, j]` = policy at (θ_i, ϵ_j) = either 1, 2 or 3; meaning 'stay put', 'new job' and 'new life'.

```
def draw(key, cdf):
```

```
    "Draw an index from the distribution with the given cdf."
    return jnp.searchsorted(cdf, jr.uniform(key), side="right")


def simulate_path(cw, greedy_star, key, t=20):
    "Simulate a career/job path of length t under the greedy policy."
    F_cdf, G_cdf = jnp.cumsum(cw.F_probs), jnp.cumsum(cw.G_probs)

    def update(state, key):
        i, j = state
        action = greedy_star[i, j]
        key_F, key_G = jr.split(key)
        # Career changes only under 'new life'; the job changes unless we stay put
        i_new = jnp.where(action == 3, draw(key_F, F_cdf), i)
        j_new = jnp.where(action == 1, j, draw(key_G, G_cdf))
        return (i_new, j_new), (i_new, j_new)

    _, (i_path, j_path) = jax.lax.scan(update, (0, 0), jr.split(key, t))
    return cw.θ[i_path], cw.ϵ[j_path]


cw = create_career_worker_problem()
v_star, _, _ = solve_model(cw)
greedy_star = get_greedy(v_star, cw)

key = jr.key(42)
fig, axes = plt.subplots(2, 1, figsize=(10, 8))

for ax in axes:
    key, subkey = jr.split(key)
    θ_path, ϵ_path = simulate_path(cw, greedy_star, subkey)
    ax.plot(ϵ_path, label='ϵ')
    ax.plot(θ_path, label='θ')
    ax.set_ylim(0, 6)
    ax.legend()

plt.show()
```



Exercise 55.4.2

Let's now consider how long it takes for the worker to settle down to a permanent job, given a starting point of $(\theta, \epsilon) = (0, 0)$.

In other words, we want to study the distribution of the random variable

$$T^* := \text{the first point in time from which the worker's job no longer changes}$$

Evidently, the worker's job becomes permanent if and only if (θ_t, ϵ_t) enters the "stay put" region of (θ, ϵ) space.

Letting S denote this region, T^* can be expressed as the first passage time to S under the optimal policy:

$$T^* := \inf\{t \geq 0 \,|\, (\theta_t, \epsilon_t) \in S\}$$

Collect 25,000 draws of this random variable and compute the median (which should be about 7).

Repeat the exercise with $\beta = 0.99$ and interpret the change.

Solution

The median for the original parameterization can be computed as follows.

Each simulation is an independent sequential search, so we write one with `jax.lax.while_loop` and then run 25,000 of them at once with `jax.vmap`.

```
def passage_time(cw, greedy_star, key, max_t=1_000):
    "Time until the worker first chooses to stay put."
    F_cdf, G_cdf = jnp.cumsum(cw.F_probs), jnp.cumsum(cw.G_probs)

    def condition(state):
        i, j, t, key = state
        return (greedy_star[i, j] != 1) & (t < max_t)

    def update(state):
        i, j, t, key = state
        action = greedy_star[i, j]
        key, key_F, key_G = jr.split(key, 3)
        i_new = jnp.where(action == 3, draw(key_F, F_cdf), i)
        j_new = jnp.where(action == 1, j, draw(key_G, G_cdf))
        return i_new, j_new, t + 1, key

    _, _, t, _ = jax.lax.while_loop(condition, update, (0, 0, 0, key))
    return t


@jax.jit
def median_passage_time(cw, greedy_star, key, M=25_000):
    "Median time to settle down, over M independent simulations."
    keys = jr.split(key, M)
    times = jax.vmap(passage_time, in_axes=(None, None, 0))(cw, greedy_star, keys)
    return jnp.median(times)


median_passage_time(cw, greedy_star, jr.key(42))
```

```
Array(7., dtype=float32)
```

To compute the median with $\beta = 0.99$ instead of the default value $\beta = 0.95$, we create a new instance and solve it again.

```
cw_patient = create_career_worker_problem(β=0.99)
v_patient, _, _ = solve_model(cw_patient)
greedy_patient = get_greedy(v_patient, cw_patient)

median_passage_time(cw_patient, greedy_patient, jr.key(42))
```

```
Array(14., dtype=float32)
```

The medians are subject to randomness but should be about 7 and 14 respectively.

Not surprisingly, more patient workers will wait longer to settle down to their final job.

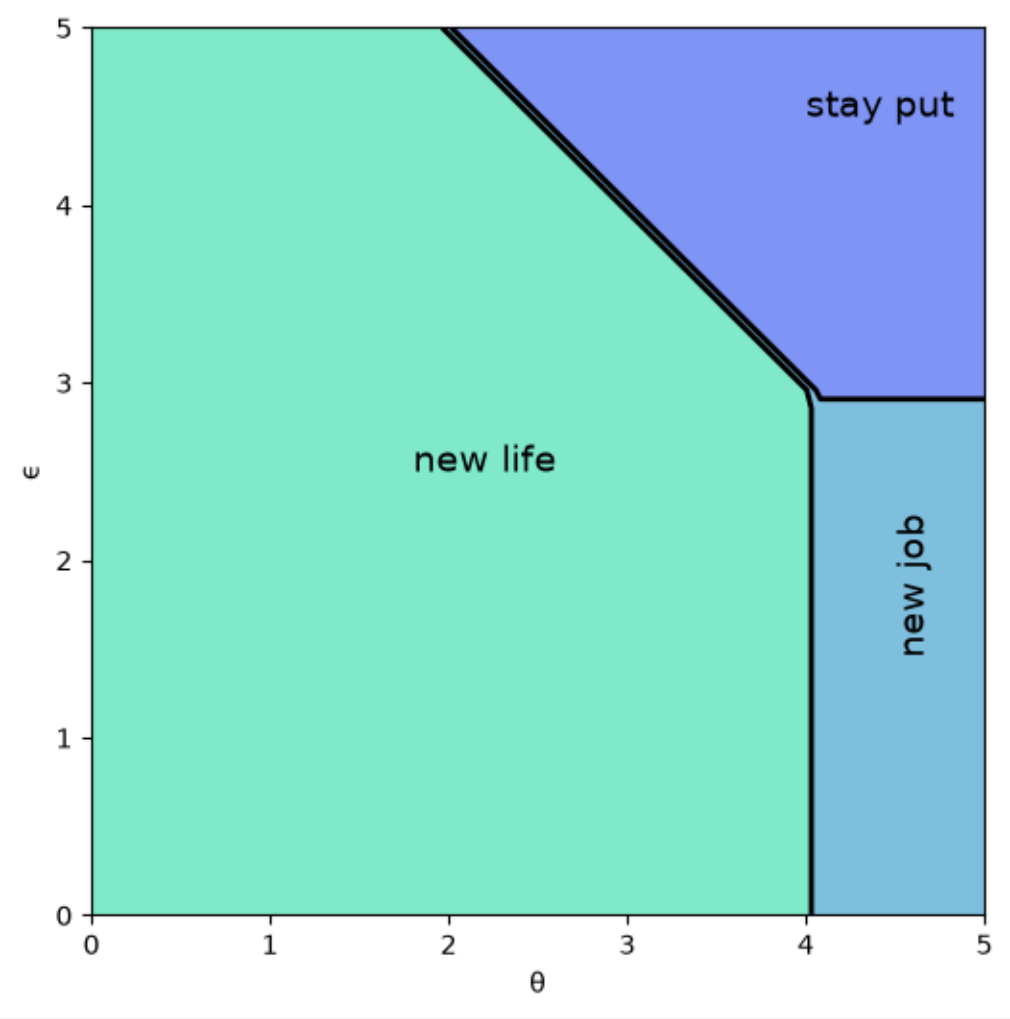
Exercise 55.4.3

Set the parameterization to `G_a = G_b = 100` and generate a new optimal policy figure – interpret.

Solution

Here is one solution

```
cw = create_career_worker_problem(G_a=100, G_b=100)
v_star, _, _ = solve_model(cw)
greedy_star = get_greedy(v_star, cw)

fig, ax = plt.subplots(figsize=(6, 6))
tg, eg = jnp.meshgrid(cw.θ, cw.ϵ)
lvls = (0.5, 1.5, 2.5, 3.5)
ax.contourf(tg, eg, greedy_star.T, levels=lvls, cmap=cm.winter, alpha=0.5)
ax.contour(tg, eg, greedy_star.T, colors='k', levels=lvls, linewidths=2)
ax.set(xlabel='θ', ylabel='ϵ')
ax.text(1.8, 2.5, 'new life', fontsize=14)
ax.text(4.5, 1.5, 'new job', fontsize=14, rotation='vertical')
ax.text(4.0, 4.5, 'stay put', fontsize=14)
plt.show()
```



In the new figure, you see that the region for which the worker stays put has grown because the distribution for ϵ has become more concentrated around the mean, making high-paying jobs less realistic.

CHAPTER

FIFTYSIX

JOB SEARCH VII: ON-THE-JOB SEARCH

Contents

- *Job Search VII: On-the-Job Search*
  - *Overview*
  - *Model*
  - *Implementation*
  - *Solving for Policies*
  - *Exercises*

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

56.1 Overview

In this section, we solve a simple on-the-job search model

- based on [Ljungqvist and Sargent, 2018], exercise 6.18, and [Jovanovic, 1979]

Let's start with some imports:

```
from typing import NamedTuple

import matplotlib.pyplot as plt
import scipy.stats as stats
import jax
import jax.numpy as jnp
import jax.random as jr
```

56.1.1 Model Features

- job-specific human capital accumulation combined with on-the-job search
- infinite-horizon dynamic programming with one state variable and two controls

56.2 Model

Let x_t denote the time-t job-specific human capital of a worker employed at a given firm and let w_t denote current wages.

Let $w_t = x_t(1 - s_t - \phi_t)$, where

- ϕ_t is investment in job-specific human capital for the current role and
- s_t is search effort, devoted to obtaining new offers from other firms.

For as long as the worker remains in the current job, evolution of $\{x_t\}$ is given by $x_{t+1} = g(x_t, \phi_t)$.

When search effort at t is s_t, the worker receives a new job offer with probability $\pi(s_t) \in [0, 1]$.

The value of the offer, measured in job-specific human capital, is u_{t+1}, where $\{u_t\}$ is IID with common distribution f.

The worker can reject the current offer and continue with existing job.

Hence $x_{t+1} = u_{t+1}$ if he/she accepts and $x_{t+1} = g(x_t, \phi_t)$ otherwise.

Let $b_{t+1} \in \{0, 1\}$ be a binary random variable, where $b_{t+1} = 1$ indicates that the worker receives an offer at the end of time t.

We can write

$$x_{t+1} = (1 - b_{t+1})g(x_t, \phi_t) + b_{t+1} \max\{g(x_t, \phi_t), u_{t+1}\} \tag{56.1}$$

Agent's objective: maximize expected discounted sum of wages via controls $\{s_t\}$ and $\{\phi_t\}$.

Taking the expectation of $v(x_{t+1})$ and using (56.1), the Bellman equation for this problem can be written as

$$v(x) = \max_{s+\phi \leq 1} \left\{ x(1 - s - \phi) + \beta(1 - \pi(s))v[g(x, \phi)] + \beta\pi(s) \int v[g(x, \phi) \vee u]f(du) \right\} \tag{56.2}$$

Here nonnegativity of s and ϕ is understood, while $a \vee b := \max\{a, b\}$.

56.2.1 Parameterization

In the implementation below, we will focus on the parameterization

$$g(x, \phi) = A(x\phi)^\alpha, \quad \pi(s) = \sqrt{s} \quad \text{and} \quad f = \text{Beta}(2, 2)$$

with default parameter values

- $A = 1.4$
- $\alpha = 0.6$
- $\beta = 0.96$

The Beta$(2, 2)$ distribution is supported on $(0, 1)$ - it has a unimodal, symmetric density peaked at 0.5.

56.2.2 Back-of-the-Envelope Calculations

Before we solve the model, let's make some quick calculations that provide intuition on what the solution should look like.

To begin, observe that the worker has two instruments to build capital and hence wages:

1. invest in capital specific to the current job via ϕ
2. search for a new job with better job-specific capital match via s

Since wages are $x(1-s-\phi)$, marginal cost of investment via either ϕ or s is identical.

Our risk-neutral worker should focus on whatever instrument has the highest expected return.

The relative expected return will depend on x.

For example, suppose first that $x = 0.05$

- If $s = 1$ and $\phi = 0$, then since $g(x, \phi) = 0$, taking expectations of (56.1) gives expected next period capital equal to $\pi(s)\mathbb{E}u = \mathbb{E}u = 0.5$.
- If $s = 0$ and $\phi = 1$, then next period capital is $g(x, \phi) = g(0.05, 1) \approx 0.23$.

Both rates of return are good, but the return from search is better.

Next, suppose that $x = 0.4$

- If $s = 1$ and $\phi = 0$, then expected next period capital is again 0.5
- If $s = 0$ and $\phi = 1$, then $g(x, \phi) = g(0.4, 1) \approx 0.8$

Return from investment via ϕ dominates expected return from search.

Combining these observations gives us two informal predictions:

1. At any given state x, the two controls ϕ and s will function primarily as substitutes — worker will focus on whichever instrument has the higher expected return.
2. For sufficiently small x, search will be preferable to investment in job-specific human capital. For larger x, the reverse will be true.

Now let's turn to implementation, and see if we can match our predictions.

56.3 Implementation

We solve the model with JAX, using a `NamedTuple` to hold the parameters and grids.

```
class JVWorker(NamedTuple):
    A: float                # Scale parameter in g
    α: float                # Curvature parameter in g
    β: float                # Discount factor
    x_grid: jnp.ndarray     # Grid of human capital values
    s_grid: jnp.ndarray     # Grid of search effort values
    ϕ_grid: jnp.ndarray     # Grid of investment values
    f_rvs: jnp.ndarray      # Draws from f, for Monte Carlo integration


def create_jv_worker(A=1.4,              # Scale parameter in g
                     α=0.6,              # Curvature parameter in g
                     β=0.96,             # Discount factor
                     a=2,                # Parameter of f
                     b=2,                # Parameter of f
```

(continues on next page)

(continued from previous page)

```
                    grid_size=50,        # Size of the state grid
                    mc_size=100,         # Number of draws from f
                    search_grid_size=15, # Size of each action grid
                    ε=1e-4,
                    seed=1234):
    """
    Create an instance of the on-the-job search model.
    """
    f_rvs = jr.beta(jr.key(seed), a, b, (mc_size,))

    # Max of grid is the max of a large quantile value for f and the
    # fixed point y = g(y, 1)
    grid_max = max(A**(1 / (1 - α)), stats.beta(a, b).ppf(1 - ε))

    x_grid = jnp.linspace(ε, grid_max, grid_size)
    s_grid = jnp.linspace(ε, 1, search_grid_size)
    ϕ_grid = jnp.linspace(ε, 1, search_grid_size)

    return JVWorker(A=A, α=α, β=β, x_grid=x_grid, s_grid=s_grid,
                    ϕ_grid=ϕ_grid, f_rvs=f_rvs)
```

Here are the transition function g and the offer probability π.

```
@jax.jit
def g(jv, x, ϕ):
    "Transition function for job-specific human capital."
    return jv.A * (x * ϕ)**jv.α


@jax.jit
def π(s):
    "Probability of receiving an offer when search effort is s."
    return jnp.sqrt(s)
```

Next we write the right-hand side of the Bellman equation (56.2), before maximization:

$$B(x, s, \phi) := x(1 - s - \phi) + \beta(1 - \pi(s))v[g(x, \phi)] + \beta\pi(s) \int v[g(x, \phi) \vee u]f(du) \tag{56.3}$$

We represent v by an array giving its values on `x_grid`, and recover a function from it by linear interpolation.

The integral is replaced by a Monte Carlo average over the draws in `f_rvs`.

The function below is written for a **single** state x and a **single** action pair (s, ϕ) — so it reads much like (56.3) itself.

```
def _B(v, jv, x, s, ϕ):
    """
    The right-hand side of the Bellman equation before maximization, for one
    state x and one action pair (s, ϕ).

    Infeasible pairs, where s + ϕ > 1, are given value -∞ so that they are
    never selected by the maximization step.
    """
    v_func = lambda z: jnp.interp(z, jv.x_grid, v)
    gxϕ = g(jv, x, ϕ)

    # Monte Carlo estimate of ∫ v[g(x, ϕ) ∨ u] f(du)
    integral = jnp.mean(v_func(jnp.maximum(gxϕ, jv.f_rvs)))
```

(continues on next page)

(continued from previous page)

```
    q = π(s) * integral + (1 - π(s)) * v_func(gxϕ)
    return jnp.where(s + ϕ <= 1, x * (1 - s - ϕ) + jv.β * q, -jnp.inf)
```

Now we evaluate `_B` at every combination of state and action.

Rather than write three nested loops, we apply `jax.vmap` three times.

Each application vectorizes over one argument, so the stack below plays the role of a triple loop — but the whole thing compiles to code that runs in parallel.

In `in_axes`, a `0` marks the argument being mapped over, while `None` holds an argument fixed.

```
# The argument order of _B is     (v,     jv,    x,     s,     ϕ)
_B_ϕ   = jax.vmap(_B,     in_axes=(None, None, None, None, 0))     # over ϕ
_B_sϕ  = jax.vmap(_B_ϕ,   in_axes=(None, None, None, 0,    None))  # then over s
_B_xsϕ = jax.vmap(_B_sϕ,  in_axes=(None, None, 0,    None, None))  # then over x
```

The result is a fully vectorized version of B.

```
@jax.jit
def B(v, jv):
    """
    Evaluate B at every (state, action) combination.

    Returns an array of shape (len(x_grid), len(s_grid), len(ϕ_grid)) where
    entry [i, j, k] holds the value of choosing (s_j, ϕ_k) in state x_i.
    """
    return _B_xsϕ(v, jv, jv.x_grid, jv.s_grid, jv.ϕ_grid)
```

With `B` in hand, the Bellman operator and the greedy policy are both one-liners — we maximize over the two action axes, taking the maximum in one case and the maximizer in the other.

```
@jax.jit
def T(v, jv):
    "The Bellman operator."
    return jnp.max(B(v, jv), axis=(1, 2))


@jax.jit
def get_greedy(v, jv):
    "Compute the v-greedy policy, returned as a pair (s_policy, ϕ_policy)."
    vals = B(v, jv)

    # Flatten the two action axes so that a single argmax picks out the best
    # pair at each state, then convert the flat index back to a (s, ϕ) pair
    n_s, n_ϕ = len(jv.s_grid), len(jv.ϕ_grid)
    best = jnp.argmax(vals.reshape(len(jv.x_grid), n_s * n_ϕ), axis=1)
    j, k = jnp.unravel_index(best, (n_s, n_ϕ))

    return jv.s_grid[j], jv.ϕ_grid[k]
```

To solve the model we iterate T to convergence.

We use `jax.lax.while_loop` so that the entire iteration compiles into a single operation, and bound the number of steps so that the loop always terminates.

```
@jax.jit
def solve_model(jv, tol=1e-4, max_iter=1_000):
    """
    Solve the model by value function iteration.

    Returns the value function, the number of iterations taken, and the final
    error, so that the caller can check convergence.
    """
    def condition(loop_state):
        i, v, error = loop_state
        return (error > tol) & (i < max_iter)

    def update(loop_state):
        i, v, error = loop_state
        v_new = T(v, jv)
        return i + 1, v_new, jnp.max(jnp.abs(v_new - v))

    v_init = jv.x_grid * 0.5
    i, v, error = jax.lax.while_loop(condition, update, (0, v_init, tol + 1))
    return v, i, error
```

Note

The grids here are small, and this model would also run perfectly well in NumPy.

We use JAX because the code is almost as readable as the NumPy equivalent, while scaling far better — to finer grids, or to richer versions of the model with additional state variables, where the same code will make full use of a GPU.

56.4 Solving for Policies

Let's generate the optimal policies and see what they look like.

```
jv = create_jv_worker()
v_star, num_iter, error = solve_model(jv)
s_star, ϕ_star = get_greedy(v_star, jv)

print(f"Converged in {num_iter} iterations with error {error:.2e}.")
```

```
Converged in 205 iterations with error 9.92e-05.
```

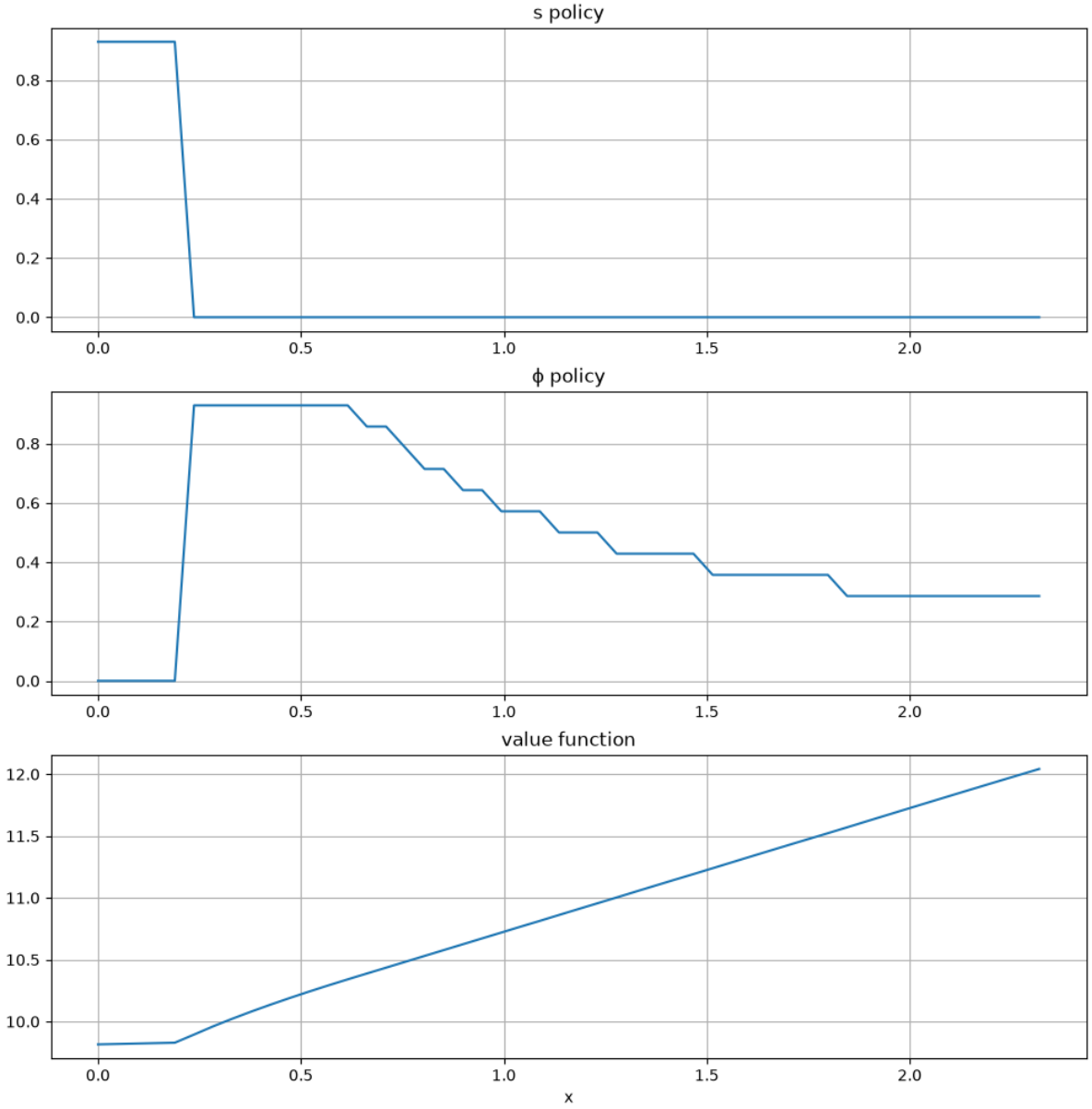
Here are the plots:

```
plots = [s_star, ϕ_star, v_star]
titles = ["s policy", "ϕ policy",  "value function"]

fig, axes = plt.subplots(3, 1, figsize=(12, 12))

for ax, plot, title in zip(axes, plots, titles):
    ax.plot(jv.x_grid, plot)
    ax.set(title=title)
    ax.grid()
```

(continues on next page)

(continued from previous page)

```
axes[-1].set_xlabel("x")
plt.show()
```



The horizontal axis is the state x, while the vertical axis gives $s(x)$ and $\phi(x)$.

Overall, the policies match well with our predictions from *above*

- Worker switches from one investment strategy to the other depending on relative return.
- For low values of x, the best option is to search for a new job.
- Once x is larger, worker does better by investing in human capital specific to the current position.

56.5 Exercises

Exercise 56.5.1

Let's look at the dynamics for the state process $\{x_t\}$ associated with these policies.

The dynamics are given by (56.1) when ϕ_t and s_t are chosen according to the optimal policies, and $\mathbb{P}\{b_{t+1} = 1\} = \pi(s_t)$.

Since the dynamics are random, analysis is a bit subtle.

One way to do it is to plot, for each x in a relatively fine grid called `plot_grid`, a large number K of realizations of x_{t+1} given $x_t = x$.

Plot this with one dot for each realization, in the form of a 45 degree diagram, setting

```
jv = create_jv_worker(grid_size=25, mc_size=50)
plot_grid_max, plot_grid_size = 1.2, 100
plot_grid = jnp.linspace(0, plot_grid_max, plot_grid_size)
fig, ax = plt.subplots()
ax.set_xlim(0, plot_grid_max)
ax.set_ylim(0, plot_grid_max)
```

By examining the plot, argue that under the optimal policies, the state x_t will converge to a constant value $\bar{x}$ close to unity.

Argue that at the steady state, $s_t \approx 0$ and $\phi_t \approx 0.6$.

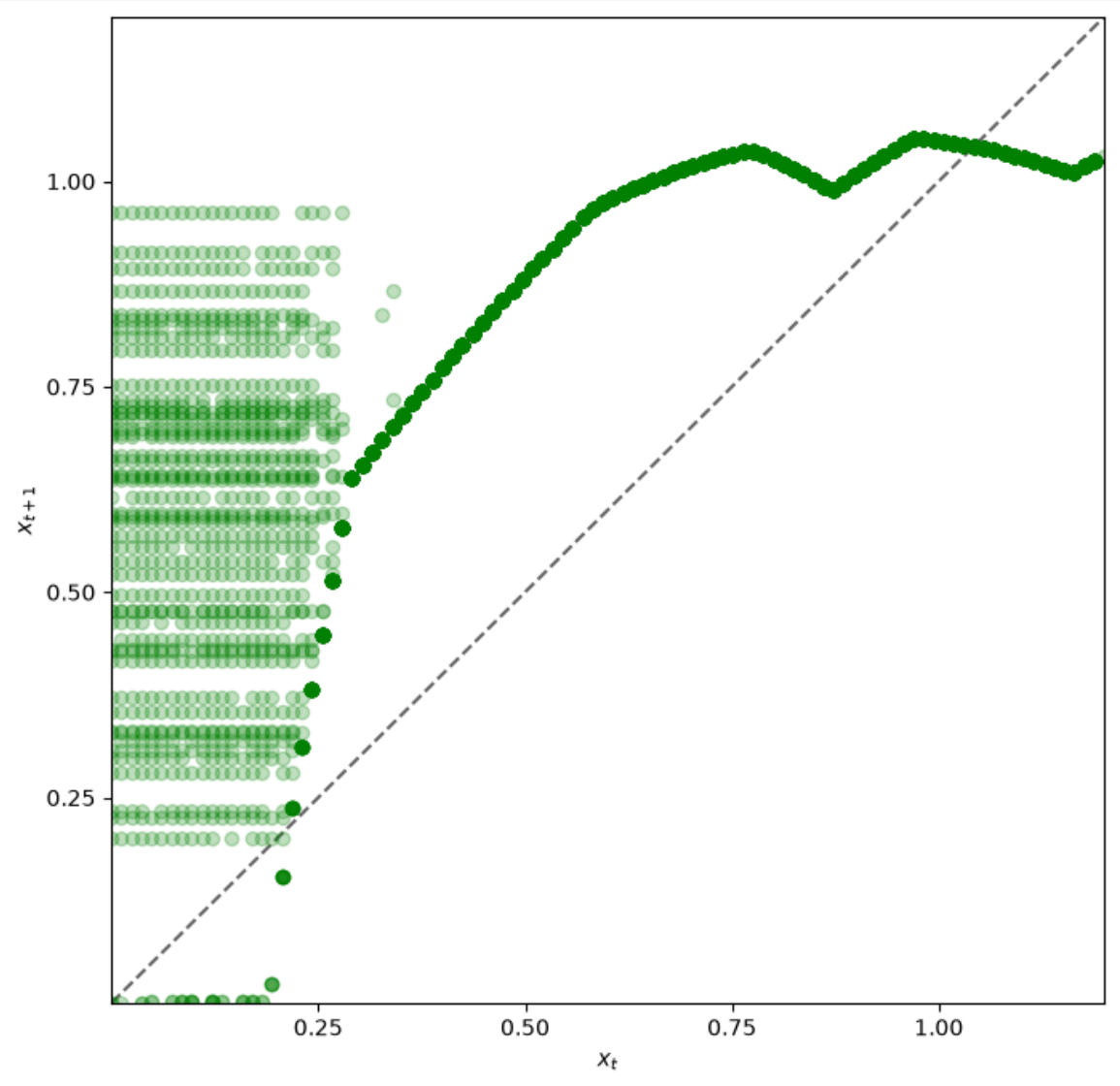
Solution

Here's code to produce the 45 degree diagram.

Note that we draw all of the realizations at once, rather than looping over states and draws.

```
jv = create_jv_worker(grid_size=25, mc_size=50)
v_star, _, _ = solve_model(jv)
s_policy, ϕ_policy = get_greedy(v_star, jv)

# Turn the policy function arrays into actual functions
s = lambda y: jnp.interp(y, jv.x_grid, s_policy)
ϕ = lambda y: jnp.interp(y, jv.x_grid, ϕ_policy)

plot_grid_max, plot_grid_size = 1.2, 100
plot_grid = jnp.linspace(0, plot_grid_max, plot_grid_size)


@jax.jit
def simulate_next(key, plot_grid):
    """
    Draw realizations of next period capital for every x in plot_grid,
    following the law of motion for x_{t+1} given above.  Returns an array of␣
↪shape (len(plot_grid), mc_size).
    """
    K = len(jv.f_rvs)
    gxϕ = g(jv, plot_grid, ϕ(plot_grid))[:, jnp.newaxis]    # Shape (n, 1)
    u = jv.f_rvs[jnp.newaxis, :]                            # Shape (1, K)

    # An offer arrives with probability π(s(x)), independently across draws
```

```
    b = jr.uniform(key, (len(plot_grid), K)) < π(s(plot_grid))[:, jnp.newaxis]

    return jnp.where(b, jnp.maximum(gxϕ, u), gxϕ)


x_next = simulate_next(jr.key(1234), plot_grid)

fig, ax = plt.subplots(figsize=(8, 8))
ticks = (0.25, 0.5, 0.75, 1.0)
ax.set(xticks=ticks, yticks=ticks,
       xlim=(0, plot_grid_max),
       ylim=(0, plot_grid_max),
       xlabel='$x_t$', ylabel='$x_{t+1}$')

ax.plot(plot_grid, plot_grid, 'k--', alpha=0.6)  # 45 degree line
ax.plot(jnp.repeat(plot_grid, x_next.shape[1]), x_next.ravel(),
        'go', alpha=0.25)

plt.show()
```



Looking at the dynamics, we can see that

- If x_t is below about 0.2 the dynamics are random, but $x_{t+1} > x_t$ is very likely.
- As x_t increases the dynamics become deterministic, and x_t converges to a steady state value close to 1.

Referring back to the figure *here* we see that $x_t \approx 1$ means that $s_t = s(x_t) \approx 0$ and $\phi_t = \phi(x_t) \approx 0.6$.

Exercise 56.5.2

In Exercise 56.5.1, we found that s_t converges to zero and ϕ_t converges to about 0.6.

Since these results were calculated at a value of β close to one, let's compare them to the best choice for an *infinitely* patient worker.

Intuitively, an infinitely patient worker would like to maximize steady state wages, which are a function of steady state capital.

You can take it as given—it's certainly true—that the infinitely patient worker does not search in the long run (i.e., $s_t = 0$ for large t).

Thus, given ϕ, steady state capital is the positive fixed point $x^*(\phi)$ of the map $x \mapsto g(x, \phi)$.

Steady state wages can be written as $w^*(\phi) = x^*(\phi)(1 - \phi)$.

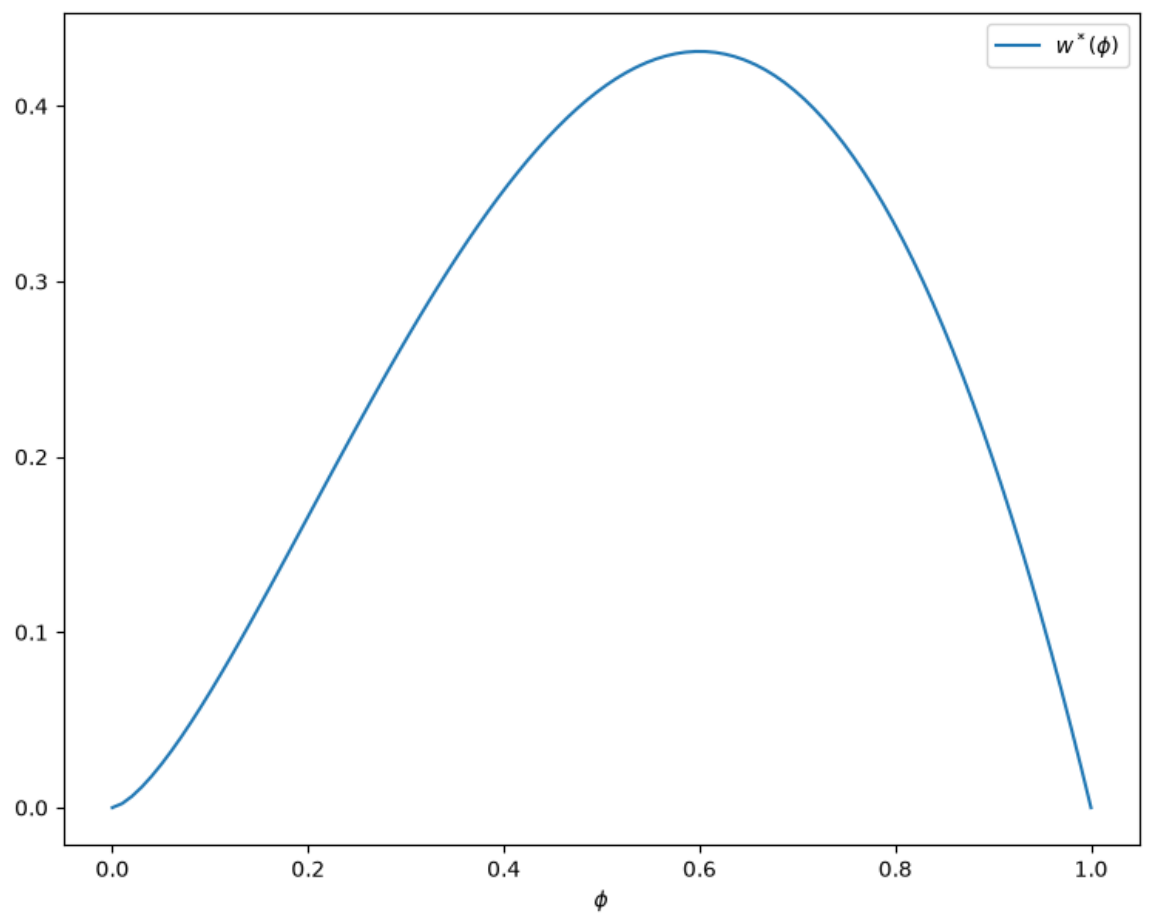
Graph $w^*(\phi)$ with respect to ϕ, and examine the best choice of ϕ.

Can you give a rough interpretation for the value that you see?

Solution

The figure can be produced as follows

```
jv = create_jv_worker()

def xbar(ϕ):
    return (jv.A * ϕ**jv.α)**(1 / (1 - jv.α))

ϕ_grid = jnp.linspace(0, 1, 100)

fig, ax = plt.subplots(figsize=(9, 7))
ax.set(xlabel=r'$\phi$')
ax.plot(ϕ_grid, xbar(ϕ_grid) * (1 - ϕ_grid), label=r'$w^*(\phi)$')
ax.legend()

plt.show()
```



Observe that the maximizer is around 0.6.

This is similar to the long-run value for ϕ obtained in Exercise 56.5.1.

Hence the behavior of the infinitely patent worker is similar to that of the worker with $\beta = 0.96$.

This seems reasonable and helps us confirm that our dynamic programming solutions are probably correct.

CHAPTER

FIFTYSEVEN

JOB SEARCH VIII: SEARCH WITH LEARNING

Contents

- *Job Search VIII: Search with Learning*
  - *Overview*
  - *Model*
  - *Take 1: Solution by VFI*
  - *Take 2: A More Efficient Method*
  - *Another Functional Equation*
  - *Solving the RWFE*
  - *Implementation*
  - *Exercises*
  - *Solutions*
  - *Appendix A*
  - *Appendix B*
  - *Examples*

In addition to what's in Anaconda, this lecture deploys the libraries:

```
!pip install interpolation
```

57.1 Overview

In this lecture, we consider an extension of the previously studied job search model of McCall [McCall, 1970].

We'll build on a model of Bayesian learning discussed in this lecture on the topic of exchangeability and its relationship to the concept of IID (identically and independently distributed) random variables and to Bayesian updating.

In the McCall model, an unemployed worker decides when to accept a permanent job at a specific fixed wage, given

- his or her discount factor
- the level of unemployment compensation

- the distribution from which wage offers are drawn

In the version considered below, the wage distribution is unknown and must be learned.

- The following is based on the presentation in [Ljungqvist and Sargent, 2018], section 6.6.

Let's start with some imports

```
import matplotlib.pyplot as plt
from numba import jit, prange, vectorize
from interpolation import mlinterp
from math import gamma
import numpy as np
from matplotlib import cm
import scipy.optimize as op
from scipy.stats import cumfreq, beta
```

57.1.1 Model Features

- Infinite horizon dynamic programming with two states and one binary control.
- Bayesian updating to learn the unknown distribution.

57.2 Model

Let's first review the basic McCall model [McCall, 1970] and then add the variation we want to consider.

57.2.1 The Basic McCall Model

Recall that, *in the baseline model*, an unemployed worker is presented in each period with a permanent job offer at wage W_t.

At time t, our worker either

1. accepts the offer and works permanently at constant wage W_t
2. rejects the offer, receives unemployment compensation c and reconsiders next period

The wage sequence W_t is IID and generated from known density q.

The worker aims to maximize the expected discounted sum of earnings $\mathbb{E} \sum_{t=0}^{\infty} \beta^t y_t$.

Let $v(w)$ be the optimal value of the problem for a previously unemployed worker who has just received offer w and is yet to decide whether to accept or reject the offer.

The value function v satisfies the recursion

$$v(w) = \max \left\{ \frac{w}{1-\beta}, \, c + \beta \int v(w') q(w') dw' \right\} \tag{57.1}$$

The optimal policy has the form $\mathbf{1}\{w \geq \bar{w}\}$, where $\bar{w}$ is a constant called the **reservation wage**.

57.2.2 Offer Distribution Unknown

Now let's extend the model by considering the variation presented in [Ljungqvist and Sargent, 2018], section 6.6.

The model is as above, apart from the fact that

- the density q is unknown
- the worker learns about q by starting with a prior and updating based on wage offers that he/she observes

The worker knows there are two possible distributions F and G.

These two distributions have densities f and g, repectively.

Just before time starts, "nature" selects q to be either f or g.

This is then the wage distribution from which the entire sequence W_t will be drawn.

The worker does not know which distribution nature has drawn, but the worker does know the two possible distributions f and g.

The worker puts a (subjective) prior probability π_0 on f having been chosen.

The worker's time 0 subjective distribution for the distribution of W_0 is

$$\pi_0 f + (1 - \pi_0) g$$

The worker's time t subjective belief about the the distribution of W_t is

$$\pi_t f + (1 - \pi_t) g,$$

where π_t updates via

$$\pi_{t+1} = \frac{\pi_t f(w_{t+1})}{\pi_t f(w_{t+1}) + (1 - \pi_t) g(w_{t+1})} \tag{57.2}$$

This last expression follows from Bayes' rule, which tells us that

$$\mathbb{P}\{q = f \,|\, W = w\} = \frac{\mathbb{P}\{W = w \,|\, q = f\}\mathbb{P}\{q = f\}}{\mathbb{P}\{W = w\}}$$

and

$$\mathbb{P}\{W = w\} = \sum_{\omega \in \{f, g\}} \mathbb{P}\{W = w \,|\, q = \omega\}\mathbb{P}\{q = \omega\}$$

The fact that (57.2) is recursive allows us to progress to a recursive solution method.

Letting

$$q_\pi(w) := \pi f(w) + (1 - \pi) g(w)$$

and

$$\kappa(w, \pi) := \frac{\pi f(w)}{\pi f(w) + (1 - \pi) g(w)}$$

we can express the value function for the unemployed worker recursively as follows

$$v(w, \pi) = \max \left\{ \frac{w}{1 - \beta}, \, c + \beta \int v(w', \pi') \, q_\pi(w') \, dw' \right\} \quad \text{where} \quad \pi' = \kappa(w', \pi) \tag{57.3}$$

Notice that the current guess π is a state variable, since it affects the worker's perception of probabilities for future rewards.

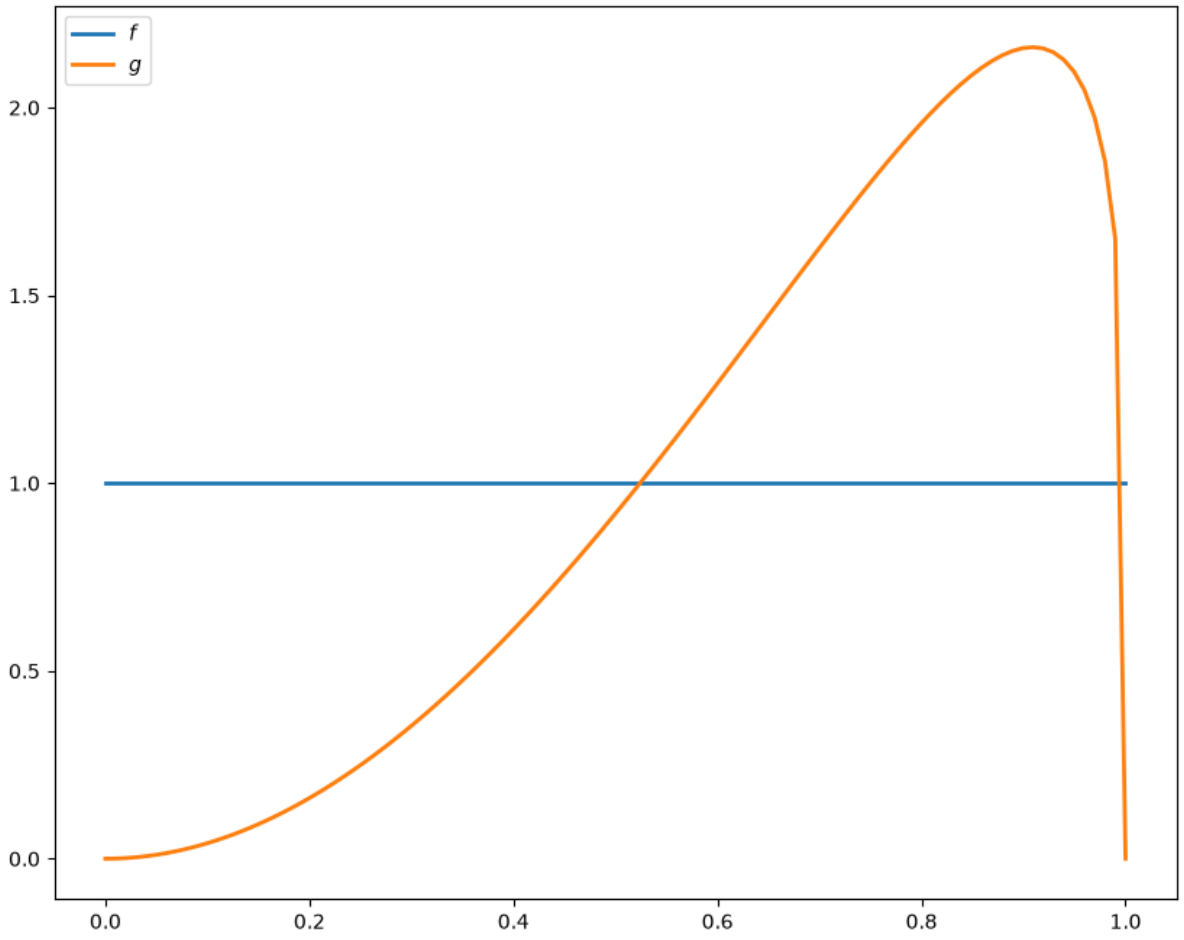
57.2.3 Parameterization

Following section 6.6 of [Ljungqvist and Sargent, 2018], our baseline parameterization will be

- f is Beta$(1, 1)$
- g is Beta$(3, 1.2)$
- $\beta = 0.95$ and $c = 0.3$

The densities f and g have the following shape

```
@vectorize
def p(x, a, b):
    r = gamma(a + b) / (gamma(a) * gamma(b))
    return r * x**(a-1) * (1 - x)**(b-1)


x_grid = np.linspace(0, 1, 100)
f = lambda x: p(x, 1, 1)
g = lambda x: p(x, 3, 1.2)

fig, ax = plt.subplots(figsize=(10, 8))
ax.plot(x_grid, f(x_grid), label='$f$', lw=2)
ax.plot(x_grid, g(x_grid), label='$g$', lw=2)

ax.legend()
plt.show()
```



57.2.4 Looking Forward

What kind of optimal policy might result from (57.3) and the parameterization specified above?

Intuitively, if we accept at w_a and $w_a \leq w_b$, then – all other things being given – we should also accept at w_b.

This suggests a policy of accepting whenever w exceeds some threshold value $\bar{w}$.

But $\bar{w}$ should depend on π – in fact, it should be decreasing in π because

- f is a less attractive offer distribution than g
- larger π means more weight on f and less on g

Thus, larger π depresses the worker's assessment of her future prospects, so relatively low current offers become more attractive.

Summary: We conjecture that the optimal policy is of the form $\mathbb{1}w \geq \bar{w}(\pi)$ for some decreasing function $\bar{w}$.

57.3 Take 1: Solution by VFI

Let's set about solving the model and see how our results match with our intuition.

We begin by solving via value function iteration (VFI), which is natural but ultimately turns out to be second best.

The class `SearchProblem` is used to store parameters and methods needed to compute optimal actions.

```
class SearchProblem:
    """
    A class to store a given parameterization of the "offer distribution
    unknown" model.

    """

    def __init__(self,
                 β=0.95,            # Discount factor
                 c=0.3,             # Unemployment compensation
                 F_a=1,
                 F_b=1,
                 G_a=3,
                 G_b=1.2,
                 w_max=1,           # Maximum wage possible
                 w_grid_size=100,
                 π_grid_size=100,
                 mc_size=500):

        self.β, self.c, self.w_max = β, c, w_max

        self.f = jit(lambda x: p(x, F_a, F_b))
        self.g = jit(lambda x: p(x, G_a, G_b))

        self.π_min, self.π_max = 1e-3, 1-1e-3    # Avoids instability
        self.w_grid = np.linspace(0, w_max, w_grid_size)
        self.π_grid = np.linspace(self.π_min, self.π_max, π_grid_size)

        self.mc_size = mc_size

        self.w_f = np.random.beta(F_a, F_b, mc_size)
        self.w_g = np.random.beta(G_a, G_b, mc_size)
```

The following function takes an instance of this class and returns jitted versions of the Bellman operator `T`, and a `get_greedy()` function to compute the approximate optimal policy from a guess `v` of the value function

```
def operator_factory(sp, parallel_flag=True):

    f, g = sp.f, sp.g
    w_f, w_g = sp.w_f, sp.w_g
    β, c = sp.β, sp.c
    mc_size = sp.mc_size
    w_grid, π_grid = sp.w_grid, sp.π_grid

    @jit
    def v_func(x, y, v):
        return mlinterp((w_grid, π_grid), v, (x, y))

    @jit
    def κ(w, π):
```

(continues on next page)

(continued from previous page)

```
        """
        Updates π using Bayes' rule and the current wage observation w.
        """
        pf, pg = π * f(w), (1 - π) * g(w)
        π_new = pf / (pf + pg)

        return π_new

    @jit(parallel=parallel_flag)
    def T(v):
        """
        The Bellman operator.

        """
        v_new = np.empty_like(v)

        for i in prange(len(w_grid)):
            for j in prange(len(π_grid)):
                w = w_grid[i]
                π = π_grid[j]

                v_1 = w / (1 - β)

                integral_f, integral_g = 0, 0
                for m in prange(mc_size):
                    integral_f += v_func(w_f[m], κ(w_f[m], π), v)
                    integral_g += v_func(w_g[m], κ(w_g[m], π), v)
                integral = (π * integral_f + (1 - π) * integral_g) / mc_size

                v_2 = c + β * integral
                v_new[i, j] = max(v_1, v_2)

        return v_new

    @jit(parallel=parallel_flag)
    def get_greedy(v):
        """"
        Compute optimal actions taking v as the value function.

        """
        σ = np.empty_like(v)

        for i in prange(len(w_grid)):
            for j in prange(len(π_grid)):
                w = w_grid[i]
                π = π_grid[j]

                v_1 = w / (1 - β)

                integral_f, integral_g = 0, 0
                for m in prange(mc_size):
                    integral_f += v_func(w_f[m], κ(w_f[m], π), v)
                    integral_g += v_func(w_g[m], κ(w_g[m], π), v)
                integral = (π * integral_f + (1 - π) * integral_g) / mc_size

                v_2 = c + β * integral
```

(continues on next page)

(continued from previous page)

```
                σ[i, j] = v_1 > v_2  # Evaluates to 1 or 0

        return σ

    return T, get_greedy
```

We will omit a detailed discussion of the code because there is a more efficient solution method that we will use later.

To solve the model we will use the following function that iterates using T to find a fixed point

```
def solve_model(sp,
                use_parallel=True,
                tol=1e-4,
                max_iter=1000,
                verbose=True,
                print_skip=5):

    """
    Solves for the value function

    * sp is an instance of SearchProblem
    """

    T, _ = operator_factory(sp, use_parallel)

    # Set up loop
    i = 0
    error = tol + 1
    m, n = len(sp.w_grid), len(sp.π_grid)

    # Initialize v
    v = np.zeros((m, n)) + sp.c / (1 - sp.β)

    while i < max_iter and error > tol:
        v_new = T(v)
        error = np.max(np.abs(v - v_new))
        i += 1
        if verbose and i % print_skip == 0:
            print(f"Error at iteration {i} is {error}.")
        v = v_new

    if error > tol:
        print("Failed to converge!")
    elif verbose:
        print(f"\nConverged in {i} iterations.")


    return v_new
```

Let's look at solutions computed from value function iteration

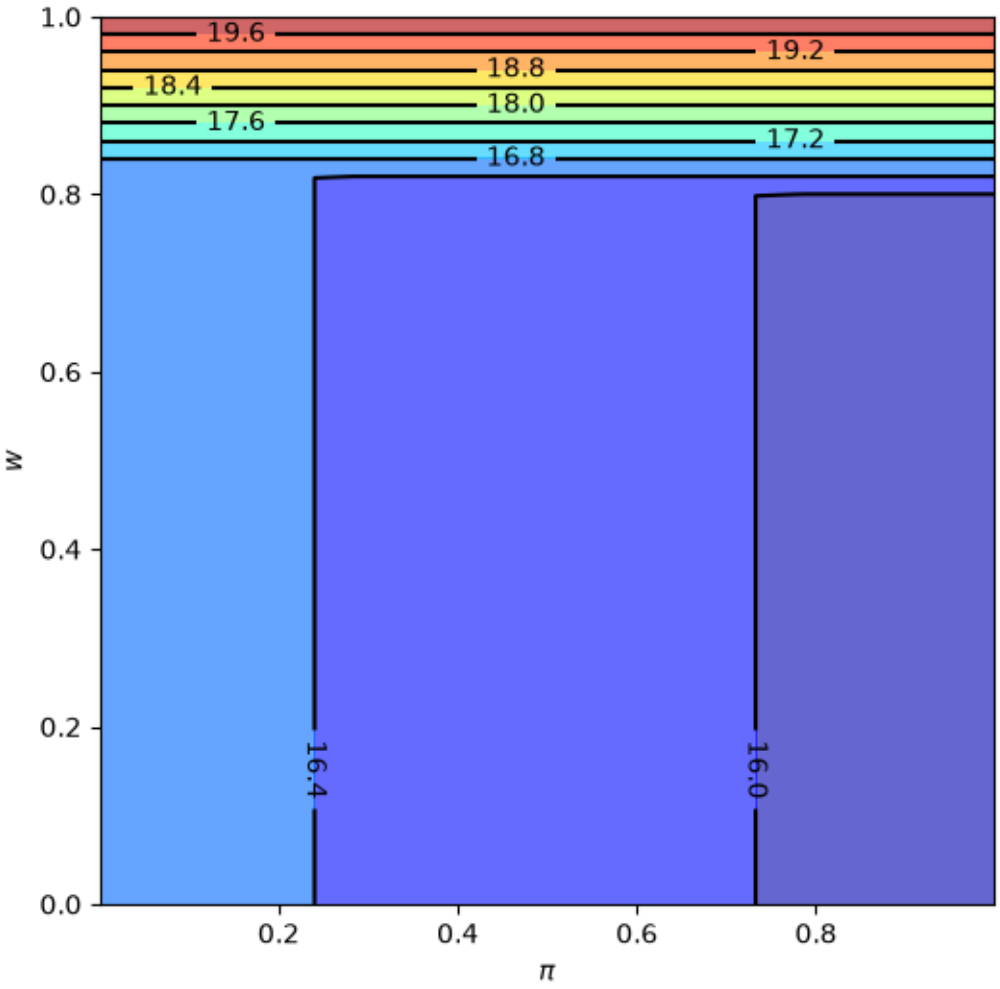
```
sp = SearchProblem()
v_star = solve_model(sp)
fig, ax = plt.subplots(figsize=(6, 6))
ax.contourf(sp.π_grid, sp.w_grid, v_star, 12, alpha=0.6, cmap=cm.jet)
cs = ax.contour(sp.π_grid, sp.w_grid, v_star, 12, colors="black")
ax.clabel(cs, inline=1, fontsize=10)
```

(continues on next page)

(continued from previous page)

```
ax.set(xlabel=r'$\pi$', ylabel='$w$')

plt.show()
```

```
Error at iteration 5 is 0.6479122277746061.
Error at iteration 10 is 0.1032318911935981.
Error at iteration 15 is 0.022188780698234822.
Error at iteration 20 is 0.004884910305646528.
Error at iteration 25 is 0.0010812556698382991.
Error at iteration 30 is 0.00023934391512447917.

Converged in 33 iterations.
```



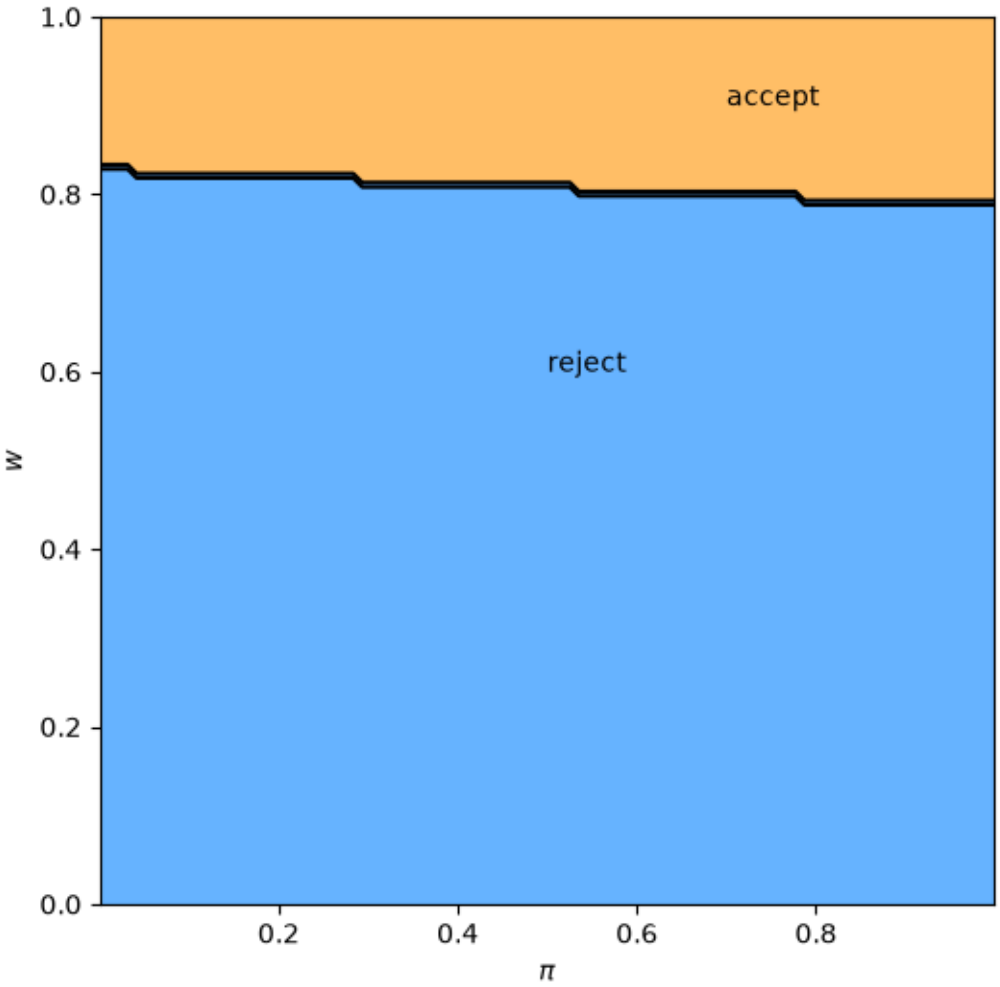
We will also plot the optimal policy

```
T, get_greedy = operator_factory(sp)
σ_star = get_greedy(v_star)

fig, ax = plt.subplots(figsize=(6, 6))
ax.contourf(sp.π_grid, sp.w_grid, σ_star, 1, alpha=0.6, cmap=cm.jet)
ax.contour(sp.π_grid, sp.w_grid, σ_star, 1, colors="black")
```

(continues on next page)

(continued from previous page)

```
ax.set(xlabel=r'$\pi$', ylabel='$w$')

ax.text(0.5, 0.6, 'reject')
ax.text(0.7, 0.9, 'accept')

plt.show()
```



The results fit well with our intuition from section *looking forward*.

- The black line in the figure above corresponds to the function $\bar{w}(\pi)$ introduced there.
- It is decreasing as expected.

57.4 Take 2: A More Efficient Method

Let's consider another method to solve for the optimal policy.

We will use iteration with an operator that has the same contraction rate as the Bellman operator, but

- one dimensional rather than two dimensional
- no maximization step

As a consequence, the algorithm is orders of magnitude faster than VFI.

This section illustrates the point that when it comes to programming, a bit of mathematical analysis goes a long way.

57.5 Another Functional Equation

To begin, note that when $w = \bar{w}(\pi)$, the worker is indifferent between accepting and rejecting.

Hence the two choices on the right-hand side of (57.3) have equal value:

$$\frac{\bar{w}(\pi)}{1-\beta} = c + \beta \int v(w', \pi')\, q_\pi(w')\, dw' \tag{57.4}$$

Together, (57.3) and (57.4) give

$$v(w, \pi) = \max\left\{\frac{w}{1-\beta}, \frac{\bar{w}(\pi)}{1-\beta}\right\} \tag{57.5}$$

Combining (57.4) and (57.5), we obtain

$$\frac{\bar{w}(\pi)}{1-\beta} = c + \beta \int \max\left\{\frac{w'}{1-\beta}, \frac{\bar{w}(\pi')}{1-\beta}\right\} q_\pi(w')\, dw'$$

Multiplying by $1-\beta$, substituting in $\pi' = \kappa(w', \pi)$ and using $\circ$ for composition of functions yields

$$\bar{w}(\pi) = (1-\beta)c + \beta \int \max\left\{w', \bar{w} \circ \kappa(w', \pi)\right\}\, q_\pi(w')\, dw' \tag{57.6}$$

Equation (57.6) can be understood as a functional equation, where $\bar{w}$ is the unknown function.

- Let's call it the **reservation wage functional equation** (RWFE).
- The solution $\bar{w}$ to the RWFE is the object that we wish to compute.

57.6 Solving the RWFE

To solve the RWFE, we will first show that its solution is the fixed point of a contraction mapping.

To this end, let

- $b[0, 1]$ be the bounded real-valued functions on $[0, 1]$
- $\|\omega\| := \sup_{x \in [0,1]} |\omega(x)|$

Consider the operator Q mapping $\omega \in b[0, 1]$ into $Q\omega \in b[0, 1]$ via

$$(Q\omega)(\pi) = (1-\beta)c + \beta \int \max\left\{w', \omega \circ \kappa(w', \pi)\right\}\, q_\pi(w')\, dw' \tag{57.7}$$

Comparing (57.6) and (57.7), we see that the set of fixed points of Q exactly coincides with the set of solutions to the RWFE.

- If $Q\bar{w} = \bar{w}$ then $\bar{w}$ solves (57.6) and vice versa.

Moreover, for any $\omega, \omega' \in b[0,1]$, basic algebra and the triangle inequality for integrals tells us that

$$|(Q\omega)(\pi) - (Q\omega')(\pi)| \leq \beta \int |\max\{w', \omega \circ \kappa(w', \pi)\} - \max\{w', \omega' \circ \kappa(w', \pi)\}| \; q_\pi(w') \, dw' \tag{57.8}$$

Working case by case, it is easy to check that for real numbers a, b, c we always have

$$|\max\{a, b\} - \max\{a, c\}| \leq |b - c| \tag{57.9}$$

Combining (57.8) and (57.9) yields

$$|(Q\omega)(\pi) - (Q\omega')(\pi)| \leq \beta \int |\omega \circ \kappa(w', \pi) - \omega' \circ \kappa(w', \pi)| \; q_\pi(w') \, dw' \leq \beta \|\omega - \omega'\| \tag{57.10}$$

Taking the supremum over π now gives us

$$\|Q\omega - Q\omega'\| \leq \beta \|\omega - \omega'\| \tag{57.11}$$

In other words, Q is a contraction of modulus β on the complete metric space $(b[0,1], \|\cdot\|)$.

Hence

- A unique solution $\bar{w}$ to the RWFE exists in $b[0,1]$.
- $Q^k \omega \to \bar{w}$ uniformly as $k \to \infty$, for any $\omega \in b[0,1]$.

57.7 Implementation

The following function takes an instance of `SearchProblem` and returns the operator `Q`

```
def Q_factory(sp, parallel_flag=True):

    f, g = sp.f, sp.g
    w_f, w_g = sp.w_f, sp.w_g
    β, c = sp.β, sp.c
    mc_size = sp.mc_size
    w_grid, π_grid = sp.w_grid, sp.π_grid

    @jit
    def ω_func(p, ω):
        return np.interp(p, π_grid, ω)

    @jit
    def κ(w, π):
        """
        Updates π using Bayes' rule and the current wage observation w.
        """
        pf, pg = π * f(w), (1 - π) * g(w)
        π_new = pf / (pf + pg)

        return π_new

    @jit(parallel=parallel_flag)
    def Q(ω):
        """
```

(continues on next page)

(continued from previous page)

```
        Updates the reservation wage function guess ω via the operator
        Q.

        """
        ω_new = np.empty_like(ω)

        for i in prange(len(π_grid)):
            π = π_grid[i]
            integral_f, integral_g = 0, 0

            for m in prange(mc_size):
                integral_f += max(w_f[m], ω_func(κ(w_f[m], π), ω))
                integral_g += max(w_g[m], ω_func(κ(w_g[m], π), ω))
            integral = (π * integral_f + (1 - π) * integral_g) / mc_size

            ω_new[i] = (1 - β) * c + β * integral

        return ω_new

    return Q
```

In the next exercise, you are asked to compute an approximation to $\bar{w}$.

57.8 Exercises

Exercise 57.8.1

Use the default parameters and `Q_factory` to compute an optimal policy.

Your result should coincide closely with the figure for the optimal policy *shown above*.

Try experimenting with different parameters, and confirm that the change in the optimal policy coincides with your intuition.

57.9 Solutions

Solution

This code solves the "Offer Distribution Unknown" model by iterating on a guess of the reservation wage function.

You should find that the run time is shorter than that of the value function approach.

Similar to above, we set up a function to iterate with `Q` to find the fixed point

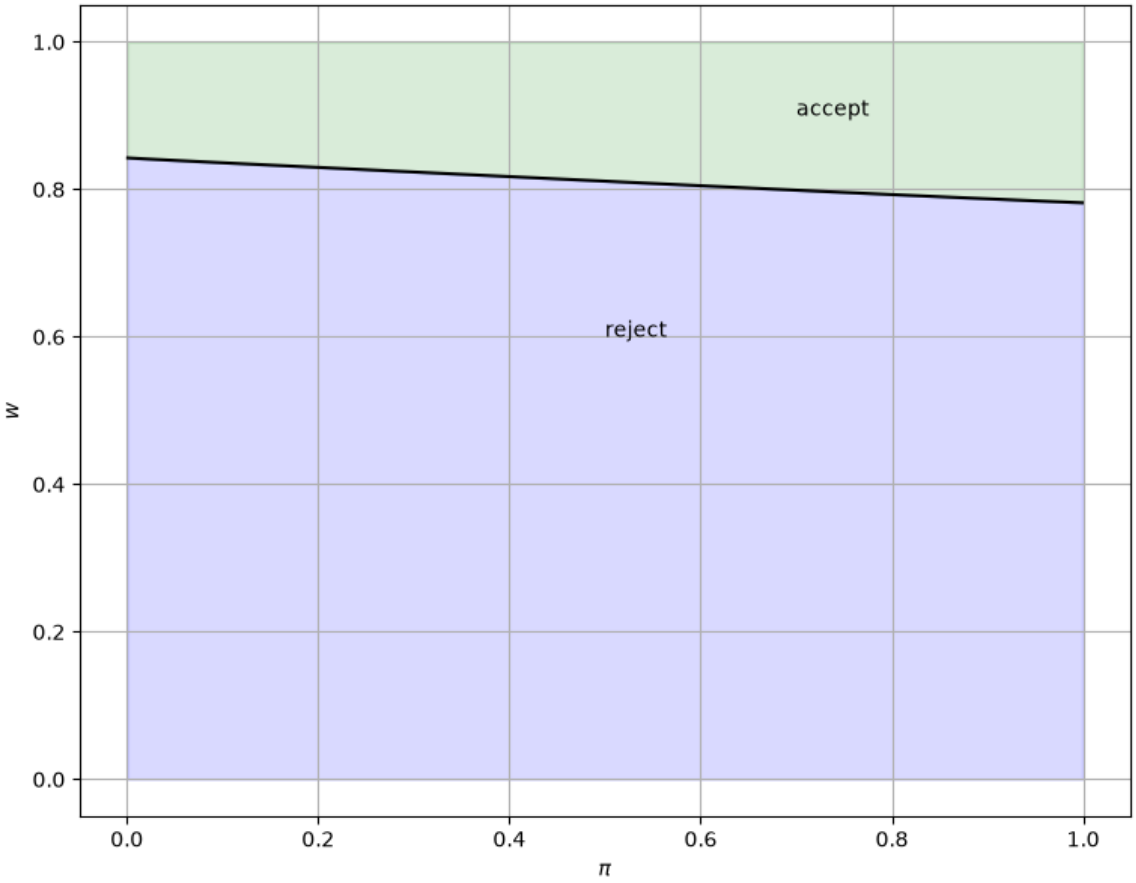
```
def solve_wbar(sp,
               use_parallel=True,
               tol=1e-4,
               max_iter=1000,
               verbose=True,
               print_skip=5):

    Q = Q_factory(sp, use_parallel)
```

```
    # Set up loop
    i = 0
    error = tol + 1
    m, n = len(sp.w_grid), len(sp.π_grid)

    # Initialize w
    w = np.ones_like(sp.π_grid)

    while i < max_iter and error > tol:
        w_new = Q(w)
        error = np.max(np.abs(w - w_new))
        i += 1
        if verbose and i % print_skip == 0:
            print(f"Error at iteration {i} is {error}.")
        w = w_new

    if error > tol:
        print("Failed to converge!")
    elif verbose:
        print(f"\nConverged in {i} iterations.")

    return w_new
```

The solution can be plotted as follows

```
sp = SearchProblem()
w_bar = solve_wbar(sp)

fig, ax = plt.subplots(figsize=(9, 7))

ax.plot(sp.π_grid, w_bar, color='k')
ax.fill_between(sp.π_grid, 0, w_bar, color='blue', alpha=0.15)
ax.fill_between(sp.π_grid, w_bar, sp.w_max, color='green', alpha=0.15)
ax.text(0.5, 0.6, 'reject')
ax.text(0.7, 0.9, 'accept')
ax.set(xlabel=r'$\pi$', ylabel='$w$')
ax.grid()
plt.show()
```

```
Error at iteration 5 is 0.021835231479283235.
Error at iteration 10 is 0.006118427747488386.
Error at iteration 15 is 0.0012482653943254318.
Error at iteration 20 is 0.00024126252683831506.

Converged in 23 iterations.
```



57.10 Appendix A

The next piece of code generates a fun simulation to see what the effect of a change in the underlying distribution on the unemployment rate is.

At a point in the simulation, the distribution becomes significantly worse.

It takes a while for agents to learn this, and in the meantime, they are too optimistic and turn down too many jobs.

As a result, the unemployment rate spikes

```
F_a, F_b, G_a, G_b = 1, 1, 3, 1.2

sp = SearchProblem(F_a=F_a, F_b=F_b, G_a=G_a, G_b=G_b)
f, g = sp.f, sp.g

# Solve for reservation wage
w_bar = solve_wbar(sp, verbose=False)

# Interpolate reservation wage function
π_grid = sp.π_grid
w_func = jit(lambda x: np.interp(x, π_grid, w_bar))
```

(continues on next page)

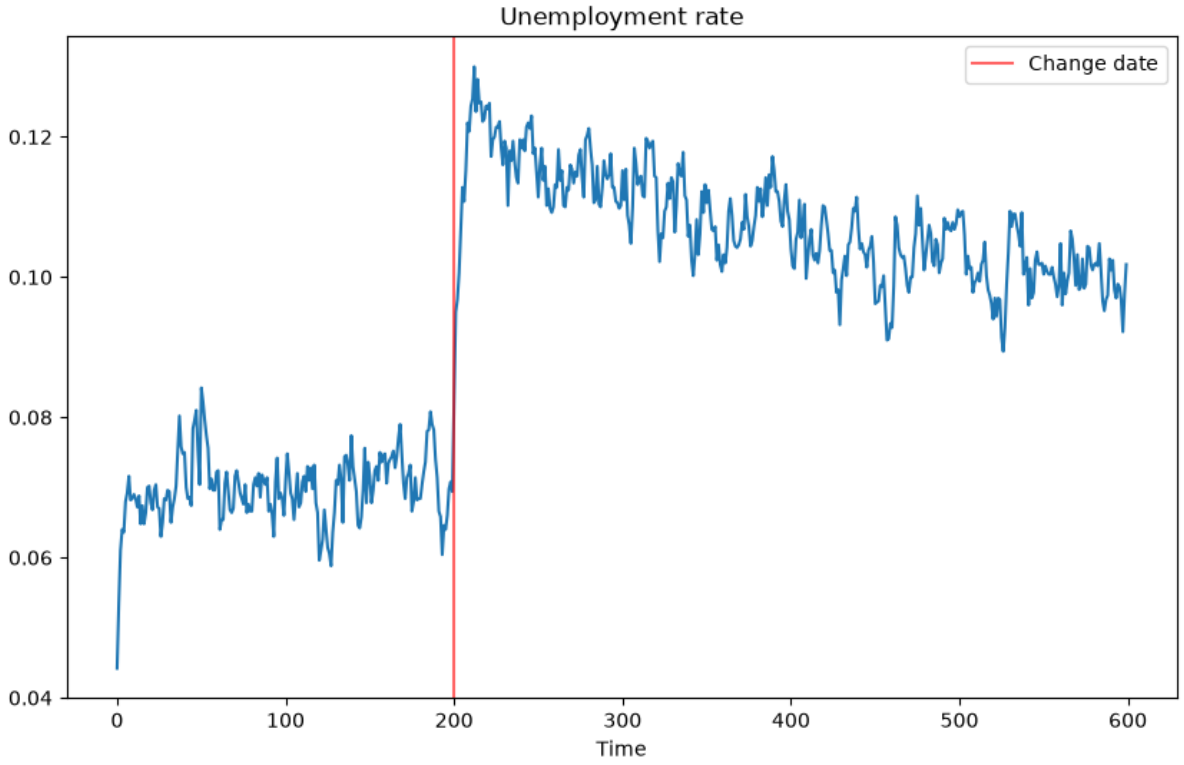
(continued from previous page)

```
@jit
def update(a, b, e, π):
    "Update e and π by drawing wage offer from beta distribution with parameters a␣
 ↪and b"

    if e == False:
        w = np.random.beta(a, b)       # Draw random wage
        if w >= w_func(π):
            e = True                   # Take new job
        else:
            π = 1 / (1 + ((1 - π) * g(w)) / (π * f(w)))

    return e, π

@jit
def simulate_path(F_a=F_a,
                  F_b=F_b,
                  G_a=G_a,
                  G_b=G_b,
                  N=5000,        # Number of agents
                  T=600,         # Simulation length
                  d=200,         # Change date
                  s=0.025):      # Separation rate

    """Simulates path of employment for N number of works over T periods"""

    e = np.ones((N, T+1))
    π = np.full((N, T+1), 1e-3)

    a, b = G_a, G_b   # Initial distribution parameters

    for t in range(T+1):

        if t == d:
            a, b = F_a, F_b  # Change distribution parameters

        # Update each agent
        for n in range(N):
            if e[n, t] == 1:                    # If agent is currently employment
                p = np.random.uniform(0, 1)
                if p <= s:                      # Randomly separate with probability s
                    e[n, t] = 0

            new_e, new_π = update(a, b, e[n, t], π[n, t])
            e[n, t+1] = new_e
            π[n, t+1] = new_π

    return e[:, 1:]

d = 200  # Change distribution at time d
unemployment_rate = 1 - simulate_path(d=d).mean(axis=0)

fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(unemployment_rate)
ax.axvline(d, color='r', alpha=0.6, label='Change date')
ax.set_xlabel('Time')
ax.set_title('Unemployment rate')
```

(continues on next page)

(continued from previous page)

```
ax.legend()
plt.show()
```



57.11 Appendix B

In this appendix we provide more details about how Bayes' Law contributes to the workings of the model.

We present some graphs that bring out additional insights about how learning works.

We build on graphs proposed in this lecture.

In particular, we'll add actions of our searching worker to a key graph presented in that lecture.

To begin, we first define two functions for computing the empirical distributions of unemployment duration and π at the time of employment.

```
@jit
def empirical_dist(F_a, F_b, G_a, G_b, w_bar, π_grid,
                   N=10000, T=600):
    """
    Simulates population for computing empirical cumulative
    distribution of unemployment duration and π at time when
    the worker accepts the wage offer. For each job searching
    problem, we simulate for two cases that either f or g is
    the true offer distribution.

    Parameters
    ----------
```

(continues on next page)

(continued from previous page)

```
    F_a, F_b, G_a, G_b : parameters of beta distributions F and G.
    w_bar : the reservation wage
    π_grid : grid points of π, for interpolation
    N : number of workers for simulation, optional
    T : maximum of time periods for simulation, optional

    Returns
    -------
    accpet_t : 2 by N ndarray. the empirical distribution of
               unemployment duration when f or g generates offers.
    accept_π : 2 by N ndarray. the empirical distribution of
               π at the time of employment when f or g generates offers.
    """

    accept_t = np.empty((2, N))
    accept_π = np.empty((2, N))

    # f or g generates offers
    for i, (a, b) in enumerate([(F_a, F_b), (G_a, G_b)]):
        # update each agent
        for n in range(N):

            # initial priori
            π = 0.5

            for t in range(T+1):

                # Draw random wage
                w = np.random.beta(a, b)
                lw = p(w, F_a, F_b) / p(w, G_a, G_b)
                π = π * lw / (π * lw + 1 - π)

                # move to next agent if accepts
                if w >= np.interp(π, π_grid, w_bar):
                    break

            # record the unemployment duration
            # and π at the time of acceptance
            accept_t[i, n] = t
            accept_π[i, n] = π

    return accept_t, accept_π

def cumfreq_x(res):
    """
    A helper function for calculating the x grids of
    the cumulative frequency histogram.
    """

    cumcount = res.cumcount
    lowerlimit, binsize = res.lowerlimit, res.binsize

    x = lowerlimit + np.linspace(0, binsize*cumcount.size, cumcount.size)

    return x
```

Now we define a wrapper function for analyzing job search models with learning under different parameterizations.

The wrapper takes parameters of beta distributions and unemployment compensation as inputs and then displays various things we want to know to interpret the solution of our search model.

In addition, it computes empirical cumulative distributions of two key objects.

```
def job_search_example(F_a=1, F_b=1, G_a=3, G_b=1.2, c=0.3):
    """
    Given the parameters that specify F and G distributions,
    calculate and display the rejection and acceptance area,
    the evolution of belief π, and the probability of accepting
    an offer at different π level, and simulate and calculate
    the empirical cumulative distribution of the duration of
    unemployment and π at the time the worker accepts the offer.
    """

    # construct a search problem
    sp = SearchProblem(F_a=F_a, F_b=F_b, G_a=G_a, G_b=G_b, c=c)
    f, g = sp.f, sp.g
    π_grid = sp.π_grid

    # Solve for reservation wage
    w_bar = solve_wbar(sp, verbose=False)

    # l(w) = f(w) / g(w)
    l = lambda w: f(w) / g(w)
    # objective function for solving l(w) = 1
    obj = lambda w: l(w) - 1.

    # the mode of beta distribution
    # use this to divide w into two intervals for root finding
    G_mode = (G_a - 1) / (G_a + G_b - 2)
    roots = np.empty(2)
    roots[0] = op.root_scalar(obj, bracket=[1e-10, G_mode]).root
    roots[1] = op.root_scalar(obj, bracket=[G_mode, 1-1e-10]).root

    fig, axs = plt.subplots(2, 2, figsize=(12, 9))

    # part 1: display the details of the model settings and some results
    w_grid = np.linspace(1e-12, 1-1e-12, 100)

    axs[0, 0].plot(l(w_grid), w_grid, label='$l$', lw=2)
    axs[0, 0].vlines(1., 0., 1., linestyle="--")
    axs[0, 0].hlines(roots, 0., 2., linestyle="--")
    axs[0, 0].set_xlim([0., 2.])
    axs[0, 0].legend(loc=4)
    axs[0, 0].set(xlabel='$l(w)=f(w)/g(w)$', ylabel='$w$')

    axs[0, 1].plot(sp.π_grid, w_bar, color='k')
    axs[0, 1].fill_between(sp.π_grid, 0, w_bar, color='blue', alpha=0.15)
    axs[0, 1].fill_between(sp.π_grid, w_bar, sp.w_max, color='green', alpha=0.15)
    axs[0, 1].text(0.5, 0.6, 'reject')
    axs[0, 1].text(0.7, 0.9, 'accept')

    W = np.arange(0.01, 0.99, 0.08)
    Π = np.arange(0.01, 0.99, 0.08)

    ΔW = np.zeros((len(W), len(Π)))
```

(continues on next page)

(continued from previous page)

```
    ΔΠ = np.empty((len(W), len(Π)))
    for i, w in enumerate(W):
        for j, π in enumerate(Π):
            lw = l(w)
            ΔΠ[i, j] = π * (lw / (π * lw + 1 - π) - 1)

    q = axs[0, 1].quiver(Π, W, ΔΠ, ΔW, scale=2, color='r', alpha=0.8)

    axs[0, 1].hlines(roots, 0., 1., linestyle="--")
    axs[0, 1].set(xlabel=r'$\pi$', ylabel='$w$')
    axs[0, 1].grid()

    axs[1, 0].plot(f(x_grid), x_grid, label='$f$', lw=2)
    axs[1, 0].plot(g(x_grid), x_grid, label='$g$', lw=2)
    axs[1, 0].vlines(1., 0., 1., linestyle="--")
    axs[1, 0].hlines(roots, 0., 2., linestyle="--")
    axs[1, 0].legend(loc=4)
    axs[1, 0].set(xlabel='$f(w), g(w)$', ylabel='$w$')

    axs[1, 1].plot(sp.π_grid, 1 - beta.cdf(w_bar, F_a, F_b), label='$f$')
    axs[1, 1].plot(sp.π_grid, 1 - beta.cdf(w_bar, G_a, G_b), label='$g$')
    axs[1, 1].set_ylim([0., 1.])
    axs[1, 1].grid()
    axs[1, 1].legend(loc=4)
    axs[1, 1].set(xlabel=r'$\pi$', ylabel=r'$\mathbb{P}\{w > \overline{w} (\pi)\}$')

    plt.show()

    # part 2: simulate empirical cumulative distribution
    accept_t, accept_π = empirical_dist(F_a, F_b, G_a, G_b, w_bar, π_grid)
    N = accept_t.shape[1]

    cfq_t_F = cumfreq(accept_t[0, :], numbins=100)
    cfq_π_F = cumfreq(accept_π[0, :], numbins=100)

    cfq_t_G = cumfreq(accept_t[1, :], numbins=100)
    cfq_π_G = cumfreq(accept_π[1, :], numbins=100)

    fig, axs = plt.subplots(2, 1, figsize=(12, 9))

    axs[0].plot(cumfreq_x(cfq_t_F), cfq_t_F.cumcount/N, label="f generates")
    axs[0].plot(cumfreq_x(cfq_t_G), cfq_t_G.cumcount/N, label="g generates")
    axs[0].grid(linestyle='--')
    axs[0].legend(loc=4)
    axs[0].title.set_text('CDF of duration of unemployment')
    axs[0].set(xlabel='time', ylabel='Prob(time)')

    axs[1].plot(cumfreq_x(cfq_π_F), cfq_π_F.cumcount/N, label="f generates")
    axs[1].plot(cumfreq_x(cfq_π_G), cfq_π_G.cumcount/N, label="g generates")
    axs[1].grid(linestyle='--')
    axs[1].legend(loc=4)
    axs[1].title.set_text('CDF of π at time worker accepts wage and leaves␣
↪unemployment')
    axs[1].set(xlabel='π', ylabel='Prob(π)')

    plt.show()
```

We now provide some examples that provide insights about how the model works.

57.12 Examples

57.12.1 Example 1 (Baseline)

F ~ Beta(1, 1), G ~ Beta(3, 1.2), c=0.3.

In the graphs below, the red arrows in the upper right figure show how π_t is updated in response to the new information w_t.

Recall the following formula from this lecture

$$\frac{\pi_{t+1}}{\pi_t} = \frac{l\left(w_{t+1}\right)}{\pi_t l\left(w_{t+1}\right) + \left(1 - \pi_t\right)} \begin{cases} > 1 & \text{if } l\left(w_{t+1}\right) > 1 \\ \leq 1 & \text{if } l\left(w_{t+1}\right) \leq 1 \end{cases}$$

The formula implies that the direction of motion of π_t is determined by the relationship between $l(w_t)$ and 1.

The magnitude is small if

- $l(w)$ is close to 1, which means the new w is not very informative for distinguishing two distributions,
- π_{t-1} is close to either 0 or 1, which means the priori is strong.

Will an unemployed worker accept an offer earlier or not, when the actual ruling distribution is g instead of f?

Two countervailing effects are at work.

- if f generates successive wage offers, then w is more likely to be low, but π is moving up toward to 1, which lowers the reservation wage, i.e., the worker becomes less selective the longer he or she remains unemployed.
- if g generates wage offers, then w is more likely to be high, but π is moving downward toward 0, increasing the reservation wage, i.e., the worker becomes more selective the longer he or she remains unemployed.

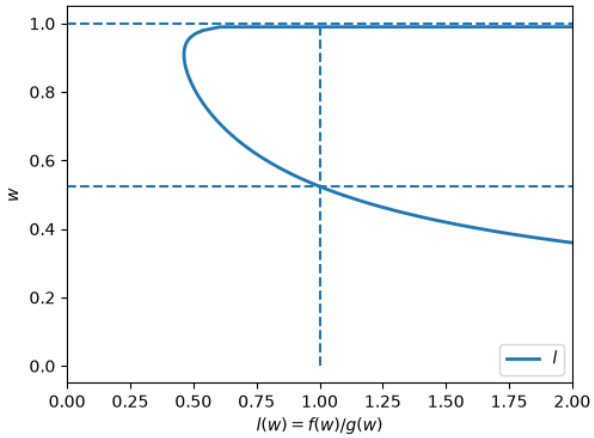
Quantitatively, the lower right figure sheds light on which effect dominates in this example.

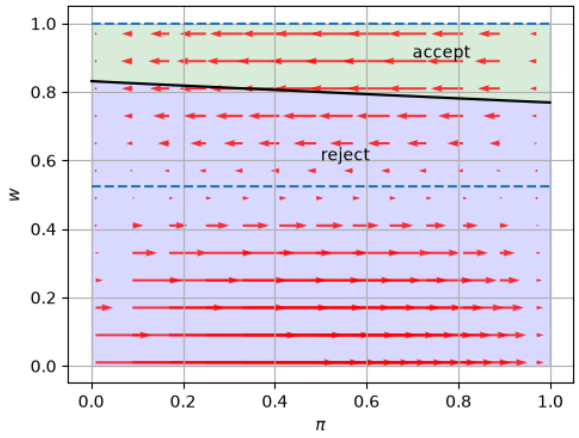
It shows the probability that a previously unemployed worker accepts an offer at different values of π when f or g generates wage offers.

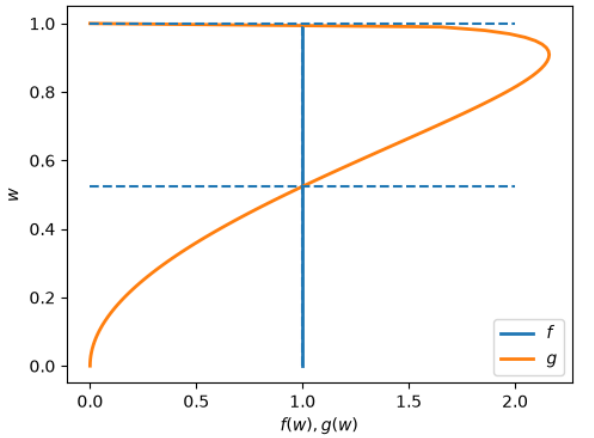
That graph shows that for the particular f and g in this example, the worker is always more likely to accept an offer when f generates the data even when π is close to zero so that the worker believes the true distribution is g and therefore is relatively more selective.

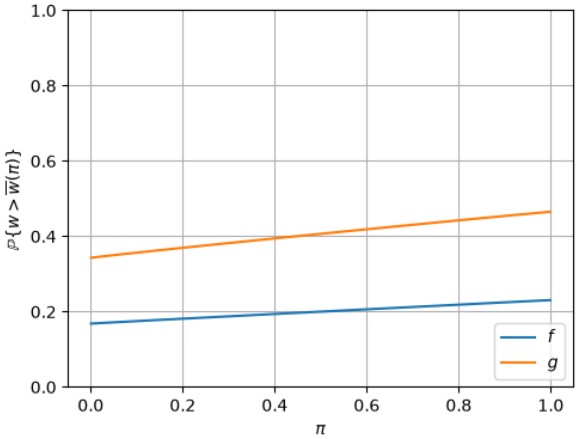
The empirical cumulative distribution of the duration of unemployment verifies our conjecture.

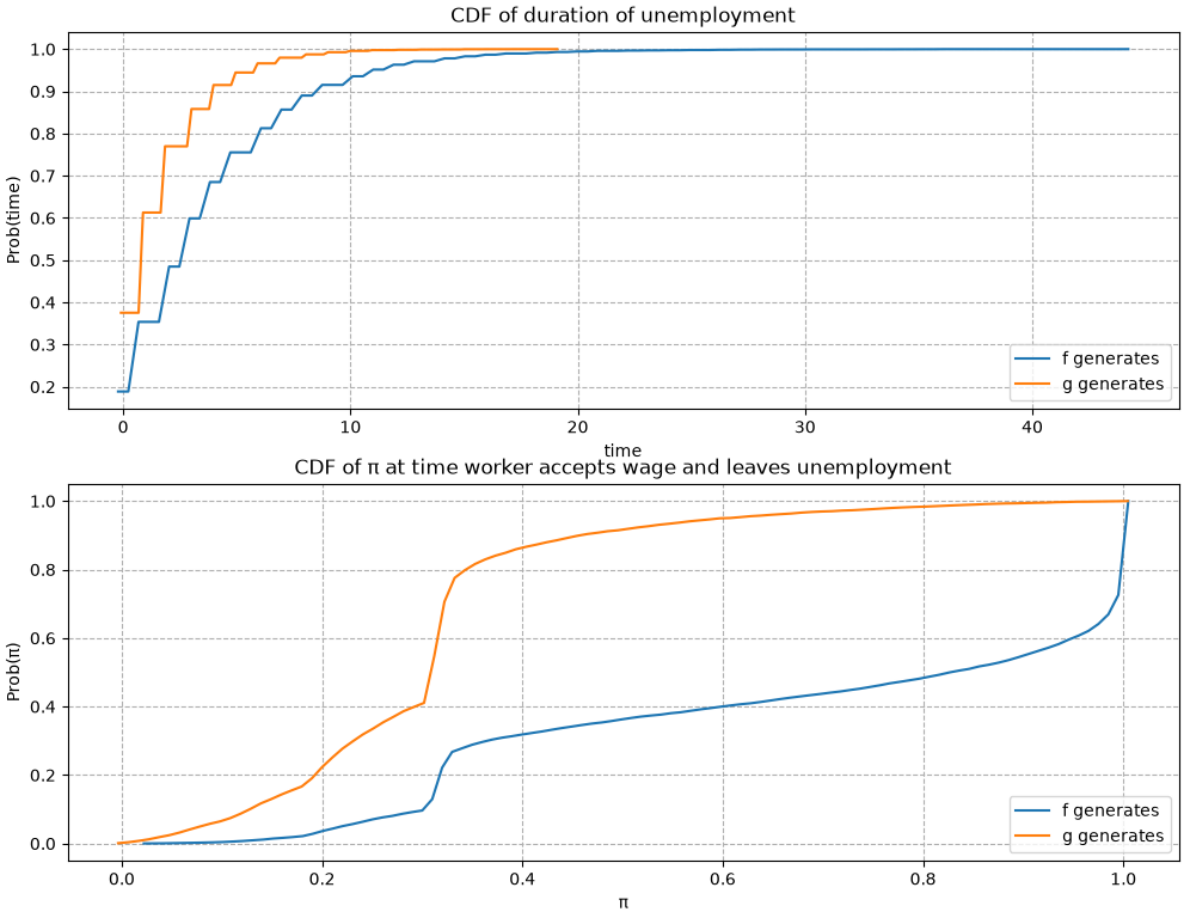
```
job_search_example()
```


w
l(w) = f(w)/g(w)
l


accept
reject
w
π


w
f(w), g(w)
f
g


$\mathbb{P}\{w > \overline{w}(\pi)\}$
π
f
g



57.12.2 Example 2

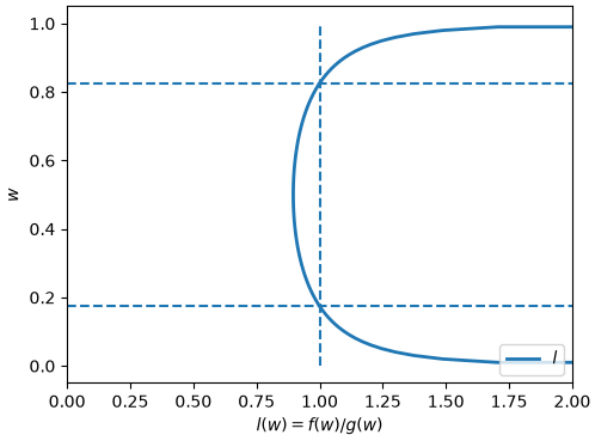
F ~ Beta(1, 1), G ~ Beta(1.2, 1.2), c=0.3.

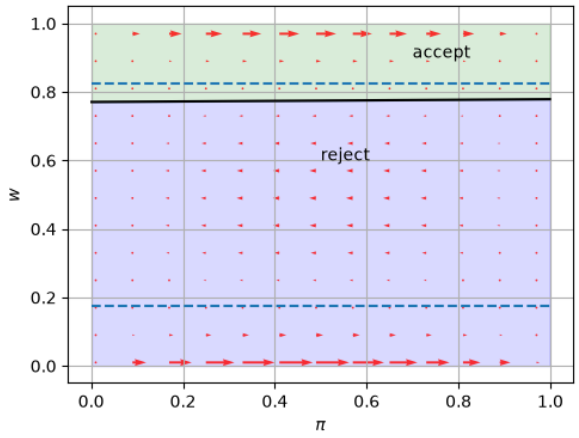
Now G has the same mean as F with a smaller variance.

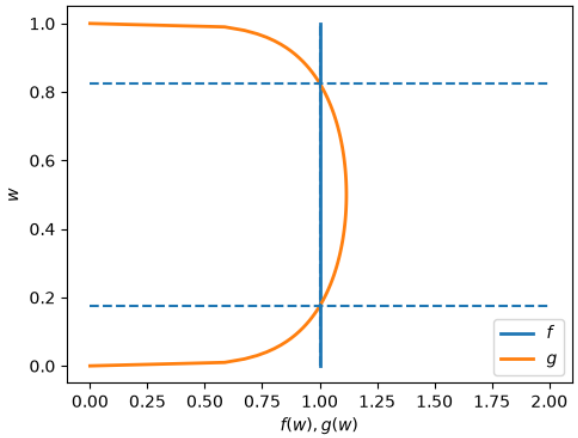
Since the unemployment compensation c serves as a lower bound for bad wage offers, G is now an "inferior" distribution to F.

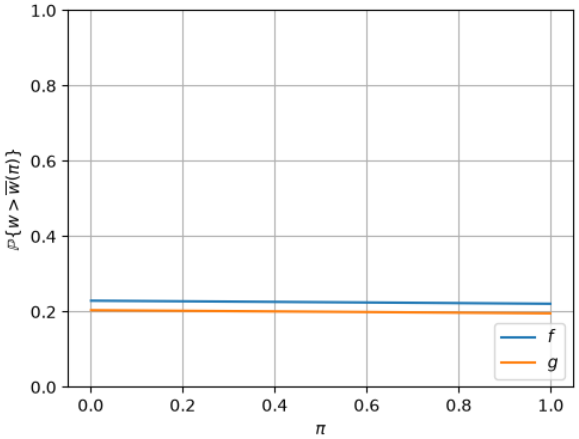
Consequently, we observe that the optimal policy $\overline{w}(\pi)$ is increasing in π.

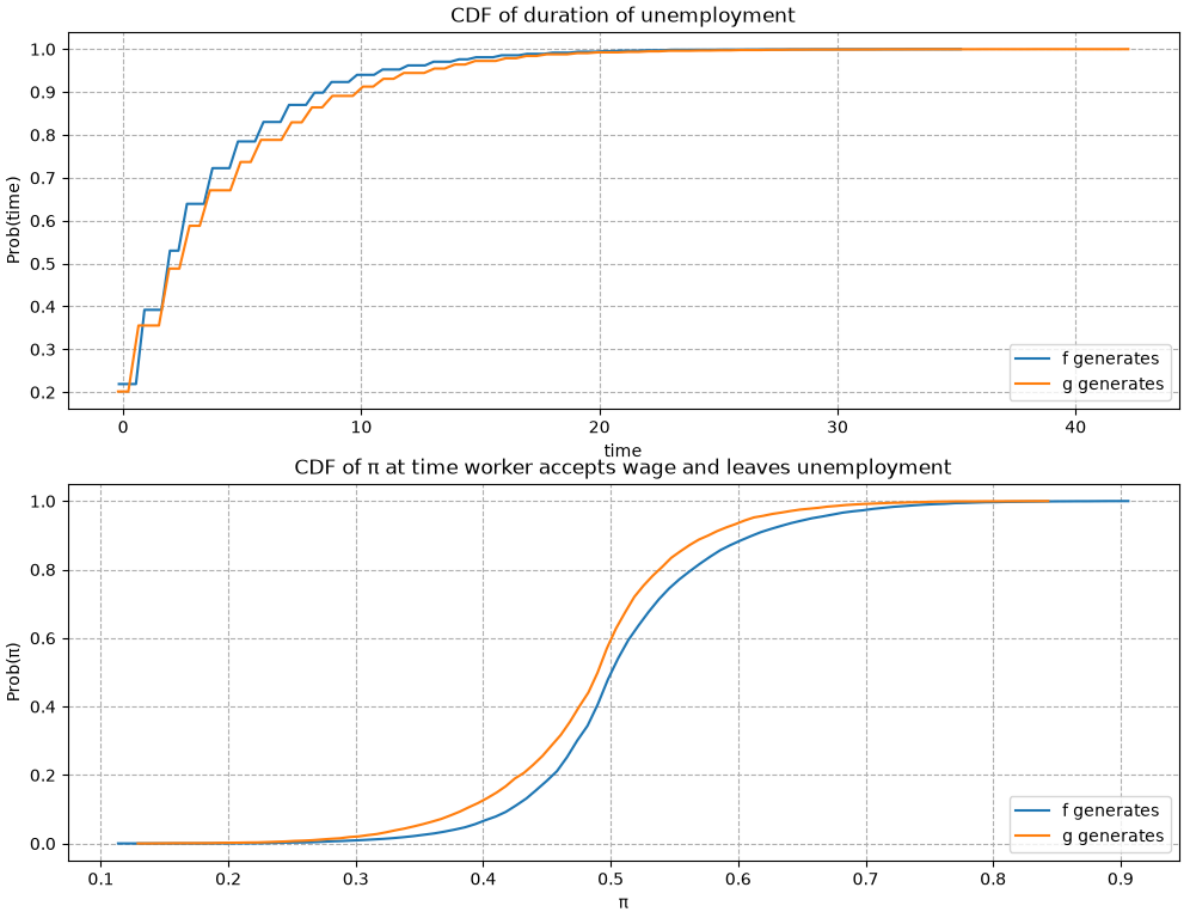
```
job_search_example(1, 1, 1.2, 1.2, 0.3)
```


w
l(w) = f(w)/g(w)
l


accept
reject
w
π


w
f(w), g(w)
f
g


$\mathbb{P}\{w > \bar{w}(\pi)\}$
π
f
g

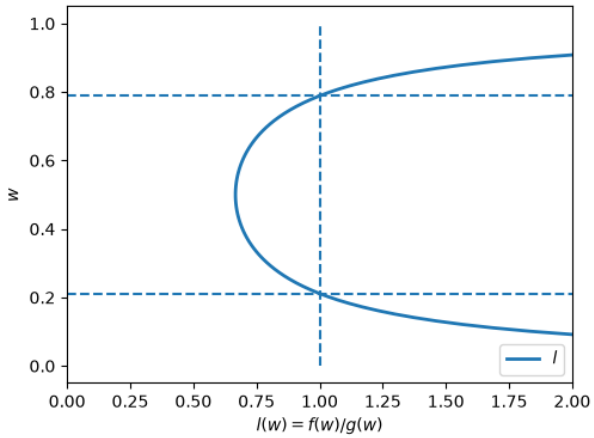


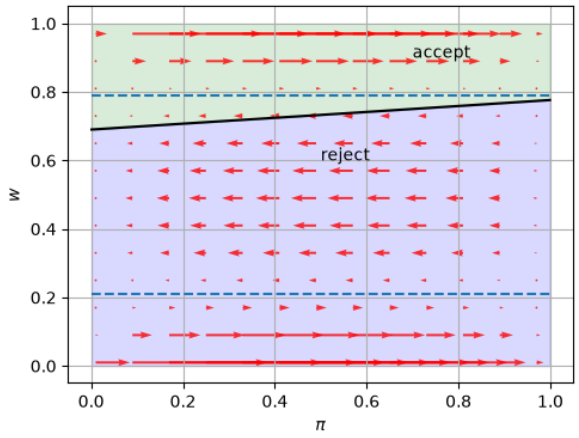
57.12.3 Example 3

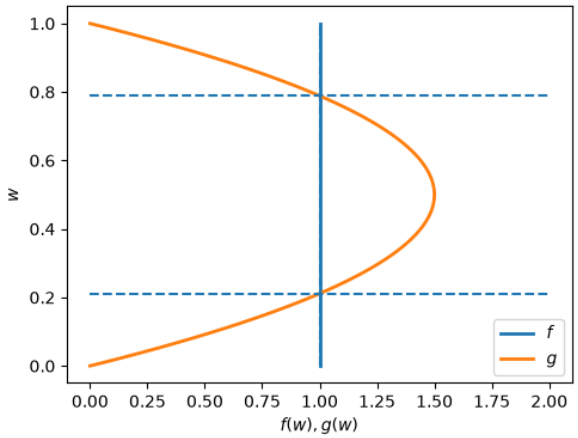
F ~ Beta(1, 1), G ~ Beta(2, 2), c=0.3.

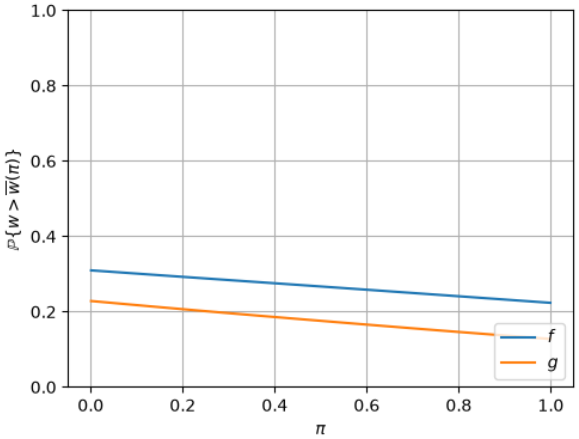
If the variance of G is smaller, we observe in the result that G is even more "inferior" and the slope of $\overline{w}(\pi)$ is larger.

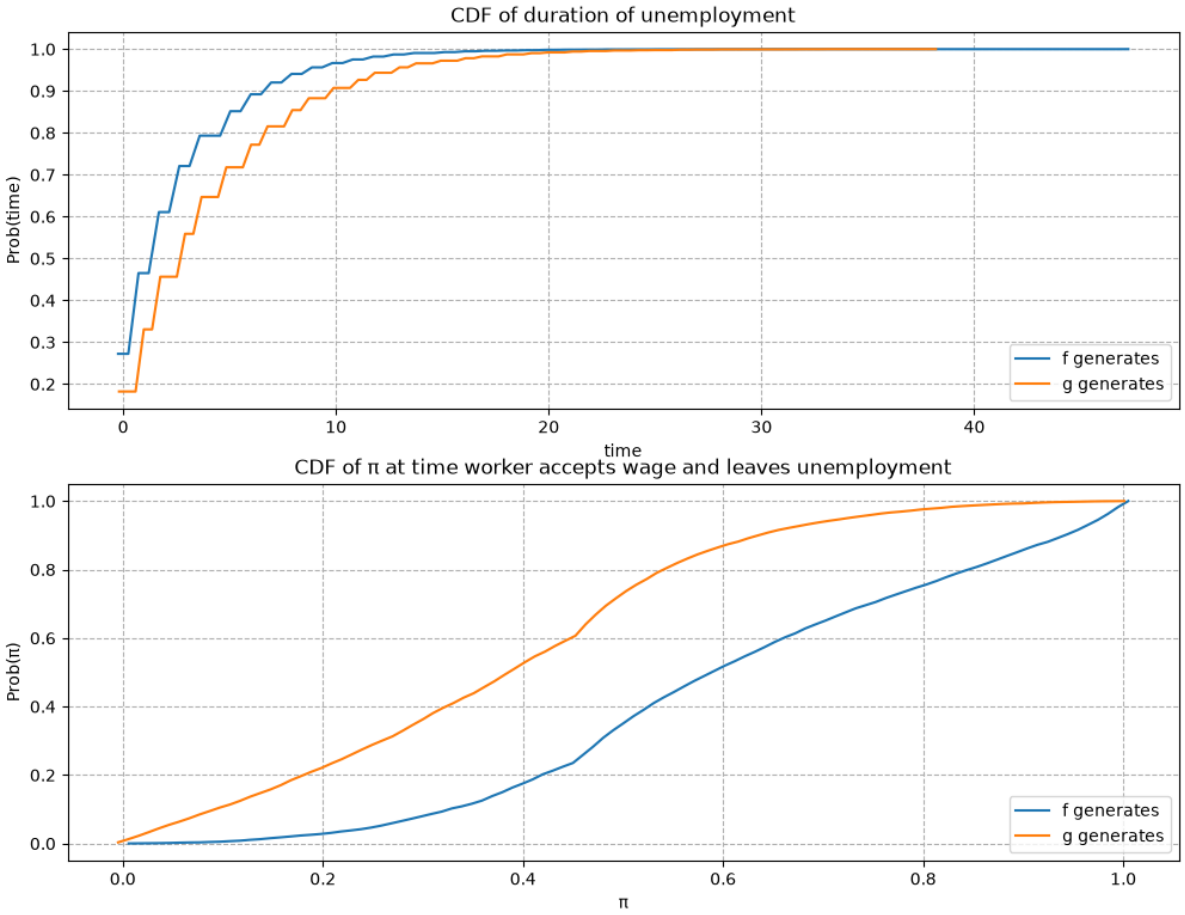
```
job_search_example(1, 1, 2, 2, 0.3)
```


w
l(w) = f(w)/g(w)
l


accept
reject
w
π


w
f(w), g(w)
f
g


$\mathbb{P}\{w > \bar{w}(\pi)\}$
π
f
g



57.12.4 Example 4

F ~ Beta(1, 1), G ~ Beta(3, 1.2), and c=0.8.

In this example, we keep the parameters of beta distributions to be the same with the baseline case but increase the unemployment compensation c.

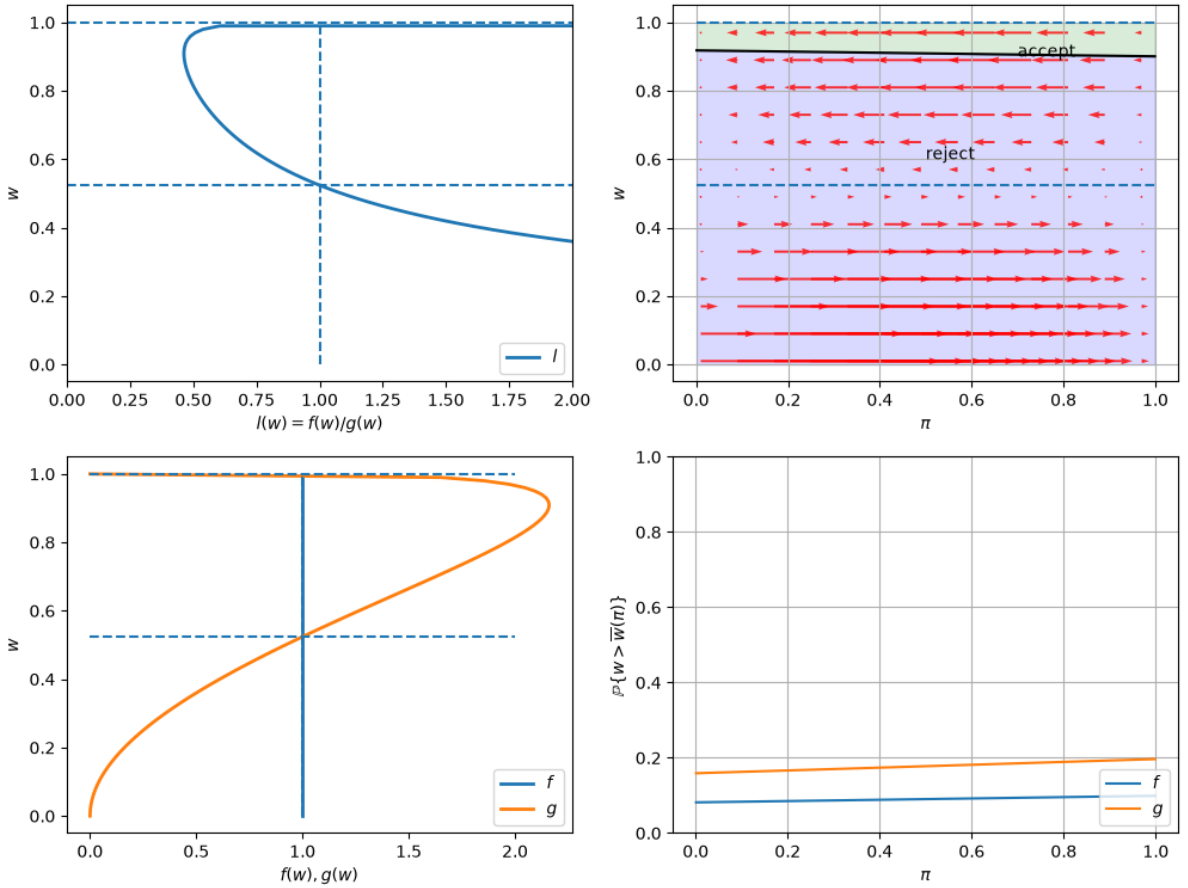
Comparing outcomes to the baseline case (example 1) in which unemployment compensation if low (c=0.3), now the worker can afford a longer learning period.

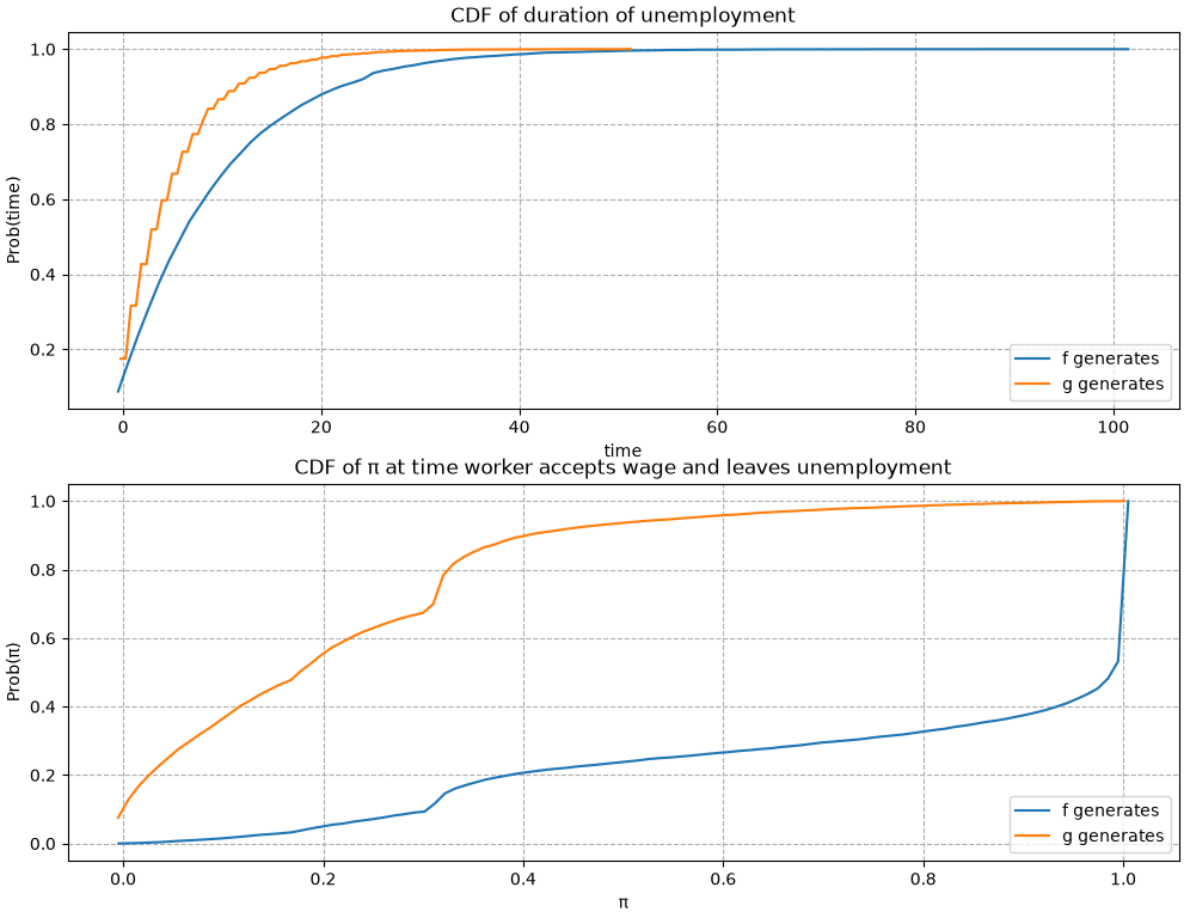
As a result, the worker tends to accept wage offers much later.

Furthermore, at the time of accepting employment, the belief π is closer to either 0 or 1.

That means that the worker has a better idea about what the true distribution is when he eventually chooses to accept a wage offer.

```
job_search_example(1, 1, 3, 1.2, c=0.8)
```



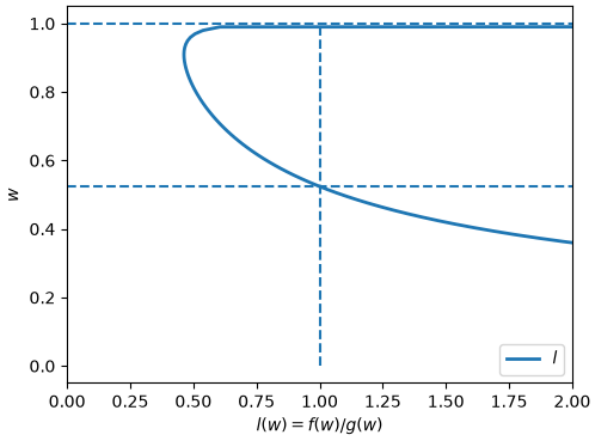


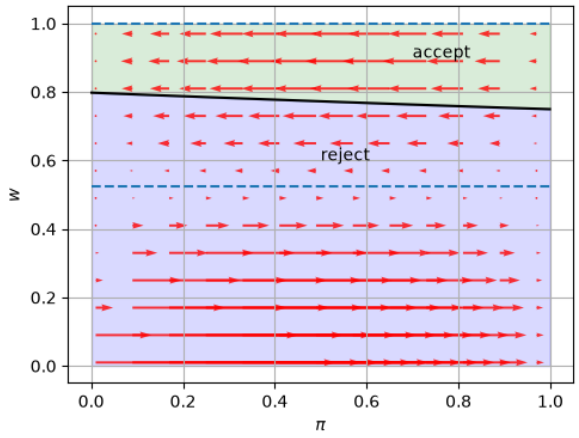
57.12.5 Example 5

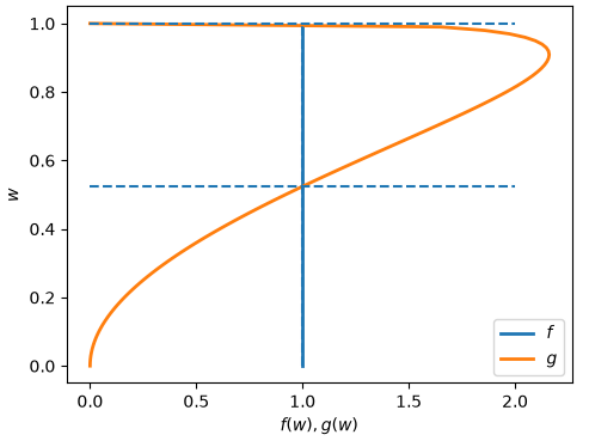
F ~ Beta(1, 1), G ~ Beta(3, 1.2), and c=0.1.

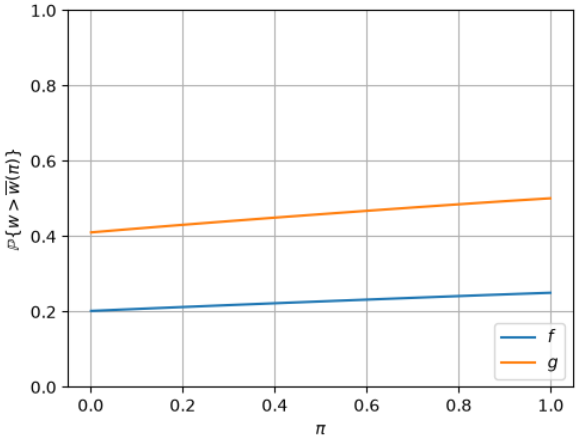
As expected, a smaller c makes an unemployed worker accept wage offers earlier after having acquired less information about the wage distribution.

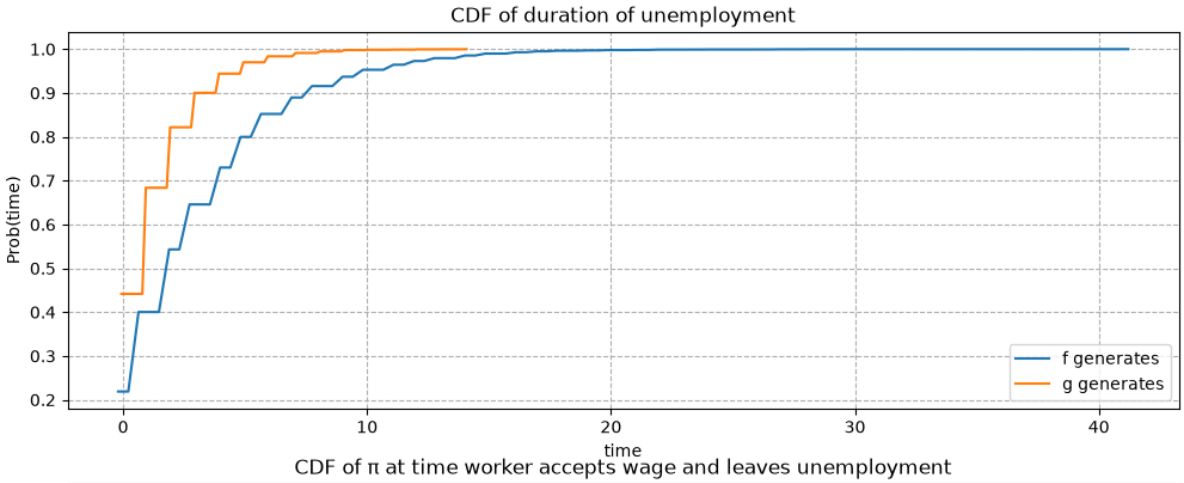
```
job_search_example(1, 1, 3, 1.2, c=0.1)
```

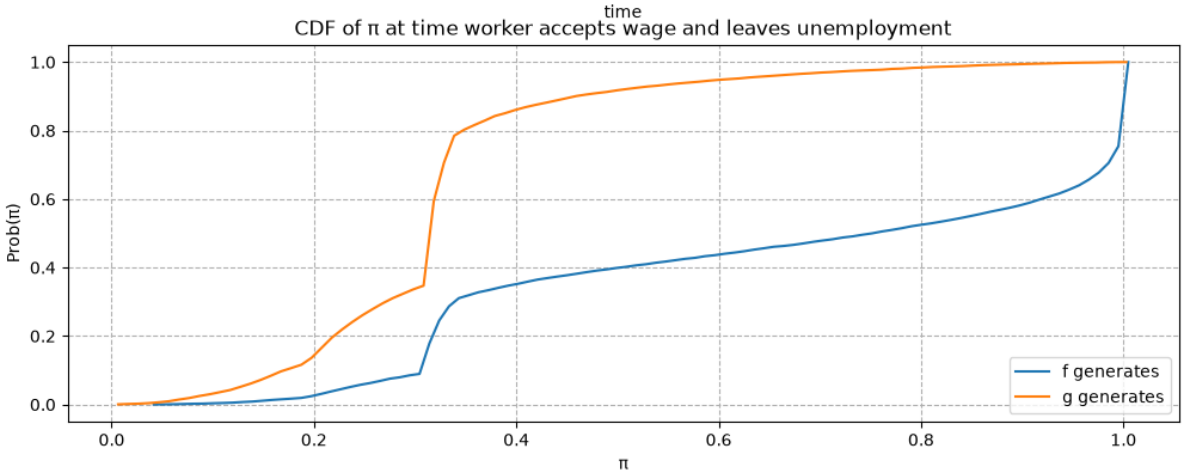

w
l(w) = f(w)/g(w)
l


accept
reject
w
π


w
f(w), g(w)
f
g


$\mathbb{P}\{w > \overline{w}(\pi)\}$
π
f
g


CDF of duration of unemployment
Prob(time)
time
f generates
g generates


CDF of π at time worker accepts wage and leaves unemployment
Prob(π)
π
f generates
g generates

Part VIII

Reinforcement Learning

CHAPTER

FIFTYEIGHT

INVENTORY MANAGEMENT VIA Q-LEARNING

Contents

- *Inventory Management via Q-Learning*
  - *Introduction*
  - *The Model*
  - *Solving via Value Function Iteration*
  - *Q-Learning*

58.1 Introduction

In this lecture we study a classic inventory management problem.

A firm must decide how much stock to order each period, facing uncertain demand and a trade-off between lost sales and ordering costs.

We approach the problem in two ways.

First, we solve it exactly using dynamic programming, assuming full knowledge of the model – the demand distribution, cost parameters, and transition dynamics.

Second, we show how a manager can learn the optimal policy from experience alone, using Q-learning.

In this setting, we assume that the manager observes only

- the inventory level,
- the order placed,
- the resulting profit, and
- the next inventory level.

The manager knows the interest rate – and hence the discount factor – but not any of the other underlying parameters.

A key idea is the *Q-factor* representation, which reformulates the Bellman equation so that the optimal policy can be recovered without knowledge of the transition dynamics.

We show that, given enough experience, the manager's learned policy converges to the optimal one.

The lecture proceeds as follows:

1. We set up the inventory model and solve it exactly via value function iteration.
2. We introduce Q-factors and derive the Q-factor Bellman equation.
3. We implement Q-learning and show the learned policy converges to the optimal one.

A risk-sensitive extension of this model is studied in *Risk-Sensitive Inventory Management via Q-Learning*.

We will use the following imports:

```
import numpy as np
import numba
import matplotlib.pyplot as plt
from typing import NamedTuple
```

58.2 The Model

We study a firm where a manager tries to maximize shareholder value by controlling inventories.

To simplify the problem, we assume that the firm only sells one product.

Letting π_t be profits at time t and $r > 0$ be the interest rate, the value of the firm is

$$V_0 = \mathbb{E} \sum_{t \geq 0} \beta^t \pi_t \qquad \text{where} \qquad \beta := \frac{1}{1+r}.$$

Suppose the firm faces exogenous demand process $(D_t)_{t \geq 0}$.

We assume $(D_t)_{t \geq 0}$ is IID with common distribution ϕ on $\{0, 1, ...\}$.

Inventory $(X_t)_{t \geq 0}$ of the product obeys

$$X_{t+1} = h(X_t, A_t, D_{t+1}) \qquad \text{where} \quad h(x, a, d) := \max(x - d, 0) + a.$$

The term A_t is units of stock ordered this period, which arrive at the start of period $t + 1$, after demand D_{t+1} is realized and served:

- observe $X_t \to$ choose $A_t \to$ demand D_{t+1} arrives $\to$ profit realized $\to X_{t+1}$ determined.

(We use a t subscript in A_t to indicate the information set: it is chosen before D_{t+1} is observed.)

We assume that the firm can store at most K items at one time.

Profits are given by

$$\pi(X_t, A_t, D_{t+1}) := \min(X_t, D_{t+1}) - cA_t - \kappa 1\{A_t > 0\}.$$

Here

- the sales price is set to unity (for convenience)
- revenue is the minimum of current stock and demand because orders in excess of inventory are lost (not back-filled)
- c is unit product cost and κ is a fixed cost of ordering inventory

We can map our inventory problem into a dynamic program with state space $\mathsf{X} := \{0, \ldots, K\}$ and action space $\mathsf{A} := \mathsf{X}$.

The feasible correspondence Γ is

$$\Gamma(x) := \{0, \ldots, K - x\}.$$

This represents the set of feasible orders when the current inventory state is x.

The Bellman equation takes the form

$$v(x) = \max_{a \in \Gamma(x)} \mathbb{E}\left[\pi(x, a, D) + \beta v(h(x, a, D))\right]. \tag{58.1}$$

Here D is a random variable with distribution ϕ.

58.3 Solving via Value Function Iteration

Let's start in a setting where the manager knows all parameters, functional forms, and distributions.

She solves the model numerically using value function iteration (VFI).

The idea is to iterate on the Bellman operator T defined by

$$(Tv)(x) = \max_{a \in \Gamma(x)} \sum_d \phi(d) \left[\pi(x, a, d) + \beta\, v(h(x, a, d))\right]$$

starting from an initial guess v_0.

Then, taking the output v from this iteration process, we compute a v**-greedy policy** σ, which obeys

$$\sigma(x) \in \arg\max_{a \in \Gamma(x)} \sum_d \phi(d) \left[\pi(x, a, d) + \beta\, v(h(x, a, d))\right]$$

When $r > 0$ (equivalently, $\beta < 1$), the sequence $v_{k+1} = Tv_k$ converges to the unique fixed point v^*, which is the value function of the optimal policy (see, e.g., [Sargent and Stachurski, 2025]).

58.3.1 Model specification

We store the model primitives in a `NamedTuple`.

Demand follows a geometric distribution with parameter p, so $\phi(d) = (1-p)^d\, p$ for $d = 0, 1, 2, \ldots$.

```
class Model(NamedTuple):
    x_values: np.ndarray        # Inventory values
    d_values: np.ndarray        # Demand values for summation
    ϕ_values: np.ndarray        # Demand probabilities
    p: float                    # Demand parameter
    c: float                    # Unit cost
    κ: float                    # Fixed cost
    β: float                    # Discount factor
```

The function below constructs a `Model` instance with default parameters.

We truncate the demand distribution at `D_MAX` for computational purposes.

```
def create_sdd_inventory_model(
        K: int = 20,          # Max inventory
        D_MAX: int = 21,      # Demand upper bound for summation
        p: float = 0.7,
        c: float = 0.2,
        κ: float = 0.8,
        β: float = 0.98
    ) -> Model:

    def demand_pdf(p, d):
```

(continues on next page)

(continued from previous page)

```
        return (1 - p)**d * p

    d_values = np.arange(D_MAX)
    ϕ_values = demand_pdf(p, d_values)      # ϕ_0, ϕ_1,...
    x_values = np.arange(K + 1)             # 0, 1, ..., K

    return Model(x_values, d_values, ϕ_values, p, c, κ, β)
```

58.3.2 The Bellman operator

The core computation is the Bellman operator T.

For each inventory level x, we loop over all feasible order quantities $a \in \{0, \ldots, K - x\}$ and compute the expected value

$$\sum_d \phi(d) \left[\min(x, d) - c\,a - \kappa\,\mathbf{1}\{a > 0\} + \beta\,v(\max(x - d, 0) + a)\right].$$

We then take the maximum over a.

The inner loops are compiled with Numba for performance.

```
@numba.jit(nopython=True)
def T_kernel(v, d_values, ϕ_values, c, κ, β, K):
    new_v = np.empty(K + 1)
    for x in range(K + 1):
        best = -np.inf
        for a in range(K - x + 1):          # loop over feasible actions
            val = 0.0
            for i in range(len(d_values)):   # compute expectation over demand
                d = d_values[i]
                x_next = max(x - d, 0) + a
                revenue = min(x, d)
                cost = c * a + κ * (a > 0)
                val += ϕ_values[i] * (revenue - cost + β * v[x_next])
            if val > best:
                best = val
        new_v[x] = best
    return new_v
```

The wrapper function `T` unpacks the model and calls the compiled kernel.

```
def T(v, model):
    """The Bellman operator."""
    x_values, d_values, ϕ_values, p, c, κ, β = model
    K = len(x_values) - 1
    return T_kernel(v, d_values, ϕ_values, c, κ, β, K)
```

58.3.3 Computing the greedy policy

Recall that, given a value function v, the **v-greedy policy** is computed via

$$\sigma(x) = \arg\max_{a \in \Gamma(x)} \sum_d \phi(d) \left[\pi(x, a, d) + \beta \, v(h(x, a, d)) \right].$$

The structure is the same as the Bellman operator, except we record the maximizing action rather than the maximized value.

```
@numba.jit(nopython=True)
def get_greedy_kernel(v, d_values, ϕ_values, c, κ, β, K):
    σ = np.empty(K + 1, dtype=np.int32)
    for x in range(K + 1):
        best = -np.inf
        best_a = 0
        for a in range(K - x + 1):
            val = 0.0
            for i in range(len(d_values)):
                d = d_values[i]
                x_next = max(x - d, 0) + a
                revenue = min(x, d)
                cost = c * a + κ * (a > 0)
                val += ϕ_values[i] * (revenue - cost + β * v[x_next])
            if val > best:
                best = val
                best_a = a
        σ[x] = best_a
    return σ
```

```
def get_greedy(v, model):
    """Get a v-greedy policy."""
    x_values, d_values, ϕ_values, p, c, κ, β = model
    K = len(x_values) - 1
    return get_greedy_kernel(v, d_values, ϕ_values, c, κ, β, K)
```

58.3.4 Value function iteration

We iterate $v_{k+1} = Tv_k$ until convergence, starting from $v_0 = 0$.

Once the value function has converged (to within tolerance `tol`), we extract the optimal policy σ^* via `get_greedy`.

```
def solve_inventory_model(v_init, model, max_iter=10_000, tol=1e-6):
    v = v_init.copy()
    i, error = 0, tol + 1

    while i < max_iter and error > tol:
        new_v = T(v, model)
        error = np.max(np.abs(new_v - v))
        i += 1
        v = new_v

    print(f"Converged in {i} iterations with error {error:.2e}")

    σ = get_greedy(v, model)
    return v, σ
```

58.3.5 Creating and solving an instance

```
model = create_sdd_inventory_model()
x_values, d_values, ϕ_values, p, c, κ, β = model
n_x = len(x_values)
v_init = np.zeros(n_x)
```

```
v_star, σ_star = solve_inventory_model(v_init, model)
```

```
Converged in 623 iterations with error 9.95e-07
```

58.3.6 Simulating the optimal policy

To visualize the solution, we simulate the inventory process under the optimal policy σ^*.

At each step, we draw a demand shock from the geometric distribution and update the state via h.

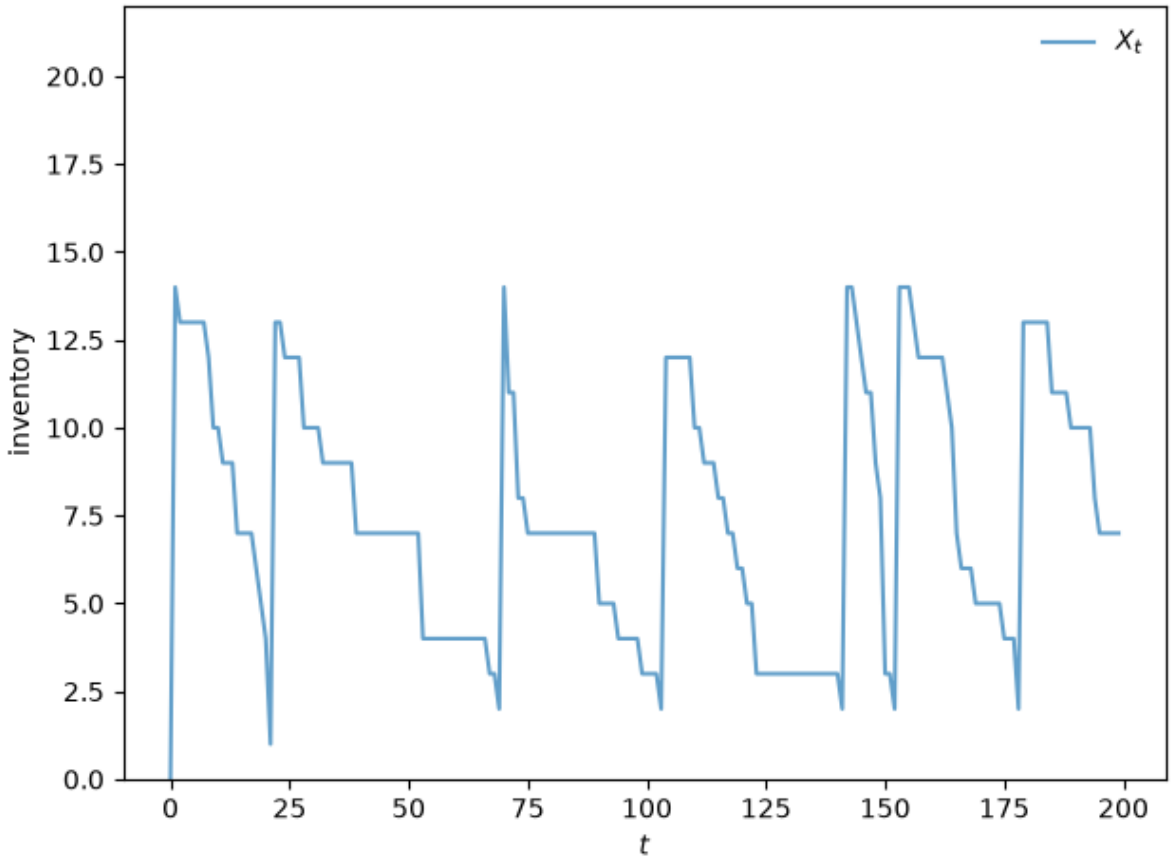
```
@numba.jit(nopython=True)
def sim_inventories(ts_length, σ, p, X_init=0, seed=0):
    """Simulate inventory dynamics under policy σ."""
    np.random.seed(seed)
    X = np.zeros(ts_length, dtype=np.int32)
    X[0] = X_init
    for t in range(ts_length - 1):
        d = np.random.geometric(p) - 1
        X[t+1] = max(X[t] - d, 0) + σ[X[t]]
    return X
```

The plot below shows a typical inventory path under the optimal policy.

Notice the **S-s pattern**: when inventory falls to a low level, the firm places a large order to replenish stock (the upward jumps), after which inventory gradually declines as demand is served.

```
def plot_ts(ts_length=200, fontsize=10):
    X = sim_inventories(ts_length, σ_star, p)
    fig, ax = plt.subplots()

    ax.plot(X, label=r"$X_t$", alpha=0.7)
    ax.set_xlabel(r"$t$", fontsize=fontsize)
    ax.set_ylabel("inventory", fontsize=fontsize)
    ax.legend(fontsize=fontsize, frameon=False)
    ax.set_ylim(0, len(σ_star) + 1)

    plt.tight_layout()
    plt.show()
```

```
plot_ts()
```



58.4 Q-Learning

We now ask: can an agent **learn** the optimal policy without knowing the model?

In particular, suppose the agent does not know the demand distribution ϕ, the cost parameters c and κ, or the transition function h.

Instead, the agent only observes the sequence of states, actions, and profits as it interacts with the environment.

58.4.1 The Q-factor Bellman equation

The first step of Q-learning is to modify the Bellman equation, placing it in a form that allows learning from this limited information.

Rather than working with the value function $v(x)$, we work with the **Q-function** (or Q-factor) $q(x, a)$.

We define q in terms of the value function v^* as

$$q(x, a) := \mathbb{E}\left[\pi(x, a, D) + \beta\, v^*(h(x, a, D))\right].$$

In words, $q(x, a)$ is the expected value of taking action a in state x and then following the optimal policy thereafter.

Note that the Bellman equation (58.1) can be written as

$$v^*(x) = \max_{a \in \Gamma(x)} q(x, a).$$

Substituting this back into the definition of q, we can eliminate v^* and obtain a fixed point equation in q alone:

$$q(x,a) = \mathbb{E}\left[\pi(x,a,D) + \beta \max_{a' \in \Gamma(x')} q(x',a')\right]$$

where $x' = h(x,a,D)$.

One advantage of working with q is that the optimal policy can be read off directly as $\sigma(x) = \arg\max_a q(x,a)$, without needing to know the transition function.

58.4.2 The Q-learning update rule

Q-learning approximates the fixed point of the Q-factor Bellman equation using **stochastic approximation**.

At each step, the agent is in state x, takes action a, observes reward $R_{t+1} = \pi(x,a,D_{t+1})$ and next state $X_{t+1} = h(x,a,D_{t+1})$, and updates

$$q_{t+1}(x,a) = (1-\alpha_t) q_t(x,a) + \alpha_t \left(R_{t+1} + \beta \max_{a' \in \Gamma(X_{t+1})} q_t(X_{t+1}, a')\right)$$

where α_t is the learning rate.

The update blends the current estimate $q_t(x,a)$ with a fresh sample of the Bellman target.

58.4.3 What the manager needs to know

Notice what is **not** required to implement the update.

The manager does not need to know the demand distribution ϕ, the unit cost c, the fixed cost κ, or the transition function h.

All the manager needs to observe at each step is:

1. the current inventory level x,
2. the order quantity a, which they choose,
3. the resulting profit R_{t+1} (which appears on the books),
4. the discount factor β, which is determined by the interest rate, and
5. the next inventory level X_{t+1} (which they can read off the warehouse).

These are all directly observable quantities – no model knowledge is required.

58.4.4 The Q-table and the role of the max

It is important to understand how the update rule relates to the manager's actions.

The manager maintains a **Q-table** – a lookup table storing an estimate $q_t(x,a)$ for every state-action pair (x,a).

At each step, the manager is in some state x and must choose a specific action a to take. Whichever a is chosen, the manager observes profit R_{t+1} and next state X_{t+1}, and updates *that one entry* $q_t(x,a)$ of the table using the rule above.

It is tempting to read the $\max_{a'}$ in the update rule as prescribing the manager's next action – that is, to interpret the update as saying "move to state X_{t+1} and take an action in $\arg\max_{a'} q_t(X_{t+1}, a')$."

But the max plays a different role.

The quantity $\max_{a' \in \Gamma(X_{t+1})} q_t(X_{t+1}, a')$ is just an estimate of the value of being in state X_{t+1} under the best possible continuation.

This scalar enters the update as part of the target value for $q_t(x, a)$.

Which action the manager *actually takes* at time $t+1$ is a separate decision.

In short, the max is doing the work of finding the optimum; it does not dictate the action that the manager actually takes.

58.4.5 The behavior policy

The rule governing how the manager chooses actions is called the **behavior policy**.

Because the max in the update target always points toward q^* regardless of how the manager selects actions, the behavior policy affects only which (x, a) entries get visited – and hence updated – over time.

In the reinforcement learning literature, this property is called **off-policy** learning: the convergence target (q^*) does not depend on the behavior policy.

As long as every (x, a) pair is visited infinitely often (so that every entry of the Q-table receives infinitely many updates) and the learning rates satisfy standard conditions (see below), the Q-table converges to q^*.

The behavior policy affects the *speed* of convergence – visiting important state-action pairs more frequently leads to faster learning – but not the *limit*.

In practice, we want the manager to mostly take good actions (to earn reasonable profits while learning), while still occasionally experimenting to discover better alternatives.

58.4.6 Learning rate

We use $\alpha_t = 1/n_t(x, a)^{0.51}$, where $n_t(x, a)$ is the number of times the pair (x, a) has been visited up to time t.

This decays slowly enough to allow learning from later (better-informed) updates, while still satisfying the Robbins–Monro conditions for convergence.

58.4.7 Exploration: epsilon-greedy

For our behavior policy, we use an ε-greedy strategy:

- With probability ε, choose a random feasible action (explore).
- With probability $1 - \varepsilon$, choose the action with the highest current q-value (exploit).

The exploration ensures that every state-action pair is visited, which is needed for convergence.

The exploitation ensures the manager earns reasonable profits while learning.

We decay ε each step: $\varepsilon_{t+1} = \max(\varepsilon_{\min},\ \varepsilon_t \cdot \lambda)$, so the manager experiments widely early on and increasingly relies on learned q-values as experience accumulates.

The stochastic demand shocks naturally drive the manager across different inventory levels, providing exploration over the state space without any artificial resets.

58.4.8 Optimistic initialization

A simple but powerful technique for accelerating learning is **optimistic initialization**: instead of starting the Q-table at zero, we initialize every entry to a value above the true optimum.

Because every untried action looks optimistically good, the agent is "disappointed" whenever it tries one – the update pulls that entry down toward reality. This drives the agent to try other actions (which still look optimistically high), producing broad exploration of the state-action space early in training.

This idea is sometimes called **optimism in the face of uncertainty** and is widely used in both bandit and reinforcement learning settings.

In our problem, the value function v^* ranges from about 13 to 18. We initialize the Q-table at 20 – modestly above the true maximum – to ensure optimistic exploration without being so extreme as to distort learning.

58.4.9 Implementation

We first define a helper to extract the greedy policy from a Q-table.

```
@numba.jit(nopython=True)
def greedy_policy_from_q(q, K):
    """Extract greedy policy from Q-table."""
    σ = np.empty(K + 1, dtype=np.int32)
    for x in range(K + 1):
        best_val = -np.inf
        best_a = 0
        for a in range(K - x + 1):
            if q[x, a] > best_val:
                best_val = q[x, a]
                best_a = a
        σ[x] = best_a
    return σ
```

The Q-learning loop runs for `n_steps` total steps in a single continuous trajectory – just as a real manager would learn from the ongoing stream of data.

At specified step counts (given by `snapshot_steps`), we record the current greedy policy.

```
@numba.jit(nopython=True)
def q_learning_kernel(K, p, c, κ, β, n_steps, X_init,
                      ε_init, ε_min, ε_decay, q_init, snapshot_steps, seed):
    np.random.seed(seed)
    q = np.full((K + 1, K + 1), q_init)
    n = np.zeros((K + 1, K + 1))       # visit counts for learning rate
    ε = ε_init

    n_snaps = len(snapshot_steps)
    snapshots = np.zeros((n_snaps, K + 1), dtype=np.int32)
    snap_idx = 0

    # Initialize state and action
    x = X_init
    a = np.random.randint(0, K - x + 1)

    for t in range(n_steps):
        # Record policy snapshot if needed
        if snap_idx < n_snaps and t == snapshot_steps[snap_idx]:
```

(continues on next page)

(continued from previous page)

```
            snapshots[snap_idx] = greedy_policy_from_q(q, K)
            snap_idx += 1

        # === Draw D_{t+1} and observe outcome ===
        d = np.random.geometric(p) - 1
        reward = min(x, d) - c * a - κ * (a > 0)
        x_next = max(x - d, 0) + a

        # === Max over next state (scalar value for update target) ===
        # Also record the argmax action for use by the behavior policy.
        best_next = -np.inf
        a_next = 0
        for aa in range(K - x_next + 1):
            if q[x_next, aa] > best_next:
                best_next = q[x_next, aa]
                a_next = aa

        # === Q-learning update (uses best_next, the max value) ===
        n[x, a] += 1
        α = 1.0 / n[x, a] ** 0.51
        q[x, a] = (1 - α) * q[x, a] + α * (reward + β * best_next)

        # === Behavior policy: ε-greedy (uses a_next, the argmax action) ===
        x = x_next
        if np.random.random() < ε:
            a = np.random.randint(0, K - x + 1)
        else:
            a = a_next
        ε = max(ε_min, ε * ε_decay)

    return q, snapshots
```

The wrapper function unpacks the model and provides default hyperparameters.

```
def q_learning(model, n_steps=20_000_000, X_init=0,
               ε_init=1.0, ε_min=0.01, ε_decay=0.999999,
               q_init=20.0, snapshot_steps=None, seed=1234):
    x_values, d_values, ϕ_values, p, c, κ, β = model
    K = len(x_values) - 1
    if snapshot_steps is None:
        snapshot_steps = np.array([], dtype=np.int64)
    return q_learning_kernel(K, p, c, κ, β, n_steps, X_init,
                             ε_init, ε_min, ε_decay, q_init, snapshot_steps, seed)
```

Next we run n = 5 million steps and take policy snapshots at steps 10,000, 1,000,000, and n.

```
n = 5_000_000
snap_steps = np.array([10_000, 1_000_000, n], dtype=np.int64)
q, snapshots = q_learning(model, n_steps=n+1, snapshot_steps=snap_steps)
```

58.4.10 Comparing with the exact solution

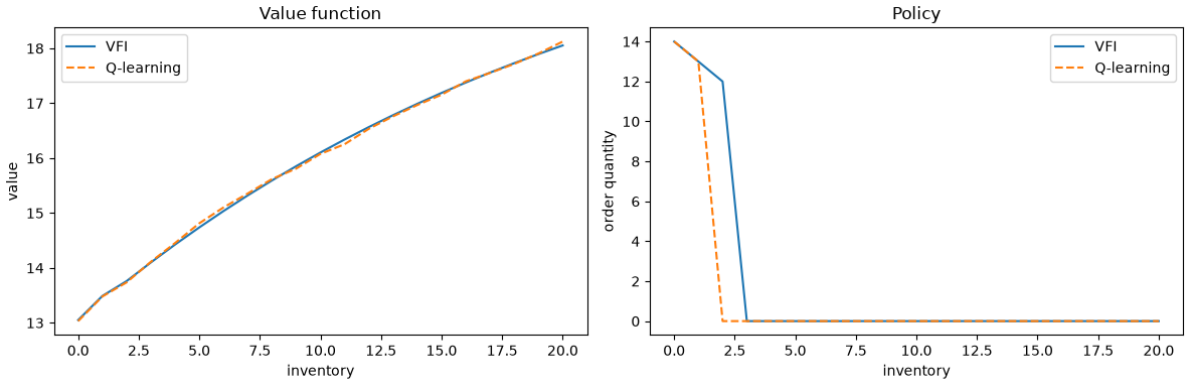
We extract the value function and policy from the final Q-table via

$$v_Q(x) = \max_{a \in \Gamma(x)} q(x,a), \qquad \sigma_Q(x) = \arg\max_{a \in \Gamma(x)} q(x,a)$$

and compare them against v^* and σ^* from VFI.

```
K = len(x_values) - 1
# restrict to feasible actions a ∈ {0, ..., K-x}
v_q = np.array([np.max(q[x, :K - x + 1]) for x in range(K + 1)])
σ_q = np.array([np.argmax(q[x, :K - x + 1]) for x in range(K + 1)])
```

```
fig, axes = plt.subplots(1, 2, figsize=(12, 4))

axes[0].plot(x_values, v_star, label="VFI")
axes[0].plot(x_values, v_q, '--', label="Q-learning")
axes[0].set_xlabel("inventory")
axes[0].set_ylabel("value")
axes[0].legend()
axes[0].set_title("Value function")

axes[1].plot(x_values, σ_star, label="VFI")
axes[1].plot(x_values, σ_q, '--', label="Q-learning")
axes[1].set_xlabel("inventory")
axes[1].set_ylabel("order quantity")
axes[1].legend()
axes[1].set_title("Policy")

plt.tight_layout()
plt.show()
```



58.4.11 Visualizing learning over time

The panels below show how the agent's behavior evolves during training.

Each panel simulates an inventory path using the greedy policy extracted from the Q-table at a given training step.

All panels use the **same demand sequence** (via a fixed random seed), so differences reflect only changes in the policy.

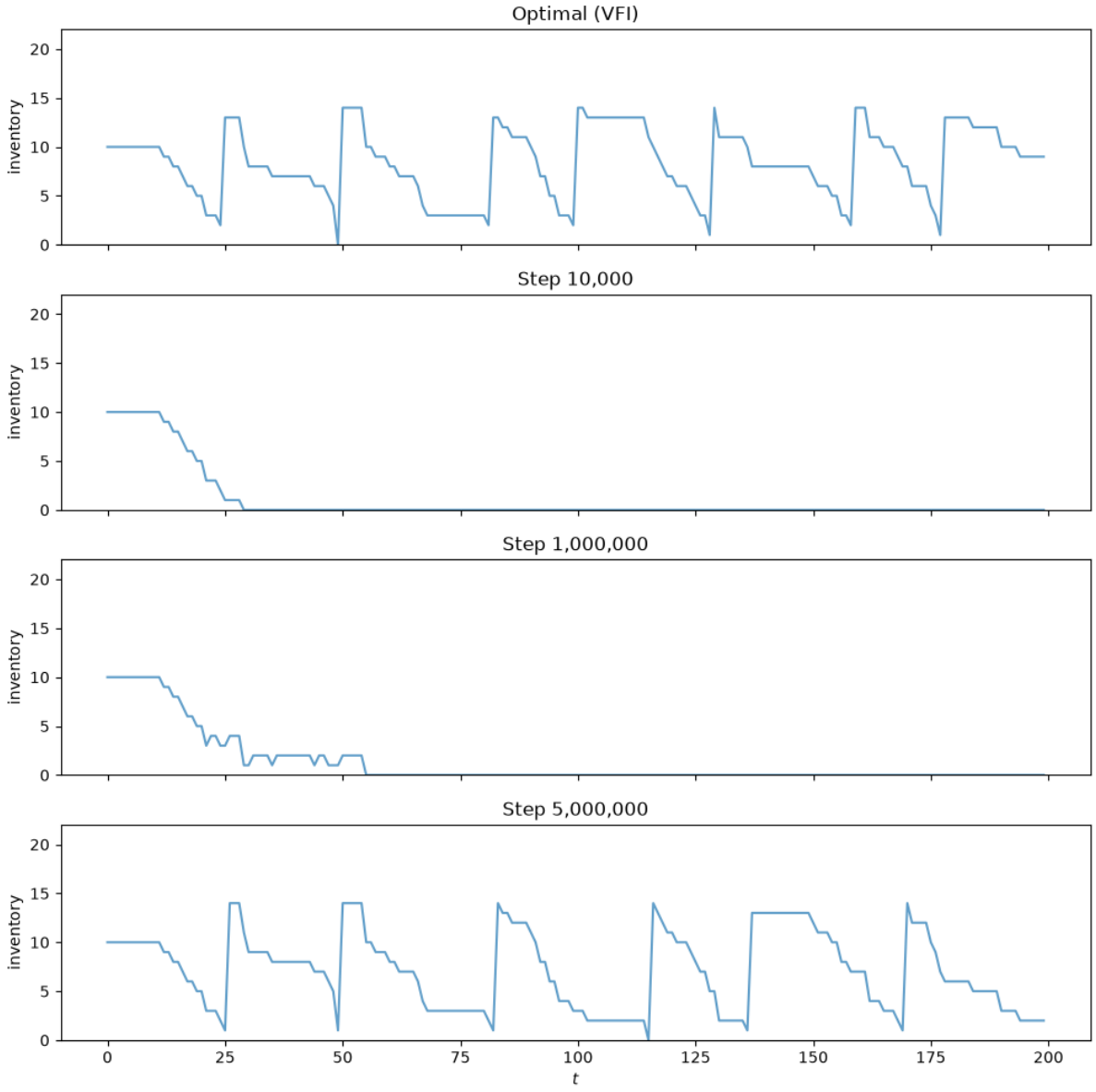
The top panel shows the optimal policy from VFI for reference.

After 10,000 steps the agent has barely explored and its policy is poor.

By 1,000,000 steps the policy has improved but still differs noticeably from the optimum.

By step 5 million, the learned policy produces inventory dynamics that closely resemble the S-s pattern of the optimal solution.

```
ts_length = 200
n_snaps = len(snap_steps)
fig, axes = plt.subplots(n_snaps + 1, 1, figsize=(10, 2.5 * (n_snaps + 1)),
                         sharex=True)

X_init = K // 2
sim_seed = 5678

# Optimal policy
X_opt = sim_inventories(ts_length, σ_star, p, X_init, seed=sim_seed)
axes[0].plot(X_opt, alpha=0.7)
axes[0].set_ylabel("inventory")
axes[0].set_title("Optimal (VFI)")
axes[0].set_ylim(0, K + 2)

# Q-learning snapshots
for i in range(n_snaps):
    σ_snap = snapshots[i]
    X = sim_inventories(ts_length, σ_snap, p, X_init, seed=sim_seed)
    axes[i + 1].plot(X, alpha=0.7)
    axes[i + 1].set_ylabel("inventory")
    axes[i + 1].set_title(f"Step {snap_steps[i]:,}")
    axes[i + 1].set_ylim(0, K + 2)

axes[-1].set_xlabel(r"$t$")
plt.tight_layout()
plt.show()
```


Optimal (VFI)
Step 10,000
Step 1,000,000
Step 5,000,000
inventory
0
5
10
15
20
0
25
50
75
100
125
150
175
200
t

CHAPTER

FIFTYNINE

RISK-SENSITIVE INVENTORY MANAGEMENT VIA Q-LEARNING

Contents

- *Risk-Sensitive Inventory Management via Q-Learning*
  - *Introduction*
  - *The Model*
  - *Solving via Value Function Iteration*
  - *Interpreting the Outcomes*
  - *Q-Learning*
  - *Conclusion*

59.1 Introduction

In *Inventory Management via Q-Learning*, we looked at an inventory management problem and solved it with both value function iteration and Q-learning.

In this lecture, we consider a risk-sensitive variation.

Injection of risk-sensitivity acknowledges the fact that, in incomplete markets with financial and informational frictions, firms typically take risk into account in their decision making.

In other words, the actions of firms are not, in general, risk neutral.

One natural way to handle this is to use a risk-sensitive version of the Bellman equation.

We show how the model can be solved using value function iteration.

We then investigate how risk sensitivity affects the optimal policy.

```
import numpy as np
import numba
import matplotlib.pyplot as plt
from typing import NamedTuple
```

59.2 The Model

The Bellman equation for the inventory management problem in *Inventory Management via Q-Learning* has the form

$$v(x) = \max_{a \in \Gamma(x)} \mathbb{E}\left[\pi(x, a, D) + \beta v(h(x, a, D))\right].$$

Here D is a random variable with distribution ϕ.

(Primitives and definitions are the same as in *Inventory Management via Q-Learning*.)

The risk-sensitive version of this Bellman equation has the form

$$v(x) = \max_{a \in \Gamma(x)} \psi^{-1}\left\{\mathbb{E}\psi\left[\pi(x, a, D) + \beta v(h(x, a, D))\right]\right\},$$

where $\psi(t) = \exp(-\gamma t)$ for fixed $\gamma > 0$.

Since $\psi^{-1}(y) = -\frac{1}{\gamma}\ln(y)$, the Bellman equation becomes

$$v(x) = \max_{a \in \Gamma(x)} \left(-\frac{1}{\gamma}\right) \ln\left\{\sum_d \phi(d) \exp\left[-\gamma\left(\pi(x, a, d) + \beta\, v(h(x, a, d))\right)\right]\right\}.$$

Here $\phi(d)$ denotes the demand probability mass function, as in *Inventory Management via Q-Learning*.

The parameter γ controls the degree of risk sensitivity.

As $\gamma \to 0$, the certainty equivalent reduces to the ordinary expectation and we recover the risk-neutral case.

Larger γ means more aversion to downside risk.

The Bellman operator, greedy policy, and VFI algorithm all carry over from the risk-neutral case, with the expectation replaced by the certainty equivalent.

59.3 Solving via Value Function Iteration

59.3.1 Model specification

We reuse the same model primitives as in *Inventory Management via Q-Learning*, adding γ as a parameter.

```
class RSModel(NamedTuple):
    x_values: np.ndarray        # Inventory values
    d_values: np.ndarray        # Demand values for summation
    ϕ_values: np.ndarray        # Demand probabilities
    p: float                    # Demand parameter
    c: float                    # Unit cost
    κ: float                    # Fixed cost
    β: float                    # Discount factor
    γ: float                    # Risk-sensitivity parameter
```

```
def create_rs_inventory_model(
        K: int = 20,          # Max inventory
        D_MAX: int = 21,      # Demand upper bound for summation
        p: float = 0.7,
        c: float = 0.2,
        κ: float = 0.8,
        β: float = 0.98,
```

(continues on next page)

(continued from previous page)

```
        γ: float = 1.0
    ) -> RSModel:

    def demand_pdf(p, d):
        return (1 - p)**d * p

    d_values = np.arange(D_MAX)
    ϕ_values = demand_pdf(p, d_values)
    x_values = np.arange(K + 1)

    return RSModel(x_values, d_values, ϕ_values, p, c, κ, β, γ)
```

59.3.2 The Bellman operator

The risk-sensitive Bellman operator replaces the expected value with the certainty equivalent.

For numerical stability, we use the log-sum-exp trick: given values $z_i = \pi(x, a, d_i) + \beta\, v(h(x, a, d_i))$, we compute

$$-\frac{1}{\gamma} \ln \sum_i \phi(d_i) \exp(-\gamma z_i) \;=\; -\frac{1}{\gamma}\left(m + \ln \sum_i \phi(d_i) \exp(-\gamma z_i - m) \right),$$

where $m = \max_i(-\gamma z_i)$.

```
@numba.jit(nopython=True)
def T_rs_kernel(v, d_values, ϕ_values, c, κ, β, γ, K):
    new_v = np.empty(K + 1)
    n_d = len(d_values)
    for x in range(K + 1):
        best = -np.inf
        for a in range(K - x + 1):
            # Compute -γ * z_i for each demand realization
            exponents = np.empty(n_d)
            for i in range(n_d):
                d = d_values[i]
                x_next = max(x - d, 0) + a
                revenue = min(x, d)
                cost = c * a + κ * (a > 0)
                z_i = revenue - cost + β * v[x_next]
                exponents[i] = -γ * z_i
            # Log-sum-exp trick for numerical stability
            m = np.max(exponents)
            weighted_sum = 0.0
            for i in range(n_d):
                weighted_sum += ϕ_values[i] * np.exp(exponents[i] - m)
            val = -(1.0 / γ) * (m + np.log(weighted_sum))
            if val > best:
                best = val
        new_v[x] = best
    return new_v
```

```
def T_rs(v, model):
    """The risk-sensitive Bellman operator."""
    x_values, d_values, ϕ_values, p, c, κ, β, γ = model
    K = len(x_values) - 1
    return T_rs_kernel(v, d_values, ϕ_values, c, κ, β, γ, K)
```

59.3.3 Computing the greedy policy

The greedy policy records the maximizing action instead of the maximized value.

```
@numba.jit(nopython=True)
def get_greedy_rs_kernel(v, d_values, ϕ_values, c, κ, β, γ, K):
    σ = np.empty(K + 1, dtype=np.int32)
    n_d = len(d_values)
    for x in range(K + 1):
        best = -np.inf
        best_a = 0
        for a in range(K - x + 1):
            exponents = np.empty(n_d)
            for i in range(n_d):
                d = d_values[i]
                x_next = max(x - d, 0) + a
                revenue = min(x, d)
                cost = c * a + κ * (a > 0)
                z_i = revenue - cost + β * v[x_next]
                exponents[i] = -γ * z_i
            m = np.max(exponents)
            weighted_sum = 0.0
            for i in range(n_d):
                weighted_sum += ϕ_values[i] * np.exp(exponents[i] - m)
            val = -(1.0 / γ) * (m + np.log(weighted_sum))
            if val > best:
                best = val
                best_a = a
        σ[x] = best_a
    return σ
```

```
def get_greedy_rs(v, model):
    """Get a v-greedy policy for the risk-sensitive model."""
    x_values, d_values, ϕ_values, p, c, κ, β, γ = model
    K = len(x_values) - 1
    return get_greedy_rs_kernel(v, d_values, ϕ_values, c, κ, β, γ, K)
```

59.3.4 Value function iteration

```
def solve_rs_inventory_model(v_init, model, max_iter=10_000, tol=1e-6):
    v = v_init.copy()
    i, error = 0, tol + 1

    while i < max_iter and error > tol:
        new_v = T_rs(v, model)
        error = np.max(np.abs(new_v - v))
        i += 1
        v = new_v

    print(f"Converged in {i} iterations with error {error:.2e}")

    σ = get_greedy_rs(v, model)
    return v, σ
```

59.3.5 Creating and solving an instance

```
model = create_rs_inventory_model()
x_values = model.x_values
n_x = len(x_values)
v_init = np.zeros(n_x)
```

```
v_star, σ_star = solve_rs_inventory_model(v_init, model)
```

```
Converged in 600 iterations with error 9.92e-07
```

59.3.6 Effect of risk sensitivity on the optimal policy

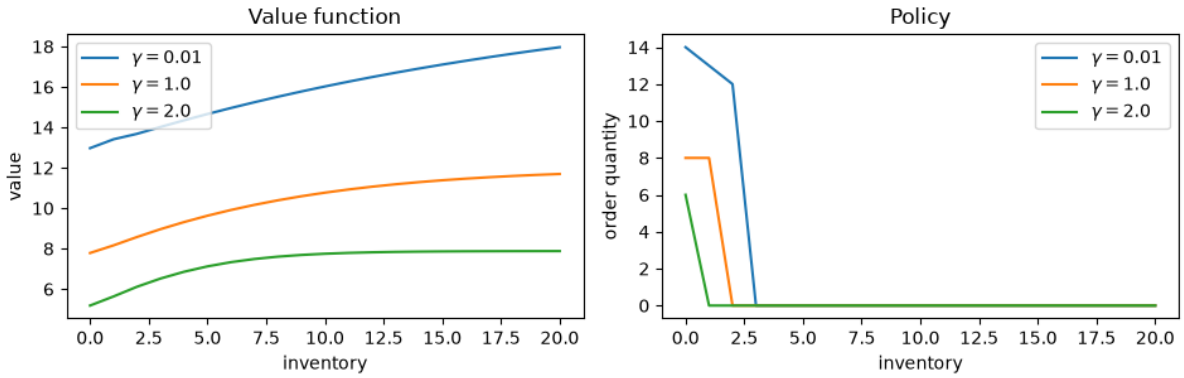
We solve the model for several values of γ and compare the resulting policies.

As we will see, a risk-sensitive firm orders less aggressively than a risk-neutral one.

```
γ_values = [0.01, 1.0, 2.0]
results = {}

for γ in γ_values:
    mod = create_rs_inventory_model(γ=γ)
    v, σ = solve_rs_inventory_model(np.zeros(n_x), mod)
    results[γ] = (v, σ)
```

```
Converged in 623 iterations with error 9.90e-07
Converged in 600 iterations with error 9.92e-07
Converged in 583 iterations with error 9.96e-07
```

```
fig, axes = plt.subplots(1, 2, figsize=(9.6, 3.2))

for γ in γ_values:
    v, σ = results[γ]
    axes[0].plot(x_values, v, label=f"$\\gamma = {γ}$")
    axes[1].plot(x_values, σ, label=f"$\\gamma = {γ}$")

axes[0].set_xlabel("inventory")
axes[0].set_ylabel("value")
axes[0].legend()
axes[0].set_title("Value function")

axes[1].set_xlabel("inventory")
axes[1].set_ylabel("order quantity")
axes[1].legend()
axes[1].set_title("Policy")

plt.tight_layout()
plt.show()
```

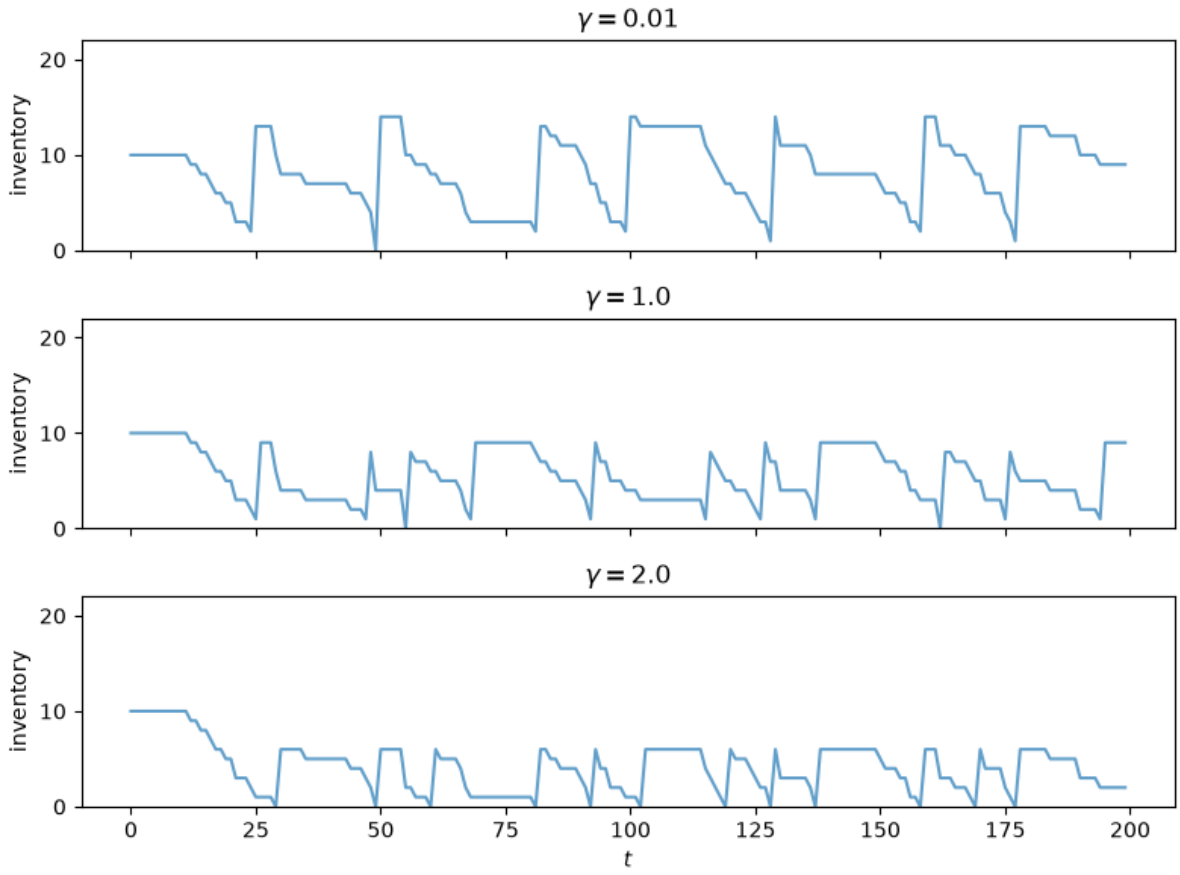


59.3.7 Simulating the optimal policy

We simulate inventory dynamics under the optimal policy for the baseline γ.

```
@numba.jit(nopython=True)
def sim_inventories(ts_length, σ, p, X_init=0, seed=0):
    """Simulate inventory dynamics under policy σ."""
    np.random.seed(seed)
    X = np.zeros(ts_length, dtype=np.int32)
    X[0] = X_init
    for t in range(ts_length - 1):
        d = np.random.geometric(p) - 1
        X[t+1] = max(X[t] - d, 0) + σ[X[t]]
    return X
```

```
fig, axes = plt.subplots(len(γ_values), 1,
                         figsize=(8, 2.0 * len(γ_values)),
                         sharex=True)

ts_length = 200
sim_seed = 5678
K = len(x_values) - 1

for i, γ in enumerate(γ_values):
    v, σ = results[γ]
    X = sim_inventories(ts_length, σ, model.p, X_init=K // 2, seed=sim_seed)
    axes[i].plot(X, alpha=0.7)
    axes[i].set_ylabel("inventory")
    axes[i].set_title(f"$\\gamma = {γ}$")
    axes[i].set_ylim(0, K + 2)

axes[-1].set_xlabel(r"$t$")
plt.tight_layout()
plt.show()
```



59.4 Interpreting the Outcomes

The plots above show that a more risk-sensitive firm (larger γ) orders less inventory and maintains lower stock levels.

At first glance this may seem surprising: wouldn't holding more inventory reduce variance by ensuring the firm can always meet demand?

The key is to identify where the randomness in profits actually comes from.

Recall that per-period profit is $\pi(x, a, d) = \min(x, d) - ca - \kappa \mathbf{1}\{a > 0\}$.

The ordering cost $ca + \kappa \mathbf{1}\{a > 0\}$ is **deterministic** – it is chosen before the demand shock is realized.

So higher ordering shifts the level of profits down but does not affect their variance.

The variance comes from **revenue**: $\min(x, D)$.

When inventory x is high, $\min(x, D) \approx D$ for most demand realizations – revenue inherits the full variance of demand.

When inventory x is low, $\min(x, D) \approx x$ for most realizations – revenue is nearly deterministic, capped at the inventory level.

A risk-sensitive agent therefore prefers lower inventory because it **caps the randomness of revenue**.

The agent accepts lower expected sales in exchange for more predictable profits.

There is also a continuation value channel: next-period inventory $\max(x - D, 0) + a$ varies with D, and higher x means $x - D$ tracks D more closely, propagating that variance forward through v.

59.5 Q-Learning

We now ask whether the optimal policy can be learned without knowledge of the model, as we did in the risk-neutral case in *Inventory Management via Q-Learning*.

59.5.1 The Q-factor

The first step is to define the Q-factor in a way that is compatible with the risk-sensitive Bellman equation.

We define

$$q(x,a) := \mathbb{E}\left[\exp(-\gamma\,(\pi(x,a,D) + \beta\, v^*(h(x,a,D))))\right].$$

In words, $q(x,a)$ applies the risk-sensitivity transformation $\psi(t) = \exp(-\gamma t)$ inside the expectation, evaluated at the return from taking action a in state x and following the optimal policy thereafter.

59.5.2 Deriving the Q-factor Bellman equation

Our goal is to obtain a fixed point equation in q alone, eliminating v^*.

Step 1. Express v^* in terms of q.

The risk-sensitive Bellman equation says $v^*(x) = \max_{a\in\Gamma(x)} \psi^{-1}(q(x,a))$.

Since $\psi^{-1}(y) = -\frac{1}{\gamma}\ln(y)$ is a **decreasing** function, the maximum over a of $\psi^{-1}(q(x,a))$ corresponds to the **minimum** over a of $q(x,a)$:

$$v^*(x) = \psi^{-1}\left(\min_{a\in\Gamma(x)} q(x,a)\right) = -\frac{1}{\gamma}\ln\left(\min_{a\in\Gamma(x)} q(x,a)\right).$$

Equivalently,

$$\exp(-\gamma\, v^*(x)) = \min_{a\in\Gamma(x)} q(x,a).$$

Step 2. Substitute back into the definition of q to eliminate v^*.

Expanding the exponential in the definition of q,

$$q(x,a) = \mathbb{E}\left[\exp(-\gamma\,\pi(x,a,D)) \;\cdot\; \exp(-\gamma\beta\, v^*(x'))\right]$$

where $x' = h(x,a,D)$.

From Step 1, $\exp(-\gamma\, v^*(x')) = \min_{a'\in\Gamma(x')} q(x',a')$, so $\exp(-\gamma\beta\, v^*(x')) = \left[\min_{a'\in\Gamma(x')} q(x',a')\right]^{\beta}$.

Substituting,

$$q(x,a) = \mathbb{E}\left[\exp(-\gamma\,\pi(x,a,D)) \;\cdot\; \left(\min_{a'\in\Gamma(x')} q(x',a')\right)^{\beta}\right].$$

This is a fixed point equation in q alone – v^* has been eliminated.

59.5.3 The Q-learning update rule

As in the risk-neutral case, we approximate the fixed point using stochastic approximation.

At each step, the agent is in state x, takes action a, observes profit $R_{t+1} = \pi(x, a, D_{t+1})$ and next state $X_{t+1} = h(x, a, D_{t+1})$, and updates

$$q_{t+1}(x,a) = (1-\alpha_t)\, q_t(x,a) + \alpha_t \left[\exp(-\gamma\, R_{t+1}) \cdot \left(\min_{a' \in \Gamma(X_{t+1})} q_t(X_{t+1}, a') \right)^{\beta} \right].$$

The term in brackets is a single-sample estimate of the right-hand side of the Q-factor Bellman equation.

The update blends the current estimate with this fresh sample, just as in standard Q-learning.

Notice several differences from the risk-neutral case:

- The Q-values are **positive** (expectations of exponentials) rather than signed.
- The optimal policy is $\sigma(x) = \arg\min_a q(x,a)$ – we **minimize** rather than maximize, because ψ^{-1} is decreasing.
- The observed profit enters through $\exp(-\gamma R_{t+1})$ rather than additively.
- The continuation value enters as a **power** $(\min_{a'} q_t)^{\beta}$ rather than a scaled sum $\beta \cdot \max_{a'} q_t$.

As before, the agent needs only to observe x, a, R_{t+1}, and X_{t+1} – no model knowledge is required.

59.5.4 Implementation plan

Our implementation follows the same structure as the risk-neutral Q-learning in *Inventory Management via Q-Learning*, with the modifications above:

1. **Initialize** the Q-table q optimistically (see below) and visit counts n to zeros.
2. **At each step:**
   - Draw demand D_{t+1} and compute observed profit R_{t+1} and next state X_{t+1}.
   - Compute $\min_{a'} q_t(X_{t+1}, a')$ over feasible actions (this is a scalar for the update target, and the arg min action is used by the ε-greedy behavior policy).
   - Update $q_t(x,a)$ using the rule above, with learning rate $\alpha_t = 1/n_t(x,a)^{0.51}$.
   - Choose the next action via ε-greedy: with probability ε pick a random feasible action, otherwise pick the arg min action.
   - Decay ε.
3. **Extract the greedy policy** from the final Q-table via $\sigma(x) = \arg\min_{a \in \Gamma(x)} q(x,a)$.
4. **Compare** the learned policy against the VFI solution.

59.5.5 Optimistic initialization

As in *Inventory Management via Q-Learning*, we use optimistic initialization to accelerate learning.

The logic is the same – initialize the Q-table so that every untried action looks attractive, driving the agent to explore broadly – but the direction is reversed.

Since the optimal policy *minimizes* q, "optimistic" means initializing the Q-table *below* the true values. When the agent tries an action, the update pushes q upward toward reality, making that entry look worse and prompting the agent to try other actions that still appear optimistically good.

The true Q-values are on the order of $\exp(-\gamma\, v^*) \approx 10^{-8}$ to 10^{-6}. We initialize the Q-table at 10^{-9}, modestly below this range.

59.5.6 Implementation

We first define a helper to extract the greedy policy from the Q-table.

Since the optimal policy minimizes q, we use arg min rather than arg max.

```
@numba.jit(nopython=True)
def greedy_policy_from_q_rs(q, K):
    """Extract greedy policy from risk-sensitive Q-table (argmin)."""
    σ = np.empty(K + 1, dtype=np.int32)
    for x in range(K + 1):
        best_val = np.inf
        best_a = 0
        for a in range(K - x + 1):
            if q[x, a] < best_val:
                best_val = q[x, a]
                best_a = a
        σ[x] = best_a
    return σ
```

The Q-learning loop mirrors the risk-neutral version, with the key changes: the update target uses $\exp(-\gamma R_{t+1}) \cdot (\min_{a'} q_t)^\beta$, and the behavior policy follows the arg min.

```
@numba.jit(nopython=True)
def q_learning_rs_kernel(K, p, c, κ, β, γ, n_steps, X_init,
                         ε_init, ε_min, ε_decay, q_init, snapshot_steps, seed):
    np.random.seed(seed)
    q = np.full((K + 1, K + 1), q_init)  # optimistic initialization
    n = np.zeros((K + 1, K + 1))         # visit counts for learning rate
    ε = ε_init

    n_snaps = len(snapshot_steps)
    snapshots = np.zeros((n_snaps, K + 1), dtype=np.int32)
    snap_idx = 0

    # Initialize state and action
    x = X_init
    a = np.random.randint(0, K - x + 1)

    for t in range(n_steps):
        # Record policy snapshot if needed
        if snap_idx < n_snaps and t == snapshot_steps[snap_idx]:
            snapshots[snap_idx] = greedy_policy_from_q_rs(q, K)
            snap_idx += 1

        # === Draw D_{t+1} and observe outcome ===
        d = np.random.geometric(p) - 1
        reward = min(x, d) - c * a - κ * (a > 0)
        x_next = max(x - d, 0) + a

        # === Min over next state (scalar value for update target) ===
        # Also record the argmin action for use by the behavior policy.
        best_next = np.inf
        a_next = 0
```

(continues on next page)

(continued from previous page)

```
        for aa in range(K - x_next + 1):
            if q[x_next, aa] < best_next:
                best_next = q[x_next, aa]
                a_next = aa

        # === Risk-sensitive Q-learning update ===
        target = np.exp(-γ * reward) * best_next ** β
        n[x, a] += 1
        α = 1.0 / n[x, a] ** 0.51
        q[x, a] = (1 - α) * q[x, a] + α * target

        # === Behavior policy: ε-greedy (uses a_next, the argmin action) ===
        x = x_next
        if np.random.random() < ε:
            a = np.random.randint(0, K - x + 1)
        else:
            a = a_next
        ε = max(ε_min, ε * ε_decay)

    return q, snapshots
```

The wrapper function unpacks the model and provides default hyperparameters.

```
def q_learning_rs(model, n_steps=20_000_000, X_init=0,
                  ε_init=1.0, ε_min=0.01, ε_decay=0.999999,
                  q_init=1e-9, snapshot_steps=None, seed=1234):
    x_values, d_values, ϕ_values, p, c, κ, β, γ = model
    K = len(x_values) - 1
    if snapshot_steps is None:
        snapshot_steps = np.array([], dtype=np.int64)
    return q_learning_rs_kernel(K, p, c, κ, β, γ, n_steps, X_init,
                                ε_init, ε_min, ε_decay, q_init, snapshot_steps, seed)
```

59.5.7 Running Q-learning

We run n = 5 million steps and take policy snapshots at steps 10,000, 1,000,000, and n.

```
n = 5_000_000
snap_steps = np.array([10_000, 1_000_000, n], dtype=np.int64)
q_table, snapshots = q_learning_rs(model, n_steps=n+1, snapshot_steps=snap_steps)
```

59.5.8 Comparing with the exact solution

We extract the value function and policy from the final Q-table.

Since Q-values represent $\mathbb{E}[\exp(-\gamma(\cdots))]$, we recover the value function via $v_Q(x) = -\frac{1}{\gamma}\ln(\min_a q(x,a))$ and the policy via $\sigma_Q(x) = \arg\min_a q(x,a)$.

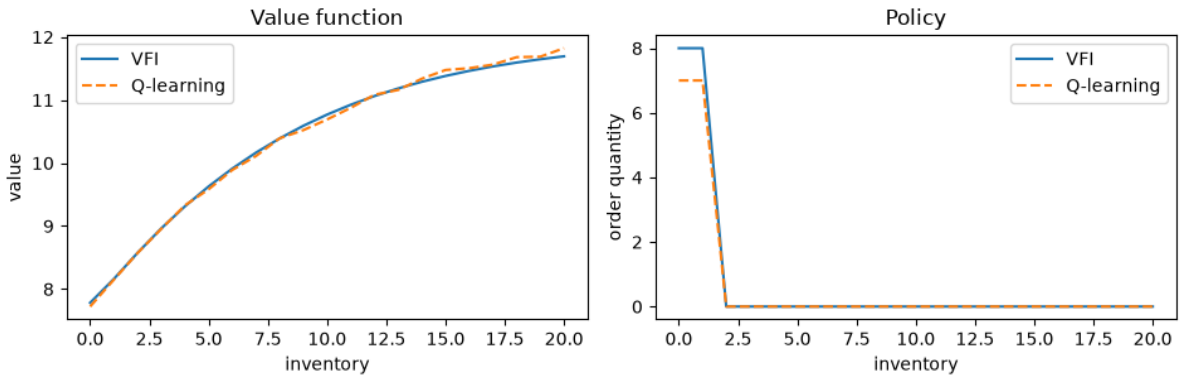
```
K = len(x_values) - 1
γ_base = model.γ
# restrict to feasible actions a ∈ {0, ..., K-x}
v_q = np.array([-(1/γ_base) * np.log(np.min(q_table[x, :K - x + 1]))
                for x in range(K + 1)])
```

(continues on next page)

(continued from previous page)

```
σ_q = np.array([np.argmin(q_table[x, :K - x + 1])
                for x in range(K + 1)])
```

```
fig, axes = plt.subplots(1, 2, figsize=(9.6, 3.2))

axes[0].plot(x_values, v_star, label="VFI")
axes[0].plot(x_values, v_q, '--', label="Q-learning")
axes[0].set_xlabel("inventory")
axes[0].set_ylabel("value")
axes[0].legend()
axes[0].set_title("Value function")

axes[1].plot(x_values, σ_star, label="VFI")
axes[1].plot(x_values, σ_q, '--', label="Q-learning")
axes[1].set_xlabel("inventory")
axes[1].set_ylabel("order quantity")
axes[1].legend()
axes[1].set_title("Policy")

plt.tight_layout()
plt.show()
```



59.5.9 Visualizing learning over time

The panels below show how the agent's policy evolves during training.

Each panel simulates an inventory path using the greedy policy extracted from the Q-table at a given training step, with the same demand sequence throughout.

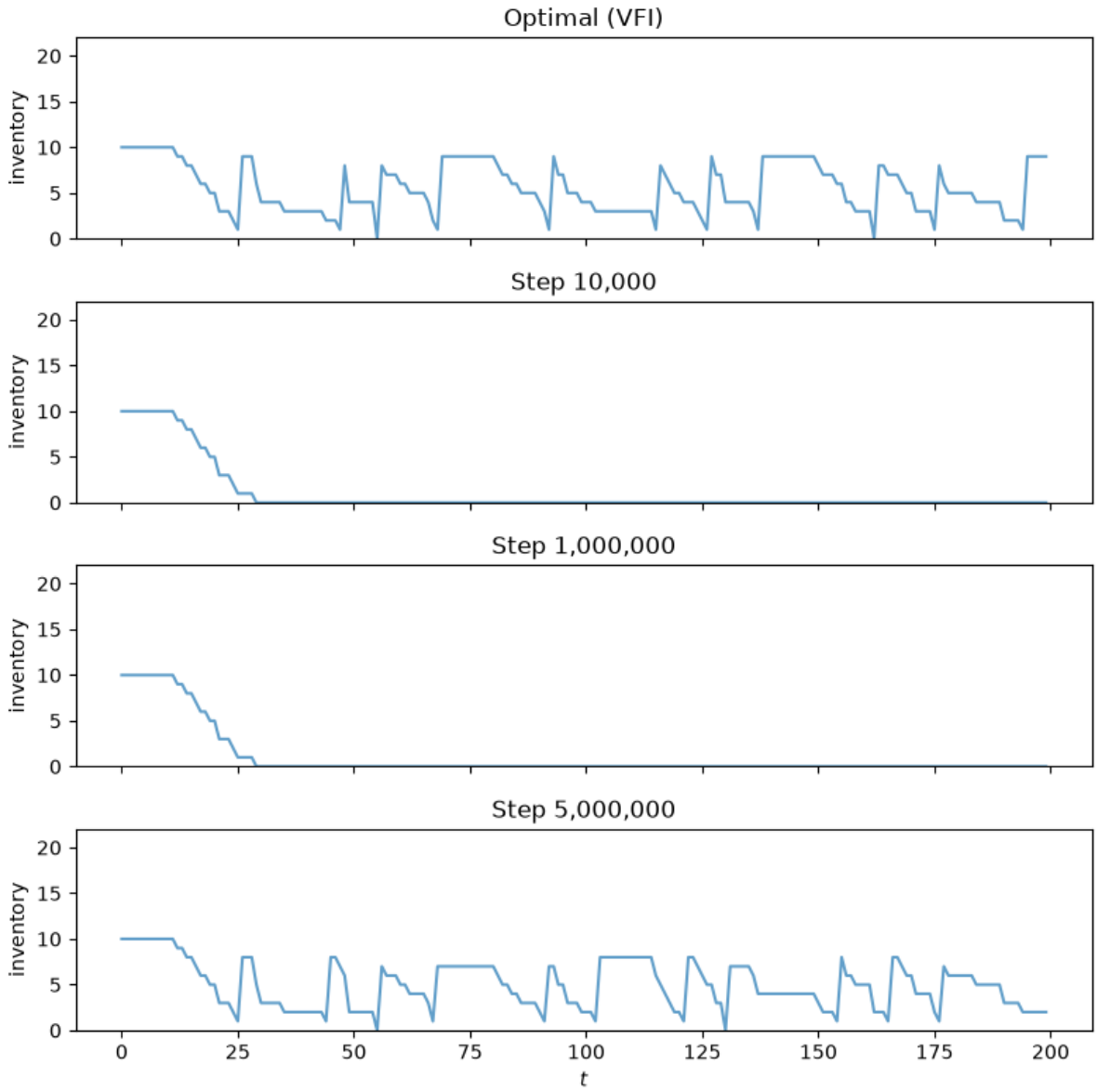
The top panel shows the optimal policy from VFI for reference.

```
ts_length = 200
n_snaps = len(snap_steps)
fig, axes = plt.subplots(n_snaps + 1, 1, figsize=(8, 2.0 * (n_snaps + 1)),
                         sharex=True)

X_init = K // 2
sim_seed = 5678

# Optimal policy
X_opt = sim_inventories(ts_length, σ_star, model.p, X_init, seed=sim_seed)
```

(continues on next page)

(continued from previous page)

```
axes[0].plot(X_opt, alpha=0.7)
axes[0].set_ylabel("inventory")
axes[0].set_title("Optimal (VFI)")
axes[0].set_ylim(0, K + 2)

# Q-learning snapshots
for i in range(n_snaps):
    σ_snap = snapshots[i]
    X = sim_inventories(ts_length, σ_snap, model.p, X_init, seed=sim_seed)
    axes[i + 1].plot(X, alpha=0.7)
    axes[i + 1].set_ylabel("inventory")
    axes[i + 1].set_title(f"Step {snap_steps[i]:,}")
    axes[i + 1].set_ylim(0, K + 2)

axes[-1].set_xlabel(r"$t$")
plt.tight_layout()
plt.show()
```



After 10,000 steps, the agent has barely explored and its policy is erratic.

By 1,000,000 steps the learned policy has improved but still differs noticeably from the optimum.

By step 5 million the inventory dynamics are nearly indistinguishable from the VFI solution.

Note that the converged policy maintains lower inventory levels than in the risk-neutral case (compare with *Inventory Management via Q-Learning*), consistent with the mechanism discussed above: a risk-sensitive agent caps its exposure to demand variance by holding less stock.

59.6 Conclusion

We extended the inventory management problem from *Inventory Management via Q-Learning* to incorporate risk sensitivity via the certainty equivalent operator $\psi^{-1}(\mathbb{E}[\psi(\cdot)])$ with $\psi(t) = \exp(-\gamma t)$.

Value function iteration confirms that risk-sensitive firms order less aggressively, preferring predictable profits over higher but more volatile returns.

We then showed that Q-learning can be adapted to the risk-sensitive setting by working with the transformed Q-factor $q(x, a) = \mathbb{E}[\exp(-\gamma(\pi + \beta v^*))]$.

The resulting update rule replaces addition with multiplication and max with min, but retains the key property of model-free learning: the agent needs only to observe states, actions, and profits.

CHAPTER

SIXTY

JOB SEARCH IX: SEARCH WITH Q-LEARNING

60.1 Overview

This lecture illustrates a powerful machine learning technique called Q-learning.

[Sutton and Barto, 2018] presents Q-learning and a variety of other statistical learning procedures.

The Q-learning algorithm combines ideas from

- dynamic programming
- a recursive version of least squares known as temporal difference learning.

This lecture applies a Q-learning algorithm to the situation faced by a McCall worker.

This lecture also considers the case where a McCall worker is given an option to quit the current job.

Relative to the dynamic programming formulation of the McCall worker model that we studied in *quantecon lecture*, a Q-learning algorithm gives the worker less knowledge about

- the random process that generates a sequence of wages
- the reward function that tells consequences of accepting or rejecting a job

The Q-learning algorithm invokes a statistical learning model to learn about these things.

Statistical learning often comes down to some version of least squares, and it will be here too.

Any time we say **statistical learning**, we have to say what object is being learned.

For Q-learning, the object that is learned is not the **value function** that is a focus of dynamic programming.

But it is something that is closely affiliated with it.

In the finite-action, finite state context studied in this lecture, the object to be learned statistically is a **Q-table**, an instance of a **Q-function** for finite sets.

Sometimes a Q-function or Q-table is called a quality-function or quality-table.

The rows and columns of a Q-table correspond to possible states that an agent might encounter, and possible actions that he can take in each state.

An equation that resembles a Bellman equation plays an important role in the algorithm.

It differs from the Bellman equation for the McCall model that we have seen in *this quantecon lecture*

In this lecture, we'll learn a little about

- the **Q-function** or **quality function** that is affiliated with any Markov decision problem whose optimal value function satisfies a Bellman equation
- **temporal difference learning**, a key component of a Q-learning algorithm

As usual, let's import some Python modules.

```
!pip install quantecon
```

```
import numpy as np

from numba import jit, float64, int64
from numba.experimental import jitclass
from quantecon.distributions import BetaBinomial

import matplotlib.pyplot as plt

np.random.seed(123)
```

60.2 Review of McCall Model

We begin by reviewing the McCall model described in *this quantecon lecture*.

We'll compute an optimal value function and a policy that attains it.

We'll eventually compare that optimal policy to what the Q-learning McCall worker learns.

The McCall model is characterized by parameters β, c and a known distribution of wage offers F.

A McCall worker wants to maximize an expected discounted sum of lifetime incomes

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t y_t$$

The worker's income y_t equals his wage w if he is employed, and unemployment compensation c if he is unemployed.

An optimal value $V(w)$ for a McCall worker who has just received a wage offer w and is deciding whether to accept or reject it satisfies the Bellman equation

$$V(w) = \max_{\text{accept, reject}} \left\{ \frac{w}{1-\beta}, c + \beta \int V(w') \, dF(w') \right\} \tag{60.1}$$

To form a benchmark to compare with results from Q-learning, we first approximate the optimal value function.

With possible states residing in a finite discrete state space indexed by $\{1, 2, ..., n\}$, we make an initial guess for the value function of $v \in \mathbb{R}^n$ and then iterate on the Bellman equation:

$$v'(i) = \max \left\{ \frac{w(i)}{1-\beta}, c + \beta \sum_{1 \leq j \leq n} v(j) q(j) \right\} \quad \text{for } i = 1, \ldots, n$$

Let's use Python code from *this quantecon lecture*.

We use a Python method called `VFI` to compute the optimal value function using value function iterations.

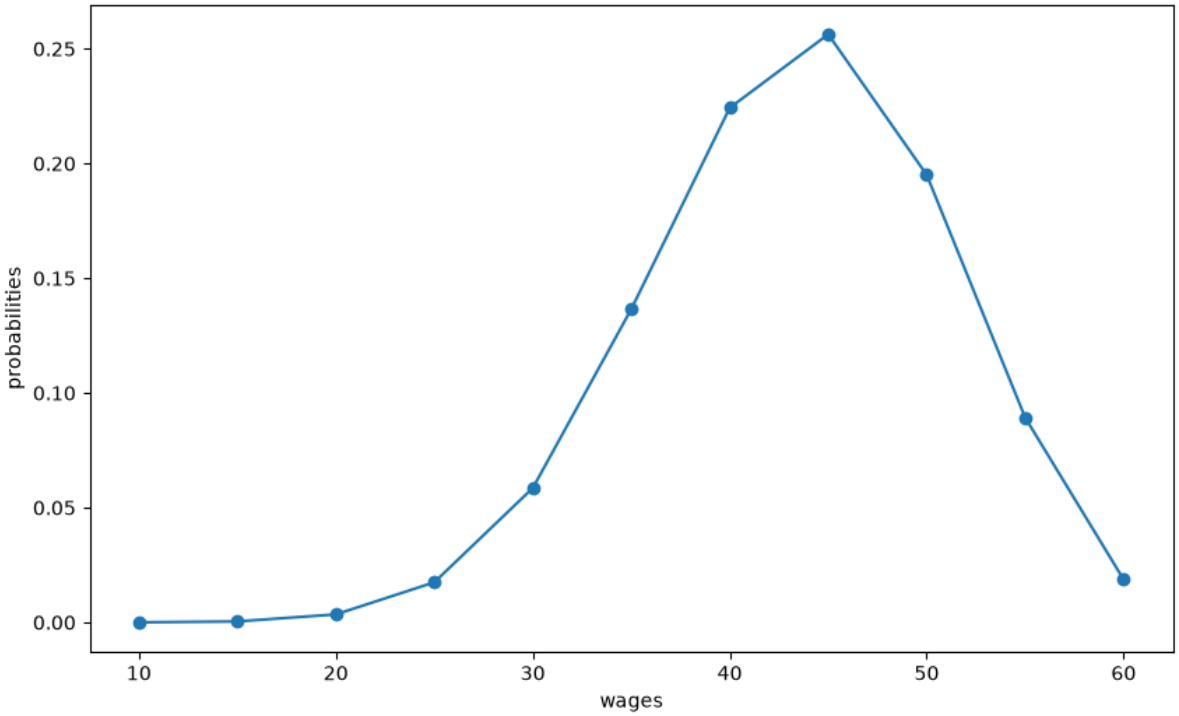
We construct an assumed distribution of wages and plot it with the following Python code

```
n, a, b = 10, 200, 100                       # default parameters
q_default = BetaBinomial(n, a, b).pdf()      # default choice of q

w_min, w_max = 10, 60
w_default = np.linspace(w_min, w_max, n+1)
```

(continues on next page)

(continued from previous page)

```
# plot distribution of wage offer
fig, ax = plt.subplots(figsize=(10,6))
ax.plot(w_default, q_default, '-o', label='$q(w(i))$')
ax.set_xlabel('wages')
ax.set_ylabel('probabilities')

plt.show()
```



Next we'll compute the worker's optimal value function by iterating to convergence on the Bellman equation.

Then we'll plot various iterates on the Bellman operator.

```
mccall_data = [
    ('c', float64),      # unemployment compensation
    ('β', float64),      # discount factor
    ('w', float64[::1]),  # array of wage values, w[i] = wage at state i
    ('q', float64[::1])   # array of probabilities
]


@jitclass(mccall_data)
class McCallModel:

    def __init__(self, c=25, β=0.99, w=w_default, q=q_default):

        self.c, self.β = c, β
        self.w, self.q = w, q

    def state_action_values(self, i, v):
        """
        The values of state-action pairs.
```

(continues on next page)

(continued from previous page)

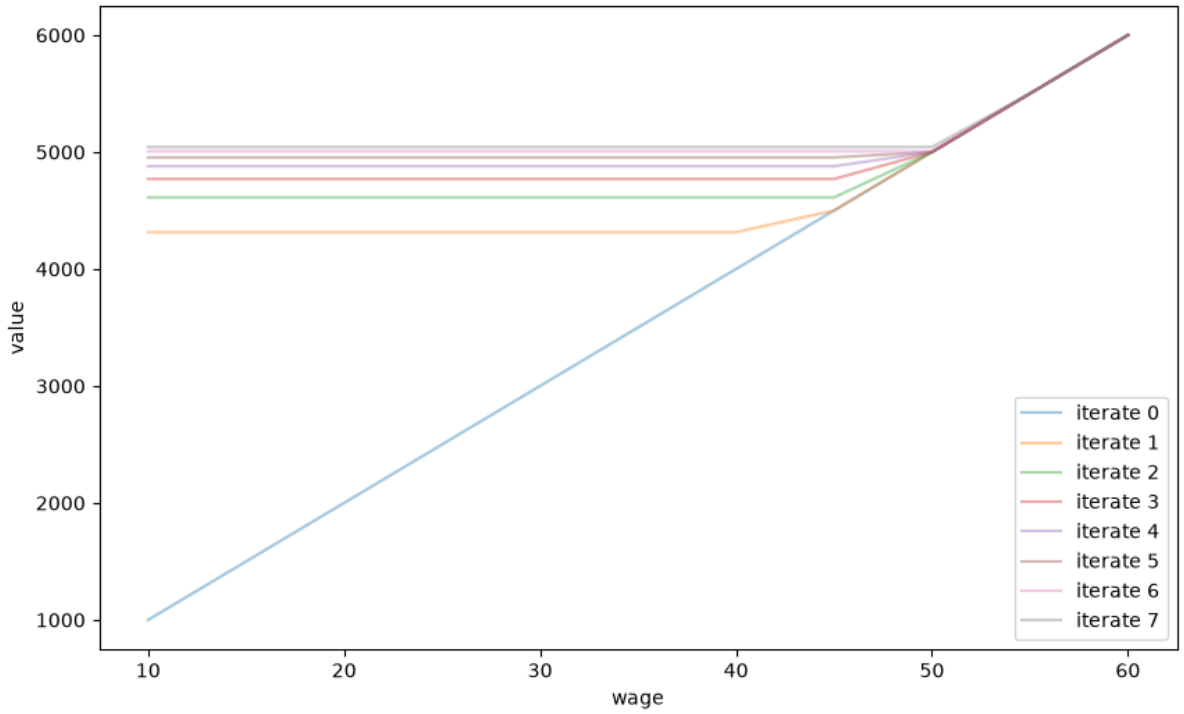
```
        """
        # Simplify names
        c, β, w, q = self.c, self.β, self.w, self.q
        # Evaluate value for each state-action pair
        # Consider action = accept or reject the current offer
        accept = w[i] / (1 - β)
        reject = c + β * (v @ q)

        return np.array([accept, reject])

    def VFI(self, eps=1e-5, max_iter=500):
        """
        Find the optimal value function.
        """

        n = len(self.w)
        v = self.w / (1 - self.β)
        v_next = np.empty_like(v)
        flag=0

        for i in range(max_iter):
            for j in range(n):
                v_next[j] = np.max(self.state_action_values(j, v))

            if np.max(np.abs(v_next - v))<=eps:
                flag=1
                break
            v[:] = v_next

        return v, flag

def plot_value_function_seq(mcm, ax, num_plots=8):
    """
    Plot a sequence of value functions.

        * mcm is an instance of McCallModel
        * ax is an axes object that implements a plot method.

    """

    n = len(mcm.w)
    v = mcm.w / (1 - mcm.β)
    v_next = np.empty_like(v)
    for i in range(num_plots):
        ax.plot(mcm.w, v, '-', alpha=0.4, label=f"iterate {i}")
        # Update guess
        for i in range(n):
            v_next[i] = np.max(mcm.state_action_values(i, v))
        v[:] = v_next  # copy contents into v

    ax.legend(loc='lower right')
```

```
mcm = McCallModel()
valfunc_VFI, flag = mcm.VFI()

fig, ax = plt.subplots(figsize=(10,6))
```

(continues on next page)

(continued from previous page)

```
ax.set_xlabel('wage')
ax.set_ylabel('value')
plot_value_function_seq(mcm, ax)
plt.show()
```



Next we'll print out the limit of the sequence of iterates.

This is the approximation to the McCall worker's value function that is produced by value function iteration.

We'll use this value function as a benchmark later after we have done some Q-learning.

```
print(valfunc_VFI)
```

```
[5322.27935875 5322.27935875 5322.27935875 5322.27935875 5322.27935875
 5322.27935875 5322.27935875 5322.27935875 5500.
 6000.        ]
```

60.3 Implied Quality Function Q

A **quality function** Q map state-action pairs into optimal values.

They are tightly linked to optimal value functions.

But value functions are functions just of states, and not actions.

For each given state, the quality function gives a list of optimal values that can be attained starting from that state, with each component of the list indicating one of the possible actions that is taken.

For our McCall worker with a finite set of possible wages

- the state space $\mathcal{W} = \{w_1, w_2, ..., w_n\}$ is indexed by integers $1, 2, ..., n$

- the action space is $\mathcal{A} = \{\text{accept}, \text{reject}\}$

Let $a \in \mathcal{A}$ be one of the two possible actions, i.e., accept or reject.

For our McCall worker, an optimal Q-function $Q(w, a)$ equals the maximum value of that a previously unemployed worker who has offer w in hand can attain if he takes action a.

This definition of $Q(w, a)$ presumes that in subsequent periods the worker takes optimal actions.

An optimal Q-function for our McCall worker satisfies

$$\begin{aligned} Q\left(w, \text{accept}\right) &= \frac{w}{1-\beta} \\ Q\left(w, \text{reject}\right) &= c + \beta \int \max_{\text{accept, reject}} \left\{ \frac{w'}{1-\beta}, Q\left(w', \text{reject}\right) \right\} dF\left(w'\right) \end{aligned} \tag{60.2}$$

Note that the first equation of system (60.2) presumes that after the agent has accepted an offer, he will not have the objection to reject that same offer in the future.

These equations are aligned with the Bellman equation for the worker's optimal value function that we studied in *this quantecon lecture*.

Evidently, the optimal value function $V(w)$ described in that lecture is related to our Q-function by

$$V(w) = \max_{\text{accept,reject}} \left\{ Q(w, \text{accept}), Q\left(w, \text{reject}\right) \right\}$$

If we stare at the second equation of system (60.2), we notice that since the wage process is identically and independently distributed over time, $Q\left(w, \text{reject}\right)$, the right side of the equation is independent of the current state w.

So we can denote it as a scalar

$$Q_r := Q\left(w, \text{reject}\right) \quad \forall\, w \in \mathcal{W}.$$

This fact provides us with an an alternative, and as it turns out in this case, a faster way to compute an optimal value function and associated optimal policy for the McCall worker.

Instead of using the value function iterations that we deployed above, we can instead iterate to convergence on a version of the second equation in system (60.2) that maps an estimate of Q_r into an improved estimate Q_r':

$$Q_r' = c + \beta \int \max \left\{ \frac{w'}{1-\beta}, Q_r \right\} dF\left(w'\right)$$

After a Q_r sequence has converged, we can recover the optimal value function $V(w)$ for the McCall worker from

$$V\left(w\right) = \max \left\{ \frac{w}{1-\beta}, Q_r \right\}$$

60.4 From Probabilities to Samples

We noted above that the optimal Q function for our McCall worker satisfies the Bellman equations

$$\begin{aligned} w + \beta \max_{\text{accept, reject}} \left\{ Q(w, \text{accept}), Q(w, \text{reject}) \right\} - Q(w, \text{accept}) &= 0 \\ c + \beta \int \max_{\text{accept, reject}} \left\{ Q(w', \text{accept}), Q\left(w', \text{reject}\right) \right\} dF\left(w'\right) - Q\left(w, \text{reject}\right) &= 0 \end{aligned} \tag{60.3}$$

Notice the integral over $F(w')$ on the second line.

Erasing the integral sign sets the stage for an illegitmate argument that can get us started thinking about Q-learning.

Thus, construct a difference equation system that keeps the first equation of (60.3) but replaces the second by removing integration over $F(w')$:

$$\begin{aligned} w + \beta \max_{\text{accept, reject}} \{Q(w, \text{accept}), Q(w, \text{reject})\} - Q(w, \text{accept}) = 0 \\ c + \beta \max_{\text{accept, reject}} \{Q(w', \text{accept}), Q\left(w', \text{reject}\right)\} - Q\left(w, \text{reject}\right) \approx 0 \end{aligned} \tag{60.4}$$

The second equation can't hold for all w, w' pairs in the appropriate Cartesian product of our state space.

But maybe an appeal to a Law of Large numbers could let us hope that it would hold **on average** for a long time series sequence of draws of w_t, w_{t+1} pairs, where we are thinking of w_t as w and w_{t+1} as w'.

The basic idea of Q-learning is to draw a long sample of wage offers from F (we know F though we assume that the worker doesn't) and iterate on a recursion that maps an estimate $\hat{Q}_t$ of a Q-function at date t into an improved estimate $\hat{Q}_{t+1}$ at date $t+1$.

To set up such an algorithm, we first define some errors or "differences"

$$\begin{aligned} w + \beta \max_{\text{accept, reject}} \left\{\hat{Q}_t(w_t, \text{accept}), \hat{Q}_t(w_t, \text{reject})\right\} - \hat{Q}_t(w_t, \text{accept}) = \text{diff}_{\text{accept},t} \\ c + \beta \max_{\text{accept, reject}} \left\{\hat{Q}_t(w_{t+1}, \text{accept}), \hat{Q}_t\left(w_{t+1}, \text{reject}\right)\right\} - \hat{Q}_t\left(w_t, \text{reject}\right) = \text{diff}_{\text{reject},t} \end{aligned} \tag{60.5}$$

The adaptive learning scheme would then be some version of

$$\hat{Q}_{t+1} = \hat{Q}_t + \alpha \, \text{diff}_t \tag{60.6}$$

where $\alpha \in (0, 1)$ is a small **gain** parameter that governs the rate of learning and $\hat{Q}_t$ and diff_t are 2×1 vectors corresponding to objects in equation system (60.5).

This informal argument takes us to the threshold of Q-learning.

60.5 Q-Learning

Let's first describe a Q-learning algorithm precisely.

Then we'll implement it.

The algorithm works by using a Monte Carlo method to update estimates of a Q-function.

We begin with an initial guess for a Q-function.

In the example studied in this lecture, we have a finite action space and also a finite state space.

That means that we can represent a Q-function as a matrix or Q-table, $\widetilde{Q}(w, a)$.

Q-learning proceeds by updating the Q-function as the decision maker acquires experience along a path of wage draws generated by simulation.

During the learning process, our McCall worker takes actions and experiences rewards that are consequences of those actions.

He learns simultaneously about the environment, in this case the distribution of wages, and the reward function, in this case the unemployment compensation c and the present value of wages.

The updating algorithm is based on a slight modification (to be described soon) of a recursion like

$$\widetilde{Q}^{new}\left(w, a\right) = \widetilde{Q}^{old}\left(w, a\right) + \alpha \widetilde{TD}\left(w, a\right) \tag{60.7}$$

where

$$\begin{aligned}\widetilde{TD}\left(w,\text{accept}\right) &= \left[w+\beta\max_{a'\in\mathcal{A}}\widetilde{Q}^{old}\left(w,a'\right)\right]-\widetilde{Q}^{old}\left(w,\text{accept}\right)\\ \widetilde{TD}\left(w,\text{reject}\right) &= \left[c+\beta\max_{a'\in\mathcal{A}}\widetilde{Q}^{old}\left(w',a'\right)\right]-\widetilde{Q}^{old}\left(w,\text{reject}\right),\ w'\sim F\end{aligned} \tag{60.8}$$

The terms $\widetilde{TD}(w,a)$ for $a=\{\text{accept,reject}\}$ are the **temporal difference errors** that drive the updates.

This system is thus a version of the adaptive system that we sketched informally in equation (60.6).

An aspect of the algorithm not yet captured by equation system (60.8) is random **experimentation** that we add by occasionally randomly replacing

$$\text{argmax}_{a'\in\mathcal{A}}\widetilde{Q}^{old}\left(w,a'\right)$$

with

$$\text{argmin}_{a'\in\mathcal{A}}\widetilde{Q}^{old}\left(w,a'\right)$$

and occasionally replacing

$$\text{argmax}_{a'\in\mathcal{A}}\widetilde{Q}^{old}\left(w',a'\right)$$

with

$$\text{argmin}_{a'\in\mathcal{A}}\widetilde{Q}^{old}\left(w',a'\right)$$

We activate such experimentation with probability ϵ in step 3 of the following pseudo-code for our McCall worker to do Q-learning:

1. Set an arbitrary initial Q-table.
2. Draw an initial wage offer w from F.
3. From the appropriate row in the Q-table, choose an action using the following ϵ-greedy algorithm:
    - with probability $1-\epsilon$, choose the action that maximizes the value, and
    - with probability ϵ, choose the alternative action.
4. Update the state associated with the chosen action and compute $\widetilde{TD}$ according to (60.8) and update $\widetilde{Q}$ according to (60.7).
5. Either draw a new state w' if required or else take existing wage if and update the Q-table again according to (60.7).
6. Stop when the old and new Q-tables are close enough, i.e., $\|\tilde{Q}^{new}-\tilde{Q}^{old}\|_\infty\leq\delta$ for given δ or if the worker keeps accepting for T periods for a prescribed T.
7. Return to step 2 with the updated Q-table.

Repeat this procedure for N episodes or until the updated Q-table has converged.

We call one pass through steps 2 to 7 an "episode" or "epoch" of temporal difference learning.

In our context, each episode starts with an agent drawing an initial wage offer, i.e., a new state.

The agent then takes actions based on the preset Q-table, receives rewards, and then enters a new state implied by this period's actions.

The Q-table is updated via temporal difference learning.

We iterate this until convergence of the Q-table or the maximum length of an episode is reached.

Multiple episodes allow the agent to start afresh and visit states that she was less likely to visit from the terminal state of a previos episode.

For example, an agent who has accepted a wage offer based on her Q-table will be less likely to draw a new offer from other parts of the wage distribution.

By using the ϵ-greedy method and also by increasing the number of episodes, the Q-learning algorithm balances gains from exploration and from exploitation.

Remark: Notice that $\widetilde{TD}$ associated with an optimal Q-table defined in (60.7) automatically above satisfies $\widetilde{TD} = 0$ for all state action pairs. Whether a limit of our Q-learning algorithm converges to an optimal Q-table depends on whether the algorithm visits all state-action pairs often enough.

We implement this pseudo code in a Python class.

For simplicity and convenience, we let `s` represent the state index between 0 and $n = 50$ and $w_s = w[s]$.

The first column of the Q-table represents the value associated with rejecting the wage and the second represents accepting the wage.

We use `numba` compilation to accelerate computations.

```
params=[
    ('c', float64),            # unemployment compensation
    ('β', float64),            # discount factor
    ('w', float64[:]),         # array of wage values, w[i] = wage at state i
    ('q', float64[:]),         # array of probabilities
    ('eps', float64),          # for epsilon greedy algorithm
    ('δ', float64),            # Q-table threshold
    ('lr', float64),           # the learning rate α
    ('T', int64),              # maximum periods of accepting
    ('quit_allowed', int64)    # whether quit is allowed after accepting the wage␣
 ↪offer
]

@jitclass(params)
class Qlearning_McCall:
    def __init__(self, c=25, β=0.99, w=w_default, q=q_default, eps=0.1,
                 δ=1e-5, lr=0.5, T=10000, quit_allowed=0):

        self.c, self.β = c, β
        self.w, self.q = w, q
        self.eps, self.δ, self.lr, self.T = eps, δ, lr, T
        self.quit_allowed = quit_allowed


    def draw_offer_index(self):
        """
        Draw a state index from the wage distribution.
        """

        q = self.q
        return np.searchsorted(np.cumsum(q), np.random.random(), side="right")

    def temp_diff(self, qtable, state, accept):
        """
        Compute the TD associated with state and action.
        """

        c, β, w = self.c, self.β, self.w
```

(continues on next page)

(continued from previous page)

```
        if accept==0:
            state_next = self.draw_offer_index()
            TD = c + β*np.max(qtable[state_next, :]) - qtable[state, accept]
        else:
            state_next = state
            if self.quit_allowed == 0:
                TD = w[state_next] + β*np.max(qtable[state_next, :]) - qtable[state,␣
↪accept]
            else:
                TD = w[state_next] + β*qtable[state_next, 1] - qtable[state, accept]

        return TD, state_next

    def run_one_epoch(self, qtable, max_times=20000):
        """
        Run an "epoch".
        """

        c, β, w = self.c, self.β, self.w
        eps, δ, lr, T = self.eps, self.δ, self.lr, self.T

        s0 = self.draw_offer_index()
        s = s0
        accept_count = 0

        for t in range(max_times):

            # choose action
            accept = np.argmax(qtable[s, :])
            if np.random.random()<=eps:
                accept = 1 - accept

            if accept == 1:
                accept_count += 1
            else:
                accept_count = 0

            TD, s_next = self.temp_diff(qtable, s, accept)

            # update qtable
            qtable_new = qtable.copy()
            qtable_new[s, accept] = qtable[s, accept] + lr*TD

            if np.max(np.abs(qtable_new-qtable))<=δ:
                break

            if accept_count == T:
                break

            s, qtable = s_next, qtable_new

        return qtable_new

@jit
def run_epochs(N, qlmc, qtable):
    """
```

(continues on next page)

(continued from previous page)

```
    Run epochs N times with qtable from the last iteration each time.
    """

    for n in range(N):
        if n%(N/10)==0:
            print(f"Progress: EPOCHs = {n}")
        new_qtable = qlmc.run_one_epoch(qtable)
        qtable = new_qtable

    return qtable

def valfunc_from_qtable(qtable):
    return np.max(qtable, axis=1)

def compute_error(valfunc, valfunc_VFI):
    return np.mean(np.abs(valfunc-valfunc_VFI))
```

```
# create an instance of Qlearning_McCall
qlmc = Qlearning_McCall()

# run
qtable0 = np.zeros((len(w_default), 2))
qtable = run_epochs(20000, qlmc, qtable0)
```

```
Progress: EPOCHs = 0
Progress: EPOCHs = 2000
Progress: EPOCHs = 4000
Progress: EPOCHs = 6000
Progress: EPOCHs = 8000
Progress: EPOCHs = 10000
Progress: EPOCHs = 12000
Progress: EPOCHs = 14000
Progress: EPOCHs = 16000
Progress: EPOCHs = 18000
```

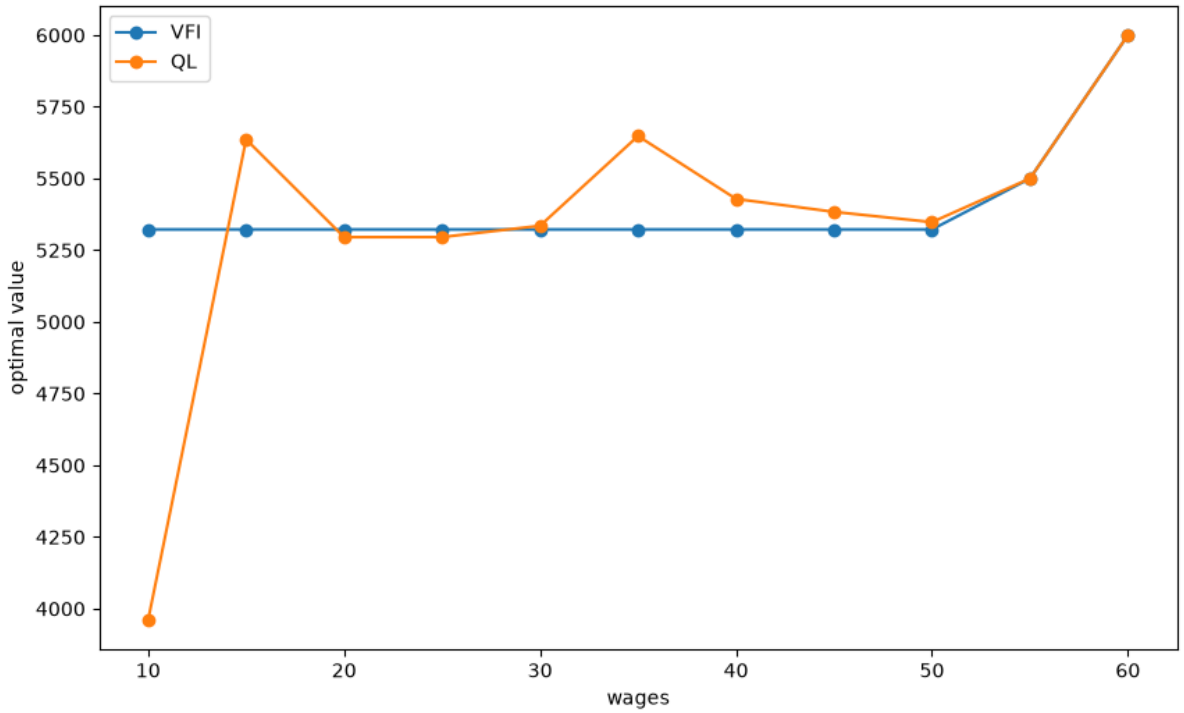
```
print(qtable)
```

```
[[3960.93847178 1299.70967816]
 [5636.36817271 5245.46113065]
 [5295.41787222 5277.16335932]
 [5295.72547219 5228.10269226]
 [5335.13133616 5268.11407615]
 [5648.02987308 5268.19487217]
 [5428.77024135 5269.28150808]
 [5374.51402128 5383.61501851]
 [5347.70011253 5347.82100605]
 [5371.10557404 5500.        ]
 [5469.69830287 6000.        ]]
```

```
# inspect value function
valfunc_qlr = valfunc_from_qtable(qtable)

print(valfunc_qlr)
```

```
[3960.93847178 5636.36817271 5295.41787222 5295.72547219 5335.13133616
 5648.02987308 5428.77024135 5383.61501851 5347.82100605 5500.
 6000.        ]
```

```
# plot
fig, ax = plt.subplots(figsize=(10,6))
ax.plot(w_default, valfunc_VFI, '-o', label='VFI')
ax.plot(w_default, valfunc_qlr, '-o', label='QL')
ax.set_xlabel('wages')
ax.set_ylabel('optimal value')
ax.legend()

plt.show()
```



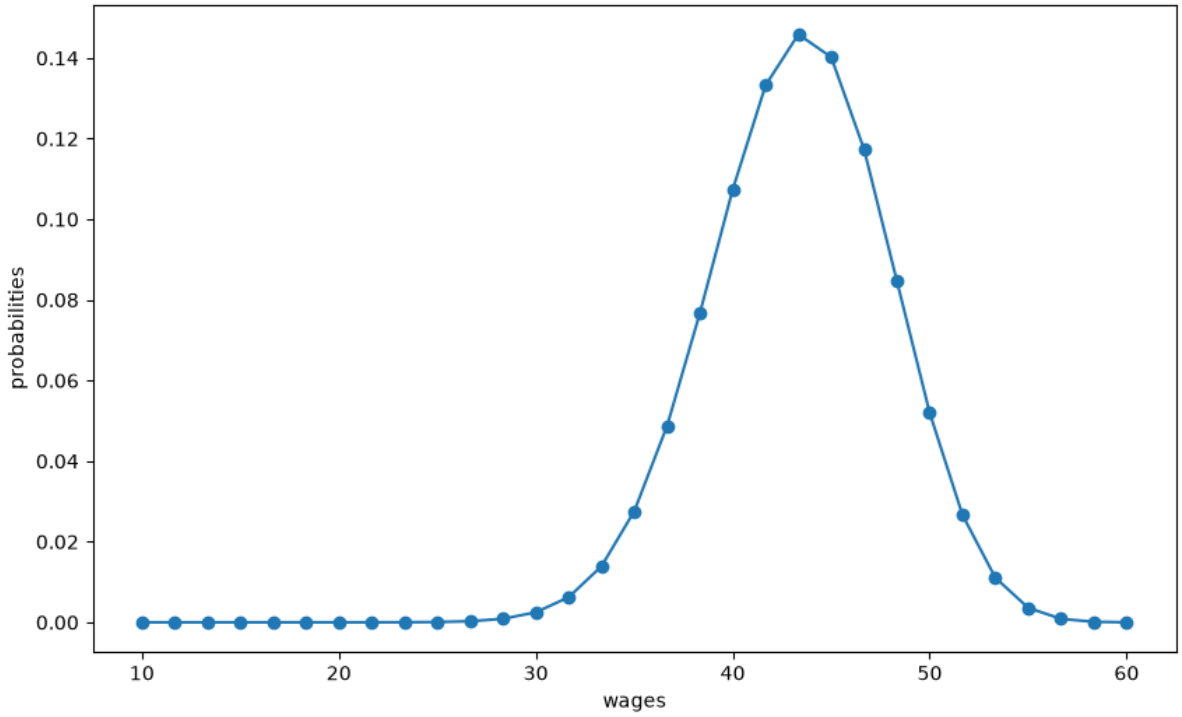
Now, let us compute the case with a larger state space: $n = 30$ instead of $n = 10$.

```
n, a, b = 30, 200, 100                    # default parameters
q_new = BetaBinomial(n, a, b).pdf()       # default choice of q

w_min, w_max = 10, 60
w_new = np.linspace(w_min, w_max, n+1)


# plot distribution of wage offer
fig, ax = plt.subplots(figsize=(10,6))
ax.plot(w_new, q_new, '-o', label='$q(w(i))$')
ax.set_xlabel('wages')
ax.set_ylabel('probabilities')

plt.show()
```

(continues on next page)

(continued from previous page)

```
# VFI
mcm = McCallModel(w=w_new, q=q_new)
valfunc_VFI, flag = mcm.VFI()
```




```
mcm = McCallModel(w=w_new, q=q_new)
valfunc_VFI, flag = mcm.VFI()
valfunc_VFI
```

```
array([4859.77015703, 4859.77015703, 4859.77015703, 4859.77015703,
       4859.77015703, 4859.77015703, 4859.77015703, 4859.77015703,
       4859.77015703, 4859.77015703, 4859.77015703, 4859.77015703,
       4859.77015703, 4859.77015703, 4859.77015703, 4859.77015703,
       4859.77015703, 4859.77015703, 4859.77015703, 4859.77015703,
       5000.        , 5166.66666667, 5333.33333333, 5500.        ,
       5666.66666667, 5833.33333333, 6000.        ])
```

```
def plot_epochs(epochs_to_plot, quit_allowed=1):
    "Plot value function implied by outcomes of an increasing number of epochs."
    qlmc_new = Qlearning_McCall(w=w_new, q=q_new, quit_allowed=quit_allowed)
    qtable = np.zeros((len(w_new),2))
    epochs_to_plot = np.asarray(epochs_to_plot)
    # plot
    fig, ax = plt.subplots(figsize=(10,6))
    ax.plot(w_new, valfunc_VFI, '-o', label='VFI')

    max_epochs = np.max(epochs_to_plot)
    # iterate on epoch numbers
    for n in range(max_epochs + 1):
        if n%(max_epochs/10)==0:
```

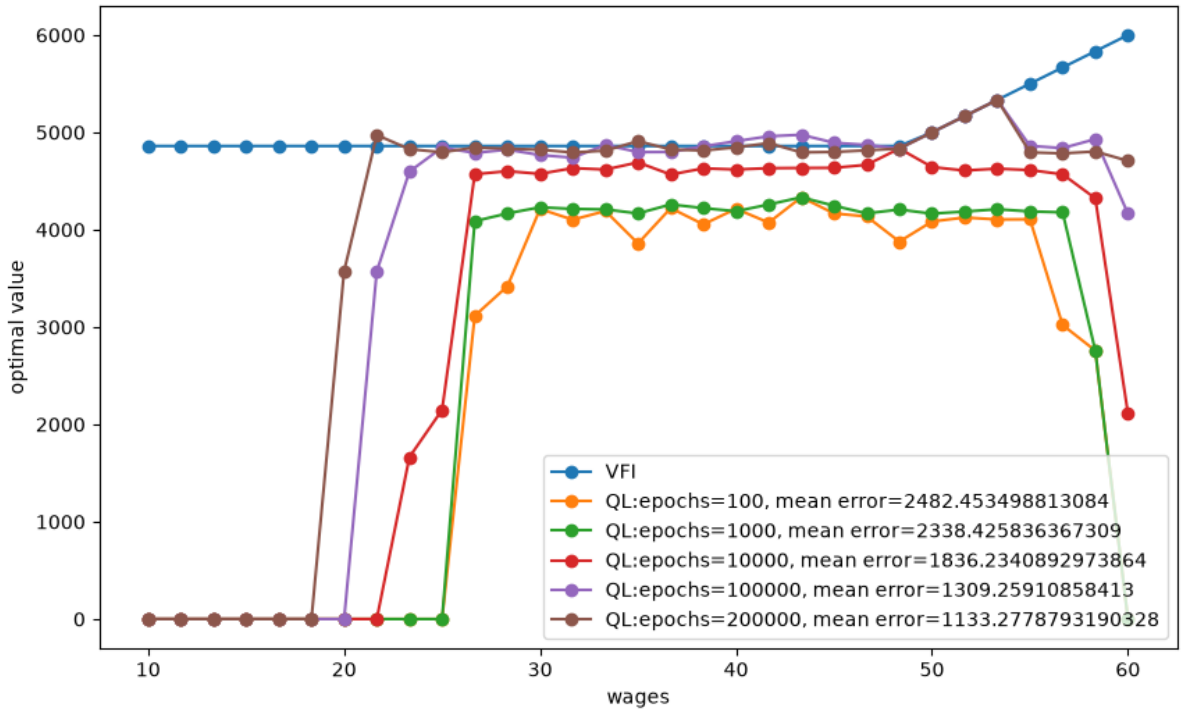
(continues on next page)

(continued from previous page)

```
            print(f"Progress: EPOCHs = {n}")
        if n in epochs_to_plot:
            valfunc_qlr = valfunc_from_qtable(qtable)
            error = compute_error(valfunc_qlr, valfunc_VFI)

            ax.plot(w_new, valfunc_qlr, '-o', label=f'QL:epochs={n}, mean error=
↪{error}')


        new_qtable = qlmc_new.run_one_epoch(qtable)
        qtable = new_qtable

    ax.set_xlabel('wages')
    ax.set_ylabel('optimal value')
    ax.legend(loc='lower right')
    plt.show()
```

```
plot_epochs(epochs_to_plot=[100, 1000, 10000, 100000, 200000])
```

```
Progress: EPOCHs = 0
Progress: EPOCHs = 20000
Progress: EPOCHs = 40000
Progress: EPOCHs = 60000
Progress: EPOCHs = 80000
Progress: EPOCHs = 100000
Progress: EPOCHs = 120000
Progress: EPOCHs = 140000
Progress: EPOCHs = 160000
Progress: EPOCHs = 180000
Progress: EPOCHs = 200000
```



The above graphs indicates that

- the Q-learning algorithm has trouble learning the Q-table well for wages that are rarely drawn
- the quality of approximation to the "true" value function computed by value function iteration improves for longer epochs

60.6 Employed Worker Can't Quit

The preceding version of temporal difference Q-learning described in equation system (60.8) lets an employed worker quit, i.e., reject her wage as an incumbent and instead receive unemployment compensation this period and draw a new offer next period.

This is an option that the McCall worker described in *this quantecon lecture* would not take.

See [Ljungqvist and Sargent, 2018], chapter 6 on search, for a proof.

But in the context of Q-learning, giving the worker the option to quit and get unemployment compensation while unemployed turns out to accelerate the learning process by promoting experimentation vis a vis premature exploitation only.

To illustrate this, we'll amend our formulas for temporal differences to forbid an employed worker from quitting a job she had accepted earlier.

With this understanding about available choices, we obtain the following temporal difference values:

$$\begin{aligned} \widetilde{TD}\left(w, \text{accept}\right) &= \left[w + \beta \widetilde{Q}^{old}\left(w, \text{accept}\right)\right] - \widetilde{Q}^{old}\left(w, \text{accept}\right) \\ \widetilde{TD}\left(w, \text{reject}\right) &= \left[c + \beta \max_{a' \in \mathcal{A}} \widetilde{Q}^{old}\left(w', a'\right)\right] - \widetilde{Q}^{old}\left(w, \text{reject}\right), \; w' \sim F \end{aligned} \tag{60.9}$$

It turns out that formulas (60.9) combined with our Q-learning recursion (60.7) can lead our agent to eventually learn the optimal value function as well as in the case where an option to redraw can be exercised.

But learning is slower because an agent who ends up accepting a wage offer prematurally loses the option to explore new states in the same episode and to adjust the value associated with that state.

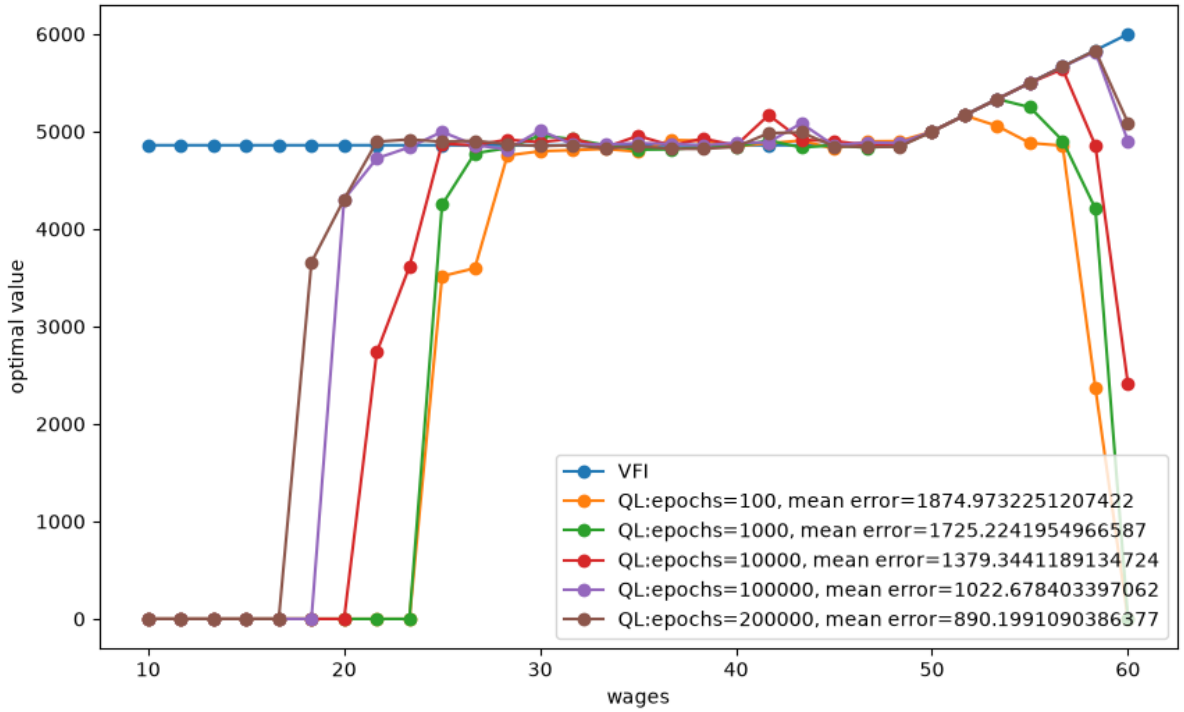
This can lead to inferior outcomes when the number of epochs/episodes is low.

But if we increase the number of epochs/episodes, we can observe that the error decreases and the outcomes get better.

We illustrate these possibilities with the following code and graph.

```
plot_epochs(epochs_to_plot=[100, 1000, 10000, 100000, 200000], quit_allowed=0)
```

```
Progress: EPOCHs = 0
Progress: EPOCHs = 20000
Progress: EPOCHs = 40000
Progress: EPOCHs = 60000
Progress: EPOCHs = 80000
Progress: EPOCHs = 100000
Progress: EPOCHs = 120000
Progress: EPOCHs = 140000
Progress: EPOCHs = 160000
Progress: EPOCHs = 180000
Progress: EPOCHs = 200000
```



60.7 Possible Extensions

To extend the algorthm to handle problems with continuous state spaces, a typical approach is to restrict Q-functions and policy functions to take particular functional forms.

This is the approach in **deep Q-learning** where the idea is to use a multilayer neural network as a good function approximator.

We will take up this topic in a subsequent quantecon lecture.

Part IX

Introduction to Optimal Savings

CHAPTER

SIXTYONE

OPTIMAL SAVINGS I: CAKE EATING

Contents

- *Optimal Savings I: Cake Eating*
  - *Overview*
  - *The model*
  - *The value function*
  - *The optimal policy*
  - *The Euler equation*
  - *Exercises*

61.1 Overview

In this lecture we introduce a simple "cake eating" problem.

The intertemporal problem is: how much to enjoy today and how much to leave for the future?

This trade-off between current and future rewards is at the heart of many savings and consumption problems.

Once we master the ideas in a simple environment, we will apply them to progressively more challenging problems.

The main tool we will use to solve the cake eating problem is dynamic programming.

The following lectures contain background on dynamic programming and might be worth reviewing:

- The shortest paths lecture
- The *basic McCall model*
- The *McCall model with separation*
- The *McCall model with separation and a continuous wage distribution*

We require the following imports:

```
import matplotlib.pyplot as plt
import numpy as np
```

61.2 The model

We consider a discrete time model with an infinite time horizon.

At $t = 0$ the agent is given a complete cake with size $\bar{x}$.

Let x_t denote the size of the cake at the beginning of each period, so that, in particular, $x_0 = \bar{x}$.

We choose how much of the cake to eat in any given period t.

After choosing to consume c_t of the cake in period t, the amount left in period $t + 1$ is

$$x_{t+1} = x_t - c_t$$

Consuming quantity c of the cake gives current utility $u(c)$.

We adopt the CRRA utility function

$$u(c) = \frac{c^{1-\gamma}}{1-\gamma} \qquad (\gamma > 0,\ \gamma \neq 1) \tag{61.1}$$

In Python this is

```
def u(c, γ):

    return c**(1 - γ) / (1 - γ)
```

Future cake consumption utility is discounted according to $\beta \in (0, 1)$.

In particular, consumption of c units t periods hence has present value $\beta^t u(c)$

The agent's problem can be written as

$$\max_{\{c_t\}} \sum_{t=0}^{\infty} \beta^t u(c_t) \tag{61.2}$$

subject to

$$x_{t+1} = x_t - c_t \quad \text{and} \quad 0 \leq c_t \leq x_t \tag{61.3}$$

for all t.

A consumption path $\{c_t\}$ satisfying (61.3) and $x_0 = \bar{x}$ is called **feasible**.

In this problem, the following terminology is standard:

- x_t is called the **state variable**
- c_t is called the **control variable** or the **action**
- β and γ are **parameters**

61.2.1 Trade-off

The key trade-off in the cake-eating problem is this:

- Delaying consumption is costly because of the discount factor.
- But delaying some consumption is also attractive because u is concave.

The concavity of u implies that the consumer gains value from **consumption smoothing**, which means spreading consumption out over time.

This is because concavity implies diminishing marginal utility—a progressively smaller gain in utility for each additional spoonful of cake consumed within one period.

61.2.2 Intuition

The reasoning given above suggests that the discount factor β and the curvature parameter γ will play a key role in determining the rate of consumption.

Here's an educated guess as to what impact these parameters will have.

1. Higher β implies less discounting, and hence the agent is more patient, which should reduce the rate of consumption.
2. Higher γ implies more curvature in u, more desire for consumption smoothing, and hence a lower rate of consumption.

Note

More formally, higher γ implies a lower intertemporal elasticity of substitution, since IES = $1/\gamma$.

This means the consumer is less willing to substitute consumption between periods.

This stronger preference for consumption smoothing results in a lower consumption rate.

In summary, we expect the rate of consumption to be decreasing in both parameters.

Let's see if this is true.

61.3 The value function

The first step of our dynamic programming treatment is to obtain the Bellman equation.

The next step is to use it to calculate the solution.

61.3.1 The Bellman equation

To this end, we let $v(x)$ be maximum lifetime utility attainable from the current time when x units of cake are left.

That is,

$$v(x) = \max \sum_{t=0}^{\infty} \beta^t u(c_t) \tag{61.4}$$

where the maximization is over all paths $\{c_t\}$ that are feasible from $x_0 = x$.

At this point, we do not have an expression for v, but we can still make inferences about it.

For example, as was the case with the *McCall model*, the value function will satisfy a version of the *Bellman equation*.

In the present case, this equation states that v satisfies

$$v(x) = \max_{0 \leq c \leq x} \{u(c) + \beta v(x - c)\} \quad \text{for any given } x \geq 0. \tag{61.5}$$

The intuition here is essentially the same as it was for the McCall model.

Choosing c optimally means trading off current vs future rewards.

Current rewards from choice c are just $u(c)$.

Future rewards given current cake size x, measured from next period and assuming optimal behavior, are $v(x - c)$.

These are the two terms on the right hand side of (61.5), after suitable discounting.

If c is chosen optimally using this trade off strategy, then we obtain maximal lifetime rewards from our current state x.

Hence, $v(x)$ equals the right hand side of (61.5), as claimed.

61.3.2 An analytical solution

It has been shown that, with u as the CRRA utility function in (61.1), the function v^* given by

$$v^*(x) = \left(1 - \beta^{1/\gamma}\right)^{-\gamma} u(x) \tag{61.6}$$

solves the Bellman equation and hence is equal to the value function.

You are asked to confirm that this is true in the exercises below.

Note

The solution (61.6) depends heavily on the CRRA utility function.

In fact, if we move away from CRRA utility, usually there is no analytical solution at all.

In other words, beyond CRRA utility, we know that the value function still satisfies the Bellman equation, but we do not have a way of writing it explicitly, as a function of the state variable and the parameters.

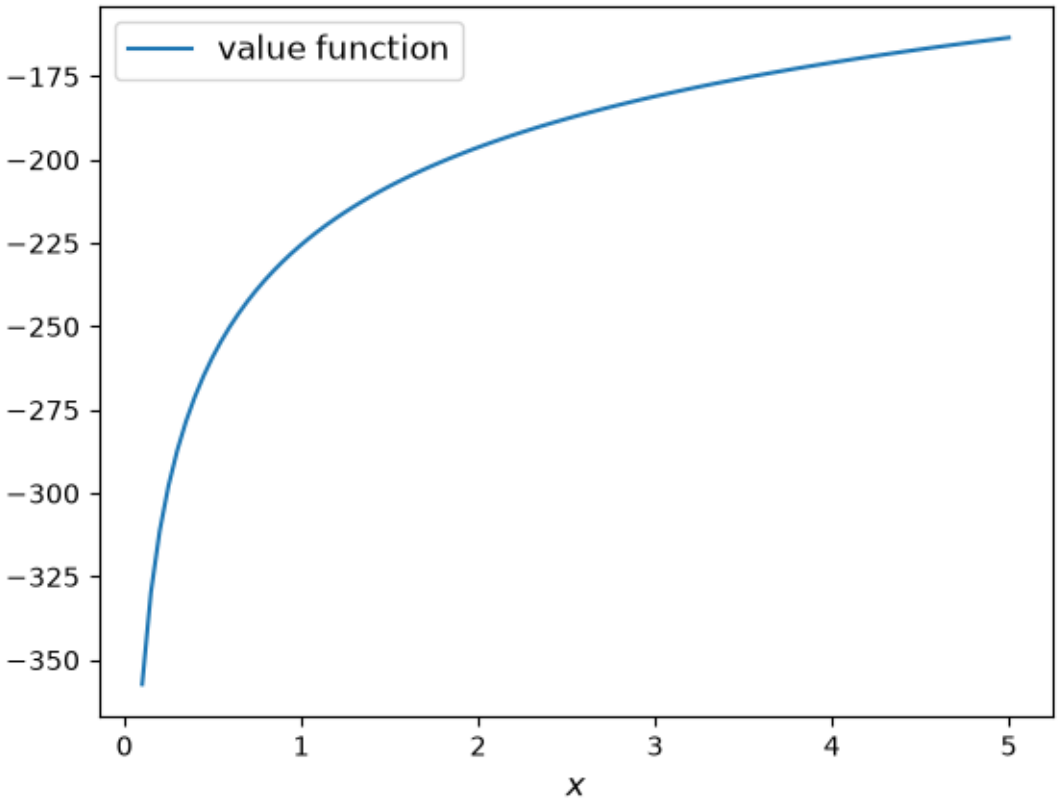
We will deal with that situation numerically in the following lectures.

Here is a Python representation of the value function:

```
def v_star(x, β, γ):

    return (1 - β**(1 / γ))**(-γ) * u(x, γ)
```

And here's a figure showing the function for fixed parameters:

```
β, γ = 0.95, 1.2
x_grid = np.linspace(0.1, 5, 100)

fig, ax = plt.subplots()

ax.plot(x_grid, v_star(x_grid, β, γ), label='value function')

ax.set_xlabel('$x$', fontsize=12)
ax.legend(fontsize=12)

plt.show()
```



61.4 The optimal policy

Now that we have the value function v^*, it is straightforward to calculate the optimal action at each state.

We should choose consumption to maximize the right hand side of the Bellman equation (61.5).

$$c^* = \arg \max_{0 \leq c \leq x} \{u(c) + \beta v^*(x - c)\}$$

We can think of this optimal choice as a *function* of the state x, in which case we call it the **optimal policy**.

We denote the optimal policy by σ^*, so that

$$\sigma^*(x) := \arg \max_c \{u(c) + \beta v^*(x - c)\} \quad \text{for all} \quad x \geq 0$$

If we plug the analytical expression (61.6) for the value function into the right hand side and compute the optimum, we find that

$$\sigma^*(x) = \left(1 - \beta^{1/\gamma}\right) x \tag{61.7}$$

Now let's recall our intuition on the impact of parameters.

We guessed that consumption would be decreasing in both parameters.

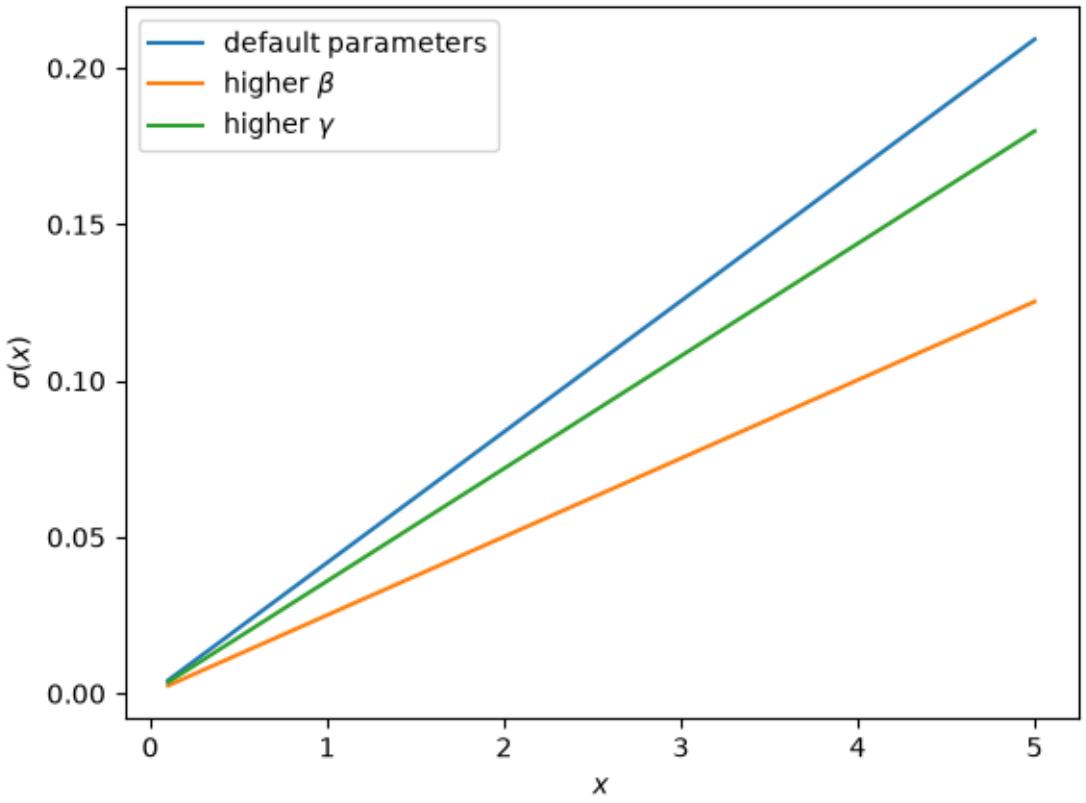
This is in fact the case, as can be seen from (61.7).

Here are some plots that illustrate.

```
def c_star(x, β, γ):

    return (1 - β ** (1/γ)) * x
```

Continuing with the values for β and γ used above, the plot is

```
fig, ax = plt.subplots()
ax.plot(x_grid, c_star(x_grid, β, γ), label='default parameters')
ax.plot(x_grid, c_star(x_grid, β + 0.02, γ), label=r'higher $\beta$')
ax.plot(x_grid, c_star(x_grid, β, γ + 0.2), label=r'higher $\gamma$')
ax.set_ylabel(r'$\sigma(x)$')
ax.set_xlabel('$x$')
ax.legend()

plt.show()
```



61.5 The Euler equation

In the discussion above we have provided a complete solution to the cake eating problem in the case of CRRA utility.

There is in fact another way to solve for the optimal policy, based on the so-called **Euler equation**.

Although we already have a complete solution, now is a good time to study the Euler equation.

This is because, for more difficult problems, this equation provides key insights that are hard to obtain by other methods.

61.5.1 Statement and implications

The Euler equation for the present problem can be stated as

$$u'(c_t^*) = \beta u'(c_{t+1}^*) \tag{61.8}$$

This is a necessary condition for an optimal consumption path $\{c_t^*\}_{t\geq 0}$.

It says that, along the optimal path, marginal rewards are equalized across time, after appropriate discounting.

This makes sense: optimality is obtained by smoothing consumption up to the point where no marginal gains remain.

We can also state the Euler equation in terms of the policy function.

A **feasible consumption policy** is a map $x \mapsto \sigma(x)$ satisfying $0 \leq \sigma(x) \leq x$.

(The last restriction says that we cannot consume more than the remaining quantity of cake.)

A feasible consumption policy σ is said to **satisfy the Euler equation** if, for all $x > 0$,

$$u'(\sigma(x)) = \beta u'(\sigma(x - \sigma(x))) \tag{61.9}$$

Evidently (61.9) is just the policy equivalent of (61.8).

It turns out that a feasible policy is optimal if and only if it satisfies the Euler equation.

In the exercises, you are asked to verify that the optimal policy (61.7) does indeed satisfy this functional equation.

Note

A **functional equation** is an equation where the unknown object is a function.

For a proof of sufficiency of the Euler equation in a very general setting, see proposition 2.2 of [Ma *et al.*, 2020].

The following arguments focus on necessity, explaining why an optimal path or policy should satisfy the Euler equation.

61.5.2 Derivation I: a perturbation approach

Let's write c as a shorthand for consumption path $\{c_t\}_{t=0}^{\infty}$.

The overall cake-eating maximization problem can be written as

$$\max_{c \in F} U(c) \quad \text{where } U(c) := \sum_{t=0}^{\infty} \beta^t u(c_t)$$

and F is the set of feasible consumption paths.

We know that differentiable functions have a zero gradient at a maximizer.

So the optimal path $c^* := \{c_t^*\}_{t=0}^{\infty}$ must satisfy $U'(c^*) = 0$.

Note

If you want to know exactly how the derivative $U'(c^*)$ is defined, given that the argument c^* is a vector of infinite length, you can start by learning about Gateaux derivatives. However, such knowledge is not assumed in what follows.

In other words, the rate of change in U must be zero for any infinitesimally small (and feasible) perturbation away from the optimal path.

So consider a feasible perturbation that reduces consumption at time t to $c_t^* - h$ and increases it in the next period to $c_{t+1}^* + h$.

Consumption does not change in any other period.

We call this perturbed path c^h.

By the preceding argument about zero gradients, we have

$$\lim_{h \to 0} \frac{U(c^h) - U(c^*)}{h} = U'(c^*) = 0$$

Recalling that consumption only changes at t and $t+1$, this becomes

$$\lim_{h \to 0} \frac{\beta^t u(c_t^* - h) + \beta^{t+1} u(c_{t+1}^* + h) - \beta^t u(c_t^*) - \beta^{t+1} u(c_{t+1}^*)}{h} = 0$$

After rearranging, the same expression can be written as

$$\lim_{h \to 0} \frac{u(c_t^* - h) - u(c_t^*)}{h} + \beta \lim_{h \to 0} \frac{u(c_{t+1}^* + h) - u(c_{t+1}^*)}{h} = 0$$

or, taking the limit,

$$-u'(c_t^*) + \beta u'(c_{t+1}^*) = 0$$

This is just the Euler equation.

61.5.3 Derivation II: using the Bellman equation

Another way to derive the Euler equation is to use the Bellman equation (61.5).

Note

The argument that follows assumes that the value function is differentiable. A proof of differentiability of the value function can be found in EDTC, theorem 10.1.13.

Taking the derivative on the right hand side of the Bellman equation with respect to c and setting it to zero, we get

$$u'(c) = \beta v'(x - c) \tag{61.10}$$

To obtain $v'(x-c)$, we set $g(c, x) = u(c) + \beta v(x - c)$, so that, at the optimal choice of consumption,

$$v(x) = g(c, x) \tag{61.11}$$

Differentiating both sides while acknowledging that the maximizing consumption will depend on x, we get

$$v'(x) = \frac{\partial}{\partial c} g(c, x) \frac{\partial c}{\partial x} + \frac{\partial}{\partial x} g(c, x)$$

When $g(c, x)$ is maximized at c, we have $\frac{\partial}{\partial c} g(c, x) = 0$.

Hence the derivative simplifies to

$$v'(x) = \frac{\partial g(c, x)}{\partial x} = \frac{\partial}{\partial x} \beta v(x - c) = \beta v'(x - c) \tag{61.12}$$

(This argument is an example of the Envelope Theorem.)

But now an application of (61.10) gives

$$u'(c) = v'(x) \tag{61.13}$$

Thus, the derivative of the value function is equal to marginal utility.

Combining this fact with (61.12) recovers the Euler equation.

61.6 Exercises

Exercise 61.6.1

How does one obtain the expressions for the value function and optimal policy given in (61.6) and (61.7) respectively?

The first step is to make a guess of the functional form for the consumption policy.

So suppose that we do not know the solutions and start with a guess that the optimal policy is linear.

In other words, we conjecture that there exists a positive θ such that setting $c_t^* = \theta x_t$ for all t produces an optimal path.

Starting from this conjecture, try to obtain the solutions (61.6) and (61.7).

In doing so, you will need to use the definition of the value function and the Bellman equation.

Solution

We start with the conjecture $c_t^* = \theta x_t$, which leads to a path for the state variable (cake size) given by

$$x_{t+1} = x_t(1-\theta)$$

Then $x_t = x_0(1-\theta)^t$ and hence

$$\begin{aligned} v(x_0) &= \sum_{t=0}^{\infty} \beta^t u(\theta x_t) \\ &= \sum_{t=0}^{\infty} \beta^t u(\theta x_0 (1-\theta)^t) \\ &= \sum_{t=0}^{\infty} \theta^{1-\gamma} \beta^t (1-\theta)^{t(1-\gamma)} u(x_0) \\ &= \frac{\theta^{1-\gamma}}{1-\beta(1-\theta)^{1-\gamma}} u(x_0) \end{aligned}$$

From the Bellman equation, then,

$$\begin{aligned} v(x) &= \max_{0 \le c \le x} \left\{ u(c) + \beta \frac{\theta^{1-\gamma}}{1-\beta(1-\theta)^{1-\gamma}} \cdot u(x-c) \right\} \\ &= \max_{0 \le c \le x} \left\{ \frac{c^{1-\gamma}}{1-\gamma} + \beta \frac{\theta^{1-\gamma}}{1-\beta(1-\theta)^{1-\gamma}} \cdot \frac{(x-c)^{1-\gamma}}{1-\gamma} \right\} \end{aligned}$$

From the first order condition, we obtain

$$c^{-\gamma} + \beta \frac{\theta^{1-\gamma}}{1-\beta(1-\theta)^{1-\gamma}} \cdot (x-c)^{-\gamma}(-1) = 0$$

or

$$c^{-\gamma} = \beta \frac{\theta^{1-\gamma}}{1-\beta(1-\theta)^{1-\gamma}} \cdot (x-c)^{-\gamma}$$

With $c = \theta x$ we get

$$(\theta x)^{-\gamma} = \beta \frac{\theta^{1-\gamma}}{1-\beta(1-\theta)^{1-\gamma}} \cdot (x(1-\theta))^{-\gamma}$$

Some rearrangement produces

$$\theta = 1 - \beta^{\frac{1}{\gamma}}$$

This confirms our earlier expression for the optimal policy:

$$c_t^* = \left(1 - \beta^{\frac{1}{\gamma}}\right) x_t$$

Substituting θ into the value function above gives

$$v^*(x_t) = \frac{\left(1-\beta^{\frac{1}{\gamma}}\right)^{1-\gamma}}{1-\beta\left(\beta^{\frac{1-\gamma}{\gamma}}\right)} u(x_t)$$

Rearranging gives

$$v^*(x_t) = \left(1-\beta^{\frac{1}{\gamma}}\right)^{-\gamma} u(x_t)$$

Our claims are now verified.

Exercise 61.6.2

Verify that the optimal policy (61.7) satisfies the Euler equation (61.9).

Solution

Recall that the optimal policy is

$$\sigma^*(x) = (1-\beta^{1/\gamma})x$$

and the Euler equation in policy form is

$$u'(\sigma(x)) = \beta u'(\sigma(x-\sigma(x)))$$

With CRRA utility $u(c) = \frac{c^{1-\gamma}}{1-\gamma}$, the marginal utility is

$$u'(c) = c^{-\gamma}$$

Now let's verify the Euler equation. The left-hand side is

$$u'(\sigma^*(x)) = [\sigma^*(x)]^{-\gamma} = [(1-\beta^{1/\gamma})x]^{-\gamma} = (1-\beta^{1/\gamma})^{-\gamma} x^{-\gamma}$$

For the right-hand side, we first compute the state in the next period:

$$x - \sigma^*(x) = x - (1 - \beta^{1/\gamma})x = x\beta^{1/\gamma}$$

Next period's consumption under the optimal policy is

$$\sigma^*(x - \sigma^*(x)) = \sigma^*(x\beta^{1/\gamma}) = (1 - \beta^{1/\gamma}) \cdot x\beta^{1/\gamma}$$

Therefore, the right-hand side of the Euler equation is

$$\begin{aligned}
\beta u'(\sigma^*(x - \sigma^*(x))) &= \beta[(1 - \beta^{1/\gamma}) \cdot x\beta^{1/\gamma}]^{-\gamma} \\
&= \beta(1 - \beta^{1/\gamma})^{-\gamma} x^{-\gamma} (\beta^{1/\gamma})^{-\gamma} \\
&= \beta(1 - \beta^{1/\gamma})^{-\gamma} x^{-\gamma} \beta^{-1} \\
&= (1 - \beta^{1/\gamma})^{-\gamma} x^{-\gamma}
\end{aligned}$$

Since the left-hand side equals the right-hand side, the optimal policy σ^* satisfies the Euler equation.

CHAPTER

SIXTYTWO

OPTIMAL SAVINGS II: NUMERICAL CAKE EATING

Contents

- *Optimal Savings II: Numerical Cake Eating*
  - *Overview*
  - *Reviewing the Model*
  - *Value Function Iteration*
  - *Exercises*

62.1 Overview

In this lecture we continue the study of the problem described in *Optimal Savings I: Cake Eating*.

The aim of this lecture is to solve the problem using numerical methods.

At first this might appear unnecessary, since we already obtained the optimal policy analytically.

However, the cake eating problem is too simple to be useful without modifications, and once we start modifying the problem, numerical methods become essential.

Hence it makes sense to introduce numerical methods now, and test them on this simple problem.

Since we know the analytical solution, this will allow us to assess the accuracy of alternative numerical methods.

Note

The code below aims for clarity rather than maximum efficiency.

In the lectures below we will explore best practice for speed and efficiency.

Let's put these algorithm and code optimizations to one side for now.

We will use the following imports:

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.optimize import minimize_scalar, bisect
from typing import NamedTuple
```

62.2 Reviewing the Model

You might like to review the details in *Optimal Savings I: Cake Eating* before we start.

Recall in particular that the Bellman equation is

$$v(x) = \max_{0 \leq c \leq x} \{u(c) + \beta v(x - c)\} \quad \text{for all } x \geq 0. \tag{62.1}$$

where u is the CRRA utility function.

The analytical solutions for the value function and optimal policy were found to be as follows.

```
def c_star(x, β, γ):

    return (1 - β ** (1/γ)) * x


def v_star(x, β, γ):

    return (1 - β**(1 / γ))**(-γ) * (x**(1-γ) / (1-γ))
```

Our first aim is to obtain these analytical solutions numerically.

62.3 Value Function Iteration

The first approach we will take is **value function iteration**.

This is a form of **successive approximation**, and was discussed in our *lecture on job search*.

The basic idea is:

1. Take an arbitrary initial guess of v.
2. Obtain an update $\hat{v}$ defined by

$$\hat{v}(x) = \max_{0 \leq c \leq x} \{u(c) + \beta v(x - c)\}$$

3. Stop if $\hat{v}$ is approximately equal to v, otherwise set $v = \hat{v}$ and go back to step 2.

Let's write this a bit more mathematically.

62.3.1 The Bellman Operator

We introduce the **Bellman operator** T that takes a function v as an argument and returns a new function Tv defined by

$$Tv(x) = \max_{0 \leq c \leq x} \{u(c) + \beta v(x - c)\}$$

From v we get Tv, and applying T to this yields $T^2 v := T(Tv)$ and so on.

This is called **iterating with the Bellman operator** from initial guess v.

As we discuss in more detail in later lectures, one can use Banach's contraction mapping theorem to prove that the sequence of functions $T^n v$ converges to the solution to the Bellman equation.

62.3.2 Fitted Value Function Iteration

Both consumption c and the state variable x are continuous.

This causes complications when it comes to numerical work.

For example, we need to store each function $T^n v$ in order to compute the next iterate $T^{n+1} v$.

But this means we have to store $T^n v(x)$ at infinitely many x, which is, in general, impossible.

To circumvent this issue we will use fitted value function iteration, as discussed previously in *one of the lectures* on job search.

The process looks like this:

1. Begin with an array of values $\{v_0, \ldots, v_I\}$ representing the values of some initial function v on the grid points $\{x_0, \ldots, x_I\}$.
2. Build a function $\hat{v}$ on the state space $\mathbb{R}_+$ by interpolation, based on the interpolation points $\{(x_i, v_i)\}$.
3. Insert $\hat{v}$ into the right hand side of the Bellman equation and obtain and record the value $T\hat{v}(x_i)$ on each grid point x_i.
4. Unless some stopping condition is satisfied, set $\{v_0, \ldots, v_I\} = \{T\hat{v}(x_0), \ldots, T\hat{v}(x_I)\}$ and go to step 2.

In step 2 we'll use piecewise linear interpolation.

62.3.3 Implementation

The `maximize` function below is a small helper function that converts a SciPy minimization routine into a maximization routine.

```
def maximize(g, upper_bound):
    """
    Maximize the function g over the interval [0, upper_bound].

    We use the fact that the maximizer of g on any interval is
    also the minimizer of -g.

    """

    objective = lambda x: -g(x)
    bounds = (0, upper_bound)
    result = minimize_scalar(objective, bounds=bounds, method='bounded')
    maximizer, maximum = result.x, -result.fun
    return maximizer, maximum
```

We'll store the parameters β and γ and the grid in a `NamedTuple` called `Model`.

We'll also create a helper function called `create_cake_eating_model` to store default parameters and build an instance of `Model`.

```
class Model(NamedTuple):
    β: float
    γ: float
    x_grid: np.ndarray

def create_cake_eating_model(
        β: float = 0.96,           # discount factor
        γ: float = 1.5,            # degree of relative risk aversion
```

(continues on next page)

(continued from previous page)

```
        x_grid_min: float = 1e-3,  # exclude zero for numerical stability
        x_grid_max: float = 2.5,   # size of cake
        x_grid_size: int = 120
    ):
    """
    Creates an instance of the cake eating model.

    """
    x_grid = np.linspace(x_grid_min, x_grid_max, x_grid_size)
    return Model(β, γ, x_grid)
```

Here's the CRRA utility function.

```
def u(c, γ):
    return (c ** (1 - γ)) / (1 - γ)
```

To work with the Bellman equation, let's write it as

$$v(x) = \max_{0 \leq c \leq x} B(x, c, v)$$

where

$$B(x, c, v) := u(c) + \beta v(x - c)$$

Now we implement the function B.

```
def B(
        x: float,               # the current state (remaining cake)
        c: float,               # current consumption
        v: np.ndarray,          # current guess of the value function
        model: Model            # instance of cake eating model
    ):
    """
    Right hand side of the Bellman equation given x and c.

    """
    # Unpack (simplify names)
    β, γ, x_grid = model

    # Convert array v into a function by linear interpolation
    vf = lambda x: np.interp(x, x_grid, v)

    # Return B(x, c, v)
    return u(c, γ) + β * vf(x - c)
```

We now define the Bellman operator acting on grid points:

$$Tv(x_i) = \max_{0 \leq c \leq x_i} B(x_i, c, v) \qquad \text{for all } i$$

```
def T(
        v: np.ndarray,          # Current guess of the value function
        model: Model            # Instance of the cake eating model
    ):
    " The Bellman operator.  Updates the guess of the value function. "

    # Allocate memory for the new array v_new = Tv
```

(continues on next page)

(continued from previous page)

```
    v_new = np.empty_like(v)

    # Calculate Tv(x) for all x
    for i, x in enumerate(model.x_grid):
        # Maximize RHS of Bellman equation with respect to c over [0, x]
        _, v_new[i] = maximize(lambda c: B(x, c, v, model), x)

    return v_new
```

After defining the Bellman operator, we are ready to solve the model.

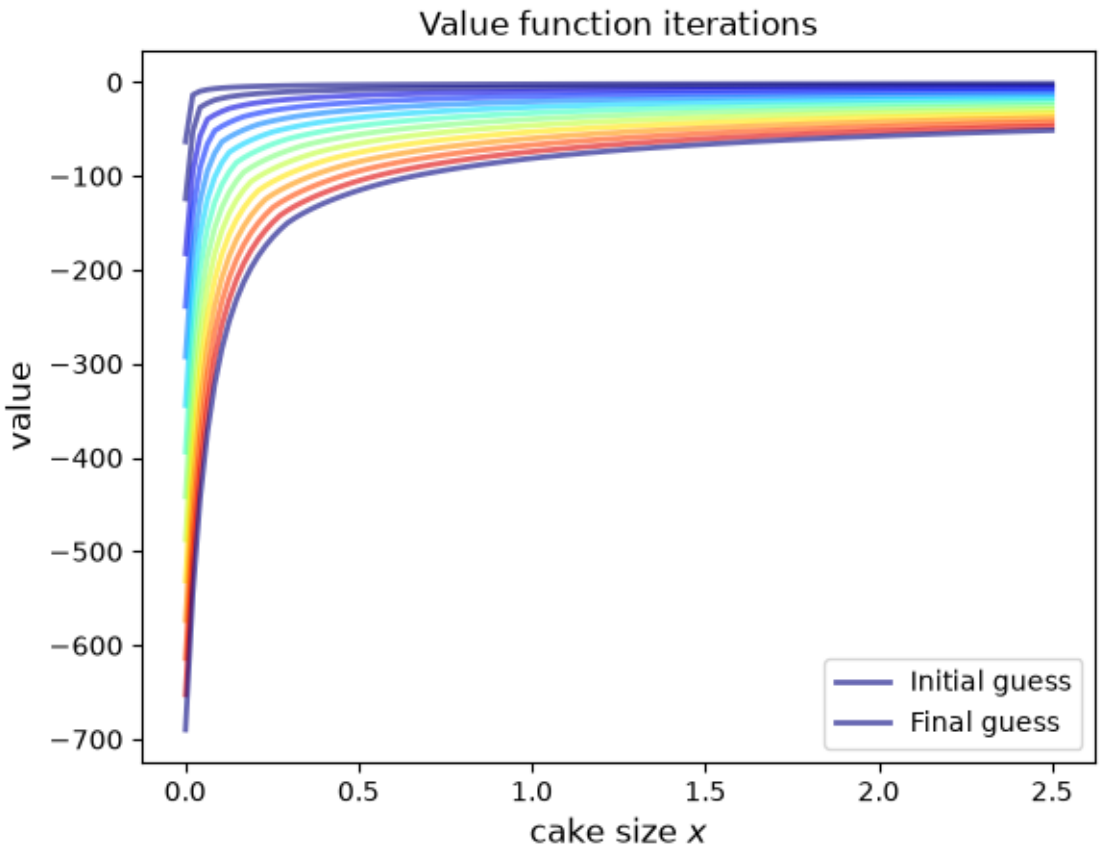
Let's start by creating a model using the default parameterization.

```
model = create_cake_eating_model()
β, γ, x_grid = model
```

Now let's see iteration of the value function in action.

We start from guess v given by $v(x) = u(x)$ for every x grid point.

```
v = u(x_grid, γ)  # Initial guess
n = 12                  # Number of iterations
fig, ax = plt.subplots()

# Initial plot
ax.plot(x_grid, v, color=plt.cm.jet(0),
        lw=2, alpha=0.6, label='Initial guess')

# Iterate
for i in range(n):
    v = T(v, model)  # Apply the Bellman operator
    ax.plot(x_grid, v, color=plt.cm.jet(i / n), lw=2, alpha=0.6)

# One last update and plot
v = T(v, model)
ax.plot(x_grid, v, color=plt.cm.jet(0),
        lw=2, alpha=0.6, label='Final guess')

ax.legend()
ax.set_ylabel('value', fontsize=12)
ax.set_xlabel('cake size $x$', fontsize=12)
ax.set_title('Value function iterations')
plt.show()
```



To iterate more systematically, we introduce a wrapper function called `compute_value_function`.

It's task is to iterate using T until some convergence conditions are satisfied.

```
def compute_value_function(
        model: Model,
        tol: float = 1e-4,
        max_iter: int = 1_000,
        verbose: bool = True,
        print_skip: int = 25
    ):

    # Set up loop
    v = np.zeros(len(model.x_grid)) # Initial guess
    i = 0
    error = tol + 1

    while i < max_iter and error > tol:
        v_new = T(v, model)

        error = np.max(np.abs(v - v_new))
        i += 1

        if verbose and i % print_skip == 0:
            print(f"Error at iteration {i} is {error}.")

        v = v_new
```

(continues on next page)

(continued from previous page)

```
    if error > tol:
        print("Failed to converge!")
    elif verbose:
        print(f"\nConverged in {i} iterations.")

    return v_new
```

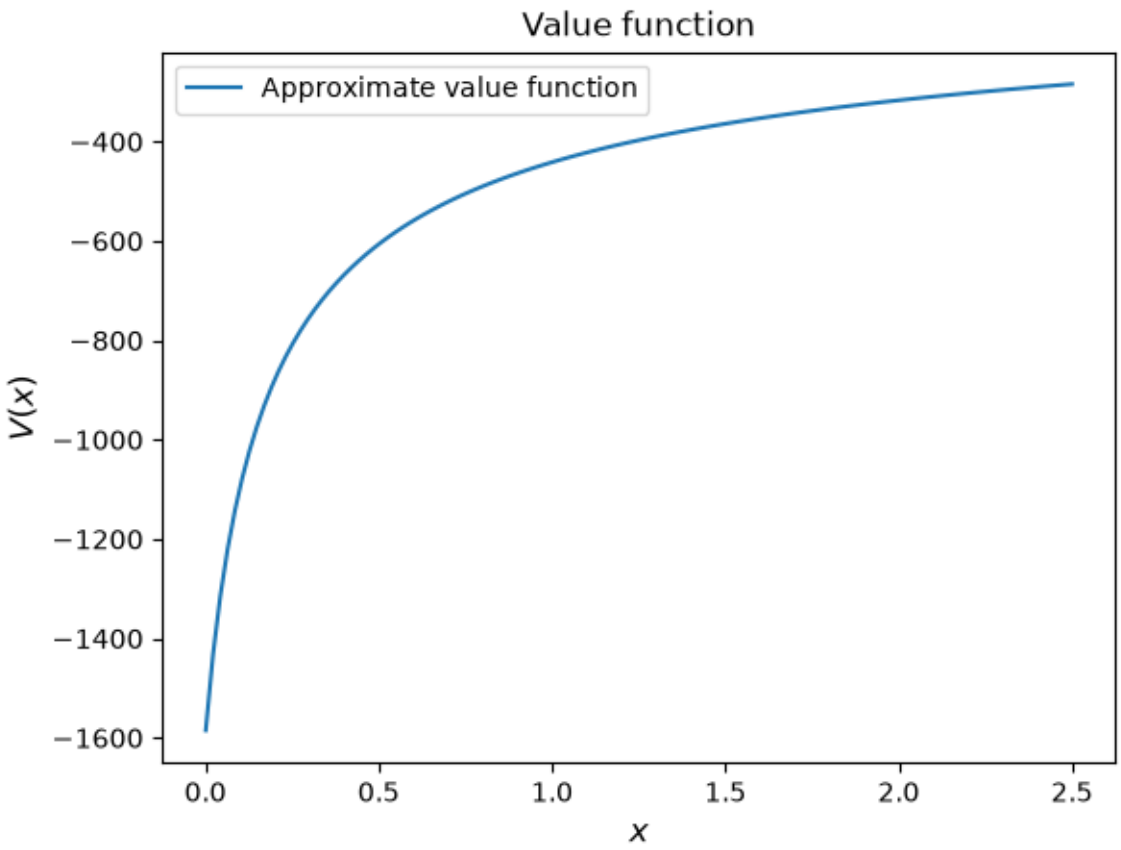
Now let's call it, noting that it takes a little while to run.

```
v = compute_value_function(model)
```

```
Error at iteration 25 is 23.80037552320357.
Error at iteration 50 is 8.577577198550443.
Error at iteration 75 is 3.091330660953872.
Error at iteration 100 is 1.1141054209301728.
Error at iteration 125 is 0.4015199359369035.
Error at iteration 150 is 0.1447064666647293.
Error at iteration 175 is 0.05215173549413521.
Error at iteration 200 is 0.018795314250610318.
Error at iteration 225 is 0.006773769548317432.
Error at iteration 250 is 0.0024412443060555233.
Error at iteration 275 is 0.0008798164330414693.
Error at iteration 300 is 0.0003170829540977138.
Error at iteration 325 is 0.00011427565573285392.

Converged in 329 iterations.
```

Now we can plot and see what the converged value function looks like.

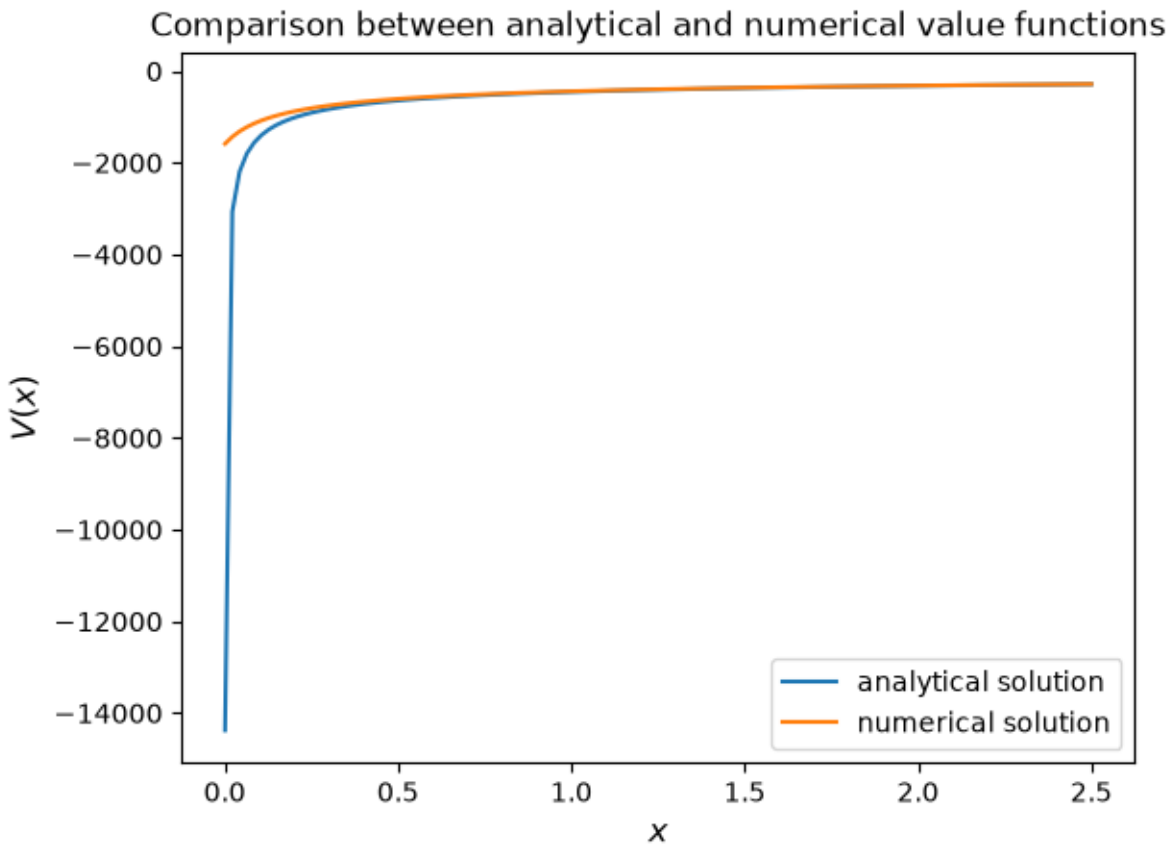
```
fig, ax = plt.subplots()

ax.plot(x_grid, v, label='Approximate value function')
ax.set_ylabel('$V(x)$', fontsize=12)
ax.set_xlabel('$x$', fontsize=12)
ax.set_title('Value function')
ax.legend()
plt.show()
```



Next let's compare it to the analytical solution.

```
v_analytical = v_star(x_grid, β, γ)
```

```
fig, ax = plt.subplots()

ax.plot(x_grid, v_analytical, label='analytical solution')
ax.plot(x_grid, v, label='numerical solution')
ax.set_ylabel('$V(x)$', fontsize=12)
ax.set_xlabel('$x$', fontsize=12)
ax.legend()
ax.set_title('Comparison between analytical and numerical value functions')
plt.show()
```



The quality of approximation is reasonably good for large x, but less so near the lower boundary.

The reason is that the utility function and hence value function is very steep near the lower boundary, and hence hard to approximate.

Note

One way to fix this issue is to use a nonlinear grid, with more points in the neighborhood of zero.

Instead of pursuing this idea, however, we will turn our attention to working with policy functions.

We will see that value function iteration can be avoided by iterating on a guess of the policy function instead.

The policy function has less curvature and hence is easier to interpolate than the value function.

These ideas will be explored over the next few lectures.

62.3.4 Policy Function

Let's try computing the optimal policy.

In *Optimal Savings I: Cake Eating*, the optimal consumption policy was shown to be

$$\sigma^*(x) = \left(1 - \beta^{1/\gamma}\right) x$$

Let's see if our numerical results lead to something similar.

Our numerical strategy will be to compute, for any given v, the policy

$$\sigma(x) = \arg \max_{0 \leq c \leq x} \{u(c) + \beta v(x - c)\}$$

This policy is called the v-**greedy policy**.

In practice we will compute σ on a grid of x points and then interpolate.

For v we will use the approximation of the value function we obtained above.

Here's the function:

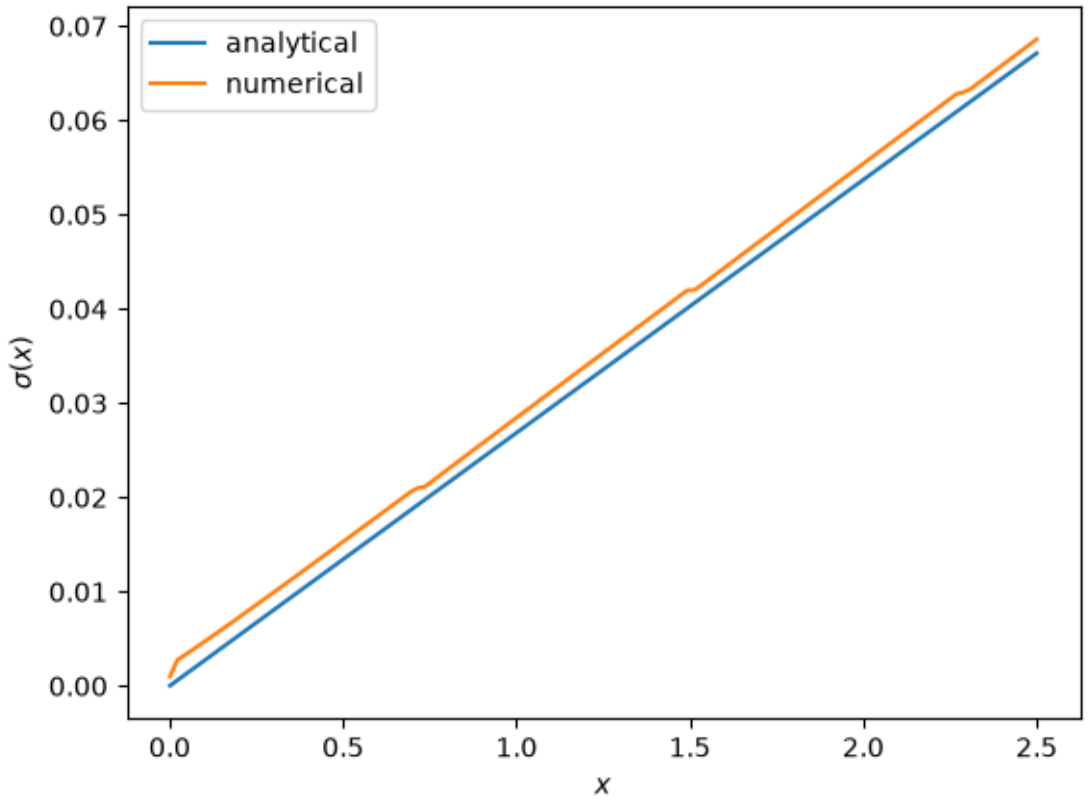
```
def get_greedy(
        v: np.ndarray,          # current guess of the value function
        model: Model            # instance of cake eating model
    ):
    " Compute the v-greedy policy on x_grid."

    σ = np.empty_like(v)

    for i, x in enumerate(model.x_grid):
        # Maximize RHS of Bellman equation at state x
        σ[i], _ = maximize(lambda c: B(x, c, v, model), x)

    return σ
```

Now let's pass the approximate value function and compute optimal consumption:

```
σ = get_greedy(v, model)
```

Let's plot this next to the true analytical solution

```
c_analytical = c_star(model.x_grid, model.β, model.γ)

fig, ax = plt.subplots()

ax.plot(model.x_grid, c_analytical, label='analytical')
ax.plot(model.x_grid, σ, label='numerical')
ax.set_ylabel(r'$\sigma(x)$')
ax.set_xlabel('$x$')
ax.legend()

plt.show()
```



The fit is reasonable but not perfect.

We can improve it by increasing the grid size or reducing the error tolerance in the value function iteration routine.

However, both changes will lead to a longer compute time.

Another possibility is to use an alternative algorithm, which offers the possibility of faster compute time and, at the same time, more accuracy.

We explore this in *Optimal Savings IV: Time Iteration*.

62.4 Exercises

Exercise 62.4.1

Try the following modification of the problem.

Instead of the cake size changing according to $x_{t+1} = x_t - c_t$, let it change according to

$$x_{t+1} = (x_t - c_t)^\alpha$$

where α is a parameter satisfying $0 < \alpha < 1$.

(We will see this kind of update rule when we study optimal growth models.)

Make the required changes to value function iteration code and plot the value and policy functions.

Try to reuse as much code as possible.

Solution

We need to create an extended version of our model and state-action value function.

We'll create a new `NamedTuple` for the extended cake model and a helper function.

```
# Create extended cake model data structure
class ExtendedModel(NamedTuple):
    β: float
    γ: float
    α: float
    x_grid: np.ndarray

def create_extended_model(β=0.96,           # discount factor
                          γ=1.5,            # degree of relative risk aversion
                          α=0.4,            # productivity parameter
                          x_grid_min=1e-3,  # exclude zero for numerical stability
                          x_grid_max=2.5,   # size of cake
                          x_grid_size=120):
    """
    Creates an instance of the extended cake eating model.
    """
    x_grid = np.linspace(x_grid_min, x_grid_max, x_grid_size)
    return ExtendedModel(β, γ, α, x_grid)

def extended_B(c, x, v, model):
    """
    Right hand side of the Bellman equation for the extended cake model given x␣
 ↪and c.

    """
    β, γ, α, x_grid = model
    vf = lambda x: np.interp(x, x_grid, v)
    return u(c, γ) + β * vf((x - c)**α)
```

We also need a modified Bellman operator:

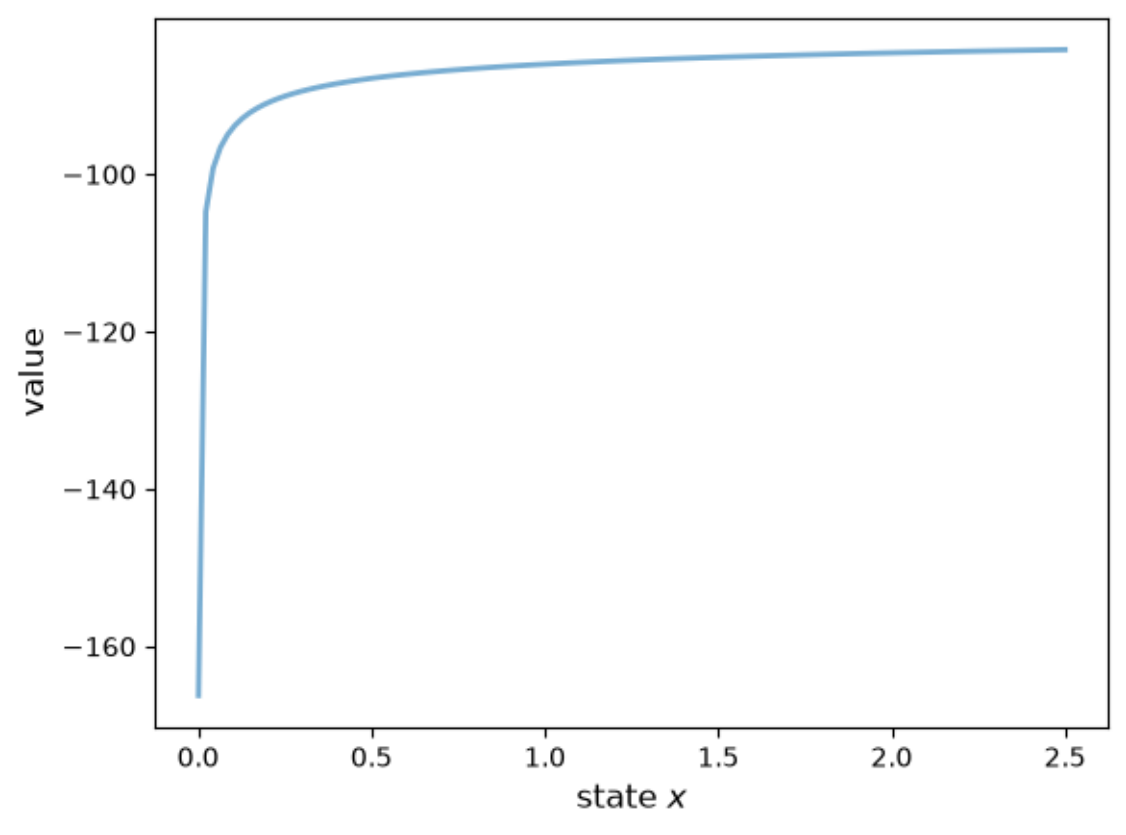
```
def extended_T(v, model):
    " The Bellman operator for the extended cake model. "

    v_new = np.empty_like(v)
    for i, x in enumerate(model.x_grid):
        _, v_new[i] = maximize(lambda c: extended_B(c, x, v, model), x)
    return v_new
```

Now create the model:

```
model = create_extended_model()
```

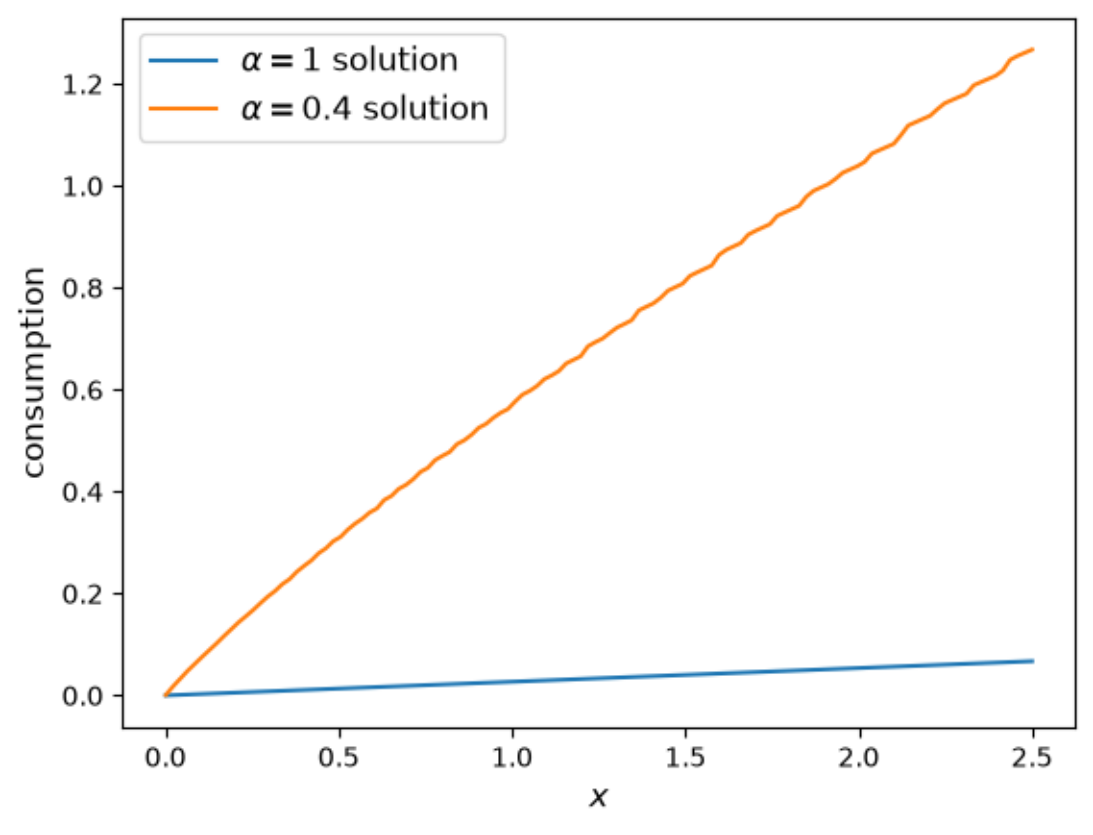
Here's a function to compute the value function.

```
def compute_value_function_extended(model,
                                    tol=1e-4,
                                    max_iter=1000,
                                    verbose=True,
                                    print_skip=25):
    """
    Compute value function for extended cake model.
    """
    v = np.zeros(len(model.x_grid))
    i = 0
    error = tol + 1

    while i < max_iter and error > tol:
        v_new = extended_T(v, model)
        error = np.max(np.abs(v - v_new))
        i += 1
        if verbose and i % print_skip == 0:
            print(f"Error at iteration {i} is {error}.")
        v = v_new

    if error > tol:
        print("Failed to converge!")
    elif verbose:
        print(f"\nConverged in {i} iterations.")

    return v_new

v = compute_value_function_extended(model, verbose=False)

fig, ax = plt.subplots()

ax.plot(model.x_grid, v, lw=2, alpha=0.6)
ax.set_ylabel('value', fontsize=12)
ax.set_xlabel('state $x$', fontsize=12)

plt.show()
```



Here's the computed policy, combined with the solution we derived above for the standard cake eating case $\alpha = 1$.

```
def extended_get_greedy(model, v):
    """
    The optimal policy function for the extended cake model.
    """
    σ = np.empty_like(v)

    for i, x in enumerate(model.x_grid):
        # Maximize extended_B with respect to c over [0, x]
        σ[i], _ = maximize(lambda c: extended_B(c, x, v, model), x)

    return σ

σ = extended_get_greedy(model, v)

# Get the baseline model for comparison
baseline_model = create_cake_eating_model()
c_analytical = c_star(baseline_model.x_grid, baseline_model.β, baseline_model.γ)

fig, ax = plt.subplots()

ax.plot(baseline_model.x_grid, c_analytical, label=r'$\alpha=1$ solution')
ax.plot(model.x_grid, σ, label=fr'$\alpha={model.α}$ solution')

ax.set_ylabel('consumption', fontsize=12)
ax.set_xlabel('$x$', fontsize=12)

ax.legend(fontsize=12)

plt.show()
```



Consumption is higher when $\alpha < 1$ because, at least for large x, the return to savings is lower.

CHAPTER

SIXTYTHREE

OPTIMAL SAVINGS III: STOCHASTIC RETURNS

Contents

- *Optimal Savings III: Stochastic Returns*
  - *Overview*
  - *The Model*
  - *Computation*
  - *Exercises*

63.1 Overview

In this lecture, we continue our study of optimal savings problems, building on *Optimal Savings I: Cake Eating* and *Optimal Savings II: Numerical Cake Eating*.

The key difference from the previous lectures is that wealth now evolves stochastically.

We can think of wealth as a harvest that regrows if we save some seeds.

Specifically, if we save and invest part of today's harvest x_t, it grows into next period's harvest x_{t+1} according to a stochastic production process.

The extensions in this lecture introduce several new elements:

- nonlinear returns to saving, through a production function, and
- stochastic returns, due to shocks to production.

Despite these additions, the model remains relatively tractable.

As a first pass, we will solve the model using dynamic programming and value function iteration (VFI).

Note

In later lectures we'll explore more efficient methods for this class of problems.

At the same time, VFI is foundational and globally convergent.

Hence we want to be sure we can use this method too.

More information on this savings problem can be found in

- [Ljungqvist and Sargent, 2018], Section 3.1
- EDTC, Chapter 1
- [Sundaram, 1996], Chapter 12

Let's start with some imports:

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.interpolate import interp1d
from scipy.optimize import minimize_scalar
from typing import NamedTuple, Callable
```

63.2 The Model

Here we described the new model and the optimization problem.

63.2.1 Setup

Consider an agent who owns an amount $x_t \in \mathbb{R}_+ := [0, \infty)$ of a consumption good at time t.

This output can either be consumed or saved and used for production.

Production is stochastic, in that it also depends on a shock ξ_{t+1} realized at the end of the current period.

Next period output is

$$x_{t+1} := f(s_t)\xi_{t+1}$$

where $f \colon \mathbb{R}_+ \to \mathbb{R}_+$ is the **production function** and

$$s_t = x_t - c_t \tag{63.1}$$

is **current savings**.

and all variables are required to be nonnegative.

In what follows,

- The sequence $\{\xi_t\}$ is assumed to be IID.
- The common distribution of each ξ_t will be denoted by ϕ.
- The production function f is assumed to be increasing and continuous.

63.2.2 Optimization

Taking x_0 as given, the agent wishes to maximize

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t u(c_t) \tag{63.2}$$

subject to

$$x_{t+1} = f(x_t - c_t)\xi_{t+1} \quad \text{and} \quad 0 \leq c_t \leq x_t \quad \text{for all } t \tag{63.3}$$

where

- u is a bounded, continuous and strictly increasing utility function and
- $\beta \in (0, 1)$ is a discount factor.

In summary, the agent's aim is to select a path $c_0, c_1, c_2, \ldots$ for consumption that is

1. nonnegative,
2. feasible,
3. optimal, in the sense that it maximizes (63.2) relative to all other feasible consumption sequences, and
4. **adapted**, in the sense that the current action c_t depends only on current and historical outcomes, not on future outcomes such as ξ_{t+1}.

In the present context

- x_t is called the **state** variable — it summarizes the "state of the world" at the start of each period.
- c_t is called the **control** variable — a value chosen by the agent each period after observing the state.

63.2.3 Optimal Policies

Let us look at **policy functions**, each one of which is a map σ from the current state x_t into a current action c_t.

Note

These kinds of policies are called Markov policies (or stationary Markov policies).

For this dynamic program, the optimal policy is always a Markov policy (see, e.g., DP1).

In essence, the current state x_t provides a sufficient statistic for the history in terms of making an optimal decision today.

In what follows, we will call σ a **feasible consumption policy** if it satisfies

$$0 \leq \sigma(x) \leq x \quad \text{for all} \quad x \in \mathbb{R}_+ \tag{63.4}$$

In other words, a feasible policy is a policy function that respects the resource constraint.

The set of all feasible consumption policies will be denoted by Σ.

Each $\sigma \in \Sigma$ determines a Markov dynamics for output $\{x_t\}$ via

$$x_{t+1} = f(x_t - \sigma(x_t))\xi_{t+1}, \quad x_0 \text{ given} \tag{63.5}$$

This is the time path for output when we choose and stick with the policy σ.

We insert this process into the objective function to get

$$\mathbb{E}\sum_{t=0}^{\infty} \beta^t u(c_t) = \mathbb{E}\sum_{t=0}^{\infty} \beta^t u(\sigma(x_t)) \tag{63.6}$$

This is the total expected present value of following policy σ forever, given initial income x_0.

The aim is to select a policy that makes this number as large as possible.

The next section covers these ideas more formally.

63.2.4 Optimality

The lifetime value v_σ associated with a given policy σ is the mapping defined by

$$v_\sigma(x) = \mathbb{E} \sum_{t=0}^{\infty} \beta^t u(\sigma(x_t)) \tag{63.7}$$

when $\{x_t\}$ is given by (63.5) with $x_0 = x$.

In other words, it is the lifetime value of following policy σ forever, starting at initial condition x.

The **value function** is then defined as

$$v^*(x) := \sup_{\sigma \in \Sigma} \; v_\sigma(x) \tag{63.8}$$

The value function gives the maximal value that can be obtained from state x, after considering all feasible policies.

A policy $\sigma \in \Sigma$ is called **optimal** if $v_\sigma(x) = v^*(x)$ for all $x \in \mathbb{R}_+$.

63.2.5 The Bellman Equation

The following equation is called the **Bellman equation** associated with this dynamic programming problem.

$$v(x) = \max_{0 \leq c \leq x} \left\{ u(c) + \beta \int v(f(x-c)z)\phi(dz) \right\} \qquad (x \in \mathbb{R}_+) \tag{63.9}$$

This is a *functional equation in* v, in the sense that a given v can either satisfy it or not satisfy it.

The term $\int v(f(x-c)z)\phi(dz)$ can be understood as the expected next period value when

- v is used to measure value
- the state is x
- consumption is set to c

As shown in DP1 and a range of other texts, the value function v^* satisfies the Bellman equation.

In other words, (63.9) holds when $v = v^*$.

The intuition is that maximal value from a given state can be obtained by optimally trading off

- current reward from a given action, vs
- expected discounted future value of the state resulting from that action

The Bellman equation is important because it

1. gives us more information about the value function and
2. suggests a way of computing the value function, which we discuss below.

63.2.6 Greedy Policies

The value function can be used to compute optimal policies.

Given a continuous function v on $\mathbb{R}_+$, we say that $\sigma \in \Sigma$ is v-**greedy** if

$$\sigma(x) \in \arg\max_{0 \leq c \leq x} \left\{ u(c) + \beta \int v(f(x-c)z)\phi(dz) \right\} \tag{63.10}$$

for every $x \in \mathbb{R}_+$.

In other words, $\sigma \in \Sigma$ is v-greedy if it optimally trades off current and future rewards when v is taken to be the value function.

In our setting, we have the following key result

Theorem 63.2.1

A feasible consumption policy is optimal if and only if it is v^*-greedy.

See, for example, Theorem 10.1.11 of EDTC.

Hence, once we have a good approximation to v^*, we can compute the (approximately) optimal policy by computing the corresponding greedy policy.

The advantage is that we are now solving a much lower dimensional optimization problem.

63.2.7 The Bellman Operator

How, then, should we compute the value function?

One way is to use the so-called **Bellman operator**.

(The term **operator** is usually reserved for functions that send functions into functions!)

The Bellman operator is denoted by T and defined by

$$Tv(x) := \max_{0 \leq c \leq x} \left\{ u(c) + \beta \int v(f(x-c)z)\phi(dz) \right\} \qquad (x \in \mathbb{R}_+) \tag{63.11}$$

In other words, T sends the function v into the new function Tv defined by (63.11).

By construction, the set of solutions to the Bellman equation (63.9) *exactly coincides with* the set of fixed points of T.

For example, if $Tv = v$, then, for any $x \geq 0$,

$$v(x) = Tv(x) = \max_{0 \leq c \leq x} \left\{ u(c) + \beta \int v(f(x-c)z)\phi(dz) \right\}$$

which says precisely that v is a solution to the Bellman equation.

It follows that v^* is a fixed point of T.

63.2.8 Review of Theoretical Results

One can also show that T is a contraction mapping on the set of continuous bounded functions on $\mathbb{R}_+$ under the supremum distance

$$\rho(g, h) = \sup_{x \geq 0} |g(x) - h(x)|$$

See EDTC, Lemma 10.1.18.

Hence, it has exactly one fixed point in this set, which we know is equal to the value function.

It follows that

- The value function v^* is bounded and continuous.
- Starting from any bounded and continuous v, the sequence $v, Tv, T^2v, \ldots$ generated by iteratively applying T converges uniformly to v^*.

This iterative method is called **value function iteration**.

We also know that a feasible policy is optimal if and only if it is v^*-greedy.

It's not too hard to show that a v^*-greedy policy exists.

Hence, at least one optimal policy exists.

Our problem now is how to compute it.

63.2.9 Unbounded Utility

The results stated above assume that u is bounded.

In practice economists often work with unbounded utility functions — and so will we.

In the unbounded setting, various optimality theories exist.

Nevertheless, their main conclusions are usually in line with those stated for the bounded case just above (as long as we drop the word "bounded").

Note

Consult the following references for more on the unbounded case:

- The lecture *The Income Fluctuation Problem V: Stochastic Returns on Assets*.
- Section 12.2 of EDTC.

63.3 Computation

Let's now look at computing the value function and the optimal policy.

Our implementation in this lecture will focus on clarity and flexibility.

(In subsequent lectures we will focus on efficiency and speed.)

We will use fitted value function iteration, which was already described in *Optimal Savings II: Numerical Cake Eating*.

63.3.1 Scalar Maximization

To maximize the right hand side of the Bellman equation (63.9), we are going to use the `minimize_scalar` routine from SciPy.

To keep the interface tidy, we will wrap `minimize_scalar` in an outer function as follows:

```
def maximize(g, upper_bound):
    """
    Maximize the function g over the interval [0, upper_bound].

    We use the fact that the maximizer of g on any interval is
    also the minimizer of -g.

    """

    objective = lambda x: -g(x)
```

(continues on next page)

(continued from previous page)

```
    bounds = (0, upper_bound)
    result = minimize_scalar(objective, bounds=bounds, method='bounded')
    maximizer, maximum = result.x, -result.fun
    return maximizer, maximum
```

63.3.2 Model

We will assume for now that ϕ is the distribution of $\xi := \exp(\mu + \nu\zeta)$ where

- ζ is standard normal,
- μ is a shock location parameter and
- ν is a shock scale parameter.

We will store the primitives of the model in a `NamedTuple`.

```
class Model(NamedTuple):
    u: Callable        # utility function
    f: Callable        # production function
    β: float           # discount factor
    μ: float           # shock location parameter
    ν: float           # shock scale parameter
    x_grid: np.ndarray # state grid
    shocks: np.ndarray # shock draws


def create_model(
        u: Callable,
        f: Callable,
        β: float = 0.96,
        μ: float = 0.0,
        ν: float = 0.1,
        grid_max: float = 4.0,
        grid_size: int = 120,
        shock_size: int = 250,
        seed: int = 1234
    ) -> Model:
    """
    Creates an instance of the optimal savings model.
    """
    # Set up grid
    x_grid = np.linspace(1e-4, grid_max, grid_size)

    # Store shocks (with a seed, so results are reproducible)
    np.random.seed(seed)
    shocks = np.exp(μ + ν * np.random.randn(shock_size))

    return Model(u, f, β, μ, ν, x_grid, shocks)
```

We set up the right-hand side of the Bellman equation

$$B(x, c, v) := u(c) + \beta \int v(f(x - c)z)\phi(dz)$$

```
def B(
        x: float,               # State
        c: float,               # Action
        v_array: np.ndarray,    # Array representing a guess of the value fn
        model: Model            # An instance of Model containing parameters
    ):

    u, f, β, μ, v, x_grid, shocks = model
    v = interp1d(x_grid, v_array)

    return u(c) + β * np.mean(v(f(x - c) * shocks))
```

In the second last line we are using linear interpolation.

In the last line, the expectation in (63.11) is computed via Monte Carlo, using the approximation

$$\int v(f(x-c)z)\phi(dz) \approx \frac{1}{n}\sum_{i=1}^{n} v(f(x-c)\xi_i)$$

where $\{\xi_i\}_{i=1}^n$ are IID draws from ϕ.

Monte Carlo is not always the most efficient way to compute integrals numerically but it does have some theoretical advantages in the present setting.

(For example, it preserves the contraction mapping property of the Bellman operator — see, e.g., [Pál and Stachurski, 2013].)

63.3.3 The Bellman Operator

The next function implements the Bellman operator.

```
def T(v: np.ndarray, model: Model) -> tuple[np.ndarray, np.ndarray]:
    """
    The Bellman operator.  Updates the guess of the value function.

      * model is an instance of Model
      * v is an array representing a guess of the value function

    """
    x_grid = model.x_grid
    v_new = np.empty_like(v)

    for i in range(len(x_grid)):
        x = x_grid[i]
        _, v_max = maximize(lambda c: B(x, c, v, model), x)
        v_new[i] = v_max

    return v_new
```

Here's the function:

```
def get_greedy(
        v: np.ndarray,          # current guess of the value function
        model: Model            # instance of optimal savings model
    ):
    " Compute the v-greedy policy on x_grid."
```

(continues on next page)

(continued from previous page)

```
    σ = np.empty_like(v)

    for i, x in enumerate(model.x_grid):
        # Maximize RHS of Bellman equation at state x
        σ[i], _ = maximize(lambda c: B(x, c, v, model), x)

    return σ
```

63.3.4 An Example

Let's suppose now that

$$f(x-c) = (x-c)^\alpha \quad \text{and} \quad u(c) = \ln c$$

For this particular problem, an exact analytical solution is available (see [Ljungqvist and Sargent, 2018], section 3.1.2), with

$$v^*(x) = \frac{\ln(1-\alpha\beta)}{1-\beta} + \frac{(\mu + \alpha \ln(\alpha\beta))}{1-\alpha}\left[\frac{1}{1-\beta} - \frac{1}{1-\alpha\beta}\right] + \frac{1}{1-\alpha\beta}\ln x \tag{63.12}$$

and optimal consumption policy

$$\sigma^*(x) = (1-\alpha\beta)x$$

It is valuable to have these closed-form solutions because it lets us check whether our code works for this particular case.

In Python, the functions above can be expressed as:

```
def v_star(x, α, β, μ):
    """
    True value function
    """
    c1 = np.log(1 - α * β) / (1 - β)
    c2 = (μ + α * np.log(α * β)) / (1 - α)
    c3 = 1 / (1 - β)
    c4 = 1 / (1 - α * β)
    return c1 + c2 * (c3 - c4) + c4 * np.log(x)

def σ_star(x, α, β):
    """
    True optimal policy
    """
    return (1 - α * β) * x
```

Next let's create an instance of the model with the above primitives and assign it to the variable `model`.

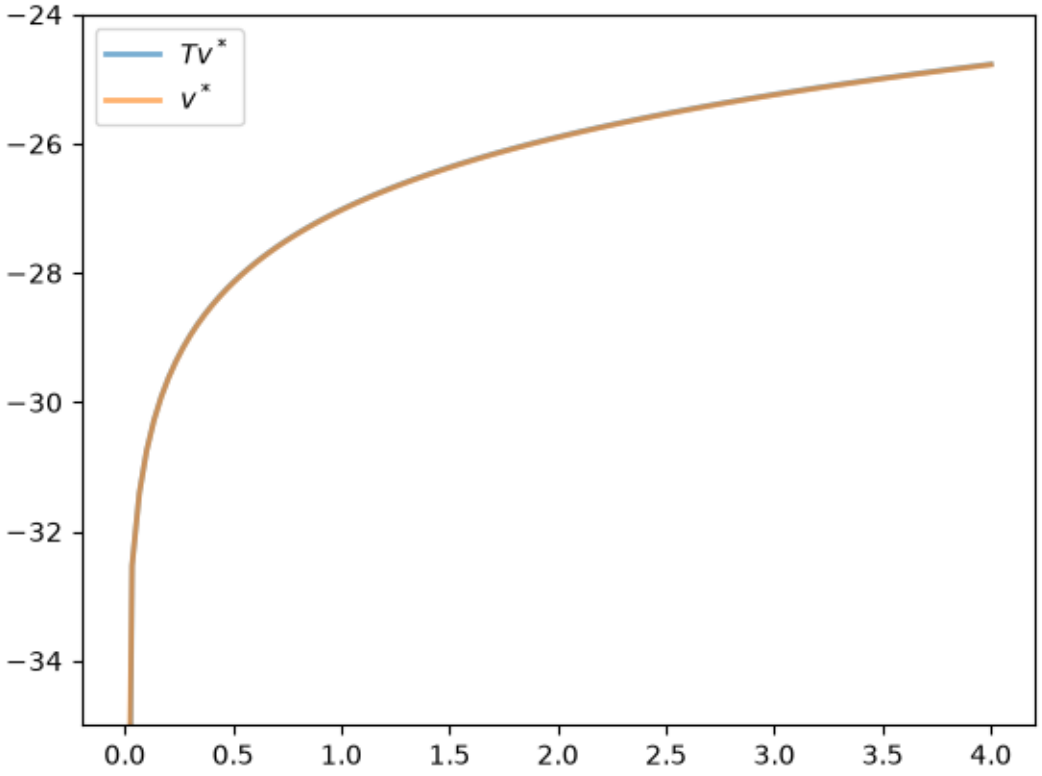
```
α = 0.4
def fcd(s):
    return s**α

model = create_model(u=np.log, f=fcd)
```

Now let's see what happens when we apply our Bellman operator to the exact solution v^* in this case.

In theory, since v^* is a fixed point, the resulting function should again be v^*.

In practice, we expect some small numerical error.

```
x_grid = model.x_grid

v_init = v_star(x_grid, α, model.β, model.μ)     # Start at the solution
v = T(v_init, model)              # Apply T once

fig, ax = plt.subplots()
ax.set_ylim(-35, -24)
ax.plot(x_grid, v, lw=2, alpha=0.6, label='$Tv^*$')
ax.plot(x_grid, v_init, lw=2, alpha=0.6, label='$v^*$')
ax.legend()
plt.show()
```



The two functions are essentially indistinguishable, so we are off to a good start.

Now let's have a look at iterating with the Bellman operator, starting from an arbitrary initial condition.

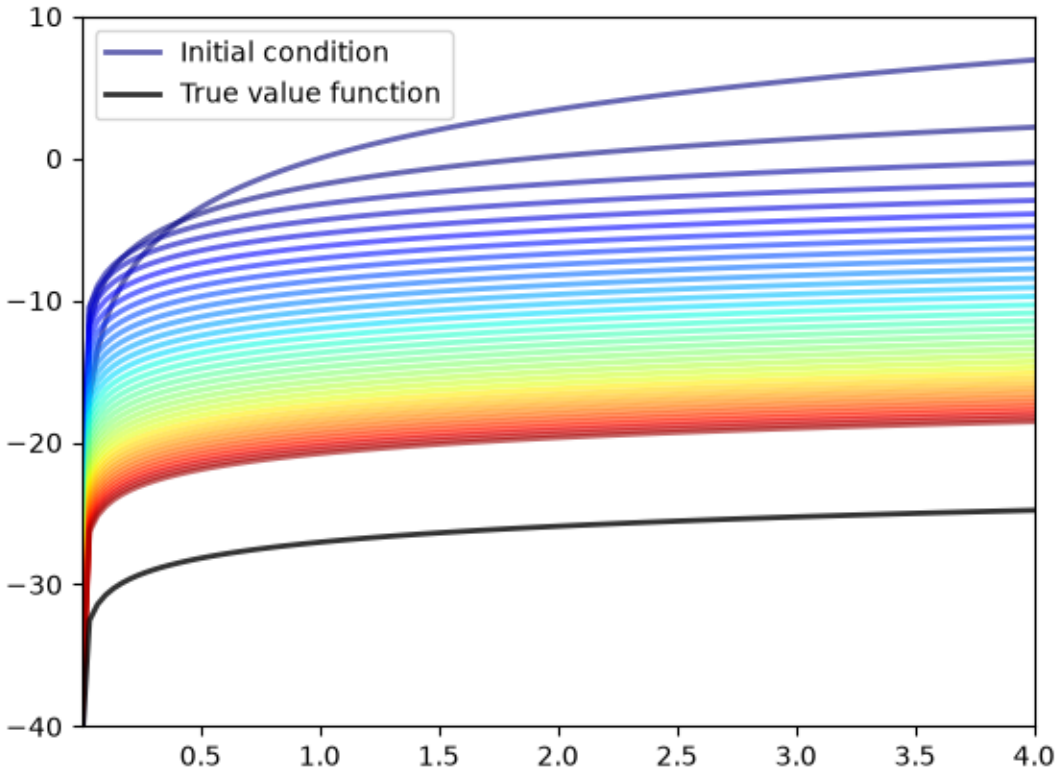
The initial condition we'll start with is, somewhat arbitrarily, $v(x) = 5 \ln(x)$.

```
v = 5 * np.log(x_grid)  # An initial condition
n = 35

fig, ax = plt.subplots()

ax.plot(x_grid, v, color=plt.cm.jet(0),
        lw=2, alpha=0.6, label='Initial condition')

for i in range(n):
    v = T(v, model)  # Apply the Bellman operator
    ax.plot(x_grid, v, color=plt.cm.jet(i / n), lw=2, alpha=0.6)
```

(continues on next page)

(continued from previous page)

```
ax.plot(x_grid, v_star(x_grid, α, model.β, model.μ), 'k-', lw=2,
        alpha=0.8, label='True value function')

ax.legend()
ax.set(ylim=(-40, 10), xlim=(np.min(x_grid), np.max(x_grid)))
plt.show()
```



The figure shows

1. the first 36 functions generated by the fitted value function iteration algorithm, with hotter colors given to higher iterates
2. the true value function v^* drawn in black

The sequence of iterates converges towards v^*.

We are clearly getting closer.

63.3.5 Iterating to Convergence

We can write a function that iterates until the difference is below a particular tolerance level.

```
def solve_model(
        model: Model,           # instance of optimal savings model
        tol: float = 1e-4,      # convergence tolerance
        max_iter: int = 1000,   # maximum iterations
        verbose: bool = True,   # print iteration info
        print_skip: int = 25    # iterations between prints
```

(continues on next page)

(continued from previous page)

```
    ):
    " Solve by value function iteration. "

    v = model.u(model.x_grid)  # Initial condition
    i = 0
    error = tol + 1

    while i < max_iter and error > tol:
        v_new = T(v, model)
        error = np.max(np.abs(v - v_new))
        i += 1
        if verbose and i % print_skip == 0:
            print(f"Error at iteration {i} is {error}.")
        v = v_new

    if error > tol:
        print("Failed to converge!")
    elif verbose:
        print(f"\nConverged in {i} iterations.")

    v_greedy = get_greedy(v_new, model)
    return v_greedy, v_new
```

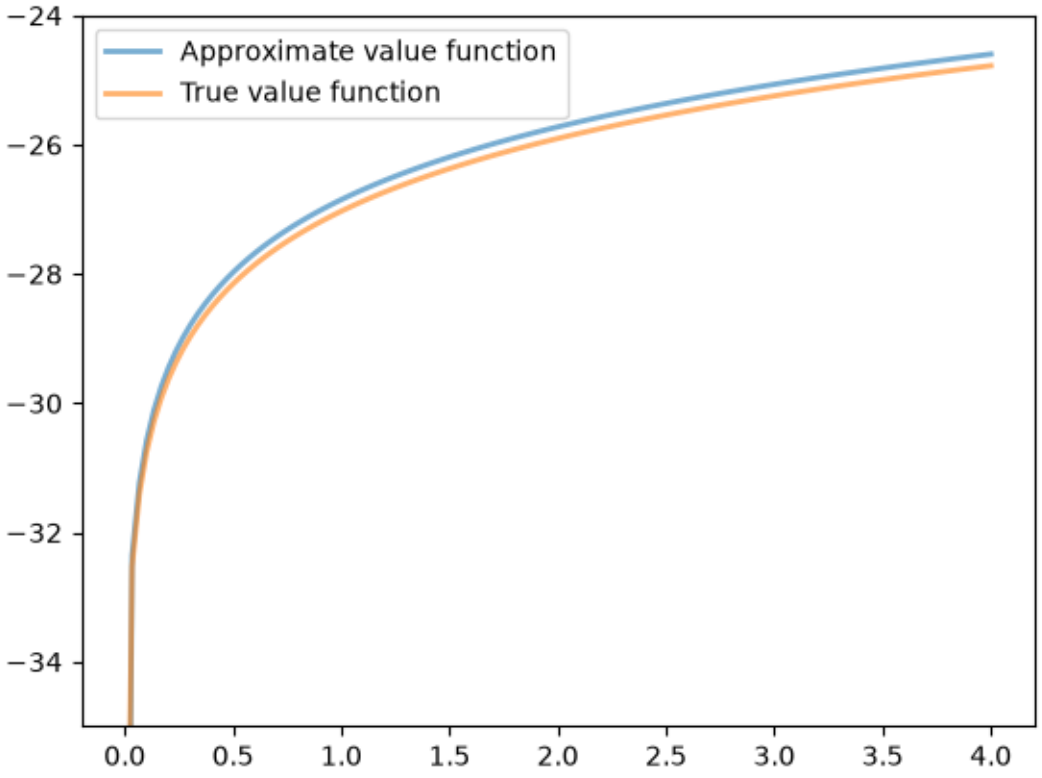
Let's use this function to compute an approximate solution at the defaults.

```
v_greedy, v_solution = solve_model(model)
```

```
Error at iteration 25 is 0.40975776844490497.
Error at iteration 50 is 0.1476753540823772.
Error at iteration 75 is 0.05322171277213883.
Error at iteration 100 is 0.019180930548646558.
Error at iteration 125 is 0.006912744396021964.
Error at iteration 150 is 0.0024913303848137502.
Error at iteration 175 is 0.0008978672913073638.
Error at iteration 200 is 0.00032358842396718046.
Error at iteration 225 is 0.00011662020561686859.

Converged in 229 iterations.
```

Now we check our result by plotting it against the true value:

```
fig, ax = plt.subplots()

ax.plot(x_grid, v_solution, lw=2, alpha=0.6,
        label='Approximate value function')

ax.plot(x_grid, v_star(x_grid, α, model.β, model.μ), lw=2,
        alpha=0.6, label='True value function')

ax.legend()
ax.set_ylim(-35, -24)
plt.show()
```



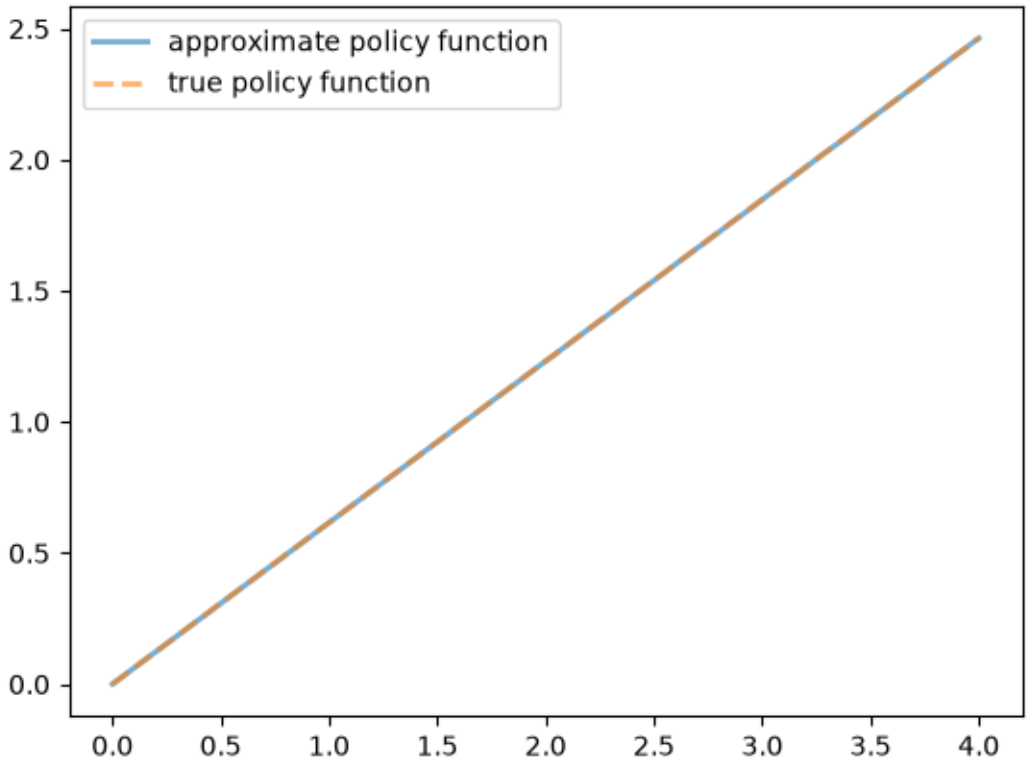
The figure shows that we are pretty much on the money.

63.3.6 The Policy Function

The policy `v_greedy` computed above corresponds to an approximate optimal policy.

The next figure compares it to the exact solution, which, as mentioned above, is $\sigma(x) = (1 - \alpha\beta)x$

```
fig, ax = plt.subplots()

ax.plot(x_grid, v_greedy, lw=2,
        alpha=0.6, label='approximate policy function')

ax.plot(x_grid, σ_star(x_grid, α, model.β), '--',
        lw=2, alpha=0.6, label='true policy function')

ax.legend()
plt.show()
```



The figure shows that we've done a good job in this instance of approximating the true policy.

63.4 Exercises

Exercise 63.4.1

A common choice for utility function in this kind of work is the CRRA specification

$$u(c) = \frac{c^{1-\gamma}}{1-\gamma}$$

Maintaining the other defaults, including the Cobb-Douglas production function, solve the optimal savings model with this utility specification.

Setting $\gamma = 1.5$, compute and plot an estimate of the optimal policy.

Solution

Here we set up the model.

```
γ = 1.5    # Preference parameter

def u_crra(c):
    return (c**(1 - γ) - 1) / (1 - γ)

model = create_model(u=u_crra, f=fcd)
```

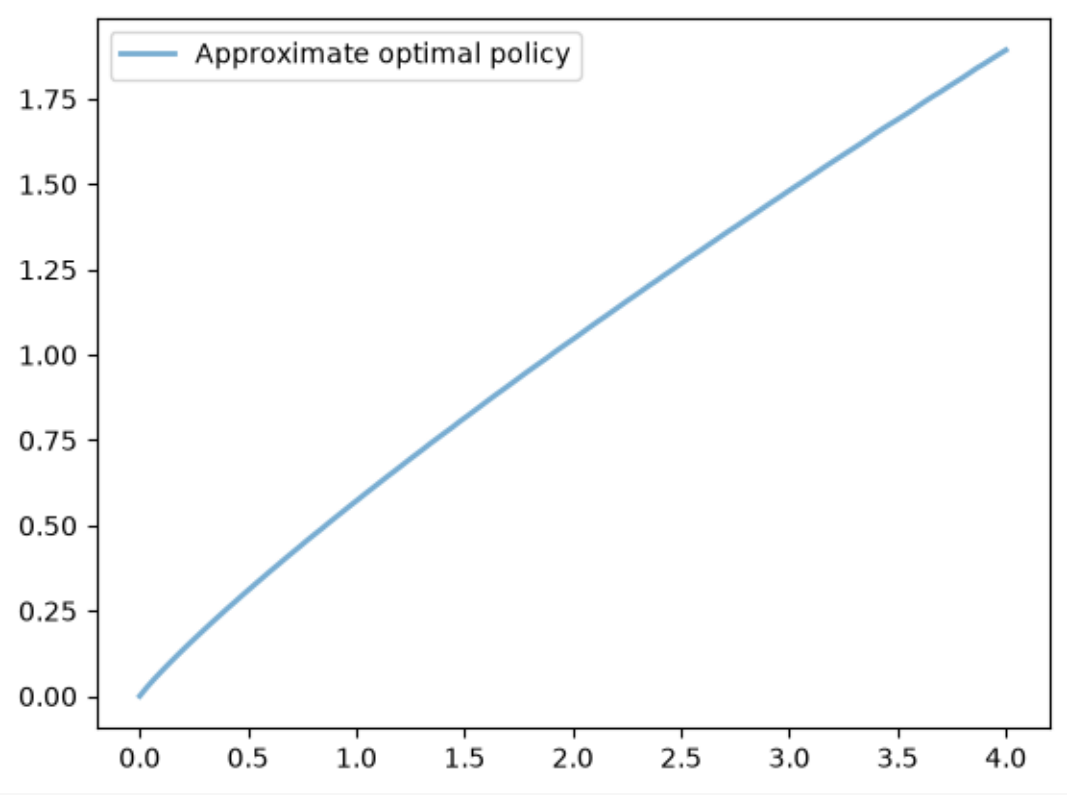
Now let's run it, with a timer.

```
%%time
v_greedy, v_solution = solve_model(model)
```

```
Error at iteration 25 is 0.5528151810315194.
Error at iteration 50 is 0.19923228425591333.
Error at iteration 75 is 0.07180266113797629.
Error at iteration 100 is 0.025877443335843964.
Error at iteration 125 is 0.009326145618956616.
Error at iteration 150 is 0.003361112262041388.
Error at iteration 175 is 0.0012113338242443206.
Error at iteration 200 is 0.0004365607333056687.
Error at iteration 225 is 0.00015733505506432266.

Converged in 237 iterations.
CPU times: user 27.1 s, sys: 4.01 ms, total: 27.1 s
Wall time: 27.1 s
```

Let's plot the policy function just to see what it looks like:

```
fig, ax = plt.subplots()
ax.plot(x_grid, v_greedy, lw=2,
        alpha=0.6, label='Approximate optimal policy')
ax.legend()
plt.show()
```



CHAPTER

SIXTYFOUR

OPTIMAL SAVINGS IV: TIME ITERATION

Contents

- *Optimal Savings IV: Time Iteration*
  - *Overview*
  - *The Euler Equation*
  - *Implementation*
  - *Exercises*

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install quantecon
```

64.1 Overview

In this lecture, we introduce the core idea of **time iteration**: iterating on a guess of the optimal policy using the Euler equation.

This approach differs from the value function iteration we used in *Optimal Savings III: Stochastic Returns*, where we iterated on the value function itself.

Time iteration exploits the structure of the Euler equation to find the optimal policy directly, rather than computing the value function as an intermediate step.

The key advantage is computational efficiency: by working directly with the policy function, we can often solve problems faster than with value function iteration.

However, time iteration is not the most efficient Euler equation-based method available.

In *Optimal Savings V: The Endogenous Grid Method*, we'll introduce the **endogenous grid method** (EGM), which provides an even more efficient way to solve the problem.

For now, our goal is to understand the basic mechanics of time iteration and how it leverages the Euler equation.

Let's start with some imports:

```
import matplotlib.pyplot as plt
import numpy as np
```

(continues on next page)

(continued from previous page)

```
from scipy.optimize import brentq
from typing import NamedTuple, Callable
```

64.2 The Euler Equation

Our first step is to derive the Euler equation, which is a generalization of the Euler equation we obtained in *Optimal Savings I: Cake Eating*.

We take the model set out in *Optimal Savings III: Stochastic Returns* and add the following assumptions:

1. u and f are continuously differentiable and strictly concave
2. $f(0) = 0$
3. $\lim_{c \to 0} u'(c) = \infty$ and $\lim_{c \to \infty} u'(c) = 0$
4. $\lim_{k \to 0} f'(k) = \infty$ and $\lim_{k \to \infty} f'(k) = 0$

The last two conditions are usually called **Inada conditions**.

Recall the Bellman equation

$$v(x) = \max_{0 \leq c \leq x} \left\{ u(c) + \beta \int v(f(x-c)z)\phi(dz) \right\} \quad \text{for all} \quad x \in \mathbb{R}_+ \tag{64.1}$$

Let v^* be the value function and let σ^* be the optimal consumption policy.

We know that σ^* is a v^*-greedy policy.

The conditions above imply that

- σ^* is the unique optimal policy for the optimal savings problem
- the optimal policy is continuous, strictly increasing and also **interior**, in the sense that $0 < \sigma^*(x) < x$ for all strictly positive x, and
- the value function is strictly concave and continuously differentiable, with

$$(v^*)'(x) = u'(\sigma^*(x)) := (u' \circ \sigma^*)(x) \tag{64.2}$$

The last result is called the **envelope condition** due to its relationship with the envelope theorem.

To see why (64.2) holds, write the Bellman equation in the equivalent form

$$v^*(x) = \max_{0 \leq k \leq x} \left\{ u(x-k) + \beta \int v^*(f(k)z)\phi(dz) \right\},$$

Differentiating with respect to x, and then evaluating at the optimum yields (64.2).

(Section 12.1 of EDTC contains full proofs of these results, and closely related discussions can be found in many other texts.)

Differentiability of the value function and interiority of the optimal policy imply that optimal consumption satisfies the first-order condition associated with (64.1), which is

$$u'(\sigma^*(x)) = \beta \int (v^*)'(f(x - \sigma^*(x))z) f'(x - \sigma^*(x)) z \phi(dz) \tag{64.3}$$

Combining (64.2) and the first-order condition (64.3) gives the **Euler equation**

$$(u' \circ \sigma^*)(x) = \beta \int (u' \circ \sigma^*)(f(x - \sigma^*(x))z) f'(x - \sigma^*(x)) z \phi(dz) \tag{64.4}$$

We can think of the Euler equation as a functional equation

$$(u' \circ \sigma)(x) = \beta \int (u' \circ \sigma)(f(x - \sigma(x))z) f'(x - \sigma(x)) z \phi(dz) \tag{64.5}$$

over interior consumption policies σ, one solution of which is the optimal policy σ^*.

Our aim is to solve the functional equation (64.5) and hence obtain σ^*.

64.2.1 The Coleman-Reffett Operator

Recall the Bellman operator

$$Tv(x) := \max_{0 \leq c \leq x} \left\{ u(c) + \beta \int v(f(x - c)z) \phi(dz) \right\} \tag{64.6}$$

Just as we introduced the Bellman operator to solve the Bellman equation, we will now introduce an operator over policies to help us solve the Euler equation.

This operator K will act on the set of all $\sigma \in \Sigma$ that are continuous, strictly increasing and interior.

Henceforth we denote this set of policies by $\mathcal{P}$

1. The operator K takes as its argument a $\sigma \in \mathcal{P}$ and
2. returns a new function $K\sigma$, where $K\sigma(x)$ is the $c \in (0, x)$ that solves

$$u'(c) = \beta \int (u' \circ \sigma)(f(x - c)z) f'(x - c) z \phi(dz) \tag{64.7}$$

We call this operator the **Coleman-Reffett operator** to acknowledge the work of [Coleman, 1990] and [Reffett, 1996].

In essence, $K\sigma$ is the consumption policy that the Euler equation tells you to choose today when your future consumption policy is σ.

The important thing to note about K is that, by construction, its fixed points coincide with solutions to the functional equation (64.5).

In particular, the optimal policy σ^* is a fixed point.

Indeed, for fixed x, the value $K\sigma^*(x)$ is the c that solves

$$u'(c) = \beta \int (u' \circ \sigma^*)(f(x - c)z) f'(x - c) z \phi(dz)$$

In view of the Euler equation, this is exactly $\sigma^*(x)$.

64.2.2 Is the Coleman-Reffett Operator Well Defined?

In particular, is there always a unique $c \in (0, x)$ that solves (64.7)?

The answer is yes, under our assumptions.

For any $\sigma \in \mathcal{P}$, the right side of (64.7)

- is continuous and strictly increasing in c on $(0, x)$
- diverges to $+\infty$ as $c \uparrow x$

The left side of (64.7)

- is continuous and strictly decreasing in c on $(0, x)$

- diverges to $+\infty$ as $c \downarrow 0$

Sketching these curves and using the information above will convince you that they cross exactly once as c ranges over $(0, x)$.

With a bit more analysis, one can show in addition that $K\sigma \in \mathcal{P}$ whenever $\sigma \in \mathcal{P}$.

64.2.3 Comparison with VFI (Theory)

It is possible to prove that there is a tight relationship between iterates of K and iterates of the Bellman operator.

Mathematically, T and K are **topologically conjugate** under a translation that involves differentiation in one direction and integration in the other.

This conjugacy implies that if iterates of one operator converge then so do iterates of the other, and vice versa.

Moreover, there is a sense in which they converge *at the same rate*, at least in theory.

However, it turns out that the operator K is more stable *numerically* and hence more efficient in the applications we consider.

This is because

- K exploits additional structure because it uses first-order conditions, and
- policies near the optimal policy have less curvature and hence are easier to approximate than value functions near the optimal value function.

Examples are given below.

64.3 Implementation

Let's turn to implementation.

Note

In this lecture we mainly focus on the algorithm, favoring clarity over efficiency in the code.

In later lectures we will optimize both the algorithm and the code.

As in *Optimal Savings III: Stochastic Returns*, we assume that

- $u(c) = \ln c$
- $f(x - c) = (x - c)^\alpha$
- ϕ is the distribution of $\xi := \exp(\mu + \nu\zeta)$ when ζ is standard normal

This allows us to compare our results to the analytical solutions we obtained in that lecture:

```
def v_star(x, α, β, μ):
    """
    True value function
    """
    c1 = np.log(1 - α * β) / (1 - β)
    c2 = (μ + α * np.log(α * β)) / (1 - α)
    c3 = 1 / (1 - β)
    c4 = 1 / (1 - α * β)
```

(continues on next page)

(continued from previous page)

```
    return c1 + c2 * (c3 - c4) + c4 * np.log(x)

def σ_star(x, α, β):
    """
    True optimal policy
    """
    return (1 - α * β) * x
```

As discussed above, our plan is to solve the model using time iteration, which means iterating with the operator K.

For this we need access to the functions u' and f, f'.

We use the same `Model` structure from *Optimal Savings III: Stochastic Returns*.

```
class Model(NamedTuple):
    u: Callable        # utility function
    f: Callable        # production function
    β: float           # discount factor
    μ: float           # shock location parameter
    ν: float           # shock scale parameter
    grid: np.ndarray   # state grid
    shocks: np.ndarray # shock draws
    α: float = 0.4     # production function parameter
    u_prime: Callable = None        # derivative of utility
    f_prime: Callable = None        # derivative of production


def create_model(
        u: Callable,
        f: Callable,
        β: float = 0.96,
        μ: float = 0.0,
        ν: float = 0.1,
        grid_max: float = 4.0,
        grid_size: int = 120,
        shock_size: int = 250,
        seed: int = 1234,
        α: float = 0.4,
        u_prime: Callable = None,
        f_prime: Callable = None
    ) -> Model:
    """
    Creates an instance of the optimal savings model.
    """
    # Set up grid
    grid = np.linspace(1e-4, grid_max, grid_size)

    # Store shocks (with a seed, so results are reproducible)
    np.random.seed(seed)
    shocks = np.exp(μ + ν * np.random.randn(shock_size))

    return Model(u, f, β, μ, ν, grid, shocks, α, u_prime, f_prime)
```

Now we implement a method called `euler_diff`, which returns

$$u'(c) - \beta \int (u' \circ \sigma)(f(x-c)z) f'(x-c) z \phi(dz) \tag{64.8}$$

```
def euler_diff(c: float, σ: np.ndarray, x: float, model: Model) -> float:
    """
    Set up a function such that the root with respect to c,
    given x and σ, is equal to Kσ(x).

    """

    # Unpack
    u, f, β, μ, ν, grid, shocks, α, u_prime, f_prime = model

    # Turn σ into a function via interpolation
    σ_func = lambda x: np.interp(x, grid, σ)

    # Now set up the function we need to find the root of.
    vals = u_prime(σ_func(f(x - c, α) * shocks)) * f_prime(x - c, α) * shocks
    return u_prime(c) - β * np.mean(vals)
```

The function `euler_diff` evaluates integrals by Monte Carlo and approximates functions using linear interpolation.

We will use a root-finding algorithm to solve (64.8) for c given state x and σ, the current guess of the policy.

Here's the operator K, that implements the root-finding step.

```
def K(σ: np.ndarray, model: Model) -> np.ndarray:
    """
    The Coleman-Reffett operator

    """

    # Unpack
    u, f, β, μ, ν, grid, shocks, α, u_prime, f_prime = model

    σ_new = np.empty_like(σ)
    for i, x in enumerate(grid):
        # Solve for optimal c at x
        c_star = brentq(euler_diff, 1e-10, x-1e-10, args=(σ, x, model))
        σ_new[i] = c_star

    return σ_new
```

64.3.1 Testing

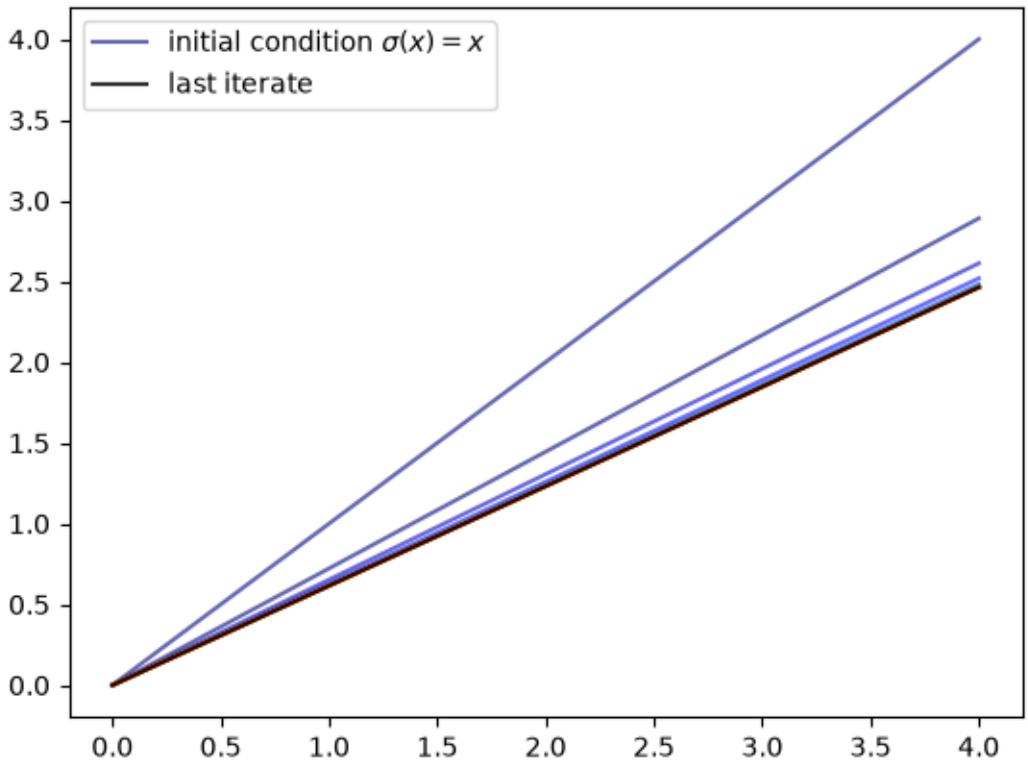
Let's generate an instance and plot some iterates of K, starting from $\sigma(x) = x$.

```
# Define utility and production functions with derivatives
α = 0.4
u = lambda c: np.log(c)
u_prime = lambda c: 1 / c
f = lambda k, α: k**α
f_prime = lambda k, α: α * k**(α - 1)

model = create_model(u=u, f=f, α=α, u_prime=u_prime, f_prime=f_prime)
grid = model.grid

n = 15
σ = grid.copy()  # Set initial condition
```

(continues on next page)

(continued from previous page)

```
fig, ax = plt.subplots()
lb = r'initial condition $\sigma(x) = x$'
ax.plot(grid, σ, color=plt.cm.jet(0), alpha=0.6, label=lb)

for i in range(n):
    σ = K(σ, model)
    ax.plot(grid, σ, color=plt.cm.jet(i / n), alpha=0.6)

# Update one more time and plot the last iterate in black
σ = K(σ, model)
ax.plot(grid, σ, color='k', alpha=0.8, label='last iterate')

ax.legend()

plt.show()
```



We see that the iteration process converges quickly to a limit that resembles the solution we obtained in *Optimal Savings III: Stochastic Returns*.

Here is a function called `solve_model_time_iter` that takes an instance of `Model` and returns an approximation to the optimal policy, using time iteration.

```
def solve_model_time_iter(
        model: Model,
        σ_init: np.ndarray,
        tol: float = 1e-5,
        max_iter: int = 1000,
        verbose: bool = True
```

(continues on next page)

(continued from previous page)

```
    ) -> np.ndarray:
    """
    Solve the model using time iteration.

    """
    σ = σ_init
    error = tol + 1
    i = 0

    while error > tol and i < max_iter:
        σ_new = K(σ, model)
        error = np.max(np.abs(σ_new - σ))
        σ = σ_new
        i += 1
        if verbose:
            print(f"Iteration {i}, error = {error}")

    if i == max_iter:
        print("Warning: maximum iterations reached")

    return σ
```

Let's call it:

```
# Unpack
grid = model.grid

σ_init = np.copy(grid)
σ = solve_model_time_iter(model, σ_init)
```

```
Iteration 1, error = 1.1098265895953756
Iteration 2, error = 0.27827989207957415
Iteration 3, error = 0.09312729948559406
Iteration 4, error = 0.034020038271351805
Iteration 5, error = 0.012820752818722525
Iteration 6, error = 0.004888081560539437
Iteration 7, error = 0.0018718902256105174
Iteration 8, error = 0.0007180512309568066
Iteration 9, error = 0.0002756205293255043
Iteration 10, error = 0.00010582190181418483
Iteration 11, error = 4.063319516811603e-05
Iteration 12, error = 1.560279084289462e-05
Iteration 13, error = 5.991419175455093e-06
```

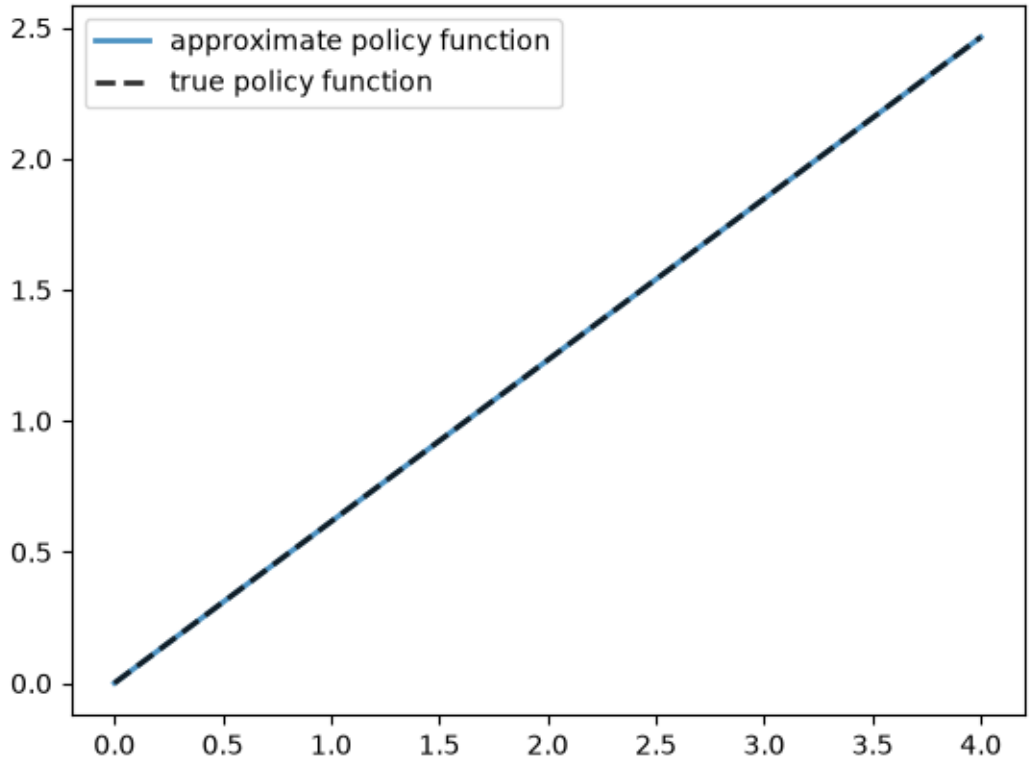
Here is a plot of the resulting policy, compared with the true policy:

```
# Unpack
grid, α, β = model.grid, model.α, model.β

fig, ax = plt.subplots()

ax.plot(grid, σ, lw=2,
        alpha=0.8, label='approximate policy function')

ax.plot(grid, σ_star(grid, α, β), 'k--',
        lw=2, alpha=0.8, label='true policy function')
```

(continues on next page)

(continued from previous page)

```
ax.legend()
plt.show()
```



Again, the fit is excellent.

The maximal absolute deviation between the two policies is

```
# Unpack
grid, α, β = model.grid, model.α, model.β

np.max(np.abs(σ - σ_star(grid, α, β)))
```

```
np.float64(3.7348959489591493e-06)
```

Time iteration runs faster than value function iteration, as discussed in *Optimal Savings III: Stochastic Returns*.

This is because time iteration exploits differentiability and the first-order conditions, while value function iteration does not use this available structure.

At the same time, there is a variation of time iteration that runs even faster.

This is the endogenous grid method, which we will introduce in *Optimal Savings V: The Endogenous Grid Method*.

64.4 Exercises

Exercise 64.4.1

Solve the optimal savings problem with CRRA utility

$$u(c) = \frac{c^{1-\gamma}}{1-\gamma}$$

Set `γ = 1.5`.

Compute and plot the optimal policy.

Solution

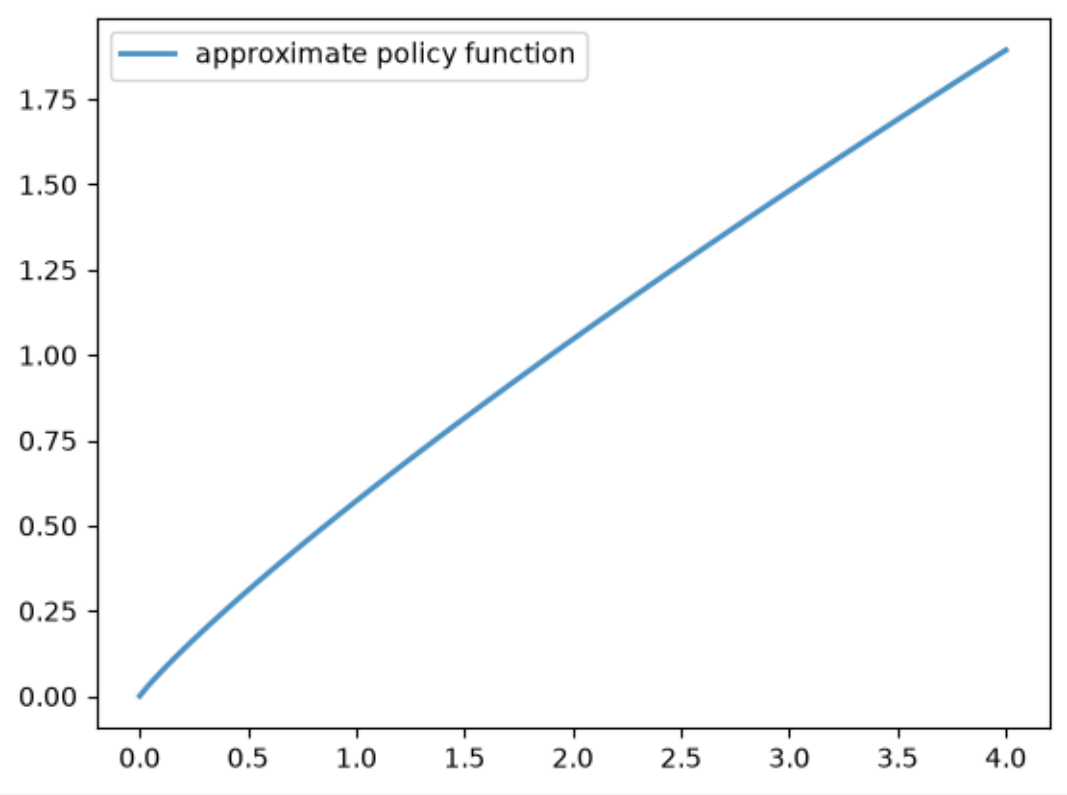
We define the CRRA utility function and its derivative.

```
γ = 1.5

def u_crra(c):
    return c**(1 - γ) / (1 - γ)

def u_prime_crra(c):
    return c**(-γ)

# Use the same production function as before
model_crra = create_model(u=u_crra, f=f, α=α,
                          u_prime=u_prime_crra, f_prime=f_prime)
```

Now we solve and plot the policy:

```
%%time
# Unpack
grid = model_crra.grid

σ_init = np.copy(grid)
σ = solve_model_time_iter(model_crra, σ_init)


fig, ax = plt.subplots()

ax.plot(grid, σ, lw=2,
        alpha=0.8, label='approximate policy function')

ax.legend()
plt.show()
```

```
Iteration 1, error = 1.449952719114732
Iteration 2, error = 0.3967698022828947
Iteration 3, error = 0.14845269076775747
Iteration 4, error = 0.06192954031818365
Iteration 5, error = 0.027017665601367424
Iteration 6, error = 0.012019070058330028
Iteration 7, error = 0.005393694573905705
Iteration 8, error = 0.0024299846499917788
Iteration 9, error = 0.0010967197524933692
Iteration 10, error = 0.0004953902833375601
Iteration 11, error = 0.0002238472234141753
Iteration 12, error = 0.0001011641350074921
Iteration 13, error = 4.572272482672446e-05
Iteration 14, error = 2.066580711579391e-05
Iteration 15, error = 9.340704450133686e-06
CPU times: user 671 ms, sys: 997 µs, total: 672 ms
```


approximate policy function
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0

CHAPTER

SIXTYFIVE

OPTIMAL SAVINGS V: THE ENDOGENOUS GRID METHOD

Contents

- *Optimal Savings V: The Endogenous Grid Method*
  - *Overview*
  - *Key Idea*
  - *Implementation*

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install quantecon
```

65.1 Overview

Previously, we solved the optimal savings problem using

1. *value function iteration*
2. *Euler equation based time iteration*

We found time iteration to be significantly more accurate and efficient.

In this lecture, we'll look at a clever twist on time iteration called the **endogenous grid method** (EGM).

EGM is a numerical method for implementing policy iteration invented by Chris Carroll.

The original reference is [Carroll, 2006].

For now we will focus on a clean and simple implementation of EGM that stays close to the underlying mathematics.

Then, in *Optimal Savings VI: EGM with JAX*, we will construct a fully vectorized and parallelized version of EGM based on JAX.

Let's start with some standard imports:

```
import matplotlib.pyplot as plt
import numpy as np
import quantecon as qe
```

65.2 Key Idea

First we remind ourselves of the theory and then we turn to numerical methods.

65.2.1 Theory

We work with the model set out in *Optimal Savings IV: Time Iteration*, following the same terminology and notation.

As we saw, the Coleman-Reffett operator is a nonlinear operator K engineered so that the optimal policy σ^* is a fixed point of K.

It takes as its argument a continuous strictly increasing consumption policy $\sigma \in \Sigma$.

It returns a new function $K\sigma$, where $(K\sigma)(x)$ is the $c \in (0, \infty)$ that solves

$$u'(c) = \beta \int (u' \circ \sigma)(f(x-c)z) f'(x-c) z \phi(dz) \tag{65.1}$$

65.2.2 Exogenous Grid

As discussed in *Optimal Savings IV: Time Iteration*, to implement the method on a computer, we represent a policy function by a set of values on a finite grid.

The function itself is reconstructed from this representation when necessary, using interpolation or some other method.

Our previous strategy in *Optimal Savings IV: Time Iteration* for obtaining a finite representation of an updated consumption policy was to

- fix a grid of income points $\{x_i\}$
- calculate the consumption value c_i corresponding to each x_i using (65.1) and a root-finding routine

Each c_i is then interpreted as the value of the function $K\sigma$ at x_i.

Thus, with the pairs $\{(x_i, c_i)\}$ in hand, we can reconstruct $K\sigma$ via approximation.

Iteration then continues…

65.2.3 Endogenous Grid

The method discussed above requires a root-finding routine to find the c_i corresponding to a given income value x_i.

Root-finding is costly because it typically involves a significant number of function evaluations.

As pointed out by Carroll [Carroll, 2006], we can avoid this step if x_i is chosen endogenously.

The only assumption required is that u' is invertible on $(0, \infty)$.

Let $(u')^{-1}$ be the inverse function of u'.

The idea is this:

- First, we fix an *exogenous* grid $\{s_i\}$ for savings ($s = x - c$).
- Then we obtain c_i via

$$c_i = (u')^{-1} \left\{ \beta \int (u' \circ \sigma)(f(s_i)z) \, f'(s_i) \, z \, \phi(dz) \right\} \tag{65.2}$$

- Finally, for each c_i we set $x_i = c_i + s_i$.

Importantly, each (x_i, c_i) pair constructed in this manner satisfies (65.1).

With the points $\{x_i, c_i\}$ in hand, we can reconstruct $K\sigma$ via approximation as before.

The name EGM comes from the fact that the grid $\{x_i\}$ is determined **endogenously**.

65.3 Implementation

As in *Optimal Savings IV: Time Iteration*, we will start with a simple setting where

- $u(c) = \ln c$,
- the function f has a Cobb-Douglas specification, and
- the shocks are lognormal.

This will allow us to make comparisons with the analytical solutions.

```
def v_star(x, α, β, μ):
    """
    True value function
    """
    c1 = np.log(1 - α * β) / (1 - β)
    c2 = (μ + α * np.log(α * β)) / (1 - α)
    c3 = 1 / (1 - β)
    c4 = 1 / (1 - α * β)
    return c1 + c2 * (c3 - c4) + c4 * np.log(x)

def σ_star(x, α, β):
    """
    True optimal policy
    """
    return (1 - α * β) * x
```

We reuse the `Model` structure from *Optimal Savings IV: Time Iteration*.

```
from typing import NamedTuple, Callable

class Model(NamedTuple):
    u: Callable           # utility function
    f: Callable           # production function
    β: float              # discount factor
    μ: float              # shock location parameter
    ν: float              # shock scale parameter
    s_grid: np.ndarray    # exogenous savings grid
    shocks: np.ndarray    # shock draws
    α: float              # production function parameter
    u_prime: Callable     # derivative of utility
    f_prime: Callable     # derivative of production
    u_prime_inv: Callable # inverse of u_prime


def create_model(
        u: Callable,
        f: Callable,
        β: float = 0.96,
        μ: float = 0.0,
        ν: float = 0.1,
```

(continues on next page)

(continued from previous page)

```
        grid_max: float = 4.0,
        grid_size: int = 120,
        shock_size: int = 250,
        seed: int = 1234,
        α: float = 0.4,
        u_prime: Callable = None,
        f_prime: Callable = None,
        u_prime_inv: Callable = None
    ) -> Model:
    """
    Creates an instance of the optimal savings model.
    """
    # Set up exogenous savings grid
    s_grid = np.linspace(1e-4, grid_max, grid_size)

    # Store shocks (with a seed, so results are reproducible)
    np.random.seed(seed)
    shocks = np.exp(μ + ν * np.random.randn(shock_size))

    return Model(
        u, f, β, μ, ν, s_grid, shocks, α, u_prime, f_prime, u_prime_inv
    )
```

65.3.1 The Operator

Here's an implementation of K using EGM as described above.

```
def K(
        c_in: np.ndarray,    # Consumption values on the endogenous grid
        x_in: np.ndarray,    # Current endogenous grid
        model: Model         # Model specification
    ):
    """
    An implementation of the Coleman-Reffett operator using EGM.

    """

    # Simplify names
    u, f, β, μ, ν, s_grid, shocks, α, u_prime, f_prime, u_prime_inv = model

    # Linear interpolation of policy on the endogenous grid
    σ = lambda x: np.interp(x, x_in, c_in)

    # Allocate memory for new consumption array
    c_out = np.empty_like(s_grid)

    for i, s in enumerate(s_grid):
        # Approximate marginal utility ∫ u'(σ(f(s, α)z)) f'(s, α) z ϕ(z)dz
        vals = u_prime(σ(f(s, α) * shocks)) * f_prime(s, α) * shocks
        mu = np.mean(vals)
        # Compute consumption
        c_out[i] = u_prime_inv(β * mu)

    # Determine corresponding endogenous grid
    x_out = s_grid + c_out        # x_i = s_i + c_i
```

(continues on next page)

(continued from previous page)

```
    return c_out, x_out
```

Note the lack of any root-finding algorithm.

Note

The routine is still not particularly fast because we are using pure Python loops.

But in the next lecture (*Optimal Savings VI: EGM with JAX*) we will use a fully vectorized and efficient solution.

65.3.2 Testing

First we create an instance.

```
# Define utility and production functions with derivatives
u = lambda c: np.log(c)
u_prime = lambda c: 1 / c
u_prime_inv = lambda x: 1 / x
f = lambda k, α: k**α
f_prime = lambda k, α: α * k**(α - 1)

model = create_model(u=u, f=f, u_prime=u_prime,
                     f_prime=f_prime, u_prime_inv=u_prime_inv)
s_grid = model.s_grid
```

Here's our solver routine:

```
def solve_model_time_iter(
        model: Model,              # Model details
        c_init: np.ndarray,        # initial guess of consumption on EG
        x_init: np.ndarray,        # initial guess of endogenous grid
        tol: float = 1e-5,         # Error tolerance
        max_iter: int = 1000,      # Max number of iterations of K
        verbose: bool = True       # If true print output
    ):
    """
    Solve the model using time iteration with EGM.
    """
    c, x = c_init, x_init
    error = tol + 1
    i = 0

    while error > tol and i < max_iter:
        c_new, x_new = K(c, x, model)
        error = np.max(np.abs(c_new - c))
        c, x = c_new, x_new
        i += 1
        if verbose:
            print(f"Iteration {i}, error = {error}")

    if i == max_iter:
        print("Warning: maximum iterations reached")
```

(continues on next page)

(continued from previous page)

```
    return c, x
```

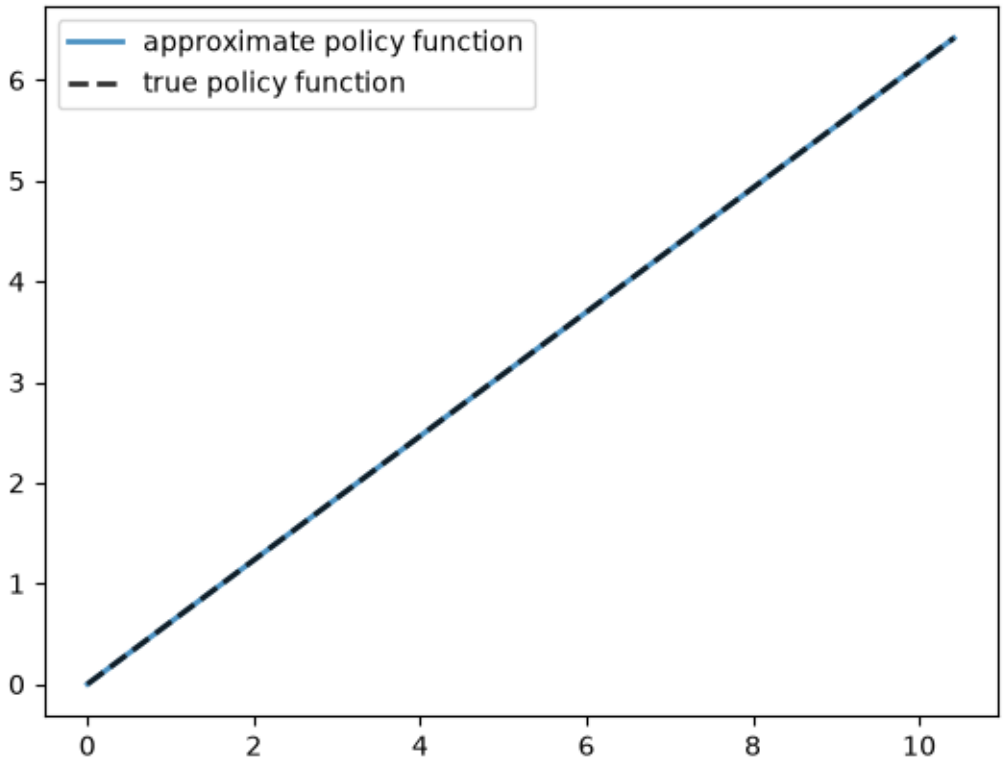
Let's call it:

```
c_init = np.copy(s_grid)
x_init = s_grid + c_init
c, x = solve_model_time_iter(model, c_init, x_init)
```

```
Iteration 1, error = 1.208333333333333
Iteration 2, error = 0.6834464555052788
Iteration 3, error = 0.3126351338414741
Iteration 4, error = 0.12905177785629984
Iteration 5, error = 0.05099394329538409
Iteration 6, error = 0.01980055511172729
Iteration 7, error = 0.007636088144566955
Iteration 8, error = 0.002937098891578671
Iteration 9, error = 0.0011285611196765188
Iteration 10, error = 0.00043347299641638415
Iteration 11, error = 0.00016646919532892213
Iteration 12, error = 6.392646634978405e-05
Iteration 13, error = 2.4548101555943447e-05
Iteration 14, error = 9.426520908739633e-06
```

Here is a plot of the resulting policy, compared with the true policy:

```
fig, ax = plt.subplots()

ax.plot(x, c, lw=2,
        alpha=0.8, label='approximate policy function')

ax.plot(x, σ_star(x, model.α, model.β), 'k--',
        lw=2, alpha=0.8, label='true policy function')

ax.legend()
plt.show()
```



The maximal absolute deviation between the two policies is

```
np.max(np.abs(c - σ_star(x, model.α, model.β)))
```

```
np.float64(2.2564941266622895e-06)
```

Here's the execution time:

```
with qe.Timer():
    c, x = solve_model_time_iter(model, c_init, x_init, verbose=False)
```

```
0.0313 seconds elapsed
```

EGM is faster than time iteration because it avoids numerical root-finding.

Instead, we invert the marginal utility function directly, which is much more efficient.

In *Optimal Savings VI: EGM with JAX*, we will use a fully vectorized and efficient version of EGM that is also parallelized using JAX.

This provides an extremely fast way to solve the optimal consumption problem we have been studying for the last few lectures.

CHAPTER

SIXTYSIX

OPTIMAL SAVINGS VI: EGM WITH JAX

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

Contents

- *Optimal Savings VI: EGM with JAX*
  - *Overview*
  - *Implementation*
  - *Exercises*

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install quantecon jax
```

66.1 Overview

In this lecture, we'll implement the endogenous grid method (EGM) using JAX.

This lecture builds on *Optimal Savings V: The Endogenous Grid Method*, which introduced EGM using NumPy.

By converting to JAX, we can leverage fast linear algebra, hardware accelerators, and JIT compilation for improved performance.

We'll also use JAX's `vmap` function to fully vectorize the Coleman-Reffett operator.

Let's start with some standard imports:

```
import matplotlib.pyplot as plt
import jax
import jax.numpy as jnp
import quantecon as qe
from typing import NamedTuple
```

66.2 Implementation

For details on the savings problem and the endogenous grid method (EGM), please see *Optimal Savings V: The Endogenous Grid Method*.

Here we focus on the JAX implementation of EGM.

We use the same setting as in *Optimal Savings V: The Endogenous Grid Method*:

- $u(c) = \ln c$,
- production is Cobb-Douglas, and
- the shocks are lognormal.

Here are the analytical solutions for comparison.

```
def v_star(x, α, β, μ):
    """
    True value function
    """
    c1 = jnp.log(1 - α * β) / (1 - β)
    c2 = (μ + α * jnp.log(α * β)) / (1 - α)
    c3 = 1 / (1 - β)
    c4 = 1 / (1 - α * β)
    return c1 + c2 * (c3 - c4) + c4 * jnp.log(x)

def σ_star(x, α, β):
    """
    True optimal policy
    """
    return (1 - α * β) * x
```

The `Model` class stores only the data (grids, shocks, and parameters).

Utility and production functions will be defined globally to work with JAX's JIT compiler.

```
class Model(NamedTuple):
    β: float              # discount factor
    μ: float              # shock location parameter
    s: float              # shock scale parameter
    s_grid: jnp.ndarray   # exogenous savings grid
    shocks: jnp.ndarray   # shock draws
    α: float              # production function parameter


def create_model(
        β: float = 0.96,
        μ: float = 0.0,
        s: float = 0.1,
        grid_max: float = 4.0,
```

(continues on next page)

(continued from previous page)

```
        grid_size: int = 120,
        shock_size: int = 250,
        seed: int = 1234,
        α: float = 0.4
    ) -> Model:
    """
    Creates an instance of the optimal savings model.
    """
    # Set up exogenous savings grid
    s_grid = jnp.linspace(1e-4, grid_max, grid_size)

    # Store shocks (with a seed, so results are reproducible)
    key = jax.random.PRNGKey(seed)
    shocks = jnp.exp(μ + s * jax.random.normal(key, shape=(shock_size,)))

    return Model(β, μ, s, s_grid, shocks, α)
```

We define utility and production functions globally.

```
# Define utility and production functions with derivatives
u = lambda c: jnp.log(c)
u_prime = lambda c: 1 / c
u_prime_inv = lambda x: 1 / x
f = lambda k, α: k**α
f_prime = lambda k, α: α * k**(α - 1)
```

Here's the Coleman-Reffett operator using EGM.

The key JAX feature here is `vmap`, which vectorizes the computation over the grid points.

```
def K(
        c_in: jnp.ndarray,  # Consumption values on the endogenous grid
        x_in: jnp.ndarray,  # Current endogenous grid
        model: Model        # Model specification
    ):
    """
    The Coleman-Reffett operator using EGM

    """
    β, μ, s, s_grid, shocks, α = model
    σ = lambda x_val: jnp.interp(x_val, x_in, c_in)

    # Define function to compute consumption at a single grid point
    def compute_c(s):
        # Approximate marginal utility ∫ u'(σ(f(s, α)z)) f'(s, α) z ϕ(z)dz
        vals = u_prime(σ(f(s, α) * shocks)) * f_prime(s, α) * shocks
        mu = jnp.mean(vals)
        # Calculate consumption
        return u_prime_inv(β * mu)

    # Vectorize and calculate on all exogenous grid points
    compute_c_vectorized = jax.vmap(compute_c)
    c_out = compute_c_vectorized(s_grid)

    # Determine corresponding endogenous grid
    x_out = s_grid + c_out  # x_i = s_i + c_i
```

(continues on next page)

(continued from previous page)

```
    return c_out, x_out
```

Now we create a model instance.

```
model = create_model()
s_grid = model.s_grid
```

The solver uses JAX's `jax.lax.while_loop` for the iteration and is JIT-compiled for speed.

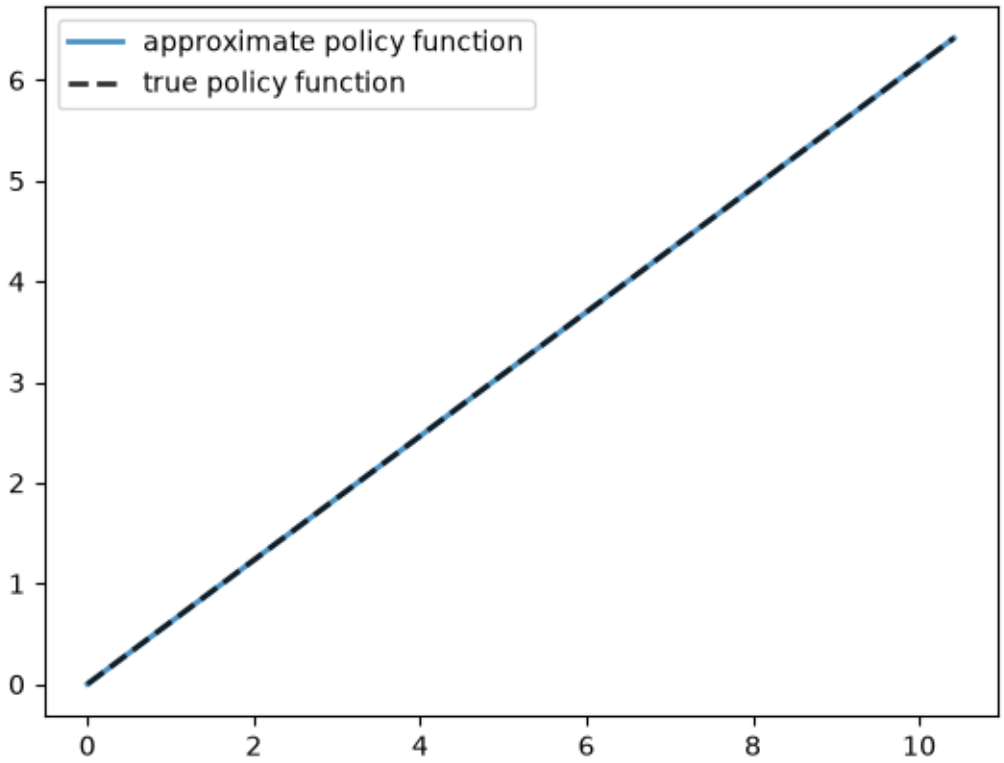
```
@jax.jit
def solve_model_time_iter(
        model: Model,
        c_init: jnp.ndarray,
        x_init: jnp.ndarray,
        tol: float = 1e-5,
        max_iter: int = 1000
    ):
    """
    Solve the model using time iteration with EGM.
    """

    def condition(loop_state):
        i, c, x, error = loop_state
        return (error > tol) & (i < max_iter)

    def body(loop_state):
        i, c, x, error = loop_state
        c_new, x_new = K(c, x, model)
        error = jnp.max(jnp.abs(c_new - c))
        return i + 1, c_new, x_new, error

    # Initialize loop state
    initial_state = (0, c_init, x_init, tol + 1)

    # Run the loop
    i, c, x, error = jax.lax.while_loop(condition, body, initial_state)

    return c, x
```

We solve the model starting from an initial guess.

```
c_init = jnp.copy(s_grid)
x_init = s_grid + c_init
c, x = solve_model_time_iter(model, c_init, x_init)
```

Let's plot the resulting policy against the analytical solution.

```
fig, ax = plt.subplots()

ax.plot(x, c, lw=2,
        alpha=0.8, label='approximate policy function')

ax.plot(x, σ_star(x, model.α, model.β), 'k--',
        lw=2, alpha=0.8, label='true policy function')

ax.legend()
plt.show()
```



The fit is very good.

```
max_dev = jnp.max(jnp.abs(c - σ_star(x, model.α, model.β)))
print(f"Maximum absolute deviation: {max_dev:.7}")
```

```
Maximum absolute deviation: 1.430511e-06
```

The JAX implementation is very fast thanks to JIT compilation and vectorization.

```
with qe.Timer(precision=8):
    c, x = solve_model_time_iter(model, c_init, x_init)
    jax.block_until_ready(c)
```

```
0.00259256 seconds elapsed
```

This speed comes from:

- JIT compilation of the entire solver
- Vectorization via `vmap` in the Coleman-Reffett operator
- Use of `jax.lax.while_loop` instead of a Python loop
- Efficient JAX array operations throughout

66.3 Exercises

Exercise 66.3.1

Solve the optimal savings problem with CRRA utility

$$u(c) = \frac{c^{1-\gamma} - 1}{1 - \gamma}$$

Compare the optimal policies for values of γ approaching 1 from above (e.g., 1.05, 1.1, 1.2).

Show that as $\gamma \to 1$, the optimal policy converges to the policy obtained with log utility ($\gamma = 1$).

Hint: Use values of γ close to 1 to ensure the endogenous grids have similar coverage and make visual comparison easier.

Solution

We need to create a version of the Coleman-Reffett operator and solver that work with CRRA utility.

The key is to parameterize the utility functions by γ.

```
def u_crra(c, γ):
    return (c**(1 - γ) - 1) / (1 - γ)

def u_prime_crra(c, γ):
    return c**(-γ)

def u_prime_inv_crra(x, γ):
    return x**(-1/γ)
```

Now we create a version of the Coleman-Reffett operator that takes γ as a parameter.

```
def K_crra(
        c_in: jnp.ndarray,  # Consumption values on the endogenous grid
        x_in: jnp.ndarray,  # Current endogenous grid
        model: Model,       # Model specification
        γ: float            # CRRA parameter
    ):
    """
    The Coleman-Reffett operator using EGM with CRRA utility
    """
    # Simplify names
    β, α = model.β, model.α
    s_grid, shocks = model.s_grid, model.shocks

    # Linear interpolation of policy using endogenous grid
    σ = lambda x_val: jnp.interp(x_val, x_in, c_in)

    # Define function to compute consumption at a single grid point
    def compute_c(s):
        vals = u_prime_crra(σ(f(s, α) * shocks), γ) * f_prime(s, α) * shocks
        return u_prime_inv_crra(β * jnp.mean(vals), γ)

    # Vectorize over grid using vmap
    compute_c_vectorized = jax.vmap(compute_c)
    c_out = compute_c_vectorized(s_grid)

    # Determine corresponding endogenous grid
    x_out = s_grid + c_out  # x_i = s_i + c_i

    return c_out, x_out
```

We also need a solver that uses this operator.

```
@jax.jit
def solve_model_crra(model: Model,
                     c_init: jnp.ndarray,
                     x_init: jnp.ndarray,
                     γ: float,
                     tol: float = 1e-5,
                     max_iter: int = 1000):
    """
    Solve the model using time iteration with EGM and CRRA utility.
    """

    def condition(loop_state):
        i, c, x, error = loop_state
        return (error > tol) & (i < max_iter)

    def body(loop_state):
        i, c, x, error = loop_state
        c_new, x_new = K_crra(c, x, model, γ)
        error = jnp.max(jnp.abs(c_new - c))
        return i + 1, c_new, x_new, error

    # Initialize loop state
    initial_state = (0, c_init, x_init, tol + 1)

    # Run the loop
    i, c, x, error = jax.lax.while_loop(condition, body, initial_state)

    return c, x
```

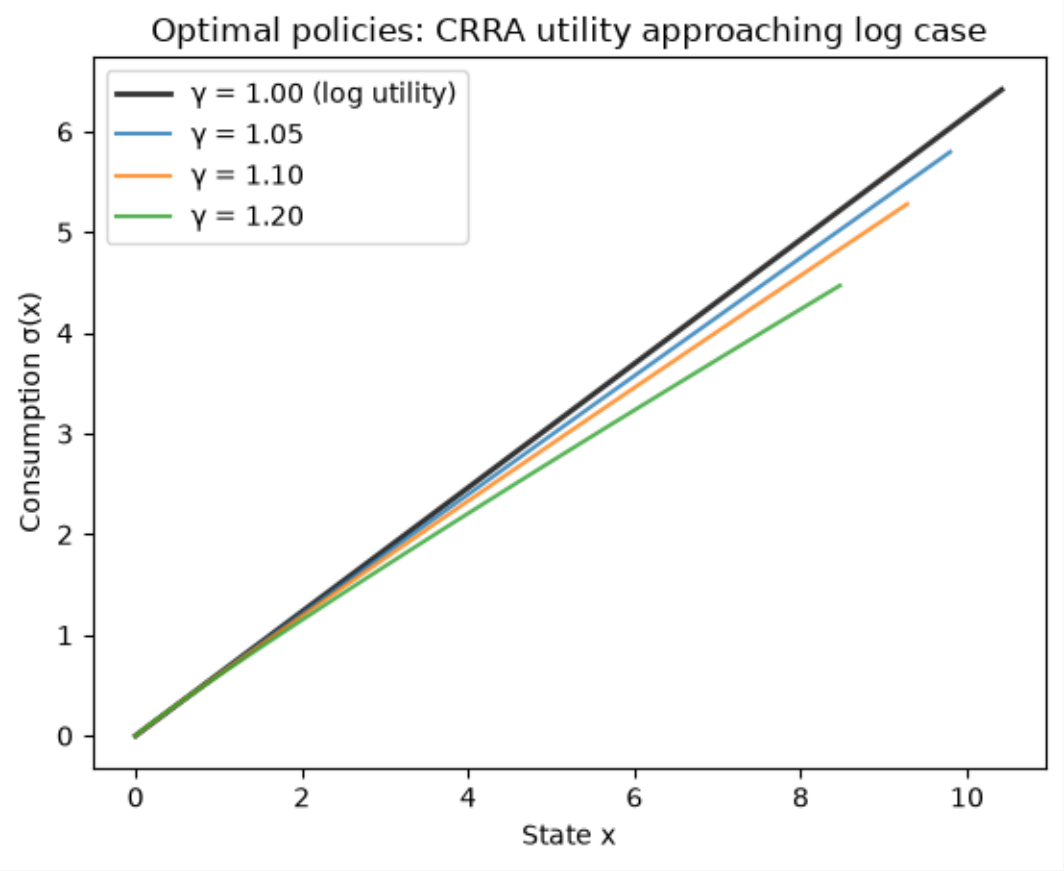
Now we solve for $\gamma = 1$ (log utility) and values approaching 1 from above.

```
γ_values = [1.0, 1.05, 1.1, 1.2]
policies = {}
endogenous_grids = {}

model_crra = create_model()

for γ in γ_values:
    c_init = jnp.copy(model_crra.s_grid)
    x_init = model_crra.s_grid + c_init
    c_gamma, x_gamma = solve_model_crra(model_crra, c_init, x_init, γ)
    jax.block_until_ready(c_gamma)
    policies[γ] = c_gamma
    endogenous_grids[γ] = x_gamma
    print(f"Solved for γ = {γ}")
```

```
Solved for γ = 1.0
Solved for γ = 1.05
Solved for γ = 1.1
Solved for γ = 1.2
```

Plot the policies on their endogenous grids.

```
fig, ax = plt.subplots()

for γ in γ_values:
    x = endogenous_grids[γ]
    if γ == 1.0:
        ax.plot(x, policies[γ], 'k-', linewidth=2,
                label=f'γ = {γ:.2f} (log utility)', alpha=0.8)
    else:
        ax.plot(x, policies[γ], label=f'γ = {γ:.2f}', alpha=0.8)

ax.set_xlabel('State x')
ax.set_ylabel('Consumption σ(x)')
ax.legend()
ax.set_title('Optimal policies: CRRA utility approaching log case')
plt.show()
```



Note that the plots for $\gamma > 1$ do not cover the entire x-axis range shown.

This is because the endogenous grid $x = s + \sigma(s)$ depends on the consumption policy, which varies with γ.

Let's check the maximum deviation between the log utility case ($\gamma = 1.0$) and values approaching from above.

```
for γ in [1.05, 1.1, 1.2]:
    max_diff = jnp.max(jnp.abs(policies[1.0] - policies[γ]))
    print(f"Max difference between γ=1.0 and γ={γ}: {max_diff:.6}")
```

```
Max difference between γ=1.0 and γ=1.05: 0.619199
Max difference between γ=1.0 and γ=1.1: 1.1362
Max difference between γ=1.0 and γ=1.2: 1.94592
```

As expected, the differences decrease as γ approaches 1 from above, confirming convergence.

Part X

Household Problems

CHAPTER

SIXTYSEVEN

THE INCOME FLUCTUATION PROBLEM I: DISCRETIZATION AND VFI

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

67.1 Overview

In this lecture, we study an optimal savings problem for an infinitely lived consumer—the "common ancestor" described in [Ljungqvist and Sargent, 2018], section 1.3.

This savings problem is often called an **income fluctuation problem** or a **household problem**.

It is an essential sub-problem for many representative macroeconomic models

- [Aiyagari, 1994]
- [Huggett, 1993]
- etc.

It is related to the decision problem in *Optimal Savings III: Stochastic Returns* but differs in significant ways.

For example,

1. The choice problem for the agent includes an additive income term that leads to an occasionally binding constraint.
2. Shocks affecting the budget constraint are correlated, forcing us to track an extra state variable.

We will begin by working with a relatively basic version of the model and solving it via old-fashioned discretization + value function iteration.

Although this approach is not the fastest or the most efficient, it is very robust and flexible.

For example, if we suddenly decided to add Epstein–Zin preferences, or modify ordinary conditional expectations to quantiles, the technique would continue to work well.

> **Note**
>
> The same is not true of some other methods we will deploy, such as the endogenous grid method.
>
> This is a general rule of computation and analysis — while we can often come up with faster algorithms by exploiting structure, these new algorithms are typically less robust.
>
> They are less robust precisely because they exploit more structure — which implies that they are, inevitably, more vulnerable to change.

In addition to Anaconda, this lecture will need the following libraries:

```
!pip install quantecon jax
```

We will use the following imports:

```
import quantecon as qe
import jax
import jax.numpy as jnp
import matplotlib.pyplot as plt
from typing import NamedTuple
from time import time
```

We'll use 64 bit floats to gain extra precision.

```
jax.config.update("jax_enable_x64", True)
```

67.2 Set Up

We study a household that chooses a state-contingent consumption plan $\{c_t\}_{t \geq 0}$ to maximize

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t u(c_t)$$

subject to

$$a_{t+1} + c_t \leq R a_t + y_t$$

Here

- c_t is consumption and $c_t \geq 0$,
- a_t is assets and $a_t \geq 0$,
- $R = 1 + r$ is a gross rate of return, and
- $(y_t)_{t \geq 0}$ is labor income, taking values in some finite set Y.

We assume below that labor income dynamics follow a discretized AR(1) process.

We set $\mathsf{S} := \mathbb{R}_+ \times \mathsf{Y}$, which represents the state space.

The **value function** $V \colon \mathsf{S} \to \mathbb{R}$ is defined by

$$V(a, y) := \max \, \mathbb{E} \left\{ \sum_{t=0}^{\infty} \beta^t u(c_t) \right\} \tag{67.1}$$

where the maximization is over all feasible consumption sequences given $(a_0, y_0) = (a, y)$.

The Bellman equation is

$$v(a, y) = \max_{0 \leq a' \leq Ra+y} \left\{ u(Ra + y - a') + \beta \sum_{y'} v(a', y') Q(y, y') \right\}$$

where

$$u(c) = \frac{c^{1-\gamma}}{1-\gamma}$$

In the code we use the function

$$B((a, y), a', v) = u(Ra + y - a') + \beta \sum_{y'} v(a', y') Q(y, y').$$

the encapsulate the right hand side of the Bellman equation.

67.3 Code

The following code defines a `NamedTuple` to store the model parameters and grids.

```
class Model(NamedTuple):
    β: float              # Discount factor
    R: float              # Gross interest rate
    γ: float              # CRRA parameter
    a_grid: jnp.ndarray   # Asset grid
    y_grid: jnp.ndarray   # Income grid
    Q: jnp.ndarray        # Markov matrix for income


def create_consumption_model(
        R=1.01,                    # Gross interest rate
        β=0.98,                    # Discount factor
        γ=2,                       # CRRA parameter
        a_min=0.01,                # Min assets
        a_max=10.0,                # Max assets
        a_size=150,                # Grid size
        ρ=0.9, ν=0.1, y_size=100   # Income parameters
    ):
    """
    Creates an instance of the consumption-savings model.

    """
    a_grid = jnp.linspace(a_min, a_max, a_size)
    mc = qe.tauchen(n=y_size, rho=ρ, sigma=ν)
    y_grid, Q = jnp.exp(mc.state_values), jax.device_put(mc.P)
    return Model(β, R, γ, a_grid, y_grid, Q)
```

Now we define the right hand side of the Bellman equation.

We'll use a vectorized coding style reminiscent of Matlab and NumPy (avoiding all loops).

Your are invited to explore an alternative style based around `jax.vmap` in the Exercises.

```
@jax.jit
def B(v, model):
    """
    A vectorized version of the right-hand side of the Bellman equation
    (before maximization), which is a 3D array representing

        B(a, y, a') = u(Ra + y - a') + β Σ_y' v(a', y') Q(y, y')

    for all (a, y, a').
    """

    # Unpack
    β, R, γ, a_grid, y_grid, Q = model
    a_size, y_size = len(a_grid), len(y_grid)

    # Compute current rewards r(a, y, ap) as array r[i, j, ip]
    a  = jnp.reshape(a_grid, (a_size, 1, 1))    # a[i]   ->  a[i, j, ip]
    y  = jnp.reshape(y_grid, (1, y_size, 1))    # z[j]   ->  z[i, j, ip]
    ap = jnp.reshape(a_grid, (1, 1, a_size))    # ap[ip] -> ap[i, j, ip]
    c = R * a + y - ap

    # Calculate continuation rewards at all combinations of (a, y, ap)
    v = jnp.reshape(v, (1, 1, a_size, y_size))  # v[ip, jp] -> v[i, j, ip, jp]
    Q = jnp.reshape(Q, (1, y_size, 1, y_size))  # Q[j, jp]  -> Q[i, j, ip, jp]
    EV = jnp.sum(v * Q, axis=3)                 # sum over last index jp

    # Compute the right-hand side of the Bellman equation
    return jnp.where(c > 0, c**(1-γ)/(1-γ) + β * EV, -jnp.inf)
```

Some readers might be concerned that we are creating high dimensional arrays, leading to inefficiency.

Could they be avoided by more careful vectorization?

In fact this is not necessary: this function will be JIT-compiled by JAX, and the JIT compiler will optimize compiled code to minimize memory use.

The Bellman operator T can be implemented by

```
@jax.jit
def T(v, model):
    "The Bellman operator."
    return jnp.max(B(v, model), axis=2)
```

The next function computes a v-greedy policy given v (i.e., the policy that maximizes the right-hand side of the Bellman equation.)

```
@jax.jit
def get_greedy(v, model):
    "Computes a v-greedy policy, returned as a set of indices."
    return jnp.argmax(B(v, model), axis=2)
```

67.3.1 Value function iteration

Now we define a solver that implements VFI.

First we write a simple version using a standard Python loop.

```
def value_function_iteration_python(model, tol=1e-5, max_iter=10_000):
    """
    Implements VFI using successive approximation with a Python loop.
    """
    v = jnp.zeros((len(model.a_grid), len(model.y_grid)))
    error = tol + 1
    k = 0

    while error > tol and k < max_iter:
        v_new = T(v, model)
        error = jnp.max(jnp.abs(v_new - v))
        v = v_new
        k += 1

    return v, get_greedy(v, model)
```

Next we write a version that uses `jax.lax.while_loop`.

```
@jax.jit
def value_function_iteration(model, tol=1e-5, max_iter=10_000):
    """
    Implements VFI using successive approximation.
    """
    def body_fun(k_v_err):
        k, v, error = k_v_err
        v_new = T(v, model)
        error = jnp.max(jnp.abs(v_new - v))
        return k + 1, v_new, error

    def cond_fun(k_v_err):
        k, v, error = k_v_err
        return jnp.logical_and(error > tol, k < max_iter)

    v_init = jnp.zeros((len(model.a_grid), len(model.y_grid)))
    k, v_star, error = jax.lax.while_loop(cond_fun, body_fun,
                                          (1, v_init, tol + 1))
    return v_star, get_greedy(v_star, model)
```

67.3.2 Timing

Let's create an instance and compare the two implementations.

```
model = create_consumption_model()
```

First let's time the Python version.

```
print("Starting VFI using Python loop.")
start = time()
v_star_python, σ_star_python = value_function_iteration_python(model)
python_time = time() - start
print(f"VFI completed in {python_time} seconds.")
```

```
Starting VFI using Python loop.
VFI completed in 2.350461721420288 seconds.
```

Now let's time the `jax.lax.while_loop` version.

```
print("Starting VFI using jax.lax.while_loop.")
start = time()
v_star_jax, σ_star_jax = value_function_iteration(model)
v_star_jax.block_until_ready()
jax_with_compile = time() - start
print(f"VFI completed in {jax_with_compile} seconds.")
```

```
Starting VFI using jax.lax.while_loop.
VFI completed in 1.6007442474365234 seconds.
```

Let's run it again to eliminate compile time.

```
start = time()
v_star_jax, σ_star_jax = value_function_iteration(model)
v_star_jax.block_until_ready()
jax_without_compile = time() - start
print(f"VFI completed in {jax_without_compile} seconds.")
```

```
VFI completed in 1.3688461780548096 seconds.
```

Let's check that the two implementations produce the same result.

```
print(f"Values match: {jnp.allclose(v_star_python, v_star_jax)}")
print(f"Policies match: {jnp.allclose(σ_star_python, σ_star_jax)}")
```

```
Values match: True
Policies match: True
```

Here's the speedup from using `jax.lax.while_loop`.

```
print(f"Relative speed = {python_time / jax_without_compile:.2f}")
```

```
Relative speed = 1.72
```

67.3.3 Asset Dynamics

To understand long-run behavior, let's examine the asset accumulation dynamics under the optimal policy.

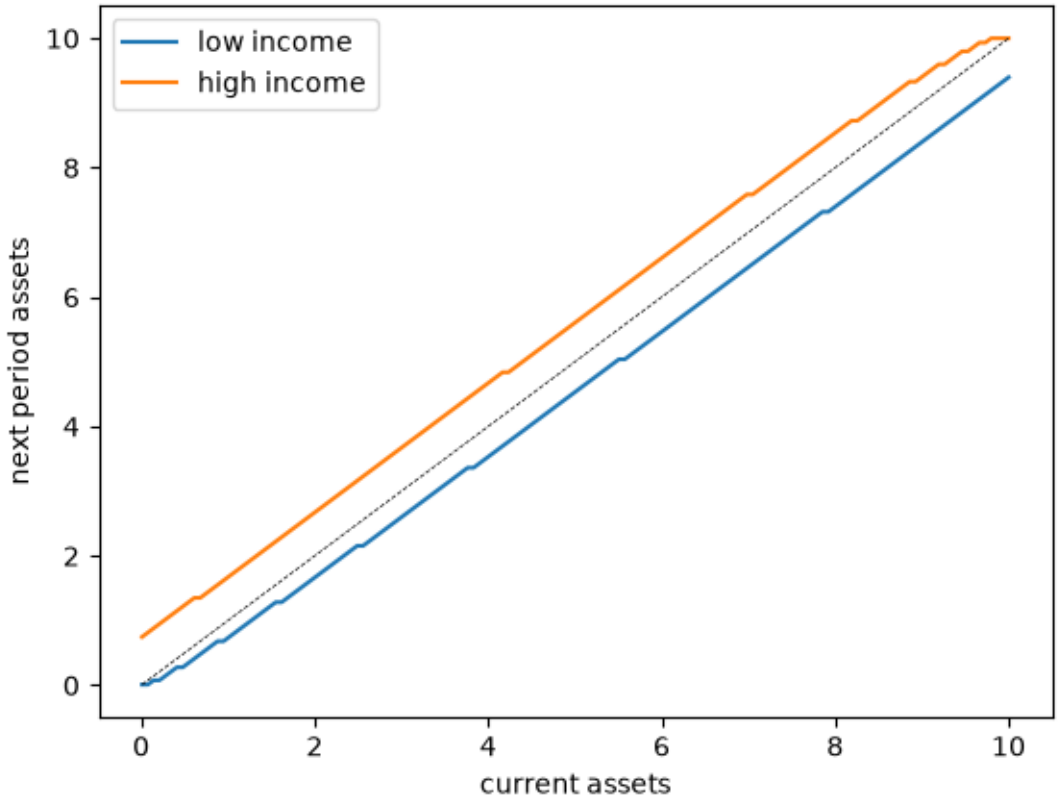
The following 45-degree diagram shows how assets evolve over time:

```
fig, ax = plt.subplots()

# Plot asset accumulation for first and last income states
for j, label in zip([0, -1], ['low income', 'high income']):
    # Get next-period assets for each current asset level
    a_next = model.a_grid[σ_star_jax[:, j]]
    ax.plot(model.a_grid, a_next, label=label)

# Add 45-degree line
```

(continues on next page)

(continued from previous page)

```
ax.plot(model.a_grid, model.a_grid, 'k--', linewidth=0.5)
ax.set(xlabel='current assets', ylabel='next period assets')
ax.legend()
plt.show()
```



The plot shows the asset accumulation rule for each income state.

The dotted line is the 45-degree line, representing points where $a_{t+1} = a_t$.

We see that:

- For low income levels, assets tend to decrease (points below the 45-degree line)
- For high income levels, assets tend to increase at low asset levels
- The dynamics suggest convergence to a stationary distribution

67.4 Exercises

Exercise 67.4.1

In this exercise, we explore an alternative approach to implementing value function iteration using `jax.vmap`.

For this simple optimal savings problem, direct vectorization is relatively easy.

In particular, it's straightforward to express the right hand side of the Bellman equation as an array that stores evaluations of the function at every state and control.

However, for more complex models, direct vectorization can be much harder.

For this reason, it helps to have another approach to fast JAX implementations up our sleeves.

Your task is to implement a version that:

1. writes the right hand side of the Bellman operator as a function of individual states and controls, and
2. applies `jax.vmap` on the outside to achieve a parallelized solution.

Specifically:

1. Rewrite `B` to take indices `(i, j, ip)` corresponding to `(a, y, a')` and compute the Bellman equation for those specific indices.
2. Use `jax.vmap` successively to vectorize over all indices (use staged vmap as shown in earlier examples).
3. Implement `T_vmap` and `get_greedy_vmap` functions using the vectorized `B`.
4. Implement `value_iteration_vmap` using `jax.lax.while_loop`.
5. Test that your implementation produces the same results as the direct vectorization approach.
6. Compare the execution times of both approaches.

Solution

Here's one solution.

First let's rewrite `B` to work with individual indices:

```
def B(v, model, i, j, ip):
    """
    The right-hand side of the Bellman equation before maximization, which takes
    the form

        B(a, y, a') = u(Ra + y - a') + β Σ_y' v(a', y') Q(y, y')

    The indices are (i, j, ip) -> (a, y, a').
    """
    β, R, γ, a_grid, y_grid, Q = model
    a, y, ap  = a_grid[i], y_grid[j], a_grid[ip]
    c = R * a + y - ap
    EV = jnp.sum(v[ip, :] * Q[j, :])
    return jnp.where(c > 0, c**(1-γ)/(1-γ) + β * EV, -jnp.inf)
```

Now we successively apply `vmap` to simulate nested loops.

```
B_1    = jax.vmap(B,   in_axes=(None, None, None, None, 0))
B_2    = jax.vmap(B_1, in_axes=(None, None, None, 0,    None))
B_vmap = jax.vmap(B_2, in_axes=(None, None, 0,    None, None))
```

Here's the Bellman operator and the `get_greedy` functions for the `vmap` case.

```
@jax.jit
def T_vmap(v, model):
    "The Bellman operator."
    a_indices = jnp.arange(len(model.a_grid))
    y_indices = jnp.arange(len(model.y_grid))
    B_values = B_vmap(v, model, a_indices, y_indices, a_indices)
    return jnp.max(B_values, axis=-1)

@jax.jit
def get_greedy_vmap(v, model):
    "Computes a v-greedy policy, returned as a set of indices."
    a_indices = jnp.arange(len(model.a_grid))
    y_indices = jnp.arange(len(model.y_grid))
    B_values = B_vmap(v, model, a_indices, y_indices, a_indices)
    return jnp.argmax(B_values, axis=-1)
```

Here's the iteration routine.

```
def value_iteration_vmap(model, tol=1e-5, max_iter=10_000):
    """
    Implements VFI using vmap and successive approximation.
    """
    def body_fun(k_v_err):
        k, v, error = k_v_err
        v_new = T_vmap(v, model)
        error = jnp.max(jnp.abs(v_new - v))
        return k + 1, v_new, error

    def cond_fun(k_v_err):
        k, v, error = k_v_err
        return jnp.logical_and(error > tol, k < max_iter)

    v_init = jnp.zeros((len(model.a_grid), len(model.y_grid)))
    k, v_star, error = jax.lax.while_loop(cond_fun, body_fun,
                                          (1, v_init, tol + 1))
    return v_star, get_greedy_vmap(v_star, model)
```

Let's see how long it takes to solve the model using the `vmap` method.

```
print("Starting VFI using vmap.")
start = time()
v_star_vmap, σ_star_vmap = value_iteration_vmap(model)
v_star_vmap.block_until_ready()
jax_vmap_with_compile = time() - start
print(f"VFI completed in {jax_vmap_with_compile} seconds.")
```

```
Starting VFI using vmap.
VFI completed in 0.6219601631164551 seconds.
```

Let's run it again to get rid of compile time.

```
start = time()
v_star_vmap, σ_star_vmap = value_iteration_vmap(model)
v_star_vmap.block_until_ready()
jax_vmap_without_compile = time() - start
print(f"VFI completed in {jax_vmap_without_compile} seconds.")
```

```
VFI completed in 0.37813782691955566 seconds.
```

We need to make sure that we got the same result.

```
print(jnp.allclose(v_star_vmap, v_star_jax))
print(jnp.allclose(σ_star_vmap, σ_star_jax))
```

```
True
True
```

Here's the comparison with the first JAX implementation (which used direct vectorization).

```
print(f"Relative speed = {jax_without_compile / jax_vmap_without_compile}")
```

```
Relative speed = 3.619966267870935
```

The execution times for the two JAX versions are relatively similar.

However, as emphasized above, having a second method up our sleeves (i.e, the `vmap` approach) will be helpful when confronting dynamic programs with more sophisticated Bellman equations.

CHAPTER

SIXTYEIGHT

THE INCOME FLUCTUATION PROBLEM II: OPTIMISTIC POLICY ITERATION

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

68.1 Overview

In *The Income Fluctuation Problem I: Discretization and VFI* we studied the income fluctuation problem and solved it using value function iteration (VFI).

In this lecture we'll solve the same problem using **optimistic policy iteration** (OPI), which is very general, typically faster than VFI and only slightly more complex.

OPI combines elements of both value function iteration and policy iteration.

A detailed discussion of the algorithm can be found in DP1.

Here our aim is to implement OPI and test whether or not it yields significant speed improvements over standard VFI for the income fluctuation problem.

In addition to Anaconda, this lecture will need the following libraries:

```
!pip install quantecon jax
```

We will use the following imports:

```
import quantecon as qe
import jax
import jax.numpy as jnp
import matplotlib.pyplot as plt
from typing import NamedTuple
from time import time
```

68.2 Model and Primitives

The model and parameters are the same as in *The Income Fluctuation Problem I: Discretization and VFI*.

We repeat the key elements here for convenience.

The household's problem is to maximize

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t u(c_t)$$

subject to

$$a_{t+1} + c_t \leq R a_t + y_t$$

where $u(c) = c^{1-\gamma}/(1-\gamma)$.

Here's the model structure:

```
class Model(NamedTuple):
    β: float              # Discount factor
    R: float              # Gross interest rate
    γ: float              # CRRA parameter
    a_grid: jnp.ndarray   # Asset grid
    y_grid: jnp.ndarray   # Income grid
    Q: jnp.ndarray        # Markov matrix for income


def create_consumption_model(
        R=1.01,                    # Gross interest rate
        β=0.98,                    # Discount factor
        γ=2,                       # CRRA parameter
        a_min=0.01,                # Min assets
        a_max=10.0,                # Max assets
        a_size=150,                # Grid size
        ρ=0.9, ν=0.1, y_size=100   # Income parameters
    ):
    """
    Creates an instance of the consumption-savings model.

    """
    a_grid = jnp.linspace(a_min, a_max, a_size)
    mc = qe.tauchen(n=y_size, rho=ρ, sigma=ν)
    y_grid, Q = jnp.exp(mc.state_values), jax.device_put(mc.P)
    return Model(β, R, γ, a_grid, y_grid, Q)
```

68.3 Operators and Policies

We repeat some functions from *The Income Fluctuation Problem I: Discretization and VFI*.

Here is the right hand side of the Bellman equation:

```
def B(v, model, i, j, ip):
    """
    The right-hand side of the Bellman equation before maximization, which takes
    the form
```

(continues on next page)

(continued from previous page)

```
        B(a, y, a') = u(Ra + y - a') + β Σ_y' v(a', y') Q(y, y')

    The indices are (i, j, ip) -> (a, y, a').
    """
    β, R, γ, a_grid, y_grid, Q = model
    a, y, ap  = a_grid[i], y_grid[j], a_grid[ip]
    c = R * a + y - ap
    EV = jnp.sum(v[ip, :] * Q[j, :])
    return jnp.where(c > 0, c**(1-γ)/(1-γ) + β * EV, -jnp.inf)
```

Now we successively apply `vmap` to vectorize over all indices:

```
B_1    = jax.vmap(B,   in_axes=(None, None, None, None, 0))
B_2    = jax.vmap(B_1, in_axes=(None, None, None, 0,    None))
B_vmap = jax.vmap(B_2, in_axes=(None, None, 0,    None, None))
```

Here's the Bellman operator:

```
def T(v, model):
    "The Bellman operator."
    a_indices = jnp.arange(len(model.a_grid))
    y_indices = jnp.arange(len(model.y_grid))
    B_values = B_vmap(v, model, a_indices, y_indices, a_indices)
    return jnp.max(B_values, axis=-1)
```

Here's the function that computes a v-greedy policy:

```
def get_greedy(v, model):
    "Computes a v-greedy policy, returned as a set of indices."
    a_indices = jnp.arange(len(model.a_grid))
    y_indices = jnp.arange(len(model.y_grid))
    B_values = B_vmap(v, model, a_indices, y_indices, a_indices)
    return jnp.argmax(B_values, axis=-1)
```

Now we define the policy operator T_σ, which is the Bellman operator with policy σ fixed.

For a given policy σ, the policy operator is defined by

$$(T_\sigma v)(a, y) = u(Ra + y - \sigma(a, y)) + \beta \sum_{y'} v(\sigma(a, y), y') Q(y, y')$$

```
def T_σ(v, σ, model, i, j):
    """
    The σ-policy operator for indices (i, j) -> (a, y).
    """
    β, R, γ, a_grid, y_grid, Q = model

    # Get values at current state
    a, y = a_grid[i], y_grid[j]
    # Get policy choice
    ap = a_grid[σ[i, j]]

    # Compute current reward
    c = R * a + y - ap
    r = jnp.where(c > 0, c**(1-γ)/(1-γ), -jnp.inf)
```

(continues on next page)

(continued from previous page)

```
    # Compute expected value
    EV = jnp.sum(v[σ[i, j], :] * Q[j, :])

    return r + β * EV
```

Apply vmap to vectorize:

```
T_σ_1    = jax.vmap(T_σ,   in_axes=(None, None, None, None, 0))
T_σ_vmap = jax.vmap(T_σ_1, in_axes=(None, None, None, 0,    None))

def T_σ_vec(v, σ, model):
    """Vectorized version of T_σ."""
    a_size, y_size = len(model.a_grid), len(model.y_grid)
    a_indices = jnp.arange(a_size)
    y_indices = jnp.arange(y_size)
    return T_σ_vmap(v, σ, model, a_indices, y_indices)
```

Now we need a function to apply the policy operator m times:

```
def iterate_policy_operator(σ, v, m, model):
    """
    Apply the policy operator T_σ exactly m times to v.
    """
    def update(i, v):
        return T_σ_vec(v, σ, model)

    v = jax.lax.fori_loop(0, m, update, v)
    return v
```

68.4 Value Function Iteration

For comparison, here's VFI from *The Income Fluctuation Problem I: Discretization and VFI*:

```
@jax.jit
def value_function_iteration(model, tol=1e-5, max_iter=10_000):
    """
    Implements VFI using successive approximation.
    """
    def body_fun(k_v_err):
        k, v, error = k_v_err
        v_new = T(v, model)
        error = jnp.max(jnp.abs(v_new - v))
        return k + 1, v_new, error

    def cond_fun(k_v_err):
        k, v, error = k_v_err
        return jnp.logical_and(error > tol, k < max_iter)

    v_init = jnp.zeros((len(model.a_grid), len(model.y_grid)))
    k, v_star, error = jax.lax.while_loop(cond_fun, body_fun,
                                          (1, v_init, tol + 1))
    return v_star, get_greedy(v_star, model)
```

68.5 Optimistic Policy Iteration

Now we implement OPI.

The algorithm alternates between

1. Performing m policy operator iterations to update the value function
2. Computing a new greedy policy based on the updated value function

```
@jax.jit
def optimistic_policy_iteration(model, m=10, tol=1e-5, max_iter=10_000):
    """
    Implements optimistic policy iteration with step size m.

    Parameters:
    -----------
    model : Model
        The consumption-savings model
    m : int
        Number of policy operator iterations per step
    tol : float
        Tolerance for convergence
    max_iter : int
        Maximum number of iterations
    """
    v_init = jnp.zeros((len(model.a_grid), len(model.y_grid)))

    def condition_function(inputs):
        i, v, error = inputs
        return jnp.logical_and(error > tol, i < max_iter)

    def update(inputs):
        i, v, error = inputs
        last_v = v
        σ = get_greedy(v, model)
        v = iterate_policy_operator(σ, v, m, model)
        error = jnp.max(jnp.abs(v - last_v))
        i += 1
        return i, v, error

    num_iter, v, error = jax.lax.while_loop(condition_function,
                                            update,
                                            (0, v_init, tol + 1))

    return v, get_greedy(v, model)
```

68.6 Timing Comparison

Let's create a model and compare the performance of VFI and OPI.

```
model = create_consumption_model()
```

First, let's time VFI:

```
print("Starting VFI.")
start = time()
v_star_vfi, σ_star_vfi = value_function_iteration(model)
v_star_vfi.block_until_ready()
vfi_time_with_compile = time() - start
print(f"VFI completed in {vfi_time_with_compile:.2f} seconds.")
```

```
Starting VFI.
VFI completed in 0.39 seconds.
```

Run it again to eliminate compile time:

```
start = time()
v_star_vfi, σ_star_vfi = value_function_iteration(model)
v_star_vfi.block_until_ready()
vfi_time = time() - start
print(f"VFI completed in {vfi_time:.2f} seconds.")
```

```
VFI completed in 0.09 seconds.
```

Now let's time OPI with different values of m:

```
print("Starting OPI with m=50.")
start = time()
v_star_opi, σ_star_opi = optimistic_policy_iteration(model, m=50)
v_star_opi.block_until_ready()
opi_time_with_compile = time() - start
print(f"OPI completed in {opi_time_with_compile:.2f} seconds.")
```

```
Starting OPI with m=50.
OPI completed in 0.34 seconds.
```

Run it again:

```
start = time()
v_star_opi, σ_star_opi = optimistic_policy_iteration(model, m=50)
v_star_opi.block_until_ready()
opi_time = time() - start
print(f"OPI completed in {opi_time:.2f} seconds.")
```

```
OPI completed in 0.04 seconds.
```

Check that we get the same result:

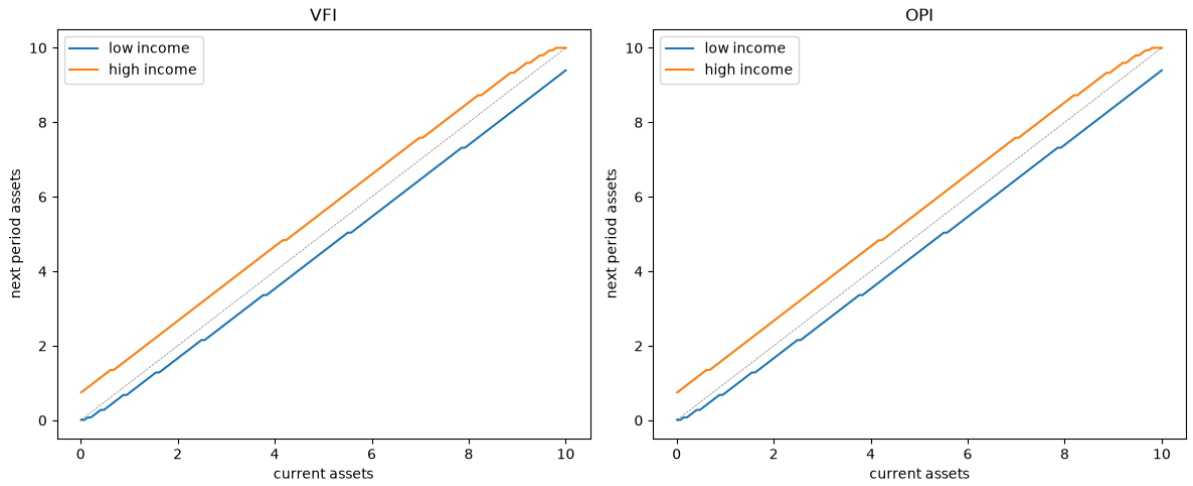
```
print(f"Values match: {jnp.allclose(v_star_vfi, v_star_opi)}")
```

```
Values match: True
```

The value functions match, confirming both algorithms converge to the same solution.

Let's visually compare the asset dynamics under both policies:

```
fig, axes = plt.subplots(1, 2, figsize=(12, 5))

# VFI policy
for j, label in zip([0, -1], ['low income', 'high income']):
    a_next_vfi = model.a_grid[σ_star_vfi[:, j]]
    axes[0].plot(model.a_grid, a_next_vfi, label=label)
axes[0].plot(model.a_grid, model.a_grid, 'k--', linewidth=0.5, alpha=0.5)
axes[0].set(xlabel='current assets', ylabel='next period assets', title='VFI')
axes[0].legend()

# OPI policy
for j, label in zip([0, -1], ['low income', 'high income']):
    a_next_opi = model.a_grid[σ_star_opi[:, j]]
    axes[1].plot(model.a_grid, a_next_opi, label=label)
axes[1].plot(model.a_grid, model.a_grid, 'k--', linewidth=0.5, alpha=0.5)
axes[1].set(xlabel='current assets', ylabel='next period assets', title='OPI')
axes[1].legend()

plt.tight_layout()
plt.show()
```



The policies are visually indistinguishable, confirming both methods produce the same solution.

Here's the speedup:

```
print(f"Speedup factor: {vfi_time / opi_time:.2f}")
```

```
Speedup factor: 2.42
```

Let's try different values of m to see how it affects performance:

```
m_vals = [1, 5, 10, 25, 50, 100, 200, 400]
opi_times = []
```

(continues on next page)

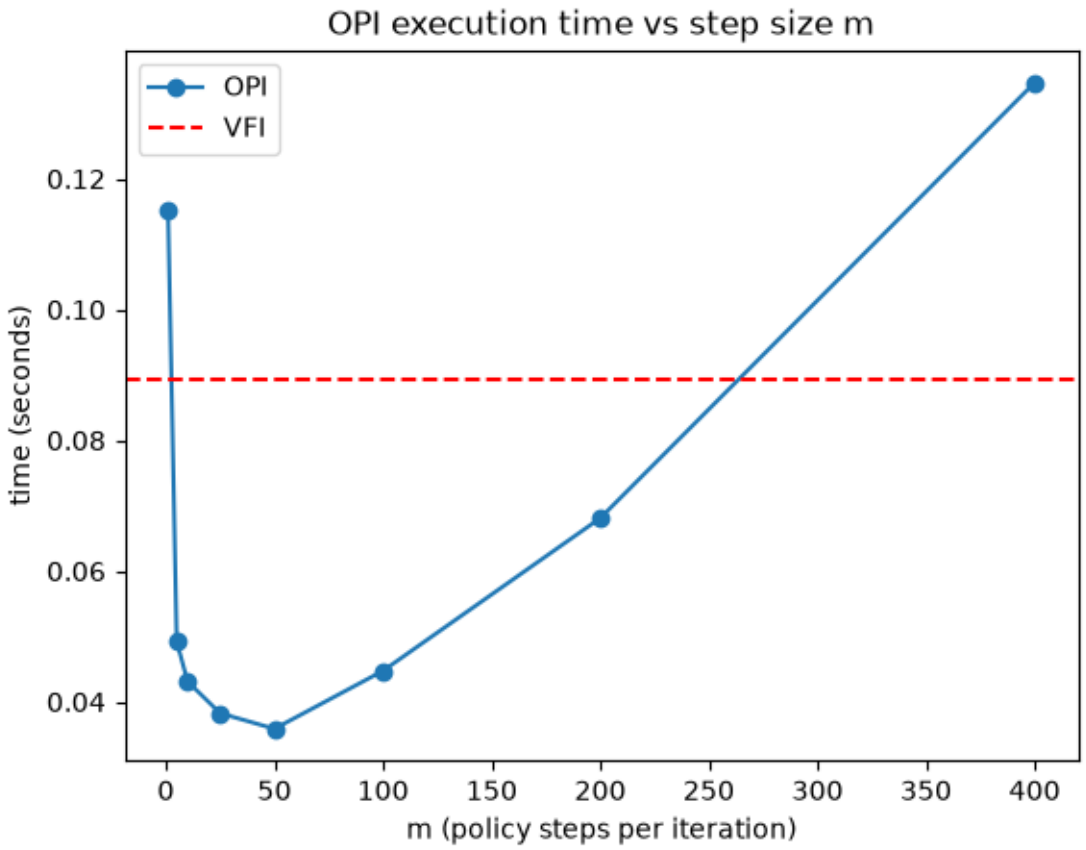
(continued from previous page)

```
for m in m_vals:
    start = time()
    v_star, σ_star = optimistic_policy_iteration(model, m=m)
    v_star.block_until_ready()
    elapsed = time() - start
    opi_times.append(elapsed)
    print(f"OPI with m={m:3d} completed in {elapsed:.2f} seconds.")
```

```
OPI with m=  1 completed in 0.12 seconds.
OPI with m=  5 completed in 0.05 seconds.
OPI with m= 10 completed in 0.04 seconds.
OPI with m= 25 completed in 0.04 seconds.
OPI with m= 50 completed in 0.04 seconds.
OPI with m=100 completed in 0.04 seconds.
OPI with m=200 completed in 0.07 seconds.
OPI with m=400 completed in 0.13 seconds.
```

Plot the results:

```
fig, ax = plt.subplots()
ax.plot(m_vals, opi_times, 'o-', label='OPI')
ax.axhline(vfi_time, linestyle='--', color='red', label='VFI')
ax.set_xlabel('m (policy steps per iteration)')
ax.set_ylabel('time (seconds)')
ax.legend()
ax.set_title('OPI execution time vs step size m')
plt.show()
```



Here's a summary of the results

- OPI outperforms VFI for a large range of m values.
- For very large m, OPI performance begins to degrade as we spend too much time iterating the policy operator.

68.7 Exercises

Exercise 68.7.1

The speed gains achieved by OPI are quite robust to parameter changes.

Confirm this by experimenting with different parameter values for the income process (ρ and ν).

Measure how they affect the relative performance of VFI vs OPI.

Try:

- $\rho \in \{0.8, 0.9, 0.95\}$
- $\nu \in \{0.05, 0.1, 0.2\}$

For each combination, compute the speedup factor (VFI time / OPI time) and report your findings.

Solution

Here's one solution:

```
ρ_vals = [0.8, 0.9, 0.95]
ν_vals = [0.05, 0.1, 0.2]

results = []

for ρ in ρ_vals:
    for ν in ν_vals:
        print(f"\nTesting ρ={ρ}, ν={ν}")

        # Create model
        model = create_consumption_model(ρ=ρ, ν=ν)

        # Time VFI
        start = time()
        v_vfi, σ_vfi = value_function_iteration(model)
        v_vfi.block_until_ready()
        vfi_t = time() - start

        # Time OPI
        start = time()
        v_opi, σ_opi = optimistic_policy_iteration(model, m=10)
        v_opi.block_until_ready()
        opi_t = time() - start

        speedup = vfi_t / opi_t
        results.append((ρ, ν, speedup))
        print(f"  VFI: {vfi_t:.2f}s, OPI: {opi_t:.2f}s, Speedup: {speedup:.2f}x")

# Print summary
print("\nSummary of speedup factors:")
for ρ, ν, speedup in results:
    print(f"ρ={ρ}, ν={ν}: {speedup:.2f}x")
```

```
Testing ρ=0.8, ν=0.05
  VFI: 0.05s, OPI: 0.03s, Speedup: 1.65x

Testing ρ=0.8, ν=0.1
  VFI: 0.06s, OPI: 0.03s, Speedup: 1.74x

Testing ρ=0.8, ν=0.2
  VFI: 0.06s, OPI: 0.03s, Speedup: 1.76x

Testing ρ=0.9, ν=0.05
  VFI: 0.06s, OPI: 0.03s, Speedup: 1.79x

Testing ρ=0.9, ν=0.1
  VFI: 0.06s, OPI: 0.03s, Speedup: 1.70x

Testing ρ=0.9, ν=0.2
  VFI: 0.06s, OPI: 0.03s, Speedup: 1.76x

Testing ρ=0.95, ν=0.05
  VFI: 0.06s, OPI: 0.04s, Speedup: 1.70x

Testing ρ=0.95, ν=0.1
  VFI: 0.06s, OPI: 0.03s, Speedup: 1.74x

Testing ρ=0.95, ν=0.2
  VFI: 0.07s, OPI: 0.04s, Speedup: 1.84x

Summary of speedup factors:
ρ=0.8, ν=0.05: 1.65x
```

CHAPTER

SIXTYNINE

THE INCOME FLUCTUATION PROBLEM III: THE ENDOGENOUS GRID METHOD

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

Contents

- *The Income Fluctuation Problem III: The Endogenous Grid Method*
  - *Overview*
  - *The Household Problem*
  - *Computation*
  - *NumPy Implementation*
  - *JAX Implementation*
  - *Simulation*

69.1 Overview

In this lecture we continue examining a version of the IFP from

- *The Income Fluctuation Problem I: Discretization and VFI* and
- *The Income Fluctuation Problem II: Optimistic Policy Iteration*.

We will make two changes.

1. Change the timing to one that is more efficient for our set up.
2. Use the endogenous grid method (EGM) to solve the model.

We use EGM because we know it to be fast and accurate from *Optimal Savings VI: EGM with JAX*.

The primary source for the technical details discussed below is [Ma *et al.*, 2020].

Other references include [Deaton, 1991], [Den Haan, 2010], [Kuhn, 2013], [Rabault, 2002], [Reiter, 2009] and [Schechtman and Escudero, 1977].

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install quantecon jax
```

We'll also need the following imports:

```
import matplotlib.pyplot as plt
import numpy as np
import numba
from quantecon import MarkovChain
import jax
import jax.numpy as jnp
from typing import NamedTuple
```

69.2 The Household Problem

Let's write down the model and then discuss how to solve it.

69.2.1 Set-Up

A household chooses a state-contingent consumption plan $\{c_t\}_{t \geq 0}$ to maximize

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t u(c_t)$$

subject to

$$a_{t+1} = R(a_t - c_t) + Y_{t+1} \quad c_t \geq 0, \quad a_t \geq 0 \quad t = 0, 1, \ldots \tag{69.1}$$

Here

- $\beta \in (0, 1)$ is the discount factor
- a_t is asset holdings at time t, with borrowing constraint $a_t \geq 0$
- c_t is consumption
- Y_t is non-capital income (wages, unemployment compensation, etc.)
- $R := 1 + r$, where $r > 0$ is the interest rate on savings

The timing here is as follows:

1. At the start of period t, the household observes current asset holdings a_t.
2. The household chooses current consumption c_t.
3. Savings $s_t := a_t - c_t$ earns interest at rate r.
4. Labor income Y_{t+1} is realized and time shifts to $t + 1$.

Non-capital income Y_t is given by $Y_t = y(Z_t)$, where

- $\{Z_t\}$ is an exogenous state process, and
- y is a function taking values in $\mathbb{R}_+$.

We take $\{Z_t\}$ to be a finite state Markov chain taking values in Z with Markov matrix Π.

Note

In previous lectures we used the more standard household budget constraint $a_{t+1} + c_t \leq Ra_t + Y_t$.

This setup, which is pervasive in quantitative economics, was developed for discretization.

It means that the control variable is also the next period state a_{t+1}, which makes it straightforward to restrict assets to a finite grid.

But fixing the control to be the next period state forces us to include more information in the current state, which expands the size of the state space.

Moreover, aiming for discretization is not always a good idea, since it suffers heavily from the curse of dimensionality.

These ideas will become clearer in the *next lecture*.

We further assume that

1. $\beta R < 1$
2. u is smooth, strictly increasing and strictly concave with $\lim_{c \to 0} u'(c) = \infty$ and $\lim_{c \to \infty} u'(c) = 0$
3. $y(z) = \exp(z)$

The asset space is $\mathbb{R}_+$ and the state is the pair $(a, z) \in \mathsf{S} := \mathbb{R}_+ \times \mathsf{Z}$.

A **feasible consumption path** from $(a, z) \in \mathsf{S}$ is a consumption sequence $\{c_t\}$ such that $\{c_t\}$ and its induced asset path $\{a_t\}$ satisfy

1. $(a_0, z_0) = (a, z)$
2. the feasibility constraints in (69.1), and
3. adaptedness, which means that c_t is a function of random outcomes up to date t but not after.

The meaning of the third point is just that consumption at time t cannot be a function of outcomes are yet to be observed.

In fact, for this problem, consumption can be chosen optimally by taking it to be contingent only on the current state.

Optimality is defined below.

69.2.2 Value Function and Euler Equation

The **value function** $V \colon \mathsf{S} \to \mathbb{R}$ is defined by

$$V(a, z) := \max \mathbb{E} \left\{ \sum_{t=0}^{\infty} \beta^t u(c_t) \right\} \tag{69.2}$$

where the maximization is overall feasible consumption paths from (a, z).

An **optimal consumption path** from (a, z) is a feasible consumption path from (a, z) that maximizes (67.1).

To pin down such paths we can use a version of the Euler equation, which in the present setting is

$$u'(c_t) \geq \beta R \, \mathbb{E}_t u'(c_{t+1}) \tag{69.3}$$

with

$$c_t < a_t \implies u'(c_t) = \beta R \, \mathbb{E}_t u'(c_{t+1}) \tag{69.4}$$

When c_t hits the upper bound a_t, the strict inequality $u'(c_t) > \beta R \, \mathbb{E}_t u'(c_{t+1})$ can occur because c_t cannot increase sufficiently to attain equality.

The case $c_t = 0$ never arises along the optimal path because $u'(0) = \infty$.

69.2.3 Optimality Results

As shown in [Ma *et al.*, 2020],

1. For each $(a, z) \in \mathsf{S}$, a unique optimal consumption path from (a, z) exists
2. This path is the unique feasible path from (a, z) satisfying the Euler equations (69.3)-(69.4) and the transversality condition

$$\lim_{t \to \infty} \beta^t \, \mathbb{E} \left[u'(c_t) a_{t+1} \right] = 0 \tag{69.5}$$

Moreover, there exists an **optimal consumption policy** $\sigma^* \colon \mathsf{S} \to \mathbb{R}_+$ such that the path from (a, z) generated by

$$(a_0, z_0) = (a, z), \quad c_t = \sigma^*(a_t, Z_t) \quad \text{and} \quad a_{t+1} = R(a_t - c_t) + Y_{t+1}$$

satisfies both the Euler equations (69.3)-(69.4) and (69.5), and hence is the unique optimal path from (a, z).

Thus, to solve the optimization problem, we need to compute the policy σ^*.

69.3 Computation

We solve for the optimal consumption policy using time iteration and the endogenous grid method, which were previously discussed in

- *Optimal Savings IV: Time Iteration*
- *Optimal Savings V: The Endogenous Grid Method*

69.3.1 Solution Method

We rewrite (69.4) to make it a statement about functions rather than random variables:

$$(u' \circ \sigma)(a, z) = \beta R \sum_{z'} (u' \circ \sigma)[R(a - \sigma(a, z)) + y(z'), \, z'] \, \Pi(z, z') \tag{69.6}$$

Here

- $(u' \circ \sigma)(s) := u'(\sigma(s))$,
- primes indicate next period states (as well as derivatives), and
- σ is the unknown function.

The equality (69.6) holds at all interior choices, meaning $\sigma(a, z) < a$.

We aim to find a fixed point σ of (69.6).

To do so we use the EGM.

Below we use the relationships $a_t = c_t + s_t$ and $a_{t+1} = R s_t + Y_{t+1}$.

We begin with an exogenous savings grid $s_0 < s_1 < \cdots < s_m$ with $s_0 = 0$.

We fix a current guess of the policy function σ.

For each exogenous savings level s_i with $i \geq 1$ and current state z_j, we set

$$c_{ij} := (u')^{-1} \left[\beta R \sum_{z'} u'[\sigma(Rs_i + y(z'), z')] \, \Pi(z_j, z') \right] \tag{69.7}$$

The Euler equation holds here because $i \geq 1$ implies $s_i > 0$ and hence consumption is interior.

For the boundary case $s_0 = 0$ we set

$$c_{0j} := 0 \quad \text{for all j}$$

We then obtain a corresponding endogenous grid of current assets via

$$a_{ij} := c_{ij} + s_i.$$

Notice that, for each j, we have $a_{0j} = c_{0j} = 0$.

This anchors the interpolation at the correct value at the origin, since, without borrowing, consumption is zero when assets are zero.

Our next guess of the policy function, which we write as $K\sigma$, is the linear interpolation of the interpolation points

$$\{(a_{0j}, c_{0j}), \ldots, (a_{mj}, c_{mj})\}$$

for each j.

(The number of one-dimensional linear interpolations is equal to the size of Z.)

69.4 NumPy Implementation

In this section we'll code up a NumPy version of the code that aims only for clarity, rather than efficiency.

Once we have it working, we'll produce a JAX version that's far more efficient and check that we obtain the same results.

We use the CRRA utility specification

$$u(c) = \frac{c^{1-\gamma}}{1-\gamma}$$

69.4.1 Set Up

Here we build a class called `IFPNumPy` that stores the model primitives.

The exogenous state process $\{Z_t\}$ defaults to a two-state Markov chain with transition matrix Π.

```
class IFPNumPy(NamedTuple):
    R: float                  # Gross interest rate R = 1 + r
    β: float                  # Discount factor
    γ: float                  # Preference parameter
    Π: np.ndarray             # Markov matrix for exogenous shock
    z_grid: np.ndarray        # Markov state values for Z_t
    s: np.ndarray             # Exogenous savings grid
```

(continues on next page)

(continued from previous page)

```
def create_ifp(r=0.01,
               β=0.96,
               γ=1.5,
               Π=((0.6, 0.4),
                  (0.05, 0.95)),
               z_grid=(-10.0, np.log(2.0)),
               savings_grid_max=16,
               savings_grid_size=200):

    s = np.linspace(0, savings_grid_max, savings_grid_size)
    Π, z_grid = np.array(Π), np.array(z_grid)
    R = 1 + r
    assert R * β < 1, "Stability condition violated."
    return IFPNumPy(R, β, γ, Π, z_grid, s)
```

69.4.2 Solver

Here is the operator K that transforms current guess σ into next period guess $K\sigma$.

In practice, it takes in

- a guess of optimal consumption values c_{ij}, stored as `c_vec`
- and a corresponding set of endogenous grid points a^e_{ij}, stored as `a_vec`

These are converted into a consumption policy $a \mapsto \sigma(a, z_j)$ by linear interpolation of (a^e_{ij}, c_{ij}) over i for each j.

Since there are no shocks to integrate out in this version of the model, we can compute (69.7) directly by summing over the finite state space Z.

```
@numba.jit
def K_numpy(
        c_in: np.ndarray,   # Initial guess of σ on grid endogenous grid
        a_in: np.ndarray,   # Initial endogenous grid
        ifp_numpy: IFPNumPy
    ) -> np.ndarray:
    """
    The Euler equation operator for the IFP model using the
    Endogenous Grid Method.

    This operator implements one iteration of the EGM algorithm to
    update the consumption policy function.

    """
    R, β, γ, Π, z_grid, s = ifp_numpy
    n_a, n_z = len(s), len(z_grid)
    c_out = np.zeros_like(c_in)
    u_prime = lambda c: c**(-γ)
    u_prime_inv = lambda c: c**(-1/γ)
    y = lambda z: np.exp(z)

    for i in range(1, n_a):  # Start from 1 for positive savings levels
        for j in range(n_z):

            # Compute Σ_z' u'(σ(R s_i + y(z'), z')) Π[z_j, z']
```

(continues on next page)

(continued from previous page)

```
            expectation = 0.0
            for k in range(n_z):
                z_prime = z_grid[k]
                # Calculate next period assets
                next_a = R * s[i] + y(z_prime)
                # Interpolate to get σ(R s_i + y(z'), z')
                next_c = np.interp(next_a, a_in[:, k], c_in[:, k])
                # Weight by transition probability and add to the expectation
                expectation += u_prime(next_c) * Π[j, k]

            # Calculate updated c_{ij} values
            c_out[i, j] = u_prime_inv(β * R * expectation)

    a_out = c_out + s[:, None]
    return c_out, a_out
```

To solve the model we use a simple while loop.

```
def solve_model_numpy(
        ifp_numpy: IFPNumPy,
        c_init: np.ndarray,
        a_init: np.ndarray,
        tol: float = 1e-5,
        max_iter: int = 1_000
    ) -> np.ndarray:
    """
    Solve the model using time iteration with EGM.

    """
    c_in, a_in = c_init, a_init
    i = 0
    error = tol + 1

    while error > tol and i < max_iter:
        c_out, a_out = K_numpy(c_in, a_in, ifp_numpy)
        error = np.max(np.abs(c_out - c_in))
        i = i + 1
        c_in, a_in = c_out, a_out

    return c_out, a_out
```

Let's road test the EGM code.

```
ifp_numpy = create_ifp()
R, β, γ, Π, z_grid, s = ifp_numpy
# Initial conditions -- agent consumes everything
a_init = s[:, None] * np.ones(len(z_grid))
c_init = a_init
# Solve from these initial conditions
c_vec, a_vec = solve_model_numpy(
    ifp_numpy, c_init, a_init
)
```

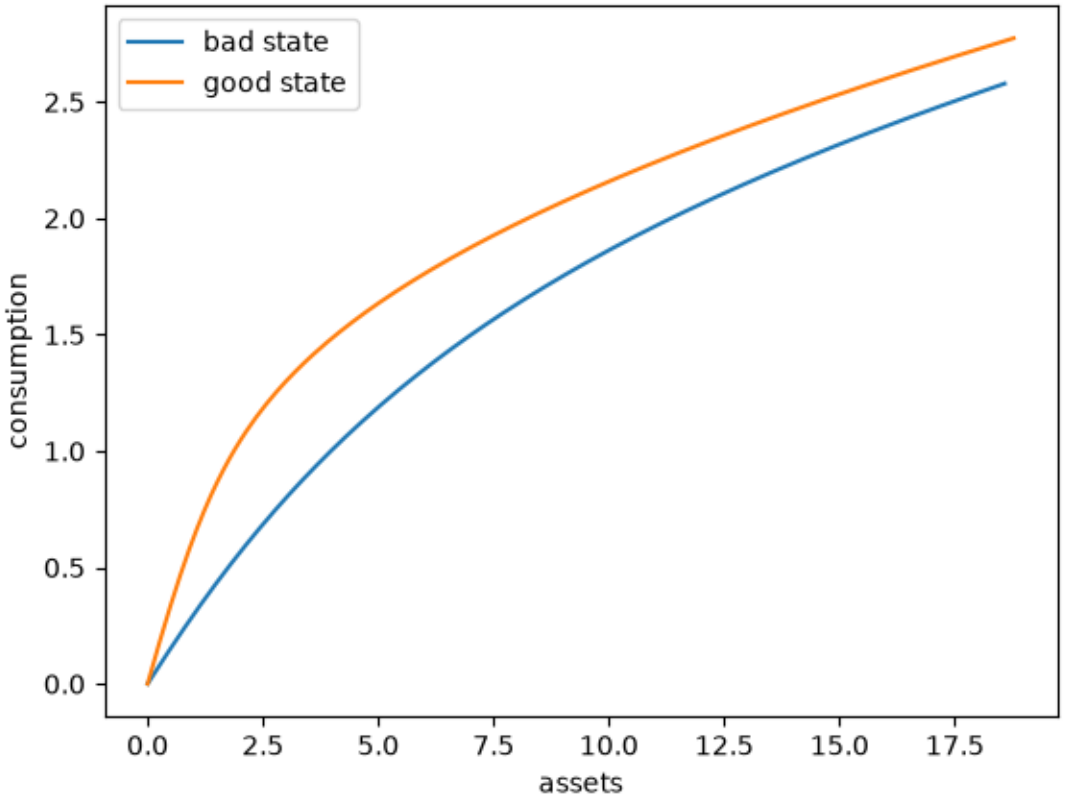
Here's a plot of the optimal consumption policy for each z state

```
fig, ax = plt.subplots()
```

(continues on next page)

(continued from previous page)

```
ax.plot(a_vec[:, 0], c_vec[:, 0], label='bad state')
ax.plot(a_vec[:, 1], c_vec[:, 1], label='good state')
ax.set(xlabel='assets', ylabel='consumption')
ax.legend()
plt.show()
```



69.5 JAX Implementation

Now we write a more efficient JAX version, which can run on a GPU.

69.5.1 Set Up

We start with a class called `IFP` that stores the model primitives.

```
class IFP(NamedTuple):
    R: float                  # Gross interest rate R = 1 + r
    β: float                  # Discount factor
    γ: float                  # Preference parameter
    Π: jnp.ndarray            # Markov matrix for exogenous shock
    z_grid: jnp.ndarray       # Markov state values for Z_t
    s: jnp.ndarray            # Exogenous savings grid


def create_ifp(r=0.01,
```

(continues on next page)

(continued from previous page)

```
                β=0.94,
                γ=1.5,
                Π=((0.6, 0.4),
                   (0.05, 0.95)),
                z_grid=(-10.0, jnp.log(2.0)),
                savings_grid_max=16,
                savings_grid_size=200):

    s = jnp.linspace(0, savings_grid_max, savings_grid_size)
    Π, z_grid = jnp.array(Π), jnp.array(z_grid)
    R = 1 + r
    assert R * β < 1, "Stability condition violated."
    return IFP(R, β, γ, Π, z_grid, s)
```

69.5.2 Solver

Here is the operator K that transforms current guess σ into next period guess $K\sigma$.

```
def K(
        c_in: jnp.ndarray,
        a_in: jnp.ndarray,
        ifp: IFP
    ) -> jnp.ndarray:
    """
    The Euler equation operator for the IFP model using the
    Endogenous Grid Method.

    This operator implements one iteration of the EGM algorithm to
    update the consumption policy function.

    """
    R, β, γ, Π, z_grid, s = ifp
    n_z = len(z_grid)
    z_indices = jnp.arange(n_z)
    u_prime = lambda c: c**(-γ)
    u_prime_inv = lambda c: c**(-1/γ)
    y = lambda z: jnp.exp(z)

    def compute_c(i, j):
        " Computes consumption for one (i, j) pair where i >= 1. "

        def compute_mu_k(k):
            " Given i, compute marginal utility u'(σ(R s_i + y(z_k), z_k)) "
            next_a = R * s[i] + y(z_grid[k])
            # Interpolate to get σ(R * s_i + y(z_k), z_k)
            next_c = jnp.interp(next_a, a_in[:, k], c_in[:, k])
            # Return u'(σ(R * s_i + y(z_k), z_k))
            return u_prime(next_c)

        # Compute marginal utility u'(σ(R * s_i + y(z_k), z_k)) for all k
        mu_values = jax.vmap(compute_mu_k)(z_indices)
        # Compute expectation Σ_k u'(σ(...)) * Π[j, k]
        expectation = jnp.sum(mu_values * Π[j, :])
        # Invert to get consumption c_{ij} at (s_i, z_j)
        return u_prime_inv(β * R * expectation)
```

(continues on next page)

(continued from previous page)

```
    # vmap over j for each i
    compute_c_i = jax.vmap(compute_c, in_axes=(None, 0))
    # vmap over i
    compute_c = jax.vmap(lambda i: compute_c_i(i, z_indices))
    # Compute consumption for i >= 1
    c_out_interior = compute_c(jnp.arange(1, len(s)))
    # For i = 0, set consumption to 0
    c_out_boundary = jnp.zeros((1, n_z))
    # Concatenate boundary and interior
    c_out = jnp.concatenate([c_out_boundary, c_out_interior], axis=0)
    # Compute endogenous asset grid a_{ij} = c_{ij} + s_i
    a_out = c_out + s[:, None]
    return c_out, a_out
```

Here's a jit-accelerated iterative routine to solve the model using this operator.

```
@jax.jit
def solve_model(
        ifp: IFP,
        c_init: jnp.ndarray,  # Initial guess of σ on grid endogenous grid
        a_init: jnp.ndarray,  # Initial endogenous grid
        tol: float = 1e-5,
        max_iter: int = 1000
    ) -> jnp.ndarray:
    """
    Solve the model using time iteration with EGM.

    """

    def condition(loop_state):
        c_in, a_in, i, error = loop_state
        return (error > tol) & (i < max_iter)

    def body(loop_state):
        c_in, a_in, i, error = loop_state
        c_out, a_out = K(c_in, a_in, ifp)
        error = jnp.max(jnp.abs(c_out - c_in))
        i += 1
        return c_out, a_out, i, error

    i, error = 0, tol + 1
    initial_state = (c_init, a_init, i, error)
    final_loop_state = jax.lax.while_loop(condition, body, initial_state)
    c_out, a_out, i, error = final_loop_state

    return c_out, a_out
```

69.5.3 Test run

Let's road test the EGM code.

```
ifp = create_ifp()
R, β, γ, Π, z_grid, s = ifp
# Set initial conditions where the agent consumes everything
a_init = s[:, None] * jnp.ones(len(z_grid))
c_init = a_init
# Solve starting from these initial conditions
c_vec_jax, a_vec_jax = solve_model(ifp, c_init, a_init)
```

To verify the correctness of our JAX implementation, let's compare it with the NumPy version we developed earlier.

```
# Compare the results
max_c_diff = np.max(np.abs(np.array(c_vec) - c_vec_jax))
max_ae_diff = np.max(np.abs(np.array(a_vec) - a_vec_jax))

print(f"Maximum difference in consumption policy: {max_c_diff:.2e}")
print(f"Maximum difference in asset grid:        {max_ae_diff:.2e}")
```

```
Maximum difference in consumption policy: 4.45e-01
Maximum difference in asset grid:        4.45e-01
```

These numbers confirm that we are computing essentially the same policy using the two approaches.

69.5.4 Timing

Now let's compare the execution time between NumPy and JAX implementations.

```
import time

# Set up initial conditions for NumPy version
s_np = np.array(s)
z_grid_np = np.array(z_grid)
a_init_np = s_np[:, None] * np.ones(len(z_grid_np))
c_init_np = a_init_np.copy()

# Set up initial conditions for JAX version
a_init_jx = s[:, None] * jnp.ones(len(z_grid))
c_init_jx = a_init_jx

# Time NumPy version
start = time.time()
c_vec_np, a_vec_np = solve_model_numpy(ifp_numpy, c_init_np, a_init_np)
numpy_time = time.time() - start

# Time JAX version (with compilation)
start = time.time()
c_vec_jx, a_vec_jx = solve_model(ifp, c_init_jx, a_init_jx)
c_vec_jx.block_until_ready()
jax_time_with_compile = time.time() - start

# Time JAX version (without compilation - second run)
start = time.time()
c_vec_jx, a_vec_jx = solve_model(ifp, c_init_jx, a_init_jx)
```

(continues on next page)

(continued from previous page)

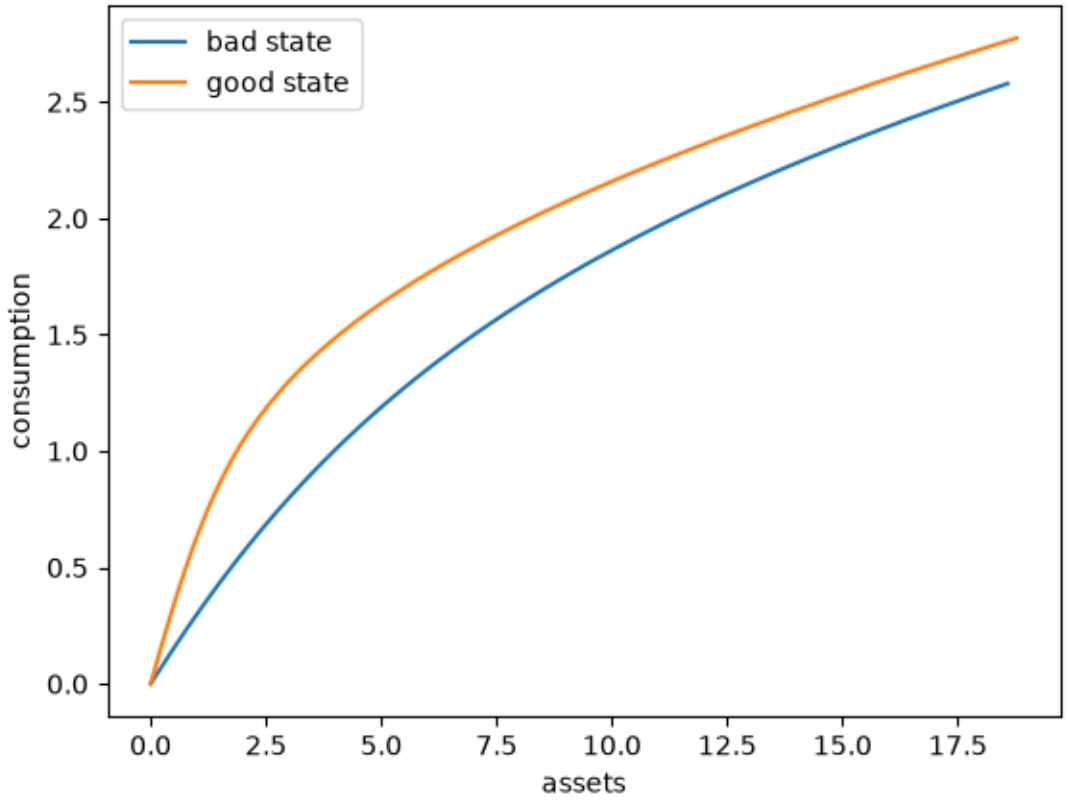
```
c_vec_jx.block_until_ready()
jax_time = time.time() - start

print(f"NumPy time:                 {numpy_time:.4f} seconds")
print(f"JAX time (with compile):    {jax_time_with_compile:.4f} seconds")
print(f"JAX time (without compile): {jax_time:.4f} seconds")
print(f"Speedup (NumPy/JAX):        {numpy_time/jax_time:.2f}x")
```

```
NumPy time:                 0.0209 seconds
JAX time (with compile):    0.0099 seconds
JAX time (without compile): 0.0097 seconds
Speedup (NumPy/JAX):        2.16x
```

The JAX implementation is faster due to JIT compilation and GPU/TPU acceleration (if available).

Here's a plot of the optimal policy for each z state

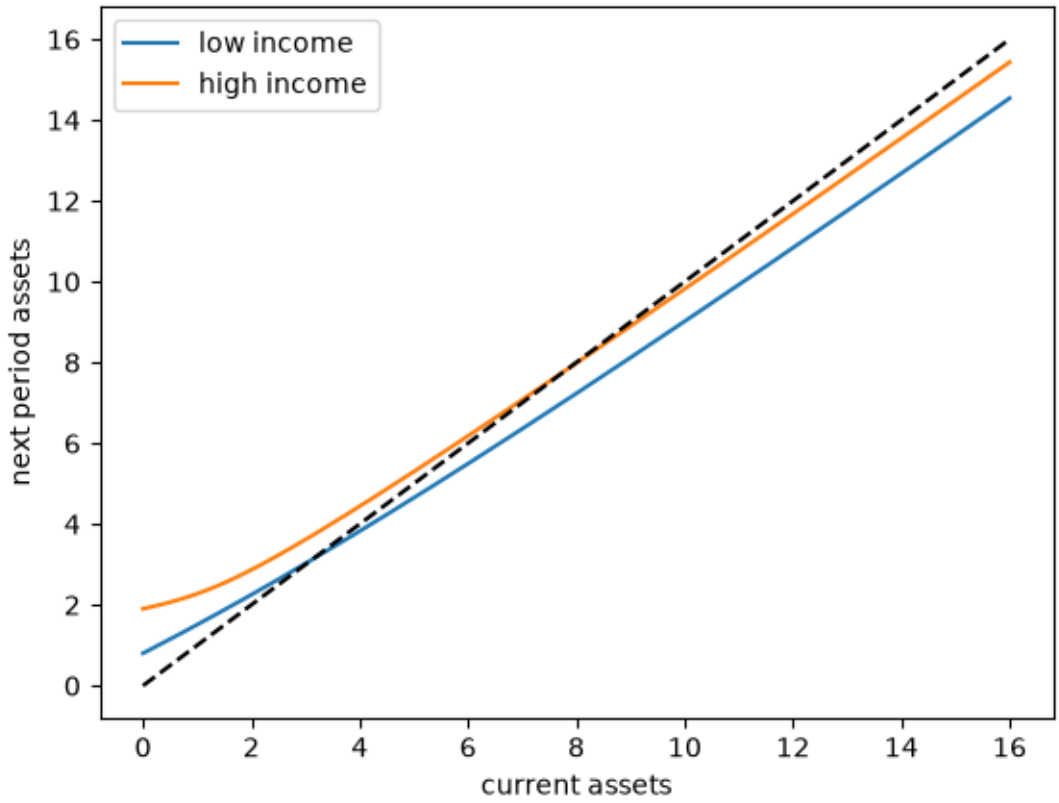
```
fig, ax = plt.subplots()
ax.plot(a_vec[:, 0], c_vec[:, 0], label='bad state')
ax.plot(a_vec[:, 1], c_vec[:, 1], label='good state')
ax.set(xlabel='assets', ylabel='consumption')
ax.legend()
plt.show()
```



69.5.5 Dynamics

To begin to understand the long run asset levels held by households under the default parameters, let's look at the 45 degree diagram showing the law of motion for assets under the optimal consumption policy.

```
fig, ax = plt.subplots()

y = lambda z: jnp.exp(z)

def y_bar(k):
    """
    Taking z = z_grid[k], compute

            E_z Y' = Σ_{z'} y(z') Π[z, z']

    This is the expectation of Y_{t+1} given Z_t = z.
    """
    # Compute y(z') for all z'
    y_values = jax.vmap(y)(z_grid)
    # Weight by transition probabilities and sum
    return jnp.sum(y_values * Π[k, :])

for k, label in zip((0, 1), ('low income', 'high income')):
    # Interpolate consumption policy on the savings grid
    c_on_grid = jnp.interp(s, a_vec[:, k], c_vec[:, k])
    ax.plot(s, R * (s - c_on_grid) + y_bar(k) , label=label)

ax.plot(s, s, 'k--')
ax.set(xlabel='current assets', ylabel='next period assets')

ax.legend()
plt.show()
```



The unbroken lines show the update function for assets at each z, which is

$$a \mapsto R(a - \sigma^*(a, z)) + \bar{y}(z)$$

where

$$\bar{y}(z) := \sum_{z'} y(z')\Pi(z, z')$$

is the expected labor income conditional on current state z.

The dashed line is the 45 degree line.

The figure suggests that, on average, the dynamics will be stable — assets do not diverge even in the highest state.

This turns out to be true: there is a unique stationary distribution of assets.

- For details see [Ma *et al.*, 2020]

This stationary distribution represents the long run dispersion of assets across households when households have idiosyncratic shocks.

69.5.6 A Sanity Check

One way to check our results is to

- set labor income to zero in each state and
- set the gross interest rate R to unity.

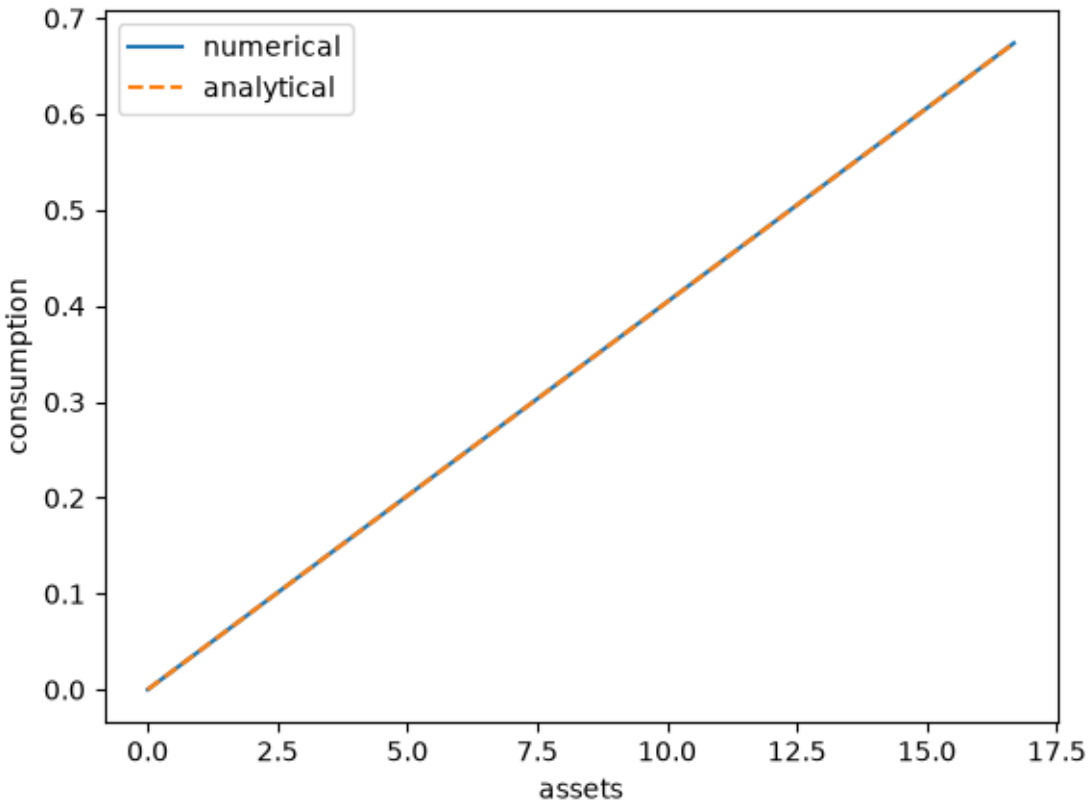
In this case, our income fluctuation problem is just a CRRA cake eating problem.

Then the value function and optimal consumption policy are given by

```
def c_star(x, β, γ):
    return (1 - β ** (1/γ)) * x


def v_star(x, β, γ):
    return (1 - β**(1 / γ))**(-γ) * (x**(1-γ) / (1-γ))
```

Let's see if we match up:

```
ifp_cake_eating = create_ifp(r=0.0, z_grid=(-jnp.inf, -jnp.inf))
R, β, γ, Π, z_grid, s = ifp_cake_eating
a_init = s[:, None] * jnp.ones(len(z_grid))
c_init = a_init
c_vec, a_vec = solve_model(ifp_cake_eating, c_init, a_init)

fig, ax = plt.subplots()
ax.plot(a_vec[:, 0], c_vec[:, 0], label='numerical')
ax.plot(a_vec[:, 0],
        c_star(a_vec[:, 0], ifp_cake_eating.β, ifp_cake_eating.γ),
        '--', label='analytical')
ax.set(xlabel='assets', ylabel='consumption')
ax.legend()
plt.show()
```



This looks pretty good.

69.6 Simulation

Let's return to the default model and study the stationary distribution of assets.

Our plan is to run a large number of households forward for T periods and then histogram the cross-sectional distribution of assets.

Set `num_households=50_000, T=500`.

First we write a function to run a single household forward in time and record the final value of assets.

The function takes a solution pair `c_vec` and `a_vec`, understanding them as representing an optimal policy associated with a given model `ifp`

```
@jax.jit
def simulate_household(
        key, a_0, z_idx_0, c_vec, a_vec, ifp, T
    ):
    """
    Simulates a single household for T periods to approximate the stationary
    distribution of assets.

    - key is the state of the random number generator
    - ifp is an instance of IFP
    - c_vec, a_vec are the optimal consumption policy, endogenous grid for ifp
```

(continues on next page)

(continued from previous page)

```
    """
    R, β, γ, Π, z_grid, s = ifp
    n_z = len(z_grid)

    y = lambda z: jnp.exp(z)
    σ = lambda a, z_idx: jnp.interp(a, a_vec[:, z_idx], c_vec[:, z_idx])

    # Simulate forward T periods
    def update(t, state):
        a, z_idx = state
        # Draw next shock z' from Π[z, z']
        current_key = jax.random.fold_in(key, t)
        z_next_idx = jax.random.choice(current_key, n_z, p=Π[z_idx]).astype(jnp.int32)
        z_next = z_grid[z_next_idx]
        # Update assets: a' = R * (a - c) + Y'
        a_next = R * (a - σ(a, z_idx)) + y(z_next)
        # Return updated state
        return a_next, z_next_idx

    initial_state = a_0, z_idx_0
    final_state = jax.lax.fori_loop(0, T, update, initial_state)
    a_final, _ = final_state
    return a_final
```

Now we write a function to simulate many households in parallel.

```
def compute_asset_stationary(
        c_vec, a_vec, ifp, num_households=50_000, T=500, seed=1234
    ):
    """
    Simulates num_households households for T periods to approximate
    the stationary distribution of assets.

    Returns the final cross-section of asset holdings.

    - ifp is an instance of IFP
    - c_vec, a_vec are the optimal consumption policy and endogenous grid.

    """
    R, β, γ, Π, z_grid, s = ifp
    n_z = len(z_grid)

    # Create interpolation function for consumption policy
    # Interpolate on the endogenous grid
    σ = lambda a, z_idx: jnp.interp(a, a_vec[:, z_idx], c_vec[:, z_idx])

    # Start with assets = savings_grid_max / 2
    a_0_vector = jnp.full(num_households, s[-1] / 2)
    # Initialize the exogenous state of each household
    z_idx_0_vector = jnp.zeros(num_households).astype(jnp.int32)

    # Vectorize over many households
    key = jax.random.PRNGKey(seed)
    keys = jax.random.split(key, num_households)
    # Vectorize simulate_household in (key, a_0, z_idx_0)
    sim_all_households = jax.vmap(
        simulate_household, in_axes=(0, 0, 0, None, None, None, None)
```

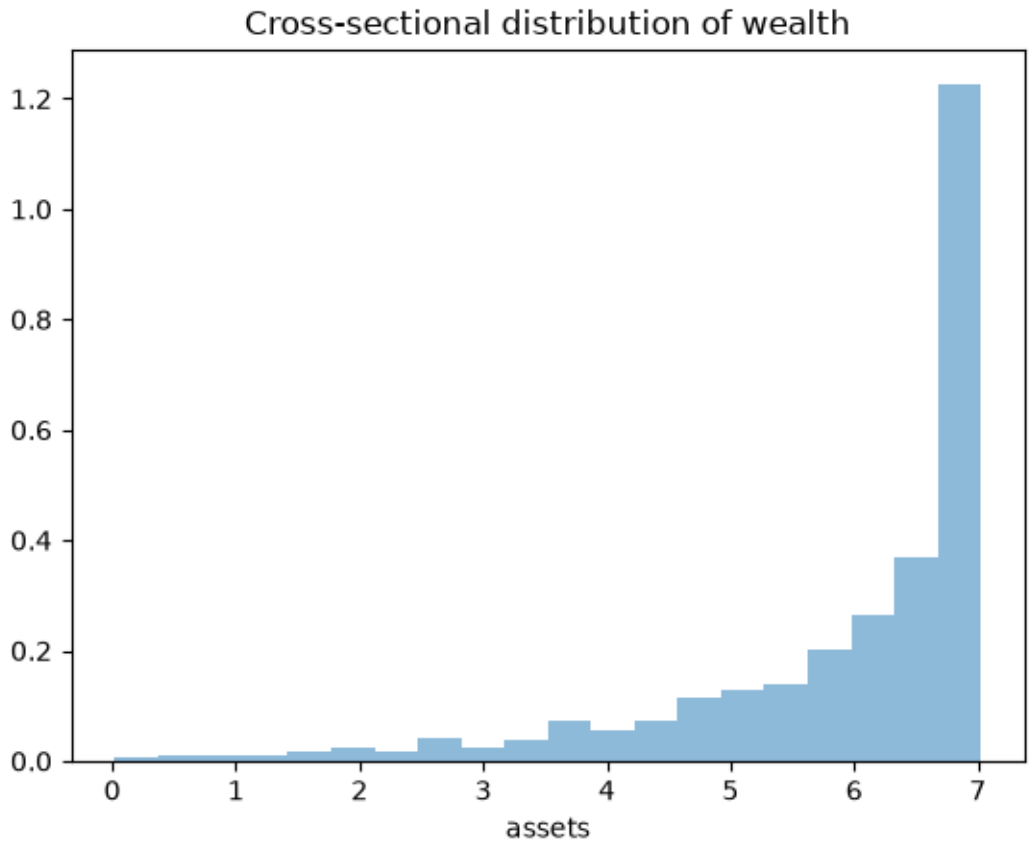
(continues on next page)

(continued from previous page)

```
    )
    assets = sim_all_households(keys, a_0_vector, z_idx_0_vector, c_vec, a_vec, ifp,␣
↪T)

    return np.array(assets)
```

Now we call the function, generate the asset distribution and histogram it:

```
ifp = create_ifp()
R, β, γ, Π, z_grid, s = ifp
a_init = s[:, None] * jnp.ones(len(z_grid))
c_init = a_init
c_vec, a_vec = solve_model(ifp, c_init, a_init)
assets = compute_asset_stationary(c_vec, a_vec, ifp)

fig, ax = plt.subplots()
ax.hist(assets, bins=20, alpha=0.5, density=True)
ax.set(xlabel='assets', title="Cross-sectional distribution of wealth")
plt.show()
```



The wealth distribution looks very different to a typical wealth distribution in the data.

For one thing it is left-skewed rather than right-skewed.

In fact there is essentially no right-hand tail, even though real-world wealth distributions have long right-hand tails.

We'll do our best to fix these issues in the next few lectures.

CHAPTER

SEVENTY

THE INCOME FLUCTUATION PROBLEM IV: TRANSIENT INCOME SHOCKS

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

Contents

- *The Income Fluctuation Problem IV: Transient Income Shocks*
  - *Overview*
  - *The Household Problem*
  - *NumPy Implementation*
  - *JAX Implementation*
  - *Simulation*
  - *Wealth Inequality*
  - *Exercises*

70.1 Overview

In this lecture we continue extend the IFP from *The Income Fluctuation Problem III: The Endogenous Grid Method* by adding transient shocks to the income process.

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install quantecon jax
```

We'll also need the following imports:

```
import matplotlib.pyplot as plt
import numpy as np
import numba
from quantecon import MarkovChain
import jax
import jax.numpy as jnp
from typing import NamedTuple
```

70.2 The Household Problem

We briefly outline the model and then discuss how to solve it.

Readers seeking a more extensive discussion of the model and the EGM solution method can review *The Income Fluctuation Problem III: The Endogenous Grid Method*.

70.2.1 Set-Up

A household chooses a state-contingent consumption plan $\{c_t\}_{t \geq 0}$ to maximize

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t u(c_t)$$

subject to

$$a_{t+1} = R(a_t - c_t) + Y_{t+1} \quad c_t \geq 0, \quad a_t \geq 0 \quad t = 0, 1, \ldots \tag{70.1}$$

The definitions of symbols and the timing are the same as in *The Income Fluctuation Problem III: The Endogenous Grid Method*.

Now, non-capital income Y_t is given by $Y_t = y(Z_t, \eta_t)$, where

- $\{Z_t\}$ is an exogenous state process (persistent component),
- $\{\eta_t\}$ is an IID shock process, and
- y is a function taking values in $\mathbb{R}_+$.

Throughout this lecture, we assume that $\eta_t \sim N(0, 1)$.

We again take $\{Z_t\}$ to be a finite state Markov chain taking values in Z with Markov matrix Π.

The shock process $\{\eta_t\}$ is independent of $\{Z_t\}$ and represents transient income fluctuations.

In addition to previous assumptions, we suppose that $y(z, \eta) = \exp(a_y \eta + z b_y)$ where a_y, b_y are positive constants

The asset space and state space are unchanged, as is the definition of an optimal path.

The functional Euler equation has the form

$$(u' \circ \sigma)(a, z) = \beta R \sum_{z'} \int (u' \circ \sigma)[R(a - \sigma(a, z)) + y(z', \eta'),\ z'] \phi(\eta') d\eta' \, \Pi(z, z') \tag{70.2}$$

Here

- $(u' \circ \sigma)(s) := u'(\sigma(s))$,
- primes indicate next period states (as well as derivatives),
- ϕ is the density of the shock η_t (standard normal), and

- σ is the unknown function.

The equality (70.2) holds at all interior choices, meaning $\sigma(a, z) < a$.

We aim to find a fixed point σ of (70.2).

To do so we use the EGM.

Below we use the relationships $a_t = c_t + s_t$ and $a_{t+1} = Rs_t + Y_{t+1}$.

We begin with an exogenous savings grid $s_0 < s_1 < \cdots < s_m$ with $s_0 = 0$.

We fix a current guess of the policy function σ.

For each exogenous savings level s_i with $i \geq 1$ and current state z_j, we set

$$c_{ij} := (u')^{-1} \left[\beta R \sum_{z'} \int u'[\sigma(Rs_i + y(z', \eta'), z')] \phi(\eta') d\eta' \, \Pi(z_j, z') \right] \tag{70.3}$$

The Euler equation holds here because $i \geq 1$ implies $s_i > 0$ and hence consumption is interior.

For the boundary case $s_0 = 0$ we set

$$c_{0j} := 0 \quad \text{for all j}$$

We then obtain a corresponding endogenous grid of current assets via

$$a_{ij} := c_{ij} + s_i.$$

Our next guess of the policy function, which we write as $K\sigma$, is the linear interpolation of the interpolation points

$$\{(a_{0j}, c_{0j}), \ldots, (a_{mj}, c_{mj})\}$$

for each j.

70.3 NumPy Implementation

In this section we'll code up a NumPy version of the code that aims only for clarity, rather than efficiency.

Once we have it working, we'll produce a JAX version that's far more efficient and check that we obtain the same results.

We use the CRRA utility specification

$$u(c) = \frac{c^{1-\gamma}}{1-\gamma}$$

70.3.1 Set Up

Here we build a class called `IFPNumPy` that stores the model primitives.

The exogenous state process $\{Z_t\}$ defaults to a two-state Markov chain with transition matrix Π.

```
class IFPNumPy(NamedTuple):
    R: float                  # Gross interest rate R = 1 + r
    β: float                  # Discount factor
    γ: float                  # Preference parameter
    Π: np.ndarray             # Markov matrix for exogenous shock
```

(continues on next page)

(continued from previous page)

```
    z_grid: np.ndarray        # Markov state values for Z_t
    s: np.ndarray             # Exogenous savings grid
    a_y: float                # Scale parameter for Y_t
    b_y: float                # Additive parameter for Y_t
    η_draws: np.ndarray       # Draws of innovation η for MC


def create_ifp(r=0.01,
               β=0.96,
               γ=1.5,
               Π=((0.6, 0.4),
                  (0.05, 0.95)),
               z_grid=(-10.0, np.log(2.0)),
               savings_grid_max=16,
               savings_grid_size=50,
               a_y=0.2,
               b_y=0.5,
               shock_draw_size=100,
               seed=1234):

    np.random.seed(seed)
    s = np.linspace(0, savings_grid_max, savings_grid_size)
    Π, z_grid = np.array(Π), np.array(z_grid)
    R = 1 + r
    η_draws = np.random.randn(shock_draw_size)
    assert R * β < 1, "Stability condition violated."
    return IFPNumPy(R, β, γ, Π, z_grid, s, a_y, b_y, η_draws)
```

70.3.2 Solver

Here is the operator K that transforms current guess σ into next period guess $K\sigma$.

In practice, it takes in

- a guess of optimal consumption values c_{ij}, stored as `c_vec`
- and a corresponding set of endogenous grid points a^e_{ij}, stored as `a_vec`

These are converted into a consumption policy $a \mapsto \sigma(a, z_j)$ by linear interpolation of (a^e_{ij}, c_{ij}) over i for each j.

When we compute consumption in (70.3), we will use Monte Carlo over η', so that the expression becomes

$$c_{ij} := (u')^{-1} \left[\beta R \sum_{z'} \frac{1}{m} \sum_{\ell=1}^{m} u'[\sigma(Rs_i + y(z', \eta_\ell), z')] \, \Pi(z_j, z') \right] \tag{70.4}$$

with each η_ℓ being a standard normal draw.

```
@numba.jit
def K_numpy(
        c_in: np.ndarray,    # Initial guess of σ on grid endogenous grid
        a_in: np.ndarray,    # Initial endogenous grid
        ifp_numpy: IFPNumPy
    ) -> np.ndarray:
    """
    The Euler equation operator for the IFP model using the
    Endogenous Grid Method.
```

(continues on next page)

(continued from previous page)

```
    This operator implements one iteration of the EGM algorithm to
    update the consumption policy function.

    """
    R, β, γ, Π, z_grid, s, a_y, b_y, η_draws = ifp_numpy
    n_a = len(s)
    n_z = len(z_grid)

    # Utility functions
    def u_prime(c):
        return c**(-γ)

    def u_prime_inv(c):
        return c**(-1/γ)

    def y(z, η):
        return np.exp(a_y * η + z * b_y)

    c_out = np.zeros_like(c_in)

    for i in range(1, n_a):  # Start from 1 for positive savings levels
        for j in range(n_z):

            # Compute Σ_z' ∫ u'(σ(R s_i + y(z', η'), z')) ψ(η') dη' Π[z_j, z']
            expectation = 0.0
            for k in range(n_z):
                z_prime = z_grid[k]
                # Integrate over η draws (Monte Carlo)
                inner_sum = 0.0
                for η in η_draws:
                    # Calculate next period assets
                    next_a = R * s[i] + y(z_prime, η)
                    # Interpolate to get σ(R s_i + y(z', η), z')
                    next_c = np.interp(next_a, a_in[:, k], c_in[:, k])
                    # Add to the inner sum
                    inner_sum += u_prime(next_c)
                # Average over η draws to approximate the integral
                # ∫ u'(σ(R s_i + y(z', η'), z')) ψ(η') dη' when z' = z_grid[k]
                inner_mean_k = (inner_sum / len(η_draws))
                # Weight by transition probability and add to the expectation
                expectation += inner_mean_k * Π[j, k]

            # Calculate updated c_{ij} values
            c_out[i, j] = u_prime_inv(β * R * expectation)

    a_out = c_out + s[:, None]

    return c_out, a_out
```

To solve the model we use a simple while loop.

```
def solve_model_numpy(
        ifp_numpy: IFPNumPy,
        c_init: np.ndarray,
        a_init: np.ndarray,
        tol: float = 1e-5,
```

(continues on next page)

(continued from previous page)

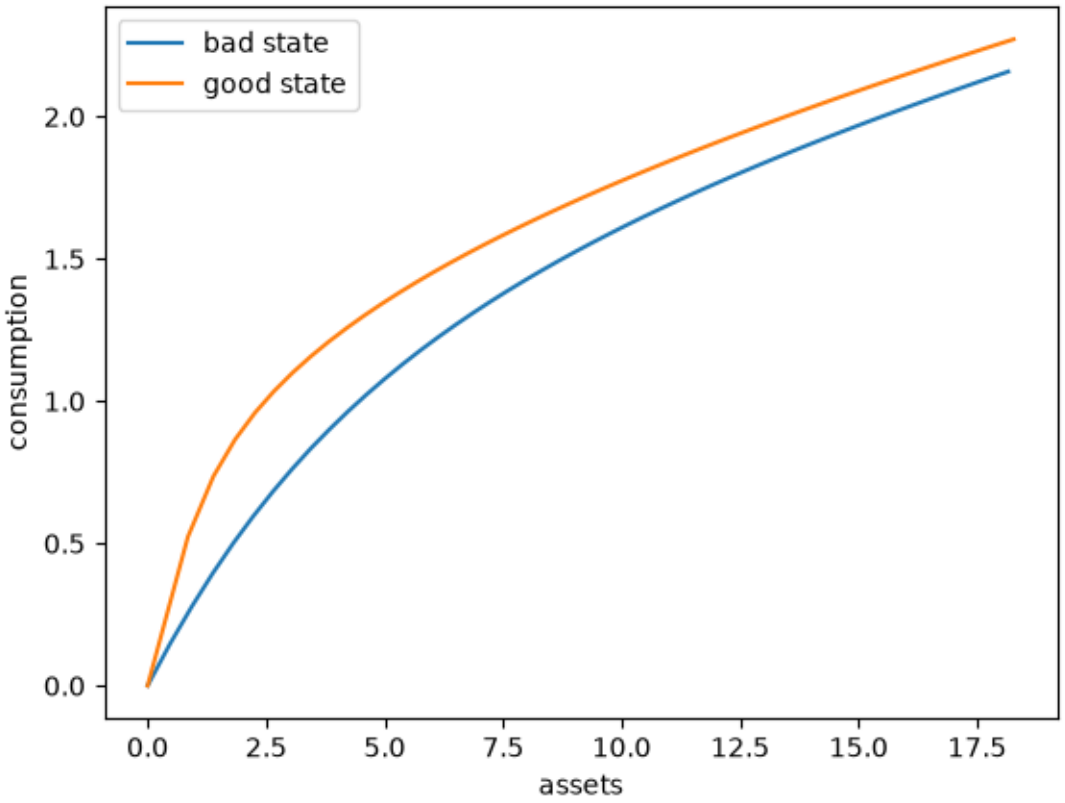
```
        max_iter: int = 1_000
    ) -> np.ndarray:
    """
    Solve the model using time iteration with EGM.

    """
    c_in, a_in = c_init, a_init
    i = 0
    error = tol + 1

    while error > tol and i < max_iter:
        c_out, a_out = K_numpy(c_in, a_in, ifp_numpy)
        error = np.max(np.abs(c_out - c_in))
        i = i + 1
        c_in, a_in = c_out, a_out

    return c_out, a_out
```

Let's road test the EGM code.

```
ifp_numpy = create_ifp()
R, β, γ, Π, z_grid, s, a_y, b_y, η_draws = ifp_numpy
# Initial conditions -- agent consumes everything
a_init = s[:, None] * np.ones(len(z_grid))
c_init = a_init
# Solve from these initial conditions
c_vec, a_vec = solve_model_numpy(
    ifp_numpy, c_init, a_init
)
```

Here's a plot of the optimal consumption policy for each z state

```
fig, ax = plt.subplots()

ax.plot(a_vec[:, 0], c_vec[:, 0], label='bad state')
ax.plot(a_vec[:, 1], c_vec[:, 1], label='good state')
ax.set(xlabel='assets', ylabel='consumption')
ax.legend()
plt.show()
```



70.4 JAX Implementation

Now we write a more efficient JAX version, which can run on a GPU.

70.4.1 Set Up

We start with a class called `IFP` that stores the model primitives.

```
class IFP(NamedTuple):
    R: float                  # Gross interest rate R = 1 + r
    β: float                  # Discount factor
    γ: float                  # Preference parameter
    Π: jnp.ndarray            # Markov matrix for exogenous shock
    z_grid: jnp.ndarray       # Markov state values for Z_t
    s: jnp.ndarray            # Exogenous savings grid
    a_y: float                # Scale parameter for Y_t
    b_y: float                # Additive parameter for Y_t
    η_draws: jnp.ndarray      # Draws of innovation η for MC


def create_ifp(r=0.01,
               β=0.94,
               γ=1.5,
               Π=((0.6, 0.4),
                  (0.05, 0.95)),
```

(continues on next page)

(continued from previous page)

```
                z_grid=(-10.0, jnp.log(2.0)),
                savings_grid_max=16,
                savings_grid_size=50,
                a_y=0.2,
                b_y=0.5,
                shock_draw_size=100,
                seed=1234):

    key = jax.random.PRNGKey(seed)
    s = jnp.linspace(0, savings_grid_max, savings_grid_size)
    Π, z_grid = jnp.array(Π), jnp.array(z_grid)
    R = 1 + r
    η_draws = jax.random.normal(key, (shock_draw_size,))
    assert R * β < 1, "Stability condition violated."
    return IFP(R, β, γ, Π, z_grid, s, a_y, b_y, η_draws)
```

70.4.2 Solver

Here is the operator K that transforms current guess σ into next period guess $K\sigma$.

```
def K(
        c_in: jnp.ndarray,
        a_in: jnp.ndarray,
        ifp: IFP
    ) -> jnp.ndarray:
    """
    The Euler equation operator for the IFP model using the
    Endogenous Grid Method.

    This operator implements one iteration of the EGM algorithm to
    update the consumption policy function.

    """
    R, β, γ, Π, z_grid, s, a_y, b_y, η_draws = ifp
    n_a = len(s)
    n_z = len(z_grid)

    # Utility functions
    def u_prime(c):
        return c**(-γ)

    def u_prime_inv(c):
        return c**(-1/γ)

    def y(z, η):
        return jnp.exp(a_y * η + z * b_y)

    def compute_c(i, j):
        " Compute c_ij when i >= 1 (interior choice). "

        def expected_mu(k):
            " Approximate ∫ u'(σ(R s_i + y(z_k, η'), z_k)) ψ(η') dη' "

            def compute_mu_at_eta(η):
                " Compute u'(σ(R * s_i + y(z_k, η), z_k)) "
```

(continues on next page)

(continued from previous page)

```
                next_a = R * s[i] + y(z_grid[k], η)
                # Interpolate to get σ(R * s_i + y(z_k, η), z_k)
                next_c = jnp.interp(next_a, a_in[:, k], c_in[:, k])
                # Return u'(σ(R * s_i + y(z_k, η), z_k))
                return u_prime(next_c)

            # Average over η draws to approximate the inner integral
            # ∫ u'(σ(R s_i + y(z_k, η'), z_k)) φ(η') dη'
            all_draws = jax.vmap(compute_mu_at_eta)(η_draws)
            return jnp.mean(all_draws)

        # Compute expectation: Σ_k [∫ u'(σ(...)) φ(η) dη] * Π[j, k]
        expectations = jax.vmap(expected_mu)(jnp.arange(n_z))
        expectation = jnp.sum(expectations * Π[j, :])
        # Invert to get consumption c_ij at (s_i, z_j)
        return u_prime_inv(β * R * expectation)

    # Set up index grids for vmap computation of all c_{ij}
    i_grid = jnp.arange(1, n_a)
    j_grid = jnp.arange(n_z)

    # vmap over j for each i
    compute_c_i = jax.vmap(compute_c, in_axes=(None, 0))
    # vmap over i
    compute_c = jax.vmap(lambda i: compute_c_i(i, j_grid))
    # Compute consumption for i >= 1
    c_out_interior = compute_c(i_grid)  # Shape: (n_a-1, n_z)
    # For i = 0, set consumption to 0
    c_out_boundary = jnp.zeros((1, n_z))

    # Concatenate boundary and interior
    c_out = jnp.concatenate([c_out_boundary, c_out_interior], axis=0)

    # Compute endogenous asset grid: a^e_{ij} = c_{ij} + s_i
    a_out = c_out + s[:, None]

    return c_out, a_out
```

Here's a jit-accelerated iterative routine to solve the model using this operator.

```
@jax.jit
def solve_model(
        ifp: IFP,
        c_init: jnp.ndarray,  # Initial guess of σ on grid endogenous grid
        a_init: jnp.ndarray,  # Initial endogenous grid
        tol: float = 1e-5,
        max_iter: int = 1000
    ) -> jnp.ndarray:
    """
    Solve the model using time iteration with EGM.

    """

    def condition(loop_state):
        c_in, a_in, i, error = loop_state
        return (error > tol) & (i < max_iter)
```

(continues on next page)

(continued from previous page)

```
    def body(loop_state):
        c_in, a_in, i, error = loop_state
        c_out, a_out = K(c_in, a_in, ifp)
        error = jnp.max(jnp.abs(c_out - c_in))
        i += 1
        return c_out, a_out, i, error

    i, error = 0, tol + 1
    initial_state = (c_init, a_init, i, error)
    final_loop_state = jax.lax.while_loop(condition, body, initial_state)
    c_out, a_out, i, error = final_loop_state

    return c_out, a_out
```

70.4.3 Test run

Let's road test the EGM code.

```
ifp = create_ifp()
R, β, γ, Π, z_grid, s, a_y, b_y, η_draws = ifp
# Set initial conditions where the agent consumes everything
a_init = s[:, None] * jnp.ones(len(z_grid))
c_init = a_init
# Solve starting from these initial conditions
c_vec_jax, a_vec_jax = solve_model(ifp, c_init, a_init)
```

To verify the correctness of our JAX implementation, let's compare it with the NumPy version we developed earlier.

```
# Compare the results
max_c_diff = np.max(np.abs(np.array(c_vec) - c_vec_jax))
max_ae_diff = np.max(np.abs(np.array(a_vec) - a_vec_jax))

print(f"Maximum difference in consumption policy: {max_c_diff:.2e}")
print(f"Maximum difference in asset grid:        {max_ae_diff:.2e}")
```

```
Maximum difference in consumption policy: 3.94e-01
Maximum difference in asset grid:        3.94e-01
```

These numbers confirm that we are computing essentially the same policy using the two approaches.

(Remaining differences are mainly due to different Monte Carlo integration outcomes over relatively small samples.)

70.4.4 Timing

Now let's compare the execution time between NumPy and JAX implementations.

```
import time

# Set up initial conditions for NumPy version
s_np = np.array(s)
z_grid_np = np.array(z_grid)
a_init_np = s_np[:, None] * np.ones(len(z_grid_np))
c_init_np = a_init_np.copy()
```

(continues on next page)

(continued from previous page)

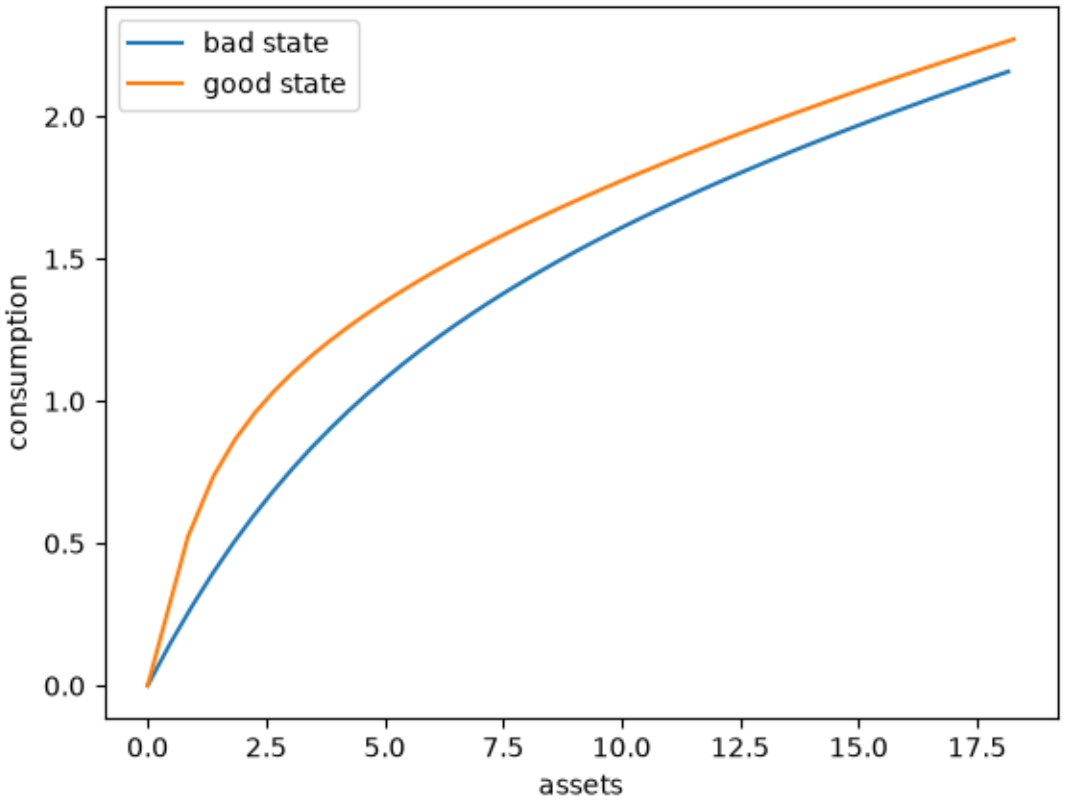
```
# Set up initial conditions for JAX version
a_init_jx = s[:, None] * jnp.ones(len(z_grid))
c_init_jx = a_init_jx

# Time NumPy version
start = time.time()
c_vec_np, a_vec_np = solve_model_numpy(ifp_numpy, c_init_np, a_init_np)
numpy_time = time.time() - start

# Time JAX version (with compilation)
start = time.time()
c_vec_jx, a_vec_jx = solve_model(ifp, c_init_jx, a_init_jx)
c_vec_jx.block_until_ready()
jax_time_with_compile = time.time() - start

# Time JAX version (without compilation - second run)
start = time.time()
c_vec_jx, a_vec_jx = solve_model(ifp, c_init_jx, a_init_jx)
c_vec_jx.block_until_ready()
jax_time = time.time() - start

print(f"NumPy time:                 {numpy_time:.4f} seconds")
print(f"JAX time (with compile):    {jax_time_with_compile:.4f} seconds")
print(f"JAX time (without compile): {jax_time:.4f} seconds")
print(f"Speedup (NumPy/JAX):        {numpy_time/jax_time:.2f}x")
```

```
NumPy time:                 0.3813 seconds
JAX time (with compile):    0.0087 seconds
JAX time (without compile): 0.0082 seconds
Speedup (NumPy/JAX):        46.33x
```

The JAX implementation is significantly faster due to JIT compilation and GPU/TPU acceleration (if available).

Here's a plot of the optimal policy for each z state

```
fig, ax = plt.subplots()
ax.plot(a_vec[:, 0], c_vec[:, 0], label='bad state')
ax.plot(a_vec[:, 1], c_vec[:, 1], label='good state')
ax.set(xlabel='assets', ylabel='consumption')
ax.legend()
plt.show()
```



70.4.5 Dynamics

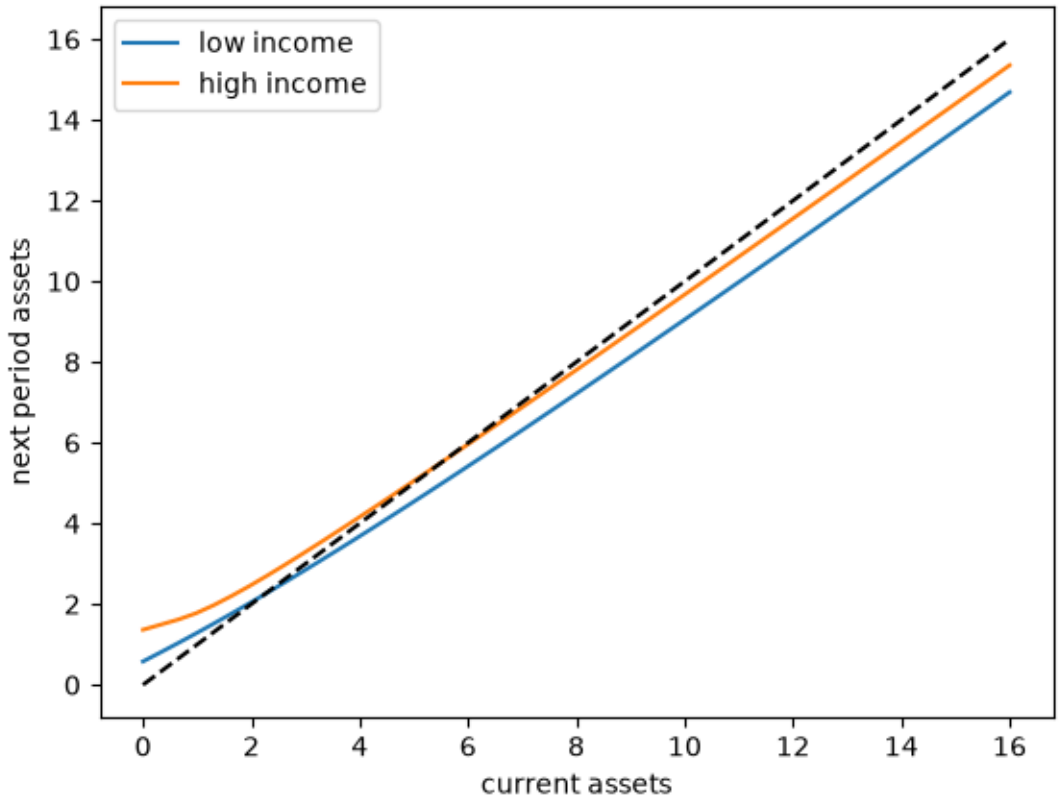
To begin to understand the long run asset levels held by households under the default parameters, let's look at the 45 degree diagram showing the law of motion for assets under the optimal consumption policy.

```
fig, ax = plt.subplots()

def y(z, η):
    return jnp.exp(a_y * η + z * b_y)

def y_bar(k):
    """
    Taking z = z_grid[k], compute an approximation to

            E_z Y' = Σ_{z'} ∫ y(z', η') ψ(η') dη' Π[z, z']

    This is the expectation of Y_{t+1} given Z_t = z.
    """
    # Approximate ∫ y(z', η') ψ(η') dη' at given z'
    def mean_y_at_z(z_prime):
        return jnp.mean(y(z_prime, η_draws))
    # Evaluate this integral across all z'
    y_means = jax.vmap(mean_y_at_z)(z_grid)
    # Weight by transition probabilities and sum
    return jnp.sum(y_means * Π[k, :])

for k, label in zip((0, 1), ('low income', 'high income')):
```

(continues on next page)

(continued from previous page)

```
    # Interpolate consumption policy on the savings grid
    c_on_grid = jnp.interp(s, a_vec[:, k], c_vec[:, k])
    ax.plot(s, R * (s - c_on_grid) + y_bar(k) , label=label)

ax.plot(s, s, 'k--')
ax.set(xlabel='current assets', ylabel='next period assets')

ax.legend()
plt.show()
```



The unbroken lines show the update function for assets at each z, which is

$$a \mapsto R(a - \sigma^*(a, z)) + \bar{y}(z)$$

where

$$\bar{y}(z) := \sum_{z'} \frac{1}{m} \sum_{\ell=1}^{m} y(z', \eta_\ell) \Pi(z, z')$$

is a Monte Carlo approximation to expected labor income conditional on current state z.

The dashed line is the 45 degree line.

The figure suggests that, on average, the dynamics will be stable — assets do not diverge even in the highest state.

This turns out to be true: there is a unique stationary distribution of assets.

- For details see [Ma *et al.*, 2020]

This stationary distribution represents the long run dispersion of assets across households when households have idiosyncratic shocks.

70.5 Simulation

Let's return to the default model and study the stationary distribution of assets.

Our plan is to run a large number of households forward for T periods and then histogram the cross-sectional distribution of assets.

Set `num_households=50_000, T=500`.

First we write a function to run a single household forward in time and record the final value of assets.

The function takes a solution pair `c_vec` and `a_vec`, understanding them as representing an optimal policy associated with a given model `ifp`

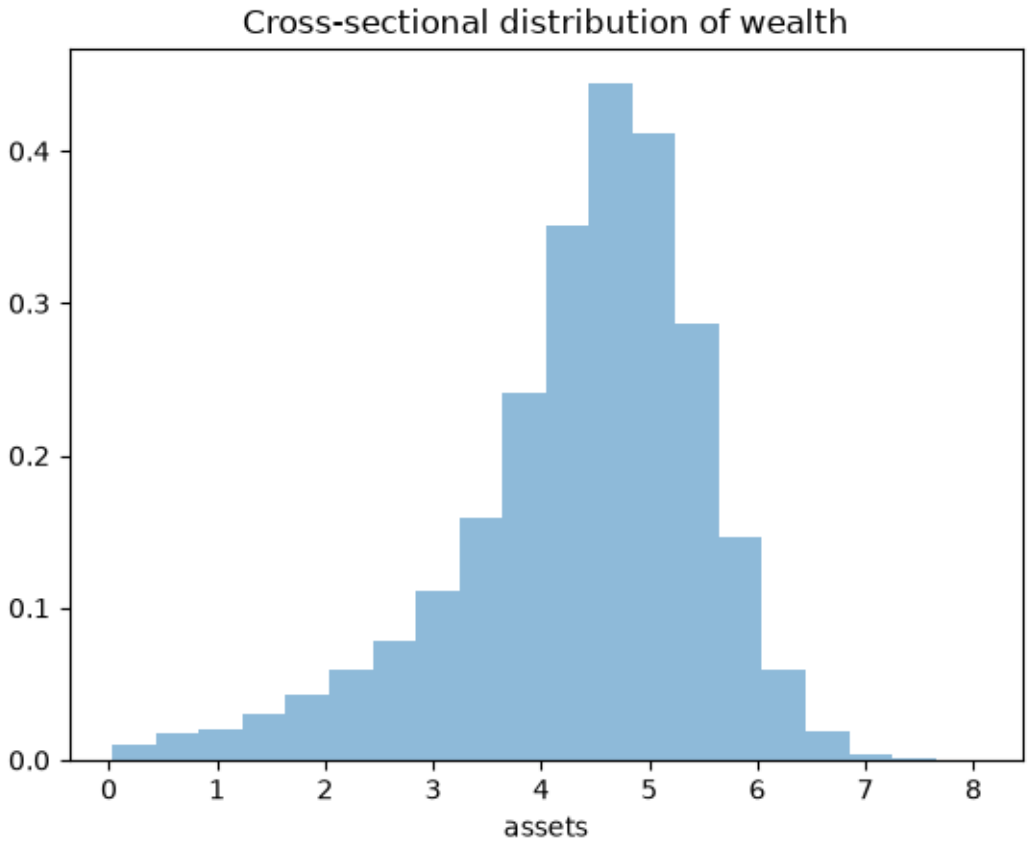
```
@jax.jit
def simulate_household(
        key, a_0, z_idx_0, c_vec, a_vec, ifp, T
    ):
    """
    Simulates a single household for T periods to approximate the stationary
    distribution of assets.

    - key is the state of the random number generator
    - ifp is an instance of IFP
    - c_vec, a_vec are the optimal consumption policy, endogenous grid for ifp

    """
    R, β, γ, Π, z_grid, s, a_y, b_y, η_draws = ifp
    n_z = len(z_grid)

    def y(z, η):
        return jnp.exp(a_y * η + z * b_y)

    # Create interpolation function for consumption policy
    σ = lambda a, z_idx: jnp.interp(a, a_vec[:, z_idx], c_vec[:, z_idx])

    # Simulate forward T periods
    def update(t, state):
        a, z_idx = state
        # Draw next shock z' from Π[z, z']
        current_key = jax.random.fold_in(key, 2*t)
        z_next_idx = jax.random.choice(current_key, n_z, p=Π[z_idx]).astype(jnp.int32)
        z_next = z_grid[z_next_idx]
        # Draw η shock
        η_key = jax.random.fold_in(key, 2*t + 1)
        η = jax.random.normal(η_key)
        # Update assets: a' = R * (a - c) + Y'
        a_next = R * (a - σ(a, z_idx)) + y(z_next, η)
        # Return updated state
        return a_next, z_next_idx

    initial_state = a_0, z_idx_0
    final_state = jax.lax.fori_loop(0, T, update, initial_state)
    a_final, _ = final_state
    return a_final
```

Now we write a function to simulate many households in parallel.

```
def compute_asset_stationary(
        c_vec, a_vec, ifp, num_households=50_000, T=500, seed=1234
    ):
    """
    Simulates num_households households for T periods to approximate
    the stationary distribution of assets.

    Returns the final cross-section of asset holdings.

    - ifp is an instance of IFP
    - c_vec, a_vec are the optimal consumption policy and endogenous grid.

    """
    R, β, γ, Π, z_grid, s, a_y, b_y, η_draws = ifp
    n_z = len(z_grid)

    # Create interpolation function for consumption policy
    # Interpolate on the endogenous grid
    σ = lambda a, z_idx: jnp.interp(a, a_vec[:, z_idx], c_vec[:, z_idx])

    # Start with assets = savings_grid_max / 2
    a_0_vector = jnp.full(num_households, s[-1] / 2)
    # Initialize the exogenous state of each household
    z_idx_0_vector = jnp.zeros(num_households).astype(jnp.int32)

    # Vectorize over many households
    key = jax.random.PRNGKey(seed)
    keys = jax.random.split(key, num_households)
    # Vectorize simulate_household in (key, a_0, z_idx_0)
    sim_all_households = jax.vmap(
        simulate_household, in_axes=(0, 0, 0, None, None, None, None)
    )
    assets = sim_all_households(keys, a_0_vector, z_idx_0_vector, c_vec, a_vec, ifp,␣
↪T)

    return np.array(assets)
```

Now we call the function, generate the asset distribution and histogram it:

```
ifp = create_ifp()
R, β, γ, Π, z_grid, s, a_y, b_y, η_draws = ifp
a_init = s[:, None] * jnp.ones(len(z_grid))
c_init = a_init
c_vec, a_vec = solve_model(ifp, c_init, a_init)
assets = compute_asset_stationary(c_vec, a_vec, ifp)

fig, ax = plt.subplots()
ax.hist(assets, bins=20, alpha=0.5, density=True)
ax.set(xlabel='assets', title="Cross-sectional distribution of wealth")
plt.show()
```



As was the case in *The Income Fluctuation Problem III: The Endogenous Grid Method*, the wealth distribution looks implausible.

While we have at least gained a nontrivial right tail, we still have a left skew.

70.6 Wealth Inequality

Lets' look at wealth inequality by computing some standard measures of this phenomenon.

We will also examine how inequality varies with the interest rate.

70.6.1 Measuring Inequality

We'll compute two common measures of wealth inequality:

1. **Gini coefficient**: A measure of inequality ranging from 0 (perfect equality) to 1 (perfect inequality)
2. **Top 1% wealth share**: The fraction of total wealth held by the richest 1% of households

Here are functions to compute these measures:

```
def gini_coefficient(x):
    """
    Compute the Gini coefficient for array x.

    """
```

(continues on next page)

(continued from previous page)

```
    x = jnp.asarray(x)
    n = len(x)
    x_sorted = jnp.sort(x)
    # Compute Gini coefficient
    cumsum = jnp.cumsum(x_sorted)
    a = (2 * jnp.sum((jnp.arange(1, n+1)) * x_sorted)) / (n * cumsum[-1])
    return a - (n + 1) / n


def top_share(
        x: jnp.array,    # array of wealth values
        p: float=0.01    # fraction of top households (default 0.01 for top 1%)
    ):
    """
    Compute the share of total wealth held by the top p fraction of households.

    """
    x = jnp.asarray(x)
    x_sorted = jnp.sort(x)
    # Number of households in top p%
    n_top = int(jnp.ceil(len(x) * p))
    # Wealth held by top p%
    wealth_top = jnp.sum(x_sorted[-n_top:])
    # Total wealth
    wealth_total = jnp.sum(x_sorted)
    return wealth_top / wealth_total
```

Let's compute these measures for our baseline simulation:

```
gini = gini_coefficient(assets)
top1 = top_share(assets, p=0.01)

print(f"Gini coefficient: {gini:.4f}")
print(f"Top 1% wealth share: {top1:.4f}")
```

```
Gini coefficient: 0.1428
Top 1% wealth share: 0.0154
```

These numbers are a long way out, at least for a country such as the US!

Recent numbers suggest that

- the Gini coefficient for wealth in the US is around 0.8
- the top 1% wealth share is over 0.3

Of course we have not made much effort to accurately estimate or calibrate our parameters.

But actually the cause is deeper — a model with this structure will always struggle to replicate the observed wealth distribution.

In a *later lecture* we'll see if we can improve on these numbers.

70.6.2 Interest Rate and Inequality

Let's examine how wealth inequality varies with the interest rate r.

We conjecture that higher interest rates will increase wealth inequality, as wealthier households benefit more from returns on their assets.

Let's investigate empirically:

```
# Test over 8 interest rate values
M = 8
r_vals = np.linspace(0, 0.05, M)

gini_vals = []
top1_vals = []

# Solve and simulate for each r
for r in r_vals:
    print(f'Analyzing inequality at r = {r:.4f}')
    ifp = create_ifp(r=r)
    R, β, γ, Π, z_grid, s, a_y, b_y, η_draws = ifp
    a_init = s[:, None] * jnp.ones(len(z_grid))
    c_init = a_init
    c_vec, a_vec = solve_model(ifp, c_init, a_init)
    assets = compute_asset_stationary(
        c_vec, a_vec, ifp, num_households=50_000, T=500
    )
    gini = gini_coefficient(assets)
    top1 = top_share(assets, p=0.01)
    gini_vals.append(gini)
    top1_vals.append(top1)
    # Use last solution as initial conditions for the policy solver
    c_init = c_vec
    a_init = a_vec
```

```
Analyzing inequality at r = 0.0000
Analyzing inequality at r = 0.0071
Analyzing inequality at r = 0.0143
Analyzing inequality at r = 0.0214
Analyzing inequality at r = 0.0286
Analyzing inequality at r = 0.0357
Analyzing inequality at r = 0.0429
Analyzing inequality at r = 0.0500
```

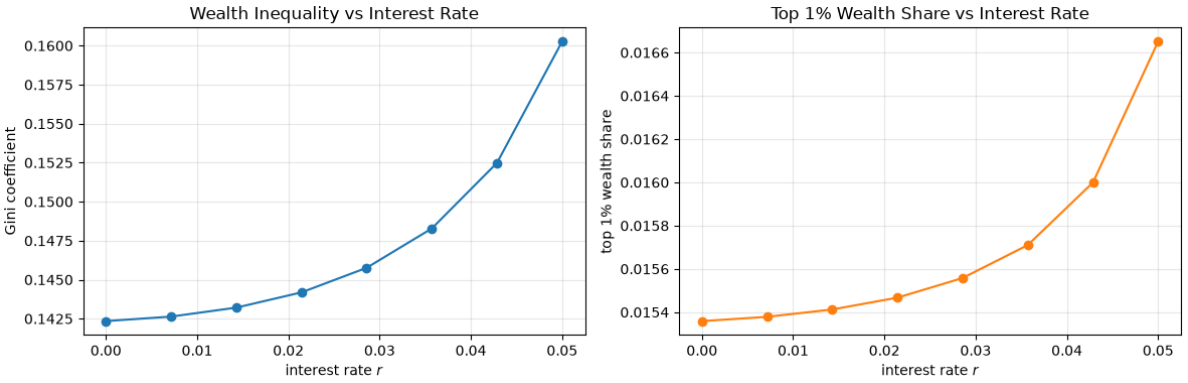
Now let's visualize the results:

```
fig, axes = plt.subplots(1, 2, figsize=(12, 4))

# Plot Gini coefficient vs interest rate
axes[0].plot(r_vals, gini_vals, 'o-')
axes[0].set_xlabel('interest rate $r$')
axes[0].set_ylabel('Gini coefficient')
axes[0].set_title('Wealth Inequality vs Interest Rate')
axes[0].grid(alpha=0.3)

# Plot top 1% share vs interest rate
axes[1].plot(r_vals, top1_vals, 'o-', color='C1')
axes[1].set_xlabel('interest rate $r$')
```

(continues on next page)

(continued from previous page)

```
axes[1].set_ylabel('top 1% wealth share')
axes[1].set_title('Top 1% Wealth Share vs Interest Rate')
axes[1].grid(alpha=0.3)

plt.tight_layout()
plt.show()
```



The results show that these two inequality measures increase with the interest rate.

However the differences are minor and we cannot increase r much more without violating the stability constraint.

Certainly changing the interest rate cannot produce the kinds of numbers that we see in the data.

70.7 Exercises

Exercise 70.7.1

Let's consider how the interest rate affects consumption.

- Step `r` through `np.linspace(0, 0.016, 4)`.
- Other than `r`, hold all parameters at their default values.
- Plot consumption against assets for income shock fixed at the smallest value.

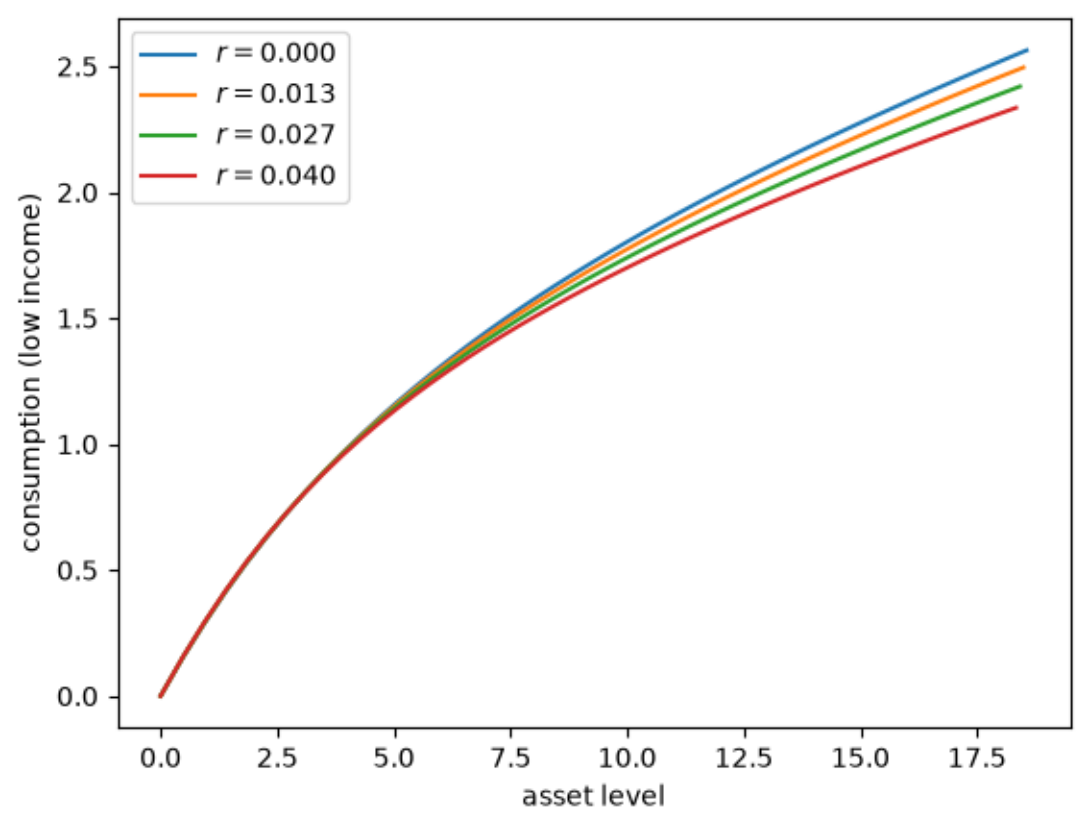
Your figure should show that, for this model, higher interest rates suppress consumption (because they encourage more savings).

Solution

Here's one solution:

```
# With β=0.96, we need R*β < 1, so r < 0.0416
r_vals = np.linspace(0, 0.04, 4)

fig, ax = plt.subplots()
for r_val in r_vals:
    ifp = create_ifp(r=r_val)
    R, β, γ, Π, z_grid, s, a_y, b_y, η_draws = ifp
    a_init = s[:, None] * jnp.ones(len(z_grid))
    c_init = a_init
```

```
    c_vec, a_vec = solve_model(ifp, c_init, a_init)
    # Plot policy
    ax.plot(a_vec[:, 0], c_vec[:, 0], label=f'$r = {r_val:.3f}$')
    # Start next round with last solution
    c_init = c_vec
    a_init = a_vec

ax.set(xlabel='asset level', ylabel='consumption (low income)')
ax.legend()
plt.show()
```



Exercise 70.7.2

Following on from Exercises 1, let's look at how savings and aggregate asset holdings vary with the interest rate

Note

[Ljungqvist and Sargent, 2018] section 18.6 can be consulted for more background on the topic treated in this exercise.

For a given parameterization of the model, the mean of the stationary distribution of assets can be interpreted as aggregate capital in an economy with a unit mass of *ex-ante* identical households facing idiosyncratic shocks.

Your task is to investigate how this measure of aggregate capital varies with the interest rate.

Intuition suggests that a higher interest rate should encourage capital formation — test this.

For the interest rate grid, use

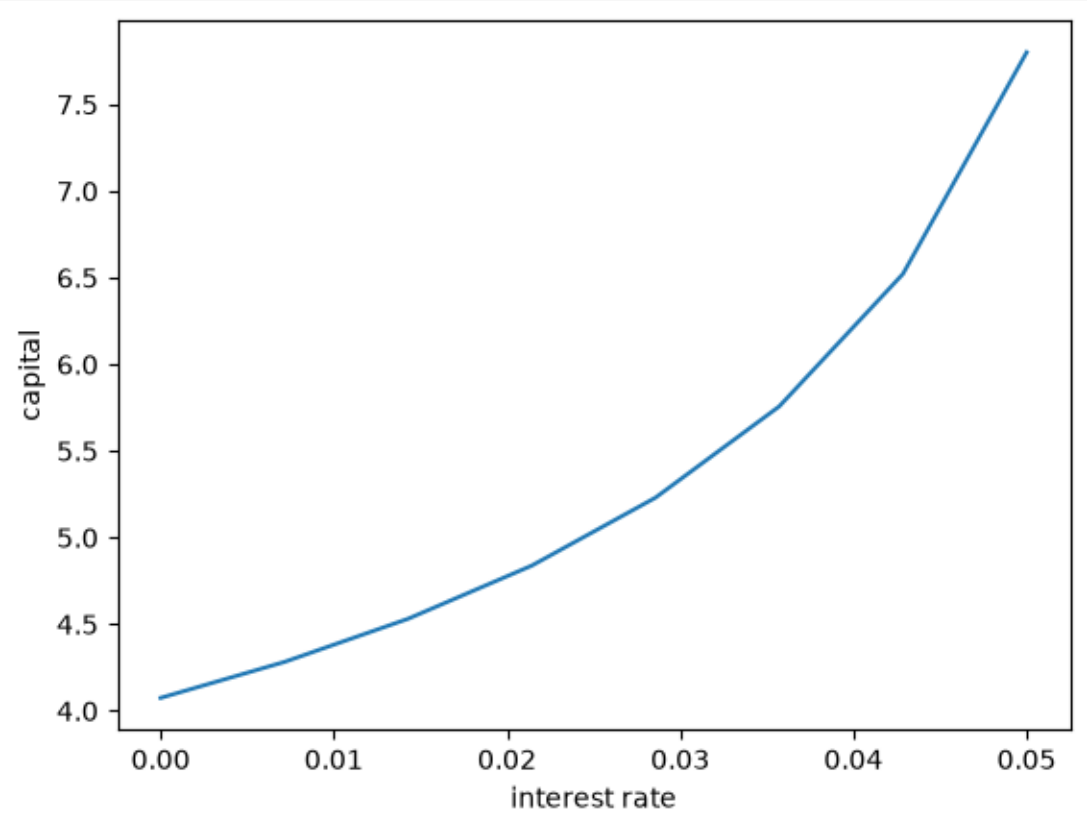
```
M = 8
r_vals = np.linspace(0, 0.05, M)
```

Solution

Here's one solution

```
fig, ax = plt.subplots()

asset_mean = []
for r in r_vals:
    print(f'Solving model at r = {r}')
    ifp = create_ifp(r=r)
    R, β, γ, Π, z_grid, s, a_y, b_y, η_draws = ifp
    a_init = s[:, None] * jnp.ones(len(z_grid))
    c_init = a_init
    c_vec, a_vec = solve_model(ifp, c_init, a_init)
    assets = compute_asset_stationary(
        c_vec, a_vec, ifp, num_households=10_000, T=500
    )
    mean = np.mean(assets)
    asset_mean.append(mean)
    print(f'  Mean assets: {mean:.4f}')
    # Start next round with last solution
    c_init = c_vec
    a_init = a_vec
ax.plot(r_vals, asset_mean)

ax.set(xlabel='interest rate', ylabel='capital')

plt.show()
```

```
Solving model at r = 0.0
  Mean assets: 4.0748
Solving model at r = 0.0071428571428571435
  Mean assets: 4.2822
Solving model at r = 0.014285714285714287
  Mean assets: 4.5313
Solving model at r = 0.02142857142857143
  Mean assets: 4.8383
Solving model at r = 0.028571428571428574
  Mean assets: 5.2304
Solving model at r = 0.03571428571428572
  Mean assets: 5.7570
Solving model at r = 0.04285714285714286
  Mean assets: 6.5214
Solving model at r = 0.05
  Mean assets: 7.8005
```



As expected, aggregate savings increases with the interest rate.

CHAPTER

SEVENTYONE

THE INCOME FLUCTUATION PROBLEM V: STOCHASTIC RETURNS ON ASSETS

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

Contents

- *The Income Fluctuation Problem V: Stochastic Returns on Assets*
  - *Overview*
  - *The Model*
  - *Solution Algorithm*
  - *Implementation*
  - *Simulation*
  - *Wealth Inequality*
  - *Exercises*

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install quantecon
```

71.1 Overview

In this lecture, we continue our study of the income fluctuation problem described in *The Income Fluctuation Problem III: The Endogenous Grid Method*.

While the interest rate was previously taken to be fixed, we now allow returns on assets to be state-dependent.

This matches the fact that most households with a positive level of assets face some capital income risk.

It has been argued that modeling capital income risk is essential for understanding the joint distribution of income and wealth (see, e.g., [Benhabib *et al.*, 2015] or [Stachurski and Toda, 2019]).

Theoretical properties of the household savings model presented here are analyzed in detail in [Ma *et al.*, 2020].

In terms of computation, we use a combination of time iteration and the endogenous grid method to solve the model quickly and accurately.

We require the following imports:

```
import matplotlib.pyplot as plt
import numpy as np
import quantecon as qe
import jax
import jax.numpy as jnp
from jax import vmap
from typing import NamedTuple
from functools import partial
```

71.2 The Model

In this section we review the household problem and optimality results.

71.2.1 Set Up

A household chooses a consumption-asset path $\{(c_t, a_t)\}$ to maximize

$$\mathbb{E}\left\{\sum_{t=0}^{\infty} \beta^t u(c_t)\right\} \tag{71.1}$$

subject to

$$a_{t+1} = R_{t+1}(a_t - c_t) + Y_{t+1} \text{ and } 0 \leq c_t \leq a_t, \tag{71.2}$$

with initial condition $(a_0, Z_0) = (a, z)$ treated as given.

The only difference from *The Income Fluctuation Problem IV: Transient Income Shocks* is that $\{R_t\}_{t\geq 1}$, the gross rate of return on wealth, is allowed to be stochastic.

In particular, we assume that

$$R_t = R(Z_t, \zeta_t) \quad \text{and} \quad Y_t = Y(Z_t, \eta_t), \tag{71.3}$$

where

- R and Y are time-invariant nonnegative functions,
- the innovation processes $\{\zeta_t\}$ and $\{\eta_t\}$ are IID and independent of each other, and

- $\{Z_t\}_{t \geq 0}$ is a Markov chain on a finite set Z

Let P represent the Markov matrix for the chain $\{Z_t\}_{t \geq 0}$.

In what follows, $\mathbb{E}_z \hat{X}$ means expectation of next period value $\hat{X}$ given current value $Z = z$.

71.2.2 Assumptions

We need restrictions to ensure that the objective (71.1) is finite and the solution methods described below converge.

We also need to ensure that the present discounted value of wealth does not grow too quickly.

When $\{R_t\}$ was constant we required that $\beta R < 1$.

Since it is now stochastic, we require (see [Ma *et al.*, 2020]) that

$$\beta G_R < 1, \quad \text{where} \quad G_R := \lim_{n \to \infty} \left(\mathbb{E} \prod_{t=1}^{n} R_t \right)^{1/n} \tag{71.4}$$

The value G_R can be thought of as the long run (geometric) average gross rate of return.

To simplify this lecture, we will *assume that the interest rate process is IID*.

In that case, it is clear from the definition of G_R that G_R is just $\mathbb{E} R_t$.

We test the condition $\beta \mathbb{E} R_t < 1$ in the code below.

Finally, we impose some routine technical restrictions on non-financial income.

$$\mathbb{E}\, Y_t < \infty \text{ and } \mathbb{E}\, u'(Y_t) < \infty$$

One relatively simple setting where all these restrictions are satisfied is the IID and CRRA environment of [Benhabib *et al.*, 2015].

71.2.3 Optimality

Let the class of candidate consumption policies $\mathcal{C}$ be defined as in *The Income Fluctuation Problem III: The Endogenous Grid Method*.

In [Ma *et al.*, 2020] it is shown that, under the stated assumptions,

- any $\sigma \in \mathcal{C}$ satisfying the Euler equation is an optimal policy and
- exactly one such policy exists in $\mathcal{C}$.

In the present setting, the Euler equation takes the form

$$(u' \circ \sigma)(a, z) = \max \left\{ \beta\, \mathbb{E}_z\, \hat{R}\, (u' \circ \sigma)[\hat{R}(a - \sigma(a, z)) + \hat{Y},\ \hat{Z}],\ u'(a) \right\} \tag{71.5}$$

(Intuition and derivation are similar to *The Income Fluctuation Problem III: The Endogenous Grid Method*.)

We again solve the Euler equation using time iteration, iterating with a Coleman–Reffett operator K defined to match the Euler equation (71.5).

71.3 Solution Algorithm

71.3.1 A Time Iteration Operator

Our definition of the candidate class $\sigma \in \mathcal{C}$ of consumption policies is the same as in *The Income Fluctuation Problem III: The Endogenous Grid Method*.

For fixed $\sigma \in \mathcal{C}$ and $(a, z) \in \mathbf{S}$, the value $K\sigma(a, z)$ of the function $K\sigma$ at (a, z) is defined as the $\xi \in (0, a]$ that solves

$$u'(\xi) = \max \left\{ \beta \, \mathbb{E}_z \, \hat{R} \, (u' \circ \sigma)[\hat{R}(a - \xi) + \hat{Y}, \, \hat{Z}], \, u'(a) \right\} \tag{71.6}$$

The idea behind K is that, as can be seen from the definitions, $\sigma \in \mathcal{C}$ satisfies the Euler equation if and only if $K\sigma(a, z) = \sigma(a, z)$ for all $(a, z) \in \mathbf{S}$.

This means that fixed points of K in $\mathcal{C}$ and optimal consumption policies exactly coincide (see [Ma *et al.*, 2020] for more details).

71.3.2 Convergence Properties

As before, we pair $\mathcal{C}$ with the distance

$$\rho(c, d) := \sup_{(a,z) \in \mathbf{S}} \left| (u' \circ c) \, (a, z) - (u' \circ d) \, (a, z) \right|,$$

It can be shown that

1. $(\mathcal{C}, \rho)$ is a complete metric space,
2. there exists an integer n such that K^n is a contraction mapping on $(\mathcal{C}, \rho)$, and
3. The unique fixed point of K in $\mathcal{C}$ is the unique optimal policy in $\mathcal{C}$.

We now have a clear path to successfully approximating the optimal policy: choose some $\sigma \in \mathcal{C}$ and then iterate with K until convergence (as measured by the distance ρ).

71.3.3 Using an Endogenous Grid

In the study of that model we found that it was possible to further accelerate time iteration via the *endogenous grid method*.

We will use the same method here.

The methodology is the same as it was for the optimal growth model, with the minor exception that we need to remember that consumption is not always interior.

In particular, optimal consumption can be equal to assets when the level of assets is low.

Finding Optimal Consumption

The endogenous grid method (EGM) calls for us to take a grid of *savings* values s_i, where each such s is interpreted as $s = a - c$.

For the lowest grid point we take $s_0 = 0$.

For the corresponding a_0, c_0 pair we have $a_0 = c_0$.

This happens close to the origin, where assets are low and the household consumes all that it can.

Although there are many solutions, the one we take is $a_0 = c_0 = 0$, which pins down the policy at the origin, aiding interpolation.

For $s > 0$, we have, by definition, $c < a$, and hence consumption is interior.

Hence the max component of (71.5) drops out, and we solve for

$$c_i = (u')^{-1} \left\{ \beta \, \mathbb{E}_z \hat{R} (u' \circ \sigma) \left[\hat{R} s_i + \hat{Y}, \, \hat{Z} \right] \right\} \tag{71.7}$$

at each s_i.

Iterating

Once we have the pairs $\{s_i, c_i\}$, the endogenous asset grid is obtained by $a_i = c_i + s_i$.

Also, we held $z \in \mathsf{Z}$ in the discussion above so we can pair it with a_i.

An approximation of the policy $(a, z) \mapsto \sigma(a, z)$ can be obtained by interpolating $\{a_i, c_i\}$ at each z.

In what follows, we use linear interpolation.

71.4 Implementation

Here's the model as a `NamedTuple`.

```
class IFP(NamedTuple):
    """
    A NamedTuple that stores primitives for the income fluctuation
    problem, using JAX.
    """
    γ: float
    β: float
    P: jnp.ndarray
    a_r: float
    b_r: float
    a_y: float
    b_y: float
    s_grid: jnp.ndarray
    η_draws: jnp.ndarray
    ζ_draws: jnp.ndarray


def create_ifp(
        γ=1.5,                      # Utility parameter
        β=0.96,                     # Discount factor
        P=jnp.array([(0.9, 0.1),    # Default Markov chain for Z
                     (0.1, 0.9)]),
        a_r=0.16,                   # Volatility term in R shock
        b_r=0.0,                    # Mean shift R shock
        a_y=0.2,                    # Volatility term in Y shock
        b_y=0.5,                    # Mean shift Y shock
        shock_draw_size=100,        # For Monte Carlo
        grid_max=100,               # Exogenous grid max
        grid_size=100,              # Exogenous grid size
        seed=1234                   # Random seed
    ):
```

(continues on next page)

(continued from previous page)

```
    """
    Create an instance of IFP with the given parameters.

    """
    # Test stability assuming {R_t} is IID and ln R ~ N(b_r, a_r)
    ER = np.exp(b_r + a_r**2 / 2)
    assert β * ER < 1, "Stability condition failed."

    # Generate random draws using JAX
    key = jax.random.PRNGKey(seed)
    subkey1, subkey2 = jax.random.split(key)
    η_draws = jax.random.normal(subkey1, (shock_draw_size,))
    ζ_draws = jax.random.normal(subkey2, (shock_draw_size,))
    s_grid = jnp.linspace(0, grid_max, grid_size)

    return IFP(
        γ, β, P, a_r, b_r, a_y, b_y, s_grid, η_draws, ζ_draws
    )


def u_prime(c, γ):
    """Marginal utility"""
    return c**(-γ)

def u_prime_inv(c, γ):
    """Inverse of marginal utility"""
    return c**(-1/γ)

def R(z, ζ, a_r, b_r):
    """Gross return on assets"""
    return jnp.exp(a_r * ζ + b_r)

def Y(z, η, a_y, b_y):
    """Labor income"""
    return jnp.exp(a_y * η + (z * b_y))
```

Here's the Coleman-Reffett operator using JAX:

```
def K(
        a_in: jnp.array,    # a_in[i, z] is an asset grid
        c_in: jnp.array,    # c_in[i, z] = consumption at a_in[i, z]
        ifp: IFP
    ):
    """
    The Coleman--Reffett operator for the income fluctuation problem,
    using the endogenous grid method with JAX.

    """

    # Extract parameters from ifp
    γ, β, P, a_r, b_r, a_y, b_y, s_grid, η_draws, ζ_draws = ifp
    n = len(P)

    def compute_expectation(s, z):
        def inner_expectation(z_hat):
            def compute_term(η, ζ):
                R_hat = R(z_hat, ζ, a_r, b_r)
```

(continues on next page)

(continued from previous page)

```
                Y_hat = Y(z_hat, η, a_y, b_y)
                a_val = R_hat * s + Y_hat
                # Interpolate consumption
                c_interp = jnp.interp(a_val, a_in[:, z_hat], c_in[:, z_hat])
                mu = u_prime(c_interp, γ)
                return R_hat * mu
            # Vectorize over all shock combinations
            η_grid, ζ_grid = jnp.meshgrid(η_draws, ζ_draws, indexing='ij')
            terms = vmap(vmap(compute_term))(η_grid, ζ_grid)
            return P[z, z_hat] * jnp.mean(terms)
        # Sum over z_hat states
        Ez = jnp.sum(vmap(inner_expectation)(jnp.arange(n)))
        return u_prime_inv(β * Ez, γ)

    # Vectorize over s_grid and z
    compute_exp_v1 = vmap(compute_expectation, in_axes=(None, 0))
    compute_exp_v2 = vmap(compute_exp_v1,      in_axes=(0, None))
    c_out = compute_exp_v2(s_grid, jnp.arange(n))
    # Calculate endogenous asset grid
    a_out = s_grid[:, None] + c_out
    # Fix consumption-asset pair at (0, 0)
    c_out = c_out.at[0, :].set(0)
    a_out = a_out.at[0, :].set(0)

    return a_out, c_out
```

The next function solves for an approximation of the optimal consumption policy via time iteration using JAX:

```
@jax.jit
def solve_model(
        ifp: IFP,
        c_init: jnp.ndarray,  # Initial guess of σ on grid endogenous grid
        a_init: jnp.ndarray,  # Initial endogenous grid
        tol: float = 1e-5,
        max_iter: int = 1000
    ) -> jnp.ndarray:
    " Solve the model using time iteration with EGM. "

    def condition(loop_state):
        c_in, a_in, i, error = loop_state
        return (error > tol) & (i < max_iter)

    def body(loop_state):
        c_in, a_in, i, error = loop_state
        c_out, a_out = K(c_in, a_in, ifp)
        error = jnp.max(jnp.abs(c_out - c_in))
        i += 1
        return c_out, a_out, i, error

    i, error = 0, tol + 1
    initial_state = (c_init, a_init, i, error)
    final_loop_state = jax.lax.while_loop(condition, body, initial_state)
    c_out, a_out, i, error = final_loop_state

    return c_out, a_out
```

Now we can create an instance and solve the model using JAX:

```
ifp = create_ifp()
```

Set up the initial condition:

```
# Initial guess of σ = consume all assets
k = len(ifp.s_grid)
n = len(ifp.P)
σ_init = jnp.empty((k, n))
for z in range(n):
    σ_init = σ_init.at[:, z].set(ifp.s_grid)
a_init = σ_init.copy()
```

Let's generate an approximation solution with JAX:

```
a_star, σ_star = solve_model(ifp, a_init, σ_init)
```

Let's try it again with a timer.

```
with qe.Timer(precision=8):
    a_star, σ_star = solve_model(ifp, a_init, σ_init)
    a_star.block_until_ready()
```

```
0.46803522 seconds elapsed
```

71.5 Simulation

Let's return to the default model and study the stationary distribution of assets.

Our plan is to run a large number of households forward for T periods and then histogram the cross-sectional distribution of assets.

Set `num_households=50_000, T=500`.

First we write a function to run a single household forward in time and record the final value of assets.

The function takes a solution pair `c_vec` and `a_vec`, understanding them as representing an optimal policy associated with a given model `ifp`

```
def simulate_household(
        key, a_0, z_idx_0, c_vec, a_vec, ifp, T
    ):
    """
    Simulates a single household for T periods to approximate the stationary
    distribution of assets.

    - key is the state of the random number generator
    - ifp is an instance of IFP
    - c_vec, a_vec are the optimal consumption policy, endogenous grid for ifp

    """
    # Extract parameters from ifp
    γ, β, P, a_r, b_r, a_y, b_y, s_grid, η_draws, ζ_draws = ifp
    n_z = len(P)

    # Create interpolation function for consumption policy
```

(continues on next page)

(continued from previous page)

```
    σ = lambda a, z_idx: jnp.interp(a, a_vec[:, z_idx], c_vec[:, z_idx])

    # Simulate forward T periods
    def update(t, state):
        a, z_idx = state
        # Draw next shock z' from P[z, z']
        current_key = jax.random.fold_in(key, 3*t)
        z_next_idx = jax.random.choice(current_key, n_z, p=P[z_idx]).astype(jnp.int32)
        # Draw η shock for income
        η_key = jax.random.fold_in(key, 3*t + 1)
        η = jax.random.normal(η_key)
        # Draw ζ shock for return
        ζ_key = jax.random.fold_in(key, 3*t + 2)
        ζ = jax.random.normal(ζ_key)
        # Compute stochastic return
        R_next = R(z_next_idx, ζ, a_r, b_r)
        # Compute income
        Y_next = Y(z_next_idx, η, a_y, b_y)
        # Update assets: a' = R' * (a - c) + Y'
        a_next = R_next * (a - σ(a, z_idx)) + Y_next
        # Return updated state
        return a_next, z_next_idx

    initial_state = a_0, z_idx_0
    final_state = jax.lax.fori_loop(0, T, update, initial_state)
    a_final, _ = final_state
    return a_final
```

Now we write a function to simulate many households in parallel.

```
@partial(jax.jit, static_argnums=(3, 4, 5))
def compute_asset_stationary(
        c_vec, a_vec, ifp, num_households=50_000, T=500, seed=1234
    ):
    """
    Simulates num_households households for T periods to approximate
    the stationary distribution of assets.

    Returns the final cross-section of asset holdings.

    - ifp is an instance of IFP
    - c_vec, a_vec are the optimal consumption policy and endogenous grid.

    """
    # Extract parameters from ifp
    γ, β, P, a_r, b_r, a_y, b_y, s_grid, η_draws, ζ_draws = ifp

    # Start with assets = savings_grid_max / 2
    a_0_vector = jnp.full(num_households, s_grid[-1] / 2)
    # Initialize the exogenous state of each household
    z_idx_0_vector = jnp.zeros(num_households).astype(jnp.int32)

    # Vectorize over many households
    key = jax.random.PRNGKey(seed)
    keys = jax.random.split(key, num_households)
    # Vectorize simulate_household in (key, a_0, z_idx_0)
    sim_all_households = jax.vmap(
```

(continues on next page)

(continued from previous page)

```
        simulate_household, in_axes=(0, 0, 0, None, None, None, None)
    )
    assets = sim_all_households(keys, a_0_vector, z_idx_0_vector, c_vec, a_vec, ifp,␣
↪T)

    return jnp.array(assets)
```

We'll need some inequality measures for visualization, so let's define them first:

```
def gini_coefficient(x):
    """
    Compute the Gini coefficient for array x.

    """
    x = jnp.asarray(x)
    n = len(x)
    x_sorted = jnp.sort(x)
    # Compute Gini coefficient
    cumsum = jnp.cumsum(x_sorted)
    a = (2 * jnp.sum((jnp.arange(1, n+1)) * x_sorted)) / (n * cumsum[-1])
    return a - (n + 1) / n


def top_share(
        x: jnp.array,    # array of wealth values
        p: float=0.01    # fraction of top households (default 0.01 for top 1%)
    ):
    """
    Compute the share of total wealth held by the top p fraction of households.

    """
    x = jnp.asarray(x)
    x_sorted = jnp.sort(x)
    # Number of households in top p%
    n_top = int(jnp.ceil(len(x) * p))
    # Wealth held by top p%
    wealth_top = jnp.sum(x_sorted[-n_top:])
    # Total wealth
    wealth_total = jnp.sum(x_sorted)
    return wealth_top / wealth_total
```

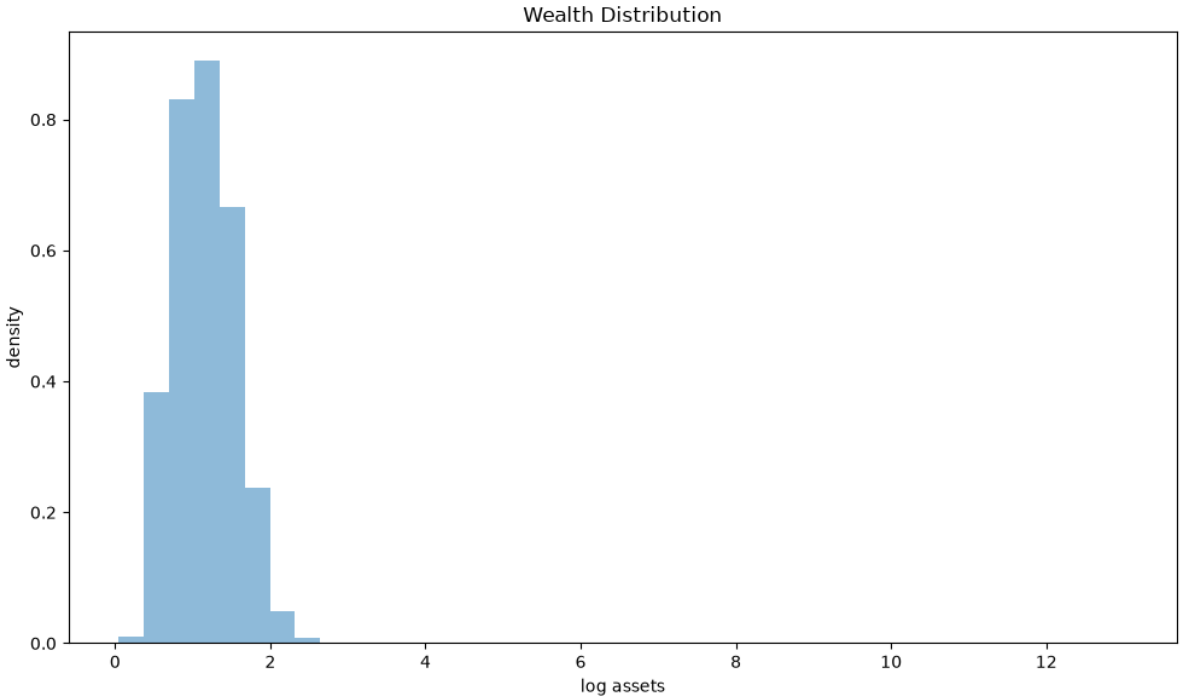
Now we call the function, generate the asset distribution and visualize it:

```
ifp = create_ifp()
# Extract parameters for initialization
s_grid = ifp.s_grid
n_z = len(ifp.P)
a_init = s_grid[:, None] * jnp.ones(n_z)
c_init = a_init
a_vec, c_vec = solve_model(ifp, a_init, c_init)
assets = compute_asset_stationary(c_vec, a_vec, ifp, num_households=200_000)

# Compute Gini coefficient for the plot
gini_plot = gini_coefficient(assets)

# Plot histogram of log wealth
fig, ax = plt.subplots(figsize=(10, 6))
```

(continues on next page)

(continued from previous page)

```
ax.hist(jnp.log(assets), bins=40, alpha=0.5, density=True)
ax.set(xlabel='log assets', ylabel='density', title="Wealth Distribution")
plt.tight_layout()
plt.show()
```



The histogram shows the distribution of log wealth.

Bearing in mind that we are looking at log values, the histogram suggests a long right tail of the distribution.

Below we examine this in more detail.

71.6 Wealth Inequality

Lets' look at wealth inequality by computing some standard measures of this phenomenon.

We will also examine how inequality varies with the interest rate.

71.6.1 Measuring Inequality

Let's print the Gini coefficient and the top 1% wealth share from our simulation:

```
gini = gini_coefficient(assets)
top1 = top_share(assets, p=0.01)

print(f"Gini coefficient: {gini:.4f}")
print(f"Top 1% wealth share: {top1:.4f}")
```

```
Gini coefficient: 0.7847
Top 1% wealth share: 0.7306
```

Recent numbers suggest that

- the Gini coefficient for wealth in the US is around 0.8
- the top 1% wealth share is over 0.3

Our model with stochastic returns generates a Gini coefficient close to the empirical value, demonstrating that capital income risk is an important factor in wealth inequality.

The top 1% wealth share is, however, too large.

Our model needs proper calibration and additional work – we set these tasks aside for now.

71.7 Exercises

Exercise 71.7.1

Plot how the Gini coefficient varies with the volatility of returns on assets.

Specifically, compute the Gini coefficient for values of `a_r` ranging from 0.10 to 0.16 (use at least 5 different values) and plot the results.

What does this tell you about the relationship between capital income risk and wealth inequality?

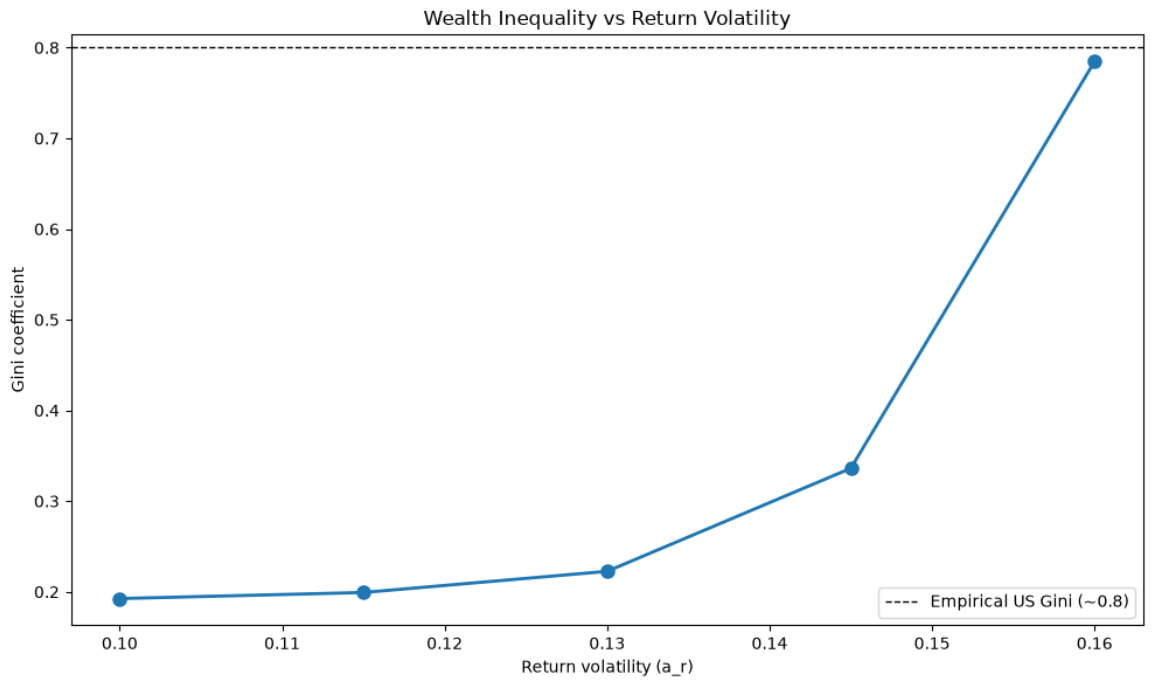
Solution

We loop over different values of `a_r`, solve the model for each, simulate the wealth distribution, and compute the Gini coefficient.

```
# Range of a_r values to explore
a_r_vals = np.linspace(0.10, 0.16, 5)
gini_vals = []

print("Computing Gini coefficients for different return volatilities...\n")

for a_r in a_r_vals:
    print(f"a_r = {a_r:.3f}...", end=" ")

    # Create model with this a_r value
    ifp_temp = create_ifp(a_r=a_r, grid_max=100)

    # Solve the model
    s_grid_temp = ifp_temp.s_grid
    n_z_temp = len(ifp_temp.P)
    a_init_temp = s_grid_temp[:, None] * jnp.ones(n_z_temp)
    c_init_temp = a_init_temp
    a_vec_temp, c_vec_temp = solve_model(
        ifp_temp, a_init_temp, c_init_temp
    )

    # Simulate households
    assets_temp = compute_asset_stationary(
        c_vec_temp, a_vec_temp, ifp_temp, num_households=200_000
    )

    # Compute Gini coefficient
```

```
    gini_temp = gini_coefficient(assets_temp)
    gini_vals.append(gini_temp)
    print(f"Gini = {gini_temp:.4f}")

# Plot the results
fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(a_r_vals, gini_vals, 'o-', linewidth=2, markersize=8)
ax.set(xlabel='Return volatility (a_r)',
       ylabel='Gini coefficient',
       title='Wealth Inequality vs Return Volatility')
ax.axhline(y=0.8, color='k', linestyle='--', linewidth=1,
           label='Empirical US Gini (~0.8)')
ax.legend()
plt.tight_layout()
plt.show()
```

```
Computing Gini coefficients for different return volatilities...

a_r = 0.100...
Gini = 0.1925
a_r = 0.115...
Gini = 0.1992
a_r = 0.130...
Gini = 0.2225
a_r = 0.145...
Gini = 0.3362
a_r = 0.160...
Gini = 0.7847
```



The plot shows that wealth inequality (measured by the Gini coefficient) increases with return volatility.

This demonstrates that capital income risk is a key driver of wealth inequality.

When returns are more volatile, lucky households who experience sequences of high returns accumulate substantially

more wealth than unlucky households, leading to greater inequality in the wealth distribution.

Exercise 71.7.2

Plot how the Gini coefficient varies with the volatility of labor income.

Specifically, compute the Gini coefficient for values of `a_y` ranging from 0.125 to 0.20 and plot the results. Set `a_r=0.10` for this exercise.

What does this tell you about the relationship between labor income risk and wealth inequality? Can we achieve the same rise in inequality by varying labor income volatility as we can by varying return volatility?

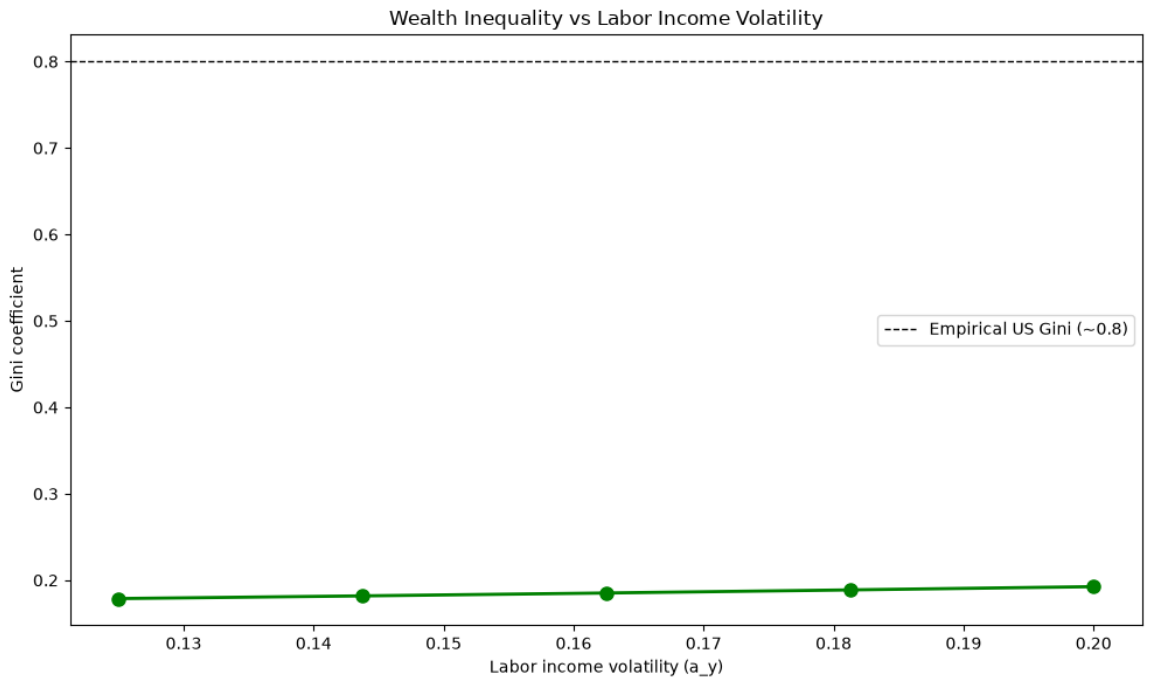
Solution

We loop over different values of `a_y`, solve the model for each, simulate the wealth distribution, and compute the Gini coefficient.

```
# Range of a_y values to explore
a_y_vals = np.linspace(0.125, 0.20, 5)
gini_vals_y = []

print("Computing Gini coefficients for different labor income volatilities...\n")

for a_y in a_y_vals:
    print(f"a_y = {a_y:.3f}...", end=" ")

    # Create model with this a_y value and a_r=0.10
    ifp_temp = create_ifp(a_y=a_y, a_r=0.10, grid_max=100)

    # Solve the model
    s_grid_temp = ifp_temp.s_grid
    n_z_temp = len(ifp_temp.P)
    a_init_temp = s_grid_temp[:, None] * jnp.ones(n_z_temp)
    c_init_temp = a_init_temp
    a_vec_temp, c_vec_temp = solve_model(
        ifp_temp, a_init_temp, c_init_temp
    )

    # Simulate households
    assets_temp = compute_asset_stationary(
        c_vec_temp, a_vec_temp, ifp_temp, num_households=200_000
    )

    # Compute Gini coefficient
    gini_temp = gini_coefficient(assets_temp)
    gini_vals_y.append(gini_temp)
    print(f"Gini = {gini_temp:.4f}")

# Plot the results
fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(a_y_vals, gini_vals_y, 'o-', linewidth=2, markersize=8, color='green')
ax.set(xlabel='Labor income volatility (a_y)',
       ylabel='Gini coefficient',
       title='Wealth Inequality vs Labor Income Volatility')
ax.axhline(y=0.8, color='k', linestyle='--', linewidth=1,
```

```
            label='Empirical US Gini (~0.8)')
ax.legend()
plt.tight_layout()
plt.show()
```

```
Computing Gini coefficients for different labor income volatilities...

a_y = 0.125...
Gini = 0.1787
a_y = 0.144...
Gini = 0.1818
a_y = 0.163...
Gini = 0.1852
a_y = 0.181...
Gini = 0.1887
a_y = 0.200...
Gini = 0.1925
```



The plot shows that wealth inequality increases with labor income volatility, but the effect is much weaker than the effect of return volatility.

Comparing the two exercises:

- When return volatility (`a_r`) varies from 0.10 to 0.16, the Gini coefficient rises dramatically from around 0.20 to 0.79
- When labor income volatility (`a_y`) varies from 0.125 to 0.20, a similar amount in percentage terms, the Gini coefficient increases but by a much smaller amount

This suggests that capital income risk is a more important driver of wealth inequality than labor income risk.

The intuition is that wealth accumulation compounds over time: households who experience favorable returns on their assets can reinvest those returns, leading to exponential growth.

In contrast, labor income shocks, while they affect current consumption and savings, do not have the same compound-

ing effect on wealth accumulation.

Part XI

LQ Control

CHAPTER

SEVENTYTWO

LQ CONTROL: FOUNDATIONS

Contents

- *LQ Control: Foundations*
  - *Overview*
  - *Introduction*
  - *Optimality – finite horizon*
  - *Implementation*
  - *Extensions and comments*
  - *Further applications*
  - *Exercises*

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install quantecon
```

72.1 Overview

Linear quadratic (LQ) control refers to a class of dynamic optimization problems that have found applications in almost every scientific field.

This lecture provides an introduction to LQ control and its economic applications.

As we will see, LQ systems have a simple structure that makes them an excellent workhorse for a wide variety of economic problems.

Moreover, while the linear-quadratic structure is restrictive, it is in fact far more flexible than it may appear initially.

These themes appear repeatedly below.

Mathematically, LQ control problems are closely related to *the Kalman filter*

- Recursive formulations of linear-quadratic control problems and Kalman filtering problems both involve matrix Riccati equations.
- Classical formulations of linear control and linear filtering problems make use of similar matrix decompositions (see for example Classical Control with Linear Algebra and Classical Prediction and Filtering With Linear Algebra).

In reading what follows, it will be useful to have some familiarity with

- matrix manipulations
- vectors of random variables
- dynamic programming and the Bellman equation (see for example Shortest Paths and *Optimal Savings III: Stochastic Returns*)

For additional reading on LQ control, see, for example,

- [Ljungqvist and Sargent, 2018], chapter 5
- [Hansen and Sargent, 2008], chapter 4
- [Hernandez-Lerma and Lasserre, 1996], section 3.5

In order to focus on computation, we leave longer proofs to these sources (while trying to provide as much intuition as possible).

Let's start with some imports:

```
import matplotlib.pyplot as plt
import numpy as np
from quantecon import LQ
from typing import NamedTuple
```

72.2 Introduction

The "linear" part of LQ is a linear law of motion for the state, while the "quadratic" part refers to preferences.

Let's begin with the former, move on to the latter, and then put them together into an optimization problem.

72.2.1 The law of motion

Let x_t be a vector describing the state of some economic system.

Suppose that x_t follows a linear law of motion given by

$$x_{t+1} = Ax_t + Bu_t + Cw_{t+1}, \qquad t = 0, 1, 2, \ldots \tag{72.1}$$

Here

- u_t is a "control" vector, incorporating choices available to a decision-maker confronting the current state x_t
- $\{w_t\}$ is a sequence of uncorrelated zero mean shocks satisfying $\mathbb{E} w_t w_t^\top = I$, where the right-hand side is the identity matrix

Regarding the dimensions

- x_t is $n \times 1$, A is $n \times n$
- u_t is $k \times 1$, B is $n \times k$
- w_t is $j \times 1$, C is $n \times j$

Example 1

Consider a household budget constraint given by

$$a_{t+1} + c_t = (1 + r)a_t + y_t$$

Here a_t is assets, r is a fixed interest rate, c_t is current consumption, and y_t is current non-financial income.

If we suppose that $\{y_t\}$ is serially uncorrelated and $N(0, \sigma^2)$, then, taking $\{w_t\}$ to be standard normal, we can write the system as

$$a_{t+1} = (1 + r)a_t - c_t + \sigma w_{t+1}$$

This is clearly a special case of (72.1), with assets being the state and consumption being the control.

Example 2

One unrealistic feature of the previous model is that non-financial income has a zero mean and is often negative.

This can easily be overcome by adding a sufficiently large mean.

Hence in this example, we take $y_t = \sigma w_{t+1} + \mu$ for some positive real number μ.

Another alteration that's useful to introduce (we'll see why soon) is to change the control variable from consumption to the deviation of consumption from some "ideal" quantity $\bar{c}$.

(Most parameterizations will be such that $\bar{c}$ is large relative to the amount of consumption that is attainable in each period, and hence the household wants to increase consumption.)

For this reason, we now take our control to be $u_t := c_t - \bar{c}$.

In terms of these variables, the budget constraint $a_{t+1} = (1 + r)a_t - c_t + y_t$ becomes

$$a_{t+1} = (1 + r)a_t - u_t - \bar{c} + \sigma w_{t+1} + \mu \tag{72.2}$$

How can we write this new system in the form of equation (72.1)?

If, as in the previous example, we take a_t as the state, then we run into a problem: the law of motion contains some constant terms on the right-hand side.

This means that we are dealing with an *affine* function, not a linear one (recall *this discussion*).

Fortunately, we can easily circumvent this problem by adding an extra state variable.

In particular, if we write

$$\begin{bmatrix} a_{t+1} \\ 1 \end{bmatrix} = \begin{bmatrix} 1 + r & -\bar{c} + \mu \\ 0 & 1 \end{bmatrix} \begin{bmatrix} a_t \\ 1 \end{bmatrix} + \begin{bmatrix} -1 \\ 0 \end{bmatrix} u_t + \begin{bmatrix} \sigma \\ 0 \end{bmatrix} w_{t+1} \tag{72.3}$$

then the first row is equivalent to (72.2).

Moreover, the model is now linear and can be written in the form of (72.1) by setting

$$x_t := \begin{bmatrix} a_t \\ 1 \end{bmatrix}, \quad A := \begin{bmatrix} 1 + r & -\bar{c} + \mu \\ 0 & 1 \end{bmatrix}, \quad B := \begin{bmatrix} -1 \\ 0 \end{bmatrix}, \quad C := \begin{bmatrix} \sigma \\ 0 \end{bmatrix} \tag{72.4}$$

In effect, we've bought ourselves linearity by adding another state.

72.2.2 Preferences

In the LQ model, the aim is to minimize flow of losses, where time-t loss is given by the quadratic expression

$$x_t^\top R x_t + u_t^\top Q u_t \tag{72.5}$$

The entries in R and Q are chosen to match the specific problem being studied.

Here

- R is assumed to be $n \times n$, symmetric and nonnegative definite.
- Q is assumed to be $k \times k$, symmetric and positive definite.

Note

In fact, for many economic problems, the definiteness conditions on R and Q can be relaxed. It is sufficient that certain submatrices of R and Q be nonnegative definite. See [Hansen and Sargent, 2008] for details.

Note also that this notation is not universal: some authors let Q be the matrix associated with the state and R the matrix associated with the control.

Example 1

A very simple example that satisfies these assumptions is to take R and Q to be identity matrices so that current loss is

$$x_t^\top I x_t + u_t^\top I u_t = \|x_t\|^2 + \|u_t\|^2$$

Thus, for both the state and the control, loss is measured as squared distance from the origin.

(In fact, the general case (72.5) can also be understood in this way, but with R and Q identifying other – non-Euclidean – notions of "distance" from the zero vector.)

Intuitively, we can often think of the state x_t as representing deviation from a target, such as

- deviation of inflation from some target level
- deviation of a firm's capital stock from some desired quantity

The aim is to put the state close to the target, while using controls parsimoniously.

Example 2

In the household problem *studied above*, setting $R = 0$ and $Q = 1$ yields preferences

$$x_t^\top R x_t + u_t^\top Q u_t = u_t^2 = (c_t - \bar{c})^2$$

Under this specification, the household's current loss is the squared deviation of consumption from the ideal level $\bar{c}$.

72.3 Optimality – finite horizon

Let's now be precise about the optimization problem we wish to consider, and look at how to solve it.

72.3.1 The objective

We will begin with the finite horizon case, with terminal time $T \in \mathbb{N}$.

In this case, the aim is to choose a sequence of controls $\{u_0, \dots, u_{T-1}\}$ to minimize the objective

$$\mathbb{E}\left\{\sum_{t=0}^{T-1} \beta^t (x_t^\top R x_t + u_t^\top Q u_t) + \beta^T x_T^\top R_f x_T\right\} \tag{72.6}$$

subject to the law of motion (72.1) and initial state x_0.

The new objects introduced here are β and the matrix R_f.

The scalar β is the discount factor, while $x^\top R_f x$ gives terminal loss associated with state x.

Comments:

- We assume R_f to be $n \times n$, symmetric and nonnegative definite.
- We allow $\beta = 1$, and hence include the undiscounted case.
- x_0 may itself be random, in which case we require it to be independent of the shock sequence $\{w_1, \dots, w_T\}$.

72.3.2 Information

There's one constraint we've neglected to mention so far, which is that the decision-maker who solves this LQ problem knows only the present and the past, not the future.

To clarify this point, consider the sequence of controls $\{u_0, \dots, u_{T-1}\}$.

When choosing these controls, the decision-maker is permitted to take into account the effects of the shocks $\{w_1, \dots, w_T\}$ on the system.

However, it is typically assumed — and will be assumed here — that the time-t control u_t can be made with knowledge of past and present shocks only.

The fancy measure-theoretic way of saying this is that u_t must be measurable with respect to the σ-algebra generated by $x_0, w_1, w_2, \dots, w_t$.

This is in fact equivalent to stating that u_t can be written in the form $u_t = g_t(x_0, w_1, w_2, \dots, w_t)$ for some Borel measurable function g_t.

(Just about every function that's useful for applications is Borel measurable, so, for the purposes of intuition, you can read that last phrase as "for some function g_t")

Now note that x_t will ultimately depend on the realizations of $x_0, w_1, w_2, \dots, w_t$.

In fact, it turns out that x_t summarizes all the information about these historical shocks that the decision-maker needs to set controls optimally.

More precisely, it can be shown that any optimal control u_t can always be written as a function of the current state alone.

Hence in what follows we restrict attention to control policies (i.e., functions) of the form $u_t = g_t(x_t)$.

Actually, the preceding discussion applies to all standard dynamic programming problems.

What's special about the LQ case is that – as we shall soon see — the optimal u_t turns out to be a linear function of x_t.

72.3.3 Solution

To solve the finite horizon LQ problem we can use a dynamic programming strategy based on backward induction that is conceptually similar to the approach adopted in Shortest Paths.

For reasons that will soon become clear, we first introduce the notation $J_T(x) = x^\top R_f x$.

Now consider the problem of the decision-maker in the second to last period.

In particular, let the time be $T-1$, and suppose that the state is x_{T-1}.

The decision-maker must trade-off current and (discounted) final losses, and hence solves

$$\min_u \{x_{T-1}^\top R x_{T-1} + u^\top Q u + \beta \, \mathbb{E} J_T(A x_{T-1} + B u + C w_T)\}$$

We use u instead of u_{T-1} here to simplify notation.

At this stage, it is convenient to define the function

$$J_{T-1}(x) = \min_u \{x^\top R x + u^\top Q u + \beta \, \mathbb{E} J_T(A x + B u + C w_T)\} \tag{72.7}$$

The function J_{T-1} will be called the $T-1$ value function, and $J_{T-1}(x)$ can be thought of as representing total "loss-to-go" from state x at time $T-1$ when the decision-maker behaves optimally.

Now let's step back to $T-2$.

For a decision-maker at $T-2$, the value $J_{T-1}(x)$ plays a role analogous to that played by the terminal loss $J_T(x) = x^\top R_f x$ for the decision-maker at $T-1$.

That is, $J_{T-1}(x)$ summarizes the future loss associated with moving to state x.

The decision-maker chooses her control u to trade off current loss against future loss, where

- the next period state is $x_{T-1} = A x_{T-2} + B u + C w_{T-1}$, and hence depends on the choice of current control.
- the "cost" of landing in state x_{T-1} is $J_{T-1}(x_{T-1})$.

Her problem is therefore

$$\min_u \{x_{T-2}^\top R x_{T-2} + u^\top Q u + \beta \, \mathbb{E} J_{T-1}(A x_{T-2} + B u + C w_{T-1})\}$$

Letting

$$J_{T-2}(x) = \min_u \{x^\top R x + u^\top Q u + \beta \, \mathbb{E} J_{T-1}(A x + B u + C w_{T-1})\}$$

the pattern for backward induction is now clear.

In particular, we define a sequence of value functions $\{J_0, \ldots, J_T\}$ via

$$J_{t-1}(x) = \min_u \{x^\top R x + u^\top Q u + \beta \, \mathbb{E} J_t(A x + B u + C w_t)\} \quad \text{and} \quad J_T(x) = x^\top R_f x$$

The first equality is the Bellman equation from dynamic programming theory specialized to the finite horizon LQ problem.

Now that we have $\{J_0, \ldots, J_T\}$, we can obtain the optimal controls.

As a first step, let's find out what the value functions look like.

It turns out that every J_t has the form $J_t(x) = x^\top P_t x + d_t$ where P_t is a $n \times n$ matrix and d_t is a constant.

We can show this by induction, starting from $P_T := R_f$ and $d_T = 0$.

Using this notation, (72.7) becomes

$$J_{T-1}(x) = \min_u \{x^\top R x + u^\top Q u + \beta \, \mathbb{E}(A x + B u + C w_T)^\top P_T (A x + B u + C w_T)\} \tag{72.8}$$

To obtain the minimizer, we can take the derivative of the r.h.s. with respect to u and set it equal to zero.

Applying the relevant rules of *matrix calculus*, this gives

$$u = -(Q + \beta B^\top P_T B)^{-1} \beta B^\top P_T A x \tag{72.9}$$

Plugging this back into (72.8) and rearranging yields

$$J_{T-1}(x) = x^\top P_{T-1} x + d_{T-1}$$

where

$$P_{T-1} = R - \beta^2 A^\top P_T B (Q + \beta B^\top P_T B)^{-1} B^\top P_T A + \beta A^\top P_T A \tag{72.10}$$

and

$$d_{T-1} := \beta \, \text{trace}(C^\top P_T C) \tag{72.11}$$

(The algebra is a good exercise — we'll leave it up to you.)

If we continue working backwards in this manner, it soon becomes clear that $J_t(x) = x^\top P_t x + d_t$ as claimed, where P_t and d_t satisfy the recursions

$$P_{t-1} = R - \beta^2 A^\top P_t B (Q + \beta B^\top P_t B)^{-1} B^\top P_t A + \beta A^\top P_t A \quad \text{with} \quad P_T = R_f \tag{72.12}$$

and

$$d_{t-1} = \beta (d_t + \text{trace}(C^\top P_t C)) \quad \text{with} \quad d_T = 0 \tag{72.13}$$

Recalling (72.9), the minimizers from these backward steps are

$$u_t = -F_t x_t \quad \text{where} \quad F_t := (Q + \beta B^\top P_{t+1} B)^{-1} \beta B^\top P_{t+1} A \tag{72.14}$$

These are the linear optimal control policies we *discussed above*.

In particular, the sequence of controls given by (72.14) and (72.1) solves our finite horizon LQ problem.

Rephrasing this more precisely, the sequence $\{u_0, \ldots, u_{T-1}\}$ given by

$$u_t = -F_t x_t \quad \text{with} \quad x_{t+1} = (A - BF_t) x_t + C w_{t+1} \tag{72.15}$$

for $t = 0, \ldots, T-1$ attains the minimum of (72.6) subject to our constraints.

72.4 Implementation

We will use code from lqcontrol.py in QuantEcon.py to solve finite and infinite horizon linear quadratic control problems.

In the module, the various updating, simulation and fixed point methods are wrapped in a class called `LQ`, which includes

- Instance data:
  - The required parameters Q, R, A, B and optional parameters C, β, T, R_f, N specifying a given LQ model
    - set T and R_f to `None` in the infinite horizon case
    - set `C = None` (or zero) in the deterministic case
  - the value function and policy data

    * d_t, P_t, F_t in the finite horizon case
    * d, P, F in the infinite horizon case

- Methods:
  - `update_values` — shifts d_t, P_t, F_t to their $t-1$ values via (72.12), (72.13) and (72.14)
  - `stationary_values` — computes P, d, F in the infinite horizon case
  - `compute_sequence` —- simulates the dynamics of x_t, u_t, w_t given x_0 and assuming standard normal shocks

72.4.1 An application

Early Keynesian models assumed that households have a constant marginal propensity to consume from current income.

Data contradicted the constancy of the marginal propensity to consume.

In response, Milton Friedman, Franco Modigliani and others built models based on a consumer's preference for an intertemporally smooth consumption stream.

(See, for example, [Friedman, 1956] or [Modigliani and Brumberg, 1954].)

One property of those models is that households purchase and sell financial assets to make consumption streams smoother than income streams.

The household savings problem *outlined above* captures these ideas.

The optimization problem for the household is to choose a consumption sequence in order to minimize

$$\mathbb{E}\left\{\sum_{t=0}^{T-1}\beta^t(c_t-\bar{c})^2+\beta^T q a_T^2\right\} \tag{72.16}$$

subject to the sequence of budget constraints $a_{t+1}=(1+r)a_t-c_t+y_t,\ t\geq 0$.

Here q is a large positive constant, the role of which is to induce the consumer to target zero debt at the end of her life.

(Without such a constraint, the optimal choice is to choose $c_t=\bar{c}$ in each period, letting assets adjust accordingly.)

As before we set $y_t=\sigma w_{t+1}+\mu$ and $u_t:=c_t-\bar{c}$, after which the constraint can be written as in (72.2).

We saw how this constraint could be manipulated into the LQ formulation $x_{t+1}=Ax_t+Bu_t+Cw_{t+1}$ by setting $x_t=(a_t\ 1)^\top$ and using the definitions in (72.4).

To match with this state and control, the objective function (72.16) can be written in the form of (72.6) by choosing

$$Q:=1,\quad R:=\begin{bmatrix}0&0\\0&0\end{bmatrix},\quad\text{and}\quad R_f:=\begin{bmatrix}q&0\\0&0\end{bmatrix}$$

Now that the problem is expressed in LQ form, we can proceed to the solution by applying (72.12) and (72.14).

After generating shocks $\{w_1,\dots,w_T\}$, the dynamics for assets and consumption can be simulated via (72.15).

The following figure was computed using $r=0.05, \beta=1/(1+r), \bar{c}=2, \mu=1, \sigma=0.25, T=45$ and $q=10^6$.

The shocks $\{w_t\}$ were taken to be IID and standard normal.

```
class LQModel(NamedTuple):
    r: float              # interest rate
    β: float              # discount factor
    T: int                # time horizon
    c_bar: float          # target consumption level
```

(continues on next page)

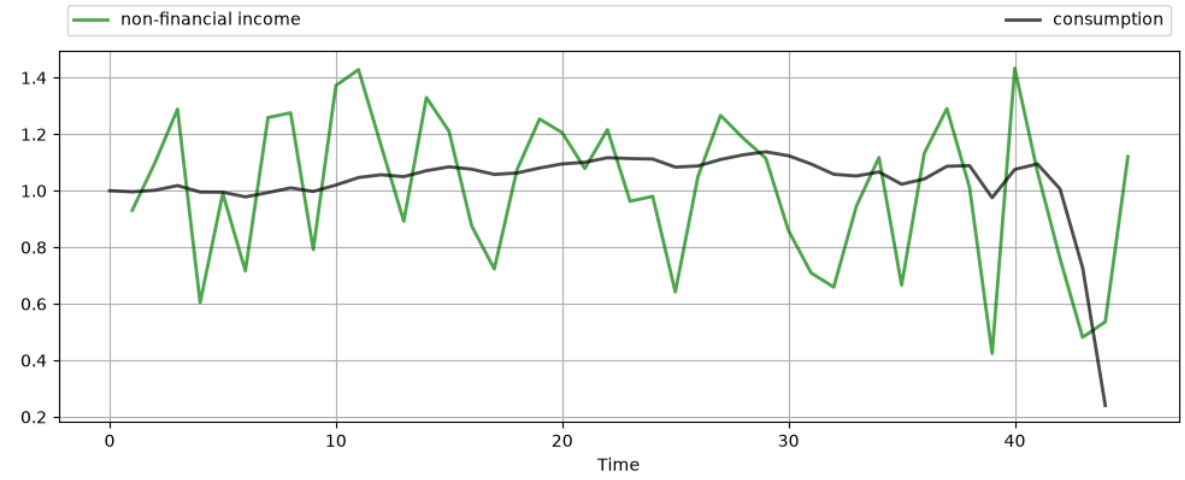
(continued from previous page)

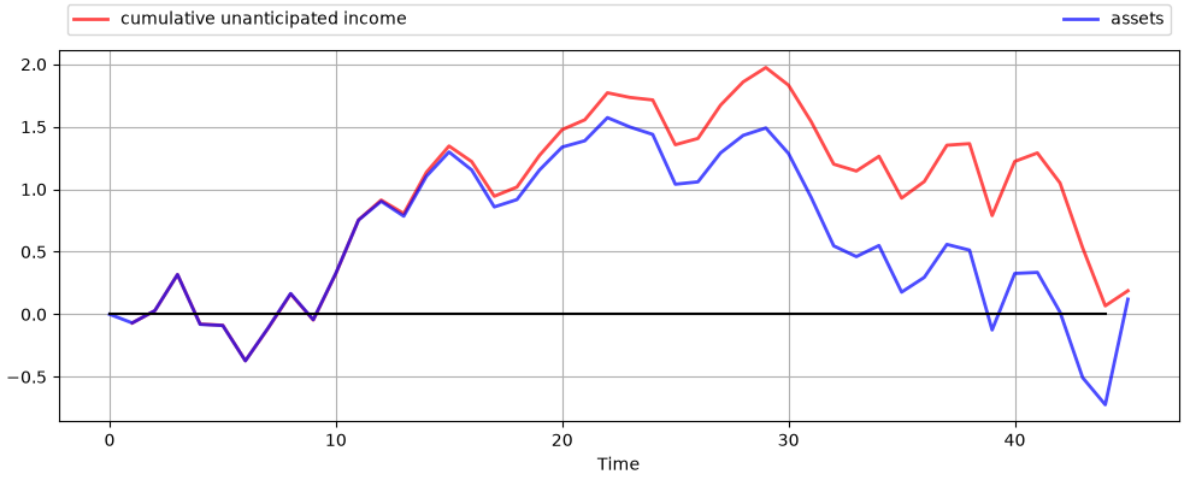
```
    σ: float               # income shock standard deviation
    μ: float               # mean income level
    q: float               # terminal penalty weight
    Q: float               # control penalty matrix
    R: np.ndarray          # state penalty matrix
    Rf: np.ndarray         # terminal state penalty matrix
    A: np.ndarray          # state transition matrix
    B: np.ndarray          # control matrix
    C: np.ndarray          # shock matrix


def create_lq_model(r=0.05,
                    T=45,
                    c_bar=2,
                    σ=0.25,
                    μ=1,
                    q=1e6):
    β = 1 / (1 + r)

    # Formulate as an LQ problem
    Q = 1
    R = np.zeros((2, 2))
    Rf = np.zeros((2, 2))
    Rf[0, 0] = q
    A = np.array([[1 + r, -c_bar + μ],
                  [0,      1]])
    B = np.array([[-1],
                  [ 0]])
    C = np.array([[σ],
                  [0]])

    return LQModel(r=r, β=β, T=T, c_bar=c_bar, σ=σ, μ=μ,
                   q=q, Q=Q, R=R, Rf=Rf, A=A, B=B, C=C)

def simulate_and_plot(model):
    # Unpack model
    r, β, T, c_bar, σ, μ, q, Q, R, Rf, A, B, C = model

    # Compute solutions and simulate
    lq = LQ(Q, R, A, B, C, beta=β, T=T, Rf=Rf)
    x0 = (0, 1)
    xp, up, wp = lq.compute_sequence(x0)

    # Convert back to assets, consumption and income
    assets = xp[0, :]            # a_t
    c = up.flatten() + c_bar     # c_t
    income = σ * wp[0, 1:] + μ   # y_t

    # Plot results
    n_rows = 2
    fig, axes = plt.subplots(n_rows, 1, figsize=(12, 10))

    plt.subplots_adjust(hspace=0.5)

    bbox = (0., 1.02, 1., .102)
    legend_args = {'bbox_to_anchor': bbox, 'loc': 3, 'mode': 'expand'}
    p_args = {'lw': 2, 'alpha': 0.7}
```

(continues on next page)

(continued from previous page)

```
    axes[0].plot(range(1, T+1), income, 'g-', label="non-financial income",
                 **p_args)
    axes[0].plot(range(T), c, 'k-', label="consumption", **p_args)

    axes[1].plot(range(1, T+1), np.cumsum(income - μ), 'r-',
                 label="cumulative unanticipated income", **p_args)
    axes[1].plot(range(T+1), assets, 'b-', label="assets", **p_args)
    axes[1].plot(range(T), np.zeros(T), 'k-')

    for ax in axes:
        ax.grid()
        ax.set_xlabel('Time')
        ax.legend(ncol=2, **legend_args)

    plt.show()

# Create model instance and simulate
model = create_lq_model()
simulate_and_plot(model)
```





The top panel shows the time path of consumption c_t and income y_t in the simulation.

As anticipated by the discussion on consumption smoothing, the time path of consumption is much smoother than that for income.

(But note that consumption becomes more irregular towards the end of life, when the zero final asset requirement impinges more on consumption choices.)

The second panel in the figure shows that the time path of assets a_t is closely correlated with cumulative unanticipated income, where the latter is defined as

$$z_t := \sum_{j=0}^{t} \sigma w_t$$

A key message is that unanticipated windfall gains are saved rather than consumed, while unanticipated negative shocks are met by reducing assets.

(Again, this relationship breaks down towards the end of life due to the zero final asset requirement.)

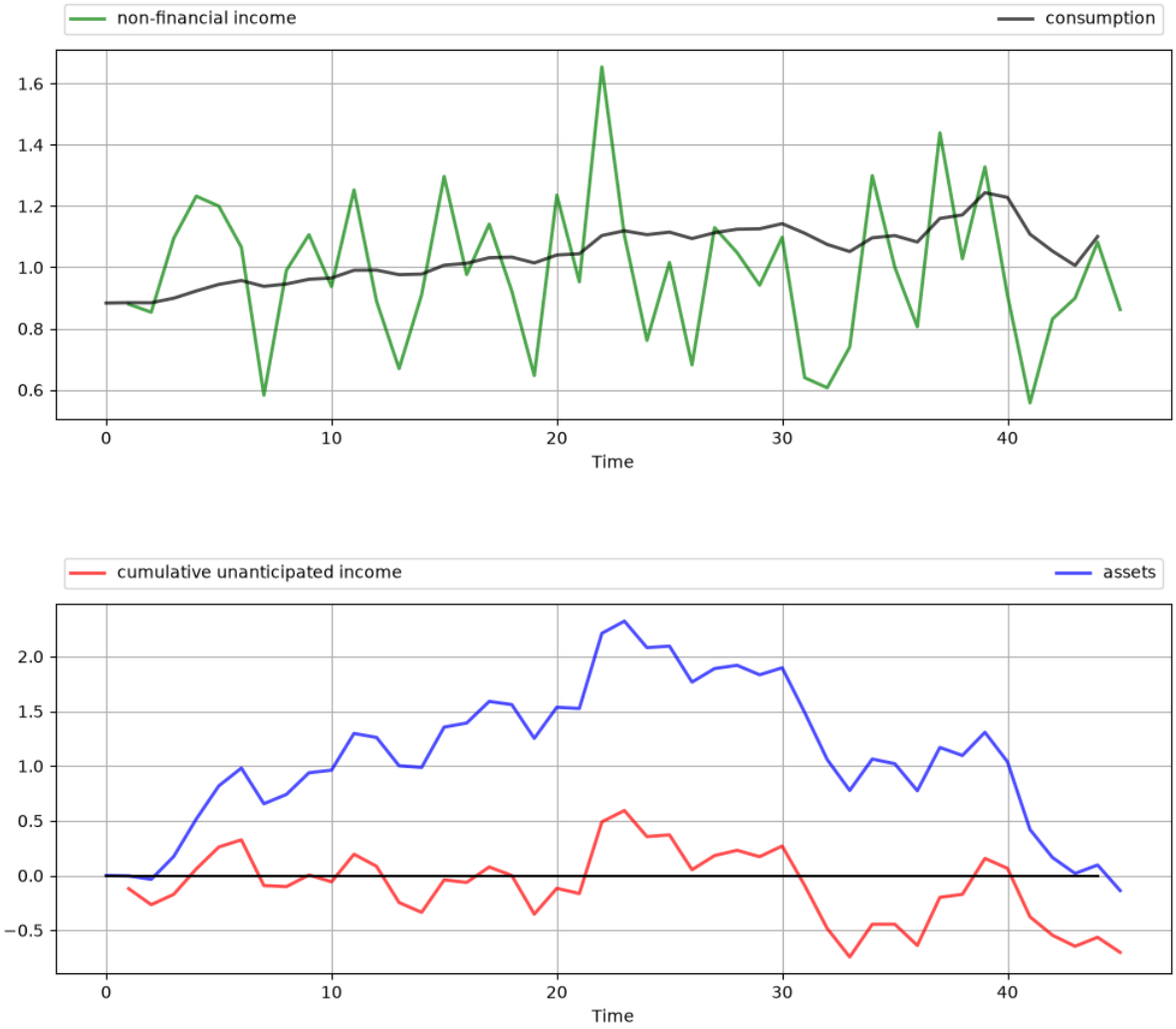
These results are relatively robust to changes in parameters.

For example, let's increase β from $1/(1+r) \approx 0.952$ to 0.96 while keeping other parameters fixed.

This consumer is slightly more patient than the last one, and hence puts relatively more weight on later consumption values.

Since `LQModel` is a `NamedTuple`, we can obtain the new model by copying the old one with a single field changed.

```
simulate_and_plot(model._replace(β=0.96))
```



We now have a slowly rising consumption stream and a hump-shaped build-up of assets in the middle periods to fund rising consumption.

However, the essential features are the same: consumption is smooth relative to income, and assets are strongly positively correlated with cumulative unanticipated income.

72.5 Extensions and comments

Let's now consider a number of standard extensions to the LQ problem treated above.

72.5.1 Time-varying parameters

In some settings, it can be desirable to allow A, B, C, R and Q to depend on t.

For the sake of simplicity, we've chosen not to treat this extension in our implementation given below.

However, the loss of generality is not as large as you might first imagine.

In fact, we can tackle many models with time-varying parameters by suitable choice of state variables.

One illustration is given *below*.

For further examples and a more systematic treatment, see [Hansen and Sargent, 2013], section 2.4.

72.5.2 Adding a cross-product term

In some LQ problems, preferences include a cross-product term $u_t^\top N x_t$, so that the objective function becomes

$$\mathbb{E}\left\{\sum_{t=0}^{T-1}\beta^t(x_t^\top R x_t + u_t^\top Q u_t + 2u_t^\top N x_t) + \beta^T x_T^\top R_f x_T\right\} \tag{72.17}$$

Our results extend to this case in a straightforward way.

The sequence $\{P_t\}$ from (72.12) becomes

$$P_{t-1} = R - (\beta B^\top P_t A + N)^\top (Q + \beta B^\top P_t B)^{-1}(\beta B^\top P_t A + N) + \beta A^\top P_t A \quad \text{with} \quad P_T = R_f \tag{72.18}$$

The policies in (72.14) are modified to

$$u_t = -F_t x_t \quad \text{where} \quad F_t := (Q + \beta B^\top P_{t+1} B)^{-1}(\beta B^\top P_{t+1} A + N) \tag{72.19}$$

The sequence $\{d_t\}$ is unchanged from (72.13).

We leave interested readers to confirm these results (the calculations are long but not overly difficult).

72.5.3 Infinite horizon

Finally, we consider the infinite horizon case, with *cross-product term*, unchanged dynamics and objective function given by

$$\mathbb{E}\left\{\sum_{t=0}^{\infty}\beta^t(x_t^\top R x_t + u_t^\top Q u_t + 2u_t^\top N x_t)\right\} \tag{72.20}$$

In the infinite horizon case, optimal policies can depend on time only if time itself is a component of the state vector x_t.

In other words, there exists a fixed matrix F such that $u_t = -Fx_t$ for all t.

That decision rules are constant over time is intuitive — after all, the decision-maker faces the same infinite horizon at every stage, with only the current state changing.

Not surprisingly, P and d are also constant.

The stationary matrix P is the solution to the discrete-time algebraic Riccati equation.

$$P = R - (\beta B^\top P A + N)^\top (Q + \beta B^\top P B)^{-1}(\beta B^\top P A + N) + \beta A^\top P A \tag{72.21}$$

Equation (72.21) is also called the *LQ Bellman equation*, and the map that sends a given P into the right-hand side of (72.21) is called the *LQ Bellman operator*.

The stationary optimal policy for this model is

$$u = -Fx \quad \text{where} \quad F = (Q + \beta B^\top P B)^{-1}(\beta B^\top P A + N) \tag{72.22}$$

The sequence $\{d_t\}$ from (72.13) is replaced by the constant value

$$d := \text{trace}(C^\top P C)\frac{\beta}{1-\beta} \tag{72.23}$$

The state evolves according to the time-homogeneous process $x_{t+1} = (A - BF)x_t + Cw_{t+1}$.

An example infinite horizon problem is treated *below*.

72.5.4 Certainty equivalence

Linear quadratic control problems of the class discussed above have the property of *certainty equivalence*.

By this, we mean that the optimal policy F is not affected by the parameters in C, which specify the shock process.

This can be confirmed by inspecting (72.22) or (72.19).

It follows that we can ignore uncertainty when solving for optimal behavior, and plug it back in when examining optimal state dynamics.

72.6 Further applications

72.6.1 Application 1: Age-dependent income process

Previously we studied a permanent income model that generated consumption smoothing.

One unrealistic feature of that model is the assumption that the mean of the random income process does not depend on the consumer's age.

A more realistic income profile is one that rises in early working life, peaks towards the middle and maybe declines toward the end of working life and falls more during retirement.

In this section, we will model this rise and fall as a symmetric inverted "U" using a polynomial in age.

As before, the consumer seeks to minimize

$$\mathbb{E}\left\{\sum_{t=0}^{T-1}\beta^t(c_t - \bar{c})^2 + \beta^T q a_T^2\right\} \tag{72.24}$$

subject to $a_{t+1} = (1+r)a_t - c_t + y_t,\ t \geq 0$.

For income we now take $y_t = p(t) + \sigma w_{t+1}$ where $p(t) := m_0 + m_1 t + m_2 t^2$.

(In *the next section* we employ some tricks to implement a more sophisticated model.)

The coefficients m_0, m_1, m_2 are chosen such that $p(0) = 0, p(T/2) = \mu$, and $p(T) = 0$.

You can confirm that the specification $m_0 = 0, m_1 = T\mu/(T/2)^2, m_2 = -\mu/(T/2)^2$ satisfies these constraints.

To put this into an LQ setting, consider the budget constraint, which becomes

$$a_{t+1} = (1+r)a_t - u_t - \bar{c} + m_1 t + m_2 t^2 + \sigma w_{t+1} \tag{72.25}$$

The fact that a_{t+1} is a linear function of $(a_t, 1, t, t^2)$ suggests taking these four variables as the state vector x_t.

Once a good choice of state and control (recall $u_t = c_t - \bar{c}$) has been made, the remaining specifications fall into place relatively easily.

Thus, for the dynamics we set

$$x_t := \begin{bmatrix} a_t \\ 1 \\ t \\ t^2 \end{bmatrix}, \quad A := \begin{bmatrix} 1+r & -\bar{c} & m_1 & m_2 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 2 & 1 \end{bmatrix}, \quad B := \begin{bmatrix} -1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad C := \begin{bmatrix} \sigma \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{72.26}$$

If you expand the expression $x_{t+1} = Ax_t + Bu_t + Cw_{t+1}$ using this specification, you will find that assets follow (72.25) as desired and that the other state variables also update appropriately.

To implement preference specification (72.24) we take

$$Q := 1, \quad R := \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad \text{and} \quad R_f := \begin{bmatrix} q & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{72.27}$$

The next figure shows a simulation of consumption and assets computed using the `compute_sequence` method of `lqcontrol.py` with initial assets set to zero.

Once again, smooth consumption is a dominant feature of the sample paths.

The asset path exhibits dynamics consistent with standard life cycle theory.

Exercise 72.7.1 gives the full set of parameters used here and asks you to replicate the figure.

72.6.2 Application 2: A permanent income model with retirement

In the *previous application*, we generated income dynamics with an inverted U shape using polynomials and placed them in an LQ framework.

It is arguably the case that this income process still contains unrealistic features.

A more common earning profile is where

1. income grows over working life, fluctuating around an increasing trend, with growth flattening off in later years
2. retirement follows, with lower but relatively stable (non-financial) income

Letting K be the retirement date, we can express these income dynamics by

$$y_t = \begin{cases} p(t) + \sigma w_{t+1} & \text{if } t \leq K \\ s & \text{otherwise} \end{cases} \tag{72.28}$$

Here

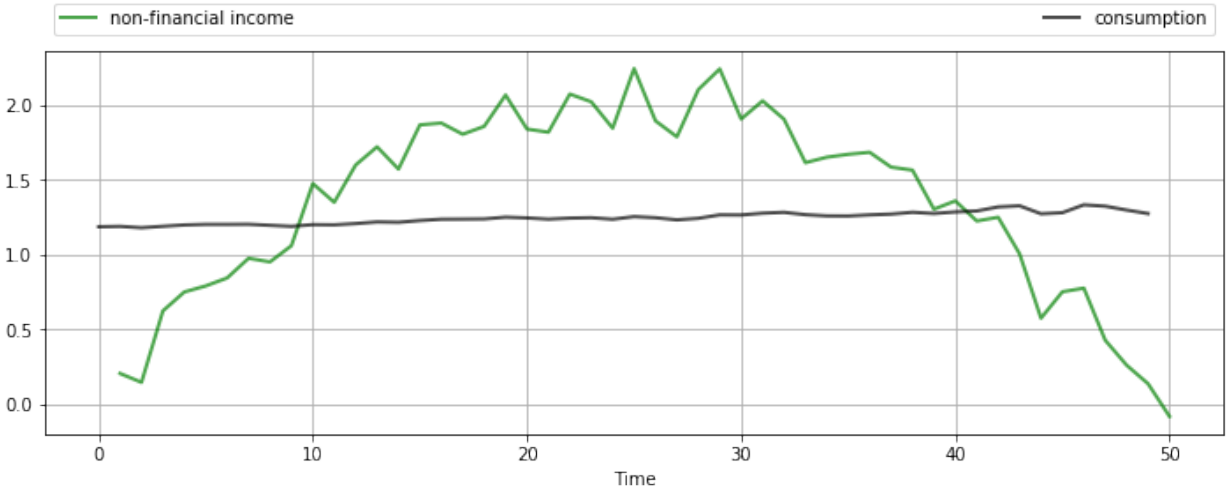
- $p(t) := m_1 t + m_2 t^2$ with the coefficients m_1, m_2 chosen such that $p(K) = \mu$ and $p(0) = p(2K) = 0$
- s is retirement income

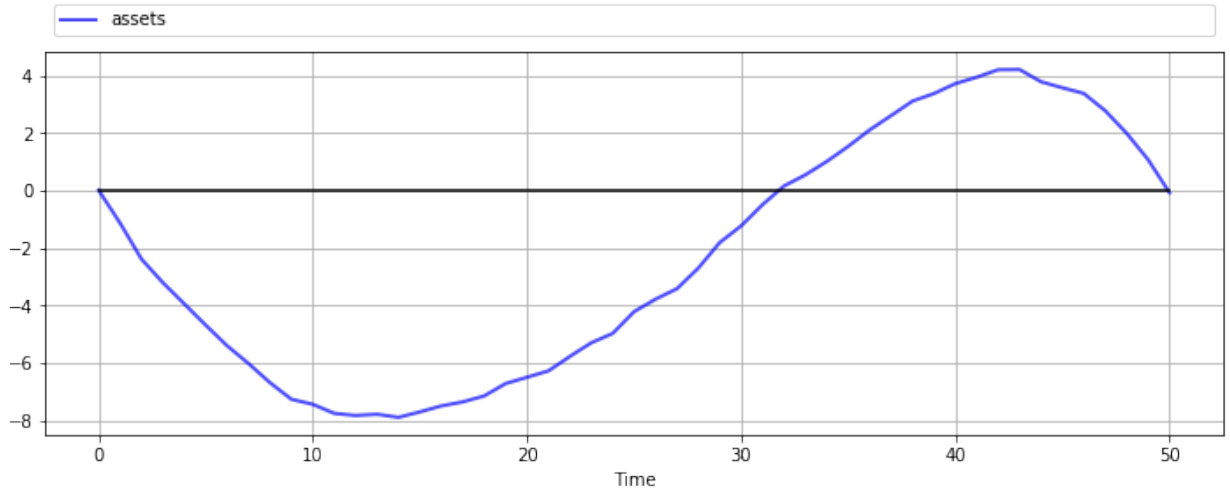
We suppose that preferences are unchanged and given by (72.16).

The budget constraint is also unchanged and given by $a_{t+1} = (1+r)a_t - c_t + y_t$.

Our aim is to solve this problem and simulate paths using the LQ techniques described in this lecture.

In fact, this is a nontrivial problem, as the kink in the dynamics (72.28) at K makes it very difficult to express the law of motion as a fixed-coefficient linear system.


non-financial income
consumption
2.0
1.5
1.0
0.5
0.0
0
10
20
30
40
50
Time


assets
4
2
0
−2
−4
−6
−8
0
10
20
30
40
50
Time

However, we can still use our LQ methods here by suitably linking two-component LQ problems.

These two LQ problems describe the consumer's behavior during her working life (`lq_working`) and retirement (`lq_retired`).

(This is possible because, in the two separate periods of life, the respective income processes [polynomial trend and constant] each fit the LQ framework.)

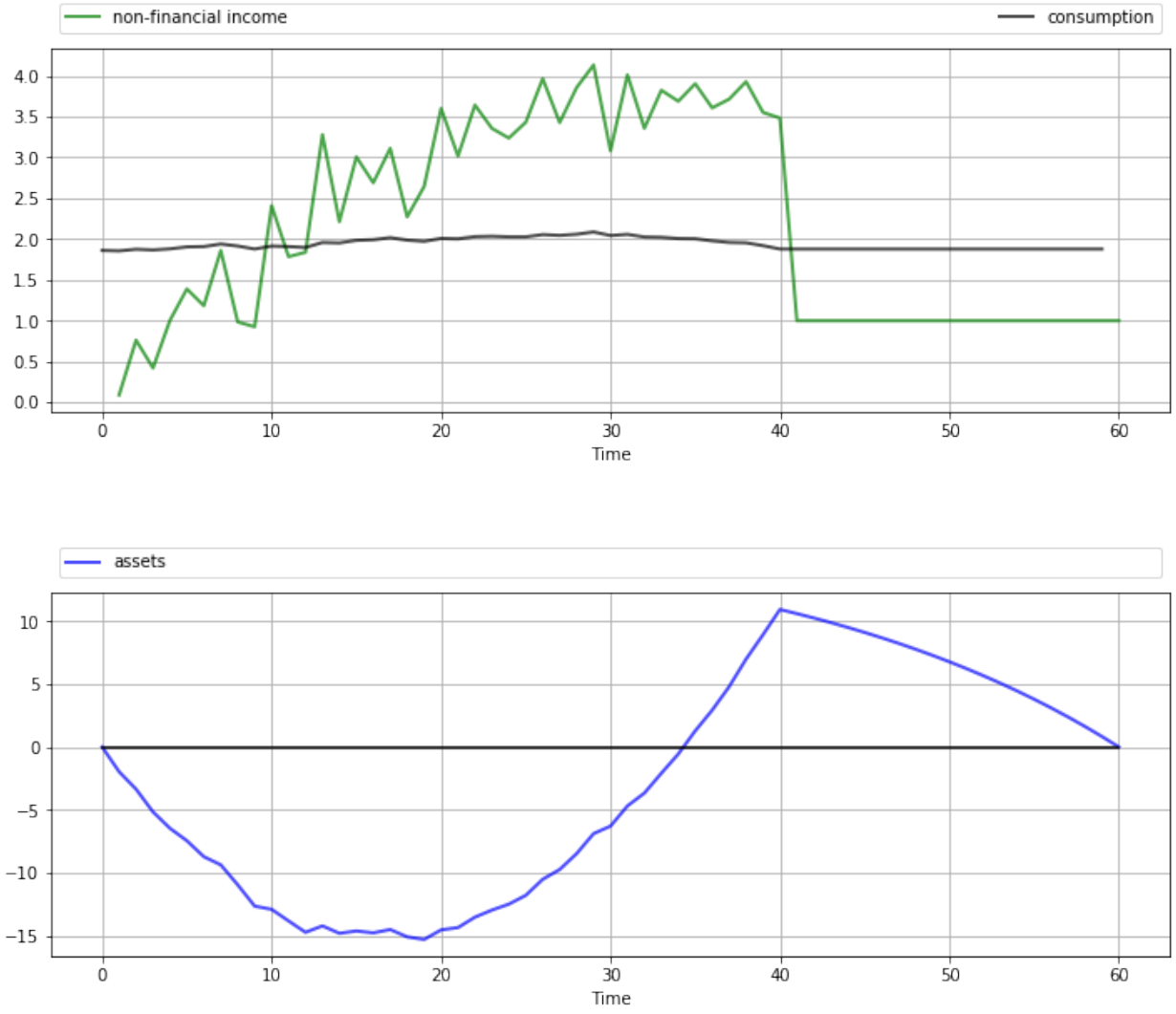
The basic idea is that although the whole problem is not a single time-invariant LQ problem, it is still a dynamic programming problem, and hence we can use appropriate Bellman equations at every stage.

Based on this logic, we can

1. solve `lq_retired` by the usual backward induction procedure, iterating back to the start of retirement.
2. take the start-of-retirement value function generated by this process, and use it as the terminal condition R_f to feed into the `lq_working` specification.
3. solve `lq_working` by backward induction from this choice of R_f, iterating back to the start of working life.

This process gives the entire life-time sequence of value functions and optimal policies.

The next figure shows one simulation based on this procedure.



The full set of parameters used in the simulation is discussed in Exercise 72.7.2, where you are asked to replicate the figure.

Once again, the dominant feature observable in the simulation is consumption smoothing.

The asset path fits well with standard life cycle theory, with dissaving early in life followed by later saving.

Assets peak at retirement and subsequently decline.

72.6.3 Application 3: Monopoly with adjustment costs

Consider a monopolist facing stochastic inverse demand function

$$p_t = a_0 - a_1 q_t + d_t$$

Here q_t is output, and the demand shock d_t follows

$$d_{t+1} = \rho d_t + \sigma w_{t+1}$$

where $\{w_t\}$ is IID and standard normal.

The monopolist maximizes the expected discounted sum of present and future profits

$$\mathbb{E}\left\{\sum_{t=0}^{\infty} \beta^t \pi_t\right\} \quad \text{where} \quad \pi_t := p_t q_t - c q_t - \gamma (q_{t+1} - q_t)^2 \tag{72.29}$$

Here

- $\gamma (q_{t+1} - q_t)^2$ represents adjustment costs
- c is average cost of production

This can be formulated as an LQ problem and then solved and simulated, but first let's study the problem and try to get some intuition.

One way to start thinking about the problem is to consider what would happen if $\gamma = 0$.

Without adjustment costs there is no intertemporal trade-off, so the monopolist will choose output to maximize current profit in each period.

It's not difficult to show that profit-maximizing output is

$$\bar{q}_t := \frac{a_0 - c + d_t}{2a_1}$$

In light of this discussion, what we might expect for general γ is that

- if γ is close to zero, then q_t will track the time path of $\bar{q}_t$ relatively closely.
- if γ is larger, then q_t will be smoother than $\bar{q}_t$, as the monopolist seeks to avoid adjustment costs.

This intuition turns out to be correct.

The following figures show simulations produced by solving the corresponding LQ problem.

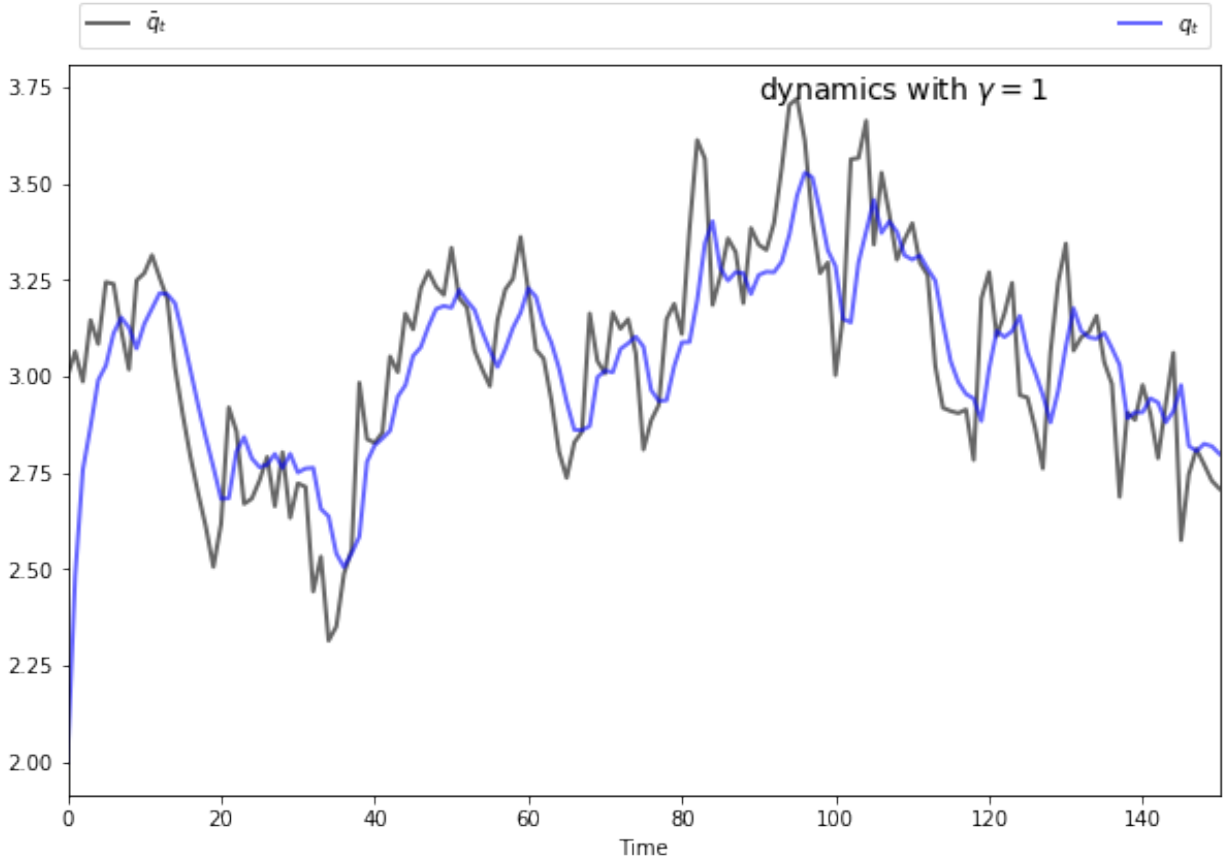
The only difference in parameters across the figures is the size of γ

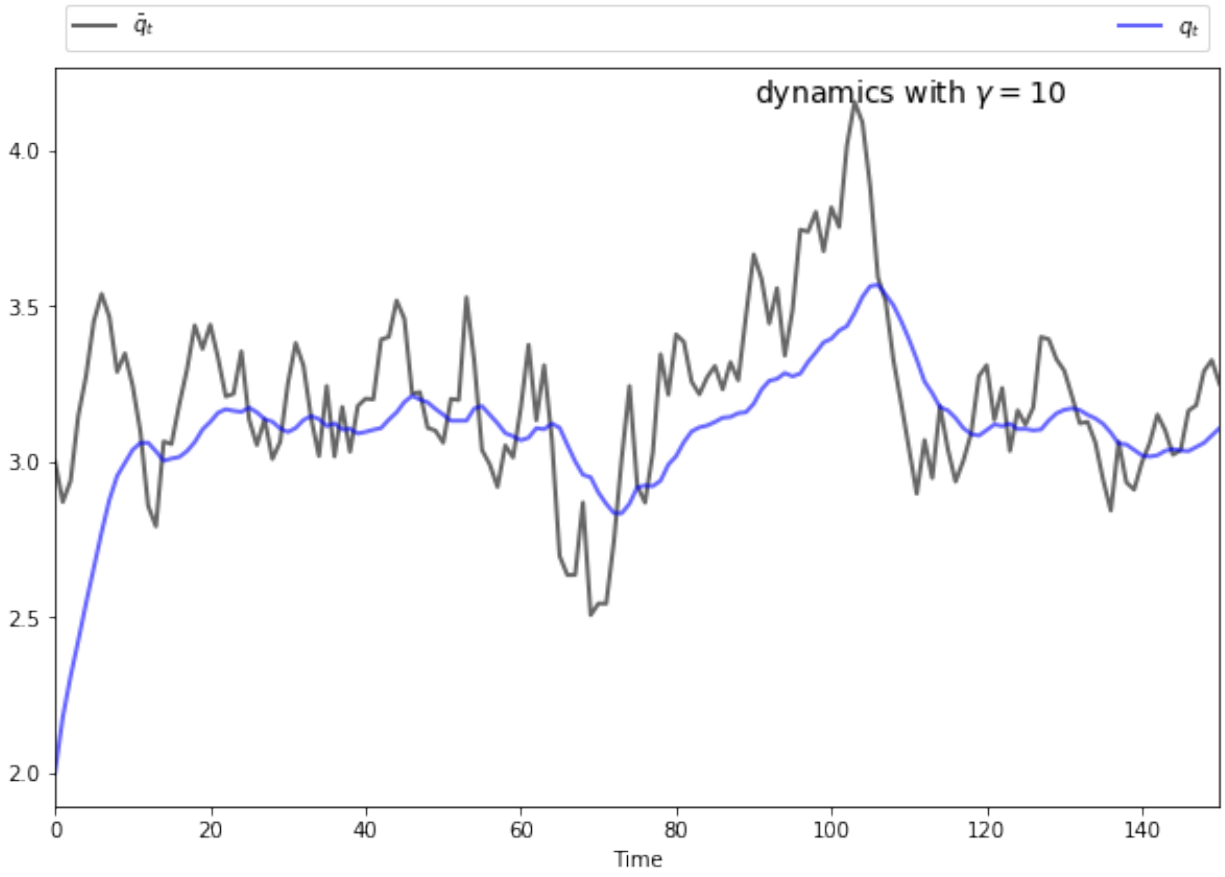
To produce these figures we converted the monopolist problem into an LQ problem.

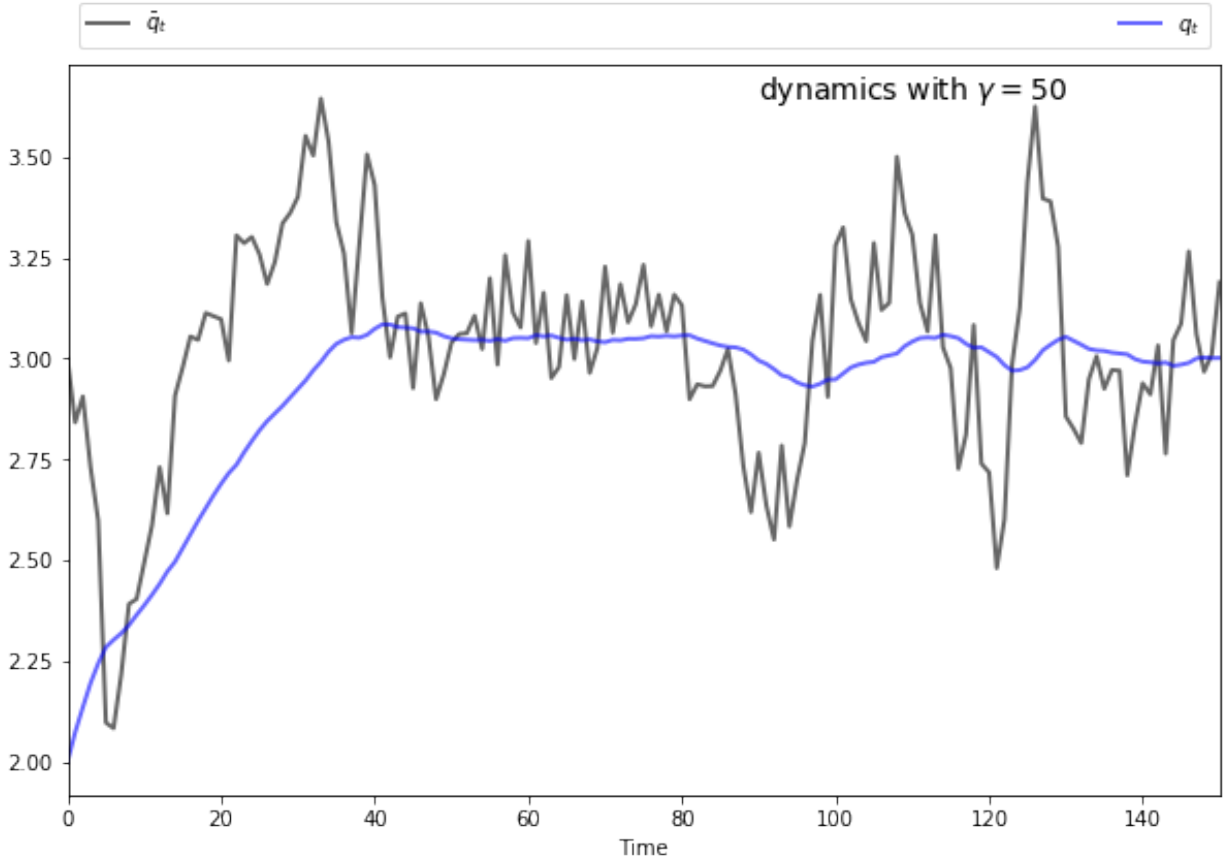
The key to this conversion is to choose the right state — which can be a bit of an art.

Here we take $x_t = (\bar{q}_t \;\; q_t \;\; 1)^\top$, while the control is chosen as $u_t = q_{t+1} - q_t$.

We also manipulated the profit function slightly.


$\bar{q}_t$
q_t
dynamics with $\gamma = 1$
3.75
3.50
3.25
3.00
2.75
2.50
2.25
2.00
0
20
40
60
80
100
120
140
Time


$\tilde{q}_t$
q_t
dynamics with $\gamma = 10$
4.0
3.5
3.0
2.5
2.0
0
20
40
60
80
100
120
140
Time


$\bar{q}_t$
q_t
dynamics with $\gamma = 50$
3.50
3.25
3.00
2.75
2.50
2.25
2.00
0
20
40
60
80
100
120
140
Time

In (72.29), current profits are $\pi_t := p_t q_t - c q_t - \gamma (q_{t+1} - q_t)^2$.

Let's now replace π_t in (72.29) with $\hat{\pi}_t := \pi_t - a_1 \bar{q}_t^2$.

This makes no difference to the solution, since $a_1 \bar{q}_t^2$ does not depend on the controls.

(In fact, we are just adding a constant term to (72.29), and optimizers are not affected by constant terms.)

The reason for making this substitution is that, as you will be able to verify, $\hat{\pi}_t$ reduces to the simple quadratic

$$\hat{\pi}_t = -a_1 (q_t - \bar{q}_t)^2 - \gamma u_t^2$$

After negation to convert to a minimization problem, the objective becomes

$$\min \mathbb{E} \sum_{t=0}^{\infty} \beta^t \left\{ a_1 (q_t - \bar{q}_t)^2 + \gamma u_t^2 \right\} \tag{72.30}$$

It's now relatively straightforward to find R and Q such that (72.30) can be written as (72.20).

Furthermore, the matrices A, B and C from (72.1) can be found by writing down the dynamics of each element of the state.

Exercise 72.7.3 asks you to complete this process, and reproduce the preceding figures.

72.7 Exercises

Exercise 72.7.1

Replicate the figure with polynomial income *shown above*.

The parameters are $r = 0.05, \beta = 1/(1+r), \bar{c} = 1.5, \mu = 2, \sigma = 0.15, T = 50$ and $q = 10^4$.

Solution

Here's one solution.

We use some fancy plot commands to get a certain style – feel free to use simpler ones.

The model is an LQ permanent income / life-cycle model with hump-shaped income

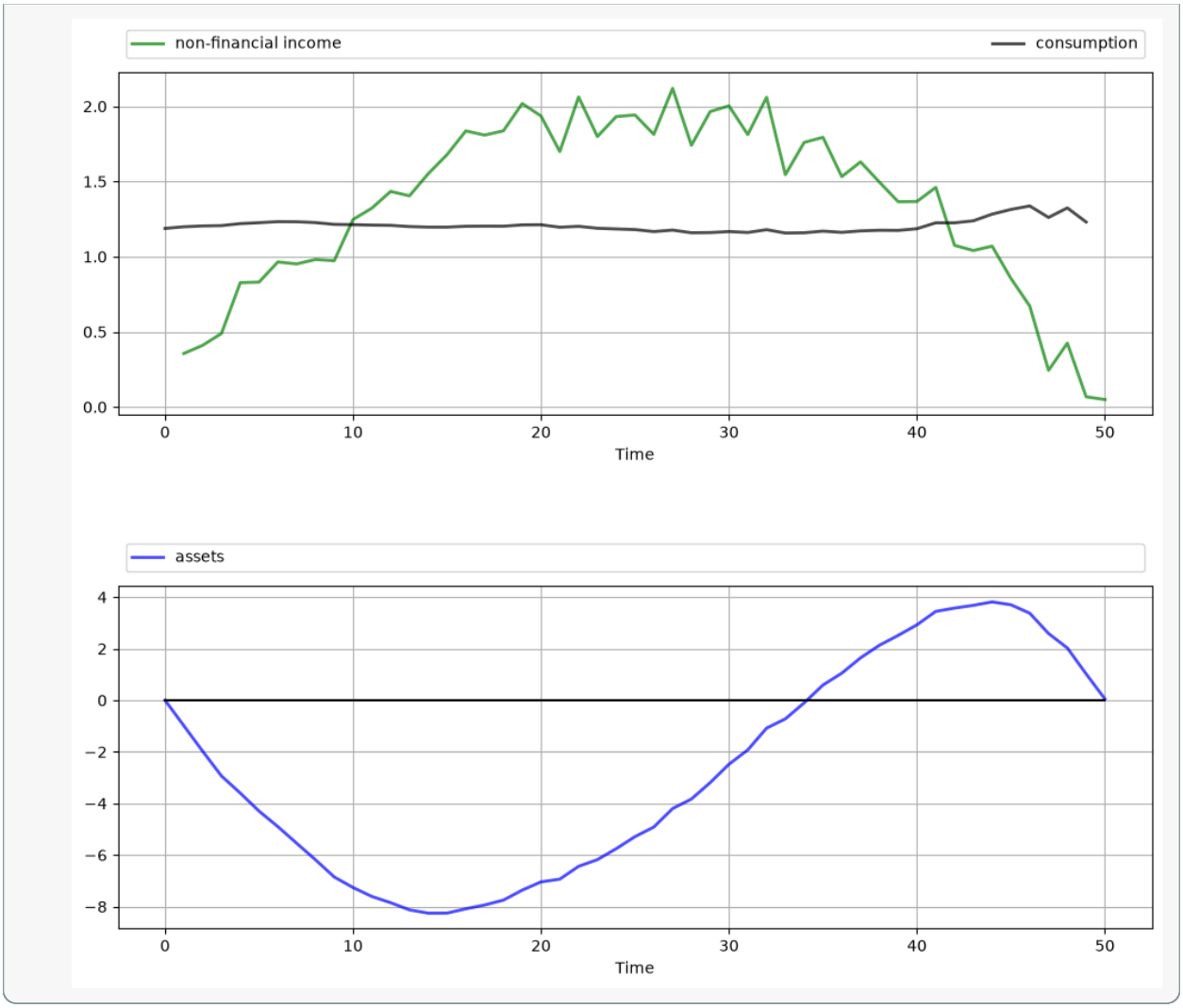
$$y_t = m_1 t + m_2 t^2 + \sigma w_{t+1}$$

where $\{w_t\}$ is IID $N(0,1)$ and the coefficients m_1 and m_2 are chosen so that $p(t) = m_1 t + m_2 t^2$ has an inverted U shape with

- $p(0) = 0, p(T/2) = \mu$, and
- $p(T) = 0$

```
# Model parameters
r = 0.05
β = 1/(1 + r)
T = 50
c_bar = 1.5
σ = 0.15
μ = 2
q = 1e4
```

```
m1 = T * (μ/(T/2)**2)
m2 = -(μ/(T/2)**2)

# Formulate as an LQ problem
Q = 1
R = np.zeros((4, 4))
Rf = np.zeros((4, 4))
Rf[0, 0] = q
A = [[1 + r, -c_bar, m1, m2],
     [0,           1,  0,  0],
     [0,           1,  1,  0],
     [0,           1,  2,  1]]
B = [[-1],
     [ 0],
     [ 0],
     [ 0]]
C = [[σ],
     [0],
     [0],
     [0]]

# Compute solutions and simulate
lq = LQ(Q, R, A, B, C, beta=β, T=T, Rf=Rf)
x0 = (0, 1, 0, 0)
xp, up, wp = lq.compute_sequence(x0)

# Convert results back to assets, consumption and income
ap = xp[0, :]               # Assets
c = up.flatten() + c_bar    # Consumption
time = np.arange(1, T+1)
income = σ * wp[0, 1:] + m1 * time + m2 * time**2  # Income


# Plot results
n_rows = 2
fig, axes = plt.subplots(n_rows, 1, figsize=(12, 10))

plt.subplots_adjust(hspace=0.5)

bbox = (0., 1.02, 1., .102)
legend_args = {'bbox_to_anchor': bbox, 'loc': 3, 'mode': 'expand'}
p_args = {'lw': 2, 'alpha': 0.7}

axes[0].plot(range(1, T+1), income, 'g-', label="non-financial income",
             **p_args)
axes[0].plot(range(T), c, 'k-', label="consumption", **p_args)

axes[1].plot(range(T+1), ap.flatten(), 'b-', label="assets", **p_args)
axes[1].plot(range(T+1), np.zeros(T+1), 'k-')

for ax in axes:
    ax.grid()
    ax.set_xlabel('Time')
    ax.legend(ncol=2, **legend_args)

plt.show()
```



Exercise 72.7.2

Replicate the figure on work and retirement *shown above*.

The parameters are $r = 0.05, \beta = 1/(1+r), \bar{c} = 4, \mu = 4, \sigma = 0.35, K = 40, T = 60, s = 1$ and $q = 10^4$.

To understand the overall procedure, carefully read the section containing that figure.

Hint

First, in order to make our approach work, we must ensure that both LQ problems have the same state variables and control.
As with previous applications, the control can be set to $u_t = c_t - \bar{c}$.
For `lq_working`, x_t, A, B, C can be chosen as in (72.26).

- Recall that m_1, m_2 are chosen so that $p(K) = \mu$ and $p(2K) = 0$.

For `lq_retired`, use the same definition of x_t and u_t, but modify A, B, C to correspond to constant income $y_t = s$.
For `lq_retired`, set preferences as in (72.27).
For `lq_working`, preferences are the same, except that R_f should be replaced by the final value function that emerges from iterating `lq_retired` back to the start of retirement.
With some careful footwork, the simulation can be generated by patching together the simulations from these two separate models.

Solution

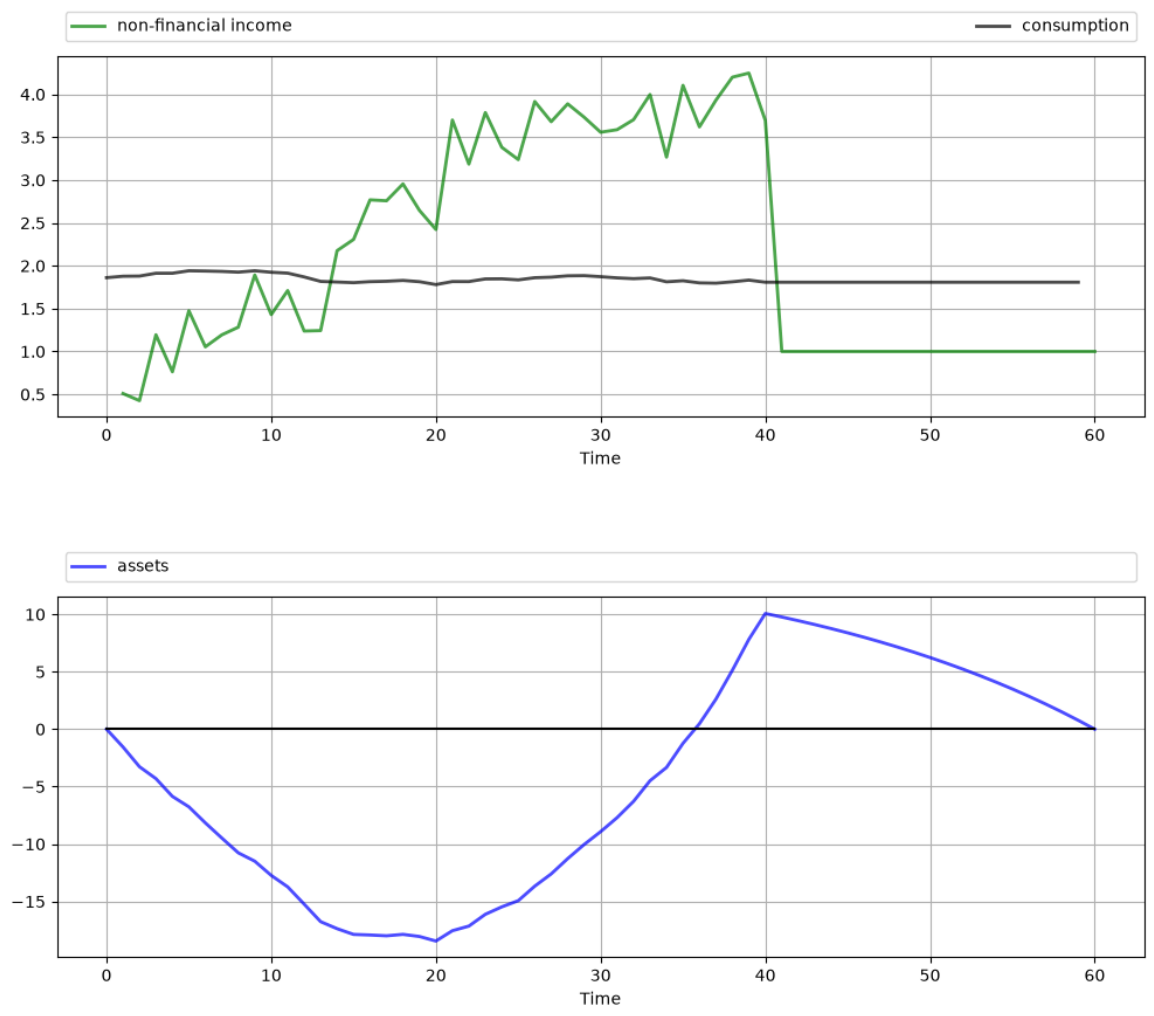
This is a permanent income / life-cycle model with polynomial growth in income over working life followed by a fixed retirement income.

The model is solved by combining two LQ programming problems as described in the lecture.

```
# Model parameters
r = 0.05
β = 1/(1 + r)
T = 60
K = 40
c_bar = 4
σ = 0.35
μ = 4
q = 1e4
s = 1
m1 = 2 * μ/K
m2 = -μ/K**2

# Formulate LQ problem 1 (retirement)
Q = 1
R = np.zeros((4, 4))
Rf = np.zeros((4, 4))
Rf[0, 0] = q
A = [[1 + r, s - c_bar, 0, 0],
     [0,             1, 0, 0],
     [0,             1, 1, 0],
     [0,             1, 2, 1]]
B = [[-1],
     [ 0],
     [ 0],
     [ 0]]
C = [[0],
     [0],
     [0],
     [0]]

# Initialize LQ instance for retired agent
lq_retired = LQ(Q, R, A, B, C, beta=β, T=T-K, Rf=Rf)
```

```
# Iterate back to start of retirement, record final value function
for i in range(T-K):
    lq_retired.update_values()
Rf2 = lq_retired.P

# Formulate LQ problem 2 (working life)
R = np.zeros((4, 4))
A = [[1 + r, -c_bar, m1, m2],
     [0,           1,  0,  0],
     [0,           1,  1,  0],
     [0,           1,  2,  1]]
B = [[-1],
     [ 0],
     [ 0],
     [ 0]]
C = [[σ],
     [0],
     [0],
     [0]]

# Set up working life LQ instance with terminal Rf from lq_retired
lq_working = LQ(Q, R, A, B, C, beta=β, T=K, Rf=Rf2)

# Simulate working state / control paths
x0 = (0, 1, 0, 0)
xp_w, up_w, wp_w = lq_working.compute_sequence(x0)
# Simulate retirement paths (note the initial condition)
xp_r, up_r, wp_r = lq_retired.compute_sequence(xp_w[:, K])

# Convert results back to assets, consumption and income
xp = np.column_stack((xp_w, xp_r[:, 1:]))
assets = xp[0, :]                    # Assets

up = np.column_stack((up_w, up_r))
c = up.flatten() + c_bar             # Consumption

time = np.arange(1, K+1)
income_w = σ * wp_w[0, 1:K+1] + m1 * time + m2 * time**2  # Income
income_r = np.full(T-K, s)
income = np.concatenate((income_w, income_r))

# Plot results
n_rows = 2
fig, axes = plt.subplots(n_rows, 1, figsize=(12, 10))

plt.subplots_adjust(hspace=0.5)

bbox = (0., 1.02, 1., .102)
legend_args = {'bbox_to_anchor': bbox, 'loc': 3, 'mode': 'expand'}
p_args = {'lw': 2, 'alpha': 0.7}

axes[0].plot(range(1, T+1), income, 'g-', label="non-financial income",
             **p_args)
axes[0].plot(range(T), c, 'k-', label="consumption", **p_args)

axes[1].plot(range(T+1), assets, 'b-', label="assets", **p_args)
axes[1].plot(range(T+1), np.zeros(T+1), 'k-')
```

```
for ax in axes:
    ax.grid()
    ax.set_xlabel('Time')
    ax.legend(ncol=2, **legend_args)

plt.show()
```



Exercise 72.7.3

Reproduce the figures from the monopolist application *given above*.

For parameters, use $a_0 = 5, a_1 = 0.5, \sigma = 0.15, \rho = 0.9, \beta = 0.95$ and $c = 2$, while γ varies between 1 and 50 (see figures).

Solution

The first task is to find the matrices A, B, C, Q, R that define the LQ problem.

Recall that $x_t = (\bar{q}_t \;\; q_t \;\; 1)^\top$, while $u_t = q_{t+1} - q_t$.

Letting $m_0 := (a_0 - c)/2a_1$ and $m_1 := 1/2a_1$, we can write $\bar{q}_t = m_0 + m_1 d_t$, and then, with some manipulation

$$\bar{q}_{t+1} = m_0(1-\rho) + \rho\bar{q}_t + m_1\sigma w_{t+1}$$

By our definition of u_t, the dynamics of q_t are $q_{t+1} = q_t + u_t$.

Using these facts you should be able to build the correct A, B, C matrices (and then check them against those found in the solution code below).

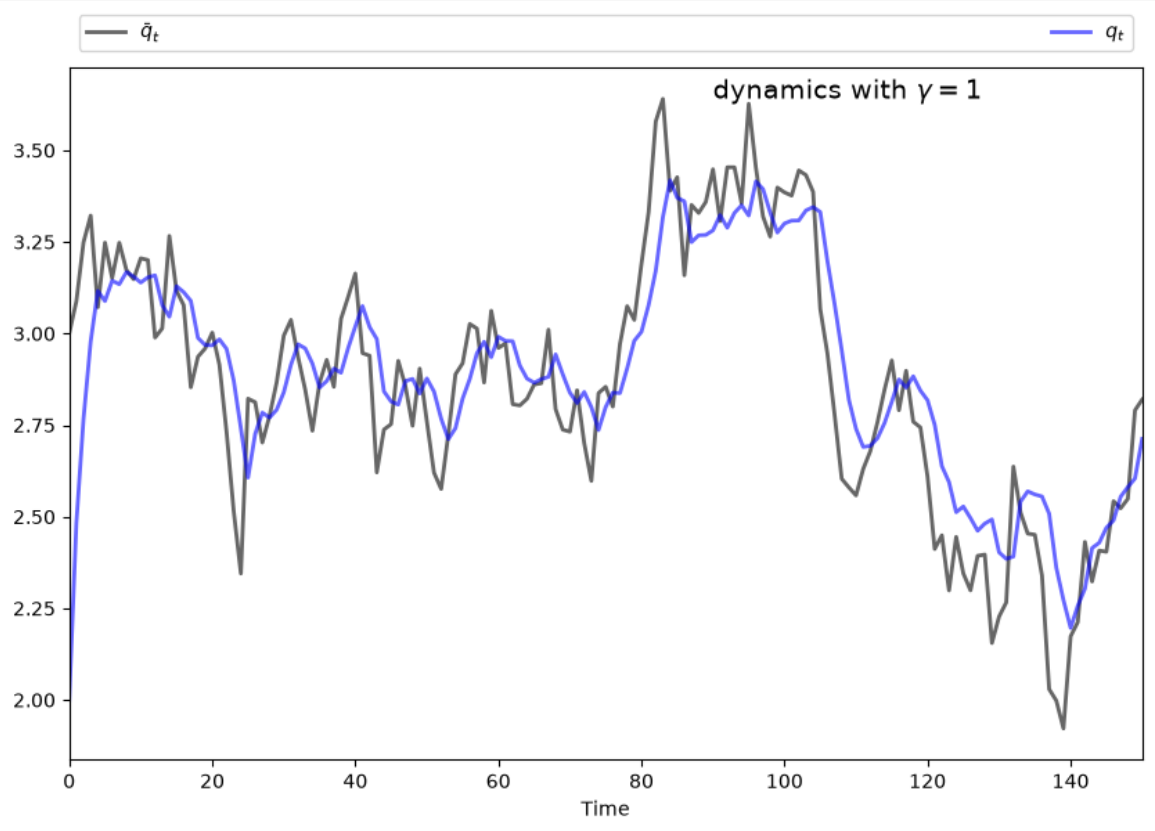
Suitable R, Q matrices can be found by inspecting the objective function, which we repeat here for convenience:

$$\min \mathbb{E}\left\{\sum_{t=0}^{\infty} \beta^t a_1(q_t - \bar{q}_t)^2 + \gamma u_t^2\right\}$$

Our solution code is

```
# Model parameters
a0 = 5
a1 = 0.5
σ = 0.15
ρ = 0.9
γ = 1
β = 0.95
c = 2
T = 120

# Useful constants
m0 = (a0-c)/(2 * a1)
m1 = 1/(2 * a1)

# Formulate LQ problem
Q = γ
R = [[ a1, -a1,  0],
     [-a1,  a1,  0],
     [  0,   0,  0]]
A = [[ρ, 0, m0 * (1 - ρ)],
     [0, 1,           0],
     [0, 0,           1]]

B = [[0],
     [1],
     [0]]
C = [[m1 * σ],
     [     0],
     [     0]]

lq = LQ(Q, R, A, B, C=C, beta=β)

# Simulate state / control paths
x0 = (m0, 2, 1)
xp, up, wp = lq.compute_sequence(x0, ts_length=150)
q_bar = xp[0, :]
q = xp[1, :]

# Plot simulation results
fig, ax = plt.subplots(figsize=(10, 6.5))
```

```
# Some fancy plotting stuff -- simplify if you prefer
bbox = (0., 1.01, 1., .101)
legend_args = {'bbox_to_anchor': bbox, 'loc': 3, 'mode': 'expand'}
p_args = {'lw': 2, 'alpha': 0.6}

time = range(len(q))
ax.set(xlabel='Time', xlim=(0, max(time)))
ax.plot(time, q_bar, 'k-', lw=2, alpha=0.6, label=r'$\bar q_t$')
ax.plot(time, q, 'b-', lw=2, alpha=0.6, label='$q_t$')
ax.legend(ncol=2, **legend_args)
s = fr'dynamics with $\gamma = {γ}$'
ax.text(max(time) * 0.6, 1 * q_bar.max(), s, fontsize=14)
plt.show()
```



CHAPTER

SEVENTYTHREE

LAGRANGIAN FOR LQ CONTROL

```
!pip install quantecon
```

```
import numpy as np
from quantecon import LQ
from scipy.linalg import schur
```

73.1 Overview

This is a sequel to this lecture *linear quadratic dynamic programming*

It can also be regarded as presenting **invariant subspace** techniques that extend ones that we encountered earlier in this lecture *stability in linear rational expectations models*

We present a Lagrangian formulation of an infinite horizon linear quadratic undiscounted dynamic programming problem.

Such a problem is also sometimes called an optimal linear regulator problem.

A Lagrangian formulation

- carries insights about connections between stability and optimality
- is the basis for fast algorithms for solving Riccati equations
- opens the way to constructing solutions of dynamic systems that don't come directly from an intertemporal optimization problem

A key tool in this lecture is the concept of an $n \times n$ **symplectic** matrix.

A symplectic matrix has eigenvalues that occur in **reciprocal pairs**, meaning that if $\lambda_i \in (-1, 1)$ is an eigenvalue, then so is λ_i^{-1}.

This reciprocal pairs property of the eigenvalues of a matrix is a tell-tale sign that the matrix describes the joint dynamics of a system of equations describing the **states** and **costates** that constitute first-order necessary conditions for solving an undiscounted linear-quadratic infinite-horizon optimization problem.

The symplectic matrix that will interest us describes the first-order dynamics of **state** and **co-state** vectors of an optimally controlled system.

In focusing on eigenvalues and eigenvectors of this matrix, we capitalize on an analysis of **invariant subspaces.**

These invariant subspace formulations of LQ dynamic programming problems provide a bridge between recursive (i.e., dynamic programming) formulations and classical formulations of linear control and linear filtering problems that make use of related matrix decompositions (see for example this lecture and this lecture).

While most of this lecture focuses on undiscounted problems, later sections describe handy ways of transforming discounted problems to undiscounted ones.

The techniques in this lecture will prove useful when we study Stackelberg and Ramsey problem in this lecture.

73.2 Undiscounted LQ DP Problem

The problem is to choose a sequence of controls $\{u_t\}_{t=0}^{\infty}$ to maximize the criterion

$$-\sum_{t=0}^{\infty}\{x_t' R x_t + u_t' Q u_t\}$$

subject to $x_{t+1} = Ax_t + Bu_t$, where x_0 is a given initial state vector.

Here x_t is an $(n \times 1)$ vector of state variables, u_t is a $(k \times 1)$ vector of controls, R is a positive semidefinite symmetric matrix, Q is a positive definite symmetric matrix, A is an $(n \times n)$ matrix, and B is an $(n \times k)$ matrix.

The optimal value function turns out to be quadratic, $V(x) = -x'Px$, where P is a positive semidefinite symmetric matrix.

Using the transition law to eliminate next period's state, the Bellman equation becomes

$$-x'Px = \max_u\{-x'Rx - u'Qu - (Ax + Bu)'P(Ax + Bu)\} \tag{73.1}$$

The first-order necessary conditions for the maximum problem on the right side of equation (73.1) are

Note

We use the following rules for differentiating quadratic and bilinear matrix forms: $\frac{\partial x'Ax}{\partial x} = (A + A')x$; $\frac{\partial y'Bz}{\partial y} = Bz$, $\frac{\partial y'Bz}{\partial z} = B'y$.

$$(Q + B'PB)u = -B'PAx,$$

which implies that an optimal decision rule for u is

$$u = -(Q + B'PB)^{-1}B'PAx$$

or

$$u = -Fx,$$

where

$$F = (Q + B'PB)^{-1}B'PA.$$

Substituting $u = -(Q + B'PB)^{-1}B'PAx$ into the right side of equation (73.1) and rearranging gives

$$P = R + A'PA - A'PB(Q + B'PB)^{-1}B'PA. \tag{73.2}$$

Equation (73.2) is called an **algebraic matrix Riccati** equation.

There are multiple solutions of equation (73.2).

But only one of them is positive definite.

The positive define solution is associated with the maximum of our problem.

It expresses the matrix P as an implicit function of the matrices R, Q, A, B.

Notice that the **gradient of the value function** is

$$\frac{\partial V(x)}{\partial x} = -2Px \tag{73.3}$$

We shall use fact (73.3) later.

73.3 Lagrangian

For the undiscounted optimal linear regulator problem, form the Lagrangian

$$L = -\sum_{t=0}^{\infty}\left\{x_t' R x_t + u_t' Q u_t + 2\mu_{t+1}'[Ax_t + Bu_t - x_{t+1}]\right\} \tag{73.4}$$

where $2\mu_{t+1}$ is a vector of Lagrange multipliers on the time t transition law $x_{t+1} = Ax_t + Bu_t$.

(We put the 2 in front of μ_{t+1} to make things match up nicely with equation (73.3).)

First-order conditions for maximization with respect to $\{u_t, x_{t+1}\}_{t=0}^{\infty}$ are

$$\begin{aligned} 2Qu_t + 2B'\mu_{t+1} &= 0 \ , \ t \geq 0 \\ \mu_t &= Rx_t + A'\mu_{t+1} \ , \ t \geq 1. \end{aligned} \tag{73.5}$$

Define μ_0 to be a vector of shadow prices of x_0 and apply an envelope condition to (73.4) to deduce that

$$\mu_0 = Rx_0 + A'\mu_1,$$

which is a time $t = 0$ counterpart to the second equation of system (73.5).

An important fact is that

$$\mu_{t+1} = Px_{t+1} \tag{73.6}$$

where P is a positive define matrix that solves the algebraic Riccati equation (73.2).

Thus, from equations (73.3) and (73.6), $-2\mu_t$ is the gradient of the value function with respect to x_t.

The Lagrange multiplier vector μ_t is often called the **costate** vector that corresponds to the **state** vector x_t.

It is useful to proceed with the following steps:

- solve the first equation of (73.5) for u_t in terms of μ_{t+1}.
- substitute the result into the law of motion $x_{t+1} = Ax_t + Bu_t$.
- arrange the resulting equation and the second equation of (73.5) into the form

$$L \begin{bmatrix} x_{t+1} \\ \mu_{t+1} \end{bmatrix} = N \begin{bmatrix} x_t \\ \mu_t \end{bmatrix} \ , \ t \geq 0, \tag{73.7}$$

where

$$L = \begin{bmatrix} I & BQ^{-1}B' \\ 0 & A' \end{bmatrix}, \quad N = \begin{bmatrix} A & 0 \\ -R & I \end{bmatrix}.$$

When L is of full rank (i.e., when A is of full rank), we can write system (73.7) as

$$\begin{bmatrix} x_{t+1} \\ \mu_{t+1} \end{bmatrix} = M \begin{bmatrix} x_t \\ \mu_t \end{bmatrix} \tag{73.8}$$

where

$$M \equiv L^{-1}N = \begin{bmatrix} A + BQ^{-1}B'A'^{-1}R & -BQ^{-1}B'A'^{-1} \\ -A'^{-1}R & A'^{-1} \end{bmatrix}. \tag{73.9}$$

73.4 State-Costate Dynamics

We seek to solve the difference equation system (73.8) for a sequence $\{x_t\}_{t=0}^{\infty}$ that satisfies

- an initial condition for x_0
- a terminal condition $\lim_{t\to+\infty} x_t = 0$

This terminal condition reflects our desire for a **stable** solution, one that does not diverge as $t \to \infty$.

We inherit our wish for stability of the $\{x_t\}$ sequence from a desire to maximize

$$-\sum_{t=0}^{\infty}[x_t'Rx_t + u_t'Qu_t],$$

which requires that $x_t'Rx_t$ converge to zero as $t \to +\infty$.

73.5 Reciprocal Pairs Property

To proceed, we study properties of the $(2n \times 2n)$ matrix M defined in (73.9).

It helps to introduce a $(2n \times 2n)$ matrix

$$J = \begin{bmatrix} 0 & -I_n \\ I_n & 0 \end{bmatrix}.$$

The rank of J is $2n$.

Definition: A matrix M is called **symplectic** if

$$MJM' = J. \tag{73.10}$$

Salient properties of symplectic matrices that are readily verified include:

- If M is symplectic, then M^2 is symplectic
- The determinant of a symplectic, then $\det(M) = 1$

It can be verified directly that M in equation (73.9) is symplectic.

It follows from equation (73.10) and from the fact $J^{-1} = J' = -J$ that for any symplectic matrix M,

$$M' = J^{-1}M^{-1}J. \tag{73.11}$$

Equation (73.11) states that M' is related to the inverse of M by a **similarity transformation**.

For square matrices, recall that

- similar matrices share eigenvalues
- eigenvalues of the inverse of a matrix are inverses of eigenvalues of the matrix
- a matrix and its transpose share eigenvalues

It then follows from equation (73.11) that the eigenvalues of M occur in reciprocal pairs: if λ is an eigenvalue of M, so is λ^{-1}.

Write equation (73.8) as

$$y_{t+1} = My_t \tag{73.12}$$

where $y_t = \begin{bmatrix} x_t \\ \mu_t \end{bmatrix}$.

Consider a **triangularization** of M

$$V^{-1}MV = \begin{bmatrix} W_{11} & W_{12} \\ 0 & W_{22} \end{bmatrix} \tag{73.13}$$

where

- each block on the right side is $(n \times n)$
- V is nonsingular
- all eigenvalues of W_{22} exceed 1 in modulus
- all eigenvalues of W_{11} are less than 1 in modulus

73.6 Schur decomposition

The **Schur decomposition** and the **eigenvalue decomposition** are two decompositions of the form (73.13).

Write equation (73.12) as

$$y_{t+1} = VWV^{-1}y_t. \tag{73.14}$$

A solution of equation (73.14) for arbitrary initial condition y_0 is evidently

$$y_t = V \begin{bmatrix} W_{11}^t & W_{12,t} \\ 0 & W_{22}^t \end{bmatrix} V^{-1}y_0 \tag{73.15}$$

where $W_{12,t} = W_{12}$ for $t = 1$ and for $t \geq 2$ obeys the recursion

$$W_{12,t} = W_{11}^{t-1}W_{12,t-1} + W_{12,t-1}W_{22}^{t-1}$$

and where W_{ii}^t is W_{ii} raised to the tth power.

Write equation (73.15) as

$$\begin{bmatrix} y_{1t}^* \\ y_{2t}^* \end{bmatrix} = \begin{bmatrix} W_{11}^t & W_{12,t} \\ 0 & W_{22}^t \end{bmatrix} \begin{bmatrix} y_{10}^* \\ y_{20}^* \end{bmatrix}$$

where $y_t^* = V^{-1}y_t$, and in particular where

$$y_{2t}^* = V^{21}x_t + V^{22}\mu_t, \tag{73.16}$$

and where V^{ij} denotes the (i, j) piece of the partitioned V^{-1} matrix.

Because W_{22} is an unstable matrix, y_t^* will diverge unless $y_{20}^* = 0$.

Let V^{ij} denote the (i, j) piece of the partitioned V^{-1} matrix.

To attain stability, we must impose $y_{20}^* = 0$, which from equation (73.16) implies

$$V^{21}x_0 + V^{22}\mu_0 = 0$$

or

$$\mu_0 = -(V^{22})^{-1}V^{21}x_0.$$

This equation replicates itself over time in the sense that it implies

$$\mu_t = -(V^{22})^{-1} V^{21} x_t.$$

But notice that because $(V^{21}\ V^{22})$ is the second row block of the inverse of V, it follows that

$$(V^{21}\ V^{22}) \quad \begin{bmatrix} V_{11} \\ V_{21} \end{bmatrix} = 0$$

which implies

$$V^{21} V_{11} + V^{22} V_{21} = 0.$$

Therefore,

$$-(V^{22})^{-1} V^{21} = V_{21} V_{11}^{-1}.$$

So we can write

$$\mu_0 = V_{21} V_{11}^{-1} x_0$$

and

$$\mu_t = V_{21} V_{11}^{-1} x_t.$$

However, we know that $\mu_t = P x_t$, where P occurs in the matrix that solves the Riccati equation.

Thus, the preceding argument establishes that

$$P = V_{21} V_{11}^{-1}. \tag{73.17}$$

Remarkably, formula (73.17) provides us with a computationally efficient way of computing the positive definite matrix P that solves the algebraic Riccati equation (73.2) that emerges from dynamic programming.

This same method can be applied to compute the solution of any system of the form (73.8) if a solution exists, even if eigenvalues of M fail to occur in reciprocal pairs.

The method will typically work so long as the eigenvalues of M split half inside and half outside the unit circle.

Systems in which eigenvalues (properly adjusted for discounting) fail to occur in reciprocal pairs arise when the system being solved is an equilibrium of a model in which there are distortions that prevent there being any optimum problem that the equilibrium solves. See [Ljungqvist and Sargent, 2018], ch 12.

73.7 Application

Here we demonstrate the computation with the deterministic permanent-income example from this *LQ Control: Foundations*.

Because that model is discounted, we apply the invariant-subspace method to the equivalent *undiscounted* system obtained from the transformed matrices $\hat{A} = \beta^{1/2} A$ and $\hat{B} = \beta^{1/2} B$.

```
# Model parameters
r = 0.05
β = 1 / (1 + r)
c_bar = 2
μ = 1
```

(continues on next page)

(continued from previous page)

```
# Formulate as an LQ problem
Q = np.array([[1]])
R = np.zeros((2, 2))
A = [[1 + r, -c_bar + μ],
     [0,              1]]
B = [[-1],
     [0]]

# Construct an LQ instance
lq = LQ(Q, R, A, B, beta=β)
```

Given matrices A, B, Q, R, we can then compute L, N, and $M = L^{-1}N$.

```
def construct_LNM(A, B, Q, R):

    n = A.shape[0]

    # construct L and N
    L = np.zeros((2*n, 2*n))
    L[:n, :n] = np.eye(n)
    L[:n, n:] = B @ np.linalg.inv(Q) @ B.T
    L[n:, n:] = A.T

    N = np.zeros((2*n, 2*n))
    N[:n, :n] = A
    N[n:, :n] = -R
    N[n:, n:] = np.eye(n)

    # compute M
    M = np.linalg.inv(L) @ N

    return L, N, M
```

```
A_bar = lq.A * lq.beta ** (1/2)
B_bar = lq.B * lq.beta ** (1/2)

L, N, M = construct_LNM(A_bar, B_bar, lq.Q, lq.R)
```

```
M
```

```
array([[ 1.02469508, -0.97590007, -0.92942864,  0.        ],
       [ 0.        ,  0.97590007,  0.        ,  0.        ],
       [ 0.        ,  0.        ,  0.97590007,  0.        ],
       [ 0.        ,  0.        ,  0.97590007,  1.02469508]])
```

Let's verify that M is symplectic.

```
n = lq.n
J = np.zeros((2*n, 2*n))
J[n:, :n] = np.eye(n)
J[:n, n:] = -np.eye(n)

M @ J @ M.T - J
```

```
array([[-5.15703240e-17,  0.00000000e+00,  0.00000000e+00,
         9.31818882e-18],
       [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         1.11022302e-16],
       [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00],
       [-9.31818882e-18, -1.11022302e-16,  0.00000000e+00,
         0.00000000e+00]])
```

We can compute the eigenvalues of M using `np.linalg.eigvals`, arranged in ascending order.

```
eigvals = sorted(np.linalg.eigvals(M), key=lambda z: (abs(z), z.real, z.imag))
eigvals
```

```
[np.float64(0.9759000729485331),
 np.float64(0.9759000729485332),
 np.float64(1.0246950765959597),
 np.float64(1.02469507659596)]
```

When we apply Schur decomposition such that $M = VWV^{-1}$, we want

- the upper left block of W, W_{11}, to have all of its eigenvalues less than 1 in modulus, and
- the lower right block W_{22} to have eigenvalues that exceed 1 in modulus.

To get what we want, let's define a sorting function that tells `scipy.schur` to sort the corresponding eigenvalues with modulus smaller than 1 to the upper left.

```
tol = 1e-10

def sort_fun(x):
    "Sort the eigenvalues with modulus smaller than 1 to the top-left."
    return abs(x) < 1 - tol

W, V, stable_dim = schur(M, sort=sort_fun)
stable_dim
```

```
2
```

```
W
```

```
array([[ 9.75900073e-01,  1.06779938e-16,  9.77000404e-01,
        -9.27110864e-01],
       [ 0.00000000e+00,  9.75900073e-01, -2.31777716e-03,
         9.77000404e-01],
       [ 0.00000000e+00,  0.00000000e+00,  1.02469508e+00,
        -2.12887870e-16],
       [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         1.02469508e+00]])
```

```
V
```

```
array([[ 0.99875234, -0.00236938,  0.04988138,  0.        ],
       [ 0.04993762,  0.0473876 , -0.99762751,  0.        ],
       [ 0.        , -0.04988138, -0.00236938,  0.99875234],
       [ 0.        ,  0.99762751,  0.0473876 ,  0.04993762]])
```

We can check the modulus of eigenvalues of W_{11} and W_{22}.

Since they are both triangular matrices, eigenvalues are the diagonal elements.

```
# W11
np.diag(W[:n, :n])
```

```
array([0.97590007, 0.97590007])
```

```
# W22
np.diag(W[n:, n:])
```

```
array([1.02469508, 1.02469508])
```

The following functions wrap M matrix construction, Schur decomposition, and stability-imposing computation of P.

```
def stable_solution(M, verbose=True):
    """
    Given a system of linear difference equations

        y' = |a b| y
        x' = |c d| x

    which is potentially unstable, find the solution
    by imposing stability.

    Parameter
    ---------
    M : np.ndarray(float)
        The matrix represents the linear difference equations system.
    """
    n = M.shape[0] // 2
    tol = 1e-10

    def sort_fun(x):
        "Sort the eigenvalues with modulus smaller than 1 to the top-left."
        return abs(x) < 1 - tol

    W, V, stable_dim = schur(M, sort=sort_fun)
    if stable_dim != n:
        raise ValueError(
    f"Expected {n} stable eigenvalues inside the unit circle, found {stable_dim}."
    )
    if verbose:
        print('eigenvalues:\n')
        print('    W11: {}'.format(np.diag(W[:n, :n])))
        print('    W22: {}'.format(np.diag(W[n:, n:])))

    # compute V21 V11^{-1} without forming the inverse explicitly
    P = np.linalg.solve(V[:n, :n].T, V[n:, :n].T).T

    return W, V, P

def stationary_P(lq, verbose=True):
    """
    Computes the matrix :math:`P` that represent the value function
```

(continues on next page)

(continued from previous page)

```
        V(x) = x' P x

    in the infinite horizon case. Computation is via imposing stability
    on the solution path and using Schur decomposition.

    Parameters
    ----------
    lq : qe.LQ
        QuantEcon class for analyzing linear quadratic optimal control
        problems of infinite horizon form.

    Returns
    -------
    P : array_like(float)
        P matrix in the value function representation.
    """

    Q = lq.Q
    R = lq.R
    A = lq.A * lq.beta ** (1/2)
    B = lq.B * lq.beta ** (1/2)

    n, k = lq.n, lq.k

    L, N, M = construct_LNM(A, B, Q, R)
    W, V, P = stable_solution(M, verbose=verbose)

    return P
```

```
# compute P
stationary_P(lq)
```

```
eigenvalues:

    W11: [0.97590007 0.97590007]
    W22: [1.02469508 1.02469508]
```

```
array([[ 0.0525, -1.05  ],
       [-1.05  , 21.    ]])
```

Note that the matrix P computed in this way is close to what we get from the routine in quantecon that solves an algebraic Riccati equation by iterating to convergence on a Riccati difference equation.

The small difference comes from computational errors and will decrease as we increase the maximum number of iterations or decrease the tolerance for convergence.

```
lq.stationary_values()
```

```
(array([[ 0.0525, -1.05  ],
        [-1.05  , 21.    ]]),
 array([[-0.05,  1.  ]]),
 np.float64(0.0))
```

Using a Schur decomposition is much more efficient.

```
%%timeit
stationary_P(lq, verbose=False)
```

```
97.2 µs ± 413 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

```
%%timeit
lq.stationary_values()
```

```
1.81 ms ± 1.46 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

73.8 Other Applications

The preceding approach to imposing stability on a system of potentially unstable linear difference equations is not limited to linear quadratic dynamic optimization problems.

For example, the same method is used in our Stability in Linear Rational Expectations Models lecture.

Let's try to solve the model described in that lecture by applying the `stable_solution` function defined in this lecture above.

```
def construct_H(ρ, λ, δ):
    "contruct matrix H given parameters."

    H = np.empty((2, 2))
    H[0, :] = ρ,δ
    H[1, :] = - (1 - λ) / λ, 1 / λ

    return H

H = construct_H(ρ=.9, λ=.5, δ=0)
```

```
W, V, P = stable_solution(H)
P
```

```
eigenvalues:

    W11: [0.9]
    W22: [2.]
```

```
array([[0.90909091]])
```

73.9 Discounted Problems

73.9.1 Transforming States and Controls to Eliminate Discounting

A pair of useful transformations allows us to convert a discounted problem into an undiscounted one.

Thus, suppose that we have a discounted problem with objective

$$-\sum_{t=0}^{\infty} \beta^t \left\{ x_t' R x_t + u_t' Q u_t \right\}$$

and that the state transition equation is again $x_{t+1} = Ax_t + Bu_t$.

Define the transformed state and control variables

- $\hat{x}_t = \beta^{\frac{t}{2}} x_t$
- $\hat{u}_t = \beta^{\frac{t}{2}} u_t$

and the transformed transition equation matrices

- $\hat{A} = \beta^{\frac{1}{2}} A$
- $\hat{B} = \beta^{\frac{1}{2}} B$

so that the adjusted state and control variables obey the transition law

$$\hat{x}_{t+1} = \hat{A}\hat{x}_t + \hat{B}\hat{u}_t.$$

Then a discounted optimal control problem defined by A, B, R, Q, β having optimal policy characterized by P, F is associated with an equivalent undiscounted problem defined by $\hat{A}, \hat{B}, Q, R$ having optimal policy characterized by $\hat{F}, \hat{P}$ that satisfy the following equations:

$$\hat{F} = (Q + B'\hat{P}B)^{-1}\hat{B}'P\hat{A}$$

and

$$\hat{P} = R + \hat{A}'P\hat{A} - \hat{A}'P\hat{B}(Q + B'\hat{P}\hat{B})^{-1}\hat{B}'P\hat{A}$$

It follows immediately from the definitions of $\hat{A}, \hat{B}$ that $\hat{F} = F$ and $\hat{P} = P$.

By exploiting these transformations, we can solve a discounted problem by solving an associated undiscounted problem.

In particular, we can first transform a discounted LQ problem to an undiscounted one and then solve that discounted optimal regulator problem using the Lagrangian and invariant subspace methods described above.

For example, when $\beta = \frac{1}{1+r}$, we can solve for P with $\hat{A} = \beta^{1/2}A$ and $\hat{B} = \beta^{1/2}B$.

These settings are adopted by default in the function `stationary_P` defined above.

```
stationary_P(lq)
```

```
eigenvalues:

    W11: [0.97590007 0.97590007]
    W22: [1.02469508 1.02469508]
```

```
array([[ 0.0525, -1.05  ],
       [-1.05  , 21.    ]])
```

We can verify that the solution agrees with one that comes from applying the routine `LQ.stationary_values` in the quantecon package.

```
lq.stationary_values()
```

```
(array([[ 0.0525, -1.05  ],
        [-1.05  , 21.    ]]),
 array([[-0.05,  1.  ]]),
 np.float64(0.0))
```

73.9.2 Lagrangian for Discounted Problem

For several purposes, it is useful explicitly briefly to describe a Lagrangian for a discounted problem.

Thus, for the discounted optimal linear regulator problem, form the Lagrangian

$$L = -\sum_{t=0}^{\infty} \beta^t \left\{ x_t' R x_t + u_t' Q u_t + 2\beta \mu_{t+1}' [A x_t + B u_t - x_{t+1}] \right\} \tag{73.18}$$

where $2\mu_{t+1}$ is a vector of Lagrange multipliers on the state vector x_{t+1}.

First-order conditions for maximization with respect to $\{u_t, x_{t+1}\}_{t=0}^{\infty}$ are

$$\begin{aligned} 2Qu_t + 2\beta B'\mu_{t+1} = 0 \ , \ t \geq 0 \\ \mu_t = Rx_t + \beta A' \mu_{t+1} \ , \ t \geq 1. \end{aligned} \tag{73.19}$$

Define $2\mu_0$ to be the vector of shadow prices of x_0 and apply an envelope condition to (73.18) to deduce that

$$\mu_0 = Rx_0 + \beta A' \mu_1,$$

which is a time $t = 0$ counterpart to the second equation of system (73.19).

Proceeding as we did above with the undiscounted system (73.5), we can rearrange the first-order conditions into the system

$$\begin{bmatrix} I & \beta B Q^{-1} B' \\ 0 & \beta A' \end{bmatrix} \begin{bmatrix} x_{t+1} \\ \mu_{t+1} \end{bmatrix} = \begin{bmatrix} A & 0 \\ -R & I \end{bmatrix} \begin{bmatrix} x_t \\ \mu_t \end{bmatrix} \tag{73.20}$$

which in the special case that $\beta = 1$ agrees with equation (73.5), as expected.

By staring at system (73.20), we can infer identities that shed light on the structure of optimal linear regulator problems, some of which will be useful in this lecture when we apply and extend the methods of this lecture to study Stackelberg and Ramsey problems.

First, note that the first block of equation system (73.20) asserts that when $\mu_{t+1} = P x_{t+1}$, then

$$(I + \beta Q^{-1} B' P B P) x_{t+1} = A x_t,$$

which can be rearranged to sbe

$$x_{t+1} = (I + \beta B Q^{-1} B' P)^{-1} A x_t.$$

This expression for the optimal closed loop dynamics of the state must agree with an alternative expression that we had derived with dynamic programming, namely,

$$x_{t+1} = (A - BF) x_t.$$

But using

$$F = \beta (Q + \beta B' P B)^{-1} B' P A \tag{73.21}$$

it follows that

$$A - BF = (I - \beta B (Q + \beta B' P B)^{-1} B' P) A.$$

Thus, our two expressions for the closed loop dynamics agree if and only if

$$(I + \beta B Q^{-1} B' P)^{-1} = (I - \beta B (Q + \beta B' P B)^{-1} B' P). \tag{73.22}$$

Matrix equation (73.22) can be verified by applying a partitioned inverse formula.

Note

Just use the formula $(a-bd^{-1}c)^{-1} = a^{-1}+a^{-1}b(d-ca^{-1}b)^{-1}ca^{-1}$ for appropriate choices of the matrices a, b, c, d.

Next, note that for *any* fixed F for which eigenvalues of $A-BF$ are less than $\frac{1}{\beta}$ in modulus, the value function associated with using this rule forever is $-x_0\tilde{P}x_0$ where $\tilde{P}$ obeys the following matrix equation:

$$\tilde{P} = (R + F'QF) + \beta(A - BF)'P(A - BF). \tag{73.23}$$

Evidently, $\tilde{P} = P$ only when F obeys formula (73.21).

Next, note that the second equation of system (73.20) implies the "forward looking" equation for the Lagrange multiplier

$$\mu_t = Rx_t + \beta A'\mu_{t+1}$$

whose solution is

$$\mu_t = Px_t,$$

where

$$P = R + \beta A'P(A - BF) \tag{73.24}$$

where we must require that F obeys equation (73.21).

Equations (73.23) and (73.24) provide different perspectives on the optimal value function.

CHAPTER

SEVENTYFOUR

ELIMINATING CROSS PRODUCTS

74.1 Overview

This lecture describes formulas for eliminating

- cross products between states and control in linear-quadratic dynamic programming problems
- covariances between state and measurement noises in Kalman filtering problems

For a linear-quadratic dynamic programming problem, the idea involves these steps

- transform states and controls in a way that leads to an equivalent problem with no cross-products between transformed states and controls
- solve the transformed problem using standard formulas for problems with no cross-products between states and controls presented in this lecture *Linear Control: Foundations*
- transform the optimal decision rule for the altered problem into the optimal decision rule for the original problem with cross-products between states and controls

74.2 Undiscounted Dynamic Programming Problem

Here is a nonstochastic undiscounted LQ dynamic programming with cross products between states and controls in the objective function.

The problem is defined by the 5-tuple of matrices (A, B, R, Q, H) where R and Q are positive definite symmetric matrices and $A \sim m \times m, B \sim m \times k, Q \sim k \times k, R \sim m \times m$ and $H \sim k \times m$.

The problem is to choose $\{x_{t+1}, u_t\}_{t=0}^{\infty}$ to maximize

$$-\sum_{t=0}^{\infty}(x_t' R x_t + u_t' Q u_t + 2 u_t H x_t)$$

subject to the linear constraints

$$x_{t+1} = A x_t + B u_t, \quad t \geq 0$$

where x_0 is a given initial condition.

The solution to this undiscounted infinite-horizon problem is a time-invariant feedback rule

$$u_t = -F x_t$$

where

$$F = -(Q + B'PB)^{-1}B'PA$$

and $P \sim m \times m$ is a positive definite solution of the algebraic matrix Riccati equation

$$P = R + A'PA - (A'PB + H')(Q + B'PB)^{-1}(B'PA + H).$$

It can be verified that an **equivalent** problem without cross-products between states and controls is defined by a 4-tuple of matrices : (A^*, B, R^*, Q).

That the omitted matrix $H = 0$ indicates that there are no cross products between states and controls in the equivalent problem.

The matrices (A^*, B, R^*, Q) defining the equivalent problem and the value function, policy function matrices P, F^* that solve it are related to the matrices (A, B, R, Q, H) defining the original problem and the value function, policy function matrices P, F that solve the original problem by

$$\begin{aligned}
A^* &= A - BQ^{-1}H, \\
R^* &= R - H'Q^{-1}H, \\
P &= R^* + A^{*'}PA - (A^{*'}PB)(Q + B'PB)^{-1}B'PA^*, \\
F^* &= (Q + B'PB)^{-1}B'PA^*, \\
F &= F^* + Q^{-1}H.
\end{aligned}$$

74.3 Kalman Filter

The **duality** that prevails between a linear-quadratic optimal control and a Kalman filtering problem means that there is an analogous transformation that allows us to transform a Kalman filtering problem with non-zero covariance matrix between between shocks to states and shocks to measurements to an equivalent Kalman filtering problem with zero covariance between shocks to states and measurments.

Let's look at the appropriate transformations.

First, let's recall the Kalman filter with covariance between noises to states and measurements.

The hidden Markov model is

$$\begin{aligned}
x_{t+1} &= Ax_t + Bw_{t+1}, \\
z_{t+1} &= Dx_t + Fw_{t+1},
\end{aligned}$$

where $A \sim m \times m, B \sim m \times p$ and $D \sim k \times m, F \sim k \times p$, and w_{t+1} is the time $t+1$ component of a sequence of i.i.d. $p \times 1$ normally distibuted random vectors with mean vector zero and covariance matrix equal to a $p \times p$ identity matrix.

Thus, x_t is $m \times 1$ and z_t is $k \times 1$.

The Kalman filtering formulas are

$$\begin{aligned}
K(\Sigma_t) &= (A\Sigma_t D' + BF')(D\Sigma_t D' + FF')^{-1}, \\
\Sigma_{t+1} &= A\Sigma_t A' + BB' - (A\Sigma_t D' + BF')(D\Sigma_t D' + FF')^{-1}(D\Sigma_t A' + FB').
\end{aligned}$$

(eq:Kalman102)

Define tranformed matrices

$$\begin{aligned}
A^* &= A - BF'(FF')^{-1}D, \\
B^*B^{*'} &= BB' - BF'(FF')^{-1}FB'.
\end{aligned}$$

74.3.1 Algorithm

A consequence of formulas {eq}`eq:Kalman102} is that we can use the following algorithm to solve Kalman filtering problems that involve non zero covariances between state and signal noises.

First, compute Σ, K^* using the ordinary Kalman filtering formula with $BF' = 0$, i.e., with zero covariance matrix between random shocks to states and random shocks to measurements.

That is, compute K^* and Σ that satisfy

$$
\begin{aligned}
K^* &= (A^*\Sigma D')(D\Sigma D' + FF')^{-1} \\
\Sigma &= A^*\Sigma A^{*'} + B^*B^{*'} - (A^*\Sigma D')(D\Sigma D' + FF')^{-1}(D\Sigma A^{*'}).
\end{aligned}
$$

The Kalman gain for the original problem **with non-zero covariance** between shocks to states and measurements is then

$$
K = K^* + BF'(FF')^{-1},
$$

The state reconstruction covariance matrix Σ for the original problem equals the state reconstrution covariance matrix for the transformed problem.

74.4 Duality table

Here is a handy table to remember how the Kalman filter and dynamic program are related.

Dynamic Program	Kalman Filter
A	A'
B	D'
H	FB'
Q	FF'
R	BB'
F	K'
P	Σ

CHAPTER

SEVENTYFIVE

THE PERMANENT INCOME MODEL

Contents

- *The Permanent Income Model*
  - *Overview*
  - *The Savings Problem*
  - *Alternative Representations*
  - *Two Classic Examples*
  - *Further Reading*
  - *Appendix: The Euler Equation*

75.1 Overview

This lecture describes a rational expectations version of the famous permanent income model of Milton Friedman [Friedman, 1956].

Robert Hall cast Friedman's model within a linear-quadratic setting [Hall, 1978].

Like Hall, we formulate an infinite-horizon linear-quadratic savings problem.

We use the model as a vehicle for illustrating

- alternative formulations of the *state* of a dynamic system
- the idea of *cointegration*
- impulse response functions
- the idea that changes in consumption are useful as predictors of movements in income

Background readings on the linear-quadratic-Gaussian permanent income model are Hall's [Hall, 1978] and chapter 2 of [Ljungqvist and Sargent, 2018].

Let's start with some imports

```
import matplotlib.pyplot as plt
import numpy as np
import random
from numba import jit
```

75.2 The Savings Problem

In this section, we state and solve the savings and consumption problem faced by the consumer.

75.2.1 Preliminaries

We use a class of stochastic processes called martingales.

A discrete-time martingale is a stochastic process (i.e., a sequence of random variables) $\{X_t\}$ with finite mean at each t and satisfying

$$\mathbb{E}_t[X_{t+1}] = X_t, \qquad t = 0, 1, 2, ...$$

Here $\mathbb{E}_t := \mathbb{E}[\cdot \mid \mathcal{F}_t]$ is a conditional mathematical expectation conditional on the time t *information set* $\mathcal{F}_t$.

The latter is just a collection of random variables that the modeler declares to be visible at t.

- When not explicitly defined, it is usually understood that $\mathcal{F}_t = \{X_t, X_{t-1}, ..., X_0\}$.

Martingales have the feature that the history of past outcomes provides no predictive power for changes between current and future outcomes.

For example, the current wealth of a gambler engaged in a "fair game" has this property.

One common class of martingales is the family of *random walks*.

A **random walk** is a stochastic process $\{X_t\}$ that satisfies

$$X_{t+1} = X_t + w_{t+1}$$

for some IID zero mean *innovation* sequence $\{w_t\}$.

Evidently, X_t can also be expressed as

$$X_t = \sum_{j=1}^{t} w_j + X_0$$

Not every martingale arises as a random walk (see, for example, Wald's martingale).

75.2.2 The Decision Problem

A consumer has preferences over consumption streams that are ordered by the utility functional

$$\mathbb{E}_0 \left[\sum_{t=0}^{\infty} \beta^t u(c_t) \right] \tag{75.1}$$

where

- $\mathbb{E}_t$ is the mathematical expectation conditioned on the consumer's time t information
- c_t is time t consumption
- u is a strictly concave one-period utility function
- $\beta \in (0, 1)$ is a discount factor

The consumer maximizes (75.1) by choosing a consumption, borrowing plan $\{c_t, b_{t+1}\}_{t=0}^{\infty}$ subject to the sequence of budget constraints

$$c_t + b_t = \frac{1}{1+r} b_{t+1} + y_t \quad t \geq 0 \tag{75.2}$$

Here

- y_t is an exogenous endowment process.
- $r > 0$ is a time-invariant risk-free net interest rate.
- b_t is one-period risk-free debt maturing at t.

The consumer also faces initial conditions b_0 and y_0, which can be fixed or random.

75.2.3 Assumptions

For the remainder of this lecture, we follow Friedman and Hall in assuming that $(1+r)^{-1} = \beta$.

Regarding the endowment process, we assume it has the *state-space representation*

$$\begin{aligned} z_{t+1} &= A z_t + C w_{t+1} \\ y_t &= U z_t \end{aligned} \tag{75.3}$$

where

- $\{w_t\}$ is an IID vector process with $\mathbb{E} w_t = 0$ and $\mathbb{E} w_t w_t' = I$.
- The *spectral radius* of A satisfies $\rho(A) < \sqrt{1/\beta}$.
- U is a selection vector that pins down y_t as a particular linear combination of components of z_t.

The restriction on $\rho(A)$ prevents income from growing so fast that discounted geometric sums of some quadratic forms to be described below become infinite.

Regarding preferences, we assume the quadratic utility function

$$u(c_t) = -(c_t - \gamma)^2$$

where γ is a bliss level of consumption.

Note

Along with this quadratic utility specification, we allow consumption to be negative. However, by choosing parameters appropriately, we can make the probability that the model generates negative consumption paths over finite time horizons as low as desired.

Finally, we impose the *no Ponzi scheme* condition

$$\mathbb{E}_0 \left[\sum_{t=0}^{\infty} \beta^t b_t^2 \right] < \infty \tag{75.4}$$

This condition rules out an always-borrow scheme that would allow the consumer to enjoy bliss consumption forever.

75.2.4 First-Order Conditions

First-order conditions for maximizing (75.1) subject to (75.2) are

$$\mathbb{E}_t[u'(c_{t+1})] = u'(c_t), \qquad t = 0, 1, \ldots \tag{75.5}$$

These optimality conditions are also known as *Euler equations*.

If you're not sure where they come from, you can find a proof sketch in the *appendix*.

With our quadratic preference specification, (75.5) has the striking implication that consumption follows a martingale:

$$\mathbb{E}_t[c_{t+1}] = c_t \tag{75.6}$$

(In fact, quadratic preferences are *necessary* for this conclusion[1].)

One way to interpret (75.6) is that consumption will change only when "new information" about permanent income is revealed.

These ideas will be clarified below.

75.2.5 The Optimal Decision Rule

Now let's deduce the optimal decision rule[2].

Note

One way to solve the consumer's problem is to apply *dynamic programming* as in *this lecture*. We do this later. But first we use an alternative approach that is revealing and shows the work that dynamic programming does for us behind the scenes.

In doing so, we need to combine

1. the optimality condition (75.6)
2. the period-by-period budget constraint (75.2), and
3. the boundary condition (75.4)

To accomplish this, observe first that (75.4) implies $\lim_{t\to\infty} \beta^{\frac{t}{2}} b_{t+1} = 0$.

Using this restriction on the debt path and solving (75.2) forward yields

$$b_t = \sum_{j=0}^{\infty} \beta^j (y_{t+j} - c_{t+j}) \tag{75.7}$$

Take conditional expectations on both sides of (75.7) and use the martingale property of consumption and the *law of iterated expectations* to deduce

$$b_t = \sum_{j=0}^{\infty} \beta^j \mathbb{E}_t[y_{t+j}] - \frac{c_t}{1-\beta} \tag{75.8}$$

[1] A linear marginal utility is essential for deriving (75.6) from (75.5). Suppose instead that we had imposed the following more standard assumptions on the utility function: $u'(c) > 0, u''(c) < 0, u'''(c) > 0$ and required that $c \geq 0$. The Euler equation remains (75.5). But the fact that $u''' < 0$ implies via Jensen's inequality that $\mathbb{E}_t[u'(c_{t+1})] > u'(\mathbb{E}_t[c_{t+1}])$. This inequality together with (75.5) implies that $\mathbb{E}_t[c_{t+1}] > c_t$ (consumption is said to be a 'submartingale'), so that consumption stochastically diverges to $+\infty$. The consumer's savings also diverge to $+\infty$.

[2] An optimal decision rule is a map from the current state into current actions—in this case, consumption.

Expressed in terms of c_t we get

$$c_t = (1-\beta)\left[\sum_{j=0}^{\infty}\beta^j \mathbb{E}_t[y_{t+j}] - b_t\right] = \frac{r}{1+r}\left[\sum_{j=0}^{\infty}\beta^j \mathbb{E}_t[y_{t+j}] - b_t\right] \tag{75.9}$$

where the last equality uses $(1+r)\beta = 1$.

These last two equations assert that consumption equals *economic income*

- **financial wealth** equals $-b_t$
- **non-financial wealth** equals $\sum_{j=0}^{\infty}\beta^j \mathbb{E}_t[y_{t+j}]$
- **total wealth** equals the sum of financial and non-financial wealth
- a **marginal propensity to consume out of total wealth** equals the interest factor $\frac{r}{1+r}$
- **economic income** equals
  - a constant marginal propensity to consume times the sum of non-financial wealth and financial wealth
  - the amount the consumer can consume while leaving its wealth intact

Responding to the State

The *state* vector confronting the consumer at t is $\begin{bmatrix} b_t & z_t \end{bmatrix}$.

Here

- z_t is an *exogenous* component, unaffected by consumer behavior.
- b_t is an *endogenous* component (since it depends on the decision rule).

Note that z_t contains all variables useful for forecasting the consumer's future endowment.

It is plausible that current decisions c_t and b_{t+1} should be expressible as functions of z_t and b_t.

This is indeed the case.

In fact, from *this discussion*, we see that

$$\sum_{j=0}^{\infty}\beta^j \mathbb{E}_t[y_{t+j}] = \mathbb{E}_t\left[\sum_{j=0}^{\infty}\beta^j y_{t+j}\right] = U(I-\beta A)^{-1} z_t$$

Combining this with (75.9) gives

$$c_t = \frac{r}{1+r}\left[U(I-\beta A)^{-1} z_t - b_t\right] \tag{75.10}$$

Using this equality to eliminate c_t in the budget constraint (75.2) gives

$$\begin{aligned} b_{t+1} &= (1+r)(b_t + c_t - y_t) \\ &= (1+r)b_t + r[U(I-\beta A)^{-1} z_t - b_t] - (1+r)U z_t \\ &= b_t + U[r(I-\beta A)^{-1} - (1+r)I]z_t \\ &= b_t + U(I-\beta A)^{-1}(A-I)z_t \end{aligned}$$

To get from the second last to the last expression in this chain of equalities is not trivial.

A key is to use the fact that $(1+r)\beta = 1$ and $(I-\beta A)^{-1} = \sum_{j=0}^{\infty}\beta^j A^j$.

We've now successfully written c_t and b_{t+1} as functions of b_t and z_t.

A State-Space Representation

We can summarize our dynamics in the form of a linear state-space system governing consumption, debt and income:

$$\begin{aligned} z_{t+1} &= Az_t + Cw_{t+1} \\ b_{t+1} &= b_t + U[(I-\beta A)^{-1}(A-I)]z_t \\ y_t &= Uz_t \\ c_t &= (1-\beta)[U(I-\beta A)^{-1}z_t - b_t] \end{aligned} \tag{75.11}$$

To write this more succinctly, let

$$x_t = \begin{bmatrix} z_t \\ b_t \end{bmatrix}, \quad \tilde{A} = \begin{bmatrix} A & 0 \\ U(I-\beta A)^{-1}(A-I) & 1 \end{bmatrix}, \quad \tilde{C} = \begin{bmatrix} C \\ 0 \end{bmatrix}$$

and

$$\tilde{U} = \begin{bmatrix} U & 0 \\ (1-\beta)U(I-\beta A)^{-1} & -(1-\beta) \end{bmatrix}, \quad \tilde{y}_t = \begin{bmatrix} y_t \\ c_t \end{bmatrix}$$

Then we can express equation (75.11) as

$$\begin{aligned} x_{t+1} &= \tilde{A}x_t + \tilde{C}w_{t+1} \\ \tilde{y}_t &= \tilde{U}x_t \end{aligned} \tag{75.12}$$

We can use the following formulas from *linear state space models* to compute population mean $\mu_t = \mathbb{E}x_t$ and covariance $\Sigma_t := \mathbb{E}[(x_t - \mu_t)(x_t - \mu_t)']$

$$\mu_{t+1} = \tilde{A}\mu_t \quad \text{with} \quad \mu_0 \text{ given} \tag{75.13}$$

$$\Sigma_{t+1} = \tilde{A}\Sigma_t\tilde{A}' + \tilde{C}\tilde{C}' \quad \text{with} \quad \Sigma_0 \text{ given} \tag{75.14}$$

We can then compute the mean and covariance of $\tilde{y}_t$ from

$$\begin{aligned} \mu_{y,t} &= \tilde{U}\mu_t \\ \Sigma_{y,t} &= \tilde{U}\Sigma_t\tilde{U}' \end{aligned} \tag{75.15}$$

A Simple Example with IID Income

To gain some preliminary intuition on the implications of (75.11), let's look at a highly stylized example where income is just IID.

(Later examples will investigate more realistic income streams.)

In particular, let $\{w_t\}_{t=1}^{\infty}$ be IID and scalar standard normal, and let

$$z_t = \begin{bmatrix} z_t^1 \\ 1 \end{bmatrix}, \quad A = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}, \quad U = \begin{bmatrix} 1 & \mu \end{bmatrix}, \quad C = \begin{bmatrix} \sigma \\ 0 \end{bmatrix}$$

Finally, let $b_0 = z_0^1 = 0$.

Under these assumptions, we have $y_t = \mu + \sigma w_t \sim N(\mu, \sigma^2)$.

Further, if you work through the state space representation, you will see that

$$b_t = -\sigma\sum_{j=1}^{t-1} w_j$$

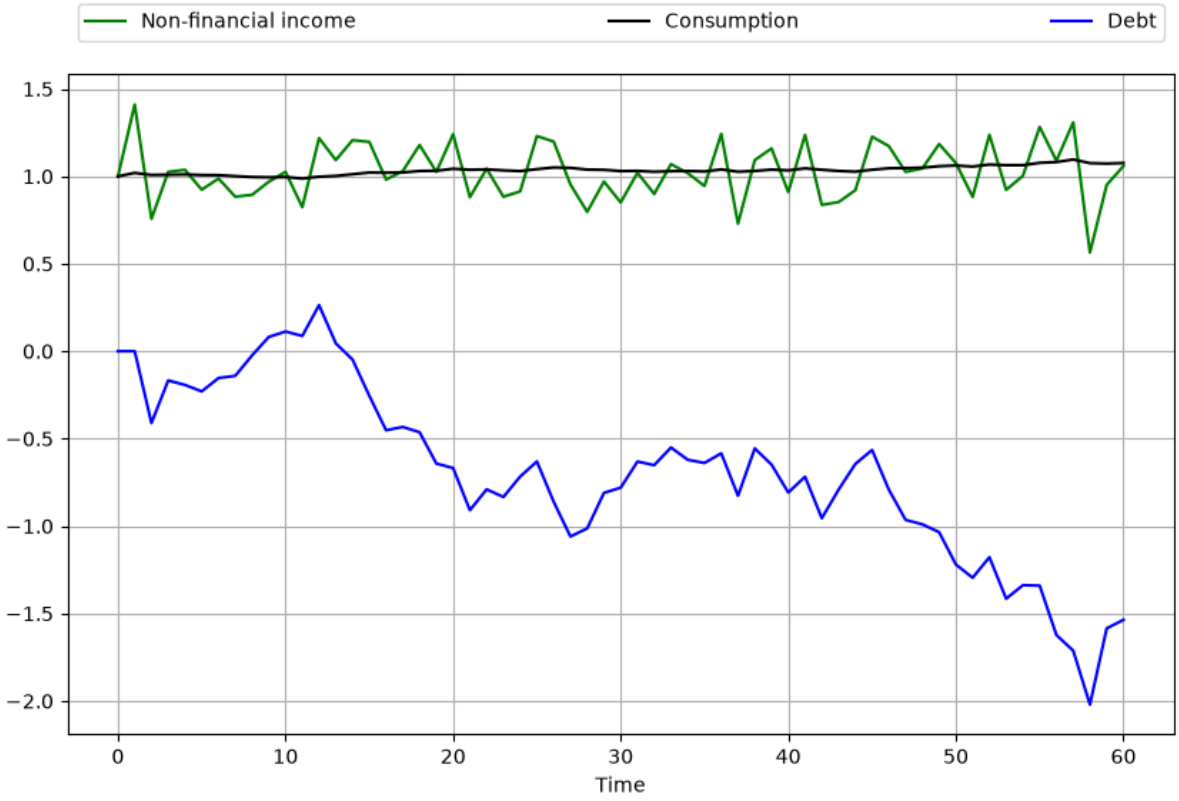
$$c_t = \mu + (1-\beta)\sigma\sum_{j=1}^{t} w_j$$

Thus, income is IID and debt and consumption are both Gaussian random walks.

Defining assets as $-b_t$, we see that assets are just the cumulative sum of unanticipated incomes prior to the present date.

The next figure shows a typical realization with $r = 0.05$, $\mu = 1$, and $\sigma = 0.15$

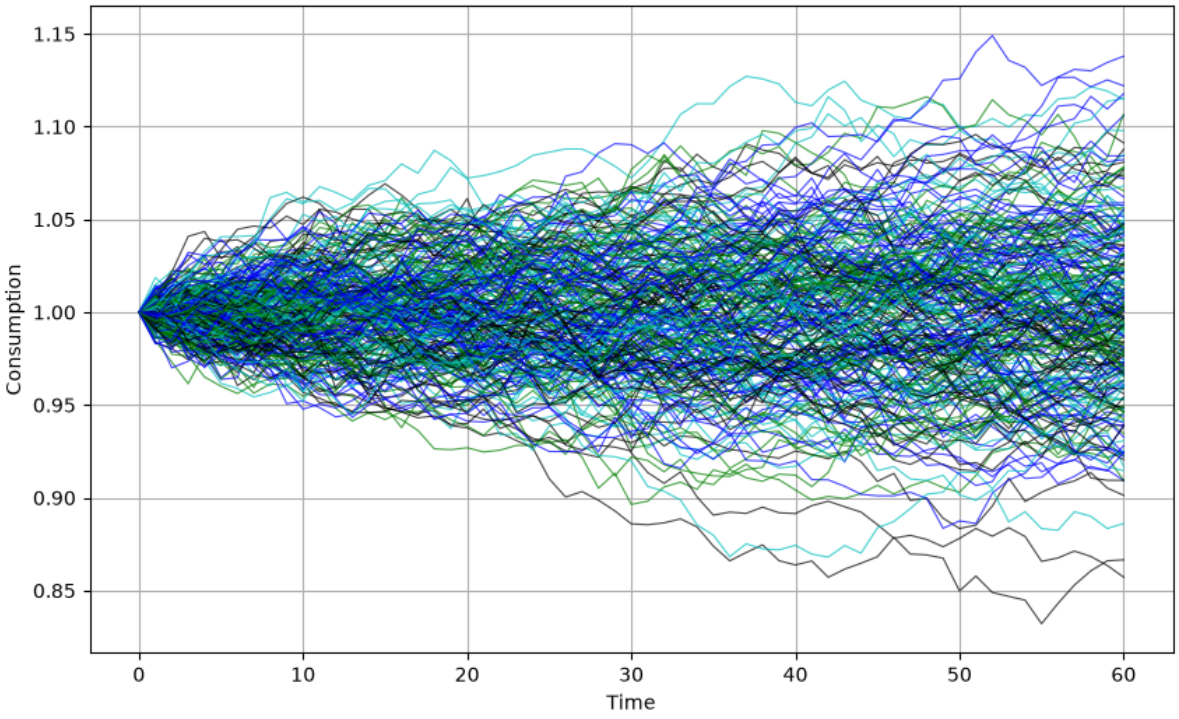
```
r = 0.05
β = 1 / (1 + r)
σ = 0.15
μ = 1
T = 60

@jit
def time_path(T, rng):
    w = rng.standard_normal(T+1)  # w_0, w_1, ..., w_T
    w[0] = 0
    b = np.zeros(T+1)
    for t in range(1, T+1):
        b[t] = w[1:t].sum()
    b = -σ * b
    c = μ + (1 - β) * (σ * w - b)
    return w, b, c

rng = np.random.default_rng()
w, b, c = time_path(T, rng)

fig, ax = plt.subplots(figsize=(10, 6))

ax.plot(μ + σ * w, 'g-', label="Non-financial income")
ax.plot(c, 'k-', label="Consumption")
ax.plot( b, 'b-', label="Debt")
ax.legend(ncol=3, mode='expand', bbox_to_anchor=(0., 1.02, 1., .102))
ax.grid()
ax.set_xlabel('Time')

plt.show()
```



Observe that consumption is considerably smoother than income.

The figure below shows the consumption paths of 250 consumers with independent income streams

```
fig, ax = plt.subplots(figsize=(10, 6))

b_sum = np.zeros(T+1)
for i in range(250):
    w, b, c = time_path(T, rng)  # Generate new time path
    rcolor = random.choice(('c', 'g', 'b', 'k'))
    ax.plot(c, color=rcolor, lw=0.8, alpha=0.7)

ax.grid()
ax.set(xlabel='Time', ylabel='Consumption')

plt.show()
```



75.3 Alternative Representations

In this section, we shed more light on the evolution of savings, debt and consumption by representing their dynamics in several different ways.

75.3.1 Hall's Representation

Hall [Hall, 1978] suggested an insightful way to summarize the implications of LQ permanent income theory.

First, to represent the solution for b_t, shift (75.9) forward one period and eliminate b_{t+1} by using (75.2) to obtain

$$c_{t+1} = (1-\beta)\sum_{j=0}^{\infty}\beta^j \mathbb{E}_{t+1}[y_{t+j+1}] - (1-\beta)\left[\beta^{-1}(c_t + b_t - y_t)\right]$$

If we add and subtract $\beta^{-1}(1-\beta)\sum_{j=0}^{\infty}\beta^j \mathbb{E}_t y_{t+j}$ from the right side of the preceding equation and rearrange, we obtain

$$c_{t+1} - c_t = (1-\beta)\sum_{j=0}^{\infty}\beta^j\left\{\mathbb{E}_{t+1}[y_{t+j+1}] - \mathbb{E}_t[y_{t+j+1}]\right\} \tag{75.16}$$

The right side is the time $t+1$ *innovation to the expected present value* of the endowment process $\{y_t\}$.

We can represent the optimal decision rule for (c_t, b_{t+1}) in the form of (75.16) and (75.8), which we repeat:

$$b_t = \sum_{j=0}^{\infty}\beta^j \mathbb{E}_t[y_{t+j}] - \frac{1}{1-\beta}c_t \tag{75.17}$$

Equation (75.17) asserts that the consumer's debt due at t equals the expected present value of its endowment minus the expected present value of its consumption stream.

A high debt thus indicates a large expected present value of surpluses $y_t - c_t$.

Recalling again our discussion on *forecasting geometric sums*, we have

$$\mathbb{E}_t \sum_{j=0}^{\infty} \beta^j y_{t+j} = U(I - \beta A)^{-1} z_t$$
$$\mathbb{E}_{t+1} \sum_{j=0}^{\infty} \beta^j y_{t+j+1} = U(I - \beta A)^{-1} z_{t+1}$$
$$\mathbb{E}_t \sum_{j=0}^{\infty} \beta^j y_{t+j+1} = U(I - \beta A)^{-1} A z_t$$

Using these formulas together with (75.3) and substituting into (75.16) and (75.17) gives the following representation for the consumer's optimum decision rule:

$$\begin{aligned} c_{t+1} &= c_t + (1-\beta) U(I - \beta A)^{-1} C w_{t+1} \\ b_t &= U(I - \beta A)^{-1} z_t - \frac{1}{1-\beta} c_t \\ y_t &= U z_t \\ z_{t+1} &= A z_t + C w_{t+1} \end{aligned} \tag{75.18}$$

Representation (75.18) makes clear that

- The state can be taken as (c_t, z_t).
  - The endogenous part is c_t and the exogenous part is z_t.
  - Debt b_t has disappeared as a component of the state because it is encoded in c_t.
- Consumption is a random walk with innovation $(1-\beta) U(I - \beta A)^{-1} C w_{t+1}$.
  - This is a more explicit representation of the martingale result in (75.6).

75.3.2 Cointegration

Representation (75.18) reveals that the joint process $\{c_t, b_t\}$ possesses the property that Engle and Granger [Engle and Granger, 1987] called cointegration.

Cointegration is a tool that allows us to apply powerful results from the theory of stationary stochastic processes to (certain transformations of) nonstationary models.

To apply cointegration in the present context, suppose that z_t is asymptotically stationary[3].

Despite this, both c_t and b_t will be non-stationary because they have unit roots (see (75.11) for b_t).

Nevertheless, there is a linear combination of c_t, b_t that *is* asymptotically stationary.

In particular, from the second equality in (75.18) we have

$$(1-\beta) b_t + c_t = (1-\beta) U(I - \beta A)^{-1} z_t \tag{75.19}$$

Hence the linear combination $(1-\beta) b_t + c_t$ is asymptotically stationary.

Accordingly, Granger and Engle would call $\begin{bmatrix} (1-\beta) & 1 \end{bmatrix}$ a **cointegrating vector** for the state.

When applied to the nonstationary vector process $\begin{bmatrix} b_t & c_t \end{bmatrix}'$, it yields a process that is asymptotically stationary.

[3] This would be the case if, for example, the *spectral radius* of A is strictly less than one.

Equation (75.19) can be rearranged to take the form

$$(1-\beta)b_t + c_t = (1-\beta)\mathbb{E}_t \sum_{j=0}^{\infty} \beta^j y_{t+j} \tag{75.20}$$

Equation (75.20) asserts that the *cointegrating residual* on the left side equals the conditional expectation of the geometric sum of future incomes on the right[4].

75.3.3 Cross-Sectional Implications

Consider again (75.18), this time in light of our discussion of distribution dynamics in the *lecture on linear systems*.

The dynamics of c_t are given by

$$c_{t+1} = c_t + (1-\beta)U(I-\beta A)^{-1}Cw_{t+1} \tag{75.21}$$

or

$$c_t = c_0 + \sum_{j=1}^{t} \hat{w}_j \quad \text{for} \quad \hat{w}_{t+1} := (1-\beta)U(I-\beta A)^{-1}Cw_{t+1}$$

The unit root affecting c_t causes the time t variance of c_t to grow linearly with t.

In particular, since $\{\hat{w}_t\}$ is IID, we have

$$\text{Var}[c_t] = \text{Var}[c_0] + t\,\hat{\sigma}^2 \tag{75.22}$$

where

$$\hat{\sigma}^2 := (1-\beta)^2 U(I-\beta A)^{-1}CC'(I-\beta A')^{-1}U'$$

When $\hat{\sigma} > 0$, $\{c_t\}$ has no asymptotic distribution.

Let's consider what this means for a cross-section of ex-ante identical consumers born at time 0.

Let the distribution of c_0 represent the cross-section of initial consumption values.

Equation (75.22) tells us that the variance of c_t increases over time at a rate proportional to t.

A number of different studies have investigated this prediction and found some support for it (see, e.g., [Deaton and Paxson, 1994], [Storesletten *et al.*, 2004]).

75.3.4 Impulse Response Functions

Impulse response functions measure responses to various impulses (i.e., temporary shocks).

The impulse response function of $\{c_t\}$ to the innovation $\{w_t\}$ is a box.

In particular, the response of c_{t+j} to a unit increase in the innovation w_{t+1} is $(1-\beta)U(I-\beta A)^{-1}C$ for all $j \geq 1$.

[4] See [Campbell and Shiller, 1988], [Lettau and Ludvigson, 2001], [Lettau and Ludvigson, 2004] for interesting applications of related ideas.

75.3.5 Moving Average Representation

It's useful to express the innovation to the expected present value of the endowment process in terms of a moving average representation for income y_t.

The endowment process defined by (75.3) has the moving average representation

$$y_{t+1} = d(L)w_{t+1} \tag{75.23}$$

where

- $d(L) = \sum_{j=0}^{\infty} d_j L^j$ for some sequence d_j, where L is the lag operator[5]
- at time t, the consumer has an information set[6] $w^t = [w_t, w_{t-1}, ...]$

Notice that

$$y_{t+j} - \mathbb{E}_t[y_{t+j}] = d_0 w_{t+j} + d_1 w_{t+j-1} + \cdots + d_{j-1} w_{t+1}$$

It follows that

$$\mathbb{E}_{t+1}[y_{t+j}] - \mathbb{E}_t[y_{t+j}] = d_{j-1} w_{t+1} \tag{75.24}$$

Using (75.24) in (75.16) gives

$$c_{t+1} - c_t = (1-\beta)d(\beta)w_{t+1} \tag{75.25}$$

The object $d(\beta)$ is the **present value of the moving average coefficients** in the representation for the endowment process y_t.

75.4 Two Classic Examples

We illustrate some of the preceding ideas with two examples.

In both examples, the endowment follows the process $y_t = z_{1t} + z_{2t}$ where

$$\begin{bmatrix} z_{1t+1} \\ z_{2t+1} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} z_{1t} \\ z_{2t} \end{bmatrix} + \begin{bmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{bmatrix} \begin{bmatrix} w_{1t+1} \\ w_{2t+1} \end{bmatrix}$$

Here

- w_{t+1} is an IID 2×1 process distributed as $N(0, I)$.
- z_{1t} is a permanent component of y_t.
- z_{2t} is a purely transitory component of y_t.

75.4.1 Example 1

Assume as before that the consumer observes the state z_t at time t.

In view of (75.18) we have

$$c_{t+1} - c_t = \sigma_1 w_{1t+1} + (1-\beta)\sigma_2 w_{2t+1} \tag{75.26}$$

Formula (75.26) shows how an increment $\sigma_1 w_{1t+1}$ to the permanent component of income z_{1t+1} leads to

[5] Representation (75.3) implies that $d(L) = U(I - AL)^{-1}C$.

[6] A moving average representation for a process y_t is said to be **fundamental** if the linear space spanned by y^t is equal to the linear space spanned by w^t. A time-invariant innovations representation, attained via the Kalman filter, is by construction fundamental.

- a permanent one-for-one increase in consumption and
- no increase in savings $-b_{t+1}$

But the purely transitory component of income $\sigma_2 w_{2t+1}$ leads to a permanent increment in consumption by a fraction $1-\beta$ of transitory income.

The remaining fraction β is saved, leading to a permanent increment in $-b_{t+1}$.

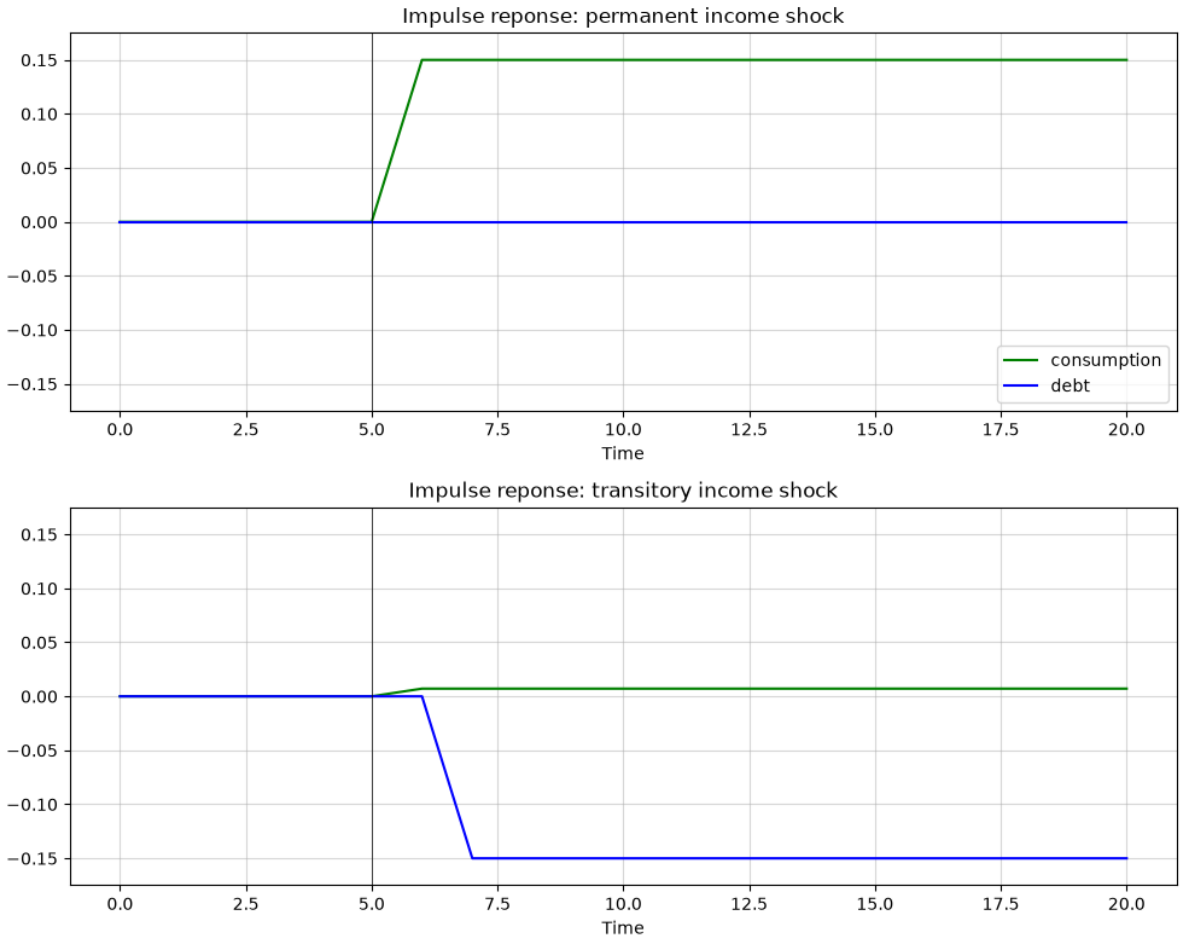
Application of the formula for debt in (75.11) to this example shows that

$$b_{t+1} - b_t = -z_{2t} = -\sigma_2 w_{2t} \tag{75.27}$$

This confirms that none of $\sigma_1 w_{1t}$ is saved, while all of $\sigma_2 w_{2t}$ is saved.

The next figure displays impulse-response functions that illustrates these very different reactions to transitory and permanent income shocks.

```
r = 0.05
β = 1 / (1 + r)
S = 5   # Impulse date
σ1 = σ2 = 0.15

@jit
def time_path(T, permanent=False):
    "Time path of consumption and debt given shock sequence"
    w1 = np.zeros(T+1)
    w2 = np.zeros(T+1)
    b = np.zeros(T+1)
    c = np.zeros(T+1)
    if permanent:
        w1[S+1] = 1.0
    else:
        w2[S+1] = 1.0
    for t in range(1, T):
        b[t+1] = b[t] - σ2 * w2[t]
        c[t+1] = c[t] + σ1 * w1[t+1] + (1 - β) * σ2 * w2[t+1]
    return b, c


fig, axes = plt.subplots(2, 1, figsize=(10, 8))
titles = ['permanent', 'transitory']

L = 0.175

for ax, truefalse, title in zip(axes, (True, False), titles):
    b, c = time_path(T=20, permanent=truefalse)
    ax.set_title(f'Impulse reponse: {title} income shock')
    ax.plot(c, 'g-', label="consumption")
    ax.plot(b, 'b-', label="debt")
    ax.plot((S, S), (-L, L), 'k-', lw=0.5)
    ax.grid(alpha=0.5)
    ax.set(xlabel=r'Time', ylim=(-L, L))

axes[0].legend(loc='lower right')

plt.tight_layout()
plt.show()
```



Notice how the permanent income shock provokes no change in assets $-b_{t+1}$ and an immediate permanent change in consumption equal to the permanent increment in non-financial income.

In contrast, notice how most of a transitory income shock is saved and only a small amount is saved.

The box-like impulse responses of consumption to both types of shock reflect the random walk property of the optimal consumption decision.

75.4.2 Example 2

Assume now that at time t the consumer observes y_t, and its history up to t, but not z_t.

Under this assumption, it is appropriate to use an *innovation representation* to form A, C, U in (75.18).

The discussion in sections 2.9.1 and 2.11.3 of [Ljungqvist and Sargent, 2018] shows that the pertinent state space representation for y_t is

$$\begin{bmatrix} y_{t+1} \\ a_{t+1} \end{bmatrix} = \begin{bmatrix} 1 & -(1-K) \\ 0 & 0 \end{bmatrix} \begin{bmatrix} y_t \\ a_t \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \end{bmatrix} a_{t+1}$$

$$y_t = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} y_t \\ a_t \end{bmatrix}$$

where

- $K :=$ the stationary Kalman gain

- $a_t := y_t - E[y_t \mid y_{t-1}, \ldots, y_0]$

In the same discussion in [Ljungqvist and Sargent, 2018] it is shown that $K \in [0, 1]$ and that K increases as σ_1/σ_2 does.

In other words, K increases as the ratio of the standard deviation of the permanent shock to that of the transitory shock increases.

Please see *first look at the Kalman filter*.

Applying formulas (75.18) implies

$$c_{t+1} - c_t = [1 - \beta(1 - K)]a_{t+1} \tag{75.28}$$

where the endowment process can now be represented in terms of the univariate innovation to y_t as

$$y_{t+1} - y_t = a_{t+1} - (1 - K)a_t \tag{75.29}$$

Equation (75.29) indicates that the consumer regards

- fraction K of an innovation a_{t+1} to y_{t+1} as *permanent*
- fraction $1 - K$ as purely transitory

The consumer permanently increases his consumption by the full amount of his estimate of the permanent part of a_{t+1}, but by only $(1 - \beta)$ times his estimate of the purely transitory part of a_{t+1}.

Therefore, in total, he permanently increments his consumption by a fraction $K + (1 - \beta)(1 - K) = 1 - \beta(1 - K)$ of a_{t+1}.

He saves the remaining fraction $\beta(1 - K)$.

According to equation (75.29), the first difference of income is a first-order moving average.

Equation (75.28) asserts that the first difference of consumption is IID.

Application of formula to this example shows that

$$b_{t+1} - b_t = (K - 1)a_t \tag{75.30}$$

This indicates how the fraction K of the innovation to y_t that is regarded as permanent influences the fraction of the innovation that is saved.

75.5 Further Reading

The model described above significantly changed how economists think about consumption.

While Hall's model does a remarkably good job as a first approximation to consumption data, it's widely believed that it doesn't capture important aspects of some consumption/savings data.

For example, liquidity constraints and precautionary savings appear to be present sometimes.

Further discussion can be found in, e.g., [Hall and Mishkin, 1982], [Parker, 1999], [Deaton, 1991], [Carroll, 2001].

75.6 Appendix: The Euler Equation

Where does the first-order condition (75.5) come from?

Here we'll give a proof for the two-period case, which is representative of the general argument.

The finite horizon equivalent of the no-Ponzi condition is that the agent cannot end her life in debt, so $b_2 = 0$.

From the budget constraint (75.2) we then have

$$c_0 = \frac{b_1}{1+r} - b_0 + y_0 \quad \text{and} \quad c_1 = y_1 - b_1$$

Here b_0 and y_0 are given constants.

Substituting these constraints into our two-period objective $u(c_0) + \beta \mathbb{E}_0[u(c_1)]$ gives

$$\max_{b_1} \left\{ u\left(\frac{b_1}{R} - b_0 + y_0\right) + \beta \, \mathbb{E}_0[u(y_1 - b_1)] \right\}$$

You will be able to verify that the first-order condition is

$$u'(c_0) = \beta R \, \mathbb{E}_0[u'(c_1)]$$

Using $\beta R = 1$ gives (75.5) in the two-period case.

The proof for the general case is similar.

CHAPTER

SEVENTYSIX

PERMANENT INCOME II: LQ TECHNIQUES

Contents

- *Permanent Income II: LQ Techniques*
  - *Overview*
  - *Setup*
  - *The LQ Approach*
  - *Implementation*
  - *Two Example Economies*

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install quantecon
```

76.1 Overview

This lecture continues our analysis of the linear-quadratic (LQ) permanent income model of savings and consumption.

As we saw in our *previous lecture* on this topic, Robert Hall [Hall, 1978] used the LQ permanent income model to restrict and interpret intertemporal comovements of nondurable consumption, nonfinancial income, and financial wealth.

For example, we saw how the model asserts that for any covariance stationary process for nonfinancial income

- consumption is a random walk
- financial wealth has a unit root and is cointegrated with consumption

Other applications use the same LQ framework.

For example, a model isomorphic to the LQ permanent income model has been used by Robert Barro [Barro, 1979] to interpret intertemporal comovements of a government's tax collections, its expenditures net of debt service, and its public debt.

This isomorphism means that in analyzing the LQ permanent income model, we are in effect also analyzing the Barro tax smoothing model.

It is just a matter of appropriately relabeling the variables in Hall's model.

In this lecture, we'll

- show how the solution to the LQ permanent income model can be obtained using LQ control methods.
- represent the model as a linear state space system as in *this lecture*.
- apply QuantEcon's LinearStateSpace class to characterize statistical features of the consumer's optimal consumption and borrowing plans.

We'll then use these characterizations to construct a simple model of cross-section wealth and consumption dynamics in the spirit of Truman Bewley [Bewley, 1986].

(Later we'll study other Bewley models—see *this lecture*.)

The model will prove useful for illustrating concepts such as

- stationarity
- ergodicity
- ensemble moments and cross-section observations

Let's start with some imports:

```
import matplotlib.pyplot as plt
import quantecon as qe
import numpy as np
import scipy.linalg as la
```

76.2 Setup

Let's recall the basic features of the model discussed in the *permanent income model*.

Consumer preferences are ordered by

$$E_0 \sum_{t=0}^{\infty} \beta^t u(c_t) \tag{76.1}$$

where $u(c) = -(c - \gamma)^2$.

The consumer maximizes (76.1) by choosing a consumption, borrowing plan $\{c_t, b_{t+1}\}_{t=0}^{\infty}$ subject to the sequence of budget constraints

$$c_t + b_t = \frac{1}{1+r} b_{t+1} + y_t, \quad t \geq 0 \tag{76.2}$$

and the no-Ponzi condition

$$E_0 \sum_{t=0}^{\infty} \beta^t b_t^2 < \infty \tag{76.3}$$

The interpretation of all variables and parameters are the same as in the *previous lecture*.

We continue to assume that $(1+r)\beta = 1$.

The dynamics of $\{y_t\}$ again follow the linear state space model

$$\begin{aligned} z_{t+1} &= A z_t + C w_{t+1} \\ y_t &= U z_t \end{aligned} \tag{76.4}$$

The restrictions on the shock process and parameters are the same as in our *previous lecture*.

76.2.1 Digression on a Useful Isomorphism

The LQ permanent income model of consumption is mathematically isomorphic with a version of Barro's [Barro, 1979] model of tax smoothing.

In the LQ permanent income model

- the household faces an exogenous process of nonfinancial income
- the household wants to smooth consumption across states and time

In the Barro tax smoothing model

- a government faces an exogenous sequence of government purchases (net of interest payments on its debt)
- a government wants to smooth tax collections across states and time

If we set

- T_t, total tax collections in Barro's model to consumption c_t in the LQ permanent income model.
- G_t, exogenous government expenditures in Barro's model to nonfinancial income y_t in the permanent income model.
- B_t, government risk-free one-period assets falling due in Barro's model to risk-free one-period consumer debt b_t falling due in the LQ permanent income model.
- R, the gross rate of return on risk-free one-period government debt in Barro's model to the gross rate of return $1 + r$ on financial assets in the permanent income model of consumption.

then the two models are mathematically equivalent.

All characterizations of a $\{c_t, y_t, b_t\}$ in the LQ permanent income model automatically apply to a $\{T_t, G_t, B_t\}$ process in the Barro model of tax smoothing.

See consumption and tax smoothing models for further exploitation of an isomorphism between consumption and tax smoothing models.

76.2.2 A Specification of the Nonfinancial Income Process

For the purposes of this lecture, let's assume $\{y_t\}$ is a second-order univariate autoregressive process:

$$y_{t+1} = \alpha + \rho_1 y_t + \rho_2 y_{t-1} + \sigma w_{t+1}$$

We can map this into the linear state space framework in (76.4), as discussed in our lecture on *linear models*.

To do so we take

$$z_t = \begin{bmatrix} 1 \\ y_t \\ y_{t-1} \end{bmatrix}, \quad A = \begin{bmatrix} 1 & 0 & 0 \\ \alpha & \rho_1 & \rho_2 \\ 0 & 1 & 0 \end{bmatrix}, \quad C = \begin{bmatrix} 0 \\ \sigma \\ 0 \end{bmatrix}, \quad \text{and} \quad U = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}$$

76.3 The LQ Approach

Previously we solved the permanent income model by solving a system of linear expectational difference equations subject to two boundary conditions.

Here we solve the same model using *LQ methods* based on dynamic programming.

After confirming that answers produced by the two methods agree, we apply QuantEcon's LinearStateSpace class to illustrate features of the model.

Why solve a model in two distinct ways?

Because by doing so we gather insights about the structure of the model.

Our earlier approach based on solving a system of expectational difference equations brought to the fore the role of the consumer's expectations about future nonfinancial income.

On the other hand, formulating the model in terms of an LQ dynamic programming problem reminds us that

- finding the state (of a dynamic programming problem) is an art, and
- iterations on a Bellman equation implicitly jointly solve both a forecasting problem and a control problem

76.3.1 The LQ Problem

Recall from our *lecture on LQ theory* that the optimal linear regulator problem is to choose a decision rule for u_t to minimize

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t \{x_t' R x_t + u_t' Q u_t\},$$

subject to x_0 given and the law of motion

$$x_{t+1} = \tilde{A} x_t + \tilde{B} u_t + \tilde{C} w_{t+1}, \qquad t \geq 0, \tag{76.5}$$

where w_{t+1} is IID with mean vector zero and $\mathbb{E} w_t w_t' = I$.

The tildes in $\tilde{A}, \tilde{B}, \tilde{C}$ are to avoid clashing with notation in (76.4).

The value function for this problem is $v(x) = -x' P x - d$, where

- P is the unique positive semidefinite solution of the *corresponding matrix Riccati equation*.
- The scalar d is given by $d = \beta(1-\beta)^{-1} \text{trace}(P \tilde{C} \tilde{C}')$.

The optimal policy is $u_t = -F x_t$, where $F := \beta(Q + \beta \tilde{B}' P \tilde{B})^{-1} \tilde{B}' P \tilde{A}$.

Under an optimal decision rule F, the state vector x_t evolves according to $x_{t+1} = (\tilde{A} - \tilde{B}F) x_t + \tilde{C} w_{t+1}$.

76.3.2 Mapping into the LQ Framework

To map into the LQ framework, we'll use

$$x_t := \begin{bmatrix} z_t \\ b_t \end{bmatrix} = \begin{bmatrix} 1 \\ y_t \\ y_{t-1} \\ b_t \end{bmatrix}$$

as the state vector and $u_t := c_t - \gamma$ as the control.

With this notation and $U_\gamma := \begin{bmatrix}\gamma & 0 & 0\end{bmatrix}$, we can write the state dynamics as in (76.5) when

$$\tilde{A} := \begin{bmatrix} A & 0 \\ (1+r)(U_\gamma - U) & 1+r \end{bmatrix} \quad \tilde{B} := \begin{bmatrix} 0 \\ 1+r \end{bmatrix} \quad \text{and} \quad \tilde{C} := \begin{bmatrix} C \\ 0 \end{bmatrix} w_{t+1}$$

Please confirm for yourself that, with these definitions, the LQ dynamics (76.5) match the dynamics of z_t and b_t described above.

To map utility into the quadratic form $x_t' R x_t + u_t' Q u_t$ we can set

- $Q := 1$ (remember that we are minimizing) and
- $R :=$ a 4×4 matrix of zeros

However, there is one problem remaining.

We have no direct way to capture the non-recursive restriction (76.3) on the debt sequence $\{b_t\}$ from within the LQ framework.

To try to enforce it, we're going to use a trick: put a small penalty on b_t^2 in the criterion function.

In the present setting, this means adding a small entry $\epsilon > 0$ in the $(4, 4)$ position of R.

That will induce a (hopefully) small approximation error in the decision rule.

We'll check whether it really is small numerically soon.

76.4 Implementation

Let's write some code to solve the model.

One comment before we start is that the bliss level of consumption γ in the utility function has no effect on the optimal decision rule.

We saw this in the previous lecture *permanent income*.

The reason is that it drops out of the Euler equation for consumption.

In what follows we set it equal to unity.

76.4.1 The Exogenous Nonfinancial Income Process

First, we create the objects for the optimal linear regulator

```
# Set parameters
α, β, ρ1, ρ2, σ = 10.0, 0.95, 0.9, 0.0, 1.0

R = 1 / β
A = np.array([[1., 0., 0.],
              [α,  ρ1, ρ2],
              [0., 1., 0.]])
C = np.array([[0.], [σ], [0.]])
G = np.array([[0., 1., 0.]])

# Form LinearStateSpace system and pull off steady state moments
μ_z0 = np.array([[1.0], [0.0], [0.0]])
Σ_z0 = np.zeros((3, 3))
Lz = qe.LinearStateSpace(A, C, G, mu_0=μ_z0, Sigma_0=Σ_z0)
μ_z, μ_y, Σ_z, Σ_y, Σ_yx = Lz.stationary_distributions()
```

(continues on next page)

(continued from previous page)

```
# Mean vector of state for the savings problem
mxo = np.vstack([μ_z, 0.0])

# Create stationary covariance matrix of x -- start everyone off at b=0
a1 = np.zeros((3, 1))
aa = np.hstack([Σ_z, a1])
bb = np.zeros((1, 4))
sxo = np.vstack([aa, bb])

# These choices will initialize the state vector of an individual at zero
# debt and the ergodic distribution of the endowment process. Use these to
# create the Bewley economy.
mxbewley = mxo
sxbewley = sxo
```

The next step is to create the matrices for the LQ system

```
A12 = np.zeros((3,1))
ALQ_l = np.hstack([A, A12])
ALQ_r = np.array([[0, -R, 0, R]])
ALQ = np.vstack([ALQ_l, ALQ_r])

RLQ = np.array([[0., 0., 0., 0.],
                [0., 0., 0., 0.],
                [0., 0., 0., 0.],
                [0., 0., 0., 1e-9]])

QLQ = np.array([1.0])
BLQ = np.array([0., 0., 0., R]).reshape(4,1)
CLQ = np.array([0., σ, 0., 0.]).reshape(4,1)
β_LQ = β
```

Let's print these out and have a look at them

```
print(f"A = \n {ALQ}")
print(f"B = \n {BLQ}")
print(f"R = \n {RLQ}")
print(f"Q = \n {QLQ}")
```

```
A =
 [[ 1.          0.          0.          0.        ]
 [10.          0.9         0.          0.        ]
 [ 0.          1.          0.          0.        ]
 [ 0.         -1.05263158  0.          1.05263158]]
B =
 [[0.        ]
 [0.        ]
 [0.        ]
 [1.05263158]]
R =
 [[0.e+00 0.e+00 0.e+00 0.e+00]
 [0.e+00 0.e+00 0.e+00 0.e+00]
 [0.e+00 0.e+00 0.e+00 0.e+00]
 [0.e+00 0.e+00 0.e+00 1.e-09]]
Q =
 [1.]
```

Now create the appropriate instance of an LQ model

```
lqpi = qe.LQ(QLQ, RLQ, ALQ, BLQ, C=CLQ, beta=β_LQ)
```

We'll save the implied optimal policy function soon compare them with what we get by employing an alternative solution method

```
P, F, d = lqpi.stationary_values()  # Compute value function and decision rule
ABF = ALQ - BLQ @ F  #  Form closed loop system
```

76.4.2 Comparison with the Difference Equation Approach

In our *first lecture* on the infinite horizon permanent income problem we used a different solution method.

The method was based around

- deducing the Euler equations that are the first-order conditions with respect to consumption and savings.
- using the budget constraints and boundary condition to complete a system of expectational linear difference equations.
- solving those equations to obtain the solution.

Expressed in state space notation, the solution took the form

$$
\begin{aligned}
z_{t+1} &= Az_t + Cw_{t+1} \\
b_{t+1} &= b_t + U[(I - \beta A)^{-1}(A - I)]z_t \\
y_t &= Uz_t \\
c_t &= (1 - \beta)[U(I - \beta A)^{-1}z_t - b_t]
\end{aligned}
$$

Now we'll apply the formulas in this system

```
# Use the above formulas to create the optimal policies for b_{t+1} and c_t
b_pol = G @ la.inv(np.eye(3, 3) - β * A) @ (A - np.eye(3, 3))
c_pol = (1 - β) * G @ la.inv(np.eye(3, 3) - β * A)

# Create the A matrix for a LinearStateSpace instance
A_LSS1 = np.vstack([A, b_pol])
A_LSS2 = np.eye(4, 1, -3)
A_LSS = np.hstack([A_LSS1, A_LSS2])

# Create the C matrix for LSS methods
C_LSS = np.vstack([C, np.zeros(1)])

# Create the G matrix for LSS methods
G_LSS1 = np.vstack([G, c_pol])
G_LSS2 = np.vstack([np.zeros(1), -(1 - β)])
G_LSS = np.hstack([G_LSS1, G_LSS2])

# Use the following values to start everyone off at b=0, initial incomes zero
μ_0 = np.array([1., 0., 0., 0.])
Σ_0 = np.zeros((4, 4))
```

`A_LSS` calculated as we have here should equal `ABF` calculated above using the LQ model

```
ABF - A_LSS
```

```
array([[ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00],
       [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00],
       [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00],
       [-9.51248364e-06,  9.51247768e-08,  0.00000000e+00,
        -1.99999900e-08]])
```

Now compare pertinent elements of `c_pol` and `F`

```
print(c_pol, "\n", -F)
```

```
[[65.51724138  0.34482759  0.        ]]
 [[ 6.55172323e+01  3.44827677e-01 -0.00000000e+00 -5.00000190e-02]]
```

We have verified that the two methods give the same solution.

Now let's create instances of the LinearStateSpace class and use it to do some interesting experiments.

To do this, we'll use the outcomes from our second method.

76.5 Two Example Economies

In the spirit of Bewley models [Bewley, 1986], we'll generate panels of consumers.

The examples differ only in the initial states with which we endow the consumers.

All other parameter values are kept the same in the two examples

- In the first example, all consumers begin with zero nonfinancial income and zero debt.
  - The consumers are thus *ex-ante* identical.
- In the second example, while all begin with zero debt, we draw their initial income levels from the invariant distribution of financial income.
  - Consumers are *ex-ante* heterogeneous.

In the first example, consumers' nonfinancial income paths display pronounced transients early in the sample

- these will affect outcomes in striking ways

Those transient effects will not be present in the second example.

We use methods affiliated with the LinearStateSpace class to simulate the model.

76.5.1 First Set of Initial Conditions

We generate 25 paths of the exogenous non-financial income process and the associated optimal consumption and debt paths.

In the first set of graphs, darker lines depict a particular sample path, while the lighter lines describe 24 other paths.

A second graph plots a collection of simulations against the population distribution that we extract from the `LinearStateSpace` instance `LSS`.

Comparing sample paths with population distributions at each date t is a useful exercise—see *our discussion* of the laws of large numbers

```
lss = qe.LinearStateSpace(A_LSS, C_LSS, G_LSS, mu_0=μ_0, Sigma_0=Σ_0)
```

76.5.2 Population and Sample Panels

In the code below, we use the LinearStateSpace class to

- compute and plot population quantiles of the distributions of consumption and debt for a population of consumers.
- simulate a group of 25 consumers and plot sample paths on the same graph as the population distribution.

```
def income_consumption_debt_series(A, C, G, μ_0, Σ_0, T=150, npaths=25):
    """
    This function takes initial conditions (μ_0, Σ_0) and uses the
    LinearStateSpace class from QuantEcon to  simulate an economy
    npaths times for T periods. It then uses that information to
    generate some graphs related to the discussion below.
    """
    lss = qe.LinearStateSpace(A, C, G, mu_0=μ_0, Sigma_0=Σ_0)

    # Simulation/Moment Parameters
    moment_generator = lss.moment_sequence()

    # Simulate various paths
    bsim = np.empty((npaths, T))
    csim = np.empty((npaths, T))
    ysim = np.empty((npaths, T))

    for i in range(npaths):
        sims = lss.simulate(T)
        bsim[i, :] = sims[0][-1, :]
        csim[i, :] = sims[1][1, :]
        ysim[i, :] = sims[1][0, :]

    # Get the moments
    cons_mean = np.empty(T)
    cons_var = np.empty(T)
    debt_mean = np.empty(T)
    debt_var = np.empty(T)
    for t in range(T):
        μ_x, μ_y, Σ_x, Σ_y = next(moment_generator)
        cons_mean[t], cons_var[t] = μ_y[1,0], Σ_y[1, 1]
        debt_mean[t], debt_var[t] = μ_x[3,0], Σ_x[3, 3]

    return bsim, csim, ysim, cons_mean, cons_var, debt_mean, debt_var

def consumption_income_debt_figure(bsim, csim, ysim):

    # Get T
    T =  bsim.shape[1]

    # Create the first figure
    fig, ax = plt.subplots(2, 1, figsize=(10, 8))
    xvals = np.arange(T)

    # Plot consumption and income
    ax[0].plot(csim[0, :], label="c", color="b")
```

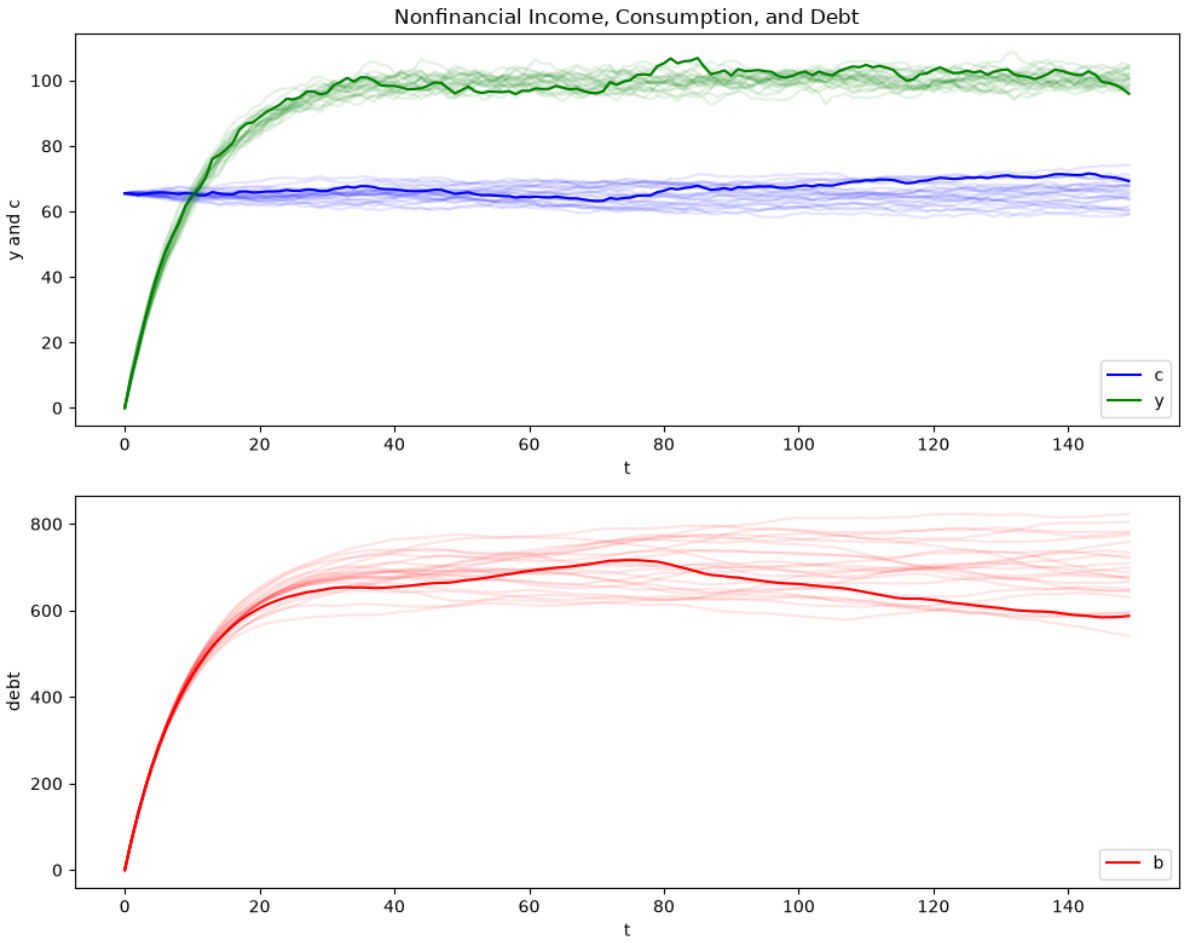
(continues on next page)

(continued from previous page)

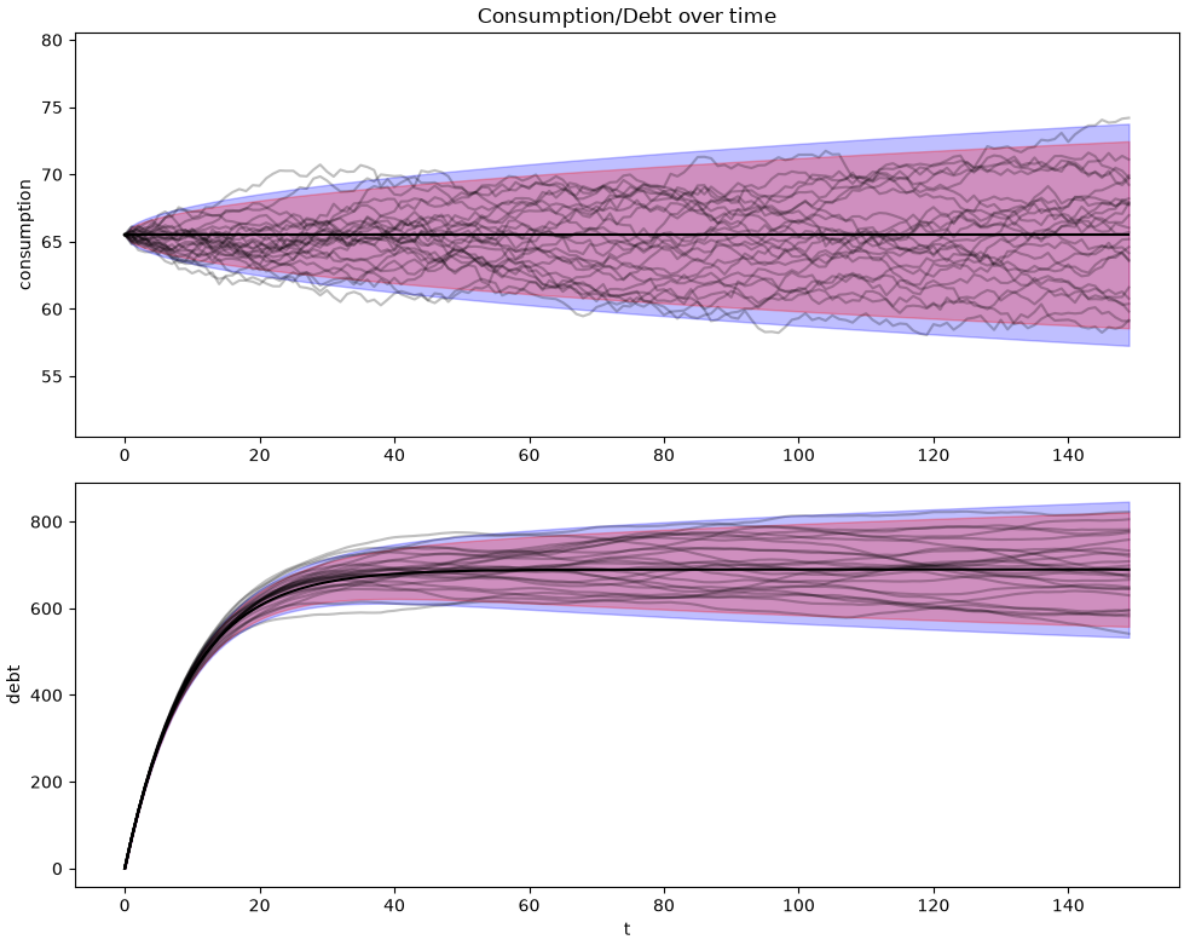
```
    ax[0].plot(ysim[0, :], label="y", color="g")
    ax[0].plot(csim.T, alpha=.1, color="b")
    ax[0].plot(ysim.T, alpha=.1, color="g")
    ax[0].legend(loc=4)
    ax[0].set(title="Nonfinancial Income, Consumption, and Debt",
              xlabel="t", ylabel="y and c")

    # Plot debt
    ax[1].plot(bsim[0, :], label="b", color="r")
    ax[1].plot(bsim.T, alpha=.1, color="r")
    ax[1].legend(loc=4)
    ax[1].set(xlabel="t", ylabel="debt")

    fig.tight_layout()
    return fig

def consumption_debt_fanchart(csim, cons_mean, cons_var,
                              bsim, debt_mean, debt_var):
    # Get T
    T =  bsim.shape[1]

    # Create percentiles of cross-section distributions
    cmean = np.mean(cons_mean)
    c90 = 1.65 * np.sqrt(cons_var)
    c95 = 1.96 * np.sqrt(cons_var)
    c_perc_95p, c_perc_95m = cons_mean + c95, cons_mean - c95
    c_perc_90p, c_perc_90m = cons_mean + c90, cons_mean - c90

    # Create percentiles of cross-section distributions
    dmean = np.mean(debt_mean)
    d90 = 1.65 * np.sqrt(debt_var)
    d95 = 1.96 * np.sqrt(debt_var)
    d_perc_95p, d_perc_95m = debt_mean + d95, debt_mean - d95
    d_perc_90p, d_perc_90m = debt_mean + d90, debt_mean - d90


    # Create second figure
    fig, ax = plt.subplots(2, 1, figsize=(10, 8))
    xvals = np.arange(T)

    # Consumption fan
    ax[0].plot(xvals, cons_mean, color="k")
    ax[0].plot(csim.T, color="k", alpha=.25)
    ax[0].fill_between(xvals, c_perc_95m, c_perc_95p, alpha=.25, color="b")
    ax[0].fill_between(xvals, c_perc_90m, c_perc_90p, alpha=.25, color="r")
    ax[0].set(title="Consumption/Debt over time",
              ylim=(cmean-15, cmean+15), ylabel="consumption")

    # Debt fan
    ax[1].plot(xvals, debt_mean, color="k")
    ax[1].plot(bsim.T, color="k", alpha=.25)
    ax[1].fill_between(xvals, d_perc_95m, d_perc_95p, alpha=.25, color="b")
    ax[1].fill_between(xvals, d_perc_90m, d_perc_90p, alpha=.25, color="r")
    ax[1].set(xlabel="t", ylabel="debt")

    fig.tight_layout()
    return fig
```

Now let's create figures with initial conditions of zero for y_0 and b_0

```
out = income_consumption_debt_series(A_LSS, C_LSS, G_LSS, μ_0, Σ_0)
bsim0, csim0, ysim0 = out[:3]
cons_mean0, cons_var0, debt_mean0, debt_var0 = out[3:]

consumption_income_debt_figure(bsim0, csim0, ysim0)

plt.show()
```




```
consumption_debt_fanchart(csim0, cons_mean0, cons_var0,
                          bsim0, debt_mean0, debt_var0)

plt.show()
```



Here is what is going on in the above graphs.

For our simulation, we have set initial conditions $b_0 = y_{-1} = y_{-2} = 0$.

Because $y_{-1} = y_{-2} = 0$, nonfinancial income y_t starts far below its stationary mean $\mu_{y,\infty}$ and rises early in each simulation.

Recall from the *previous lecture* that we can represent the optimal decision rule for consumption in terms of the **cointegrating relationship**

$$(1-\beta)b_t + c_t = (1-\beta)E_t \sum_{j=0}^{\infty} \beta^j y_{t+j} \tag{76.6}$$

So at time 0 we have

$$c_0 = (1-\beta)E_0 \sum_{t=0}^{\infty} \beta^j y_t$$

This tells us that consumption starts at the income that would be paid by an annuity whose value equals the expected discounted value of nonfinancial income at time $t = 0$.

To support that level of consumption, the consumer borrows a lot early and consequently builds up substantial debt.

In fact, he or she incurs so much debt that eventually, in the stochastic steady state, he consumes less each period than his nonfinancial income.

He uses the gap between consumption and nonfinancial income mostly to service the interest payments due on his debt.

Thus, when we look at the panel of debt in the accompanying graph, we see that this is a group of *ex-ante* identical people each of whom starts with zero debt.

All of them accumulate debt in anticipation of rising nonfinancial income.

They expect their nonfinancial income to rise toward the invariant distribution of income, a consequence of our having started them at $y_{-1} = y_{-2} = 0$.

Cointegration Residual

The following figure plots realizations of the left side of (76.6), which, *as discussed in our last lecture*, is called the **cointegrating residual**.

As mentioned above, the right side can be thought of as an annuity payment on the expected present value of future income $E_t \sum_{j=0}^{\infty} \beta^j y_{t+j}$.

Early along a realization, c_t is approximately constant while $(1-\beta)b_t$ and $(1-\beta)E_t \sum_{j=0}^{\infty} \beta^j y_{t+j}$ both rise markedly as the household's present value of income and borrowing rise pretty much together.

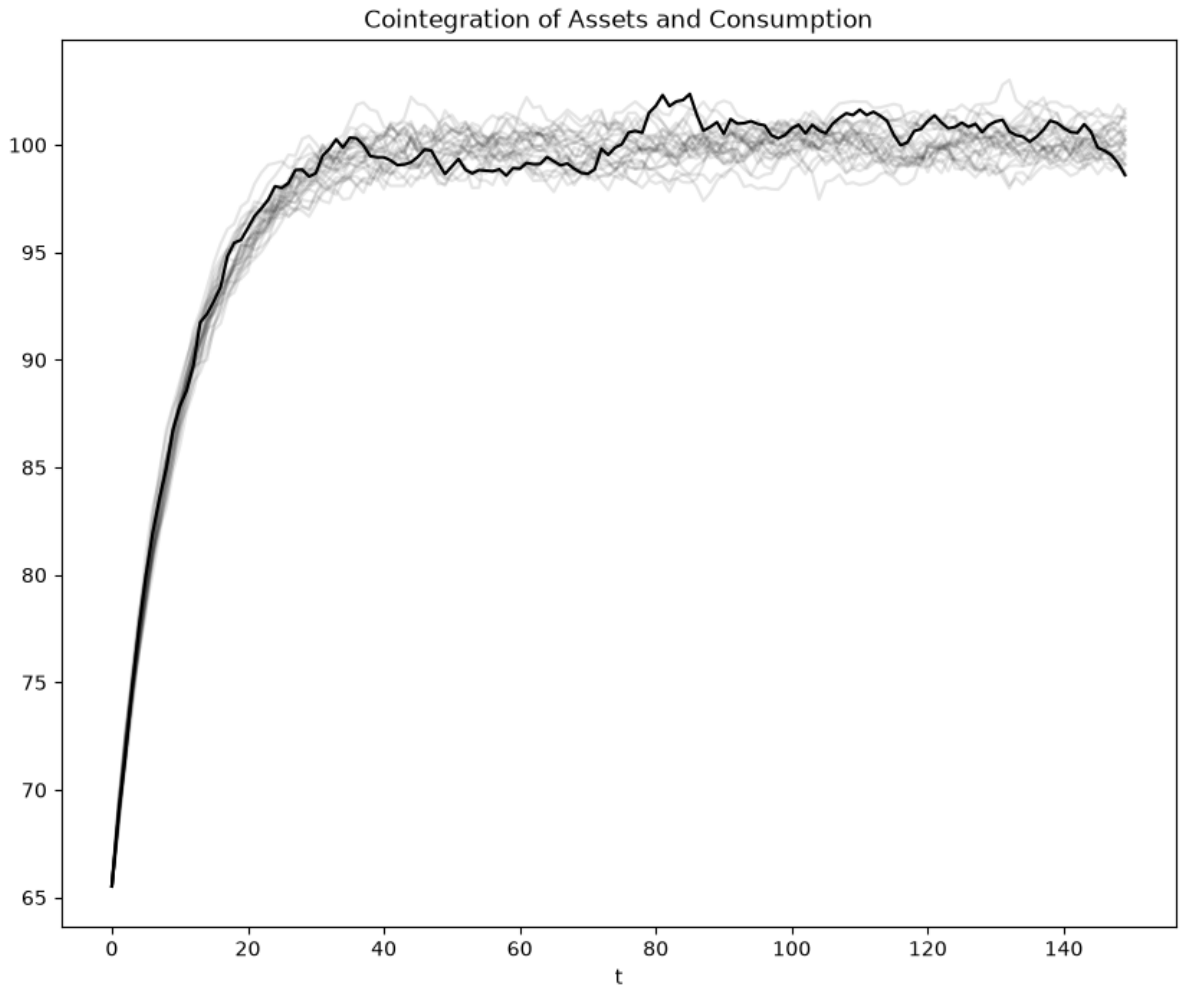
This example illustrates the following point: the definition of cointegration implies that the cointegrating residual is *asymptotically* covariance stationary, not *covariance stationary*.

The cointegrating residual for the specification with zero income and zero debt initially has a notable transient component that dominates its behavior early in the sample.

By altering initial conditions, we shall remove this transient in our second example to be presented below

```
def cointegration_figure(bsim, csim):
    """
    Plots the cointegration
    """
    # Create figure
    fig, ax = plt.subplots(figsize=(10, 8))
    ax.plot((1 - β) * bsim[0, :] + csim[0, :], color="k")
    ax.plot((1 - β) * bsim.T + csim.T, color="k", alpha=.1)

    ax.set(title="Cointegration of Assets and Consumption", xlabel="t")

    return fig
```

```
cointegration_figure(bsim0, csim0)
plt.show()
```



76.5.3 A "Borrowers and Lenders" Closed Economy

When we set $y_{-1} = y_{-2} = 0$ and $b_0 = 0$ in the preceding exercise, we make debt "head north" early in the sample.

Average debt in the cross-section rises and approaches the asymptote.

We can regard these as outcomes of a "small open economy" that borrows from abroad at the fixed gross interest rate $R = r + 1$ in anticipation of rising incomes.

So with the economic primitives set as above, the economy converges to a steady state in which there is an excess aggregate supply of risk-free loans at a gross interest rate of R.

This excess supply is filled by "foreigner lenders" willing to make those loans.

We can use virtually the same code to rig a "poor man's Bewley [Bewley, 1986] model" in the following way

- as before, we start everyone at $b_0 = 0$.
- But instead of starting everyone at $y_{-1} = y_{-2} = 0$, we draw $\begin{bmatrix} y_{-1} \\ y_{-2} \end{bmatrix}$ from the invariant distribution of the $\{y_t\}$ process.

This rigs a closed economy in which people are borrowing and lending with each other at a gross risk-free interest rate of $R = \beta^{-1}$.

Across the group of people being analyzed, risk-free loans are in zero excess supply.

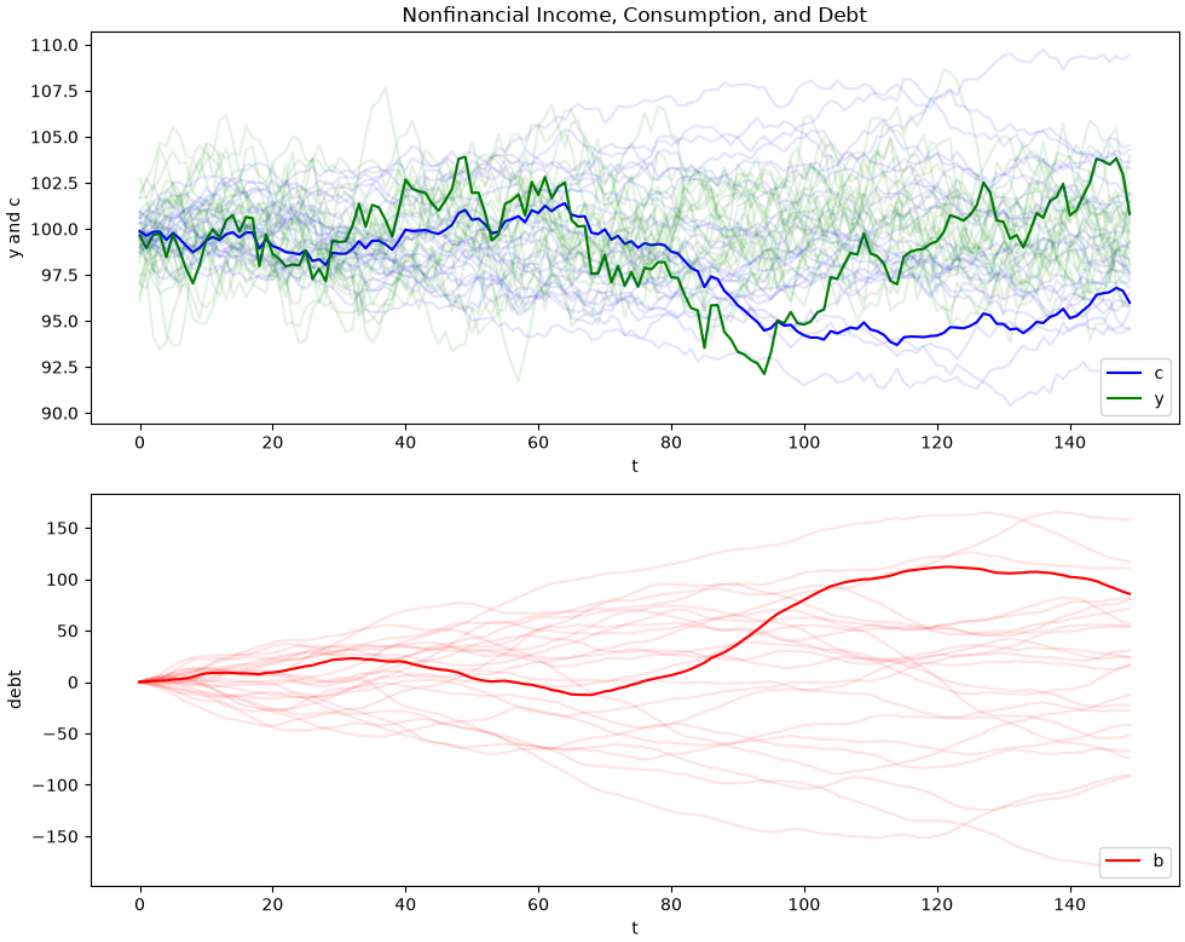
We have arranged primitives so that $R = \beta^{-1}$ clears the market for risk-free loans at zero aggregate excess supply.

So the risk-free loans are being made from one person to another within our closed set of agents.

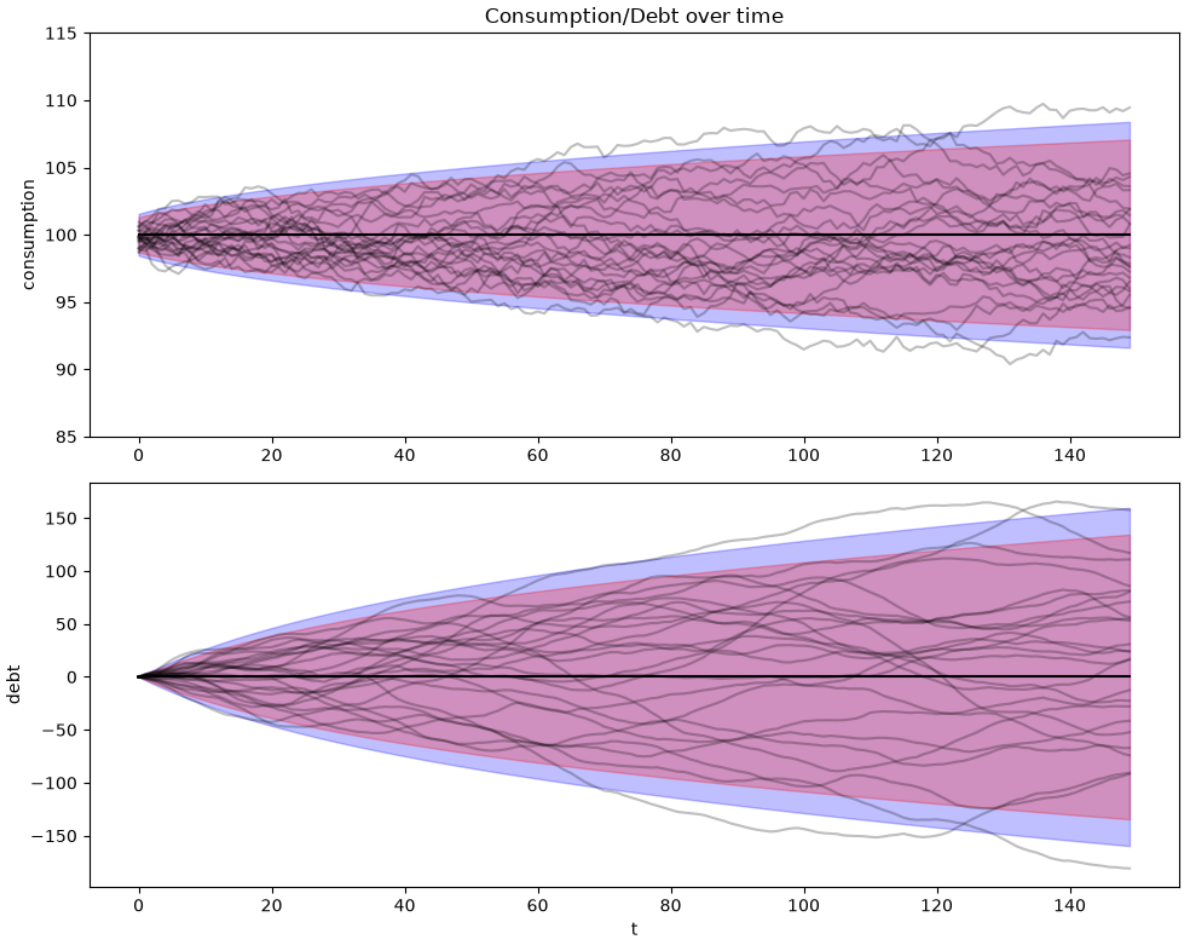
There is no need for foreigners to lend to our group.

Let's have a look at the corresponding figures

```
out = income_consumption_debt_series(A_LSS, C_LSS, G_LSS, mxbewley, sxbewley)
bsimb, csimb, ysimb = out[:3]
cons_meanb, cons_varb, debt_meanb, debt_varb = out[3:]

consumption_income_debt_figure(bsimb, csimb, ysimb)

plt.show()
```




```
consumption_debt_fanchart(csimb, cons_meanb, cons_varb,
                          bsimb, debt_meanb, debt_varb)

plt.show()
```



The graphs confirm the following outcomes:

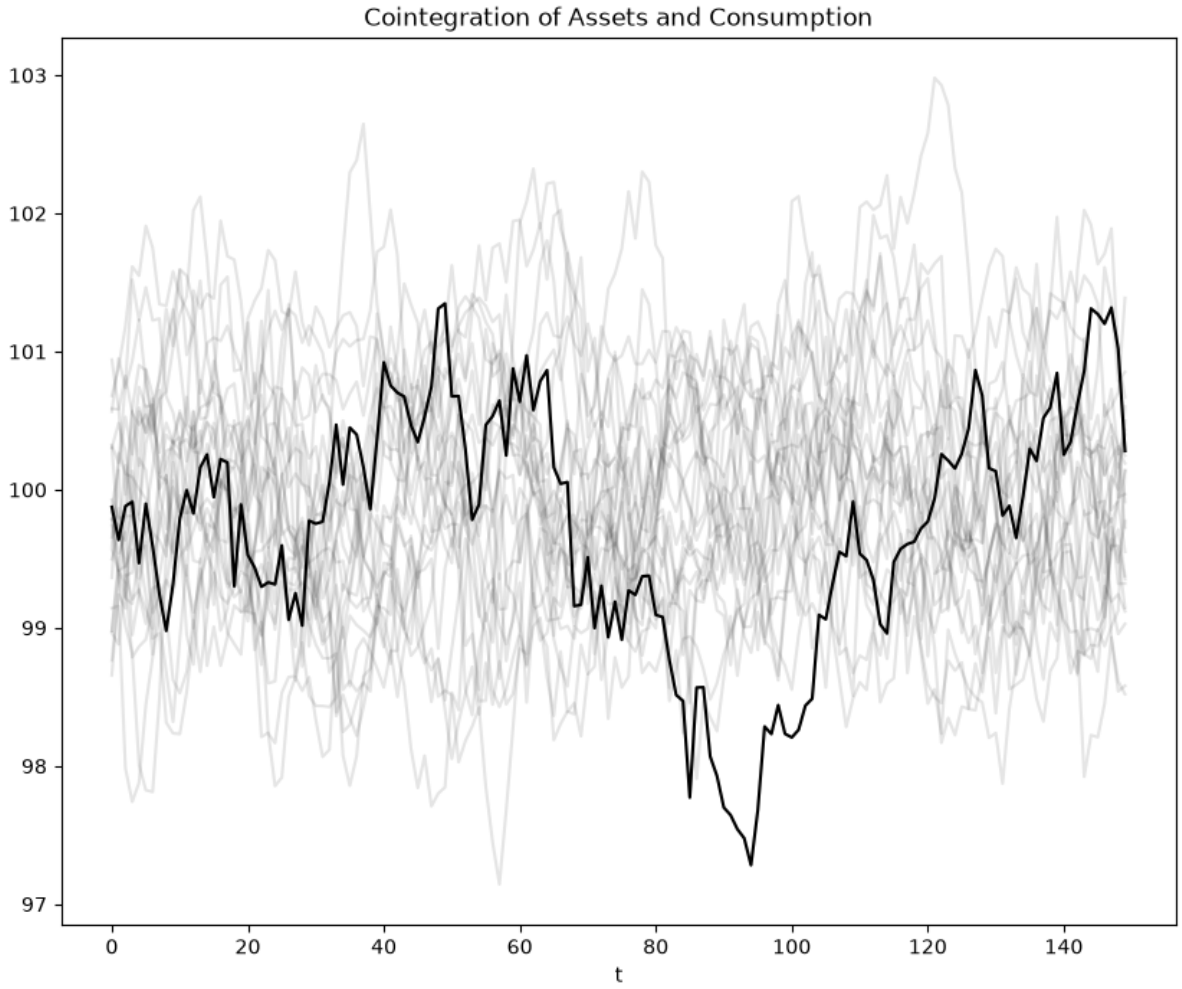
- As before, the consumption distribution spreads out over time.

But now there is some initial dispersion because there is *ex-ante* heterogeneity in the initial draws of $\begin{bmatrix} y_{-1} \\ y_{-2} \end{bmatrix}$.

- As before, the cross-section distribution of debt spreads out over time.
- Unlike before, the average level of debt stays at zero, confirming that this is a closed borrower-and-lender economy.
- Now the cointegrating residual seems stationary, and not just asymptotically stationary.

Let's have a look at the cointegration figure

```
cointegration_figure(bsimb, csimb)
plt.show()
```


Cointegration of Assets and Consumption
103
102
101
100
99
98
97
0
20
40
60
80
100
120
140
t

CHAPTER

SEVENTYSEVEN

ROBUST PERMANENT INCOME AND PRICING

Contents

- *Robust Permanent Income and Pricing*
  - *Overview*
  - *Risk-sensitive recursive preferences*
  - *A preference for robustness*
  - *The permanent income economy*
  - *Observational equivalence*
  - *Estimation*
  - *Asset pricing and the market price of risk*
  - *A risk-sensitive regulator*
  - *Exercises*
  - *Related lectures*

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install --upgrade quantecon
```

77.1 Overview

This lecture studies the model of Hansen *et al.* [1999].

The paper asks a simple question with surprising consequences.

What happens to a classic permanent income consumer when, instead of trusting a single probability model of their income, they *fear that their model is misspecified* and want decisions that work well across a *family* of nearby models?

Such a consumer is called a **robust** decision maker.

The central findings are:

- A preference for robustness is hidden inside the quantity implications of the ordinary permanent income model.
- Robustness and risk-sensitivity are two interpretations of the *same* decision rules, with a single parameter σ governing both.

- Concern about *small* amounts of model misspecification can show up as *large* market-based measures of risk aversion.
- The consumption and savings data alone cannot identify the robustness parameter: the model is **observationally equivalent** to a standard permanent income model with a lower discount factor.
- But asset prices, in particular the **market price of risk**, *can* be used to pin robustness down.

We will learn about

- risk-sensitive recursive preferences and the operator $\mathcal{R}_t$
- how a malevolent "second agent" implements a preference for robustness through a two-player zero-sum game
- the link between robustness and **Knightian uncertainty** in the sense of Gilboa and Schmeidler [1989] and Epstein and Wang [1994]
- an **observational equivalence** result that we will reproduce numerically
- how a small worst-case distortion of conditional means translates almost one-for-one into a market price of risk

This lecture builds on ideas in *The Permanent Income Model*, *Permanent Income II: LQ Techniques*, and *LQ Control: Foundations*.

The robustness machinery here is developed at book length in Hansen and Sargent [2008], extended in Anderson *et al.* [2003], and reinterpreted through detection-error probabilities in Barillas *et al.* [2009].

Let's start with some imports.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import brentq
from scipy.stats import norm
import quantecon as qe
```

77.2 Risk-sensitive recursive preferences

The theory rests on a recursive linear-quadratic optimization problem with a twist in how continuation utility is aggregated.

The state evolves as

$$x_{t+1} = Ax_t + Bi_t + Cw_{t+1}, \tag{77.1}$$

where i_t is a control vector, x_t is the state vector, and w_{t+1} is an IID Gaussian vector with $\mathbb{E} w_{t+1} = 0$ and $\mathbb{E} w_{t+1} w_{t+1}^\top = I$.

The one-period return is

$$u(i_t, x_t) = -i_t^\top Q i_t - x_t^\top R x_t, \tag{77.2}$$

with Q positive definite and R positive semidefinite.

Following Epstein and Zin [1989], Weil [1989], and Hansen and Sargent [1995], intertemporal preferences are induced by the recursion

$$U_t = u(i_t, x_t) + \beta \, \mathcal{R}_t(U_{t+1}), \tag{77.3}$$

where the **risk-sensitivity operator** is

$$\mathcal{R}_t(U_{t+1}) \equiv \frac{2}{\sigma} \log \mathbb{E}\left[\exp\left(\frac{\sigma U_{t+1}}{2} \right) \Big| J_t \right]. \tag{77.4}$$

Here J_t is the information available at t.

When $\sigma = 0$ we set $\mathcal{R}_t \equiv \mathbb{E}(\cdot \mid J_t)$ and recover the usual von Neumann–Morgenstern, state-additive specification.

When $\sigma \neq 0$ the operator $\mathcal{R}_t$ applies an additional risk adjustment *over and above* the one coming from the curvature of u.

Values of $\sigma < 0$ correspond to *more* aversion to risk than the von Neumann–Morgenstern benchmark, which is the case studied throughout the paper.

Note

The exponential-of-utility form in (77.4) originates in the *risk-sensitive control* literature started by Jacobson [1973] and extended by Whittle [1981] and Whittle [1990].

Hansen *et al.* [1999] give it an economic reinterpretation as a *preference for robustness*.

77.2.1 The operator under Gaussian uncertainty

The operator $\mathcal{R}_t$ has a transparent closed form when continuation utility is Gaussian.

Suppose $U_{t+1} \sim N(\mu, s^2)$ conditional on J_t. Using the Gaussian moment generating function $\mathbb{E}[\exp(aU_{t+1})] = \exp(a\mu + \frac{1}{2}a^2 s^2)$ with $a = \sigma/2$,

$$\mathcal{R}_t(U_{t+1}) = \frac{2}{\sigma} \log \mathbb{E}\left[\exp\left(\frac{\sigma U_{t+1}}{2}\right) \middle| J_t\right] = \frac{2}{\sigma}\left(\frac{\sigma}{2}\mu + \frac{\sigma^2}{8}s^2\right) = \mu + \frac{\sigma}{4}s^2. \tag{77.5}$$

For $\sigma < 0$ this is *below* the conditional mean μ: the decision maker evaluates uncertain prospects *pessimistically*, and the penalty grows with the conditional variance s^2.

This certainty equivalent has a revealing decomposition.

The expectation in (77.4) re-weights outcomes by $\exp(\sigma U_{t+1}/2)$; for a Gaussian this **exponential tilting** produces a new normal density with the *same* variance s^2 but a mean shifted to $\mu + \frac{\sigma}{2}s^2$.

The operator value $\mu + \frac{\sigma}{4}s^2$ lies *halfway* between the original mean μ and this worst-case mean: it equals the worst-case expected utility $\mu + \frac{\sigma}{2}s^2$ *plus* the relative-entropy penalty $-\frac{\sigma}{4}s^2$ that restrains the distortion.

Note

The two coefficients describe different objects.

The **worst-case mean** of U_{t+1} shifts by $\frac{\sigma}{2}s^2$, while the **operator value** (certainty equivalent) shifts by $\frac{\sigma}{4}s^2$.

Both are correct; the smaller shift of $\mathcal{R}_t$ reflects the entropy cost the malevolent player pays for the distortion.

A self-contained derivation of (77.5) is requested in Exercise 77.9.1.

Let's visualize both facts, with the certainty equivalent on the left and the worst-case (tilted) density of continuation utility on the right.

```
μ, s = 0.0, 1.0           # conditional mean and std of continuation utility

def R_operator(μ, s, σ):
    "Risk-sensitive operator for a Gaussian U ~ N(μ, s^2)."
    if σ == 0:
        return μ
```

(continues on next page)

(continued from previous page)

```
    return μ + σ * s**2 / 4

σ_vals = np.linspace(-1.5, 0.0, 200)
R_vals = [R_operator(μ, s, sg) for sg in σ_vals]

fig, axes = plt.subplots(1, 2, figsize=(11, 4))

axes[0].plot(σ_vals, R_vals, lw=2)
axes[0].axhline(μ, color='k', ls='--', lw=1, label=r'$E[U_{t+1}]=\mu$')
axes[0].set_xlabel(r'risk-sensitivity $\sigma$')
axes[0].set_ylabel(r'$\mathcal{R}_t(U_{t+1})$')
axes[0].set_title('certainty equivalent')
axes[0].legend()

# Tilted density
grid = np.linspace(μ - 4*s, μ + 4*s, 400)
axes[1].plot(grid, norm.pdf(grid, μ, s), lw=2,
             label=r'reference $N(\mu, s^2)$')
for σ in [-0.5, -1.0]:
    shift = σ * s**2 / 2      # worst-case mean shift of U_{t+1}
    axes[1].plot(grid, norm.pdf(grid, μ + shift, s), lw=2, ls='--',
                 label=f'worst-case, $\\sigma={σ}$')
    axes[1].axvline(μ + σ * s**2 / 4, color='C3', lw=0.8, ls=':')
axes[1].set_xlabel(r'continuation utility $U_{t+1}$')
axes[1].set_ylabel('density')
axes[1].set_title('worst-case density')
axes[1].legend()
fig.tight_layout()
plt.show()
```

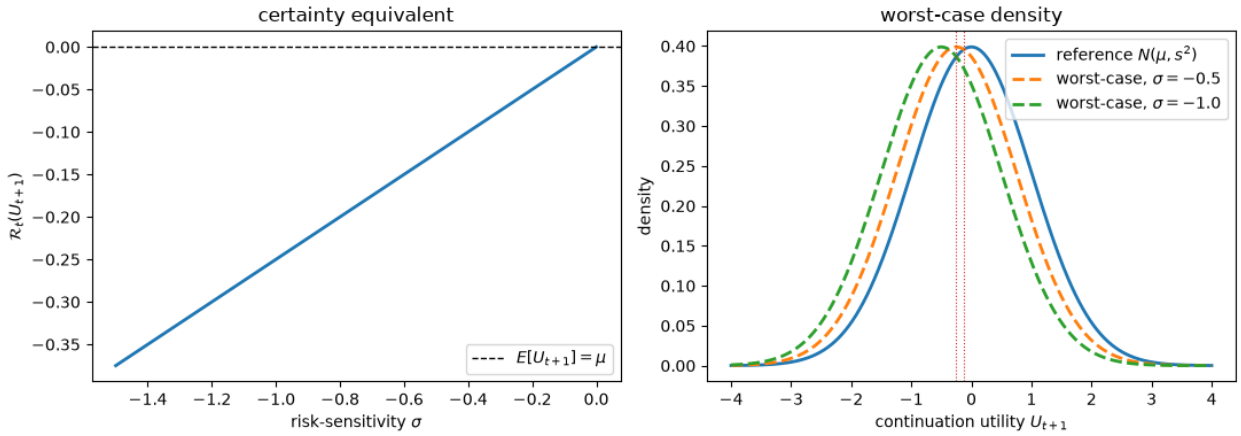


Fig. 77.1: Risk-sensitive operator and worst-case density

The left panel shows the certainty equivalent $\mathcal{R}_t$ sliding below the mean as σ becomes more negative.

The right panel shows the associated worst-case density of continuation utility: a robust agent behaves *as if* U_{t+1} were drawn from a pessimistically re-centered distribution (mean $\mu + \frac{\sigma}{2}s^2$, dashed), while the operator value $\mathcal{R}_t = \mu + \frac{\sigma}{4}s^2$ (dotted) sits halfway between it and the reference mean μ.

77.3 A preference for robustness

The pessimistic tilt in the right panel above is not just an analogy.

Hansen *et al.* [1999] show that the risk-sensitive problem is the value function of a **two-player zero-sum game**.

In this game, one player chooses the control $\{i_t\}$ while a second, malevolent player chooses a distortion $\{v_t\}$ to the conditional mean of the shocks.

The distorted law of motion is

$$x_{t+1} = Ax_t + Bi_t + C(w_{t+1} + v_t). \tag{77.6}$$

The minimizing player would like to push the state in painful directions, but is restrained by a penalty on the size of the distortion.

With $-1/\sigma \geq 0$ acting as a Lagrange multiplier on a constraint that bounds the distortion sequence, the Markov perfect equilibrium has value function

$$\tilde{W}(x) = \inf_v \sup_i \left\{ -i^\top Q i - x^\top R x + \beta \left[-\frac{1}{\sigma} v^\top v + \mathbb{E}\, \tilde{W}(Ax + Bi + C(w + v)) \right] \right\}. \tag{77.7}$$

Because $\sigma < 0$ makes $-1/\sigma > 0$, the term $-\frac{1}{\sigma} v^\top v$ *penalizes* the malevolent player for large distortions.

A smaller $|1/\sigma|$ (more negative σ) means a cheaper distortion budget and hence a larger family of models the agent guards against, reflecting a *stronger* preference for robustness.

This is exactly the max-min expected utility structure of Gilboa and Schmeidler [1989]: the agent's "nominal model" sets $v_t = 0$, but they entertain a whole family of alternatives indexed by $\{v_t\}$ and act against the worst case.

Following Epstein and Wang [1994], the non-uniqueness of the implied probability measures is a form of **Knightian uncertainty**.

The robust and risk-sensitive problems share the same value function matrix Ω and the same decision rule $i_t = -Fx_t$; they differ only in interpretation.

77.4 The permanent income economy

We now specialize to a habit-persistence version of the permanent income model.

A planner orders consumption streams $\{c_t\}$ through a service stream $\{s_t\}$ using the recursion

$$U_t = -(s_t - b_t)^2 + \beta\, \mathcal{R}_t(U_{t+1}), \tag{77.8}$$

where $\{b_t\}$ is an exogenous preference (bliss-point) shock.

Services are produced from consumption via the household technology

$$s_t = (1 + \lambda)c_t - \lambda h_{t-1}, \tag{77.9}$$

$$h_t = \delta_h h_{t-1} + (1 - \delta_h)c_t, \tag{77.10}$$

with $\lambda > 0$ and $\delta_h \in (0, 1)$.

Here h_t is a geometric average of current and past consumption, so (77.9) makes services depend *negatively* on a weighted average of past consumption, which is the **habit persistence**.

There is a linear production technology

$$c_t + i_t = \gamma k_{t-1} + d_t,$$

and capital accumulates as $k_t = \delta_k k_{t-1} + i_t$, where $\{d_t\}$ is an exogenous endowment.

Combining,

$$c_t + k_t = (\delta_k + \gamma) k_{t-1} + d_t, \qquad R \equiv \delta_k + \gamma, \tag{77.11}$$

so R is the gross physical return on capital, which in a decentralized economy equals the gross **risk-free rate**.

The endowment and preference shocks are driven by a common linear state,

$$z_{t+1} = A_{22} z_t + C_2 w_{t+1}, \qquad d_t = U_d z_t, \quad b_t = U_b z_t. \tag{77.12}$$

This whole economy is a special case of the control problem (77.1)–(77.3): stack h_{t-1}, k_{t-1}, and z_t into the state x_t and let the control be $i_t = s_t - b_t$.

77.4.1 The $\sigma = 0$ benchmark and the martingale

To build intuition, set $\sigma = 0$ and impose the permanent income restriction $\beta R = 1$, as in Hall [1978].

The first-order conditions then imply that the marginal utility of consumption services is a **martingale**,

$$\mathbb{E}_t \, \mu_{c,t+1} = \mu_{c,t}, \tag{77.13}$$

and that $\mu_{s,t}$ inherits the representation

$$\mu_{s,t} = \mu_{s,t-1} + v^\top w_t \tag{77.14}$$

for some loading vector v.

Equation (77.13) is the classic statement that, under $\beta R = 1$, consumption responds only to *news*, so it is a random walk.

This is the result that Hall [1978] and Campbell [1987] tested on aggregate U.S. data.

The scalar

$$\theta^2 \equiv v^\top v$$

measures the variance of the innovation to the marginal-utility martingale (77.14).

It will be the one summary statistic of the benchmark economy that we need below.

> **Note**
>
> Under a rational-expectations reading, the benchmark $\sigma = 0$ permanent income model has *no precautionary savings*, as emphasized by Zeldes.
>
> Introducing robustness ($\sigma < 0$) revives a precautionary motive: the consumer guards against worst-case mistakes in the conditional means of shocks.

77.5 Observational equivalence

Here is the paper's first headline result.

Proposition 77.5.1 (Observational Equivalence)

Fix all parameters except β and σ. Suppose $\beta R = 1$. There exists $\underline{\sigma} < 0$ such that the optimal consumption–investment plan for $\sigma = 0$ is *also* optimal for any $\sigma \in (\underline{\sigma}, 0)$, provided the discount factor is lowered to a value $\hat{\beta}(\sigma)$ that varies directly with σ.

In words: as far as the *quantities* $\{c_t, k_t\}$ are concerned, the robust ($\sigma < 0$) permanent income model is indistinguishable from the standard ($\sigma = 0$) one with a smaller discount factor.

Increasing the preference for robustness stimulates a precautionary motive for saving; lowering β makes saving less attractive; along a particular locus the two effects exactly cancel.

The proof is constructive and delivers an explicit locus of observationally equivalent $(\sigma, \hat{\beta})$ pairs. Define

$$\Omega(\beta) = \frac{\beta - 1 + \sigma\theta^2 + \sqrt{(\beta - 1 + \sigma\theta^2)^2 + 4\sigma\theta^2}}{-2\sigma\theta^2}, \tag{77.15}$$

$$\hat{\zeta}(\beta) = 1 + \frac{\theta^2 \sigma\, \Omega(\beta)}{1 - \sigma\theta^2 \Omega(\beta)}. \tag{77.16}$$

The equivalent discount factor $\hat{\beta}$ solves

$$\hat{\beta}\, R\, \hat{\zeta}(\hat{\beta}) = 1. \tag{77.17}$$

The lower bound $\underline{\sigma}$ is the most negative σ for which the square root in (77.15) stays real.

Let's reproduce the locus, a version of Figure 1 in Hansen *et al.* [1999].

```
β_bench = 0.9971          # benchmark discount factor (annual rate ~2.5%)
Rf = 1 / β_bench          # gross risk-free return fixed by β R = 1
θ2 = 0.01                # variance of the marginal-utility innovation, v^T v

def Ω_scalar(β, σ, θ2):
    disc = (β - 1 + σ * θ2)**2 + 4 * σ * θ2
    if disc < 0:
        return np.nan        # below σ-underbar: no real solution
    return (β - 1 + σ * θ2 + np.sqrt(disc)) / (-2 * σ * θ2)

def ζ_hat(β, σ, θ2):
    Ω = Ω_scalar(β, σ, θ2)
    return 1 + (θ2 * σ * Ω) / (1 - σ * θ2 * Ω)

def β_hat(σ, θ2, Rf):
    "Discount factor that makes σ observationally equivalent to σ=0."
    if σ == 0:
        return 1 / Rf
    f = lambda b: b * Rf * ζ_hat(b, σ, θ2) - 1
    return brentq(f, 0.95, 1 / Rf - 1e-12)

σ_vals = np.linspace(-1.2e-4, 0.0, 200)
β_vals = np.array([β_hat(sg, θ2, Rf) for sg in σ_vals])

fig, ax = plt.subplots()
ax.plot(β_vals, σ_vals, lw=2)
ax.set_xlabel(r'discount factor $\hat\beta$')
ax.set_ylabel(r'risk-sensitivity $\sigma$')
ax.axhline(0, color='k', lw=0.8, ls='--')
fig.tight_layout()
plt.show()
```

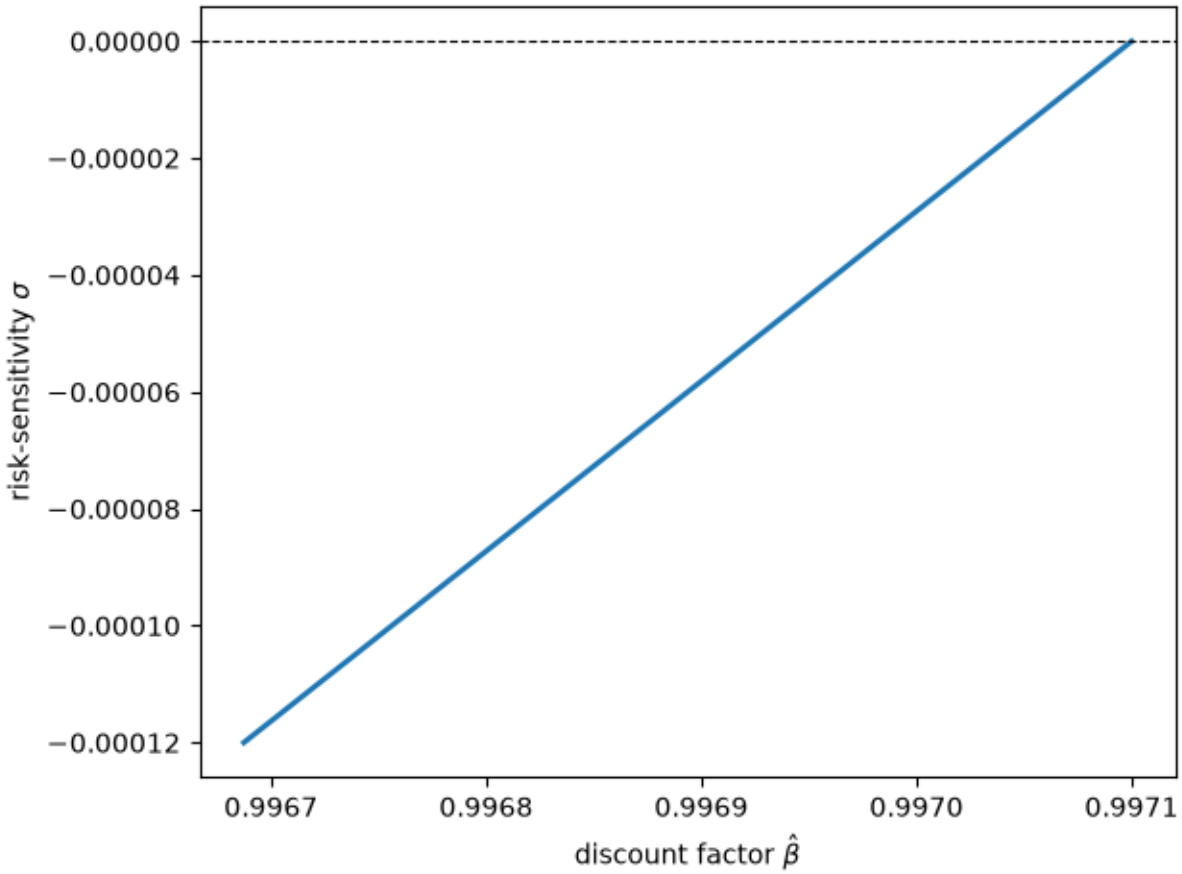


Fig. 77.2: Equivalent $(\sigma, \hat{\beta})$ pairs

Every point on this curve generates *exactly the same* consumption and investment data.

Moving down the curve (more negative σ, i.e. a stronger preference for robustness) requires a *lower* discount factor $\hat{\beta}$ to keep quantities unchanged.

This is why consumption and savings data alone cannot tell us how much the consumer fears model misspecification.

77.6 Estimation

Hansen *et al.* [1999] turns the observational-equivalence result into an empirical strategy.

Because the quantity data cannot pin down σ, the authors first estimate the $\sigma = 0$ version of the model, conditioning the likelihood *only on consumption and investment*, and then use the locus of *Proposition 77.5.1* to trace out the family of $(\sigma, \hat{\beta})$ pairs consistent with those estimates.

Asset prices (the next section) break the tie.

77.6.1 The data and the likelihood

The model is fit to U.S. post-war quarterly data, 1970:I–1996:III.

- *Consumption* is measured as nondurables plus services.
- *Investment* is measured as durables plus gross private investment.

Both series are deflated by the deterministic growth factor 1.0033^t, so the model is fit to *detrended* data.

The likelihood is Gaussian, built recursively (a Kalman filter), with the unobserved part of the initial state estimated using the methods of Hansen and Sargent.

77.6.2 Specification

The preference shock is a constant, $b_t = \mu_b$, fixed at $\mu_b = 32$; recall from the discussion of (77.11) that the *level* of b_t does not affect the decision rules, only prices.

The endowment is the sum of a *persistent* and a *transitory* component, each a second-order autoregression driven by orthogonal shocks,

$$(1 - \phi_1 L)(1 - \phi_2 L)\, d_t^* = c_{d^*}\, w_t^{d^*}, \tag{77.18}$$

$$(1 - \alpha_1 L)(1 - \alpha_2 L)\, \hat{d}_t = c_{\hat{d}}\, w_t^{\hat{d}}, \tag{77.19}$$

with $d_t = \mu_d + d_t^* + \hat{d}_t$.

A likelihood comparison (a gain from AR(1) to AR(2) but not beyond) led the authors to a second-order specification for the transitory component.

The four parameters governing the endogenous dynamics are $(\gamma, \delta_k, \beta, \lambda)$.

The depreciation factor is set to $\delta_k = 0.975$, and the permanent-income restriction $\beta R = 1$ (confirmed by the unrestricted estimates) is imposed with $\beta = 0.9971$, implying a 2.5% annual real interest rate after the growth adjustment.

The maximum-likelihood estimates (with habit persistence) are reproduced below, a version of Table 2 in Hansen *et al.* [1999].

Parameter	Symbol	Estimate
Discount factor	β	0.997
Habit depreciation	δ_h	0.682
Habit weight	λ	2.443
Transitory AR roots	α_1, α_2	0.813, 0.189
Persistent AR roots	ϕ_1, ϕ_2	0.998, 0.704
Endowment mean	μ_d	13.710
Transitory shock scale	$c_{\hat{d}}$	0.155
Persistent shock scale	c_{d^*}	0.108

The single most striking estimate is the autoregressive root $\phi_1 = 0.998$ of the persistent endowment component, a number all but indistinguishable from a unit root.

77.6.3 Impulse responses and the permanent income mechanism

The persistence of a shock is what determines how strongly consumption reacts to it.

Under the permanent income logic (with $\beta R = 1$ and, for transparency, *no* habit), consumption jumps on impact by the **annuity value** of the change in wealth and is a martingale thereafter,

$$\Delta c = \left(1 - \frac{1}{R}\right) \sum_{j \geq 0} R^{-j} \, \psi_j, \tag{77.20}$$

where $\psi_j = \partial d_{t+j} / \partial w_t$ is the endowment's own impulse response.

A near-permanent shock has a large present value and moves consumption a lot; a transitory shock has a small present value and barely moves it.

Let's compute the endowment responses (77.18)–(77.19) and the implied consumption responses.

```
def ar2_irf(r1, r2, c, H):
    "Impulse response of (1 - r1 L)(1 - r2 L) x = c w to a unit w shock."
    ψ = np.zeros(H)
    ψ[0] = c
    if H > 1:
        ψ[1] = (r1 + r2) * ψ[0]
    for j in range(2, H):
        ψ[j] = (r1 + r2) * ψ[j-1] - r1 * r2 * ψ[j-2]
    return ψ

H = 50
disc = Rf**(-np.arange(H))    # Rf = 1/β from the previous cell

ψ_p = ar2_irf(0.998, 0.704, 0.108, H)          # persistent endowment d*
ψ_t = ar2_irf(0.813, 0.189, 0.155, H)          # transitory endowment d_hat

# Flat consumption response
dc_p = (1 - 1/Rf) * np.sum(disc * ψ_p)
dc_t = (1 - 1/Rf) * np.sum(disc * ψ_t)

print(f"persistent shock: consumption responds to "
      f"{100*dc_p/ψ_p[0]:.0f}% of the impact")
print(f"transitory shock: consumption responds to "
      f"{100*dc_t/ψ_t[0]:.0f}% of the impact")
```

(continues on next page)

(continued from previous page)

```
fig, axes = plt.subplots(1, 2, figsize=(11, 4), sharey=True)
for ax, ψ, dc, shock_label in [
        (axes[0], ψ_p, dc_p, 'persistent shock $d^{*}$'),
        (axes[1], ψ_t, dc_t, 'transitory shock $\\hat d$')]:
    ax.plot(ψ, lw=2, label=fr'{shock_label}: endowment response $\psi_j$')
    ax.axhline(dc, color='C3', ls='--', lw=2,
               label=r'consumption response $\Delta c$')
    ax.set_title(shock_label)
    ax.set_xlabel('horizon $j$ (quarters)')
    ax.legend()
axes[0].set_ylabel('response')
fig.tight_layout()
plt.show()
```

```
persistent shock: consumption responds to 41% of the impact
transitory shock: consumption responds to 2% of the impact
```

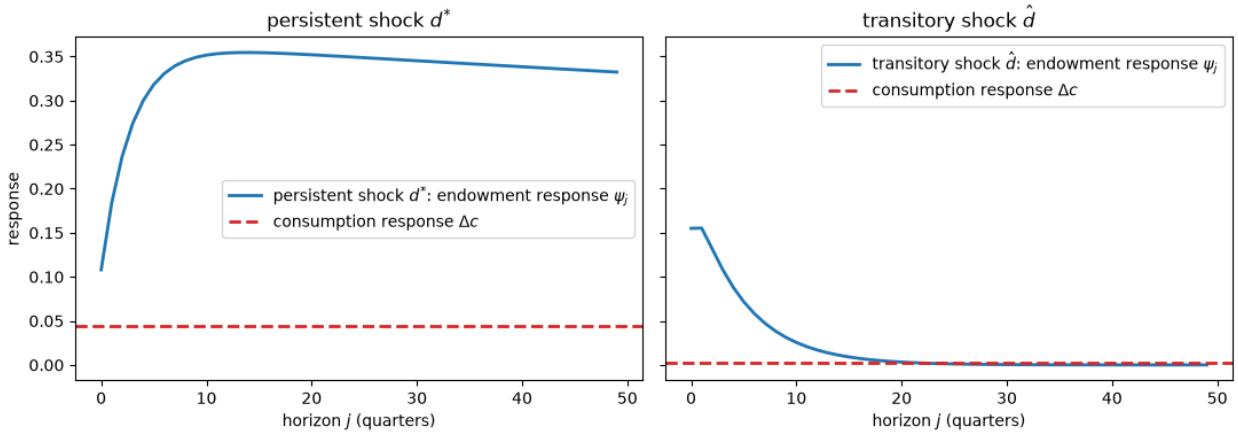


Fig. 77.3: Endowment and consumption impulse responses

The contrast is the heart of the permanent income hypothesis tested by Hall [1978] and Campbell [1987].

Consumption tracks a large fraction of the persistent shock, whose near-unit root makes it almost permanent income, but only a sliver of the transitory shock, the bulk of which is saved and shows up as investment.

Note

With habit persistence ($\lambda > 0$) the consumption responses are no longer flat: they become hump-shaped, because services rather than consumption obey the martingale logic.

The estimated $\lambda = 2.443$ and $\delta_h = 0.682$ imply economically important habit effects, and a likelihood-ratio comparison strongly rejects $\lambda = 0$.

Hansen *et al.* [1999] compare these magnitudes with the habit estimates in the time-nonseparable preference literature.

77.7 Asset pricing and the market price of risk

Hansen *et al.* [1999] shows how the observationally-equivalent pairs that look identical in quantity data have *different* implications for asset prices.

77.7.1 Decentralization

Following Lucas [1978], we regard the robust (or risk-sensitive) planning solution as the allocation of a competitive economy populated by a large number of identical agents who trade securities.

Equilibrium prices are the **shadow prices** that leave each agent content to consume the planner's allocation, treating it as an endowment process.

The equilibrium law of motion for the state is

$$x_{t+1} = A^0 x_t + C w_{t+1}, \tag{77.21}$$

and the value function at the optimum is $U_t^e = x_t^\top \Omega x_t + \rho$.

To support the robust allocation, prices must be computed using the *same* pessimistic, distorted beliefs that rationalize the planner's choices.

This is where risk-sensitivity ($\sigma < 0$) leaves its fingerprint on prices even though, by observational equivalence, it leaves no trace in quantities.

77.7.2 The twisting operator and distorted beliefs

Pricing a claim to next period's *utility* is trivial under the von Neumann–Morgenstern specification but nontrivial under risk-sensitivity.

The key object is the **twisting operator**

$$\mathcal{T}_t U_{t+1} \equiv \frac{\mathbb{E}(V_{t+1} U_{t+1} \mid J_t)}{\mathbb{E}(V_{t+1} \mid J_t)}, \qquad V_{t+1} \equiv \exp\left(\frac{\sigma U_{t+1}^e}{2}\right), \tag{77.22}$$

which re-weights outcomes by the exponential of equilibrium continuation utility.

It satisfies the subgradient inequality

$$\mathcal{R}_t(U_{t+1}) - \mathcal{R}_t(U_{t+1}^e) \leq \mathcal{T}_t U_{t+1} - \mathcal{T}_t U_{t+1}^e, \tag{77.23}$$

so $\mathcal{T}_t$ behaves like a *distorted* conditional expectation, exactly the change of measure used to price derivative claims in Epstein and Wang [1994].

Concretely, $\mathcal{T}_t$ is the ordinary conditional expectation under a **distorted transition law**

$$x_{t+1} = \hat{A} x_t + \hat{C} w_{t+1}, \qquad \hat{C}\hat{C}^\top = C(I - \sigma C^\top \Omega C)^{-1} C^\top, \tag{77.24}$$

with $\hat{A}$ given by (77.30)-style risk corrections.

Because $\sigma < 0$ and Ω is negative semidefinite, $(I - \sigma C^\top \Omega C)^{-1}$ exceeds the identity: the pricing measure assigns a *pessimistically shifted conditional mean and* an *inflated conditional variance* to next period's state.

These two distortions are precisely what generate risk premia.

77.7.3 Multi-period claims and the one-period stochastic discount factor

Prices of streams are built by iterating the operator.

Define $\mathcal{S}_{t,\tau} = \mathcal{T}_t \mathcal{T}_{t+1} \cdots \mathcal{T}_{t+\tau-1}$. The time-$t$ price of a claim to the consumption stream $\{c_{t+\tau}\}$ is then a discounted sum of twisted marginal utilities, and the one-period security with payoff p_{t+1} is priced as

$$q_t = \mathcal{T}_t \left\{ \beta \frac{\mathcal{M}^c_{t+1}}{\mathcal{M}^c_t} \, p_{t+1} \right\} = \mathbb{E}\left(m_{t+1,t} \, p_{t+1} \mid J_t \right), \tag{77.25}$$

where $\mathcal{M}^c_t$ is the marginal utility of consumption and $m_{t+1,t}$ is the one-period **stochastic discount factor** (intertemporal marginal rate of substitution).

Under risk-sensitivity, $m_{t+1,t}$ factors into two pieces,

$$m_{t+1,t} = m^f_{t+1,t} \, m^r_{t+1,t}, \tag{77.26}$$

where

$$m^f_{t+1,t} = \beta \frac{\mathcal{M}^c_{t+1}}{\mathcal{M}^c_t}$$

is the *familiar* intertemporal marginal rate of substitution (the only term present when $\sigma = 0$), and

$$m^r_{t+1,t} = \frac{\exp(\sigma U^e_{t+1}/2)}{\mathbb{E}[\exp(\sigma U^e_{t+1}/2) \mid J_t]} \tag{77.27}$$

is a multiplicative adjustment with conditional mean one.

The factor (77.27) is the source of the extra risk premia.

77.7.4 The market price of risk

Under the *robustness* interpretation, the same multiplicative factor equals a **likelihood ratio** between the worst-case and reference shock densities,

$$m^u_{t+1,t} = \frac{\exp[-(w_{t+1} - \hat{v}_t)^\top (w_{t+1} - \hat{v}_t)/2]}{\exp(-w^\top_{t+1} w_{t+1}/2)}, \tag{77.28}$$

where $\hat{v}_t$ is the worst-case conditional-mean distortion chosen by the malevolent player.

A direct calculation gives

$$\mathbb{E}_t(m^u_{t+1,t}) = 1, \qquad \mathbb{E}_t[(m^u_{t+1,t})^2] = \exp(\hat{v}_t^\top \hat{v}_t),$$

so that, for small distortions,

$$\mathrm{std}_t(m^u_{t+1,t}) = \left[\exp(\hat{v}_t^\top \hat{v}_t) - 1 \right]^{1/2} \approx |\hat{v}_t|. \tag{77.29}$$

The **market price of risk** is the maximal Sharpe ratio attainable, equal to $\mathrm{std}_t(m_{t+1,t})/\mathbb{E}_t(m_{t+1,t})$ along the efficient frontier (the Hansen and Jagannathan [1991] bound).

It is therefore approximately equal to the *magnitude of the worst-case distortion* $|\hat{v}_t|$.

This is the paper's punchline: a conditional-mean misspecification of $x\%$ of a unit-norm direction raises the market price of risk by roughly $x/100$.

A *small*, statistically-hard-to-detect doubt about the model can generate the *large* price of risk seen in the data.

Let's check the key identity (77.29) by Monte Carlo.

```
rng = np.random.default_rng(12345)

def mpr_check(v_hat, n=2_000_000):
    """
    Simulate the worst-case likelihood ratio m^u and compare its
    conditional standard deviation to |v_hat|.
    """
    k = len(v_hat)
    w = rng.standard_normal((n, k))
    # log likelihood ratio of N(v_hat, I) relative to N(0, I)
    log_mu = w @ v_hat - 0.5 * v_hat @ v_hat
    mu = np.exp(log_mu)
    return mu.mean(), mu.std(), np.linalg.norm(v_hat)

print(f"{'|v_hat|':>10}{'E[m^u]':>12}{'std(m^u)':>12}{'approx |v_hat|':>16}")
for scale in [0.05, 0.10, 0.20]:
    v_hat = np.array([scale, 0.0])      # distortion in one direction
    mean, std, norm_v = mpr_check(v_hat)
    print(f"{norm_v:10.3f}{mean:12.4f}{std:12.4f}{norm_v:16.3f}")
```

```
   |v_hat|      E[m^u]    std(m^u)  approx |v_hat|
     0.050      1.0000      0.0500           0.050
     0.100      0.9999      0.1003           0.100
     0.200      0.9997      0.2022           0.200
```

The simulated conditional mean of m^u is one, and its conditional standard deviation tracks $|\hat{v}_t|$ closely, confirming (77.29).

A 10% distortion delivers a market price of risk near 0.10.

77.8 A risk-sensitive regulator

To see the robust decision rules and worst-case shocks concretely, we solve the recursive risk-sensitive control problem (77.1)–(77.3) directly.

Guess a value function $W(x) = x^\top \Omega x + \rho$ with Ω negative semidefinite. The risk-sensitive operator (77.4) acting on this quadratic form introduces the **risk adjustment**

$$\mathcal{D}(\Omega) = \Omega + \sigma \Omega C (I - \sigma C^\top \Omega C)^{-1} C^\top \Omega, \tag{77.30}$$

so that, replacing Ω by $\mathcal{D}(\Omega)$, the Bellman equation becomes an ordinary linear-quadratic one.

Iterating

$$F = (Q - \beta B^\top \mathcal{D} B)^{-1} (N - \beta B^\top \mathcal{D} A),$$

$$\Omega \leftarrow -R - F^\top Q F + (F^\top N + N^\top F) + \beta (A - BF)^\top \mathcal{D} (A - BF),$$

to a fixed point yields the optimal rule $i_t = -F x_t$.

The worst-case mean distortion is then linear in the state, $\hat{v}_t = G x_t$, with

$$G = \sigma (I - \sigma C^\top \Omega C)^{-1} C^\top \Omega (A - BF). \tag{77.31}$$

When $\sigma = 0$ we have $\mathcal{D}(\Omega) = \Omega$ and $G = 0$, recovering the standard regulator.

```
def solve_rslq(A, B, C, Q, R, β, σ, N=None,
               tol=1e-12, max_iter=100_000):
    """
    Solve the recursive risk-sensitive LQ problem

        U_t = -(x^T R x + i^T Q i + 2 i^T N x) + β R_t(U_{t+1})
        x_{t+1} = A x_t + B i_t + C w_{t+1}

    Returns the feedback rule F (i = -F x), the value matrix Ω,
    the closed-loop matrix A - B F, and the worst-case loading G (v = G x).
    """
    A, B, C, Q, R = map(np.atleast_2d, (A, B, C, Q, R))
    n, kw = A.shape[0], C.shape[1]
    if N is None:
        N = np.zeros((B.shape[1], n))
    Ω = -np.eye(n)                # negative-definite start
    Iw = np.eye(kw)
    for it in range(max_iter):
        M = Iw - σ * C.T @ Ω @ C
        D = Ω + σ * Ω @ C @ np.linalg.solve(M, C.T @ Ω)
        F = np.linalg.solve(Q - β * B.T @ D @ B, N - β * B.T @ D @ A)
        Acl = A - B @ F
        Ω_new = (-R - F.T @ Q @ F + (F.T @ N + N.T @ F)
                 + β * Acl.T @ D @ Acl)
        if np.max(np.abs(Ω_new - Ω)) < tol:
            Ω = Ω_new
            break
        Ω = Ω_new
    M = Iw - σ * C.T @ Ω @ C
    G = σ * np.linalg.solve(M, C.T @ Ω @ (A - B @ F))
    return F, Ω, A - B @ F, G
```

We first verify that at $\sigma = 0$ our solver reproduces QuantEcon's ordinary LQ regulator.

```
# Stable regulator
A = np.array([[0.9, 0.0],
              [0.0, 0.8]])
B = np.array([[1.0],
              [0.0]])
C = np.array([[0.3],
              [0.2]])
Q = np.array([[1.0]])
R = np.eye(2)
β = 0.95

# QuantEcon ordinary LQ
lq = qe.LQ(Q, R, A, B, C=C, beta=β)
P, F_qe, d = lq.stationary_values()

# Sigma-zero check
F0, Ω0, Acl0, G0 = solve_rslq(A, B, C, Q, R, β, σ=0.0)

print("QuantEcon LQ feedback rule F :", F_qe.flatten())
print("solve_rslq  feedback rule F  :", F0.flatten())
print("max |difference|             :", np.max(np.abs(F0 - F_qe)))
```

```
QuantEcon LQ feedback rule F : [0.52479699 0.        ]
```

(continues on next page)

(continued from previous page)

```
solve_rslq  feedback rule F  : [0.52479699 0.        ]
max |difference|             : 1.1102230246251565e-16
```

The two agree to machine precision.

Now we crank up the preference for robustness and inspect how the control rule and the worst-case shock respond.

```
σ_vals = [0.0, -0.3, -0.6]

print(f"{'σ':>7}{'F[0]':>10}{'F[1]':>10}{'G[0]':>10}{'G[1]':>10}")
for σ in σ_vals:
    F, Q, Acl, G = solve_rslq(A, B, C, Q, R, β, σ)
    print(f"{σ:7.2f}{F[0,0]:10.4f}{F[0,1]:10.4f}"
          f"{G[0,0]:10.4f}{G[0,1]:10.4f}")
```

```
      σ      F[0]      F[1]      G[0]      G[1]
   0.00    0.5248    0.0000   -0.0000   -0.0000
  -0.30    0.5357    0.0340    0.0532    0.1370
  -0.60    0.5490    0.0803    0.1154    0.3169
```

As σ becomes more negative:

- the feedback gain F grows, since the robust agent reacts *more aggressively* to the state, fearing the worst-case shock will amplify deviations;
- the worst-case loading G moves away from zero, as the malevolent player pushes the shock in the direction that hurts most.

Finally, let's see the worst-case distortion in action by simulating the controlled state under the *reference* model while displaying the conditional mean distortion $\hat{v}_t = Gx_t$ the robust agent is guarding against.

```
def simulate(A, B, C, F, G, T=60, seed=0):
    rng = np.random.default_rng(seed)
    n, kw = A.shape[0], C.shape[1]
    x = np.zeros((T + 1, n))
    v = np.zeros((T, kw))
    x[0] = np.array([1.0, 1.0])          # initial deviation
    for t in range(T):
        v[t] = (G @ x[t]).flatten()
        w = rng.standard_normal(kw)
        x[t + 1] = A @ x[t] + (B @ (-F @ x[t])).flatten() + (C @ w).flatten()
    return x, v

fig, axes = plt.subplots(1, 2, figsize=(11, 4))

for σ, color in zip([0.0, -0.6], ['C0', 'C3']):
    F, Q, Acl, G = solve_rslq(A, B, C, Q, R, β, σ)
    x, v = simulate(A, B, C, F, G, seed=42)
    axes[0].plot(x[:, 0], color=color, lw=2, label=f'$\\sigma={σ}$')
    axes[1].plot(v[:, 0], color=color, lw=2, label=f'$\\sigma={σ}$')

axes[0].set_xlabel('$t$')
axes[0].set_ylabel(r'controlled state $x_{1,t}$')
axes[0].axhline(0, color='k', lw=0.8, ls='--')
axes[0].set_title('controlled state')
axes[0].legend()
```

(continues on next page)

(continued from previous page)

```
axes[1].set_xlabel('$t$')
axes[1].set_ylabel(r'worst-case mean distortion $\hat v_{1,t}$')
axes[1].axhline(0, color='k', lw=0.8, ls='--')
axes[1].set_title('mean distortion')
axes[1].legend()
fig.tight_layout()
plt.show()
```

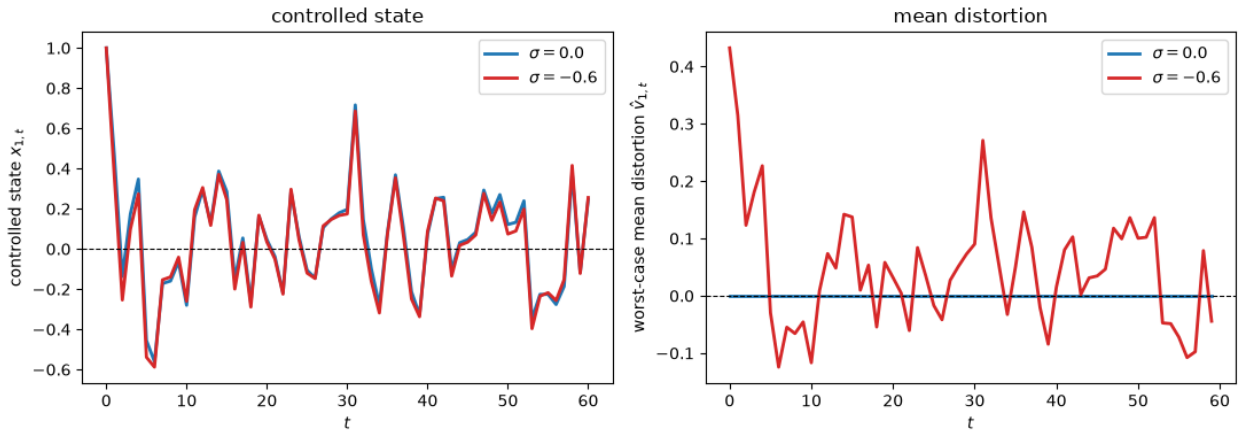


Fig. 77.4: Controlled state and distortion

For $\sigma = 0$ the distortion is identically zero, since the agent fully trusts the model.

For $\sigma < 0$ the robust agent's decisions are shaped by a nonzero worst-case distortion that feeds back on the state, exactly the mechanism that, through (77.29), inflates the market price of risk.

77.9 Exercises

Exercise 77.9.1

The Gaussian formula (77.5) says that for $U_{t+1} \sim N(\mu, s^2)$,

$$\mathcal{R}_t(U_{t+1}) = \mu + \frac{\sigma}{4} s^2.$$

Derive this result directly from the definition (77.4).

Hint: use the moment generating function of a normal random variable, $\mathbb{E}[\exp(aU)] = \exp(a\mu + a^2 s^2/2)$.

Solution

Here is one solution:

Start from the definition with $a = \sigma/2$:

$$\mathcal{R}_t(U_{t+1}) = \frac{2}{\sigma}\log\mathbb{E}\Big[\exp\Big(\frac{\sigma}{2}U_{t+1}\Big)\Big].$$

Since $U_{t+1} \sim N(\mu, s^2)$, the moment generating function gives

$$\mathbb{E}\Big[\exp\Big(\frac{\sigma}{2}U_{t+1}\Big)\Big] = \exp\Big(\frac{\sigma}{2}\mu + \frac{1}{2}\frac{\sigma^2}{4}s^2\Big).$$

Taking logs and multiplying by $2/\sigma$,

$$\mathcal{R}_t(U_{t+1}) = \frac{2}{\sigma}\left(\frac{\sigma}{2}\mu + \frac{\sigma^2}{8}s^2\right) = \mu + \frac{\sigma}{4}s^2.$$

Letting $\sigma \to 0$ recovers $\mathcal{R}_t(U_{t+1}) = \mu = \mathbb{E}_t U_{t+1}$, as expected.

Exercise 77.9.2

The observational-equivalence locus has a left endpoint $\underline{\sigma}$: the most negative σ for which (77.15) has a real solution.

Using the code from the lecture, find $\underline{\sigma}$ numerically for `theta2 = 0.01` and for `theta2 = 0.02`, and explain why a larger θ^2 shrinks the admissible range of σ.

Hint: the square root is real when the discriminant $(\beta - 1 + \sigma\theta^2)^2 + 4\sigma\theta^2 \geq 0$, evaluated at the relevant $\hat{\beta}$.

Solution

Here is one solution:

The boundary $\underline{\sigma}$ is the most negative σ at which a valid $\hat{\beta}$ can still be found.

We scan σ downward and stop when `beta_hat` can no longer return a real solution.

```
def σ_underbar(θ2, Rf, grid=np.linspace(-1e-6, -5e-4, 5000)):
    last_ok = 0.0
    for sg in grid:
        try:
            b = β_hat(sg, θ2, Rf)
            if np.isnan(Ω_scalar(b, sg, θ2)):
                break
            last_ok = sg
        except ValueError:
            break
    return last_ok

for θ2 in [0.01, 0.02]:
    sb = σ_underbar(θ2, Rf)
    print(f"θ2 = {θ2}:  σ_underbar ≈ {sb:.3e}")
```

```
θ2 = 0.01:  σ_underbar ≈ -2.105e-04
θ2 = 0.02:  σ_underbar ≈ -1.052e-04
```

A larger θ^2 means the marginal-utility martingale (77.14) carries a bigger innovation variance, so each unit of $|\sigma|$ generates a larger risk adjustment.

The discriminant in (77.15), which contains the term $4\sigma\theta^2 < 0$, turns negative at a *smaller* $|\sigma|$.

Hence the admissible range $(\underline{\sigma}, 0)$ shrinks as θ^2 grows.

Exercise 77.9.3

The market-price-of-risk approximation (77.29) states that $\text{std}_t(m^u_{t+1,t}) = [\exp(\hat{v}_t^\top \hat{v}_t) - 1]^{1/2} \approx |\hat{v}_t|$.

For the risk-sensitive regulator solved in the lecture (with A, B, C, Q, R, β as given there), compute the *exact* market price of risk $[\exp(\hat{v}_t^\top \hat{v}_t) - 1]^{1/2}$ as a function of σ, evaluated at the state $x = (1, 1)^\top$, and plot it together with the linear approximation $|\hat{v}_t|$.

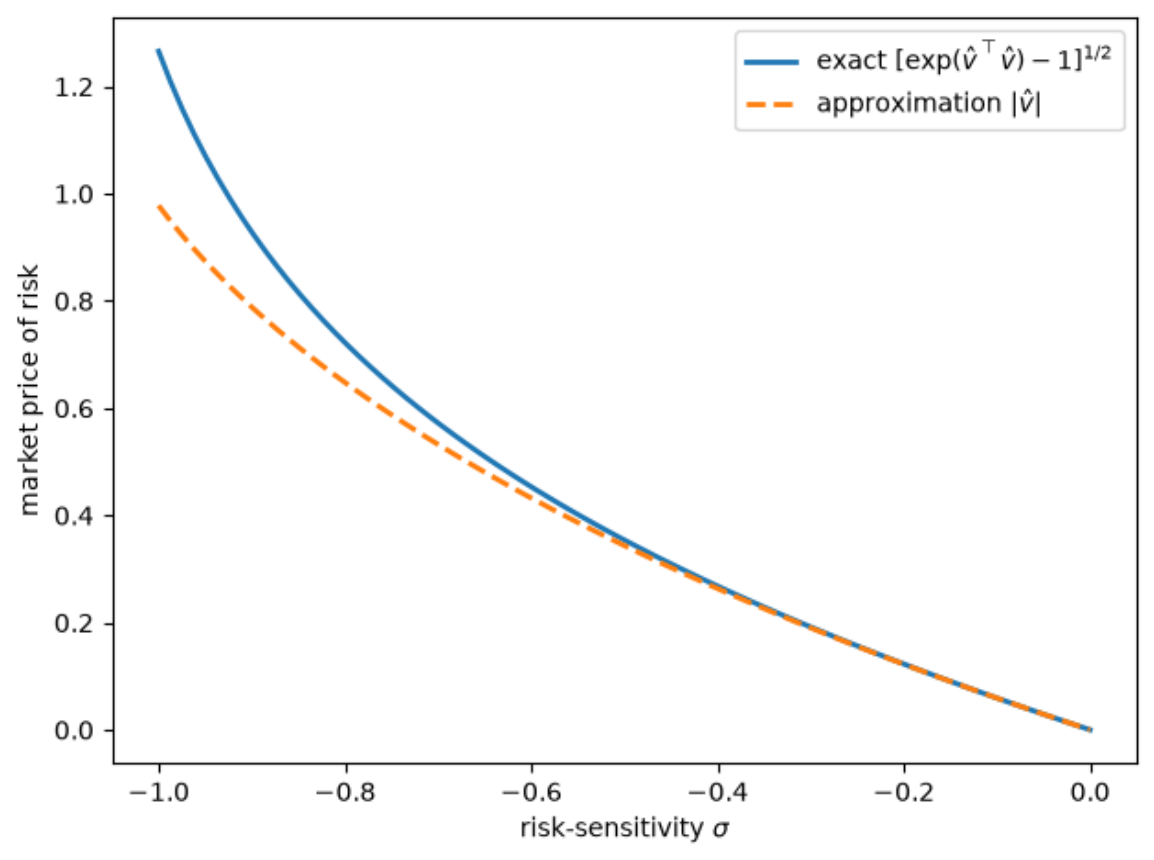
Comment on the range of σ over which the approximation is accurate.

Solution

Here is one solution:

We solve the regulator for a grid of σ values, evaluate $\hat{v}_t = Gx$ at $x = (1, 1)^\top$, and compare the exact and approximate market prices of risk.

```
x_eval = np.array([1.0, 1.0])
σ_grid = np.linspace(-1.0, 0.0, 80)

exact, approx = [], []
for σ in σ_grid:
    F, Ω, Acl, G = solve_rslq(A, B, C, Q, R, β, σ)
    v_hat = (G @ x_eval).flatten()
    nv2 = v_hat @ v_hat
    exact.append(np.sqrt(np.exp(nv2) - 1))
    approx.append(np.sqrt(nv2))            # |v_hat|

fig, ax = plt.subplots()
ax.plot(σ_grid, exact, lw=2,
        label=r'exact $[\exp(\hat v^\top \hat v)-1]^{1/2}$')
ax.plot(σ_grid, approx, lw=2, ls='--', label=r'approximation $|\hat v|$')
ax.set_xlabel(r'risk-sensitivity $\sigma$')
ax.set_ylabel('market price of risk')
ax.legend()
fig.tight_layout()
plt.show()
```



The two curves are nearly indistinguishable for small $|\hat{v}_t|$ (i.e. σ close to zero) because $\exp(z) - 1 \approx z$ when z is small.

They separate only when the preference for robustness, and hence the worst-case distortion, becomes large.

This is precisely the regime Hansen *et al.* [1999] emphasize: *small*, hard-to-detect distortions map almost linearly into the market price of risk.

77.10 Related lectures

This lecture connects to several others in the QuantEcon collection.

The underlying consumption-smoothing economics is developed in *The Permanent Income Model* and *Permanent Income II: LQ Techniques*, and the linear-quadratic control machinery is laid out in *LQ Control: Foundations*.

The reinterpretation of the market price of risk as a *price of model uncertainty*, calibrated through detection-error probabilities, is the subject of Barillas *et al.* [2009]; the broader semigroup treatment of robustness, pricing, and model detection is in Anderson *et al.* [2003].

Both build directly on Hansen *et al.* [1999].

CHAPTER

SEVENTYEIGHT

CERTAINTY EQUIVALENCE

Contents

- *Certainty Equivalence*
  - *Overview*
  - *A central problem of empirical economics*
  - *A formal setup*
  - *Estimated rules are not enough*
  - *An optimization problem*
  - *A linear-quadratic DP problem and certainty equivalence*
  - *A trouble with ad hoc expectations*
  - *Exercises*
  - *Concluding remarks*

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install quantecon
```

```
import numpy as np
import matplotlib.pyplot as plt
from quantecon import LQ
```

78.1 Overview

Simon [1956] and Theil [1957] established a celebrated *certainty equivalence* (CE) property for linear-quadratic (LQ) dynamic programming problems.

Their result justifies a convenient two-step algorithm:

1. **Optimize** under perfect foresight (treat future exogenous variables as known).
2. **Forecast** – substitute optimal forecasts for the unknown future values.

The striking insight is that these two steps are completely separable.

The decision rule that emerges from step 1 is *identical* to the decision rule for the original stochastic problem once optimal forecasts are substituted in step 2.

The decision rule does not depend on the variance of the shocks, but the *level* of the optimal value function *does*.

After describing the structure of the certainty equivalence property in detail, this lecture describes its role in rational expectations modeling.

We do so by drawing heavily on the introduction to Lucas and Sargent [1981].

In addition to learning the certainty equivalence principle, this lecture describes troubles with pre-rational expectations econometric policy evaluation procedures described by Lucas [1976].

Note

That volume collected early papers on rational expectations modeling and econometrics.

78.2 A central problem of empirical economics

To set the stage, Lucas and Sargent [1981] stated the central question for empirical economics that had been posed by Leonid Hurwicz ([Hurwicz, 1962]):

- Given observations on an agent's behavior in a particular economic environment, what can we infer about how that behavior *would have differed* had the environment been altered?

Note

Hurwicz formulates a notion of 'causality' as a context-specific concept that he casts in terms of a well posed decision problem.

This is the problem of policy-invariant structural inference in the following setting.

- Observations emerged under one environment or 'regime'
- We want to predict behavior under another 'regime'
- Unless we understand *why* agents behaved as they did in the historical regime, i.e., their purposes, we can't predict their behavior under the constraints they face in the new regime.

To confront the problem that Hurwicz had posed, Lucas and Sargent [1981] formulated the following decision framework.

78.3 A formal setup

Consider a single decision maker whose situation at date t is fully described by two state variables (x_t, z_t).

The environment $z_t \in S_1$ is selected by "nature" and evolves exogenously according to

$$z_{t+1} = f(z_t, \epsilon_{t+1}), \tag{78.1}$$

where the innovations $\{\epsilon_t\}$ are IID draws from a fixed CDF $\Phi(\cdot) : \mathcal{E} \to [0, 1]$.

The function $f : S_1 \times \mathcal{E} \to S_1$ is called the **decision maker's environment**.

The endogenous state $x_t \in S_2$ is under partial control of the agent.

Each period the agent selects an action $u_t \in U$.

A fixed technology $g : S_1 \times S_2 \times U \to S_2$ governs the transition

$$x_{t+1} = g(z_t,\, x_t,\, u_t). \tag{78.2}$$

The decision rule $h : S_1 \times S_2 \to U$ maps the agent's current situation into an action:

$$u_t = h(z_t,\, x_t). \tag{78.3}$$

The econometrician observes (some or all of) the process $\{z_t, x_t, u_t\}$, the joint motion of which is determined by (78.1), (78.2), and (78.3).

78.4 Estimated rules are not enough

Suppose we have estimated f, g, and h from a long time series generated under a fixed environment f_0.

This gives us $h_0 = T(f_0)$, where T is the (unknown) functional mapping environments into optimal decision rules.

But this single estimate, however precise, *reveals nothing* about how $T(f)$ varies with f.

Policy evaluation requires knowledge of the entire map $f \mapsto T(f)$.

Under an environment change $f_0 \to f_1$, agents will in general revise their decision rules $h_0 \to h_1 = T(f_1)$, rendering the estimated rule h_0 invalid for forecasting behavior under f_1.

Lucas [1976] and the introduction to Lucas and Sargent [1981] conclude that the only nonexperimental path forward is to recover the **return function** V from which h is derived as the solution to an optimization problem, and then re-solve that problem under the counterfactual environment f_1.

78.5 An optimization problem

Assume the agent selects h to maximize the expected discounted sum of current-period returns $V : S_1 \times S_2 \times U \to \mathbb{R}$:

$$E_0\left\{\sum_{t=0}^{\infty} \beta^t\, V(z_t,\, x_t,\, u_t)\right\}, \qquad 0 < \beta < 1, \tag{78.4}$$

given initial conditions (z_0, x_0), the environment f, and the technology g.

Here $E_0\{\cdot\}$ denotes expectation conditional on (z_0, x_0) with respect to the distribution of $\{z_1, z_2, \ldots\}$ induced by (78.1).

In principle, knowledge of V (together with g and f) allows one to compute $h = T(f)$ theoretically and hence to trace out $T(f)$ for any counterfactual f.

The essential question is whether V can itself be recovered from observations on $\{f, g, h\}$.

> **Note**
>
> The decision rule is in general a functional $h = T(f, g, V)$. The dependence on g and V is suppressed in the main text but made explicit when needed.

78.6 A linear-quadratic DP problem and certainty equivalence

Progress beyond the level of generality of the previous section requires restricting the primitives.

A productive restriction, exploited in the papers collected in Lucas and Sargent [1981], imposes *quadratic* V and *linear* g, which forces h to be linear.

As part of its computational tractability, this specialization delivers a striking structural result:

- the **certainty equivalence** theorem of Simon [1956] and Theil [1957].

78.6.1 Decomposition of h

Under quadratic V and linear g, the optimal decision rule h decomposes into two components applied in sequence.

Step 1 – Forecasting. Define the infinite sequence of optimal point forecasts of all current and future states of nature:

$$\tilde{z}_t \;=\; (z_t,\;\; {}_{t+1}z_t^e,\;\; {}_{t+2}z_t^e,\;\ldots) \;\in\; S_1^\infty, \tag{78.5}$$

where ${}_{t+j}z_t^e$ denotes the least-mean-squared-error forecast of z_{t+j} formed at time t.

The optimal forecast sequence is a (generally nonlinear) function of the current state:

$$\tilde{z}_t = h_2(z_t). \tag{78.6}$$

The function $h_2 : S_1 \to S_1^\infty$ depends entirely on the environment (f, Φ) and is obtained as the solution to a **pure forecasting problem**, with no reference to preferences or technology.

Step 2 – Optimization. Given the forecast sequence $\tilde{z}_t$, the optimal action is a **linear** function of $\tilde{z}_t$ and x_t:

$$u_t = h_1(\tilde{z}_t,\, x_t). \tag{78.7}$$

The function $h_1 : S_1^\infty \times S_2 \to U$ depends entirely on preferences (V) and technology (g) but **not** on the stochastic environment (f, Φ).

The ultimate decision rule is therefore the **composite**:

$$\boxed{h(z_t, x_t) \;=\; h_1[h_2(z_t),\; x_t].} \tag{78.8}$$

78.6.2 The separation principle

(78.8) embodies a clean **separation** of the two sources of dependence in h:

Component	Depends on	Independent of
h_1 (optimization)	V, g	f, Φ
h_2 (forecasting)	f, Φ	V, g

Since policy analysis concerns changes in f, and since h_1 is invariant to f, the policy analyst need only re-solve the forecasting problem $h_2 = S(f)$ under the new environment, keeping h_1 fixed.

The relationship of original interest, $h = T(f)$, then follows directly from (78.8).

78.6.3 Certainty equivalence and perfect foresight

The name "certainty equivalence" reflects a further implication of the LQ structure: the function h_1 can be derived as if the agent **knew the future path** $z_{t+1}, z_{t+2}, \ldots$ **with certainty** – i.e., by solving the deterministic problem in which $\tilde{z}_t$ is treated as the realized path rather than a forecast.

Randomness of the environment affects actions only through the forecast $\tilde{z}_t$; conditional on $\tilde{z}_t$, the optimization problem is deterministic.

This means the LQ problem decouples into:

- **Dynamic optimization under perfect foresight** – solve for h_1 from (V, g) by treating $\tilde{z}_t$ as known, yielding a standard deterministic LQ regulator problem independent of the environment (f, Φ).
- **Optimal linear prediction** – solve for $h_2 = S(f)$ from (f, Φ) using least-squares forecasting theory, which reduces to a standard Kalman/Wiener prediction formula when f is itself linear.

78.6.4 Cross-equation restrictions

A hallmark of the rational expectations hypothesis as it appears in this framework is that it ties together what would otherwise be free parameters in different equations.

The requirement that $\tilde{z}_t = h_2(z_t) = S(f)(z_t)$ – i.e., that agents' forecasts be *optimal* with respect to the *actual* law of motion f – imposes **cross-equation restrictions** between the parameters of the forecasting rule h_2 and the parameters of the environment f.

These restrictions, rather than any conditions on distributed lags within a single equation, are the operative empirical content of rational expectations.

Note

This is the message of Lucas [1976] and Sargent [1981].

The following code verifies the CE principle numerically.

We consider a simple scalar LQ problem:

$$y_{t+1} = a\, y_t + b\, u_t + \sigma\, \varepsilon_{t+1}, \qquad r(y_t, u_t) = -(q\, y_t^2 + r\, u_t^2)$$

and vary the noise standard deviation σ across a wide range.

The CE theorem predicts that:

- the **policy gain** F (the coefficient in $u_t = -F y_t$) is independent of σ, and
- the **value constant** d (the additive term in $V(y) = -y^\top P y - d$) grows with σ.

```
a, b_coeff = 0.9, 1.0
q, r = 1.0, 1.0
β = 0.95

A = np.array([[a]])
B = np.array([[b_coeff]])
R_mat = np.array([[q]])          # state cost
Q_mat = np.array([[r]])          # control cost

σ_vals = np.linspace(0.0, 3.0, 80)
```

(continues on next page)

(continued from previous page)

```
F_vals, d_vals = [], []

for σ in σ_vals:
    C = np.array([[σ]])
    lq = LQ(Q_mat, R_mat, A, B, C=C, beta=β)
    P, F, d = lq.stationary_values()
    F_vals.append(float(F[0, 0]))
    d_vals.append(float(d))

fig, axes = plt.subplots(1, 2, figsize=(12, 4))

axes[0].plot(σ_vals, F_vals, lw=2)
axes[0].set_xlabel('noise level $\\sigma$')
axes[0].set_ylabel('policy gain $F$')
axes[0].set_title('policy gain')
axes[0].set_ylim(0, 2 * max(F_vals) + 0.1)

axes[1].plot(σ_vals, d_vals, lw=2, color='C1')
axes[1].set_xlabel('noise level $\\sigma$')
axes[1].set_ylabel('value constant $d$')
axes[1].set_title('value constant')
fig.tight_layout()
plt.show()
```

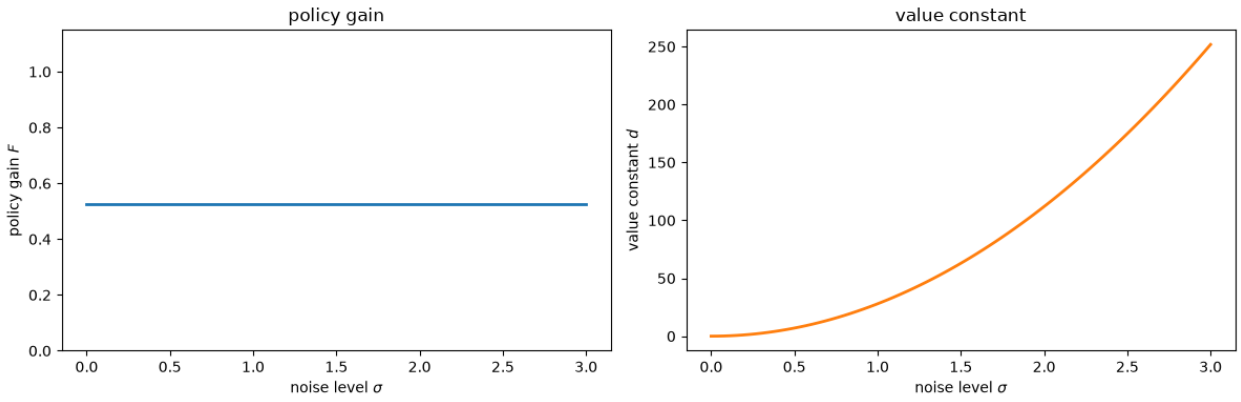


Fig. 78.1: CE: policy does not depend on noise

As the plot confirms, F (the policy gain) is *flat* across all noise levels, while the value constant d increases monotonically with σ.

This is the CE principle in action: uncertainty changes the value of the problem but not the optimal decision rule.

78.7 A trouble with ad hoc expectations

Prior practice, exemplified by the adaptive expectations mechanisms of Friedman [1956] and Cagan [1956], directly postulated a particular form of (78.6):

$$\theta_t^e = \lambda \sum_{i=0}^{\infty} (1-\lambda)^i \, \theta_{t-i}, \qquad 0 < \lambda < 1, \tag{78.9}$$

treating the coefficient λ as a free parameter to be estimated from data, with no reference to the underlying environment f.

The deficiency is not that (78.9) is a distributed lag – linear forecasting rules are perfectly acceptable simplifications.

The deficiency is that the **coefficients** of the distributed lag are left unrestricted by theory.

The mapping $h_2 = S(f)$ shows that optimal forecasting coefficients are *determined* by f: when f changes, h_2 changes, and so does h.

An estimated λ calibrated under f_0 is therefore non-structural and will give incorrect predictions whenever f is altered.

This is the econometric content of the critique delivered by Muth [1960].

Rational expectations equates the subjective distribution that agents use in forming $\tilde{z}_t$ to the objective distribution f that actually generates the data, thereby closing the model and eliminating free parameters in h_2.

78.8 Exercises

Exercise 78.8.1

Using the scalar LQ setup in the code cell above (with $a = 0.9$, $b = 1$, $q = r = 1$, $\beta = 0.95$), verify numerically that the value constant d satisfies $d \propto \sigma^2$.

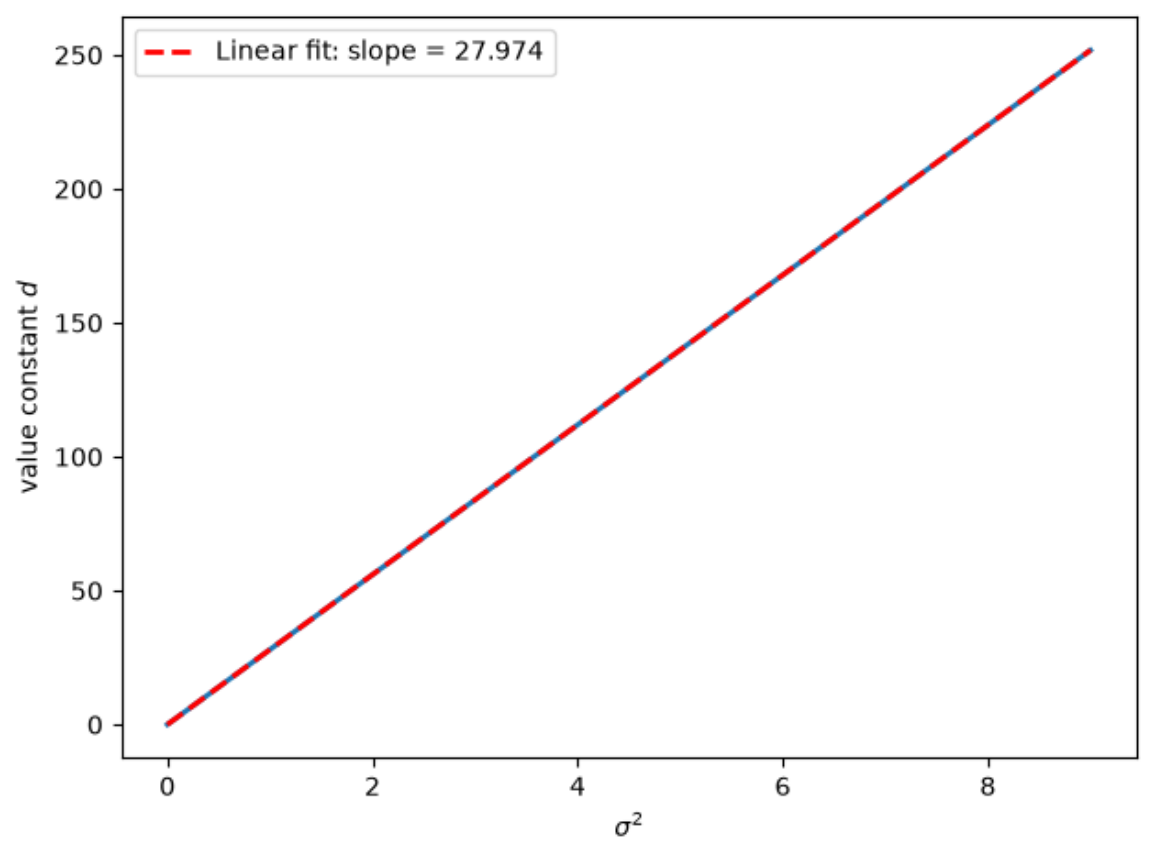
Hint: From the CE analysis, the value constant satisfies $d = \frac{\beta}{1-\beta} \operatorname{tr}(C^\top P C)$, and since $C = \sigma$ in the scalar case, this gives $d = \frac{\beta}{1-\beta} P \, \sigma^2$.

Confirm that a plot of d against σ^2 is linear and compute the theoretical slope $\frac{\beta}{1-\beta} P$.

Solution

```
σ_sq_vals = σ_vals ** 2

fig, ax = plt.subplots()
ax.plot(σ_sq_vals, d_vals, lw=2)
ax.set_xlabel('$\\sigma^2$')
ax.set_ylabel('value constant $d$')

coeffs = np.polyfit(σ_sq_vals, d_vals, 1)
ax.plot(σ_sq_vals, np.polyval(coeffs, σ_sq_vals),
        'r--', lw=2, label=f'Linear fit: slope = {coeffs[0]:.3f}')
ax.legend()
fig.tight_layout()
plt.show()

P_scalar = float(LQ(Q_mat, R_mat, A, B, C=np.zeros((1, 1)),
```

```
                    beta=β).stationary_values()[0].item())
theoretical_slope = β / (1 - β) * P_scalar
print(f"Empirical slope:    {coeffs[0]:.4f}")
print(f"Theoretical slope β/(1-β)*P = {theoretical_slope:.4f}")
```




```
Empirical slope:    27.9740
Theoretical slope β/(1-β)*P = 27.9740
```

The slope is indeed $\frac{\beta}{1-\beta}P$, confirming the analytic formula.

The value matrix P is determined entirely by preferences and technology, not by the noise level – a direct consequence of the certainty equivalence principle.

78.9 Concluding remarks

This sequel *certainty equivalence and model uncertainty* describes how to extend the certainty equivalence principle to a linear-quadratic setting in which a decision maker distrusts the transition dynamics specified in his baseline model.

CHAPTER

SEVENTYNINE

CERTAINTY EQUIVALENCE AND MODEL UNCERTAINTY

Contents

- *Certainty Equivalence and Model Uncertainty*
  - *Overview*
  - *Recap: ordinary certainty equivalence*
  - *Model uncertainty and robustness*
  - *Value function under robustness*
  - *Risk-sensitive preferences*
  - *Application: permanent income model*
  - *Summary*
  - *Exercises*

79.1 Overview

This is a sequel to *this lecture on certainty equivalence* that established an important *certainty equivalence* (CE) property for linear-quadratic (LQ) dynamic programming problems.

The property justifies a two-step algorithm for computing optimal decision rules:

1. *Optimize* under perfect foresight (treat future exogenous variables as known).
2. *Forecast* – substitute optimal forecasts for the unknown future values.

This lecture extends the certainty equivalence property in two directions motivated by [Hansen and Sargent, 2004]:

- *Model uncertainty and robustness.* What happens when the decision maker does not trust his model? A remarkable version of CE survives, but now the "forecasting" step uses a *distorted* probability distribution that the decision maker deliberately tilts against himself in order to achieve robustness.
- *Risk-sensitive preferences.* A mathematically equivalent reformulation interprets the same decision rules through recursive risk-sensitive preferences.

  The robustness parameter θ and the risk-sensitivity parameter σ are linked by $\theta = -\sigma^{-1}$.

We illustrate all three settings – ordinary CE, robust CE, and the permanent income application – with Python code using `quantecon`.

79.1.1 Model features

- Linear transition laws and quadratic objectives (LQ framework).
- Ordinary CE: optimal policy independent of noise variance.
- Robust CE: distorted forecasts replace baseline model forecasts; policy function depends on θ.
- Permanent income application: Hall's martingale, precautionary savings under robustness, and observational equivalence between robustness and patience.

This lecture draws on [Hansen and Sargent, 2004] and [Hansen and Sargent, 2008].

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install quantecon
```

We use the following imports:

```
import numpy as np
import matplotlib.pyplot as plt
from quantecon import LQ, RBLQ
```

79.2 Recap: ordinary certainty equivalence

The *companion lecture* established the CE property in detail.

Here we collect only the elements needed for the robustness extension below.

The state vector $y_t = \begin{bmatrix} x_t \\ z_t \end{bmatrix}$ has an exogenous component z_t with transition law

$$z_{t+1} = f(z_t,\, \epsilon_{t+1}) \tag{79.1}$$

and an endogenous component x_t obeying

$$x_{t+1} = g(x_t,\, z_t,\, u_t). \tag{79.2}$$

Under the LQ assumption (quadratic return $r(y,u) = -y^\top Q y - u^\top R u$, linear f and g, Gaussian shocks), the optimal decision rule h decomposes as $u_t = h_1(x_t,\, h_2 \cdot z_t)$ where h_1 solves a nonstochastic control problem and h_2 solves an optimal forecasting problem.

The optimal value function is $V(y_0) = -y_0^\top P\, y_0 - p$ where, writing $z_{t+1} = f_1 z_t + f_2 \epsilon_{t+1}$:

- P is the fixed point of an operator $T(P; r, g, f_1)$ that does *not* involve the volatility matrix f_2, so neither P nor the decision rule h depends on the noise loadings.
- The constant $p = \beta/(1-\beta)\,\mathrm{tr}(f_2^\top P f_2)$ grows with volatility.

Uncertainty lowers the value (larger p) but does not alter behaviour.

The following code sets up a scalar LQ problem and confirms that the policy gain F is invariant to the noise level σ while d grows with it.

```
a, b_coeff = 0.9, 1.0
q, r = 1.0, 1.0
β = 0.95

A = np.array([[a]])
```

(continues on next page)

(continued from previous page)

```
B = np.array([[b_coeff]])
Q_mat = np.array([[q]])           # state cost
R_mat = np.array([[r]])           # control cost

σ_vals = np.linspace(0.0, 3.0, 80)
F_vals, d_vals = [], []

for σ in σ_vals:
    C = np.array([[σ]])
    lq = LQ(R_mat, Q_mat, A, B, C=C, beta=β)
    P, F, d = lq.stationary_values()
    F_vals.append(float(F[0, 0]))
    d_vals.append(float(d))

fig, axes = plt.subplots(1, 2, figsize=(12, 4))

axes[0].plot(σ_vals, F_vals, lw=2)
axes[0].set_xlabel('noise level $\\sigma$')
axes[0].set_ylabel('policy gain $F$')
axes[0].set_ylim(0, 2 * max(F_vals) + 0.1)

axes[1].plot(σ_vals, d_vals, lw=2, color='darkorange')
axes[1].set_xlabel('noise level $\\sigma$')
axes[1].set_ylabel('value constant $d$')

plt.tight_layout()
plt.show()
```

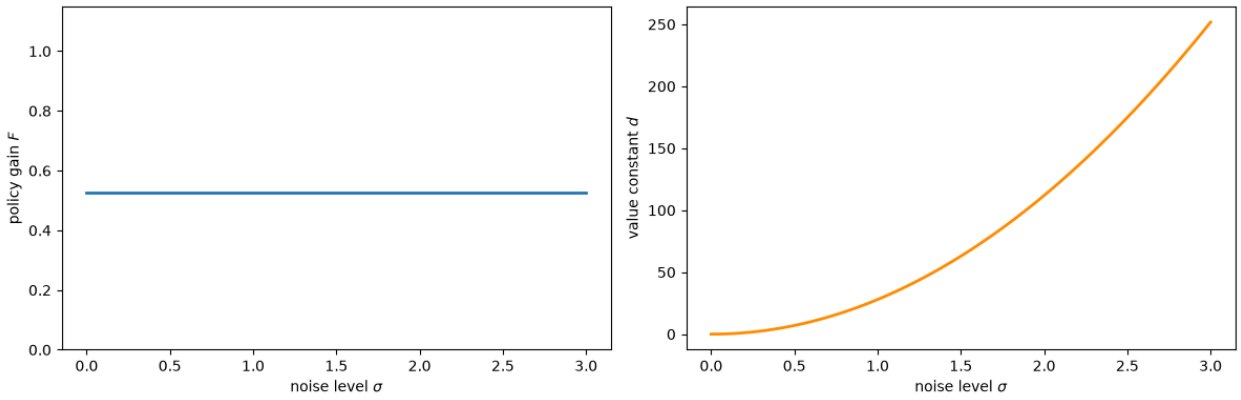


Fig. 79.1: CE principle – policy vs. value

79.3 Model uncertainty and robustness

79.3.1 Setup and the multiplier problem

The decision maker in Simon and Theil's setting knows his model exactly – he has no doubt about the transition law (79.1).

Now suppose he suspects that the true data-generating process is

$$z_{t+1} = f(z_t,\ \epsilon_{t+1} + w_{t+1}) \tag{79.3}$$

where $w_{t+1} = \omega_t(x^t, z^t)$ is a misspecification term chosen by an adversarial "nature."

The decision maker believes his approximating model is a good approximation in the sense that

$$\hat{\mathbb{E}}\left[\sum_{t=0}^{\infty} \beta^t\, w_{t+1}^\top w_{t+1} \,\Big|\, y_0\right] \le \eta_0, \tag{79.4}$$

where η_0 parametrises the tolerated misspecification budget and $\hat{\mathbb{E}}$ is the expectation under the distorted law (79.3).

To construct a *robust* decision rule the decision maker solves the **multiplier problem** – a two-player zero-sum dynamic game:

$$\min_{\{w_{t+1}\}} \max_{\{u_t\}} \hat{\mathbb{E}}\left[\sum_{t=0}^{\infty} \beta^t \Big\{ r(y_t, u_t) + \theta\beta\, w_{t+1}^\top w_{t+1} \Big\} \,\Big|\, y_0\right] \tag{79.5}$$

where $\theta > 0$ penalises large distortions.

A larger θ shrinks the feasible misspecification set; as $\theta \to \infty$ the problem reduces to ordinary LQ.

The Markov perfect equilibrium of (79.5) delivers a *robust* rule $u_t = h(x_t, z_t)$ together with a worst-case distortion process $w_{t+1} = W(x_t, z_t)$.

79.3.2 Stackelberg timing and the modified CE

The Markov perfect equilibrium *conceals* a form of CE.

To reveal it, Hansen and Sargent [2001] impose a **Stackelberg timing protocol**: at time 0, the *minimising* player commits once and for all to a plan $\{w_{t+1}\}$, after which the *maximising* player chooses u_t sequentially.

This makes the minimiser the Stackelberg leader.

To describe the leader's committed plan, introduce "big-letter" state variables (X_t, Z_t) (same dimensions as (x_t, z_t)) that encode the leader's pre-committed strategy:

$$\begin{aligned} w_{t+1} &= W(X_t, Z_t), \\ X_{t+1} &= g(X_t, Z_t,\ h(X_t, Z_t)), \\ Z_{t+1} &= f(Z_t,\ W(X_t, Z_t) + \epsilon_{t+1}). \end{aligned} \tag{79.6}$$

Summarised with $Y_t = \begin{bmatrix} X_t \\ Z_t \end{bmatrix}$:

$$Y_{t+1} = MY_t + N\epsilon_{t+1}, \qquad w_{t+1} = W(Y_t). \tag{79.7}$$

The maximising player then faces an *ordinary* dynamic programming problem subject to his own dynamics (79.2), the distorted z-law (79.3), and the exogenous process (79.7).

His optimal rule takes the form

$$u_t = \tilde{H}(x_t, z_t, Y_t). \tag{79.8}$$

Başar and Bernhard [2008] and Hansen and Sargent [2008] establish that at equilibrium (with "big K = little k" imposed) this collapses to

$$\tilde{H}(X_t, Z_t, Y_t) = h(Y_t), \tag{79.9}$$

the *same* rule as the Markov perfect equilibrium of (79.5).

79.3.3 Modified separation principle

The Stackelberg timing permits an Euler-equation approach.

The two-step algorithm becomes:

The first step is unchanged: solve the same nonstochastic control problem as before, with $\mathbf{z}_t = (z_t, z_{t+1}, ...)$ treated as known, giving $u_t = h_1(x_t, \mathbf{z}_t)$.

The second step is modified: form forecasts using the *distorted* law of motion (79.7). By the linearity and Gaussianity of the system,

$$\hat{\mathbb{E}}[\mathbf{z}_t \mid z^t, Y^t] = \hat{h}_2 \begin{bmatrix} z_t \\ Y_t \end{bmatrix} \tag{79.10}$$

where $\hat{\mathbb{E}}$ uses the distorted model.

Substituting (79.10) into h_1 and imposing $Y_t = y_t$ gives the robust rule

$$u_t = h_1\big(x_t,\ \hat{h}_2 \cdot y_t\big) = h(x_t, z_t). \tag{79.11}$$

This is the modified CE: *step 1 is identical to the non-robust case*; only step 2 changes, using distorted rather than rational forecasts.

In contrast to ordinary CE, the robust policy *does* change as θ varies.

As $\theta \to \infty$ (no robustness) the robust policy converges to the standard LQ policy.

```
σ_fixed = 1.0
C_fixed = np.array([[σ_fixed]])

lq_std = LQ(R_mat, Q_mat, A, B, C=C_fixed, beta=β)
P_std, F_std_arr, d_std = lq_std.stationary_values()
F_standard = float(F_std_arr[0, 0])
P_standard = float(P_std[0, 0])

θ_vals = np.linspace(2.0, 30.0, 120)   # restrict attention to a numerically stable␣
↪range
F_rob_vals, P_rob_vals = [], []

for θ in θ_vals:
    rblq = RBLQ(R_mat, Q_mat, A, B, C_fixed, β, θ)
    F_rob, K_rob, P_rob = rblq.robust_rule()
    F_rob_vals.append(float(F_rob[0, 0]))
    P_rob_vals.append(float(P_rob[0, 0]))

fig, axes = plt.subplots(1, 2, figsize=(12, 4))
```

(continues on next page)

(continued from previous page)

```
axes[0].plot(θ_vals, F_rob_vals, lw=2, label='Robust $F(\\theta)$')
axes[0].axhline(F_standard, color='r', linestyle='--', lw=2,
                label=f'Standard LQ ($F = {F_standard:.3f}$)')
axes[0].set_xlabel('robustness parameter $\\theta$')
axes[0].set_ylabel('policy gain $F$')
axes[0].legend()

axes[1].plot(θ_vals, P_rob_vals, lw=2, color='purple',
             label='Robust $P(\\theta)$')
axes[1].axhline(P_standard, color='r', linestyle='--', lw=2,
                label=f'Standard LQ ($P = {P_standard:.3f}$)')
axes[1].set_xlabel('robustness parameter $\\theta$')
axes[1].set_ylabel('value matrix $P$')
axes[1].legend()

plt.tight_layout()
plt.show()
```

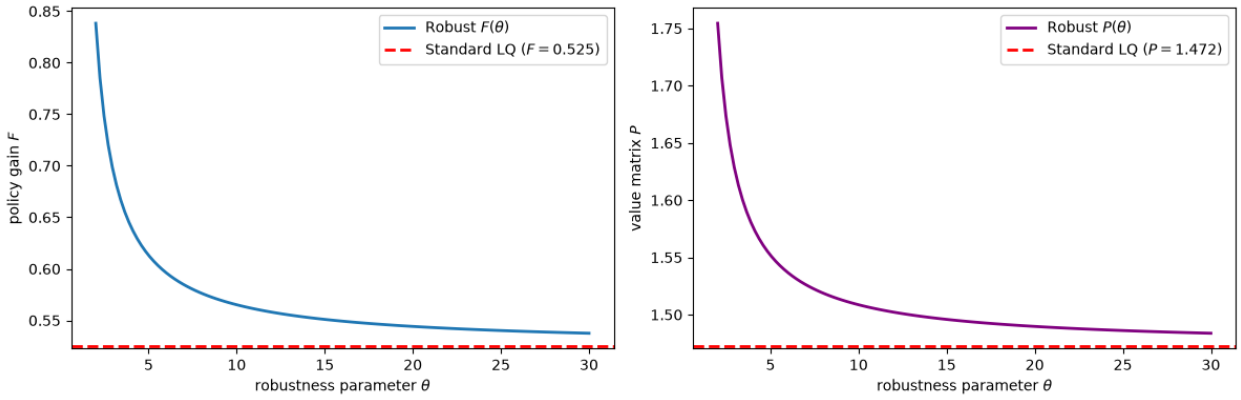


Fig. 79.2: Robust policy varies with θ

Observe that for small θ (strong preference for robustness) both F and P deviate substantially from their non-robust counterparts, converging to the standard values as $\theta \to \infty$.

This contrasts sharply with ordinary CE: under robustness, *both the policy gain and the value matrix depend on the robustness parameter θ and the noise-loading matrix C*.

79.4 Value function under robustness

Under a preference for robustness, the optimised value of (79.5) is again quadratic,

$$V(y_0) = -y_0^\top P\, y_0 - p, \tag{79.12}$$

but now *both P and p* depend on the volatility parameter f_2.

Specifically, P is the fixed point of the composite operator $T \circ \mathcal{D}$ where T is the same Bellman operator as in the non-robust case and $\mathcal{D}$ is the **distortion operator**:

$$\mathcal{D}(P) = \mathcal{D}(P;\, f_2,\, \theta). \tag{79.13}$$

Given the fixed point $P = T(\mathcal{D}(P))$, the constant is

$$p = p(P;\, f_2,\, \beta,\, \theta). \tag{79.14}$$

Despite P now depending on f_2, a form of CE still prevails: the same decision rule (79.11) also emerges from the *nonstochastic* game that maximises (79.5) subject to (79.2) and

$$z_{t+1} = f(z_t,\, w_{t+1}), \tag{79.15}$$

i.e., setting $\epsilon_{t+1} \equiv 0$.

The presence of randomness lowers the value (the constant p) but does not change the decision rule.

79.5 Risk-sensitive preferences

Building on Jacobson [1973] and Whittle [1990], Hansen and Sargent [2004] showed that the same decision rules can be reinterpreted through **risk-sensitive preferences**.

Suppose the decision maker *fully trusts* his model

$$y_{t+1} = A\, y_t + B\, u_t + C\, \epsilon_{t+1} \tag{79.16}$$

but evaluates stochastic processes according to the recursion

$$U_t = r(y_t, u_t) + \beta\, \mathcal{R}_t(U_{t+1}) \tag{79.17}$$

where the *risk-adjusted* continuation operator is

$$\mathcal{R}_t(U_{t+1}) = \frac{2}{\sigma} \log \mathbb{E}\left[\exp\left(\frac{\sigma U_{t+1}}{2}\right) \,\middle|\, y^t\right], \qquad \sigma \leq 0. \tag{79.18}$$

When $\sigma = 0$, L'Hôpital's rule recovers the standard expectation operator.

When $\sigma < 0$, $\mathcal{R}_t$ penalises right-tail risk in the continuation utility U_{t+1}.

For a candidate quadratic continuation value $U^e_{t+1} = -y^\top_{t+1} \Omega\, y_{t+1} - \rho$, let $\widehat{y}_{t+1} \equiv Ay_t + Bu_t$ denote the conditional mean of y_{t+1}.

Evaluating $\mathcal{R}_t$ via the log-moment-generating function of the Gaussian distribution yields

$$\mathcal{R}_t U^e_{t+1} = -\widehat{y}^\top_{t+1} \mathcal{D}(\Omega)\, \widehat{y}_{t+1} - \widehat{\rho} \tag{79.19}$$

where $\mathcal{D}$ is the *same* distortion operator as in (79.13) with $\theta = -\sigma^{-1}$, and $\widehat{\rho}$ is the corresponding scalar adjustment term.

Consequently, the risk-sensitive Bellman equation has the *same* fixed point P as the robust control problem, and therefore the *same decision rule* $u_t = -Fy_t$.

> **Key equivalence:** robust control with parameter θ and risk-sensitive control with parameter $\sigma = -\theta^{-1}$ produce identical decision rules.

79.6 Application: permanent income model

We now illustrate all of the above in a concrete linear-quadratic permanent income model.

79.6.1 Model setup

A consumer receives an exogenous endowment process $\{z_t\}$ and allocates it between consumption c_t and savings x_t to maximise

$$-\mathbb{E}_0 \sum_{t=0}^{\infty} \beta^t (c_t - b)^2, \qquad \beta \in (0,1) \tag{79.20}$$

where b is a bliss level of consumption.

Defining the **marginal utility of consumption** $\mu_{ct} \equiv b - c_t$ (the control), the budget constraint and endowment process are

$$x_{t+1} = R\, x_t + z_t - b + \mu_{ct} \tag{79.21}$$

$$z_{t+1} = \mu_d(1-\rho) + \rho\, z_t + c_d(\epsilon_{t+1} + w_{t+1}) \tag{79.22}$$

where $R > 1$ is the gross return on savings, $|\rho| < 1$, and w_{t+1} is an optional shock-mean distortion representing model misspecification.

After absorbing the constants $-b$ and $\mu_d(1-\rho)$ by augmenting the state vector, or equivalently by working with deviations from steady state, setting $w_{t+1} \equiv 0$ and taking $Q = 0$ (return depends only on the control μ_{ct}) and $R_{\text{ctrl}} = 1$ puts this in the standard LQ form

$$y_t = \begin{bmatrix} x_t \\ z_t \end{bmatrix}, \quad A = \begin{bmatrix} R & 1 \\ 0 & \rho \end{bmatrix}, \quad B = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad C = \begin{bmatrix} 0 \\ c_d \end{bmatrix}. \tag{79.23}$$

In the numerical code below we add a negligible `1e-8 * I` regularisation to the state-cost matrix to keep the Riccati computation well conditioned in Hall's unit-root case $\beta R = 1$.

We calibrate to parameters estimated by Hansen *et al.* [1999] from post-WWII U.S. data:

```
β_hat = 0.9971
R_rate = 1.0 / β_hat          # β*R = 1 (Hall's case)
ρ     = 0.9992
c_d   = 5.5819
σ_rs  = -2e-7                  # σ_hat < 0
θ_pi  = -1.0 / σ_rs           # θ = -1/σ_hat

A_pi = np.array([[R_rate, 1.0],
                 [0.0,    ρ]])
B_pi = np.array([[1.0],
                 [0.0]])
C_pi = np.array([[0.0],
                 [c_d]])
Q_pi = 1e-8 * np.eye(2)       # regularise for β*R = 1
R_pi = np.array([[1.0]])
```

79.6.2 Without robustness: Hall's martingale

Setting $\sigma = 0$ (no preference for robustness), the consumer's Euler equation is

$$\mathbb{E}_t[\mu_{c,t+1}] = (\beta R)^{-1} \mu_{ct}. \tag{79.24}$$

With $\beta R = 1$ (Hall's case), this is $\mathbb{E}_t[\mu_{c,t+1}] = \mu_{ct}$, i.e., the **marginal utility of consumption is a martingale** – equivalently, consumption follows a random walk.

The optimal policy is $\mu_{ct} = -F y_t$ where, from the solved-forward Euler equation, $F = [(R-1),\ (R-1)/(R-\rho)]$.

The resulting closed-loop projection onto the one-dimensional direction of μ_{ct} gives the scalar AR(1) representation

$$\mu_{c,t+1} = \varphi\,\mu_{ct} + \nu\,\epsilon_{t+1}. \tag{79.25}$$

```
F_pi     = np.array([[(R_rate - 1.0), (R_rate - 1.0) / (R_rate - ρ)]])
A_cl_std = A_pi - B_pi @ F_pi

ψ_std = 1.0 / (β_hat * R_rate)
ν_std = (R_rate - 1.0) * c_d / (R_rate - ρ)

print(f"ψ = {ψ_std:.6f}, ν = {ν_std:.4f}")
```

```
ψ = 1.000000, ν = 4.3777
```

79.6.3 With robustness: precautionary savings

Under a preference for robustness ($\sigma < 0$, $\theta < \infty$), the consumer uses distorted forecasts $\hat{\mathbb{E}}_t[\cdot]$ evaluated under the worst-case model.

The consumption rule takes the certainty-equivalent form

$$\mu_{ct} = -(1 - R^{-2}\beta^{-1})\left(R\,x_t + \hat{\mathbb{E}}_t\left[\sum_{j=0}^{\infty} R^{-j}(z_{t+j} - b)\right]\right) \tag{79.26}$$

where h_1 – the first step of the CE algorithm – is *identical* to the non-robust case.

Only the expectations operator changes.

The resulting AR(1) dynamics for μ_{ct} become:

$$\mu_{c,t+1} = \tilde{\varphi}\,\mu_{ct} + \tilde{\nu}\,\epsilon_{t+1} \tag{79.27}$$

with $\tilde{\varphi} < 1$, implying $\mathbb{E}_t[c_{t+1}] > c_t$ under the approximating model – a form of **precautionary saving**.

The observational equivalence formula (79.28) (derived below) immediately gives the robust AR(1) coefficient: $\tilde{\varphi} = 1/(\tilde{\beta}R)$ where $\tilde{\beta} = \tilde{\beta}(\sigma)$.

The innovation scale $\tilde{\nu}$ follows from the robust permanent income formula with the distorted persistence; Hansen *et al.* [1999] report $\tilde{\nu} \approx 8.0473$ for their calibration.

```
def beta_tilde(σ, β_hat_val, α_sq_val):
    """Observational-equivalence locus: β_tilde(σ)."""
    denom = 2.0 * (1.0 + σ * α_sq_val)
    numer = β_hat_val * (1.0 + β_hat_val)
    disc  = 1.0 - 4.0 * β_hat_val * (1.0 + σ * α_sq_val) / \
            (1.0 + β_hat_val) ** 2
    return (numer / denom) * (1.0 + np.sqrt(np.maximum(disc, 0.0)))

ν_rob = 8.0473
α_sq  = ν_rob ** 2
bt    = beta_tilde(σ_rs, β_hat, α_sq)
ψ_rob = 1.0 / (bt * R_rate)

print(f"β_tilde = {bt:.5f}, ψ_tilde = {ψ_rob:.4f}, ν_tilde = {ν_rob:.4f}")
```

```
β_tilde = 0.99953, φ_tilde = 0.9976, ν_tilde = 8.0473
```

```
rng = np.random.default_rng(0)
T_sim = 100

def simulate_ar1(φ, ν, shocks, mu0=0.0):
    path = np.empty(len(shocks) + 1)
    path[0] = mu0
    for t, ε in enumerate(shocks, start=1):
        path[t] = φ * path[t-1] + ν * ε
    return path

shock_path = rng.standard_normal(T_sim - 1)
mu0_init = 10.0
mu_std_path = simulate_ar1(φ_std, ν_std, shock_path, mu0=mu0_init)
mu_rob_path = simulate_ar1(φ_rob, ν_rob, shock_path, mu0=mu0_init)

fig, axes = plt.subplots(2, 1, figsize=(11, 6), sharex=True)
t_grid = np.arange(T_sim)

axes[0].plot(t_grid, mu_std_path, lw=2, label=f'$\\mu_{{ct}}$ (standard, $\\varphi={φ_
↪std:.4f}$)')
axes[0].axhline(0, color='k', lw=0.8, linestyle='--')
axes[0].set_ylabel('$\\mu_{ct}$')
axes[0].legend(loc='upper right')

axes[1].plot(t_grid, mu_rob_path, lw=2, color='darkorange',
             label=f'$\\mu_{{ct}}$ (robust, $\\tilde{{\\varphi}}={φ_rob:.4f}$)')
axes[1].axhline(0, color='k', lw=0.8, linestyle='--')
axes[1].set_xlabel('period $t$')
axes[1].set_ylabel('$\\mu_{ct}$')
axes[1].legend(loc='upper right')

plt.tight_layout()
plt.show()
```

79.6.4 Observational equivalence: robustness acts like patience

A key insight of Hansen and Sargent [2001] is that, in the permanent income model, a preference for robustness ($\sigma < 0$) is *observationally equivalent* to an increase in the discount factor from $\hat{\beta}$ to a larger value $\tilde{\beta}(\sigma)$, with σ set back to zero.

The equivalence locus is given by

$$\tilde{\beta}(\sigma) = \frac{\hat{\beta}(1+\hat{\beta})}{2(1+\sigma\alpha^2)}\left[1 + \sqrt{1 - \frac{4\hat{\beta}(1+\sigma\alpha^2)}{(1+\hat{\beta})^2}}\right] \tag{79.28}$$

where $\alpha^2 = \tilde{\nu}^2$ is the squared innovation loading in the robust AR(1) representation (79.27).

```
σ_range = np.linspace(-3e-7, 0.0, 200)
bt_vals = [beta_tilde(s, β_hat, α_sq) for s in σ_range]
bt_check = beta_tilde(σ_rs, β_hat, α_sq)

fig, ax = plt.subplots(figsize=(9, 5))
```

(continues on next page)

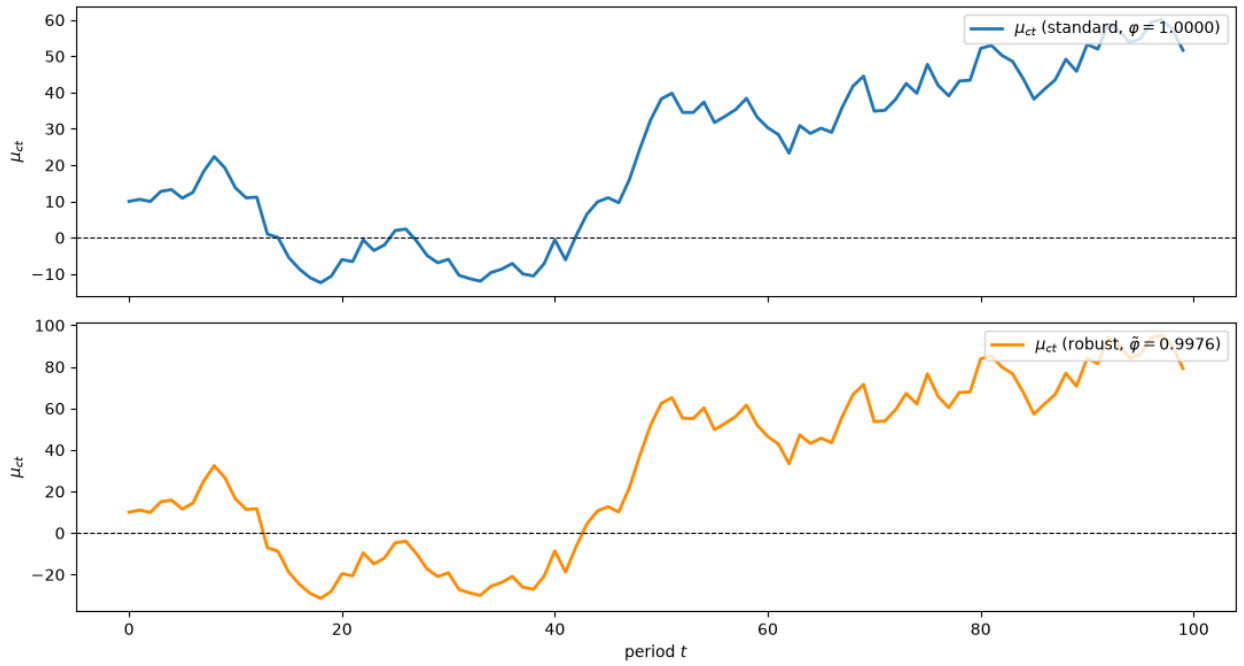


Fig. 79.3: Standard vs robust consumption paths

(continued from previous page)

```
ax.plot(-σ_range * 1e7, bt_vals, lw=2, color='steelblue',
        label='$\\tilde{\\beta}(\\sigma)$')
ax.axhline(β_hat, color='r', linestyle='--', lw=2,
           label=f'$\\hat{{\\beta}} = {β_hat}$')
ax.scatter([-σ_rs * 1e7], [bt_check], zorder=5, color='darkorange', s=80,
           label=f'$(\\hat{{\\sigma}},\\, \\tilde{{\\beta}}) '
                 f'= ({σ_rs:.0e},\\, {bt_check:.4f})$')
ax.set_xlabel('risk sensitivity $-\\sigma$ ($\\times 10^{-7}$)')
ax.set_ylabel('observationally equivalent discount factor $\\tilde{\\beta}$')
ax.legend()
plt.tight_layout()
plt.show()
```

The plot confirms the paper's key finding: *activating a preference for robustness is observationally equivalent – for consumption and saving behaviour – to increasing the discount factor*.

However, Hansen *et al.* [1999] show that the two parametrisations do *not* imply the same asset prices.

This happens because a preference for robustness generates different state-prices through the $\mathcal{D}(P)$ matrix that enters the stochastic discount factor.

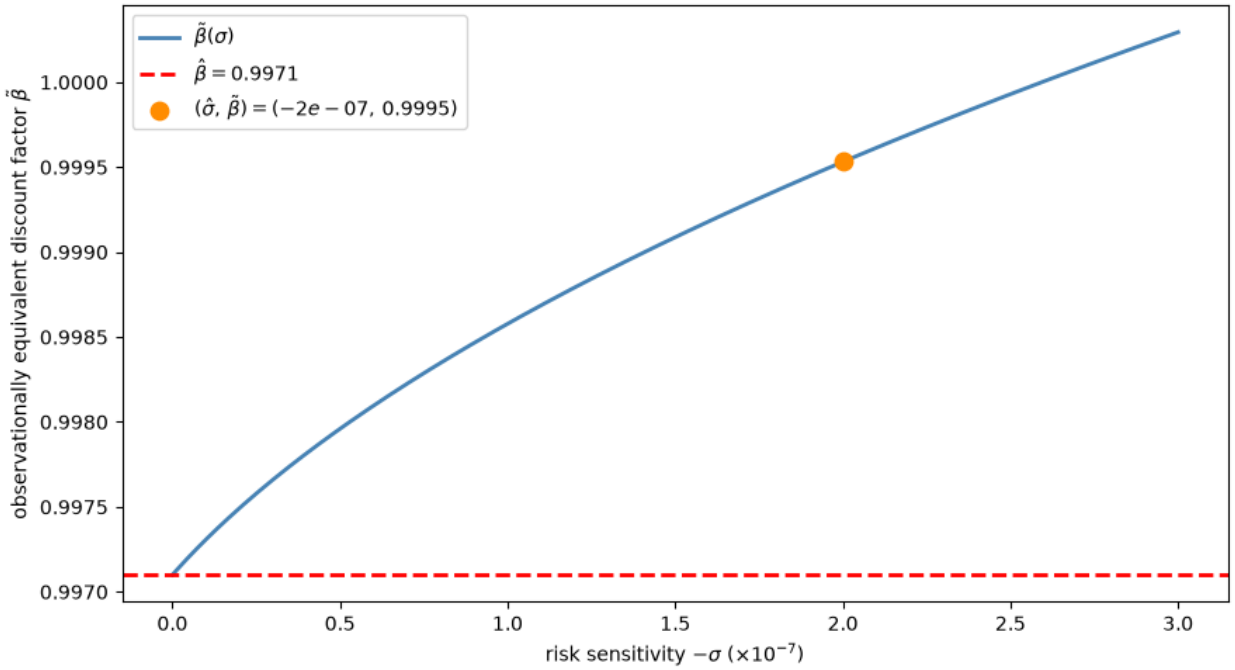


Fig. 79.4: Observational equivalence locus

79.7 Summary

The table below condenses the main results:

Setting	Policy depends on noise?	Forecasts used	CE survives?
Simon–Theil (ordinary LQ)	No	Rational	Yes
Robust control (multiplier)	Yes (P changes with f_2 and θ)	Distorted (worst-case)	Yes (modified)
Risk-sensitive preferences	Yes (same as robust)	Distorted (same)	Yes (same)

In all three cases, the decision maker can be described as following a two-step procedure: first solve a nonstochastic control problem, then form beliefs.

The difference is in which beliefs are formed in the second step.

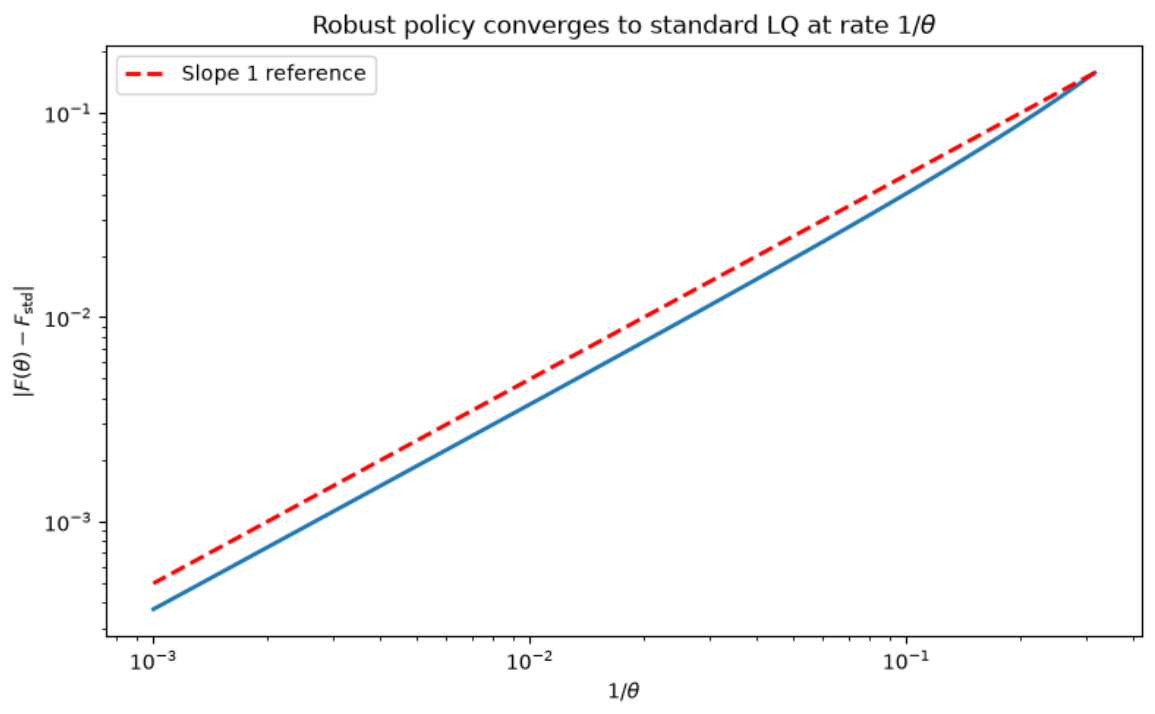
79.8 Exercises

Exercise 79.8.1

Show numerically that as $\theta \to \infty$ the robust policy $F(\theta)$ converges to the standard LQ policy F_{std} and that the rate of convergence is of order $1/\theta$. Plot $|F(\theta) - F_{\text{std}}|$ against $1/\theta$ on a log–log scale.

Solution

```
θ_large = np.logspace(0.5, 3.0, 100)
gap_vals = []

for θ in θ_large:
    rblq = RBLQ(R_mat, Q_mat, A, B, C_fixed, β, θ)
    F_r, _, _ = rblq.robust_rule()
    gap_vals.append(abs(float(F_r[0, 0]) - F_standard))

fig, ax = plt.subplots(figsize=(8, 5))
ax.loglog(1.0 / θ_large, gap_vals, lw=2)
ax.set_xlabel('$1/\\theta$')
ax.set_ylabel('$|F(\\theta) - F_{\\mathrm{std}}|$')
ax.set_title('Robust policy converges to standard LQ at rate $1/\\theta$')

x_ref = 1.0 / θ_large
ax.loglog(x_ref, x_ref * gap_vals[0] / x_ref[0],
          'r--', lw=2, label='Slope 1 reference')
ax.legend()
plt.tight_layout()
plt.show()
```



The log–log plot reveals an approximately linear relationship, confirming $O(1/\theta)$ convergence.

Exercise 79.8.2

Pick three values $\sigma_i < 0$ and verify numerically that the robust permanent income model with $(\sigma_i, \hat{\beta})$ produces the same policy matrix F as a suitably chosen non-robust model with $(0, \tilde{\beta}_i)$.

To find $\tilde{\beta}_i$, extract the AR(1) coefficient φ_i for μ_{ct} from the robust closed-loop dynamics and set $\tilde{\beta}_i = 1/(\varphi_i R)$.

Show that $\tilde{\beta}_i > \hat{\beta}$ in every case, confirming that robustness acts like increased patience.

Solution

For each σ_i we solve the robust problem with `RBLQ` and extract the AR(1) coefficient φ for μ_{ct} from the closed-loop dynamics $A_{\text{cl}} = A - BF_{\text{rob}}$.

If F is a left eigenvector of A_{cl} with eigenvalue φ, then $\mu_{ct} = -Fy_t$ satisfies $\mu_{c,t+1} = \varphi\,\mu_{ct} + \nu\,\epsilon_{t+1}$.

Setting $\tilde{\beta} = 1/(\varphi R)$ and solving a standard (non-robust) LQ problem with discount factor $\tilde{\beta}$ should reproduce F.

```
σ_trio = np.array([-5e-8, -1e-7, -2e-7])

for s in σ_trio:
    # Robust model: (σ, β_hat)
    θ_val = -1.0 / s
    rblq = RBLQ(R_pi, Q_pi, A_pi, B_pi, C_pi, β_hat, θ_val)
    F_rob, K_rob, P_rob = rblq.robust_rule()

    # Extract ψ from closed-loop under the approximating model
    A_cl = A_pi - B_pi @ F_rob
    ψ_rob = float((F_rob @ A_cl)[0, 1] / F_rob[0, 1])

    # Implied discount factor
    bt = 1.0 / (ψ_rob * R_rate)

    # Non-robust model with β_tilde
    lq_nr = LQ(R_pi, Q_pi, A_pi, B_pi, C=C_pi, beta=bt)
    P_nr, F_nr, d_nr = lq_nr.stationary_values()

    print(f"σ = {s:.1e},  θ = {θ_val:.1e},  β = {bt:.6f} (> β = {β_hat})")
    print(f"  ψ_rob = {ψ_rob:.8f}")
    print(f"  F_robust  = [{F_rob[0,0]:.6f}, {F_rob[0,1]:.6f}]")
    print(f"  F_non-rob = [{F_nr[0,0]:.6f}, {F_nr[0,1]:.6f}]")
    print(f"  |F_rob - F_nr| = {np.max(np.abs(F_rob - F_nr)):.2e}")
    print(f"  K (worst-case distortion): [{K_rob[0,0]:.2e}, {K_rob[0,1]:.2e}]")
    print()
```

```
σ = -5.0e-08,  θ = 2.0e+07,  β = 0.997472 (> β = 0.9971)
  ψ_rob = 0.99962721
  F_robust  = [0.003284, 0.884886]
  F_non-rob = [0.003284, 0.884894]
  |F_rob - F_nr| = 8.14e-06
  K (worst-case distortion): [2.48e-07, 6.68e-05]

σ = -1.0e-07,  θ = 1.0e+07,  β = 0.997953 (> β = 0.9971)
  ψ_rob = 0.99914484
  F_robust  = [0.003766, 1.014865]
  F_non-rob = [0.003766, 1.014863]
  |F_rob - F_nr| = 2.05e-06
  K (worst-case distortion): [5.68e-07, 1.53e-04]

σ = -2.0e-07,  θ = 5.0e+06,  β = 0.999522 (> β = 0.9971)
  ψ_rob = 0.99757687
  F_robust  = [0.005332, 1.437585]
  F_non-rob = [0.005333, 1.437465]
  |F_rob - F_nr| = 1.20e-04
  K (worst-case distortion): [1.01e-06, 2.72e-04]
```

The policy matrices F match to high precision, confirming observational equivalence for consumption and saving decisions.

In every case $\tilde{\beta} > \hat{\beta}$: a preference for robustness makes the agent behave as if he were more patient.

The non-zero worst-case distortion K in the robust model has no analogue in the non-robust model.

As Hansen *et al.* [1999] show, this is why the two parametrisations imply different asset prices even though saving plans coincide.

CHAPTER

EIGHTY

THE LQ PERMANENT INCOME MODEL

Contents

- *The LQ Permanent Income Model*
  - *Overview*
  - *The standard LQ permanent income model*
  - *Exercises*

80.1 Overview

This lecture studies consumption smoothing in a linear-quadratic (LQ) permanent income model.

It presents a rational-expectations version of the permanent income theories of Friedman [1956] and Hall [1978].

Throughout, we set $\beta R = 1$, so that the baseline consumer's subjective discount factor equals the bond price.

The model is useful for studying

- impulse response functions
- alternative state-space representations of the optimal decision rule
- cointegration of consumption and assets

We derive the consumer's optimal consumption function, present two state-space representations of the optimal decision rule, and illustrate them with two classic examples.

This is the first of three lectures on the LQ permanent income model.

The two sequels build directly on the tools developed here.

- *Consumption Smoothing with Incomplete and Complete Markets* studies the cross-section behavior of consumption and embeds the single consumer in closed economies with incomplete and complete markets.
- *Robust Consumption Smoothing and Precautionary Savings* studies a consumer who distrusts his model of income and engages in precautionary savings.

Let's begin with some imports.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.linalg import solve, inv, solve_discrete_lyapunov
from scipy.stats import norm
```

80.2 The standard LQ permanent income model

80.2.1 Setup

A consumer has preferences over consumption streams ordered by

$$\mathbb{E}_0 \sum_{t=0}^{\infty} \beta^t u(c_t) \tag{80.1}$$

where $\mathbb{E}_t$ is a mathematical expectation conditioned on the consumer's time-t information, c_t is time-t consumption, $u(c)$ is a strictly concave one-period utility function, and $\beta \in (0,1)$ is a discount factor.

The consumer maximizes (80.1) by choosing a plan $\{c_t, b_{t+1}\}_{t=0}^{\infty}$ subject to the sequence of budget constraints

$$c_t + b_t = R^{-1} b_{t+1} + y_t, \quad t \geq 0 \tag{80.2}$$

where $\{y_t\}$ is an exogenous stationary endowment process, R is a constant gross risk-free interest rate, b_t is a one-period risk-free bond maturing at t, and b_0 is a given initial condition.

Note

For $t \geq 1$, b_t is chosen at time $t-1$.

The bond $b_t > 0$ represents debt owed by the consumer at the start of period t.

We assume $R^{-1} = \beta$.

The endowment or non-financial income process has the state-space representation

$$\begin{aligned} z_{t+1} &= \check{A} z_t + \check{C} w_{t+1} \\ y_t &= \check{G} z_t \end{aligned} \tag{80.3}$$

where w_{t+1} is IID with mean zero and identity covariance matrix, $\check{A}$ is a stable matrix (eigenvalues strictly less than one in modulus), and $\check{G}$ is a row vector.

The state confronting the household at t is $\begin{bmatrix} b_t & z_t^\top \end{bmatrix}^\top$, where b_t is its one-period debt due at the start of period t and z_t contains all variables useful for forecasting its future endowment.

To make the problem linear-quadratic, we adopt the **quadratic utility function**

$$u(c_t) = -\tfrac{1}{2}(c_t - \gamma)^2$$

where $\gamma > 0$ is a bliss level of consumption.

We allow c_t to be negative (a producer rather than a consumer).

We impose a **transversality condition**

$$\mathbb{E}_0 \sum_{t=0}^{\infty} \beta^t b_t^2 < +\infty, \tag{80.4}$$

which rules out Ponzi schemes.

80.2.2 Euler equation and certainty equivalence

With quadratic utility, the first-order conditions for the consumer's problem imply that

$$\mathbb{E}_t c_{t+1} = c_t \tag{80.5}$$

Note

Equation (80.5) says that consumption is a martingale.

This is the key implication of the LQ permanent income model.

It contrasts with models that have convex marginal utility ($u''' > 0$), where consumption is instead a submartingale.

Because the consumer maximizes a quadratic objective subject to a linear transition equation, the problem satisfies a **certainty-equivalence** property.

This implies that we can find the optimal plan by

- first solving the problem while pretending to have perfect foresight; this lets us express c_t as a function of b_t and the continuation sequence $\{y_{t+j}\}_{j=0}^{\infty}$
- then simply replace $\{y_{t+j}\}_{j=0}^{\infty}$ with $\{\mathbb{E}_t y_{t+j}\}_{j=0}^{\infty}$.

80.2.3 The optimal consumption function

Solving the budget constraint (80.2) forward, imposing the transversality condition, and taking conditional expectations gives

$$b_t = \sum_{j=0}^{\infty} \beta^j \mathbb{E}_t y_{t+j} - \frac{1}{1-\beta} c_t \tag{80.6}$$

Rearranging yields the **consumption function**

$$c_t = (1-\beta)\left[\sum_{j=0}^{\infty} \beta^j \mathbb{E}_t y_{t+j} - b_t\right] \tag{80.7}$$

Equivalently, with net interest rate r defined by $\beta = 1/(1+r)$,

$$c_t = \frac{r}{1+r}\left[\sum_{j=0}^{\infty} \beta^j \mathbb{E}_t y_{t+j} - b_t\right] \tag{80.8}$$

Evidently, consumption at t equals $r/(1+r)$ times total wealth, where total wealth is the sum of human wealth $\sum_{j=0}^{\infty} \beta^j \mathbb{E}_t y_{t+j}$ and financial wealth $-b_t$.

Using state-space representation (80.3) to evaluate the geometric sum of expected future endowments,

$$\sum_{j=0}^{\infty} \beta^j \mathbb{E}_t y_{t+j} = \check{G}(I - \beta \check{A})^{-1} z_t, \tag{80.9}$$

we obtain

$$c_t = (1-\beta)\left[\check{G}(I - \beta \check{A})^{-1} z_t - b_t\right] \tag{80.10}$$

This expresses c_t as a function of the state $[b_t,\ z_t^\top]^\top$ that confronts the household.

80.2.4 Representation 1: state (b_t, z_t)

Combining the endowment law of motion with the optimal debt dynamics (derived by substituting (80.10) into (80.2)) gives the following representation:

$$\begin{aligned} z_{t+1} &= \check{A}\, z_t + \check{C}\, w_{t+1} \\ b_{t+1} &= b_t + \check{G}[(I - \beta\check{A})^{-1}(\check{A} - I)]z_t \\ y_t &= \check{G}\, z_t \\ c_t &= (1-\beta)\left[\check{G}(I - \beta\check{A})^{-1} z_t - b_t\right] \end{aligned} \tag{80.11}$$

In this representation the **exogenous** state is z_t and the **endogenous** state is b_t.

We turn now to an alternative representation.

80.2.5 Representation 2: state (c_t, z_t)

Hall [1978] showed that the LQ permanent income model implies a representation in which the state consists of current consumption c_t and the exogenous endowment state z_t.

In this representation, b_t becomes an outcome rather than a state variable.

Shifting (80.7) forward, eliminating b_{t+1} via (80.2), and rearranging yields

$$c_{t+1} - c_t = (1-\beta)\sum_{j=0}^{\infty} \beta^j (\mathbb{E}_{t+1} y_{t+j+1} - \mathbb{E}_t y_{t+j+1}) \tag{80.12}$$

The right-hand side is $(1-\beta)$ times the time-$(t+1)$ **innovation** to the expected present value of the endowment stream.

Suppose the endowment has the (Wold) moving-average representation

$$y_{t+1} = d(L)\, w_{t+1}, \qquad d(L) = \sum_{j=0}^{\infty} d_j L^j \tag{80.13}$$

where $d(L) = \check{G}(I - \check{A}L)^{-1}\check{C}$.

Then

$$\mathbb{E}_{t+1} y_{t+j} - \mathbb{E}_t y_{t+j} = d_{j-1}\, w_{t+1} \tag{80.14}$$

Substituting (80.14) into (80.12) gives the key result

$$c_{t+1} - c_t = (1-\beta)\, d(\beta)\, w_{t+1}. \tag{80.15}$$

Here, $d(\beta) = \check{G}(I - \beta\check{A})^{-1}\check{C}$ is the **present value of the (Wold) moving-average coefficients**.

Thus, consumption is a **random walk** with innovation $(1-\beta)d(\beta)w_{t+1}$.

Combining (80.15) and (80.6) gives

$$\begin{aligned} c_{t+1} &= c_t + (1-\beta)\,\check{G}(I - \beta\check{A})^{-1}\check{C}\, w_{t+1} \\ b_t &= \check{G}(I - \beta\check{A})^{-1} z_t - \frac{1}{1-\beta}\, c_t \\ y_t &= \check{G}\, z_t \\ z_{t+1} &= \check{A}\, z_t + \check{C}\, w_{t+1} \end{aligned} \tag{80.16}$$

This representation reveals several important features of the optimal decision rule:

1. **State**: The state consists of the endogenous component c_t and the exogenous component z_t, with financial assets b_t encoded in c_t rather than carried as a separate state.
2. **Random walk**: Consumption is a random walk with innovation $(1-\beta)d(\beta)w_{t+1}$, which confirms that the Euler equation (80.5) is built into the solution and implies that consumption has no asymptotic stationary distribution.
3. **Box impulse response**: For all $j \geq 1$, the response of c_{t+j} to the innovation w_{t+1} is the constant $(1-\beta)d(\beta)$, giving a "box-shaped" impulse response.
4. **Cointegration**: Both c_t and b_t are nonstationary (unit-root processes), but the linear combination $(1-\beta)b_t + c_t$ is stationary.

   From (80.6),

$$(1-\beta)b_t + c_t = (1-\beta)\mathbb{E}_t \sum_{j=0}^{\infty} \beta^j y_{t+j} \tag{80.17}$$

The left side is the cointegrating residual.

80.2.6 Debt dynamics

Subtracting (80.16) (equation for b_t) at time t from the same equation at time $t+1$ and substituting gives

$$b_{t+1} - b_t = \check{G}(I - \beta\check{A})^{-1}(\check{A} - I)\, z_t \tag{80.18}$$

This shows that b_{t+1} is **predetermined** at time t as a function of z_t alone.

Solving backward from any t, b_t depends on the entire history $z^{t-1} = [z_{t-1}, \ldots, z_0]$ and the initial condition b_0.

Such **history dependence** is a hallmark of a consumption plan in various incomplete-markets economies.

80.2.7 Two classic examples

We illustrate formulas (80.16) with two examples.

In both, the endowment is $y_t = z_{1t} + z_{2t}$, where

$$\begin{pmatrix} z_{1,t+1} \\ z_{2,t+1} \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} z_{1t} \\ z_{2t} \end{pmatrix} + \begin{pmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{pmatrix} \begin{pmatrix} w_{1,t+1} \\ w_{2,t+1} \end{pmatrix} \tag{80.19}$$

Here z_{1t} is a **permanent** component of y_t and z_{2t} is a **purely transitory** component.

In the full-information example, the consumer observes the state z_t at time t, so he can reconstruct w_{t+1} from z_{t+1} and z_t.

Applying (80.16):

$$c_{t+1} - c_t = \sigma_1 w_{1,t+1} + (1-\beta)\,\sigma_2\, w_{2,t+1} \tag{80.20}$$

A unit increment to the permanent component z_{1t} raises consumption *one-for-one* permanently and causes **zero net saving**.

A unit increment to the purely transitory component raises consumption by only the fraction $(1-\beta)$ permanently, while the remaining fraction β is saved.

From (80.18):

$$b_{t+1} - b_t = -z_{2t} = -\sigma_2 w_{2t} \tag{80.21}$$

confirming that none of the permanent shock is saved, while all of the transitory shock is saved.

In the incomplete-information (Muth model) example, the consumer observes y_t and its history, but not z_{1t} and z_{2t} separately.

The appropriate approach uses an **innovations representation** derived by the Kalman filter.

At the Kalman filter steady state, the **Kalman gain** $K \in [0, 1]$ satisfies

$$K = \frac{\Sigma}{\Sigma + \sigma_2^2}, \qquad \Sigma = \frac{\sigma_1^2 + \sqrt{\sigma_1^4 + 4\sigma_1^2\sigma_2^2}}{2} \tag{80.22}$$

where K increases with the ratio σ_1^2/σ_2^2 (the variance of the permanent shock relative to the transitory shock).

The innovations representation expresses the endowment as an ARMA(1,1) in its own innovation $a_t = y_t - \mathbb{E}[y_t \mid y^{t-1}]$ (the one-step-ahead forecast error):

$$y_{t+1} = y_t - (1 - K)\, a_t + a_{t+1} \tag{80.23}$$

Here the coefficient $-(1 - K)$ on the lagged innovation reflects that only the fraction K of last period's surprise was treated as permanent; the remainder mean-reverts.

The scalar a_t is IID with variance $\Sigma + \sigma_2^2$.

Applying (80.16) to this innovation representation:

$$c_{t+1} - c_t = [1 - \beta(1 - K)]\, a_{t+1} \tag{80.24}$$

The consumer regards a fraction K of the innovation a_{t+1} as permanent and fraction $1 - K$ as transitory.

He permanently increments consumption by $K + (1 - \beta)(1 - K) = 1 - \beta(1 - K)$ of a_{t+1} and saves the remaining fraction $\beta(1 - K)$.

The first difference of income obeys a first-order moving average:

$$y_{t+1} - y_t = a_{t+1} - (1 - K)\, a_t \tag{80.25}$$

By contrast, the first difference of consumption is IID by (80.24).

80.2.8 Implementation

```
# Parameters
β = 0.95       # discount factor (so R = 1/β)
σ1 = 0.15      # std of permanent shock
σ2 = 0.30      # std of transitory shock

# Example 1: full information
A_check = np.array([[1.0, 0.0],
                    [0.0, 0.0]])
C_check = np.array([[σ1, 0.0],
                    [0.0, σ2]])
G_check = np.array([[1.0, 1.0]])

# Key matrix M = G(I - βA)^{-1}
IbA = np.eye(2) - β * A_check
M = G_check @ inv(IbA)   # shape (1, 2)

# Consumption impulse responses
h = (1 - β) * M @ C_check          # shape (1, 2)
irf_perm_ex1 = h[0, 0] / σ1       # response per unit std of permanent shock
```

(continues on next page)

(continued from previous page)

```
irf_trans_ex1 = h[0, 1] / σ2       # response per unit std of transitory shock

print("Example 1 (full information)")
print(f"  IRF c to permanent shock  (normalised): {irf_perm_ex1:.4f}   "
      f"(theory: 1.0)")
print(f"  IRF c to transitory shock (normalised): {irf_trans_ex1:.4f}   "
      f"(theory: {1-β:.4f})")
```

```
Example 1 (full information)
  IRF c to permanent shock  (normalised): 1.0000   (theory: 1.0)
  IRF c to transitory shock (normalised): 0.0500   (theory: 0.0500)
```

```
# Example 2: partial information
Σ = (σ1**2 + np.sqrt(σ1**4 + 4 * σ1**2 * σ2**2)) / 2
K = Σ / (Σ + σ2**2)

print("Example 2 (partial information)")
print(f"  Steady-state Kalman gain K = {K:.4f}")
print(f"  IRF c to unit innovation a_{{t+1}}: {1 - β*(1-K):.4f}")
print(f"  Fraction of innovation treated as permanent (K): {K:.4f}")
print(f"  Fraction saved: β(1-K) = {β*(1-K):.4f}")
```

```
Example 2 (partial information)
  Steady-state Kalman gain K = 0.3904
  IRF c to unit innovation a_{t+1}: 0.4209
  Fraction of innovation treated as permanent (K): 0.3904
  Fraction saved: β(1-K) = 0.5791
```

```
# Compare impulse responses
T = 30
irf_c_ex1_perm = np.ones(T) * irf_perm_ex1 * σ1
irf_c_ex1_trans = np.ones(T) * irf_trans_ex1 * σ2

irf_c_ex2 = np.ones(T) * (1 - β * (1 - K))   # per unit innovation a_t

fig, axes = plt.subplots(1, 2, figsize=(12, 4))

axes[0].axhline(0, color='k', linewidth=0.8)
axes[0].step(range(T), irf_c_ex1_perm, where='post',
             label='permanent shock ($z_1$)', color='C0', lw=2)
axes[0].step(range(T), irf_c_ex1_trans, where='post',
             label='transitory shock ($z_2$)', color='C3',
             linestyle='--', lw=2)
axes[0].set_xlabel('periods after shock')
axes[0].set_ylabel('response of $c$')
axes[0].set_title('full information')
axes[0].legend()

axes[1].axhline(0, color='k', linewidth=0.8)
axes[1].step(range(T), irf_c_ex2, where='post',
             label=f'unit innovation $a_{{t+1}}$ (K = {K:.2f})',
             color='C4', lw=2)
axes[1].set_xlabel('periods after shock')
axes[1].set_ylabel('response of $c$')
axes[1].set_title('partial information')
```

(continues on next page)

(continued from previous page)

```
axes[1].legend()
fig.tight_layout()
plt.show()
```

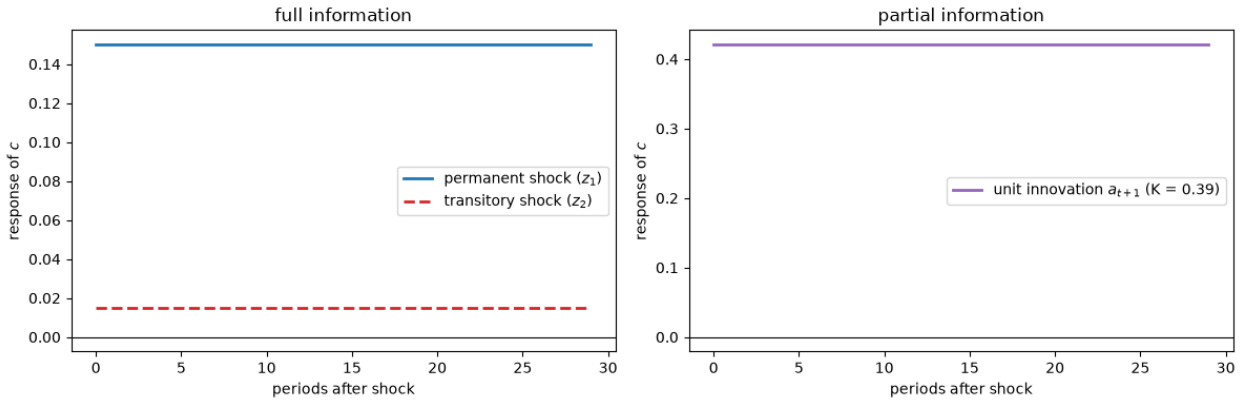


Fig. 80.1: Consumption impulse responses

Note

The impulse responses have the "box" shape characteristic of the LQ permanent income model: once a shock occurs, consumption shifts permanently to a new level and stays there.

The two representations and examples developed here are the foundation for the two sequel lectures.

Consumption Smoothing with Incomplete and Complete Markets uses the (c_t, z_t) representation (80.16) to study how the cross-section distribution of consumption evolves in closed economies with incomplete and complete markets.

Robust Consumption Smoothing and Precautionary Savings studies a consumer who distrusts the endowment process (80.3) and engages in precautionary savings.

80.3 Exercises

Exercise 80.3.1

This exercise verifies the response of consumption and debt to permanent and transitory shocks in the full-information two-factor model.

Consider the model of (80.19) with the calibration used above.

Suppose the economy starts at $z_0 = 0$ and $b_0 = 0$ and is hit by a single shock at $t = 1$: either a unit permanent shock $w_1 = (1, 0)^\top$ or a unit transitory shock $w_1 = (0, 1)^\top$, with no further shocks thereafter.

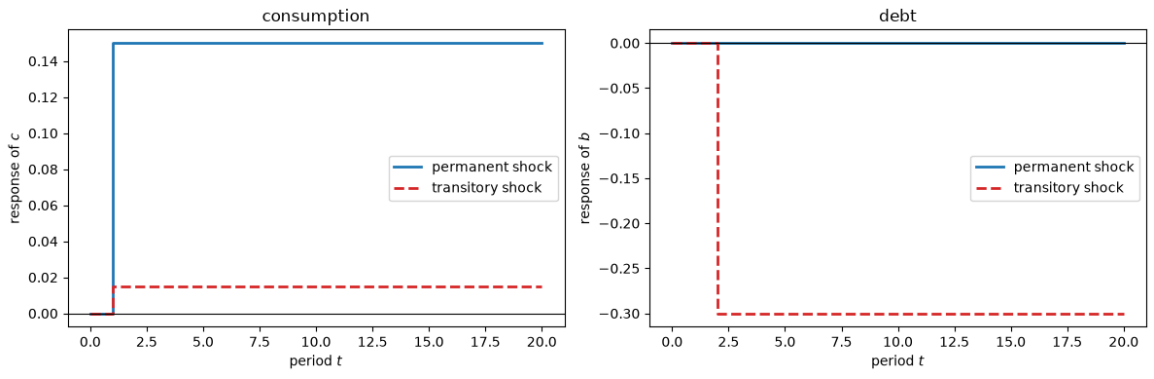
1. Using representation (80.11), compute and plot the paths of consumption c_t and debt b_t following each shock.
2. Confirm that the permanent shock raises consumption permanently by σ_1 and induces no saving, while the transitory shock raises consumption permanently by $(1 - \beta)\sigma_2$ and is otherwise saved.

Solution

Here is one solution:

We iterate representation (80.11) forward with a single shock at $t = 1$.

```
def impulse_response(shock, T=20):
    """Paths of c and b after a single shock at t=1 (z_0 = b_0 = 0)."""
    I2 = np.eye(2)
    b_coef = M @ (A_check - I2)      # coefficient on z_t in the b law

    z = np.zeros((T + 1, 2))
    b = np.zeros(T + 1)

    z[1] = C_check @ shock           # shock realized at t = 1
    for t in range(1, T):
        z[t + 1] = A_check @ z[t]
    for t in range(T):
        b[t + 1] = b[t] + (b_coef @ z[t]).item()

    c = np.array([((1 - β) * (M @ z[t] - b[t])).item() for t in range(T + 1)])
    return c, b


c_perm, b_perm = impulse_response(np.array([1.0, 0.0]))
c_tran, b_tran = impulse_response(np.array([0.0, 1.0]))

fig, axes = plt.subplots(1, 2, figsize=(12, 4))

axes[0].step(range(len(c_perm)), c_perm, where='post',
             label='permanent shock', color='C0', lw=2)
axes[0].step(range(len(c_tran)), c_tran, where='post',
             label='transitory shock', color='C3', linestyle='--', lw=2)
axes[0].axhline(0, color='k', lw=0.8)
axes[0].set_xlabel('period $t$')
axes[0].set_ylabel('response of $c$')
axes[0].set_title('consumption')
axes[0].legend()

axes[1].step(range(len(b_perm)), b_perm, where='post',
             label='permanent shock', color='C0', lw=2)
axes[1].step(range(len(b_tran)), b_tran, where='post',
             label='transitory shock', color='C3', linestyle='--', lw=2)
axes[1].axhline(0, color='k', lw=0.8)
axes[1].set_xlabel('period $t$')
axes[1].set_ylabel('response of $b$')
axes[1].set_title('debt')
axes[1].legend()
fig.tight_layout()
plt.show()

print(f"permanent shock:  Δc = {c_perm[-1]:.4f}  (theory σ1 = {σ1:.4f}),"
      f"  Δb = {b_perm[-1]:.4f}")
print(f"transitory shock: Δc = {c_tran[-1]:.4f}  "
      f"(theory (1-β)σ2 = {(1-β)*σ2:.4f}),  Δb = {b_tran[-1]:.4f}")
```




```
permanent shock:  Δc = 0.1500  (theory σ1 = 0.1500),  Δb = 0.0000
transitory shock: Δc = 0.0150  (theory (1-β)σ2 = 0.0150),  Δb = -0.3000
```

The permanent shock lifts consumption by σ_1 and leaves debt at zero: the shock is fully capitalised, so there is no net saving.

The transitory shock lifts consumption by only $(1-\beta)\sigma_2$; the consumer saves the remainder, so debt falls to $-\sigma_2$ (assets accumulate).

Exercise 80.3.2

This exercise illustrates the **cointegration** of consumption and debt described in (80.17).

The cointegration result requires a *stationary* endowment, so we replace the two-factor process with a scalar AR(1),

$$z_{t+1} = \rho z_t + \sigma_\varepsilon w_{t+1}, \qquad y_t = z_t,$$

with $\rho = 0.7$ and $\sigma_\varepsilon = 0.5$.

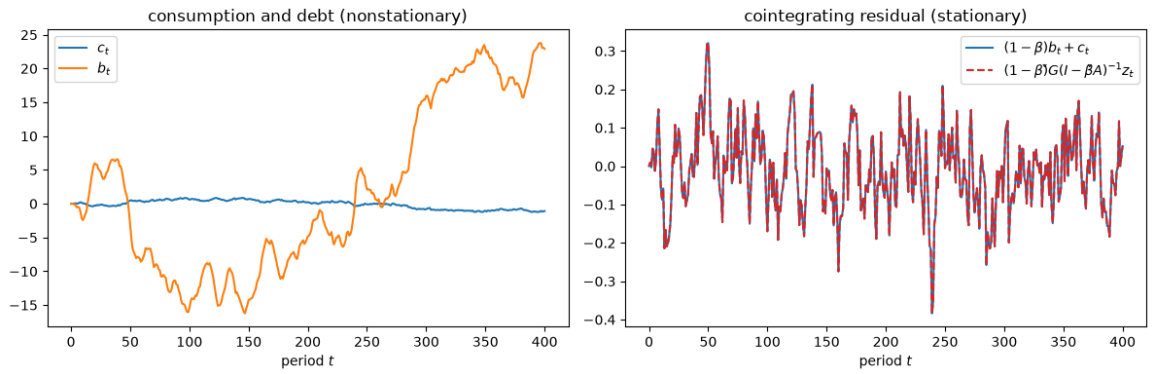
1. Simulate a long path of c_t and b_t using representation (80.11).
2. Verify that c_t and b_t each inherit a unit root (they wander), while the cointegrating residual $(1-\beta)b_t + c_t$ is stationary and equals $(1-\beta)\check{G}(I-\beta\check{A})^{-1}z_t$.

Solution

Here is one solution:

```
ρ, σε = 0.7, 0.5
A_ar = np.array([[ρ]])
C_ar = np.array([[σε]])
G_ar = np.array([[1.0]])

M_ar = G_ar @ inv(np.eye(1) - β * A_ar)      # G(I - βA)^{-1}
b_coef_ar = M_ar @ (A_ar - np.eye(1))        # coefficient on z_t in b law

rng = np.random.default_rng(0)
T = 400
z = np.zeros((T + 1, 1))
b = np.zeros(T + 1)
for t in range(T):
    z[t + 1] = A_ar @ z[t] + C_ar @ rng.standard_normal(1)
```

```
    b[t + 1] = b[t] + (b_coef_ar @ z[t]).item()

c = np.array([((1 - β) * (M_ar @ z[t] - b[t])).item() for t in range(T + 1)])
residual = (1 - β) * b + c
theory = np.array([((1 - β) * (M_ar @ z[t])).item() for t in range(T + 1)])

fig, axes = plt.subplots(1, 2, figsize=(12, 4))
axes[0].plot(c, label='$c_t$', lw=1.5)
axes[0].plot(b, label='$b_t$', lw=1.5)
axes[0].set_xlabel('period $t$')
axes[0].set_title('consumption and debt (nonstationary)')
axes[0].legend()

axes[1].plot(residual, label=r'$(1-\beta)b_t + c_t$', lw=1.5, color='C0')
axes[1].plot(theory, label=r'$(1-\beta)\check{G}(I-\beta\check{A})^{-1}z_t$',
             lw=1.5, linestyle='--', color='C3')
axes[1].set_xlabel('period $t$')
axes[1].set_title('cointegrating residual (stationary)')
axes[1].legend()
fig.tight_layout()
plt.show()

print(f"max |residual - theory| = {np.max(np.abs(residual - theory)):.2e}")
```




```
max |residual - theory| = 2.36e-16
```

Both c_t and b_t inherit the unit root that the random-walk consumption rule builds into the solution, so they drift without settling down.

Their linear combination $(1-\beta)b_t + c_t$ coincides with $(1-\beta)\check{G}(I-\beta\check{A})^{-1}z_t$ (up to floating-point error), which is a stationary function of the stationary state z_t.

Exercise 80.3.3

This exercise explores how the consumer's information problem shapes his response to income surprises in the Muth model.

Recall from (80.24) that the permanent consumption response to a unit income innovation a_{t+1} is $1-\beta(1-K)$, where the Kalman gain K in (80.22) depends on the ratio σ_1/σ_2.

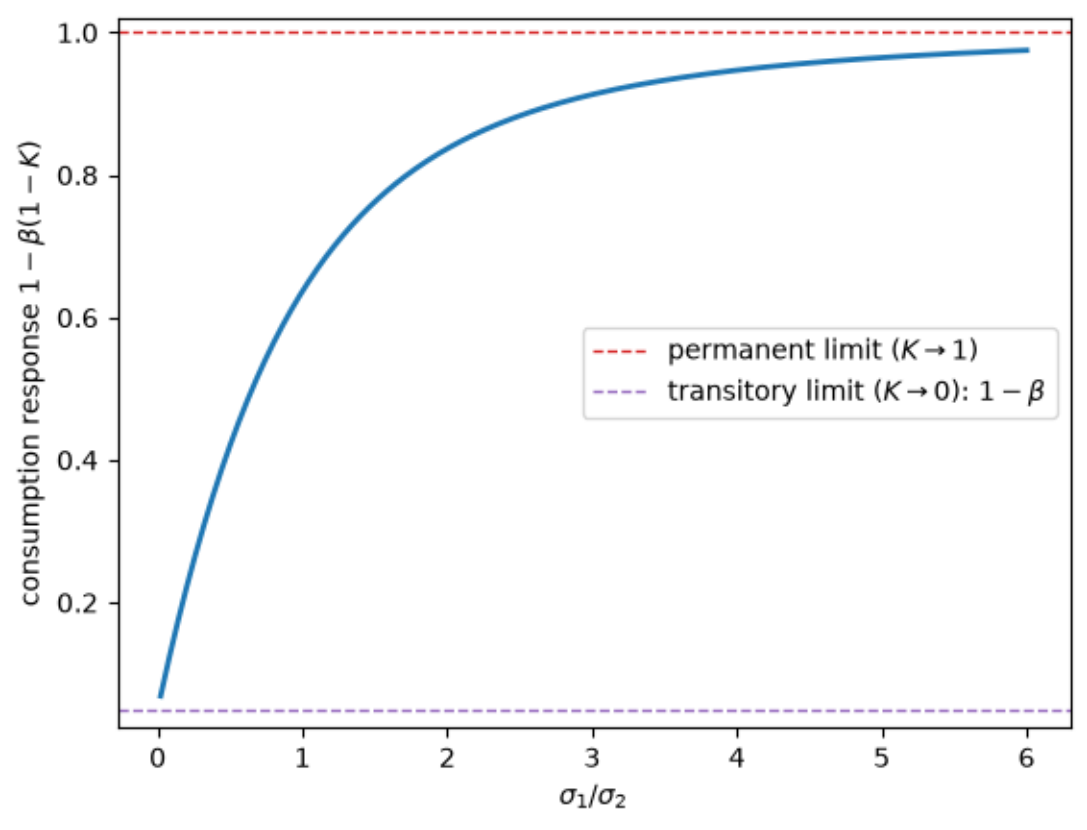
1. Plot the response $1-\beta(1-K)$ as a function of the ratio σ_1/σ_2, holding σ_2 fixed.
2. Explain the two limiting cases $\sigma_1/\sigma_2 \to 0$ and $\sigma_1/\sigma_2 \to \infty$, and relate them to the two shocks of the full-

information model.

Solution

Here is one solution:

```
ratios = np.linspace(0.02, 6.0, 300)
σ2_fixed = 0.30

responses = []
for r in ratios:
    s1 = r * σ2_fixed
    Σ = (s1**2 + np.sqrt(s1**4 + 4 * s1**2 * σ2_fixed**2)) / 2
    K = Σ / (Σ + σ2_fixed**2)
    responses.append(1 - β * (1 - K))

fig, ax = plt.subplots()
ax.plot(ratios, responses, lw=2, color='C0')
ax.axhline(1.0, color='C3', linestyle='--', lw=1,
           label=r'permanent limit ($K\to1$)')
ax.axhline(1 - β, color='C4', linestyle='--', lw=1,
           label=r'transitory limit ($K\to0$): $1-\beta$')
ax.set_xlabel(r'$\sigma_1/\sigma_2$')
ax.set_ylabel(r'consumption response $1-\beta(1-K)$')
ax.legend()
plt.show()
```



As $\sigma_1/\sigma_2 \to 0$ the endowment is dominated by transitory noise, so $K \to 0$: the consumer treats each innovation as transitory and raises consumption by only $1-\beta$, exactly as for the purely transitory shock in the full-information model.

As $\sigma_1/\sigma_2 \to \infty$ the endowment is dominated by permanent shocks, so $K \to 1$: the consumer treats each innovation as permanent and raises consumption one-for-one, exactly as for the permanent shock.

For intermediate ratios the consumer optimally splits each surprise, capitalising the fraction K he attributes to the permanent component and saving the rest.

CHAPTER

EIGHTYONE

CONSUMPTION SMOOTHING WITH INCOMPLETE AND COMPLETE MARKETS

Contents

- *Consumption Smoothing with Incomplete and Complete Markets*
  - *Overview*
  - *A brief review*
  - *Spreading consumption cross sections*
  - *A borrowers and lenders economy*
  - *Consumption smoothing with complete markets*
  - *Exercises*

81.1 Overview

This lecture studies how the cross-section distribution of consumption evolves when many consumers each solve the LQ permanent income problem.

It is the second of three lectures on the LQ permanent income model and builds directly on *The LQ Permanent Income Model*.

We first show that the unit root in individual consumption causes the cross-section variance of consumption to grow linearly with time.

We then embed the individual consumer in a **closed economy** with a continuum of consumers, following Bewley [1977], and show how the gross interest rate $R = \beta^{-1}$ emerges as an equilibrium outcome.

Finally, we replace the single risk-free bond with a complete set of Arrow securities and show how complete markets deliver a time-invariant cross-section distribution of consumption.

The third lecture, *Robust Consumption Smoothing and Precautionary Savings*, relaxes the assumption that the consumer fully trusts his income model.

Let's begin with some imports.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.linalg import inv
```

81.2 A brief review

We recall the essentials from *The LQ Permanent Income Model*.

A consumer with quadratic utility, discount factor β, and access to a risk-free bond with gross return $R = \beta^{-1}$ faces the endowment process

$$\begin{aligned} z_{t+1} &= \check{A}\, z_t + \check{C}\, w_{t+1} \\ y_t &= \check{G}\, z_t \end{aligned} \tag{81.1}$$

where w_{t+1} is IID with mean zero and identity covariance matrix.

The optimal consumption function expresses consumption as $r/(1+r) = (1-\beta)$ times total wealth,

$$c_t = (1-\beta)\left[\check{G}(I - \beta\check{A})^{-1} z_t - b_t\right] \tag{81.2}$$

The model has a state-space representation in which the state is current consumption c_t and the exogenous endowment state z_t:

$$\begin{aligned} c_{t+1} &= c_t + (1-\beta)\,\check{G}(I - \beta\check{A})^{-1}\check{C}\, w_{t+1} \\ b_t &= \check{G}(I - \beta\check{A})^{-1} z_t - \frac{1}{1-\beta}\, c_t \\ y_t &= \check{G}\, z_t \\ z_{t+1} &= \check{A}\, z_t + \check{C}\, w_{t+1} \end{aligned} \tag{81.3}$$

Consumption is a **random walk**: its first difference is the IID innovation $h\, w_{t+1}$, where $h = (1-\beta)\,\check{G}(I - \beta\check{A})^{-1}\check{C}$.

Throughout we use the two-factor endowment $y_t = z_{1t} + z_{2t}$ from *The LQ Permanent Income Model*, with z_{1t} a permanent component and z_{2t} a purely transitory component, so that $\check{A} = \mathrm{diag}(1, 0)$ and $\check{C} = \mathrm{diag}(\sigma_1, \sigma_2)$.

The following cell reproduces the calibration and the key matrices.

```
# Parameters
β = 0.95       # discount factor (so R = 1/β)
σ1 = 0.15      # std of permanent shock
σ2 = 0.30      # std of transitory shock

# Two-factor endowment
A_check = np.array([[1.0, 0.0],
                    [0.0, 0.0]])
C_check = np.array([[σ1, 0.0],
                    [0.0, σ2]])
G_check = np.array([[1.0, 1.0]])

# Key matrix M = G(I - βA)^{-1}
IbA = np.eye(2) - β * A_check
M = G_check @ inv(IbA)    # shape (1, 2)
```

81.3 Spreading consumption cross sections

The unit root in consumption (representation (81.3)) causes a **cross-section variance** of consumption to grow linearly with time.

Consider a continuum of *ex ante* identical households born at $t = 0$.

All households i share the same preferences.

They all face a stochastic process for non-financial income of the same form

$$y_t^i = g(w_t^i, w_{t-1}^i, \ldots, w_0^i; z_0^i), \tag{81.4}$$

While all consumers face the same g process, they have different, statistically independent realizations of the **idiosyncratic** shock sequences $\{w_t^i\}_{t=0}^{\infty}$.

Let all households start from the same initial conditions $c_0^i = c_0$ and z_0^i.

From (81.3), household i's consumption follows

$$c_{t+1}^i = c_t^i + h\, w_{t+1}^i, \qquad h = (1-\beta)\,\check{G}(I - \beta\check{A})^{-1}\check{C}$$

Since $\{w_t^i\}$ realizations are independent across agents,

$$\mathbb{E}_0(c_t^i - c_0^i)^2 = t\, hh^\top \tag{81.5}$$

In the two-factor model, h is a 1×2 row vector so $hh^\top$ is a positive scalar equal to $\sigma_1^2 + (1-\beta)^2\sigma_2^2$.

The cross-section variance of consumption grows like t.

```
# Simulate cross-section spreading
rng = np.random.default_rng(42)
N = 5000        # number of agents
T_sim = 80          # number of periods

h_vec = (1 - β) * (M @ C_check)    # shape (1, 2), then flatten
h_vec = h_vec.flatten()             # h = [h1, h2]

c = np.zeros((N, T_sim + 1))        # consumption paths
# initialise all agents at c_0 = 0 (demeaned)
for t in range(T_sim):
    eps = rng.standard_normal((N, 2))     # N draws of 2D shock
    dc = eps @ h_vec                # shape (N,)
    c[:, t+1] = c[:, t] + dc

# Cross-section variance at each date
var_c = np.var(c, axis=0)
theory = np.arange(T_sim + 1) * np.dot(h_vec, h_vec)

fig, ax = plt.subplots()
ax.plot(var_c, label='simulated cross-section variance', lw=2)
ax.plot(theory, label=r'theoretical: $t \cdot h h^\top$',
        linestyle='--', color='C3', lw=2)
ax.set_xlabel('period $t$')
ax.set_ylabel('cross-section variance of $c$')
ax.legend()
plt.show()
```

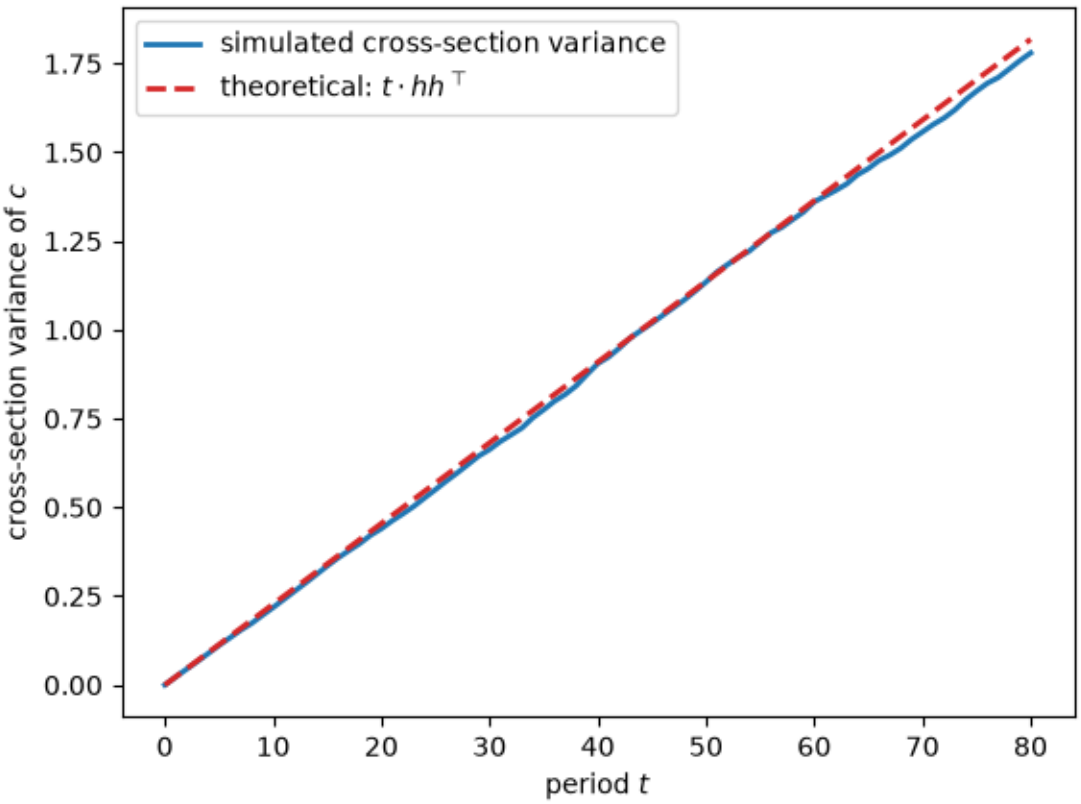


Fig. 81.1: Spreading consumption cross sections

81.4 A borrowers and lenders economy

Up to now we have set $R = \beta^{-1}$ and taken it as determined outside the model ("small open economy").

Following ideas of Bewley [1977], we can construct a **closed economy** in which $R = \beta^{-1}$ is an **equilibrium outcome**.

A continuum of measure one of consumers, indexed by $i \in [0, 1]$, trade a risk-free one-period bond with price β.

All consumers have the same preferences and the same stochastic income process (81.4), but face **idiosyncratic** non-financial income shock process realizations.

Initial bond positions are zero: $b_0^i = 0$ for all i.

Initial endowment states z_0^i are independent draws from a common initial distribution.

Because the permanent component z_{1t} has a unit root, process (81.1) has no stationary distribution, so in the simulation below we draw the permanent component $z_{10}^i \sim N(0, 1)$ and draw the transitory component $z_{20}^i \sim N(0, \sigma_2^2)$ from its stationary distribution.

From (81.2), with $b_0^i = 0$, agent i's time-0 consumption is

$$c_0^i = (1 - \beta)\, \check{G}(I - \beta \check{A})^{-1} z_0^i \tag{81.6}$$

For $t \geq 1$, from (81.3):

$$c_{t+1}^i = c_t^i + h\, w_{t+1}^i, \qquad h = (1 - \beta)\, \check{G}(I - \beta \check{A})^{-1} \check{C} \tag{81.7}$$

Let Y denote the stationary mean of the cross-section average of non-financial income.

Integrating (81.6) over all agents:

$$\int_0^1 c_0^i \, di = (1 - \beta) \sum_{j=0}^{\infty} \beta^j \mathbb{E}_0 \int_0^1 y_j^i \, di = Y \tag{81.8}$$

because the continuum of idiosyncratic shocks averages to zero.

For future periods, integrating (81.7):

$$\int_0^1 c_{t+1}^i \, di = \int_0^1 c_t^i \, di + h \underbrace{\int_0^1 w_{t+1}^i \, di}_{=0} = Y$$

The goods market clears at every date at *constant* aggregate consumption equal to Y.

The bond market clears at zero net supply each period.

Thus $R = \beta^{-1}$ is an equilibrium outcome.

While the cross-section mean of consumption is constant, the cross-section *variance* grows without bound according to (81.5).

Initial differences in endowment draws z_0^i create permanent differences in consumption levels.

```
# Verify Bewley market clearing via simulation
# Online mean and variance avoid storing all paths.
rng = np.random.default_rng(0)
N_bew = 10000    # number of agents
T_bew = 60

# Draw initial states for the simulation.
```

(continues on next page)

(continued from previous page)

```
z0_i = rng.standard_normal((N_bew, 2)) * np.array([1.0, σ2])
c0_i = ((1 - β) * (M @ z0_i.T)).flatten()   # shape (N_bew,)

# Propagate consumption across agents.
mean_c = np.zeros(T_bew + 1)
var_c2 = np.zeros(T_bew + 1)
mean_c[0] = c0_i.mean()
var_c2[0] = c0_i.var()

c_now = c0_i.copy()
for t in range(T_bew):
    eps = rng.standard_normal((N_bew, 2))
    c_now = c_now + eps @ h_vec
    mean_c[t + 1] = c_now.mean()
    var_c2[t + 1] = c_now.var()

# Reuse initial consumption below.
c_bew_t0 = c0_i

fig, axes = plt.subplots(1, 2, figsize=(12, 4))

axes[0].plot(mean_c, lw=2, color='C0')
axes[0].axhline(mean_c[0], linestyle='--', color='C3', label='initial mean')
axes[0].set_xlabel('period $t$')
axes[0].set_ylabel('mean consumption')
axes[0].set_title('mean')
axes[0].legend()

axes[1].plot(var_c2, lw=2, color='C0', label='simulated variance')
axes[1].set_xlabel('period $t$')
axes[1].set_ylabel('variance of consumption')
axes[1].set_title('variance')
axes[1].legend()
fig.tight_layout()
plt.show()
```

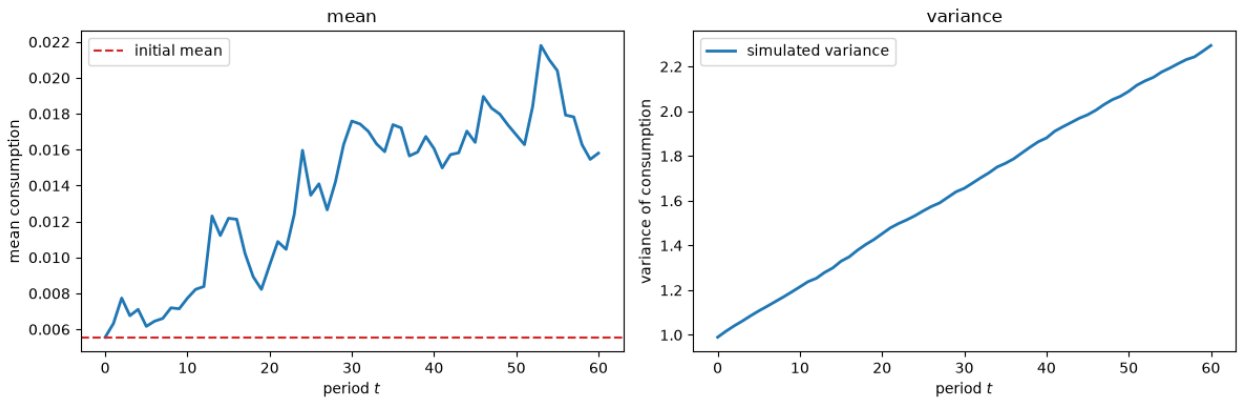


Fig. 81.2: Bewley economy cross-section moments

Because each consumer dislikes variation of consumption over time, each consumer would prefer a completely smoothed stream $c_t^i = c_0^i$ for all t.

Such an allocation is feasible because the cross-section average of income is constant.

The next section describes a complete-markets allocation that supports this outcome.

81.5 Consumption smoothing with complete markets

We replace the single bond with a **complete set of Arrow securities**.

The budget constraint becomes

$$c_t + b_{t-1}(z_t) = \int q(z_{t+1}|z_t)\, b_t(z_{t+1})\, dz_{t+1} + y_t \tag{81.9}$$

where $q(z_{t+1}|z_t)$ is the pricing kernel for one-period state-contingent claims and $b_t(z_{t+1})$ is the household's portfolio of Arrow securities chosen at t.

We guess (and verify) that the equilibrium pricing kernel is

$$q(z_{t+1}|z_t) = \beta\, \phi(z_{t+1}|z_t) \tag{81.10}$$

where $\phi(z_{t+1}|z_t)$ is the transition density of z.

This kernel prices a one-period risk-free bond at β, so $R = \beta^{-1}$, consistent with the incomplete-markets equilibrium.

We conjecture that the equilibrium delivers each consumer i a *constant* consumption level:

$$c_t^i = \bar{c}^i = c_0^i, \quad \forall\, t \geq 0 \tag{81.11}$$

where $c_0^i = (1-\beta)\, \check{G}(I - \beta \check{A})^{-1} z_0^i$ is the consumer's time-0 consumption in the incomplete-markets economy.

The state-contingent debt that supports constant consumption is

$$b_{t-1}(z_t) = \check{G}(I - \beta \check{A})^{-1} z_t - \frac{1}{1-\beta}\, \bar{c}^i \;\equiv\; b(z_t, \bar{c}^i) \tag{81.12}$$

Note that indebtedness depends only on the current Markov state z_t, *not* on the history of earlier states.

This absence of history dependence reflects the **complete risk sharing** attained under complete markets.

Substituting the pricing kernel (81.10) and the portfolio conjecture (81.12) into the budget constraint (81.9) and using the law of iterated expectations confirms that the budget constraint simplifies to $c_t = \bar{c}^i$ in every state and period.

Under complete markets, the cross-section distribution of consumption is **time-invariant**.

Consumer i's rank in the consumption distribution is fixed forever.

A lucky initial draw z_0^i manifests itself as perpetually high consumption $\bar{c}^i$ and lower indebtedness $b(z_t^i, \bar{c}^i)$ across all future states.

This outcome contrasts with what happens in the incomplete-markets Bewley economy, where the cross-section variance of consumption grows without bound.

```
# Complete and incomplete consumption distributions
rng = np.random.default_rng(1)
N_cm = 5000
T_cm = 50

# Initial consumption draws (same as Bewley economy)
c0_cm = c_bew_t0[:N_cm]
```

(continues on next page)

(continued from previous page)

```
# Incomplete markets: consumption evolves (random walk)
c_inc = np.zeros((N_cm, T_cm + 1))
c_inc[:, 0] = c0_cm
for t in range(T_cm):
    eps = rng.standard_normal((N_cm, 2))
    c_inc[:, t+1] = c_inc[:, t] + eps @ h_vec

# Complete markets: consumption stays constant
c_comp = np.tile(c0_cm[:, np.newaxis], T_cm + 1)

fig, axes = plt.subplots(1, 2, figsize=(12, 4))

for t_plot, color in zip([0, 10, 30, 50], ['C0', 'C1', 'C3', 'C4']):
    axes[0].hist(c_inc[:, t_plot], bins=60, alpha=0.4,
                 label=f't = {t_plot}', color=color, density=True)
axes[0].set_xlabel('$c$')
axes[0].set_ylabel('density, incomplete markets')
axes[0].set_title('incomplete markets')
axes[0].legend(fontsize=9)

for t_plot, color in zip([0, 10, 30, 50], ['C0', 'C1', 'C3', 'C4']):
    axes[1].hist(c_comp[:, t_plot], bins=60, alpha=0.4,
                 label=f't = {t_plot}', color=color, density=True)
axes[1].set_xlabel('$c$')
axes[1].set_ylabel('density, complete markets')
axes[1].set_title('complete markets')
axes[1].legend(fontsize=9)
fig.tight_layout()
plt.show()
```

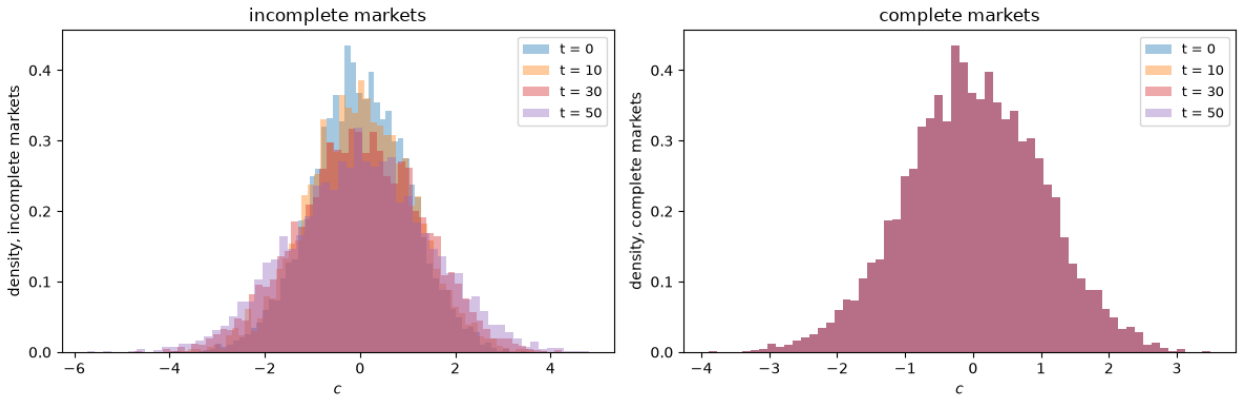


Fig. 81.3: Cross section distributions with incomplete and complete markets

Note

Under **complete markets** the histogram stays the same across all t (distributions coincide perfectly), while under **incomplete markets** distributions spread out over time.

So far the consumer fully trusts his stochastic income model.

In *Robust Consumption Smoothing and Precautionary Savings* we relax that assumption and let the consumer seek decision rules that are robust to plausible misspecifications.

The optimal robust rule takes the same form as the rule above, but under a distorted model of the income process that looks more persistent than the approximating one.

81.6 Exercises

Exercise 81.6.1

This exercise studies how patience governs the speed at which the cross-section of consumption spreads out.

From (81.5), the cross-section variance of consumption grows by $hh^\top = \sigma_1^2 + (1-\beta)^2\sigma_2^2$ per period.

1. Compute this per-period growth rate for $\beta \in \{0.90, 0.95, 0.99\}$ and report the permanent and transitory contributions separately.
2. Confirm by simulation that the cross-section variance grows linearly at the predicted rate for $\beta = 0.95$.
3. Explain why increasing β slows the spreading due to transitory shocks but leaves the contribution of permanent shocks unchanged.

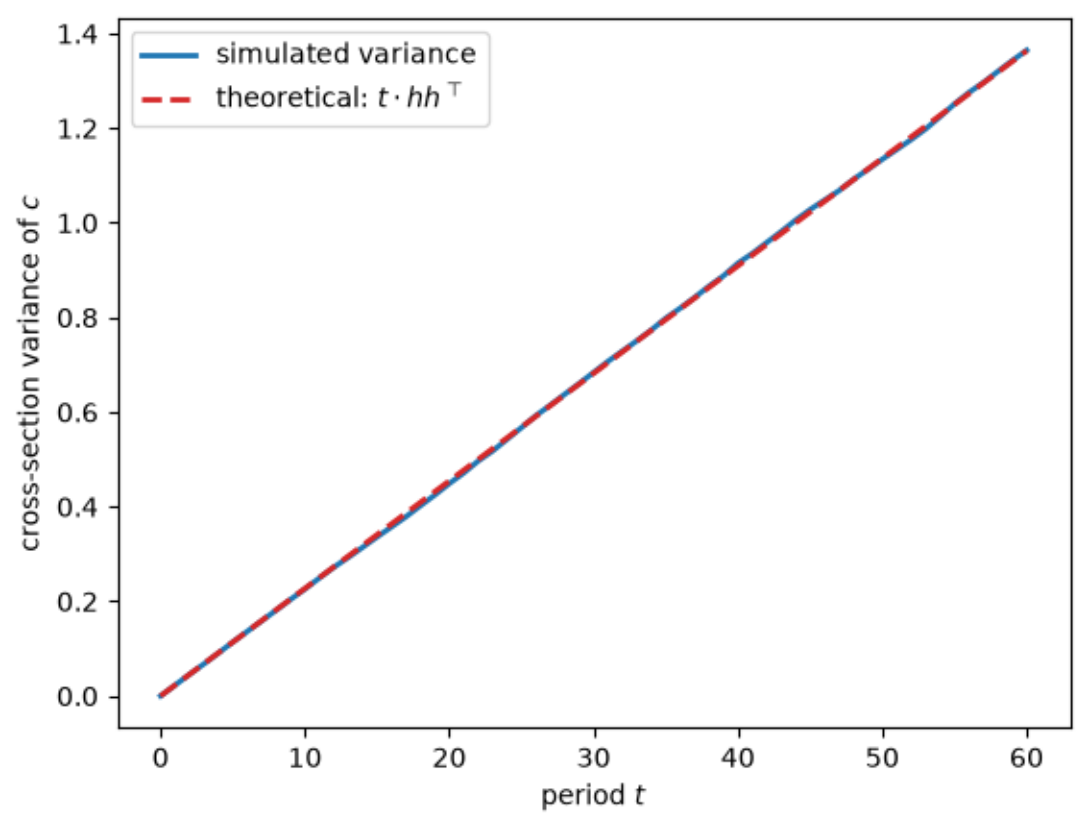
Solution

Here is one solution:

```
for b in (0.90, 0.95, 0.99):
    perm = σ1**2
    tran = (1 - b)**2 * σ2**2
    print(f"β = {b}:  growth = {perm + tran:.5f}  "
          f"(permanent {perm:.5f}, transitory {tran:.5f})")
```

```
β = 0.9:  growth = 0.02340  (permanent 0.02250, transitory 0.00090)
β = 0.95:  growth = 0.02272  (permanent 0.02250, transitory 0.00023)
β = 0.99:  growth = 0.02251  (permanent 0.02250, transitory 0.00001)
```

```
rng = np.random.default_rng(7)
N, T_sim = 20000, 60
hh = float(h_vec @ h_vec)

c = np.zeros((N, T_sim + 1))
for t in range(T_sim):
    c[:, t + 1] = c[:, t] + rng.standard_normal((N, 2)) @ h_vec

var_c = c.var(axis=0)
theory = np.arange(T_sim + 1) * hh

fig, ax = plt.subplots()
ax.plot(var_c, lw=2, label='simulated variance')
ax.plot(theory, lw=2, linestyle='--', color='C3',
        label=r'theoretical: $t\cdot h h^\top$')
ax.set_xlabel('period $t$')
ax.set_ylabel('cross-section variance of $c$')
ax.legend()
plt.show()
```



The permanent contribution σ_1^2 does not depend on β because a permanent shock is capitalised one-for-one into consumption regardless of patience.

The transitory contribution $(1-\beta)^2\sigma_2^2$ shrinks as $\beta \to 1$ because a more patient consumer smooths a transitory shock over a longer horizon, moving consumption by only the small annuity value $(1-\beta)$.

Exercise 81.6.2

This exercise contrasts the cross-section variance of consumption under incomplete and complete markets.

Start all consumers from the initial consumption draws `c_bew_t0` computed above.

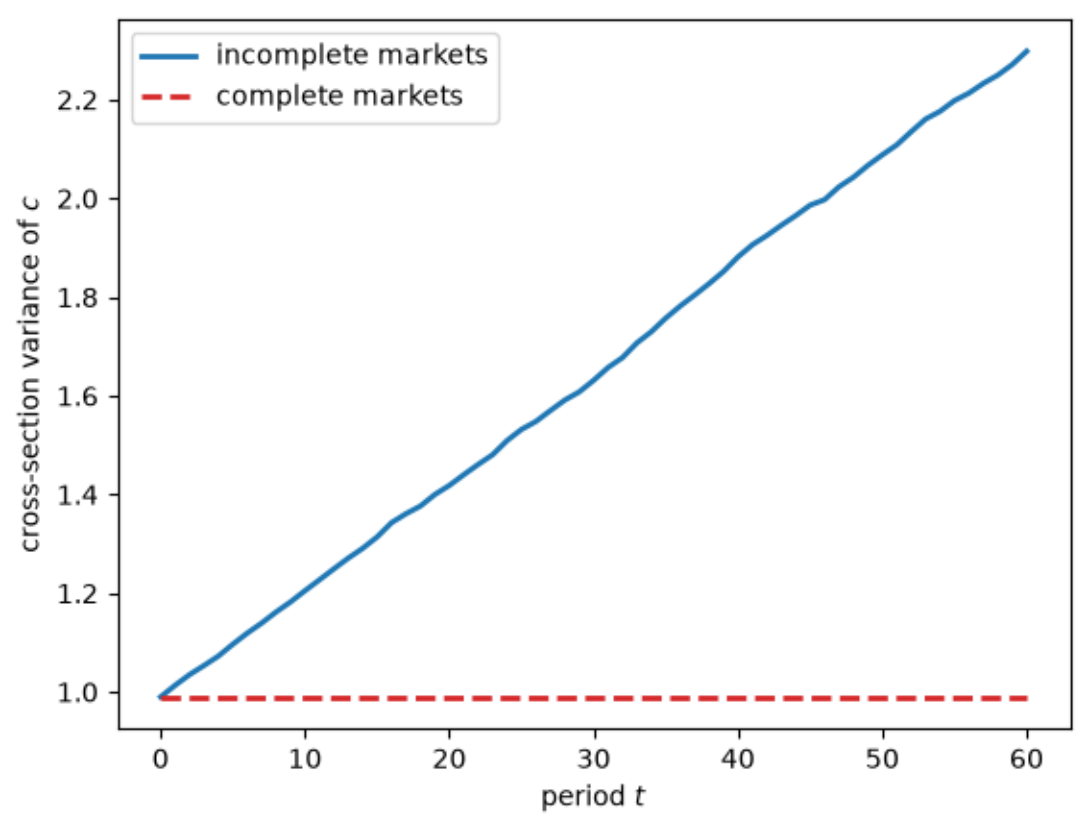
1. Compute the cross-section variance at each date $t = 0, 1, \ldots, 60$ under incomplete markets (consumption is a random walk) and under complete markets (consumption is constant).
2. Plot the two variance paths on the same axes and comment on the difference.

Solution

Here is one solution:

```
rng = np.random.default_rng(11)
T = 60
c0 = c_bew_t0

# Incomplete markets: random-walk consumption
```

```
c_inc = np.zeros((len(c0), T + 1))
c_inc[:, 0] = c0
for t in range(T):
    c_inc[:, t + 1] = c_inc[:, t] + rng.standard_normal((len(c0), 2)) @ h_vec

var_inc = c_inc.var(axis=0)
var_comp = np.full(T + 1, c0.var())   # complete markets: constant consumption

fig, ax = plt.subplots()
ax.plot(var_inc, lw=2, label='incomplete markets')
ax.plot(var_comp, lw=2, linestyle='--', color='C3', label='complete markets')
ax.set_xlabel('period $t$')
ax.set_ylabel('cross-section variance of $c$')
ax.legend()
plt.show()
```



Under incomplete markets the variance rises linearly without bound: each consumer accumulates an independent random walk of consumption innovations.

Under complete markets the variance is flat: each consumer locks in a constant consumption level $\bar{c}^i$, so the cross-section distribution never changes.

Exercise 81.6.3

This exercise shows how complete markets freeze each consumer's *rank* in the consumption distribution.

Under complete markets consumption is constant, so a consumer's position in the cross-section distribution never

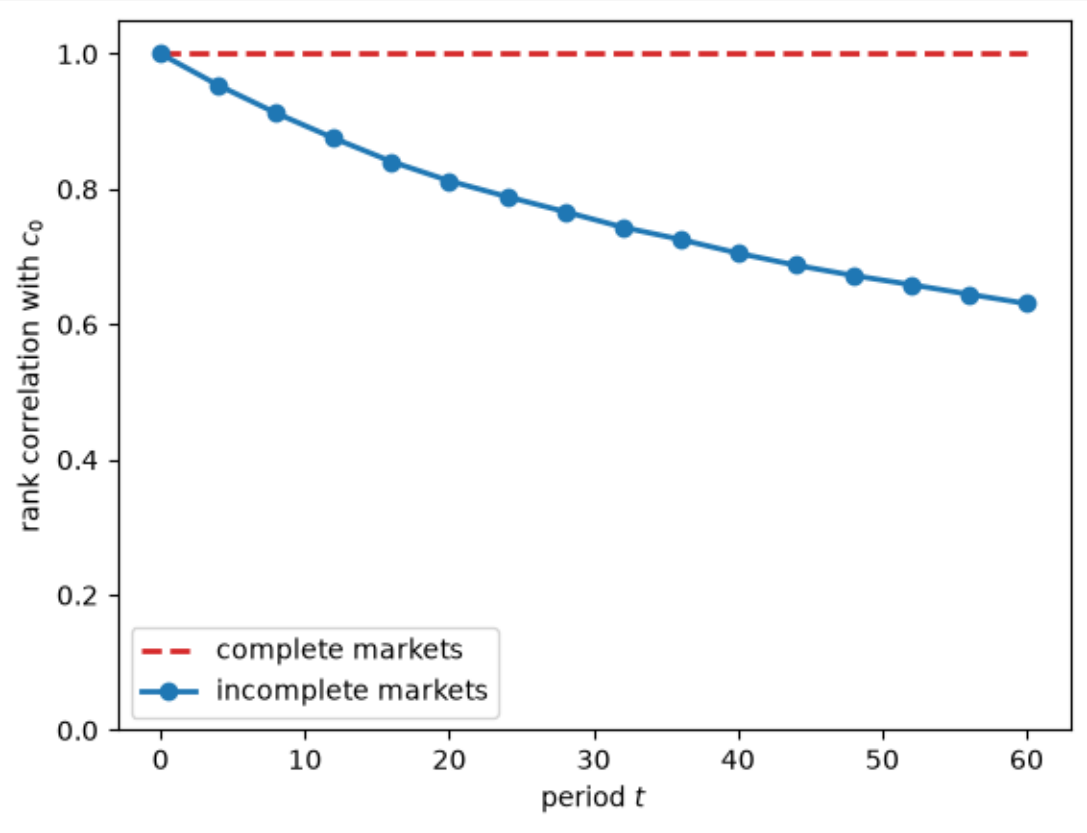
changes; under incomplete markets the random walk gradually scrambles ranks.

1. Simulate consumption paths under both market structures, starting from `c_bew_t0`.
2. Compute the Spearman rank correlation between consumption at $t = 0$ and at later dates t under each market structure, and plot it against t.
3. Interpret the result.

Solution

Here is one solution:

```
from scipy.stats import spearmanr

rng = np.random.default_rng(3)
T = 60
c0 = c_bew_t0

c_inc = np.zeros((len(c0), T + 1))
c_inc[:, 0] = c0
for t in range(T):
    c_inc[:, t + 1] = c_inc[:, t] + rng.standard_normal((len(c0), 2)) @ h_vec

dates = np.arange(0, T + 1, 4)
rank_inc = [spearmanr(c_inc[:, 0], c_inc[:, t]).statistic for t in dates]
rank_comp = [1.0 for _ in dates]   # constant consumption ⇒ ranks fixed

fig, ax = plt.subplots()
ax.plot(dates, rank_comp, lw=2, linestyle='--', color='C3',
        label='complete markets')
ax.plot(dates, rank_inc, lw=2, marker='o', color='C0',
        label='incomplete markets')
ax.set_xlabel('period $t$')
ax.set_ylabel(r'rank correlation with $c_0$')
ax.set_ylim(0, 1.05)
ax.legend()
plt.show()
```



Under complete markets the rank correlation stays exactly one: each consumer keeps the rank determined by the initial draw z_0^i forever.

Under incomplete markets the rank correlation decays toward zero as the accumulated random walk reshuffles who is rich and who is poor, even though every consumer faces the same income process.

CHAPTER

EIGHTYTWO

ROBUST CONSUMPTION SMOOTHING AND PRECAUTIONARY SAVINGS

Contents

- *Robust Consumption Smoothing and Precautionary Savings*
  - *Overview*
  - *A brief review*
  - *A robust permanent income model*
  - *Exercises*

82.1 Overview

This lecture studies a robust version of the LQ permanent income model due to Hansen *et al.* [1999] and Hansen and Sargent [2008].

It is the third of three lectures on the LQ permanent income model.

It builds on *The LQ Permanent Income Model*, which develops the standard model, and *Consumption Smoothing with Incomplete and Complete Markets*, which studies its cross-section and market-structure implications.

A consumer who distrusts his specification of the labor income process engages in a form of precautionary savings.

Our description of the model with concerns about robustness includes

- how (for quantities) a concern for robustness is observationally equivalent to an increase in impatience
- how the worst-case model that the consumer uses to shape his decision rule distorts the baseline model's endowment process toward greater persistence
- a frequency-domain representation of the effects of concerns about misspecification of the endowment process
- a detection-error-probability characterization of the amount of model uncertainty

The lecture concludes by combining the Bewley economy of *Consumption Smoothing with Incomplete and Complete Markets* with the robustness machinery.

Using tools from Hansen and Sargent [2008], we show:

- how a continuum of consumers i who use identical decision rules can nevertheless differ in their robustness parameters $\sigma_i \leq 0$ and their discount factors β_i, provided that the pair (σ_i, β_i) lies on an observational-equivalence locus derived below
- how every such consumer chooses the **same consumption-saving rule** as a baseline plain-vanilla $(\sigma = 0, \beta)$ agent with no concerns about misspecification of the endowment process
- how the equilibrium interest rate $R = \beta^{-1}$ and all aggregate dynamics therefore coincide with those of a benchmark Bewley model
- how distinct (σ_i, β_i) agents act as if they have different subjective models of their non-financial income process

We first present the HST model in its general form, which includes physical capital and investment i_t.

When we return to the Bewley economy of *Consumption Smoothing with Incomplete and Complete Markets*, we specialise to a pure endowment economy with no capital, so investment plays no role there.

Let's begin with some imports.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm
```

82.2 A brief review

We recall the essentials from *The LQ Permanent Income Model* and *Consumption Smoothing with Incomplete and Complete Markets*.

A consumer with quadratic utility and discount factor β faces the endowment process

$$\begin{aligned} z_{t+1} &= \check{A}\, z_t + \check{C}\, w_{t+1} \\ y_t &= \check{G}\, z_t \end{aligned} \tag{82.1}$$

The optimal decision rule has a state-space representation in which the state is current consumption c_t and the exogenous endowment state z_t:

$$\begin{aligned} c_{t+1} &= c_t + (1-\beta)\, \check{G}(I - \beta \check{A})^{-1} \check{C}\, w_{t+1} \\ b_t &= \check{G}(I - \beta \check{A})^{-1} z_t - \frac{1}{1-\beta}\, c_t \\ y_t &= \check{G}\, z_t \\ z_{t+1} &= \check{A}\, z_t + \check{C}\, w_{t+1} \end{aligned} \tag{82.2}$$

We again use the two-factor endowment $y_t = z_{1t} + z_{2t}$,

$$\begin{pmatrix} z_{1,t+1} \\ z_{2,t+1} \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} z_{1t} \\ z_{2t} \end{pmatrix} + \begin{pmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{pmatrix} \begin{pmatrix} w_{1,t+1} \\ w_{2,t+1} \end{pmatrix} \tag{82.3}$$

with z_{1t} a permanent component and z_{2t} a purely transitory component.

The following cell fixes the calibration used below.

```
# Parameters (as in the preceding lectures)
β = 0.95        # discount factor
σ1 = 0.15       # std of permanent shock
σ2 = 0.30       # std of transitory shock
```

82.3 A robust permanent income model

82.3.1 Robustness and precautionary savings

We now study a consumer who *distrusts* his specification of the stochastic process governing his labor income.

The model is due to Hansen *et al.* [1999] (HST), who estimated it on US quarterly consumption and investment data.

For a fuller treatment of the HST model and its asset-pricing implications, see *Robust Permanent Income and Pricing*.

A consumer who fears model misspecification engages in a form of **precautionary savings** that is distinct from the usual precautionary motive (which requires a convex marginal utility).

Here, the precautionary motive arises because the consumer wants to protect against misspecification of the **conditional means** of income shocks, and it operates even with quadratic preferences.

HST showed an important **observational equivalence** result: for quantities (c_t, i_t) alone, a concern for robustness is indistinguishable from an increase in impatience (a decrease in β).

We develop this result carefully below.

82.3.2 The HST model

HST's model features a planner with preferences over consumption streams $\{c_t\}$, mediated through **service streams** $\{s_t\}$.

Let b be a preference shifter (utility bliss point).

The **Bellman equation for the robust planner** is

$$-x^\top P x - p = \sup_c \inf_w \Big\{ -(s-b)^2 + \beta(\theta (w^*)^\top w^* - \mathbb{E}\,(x^*)^\top P x^* - p) \Big\} \tag{82.4}$$

subject to the household technology, capital accumulation, endowment dynamics, and the state law:

$$\begin{aligned} s &= (1+\lambda)c - \lambda h \\ h^* &= \delta_h h + (1-\delta_h)c \\ k^* &= \delta_k k + i \\ c + i &= \gamma k + d \\ \begin{pmatrix} d \\ b \end{pmatrix} &= U z \\ z^* &= A_{22} z + C_2(\epsilon^* + w^*) \end{aligned} \tag{82.5}$$

Here * denotes the next-period value; c is consumption; s is the scalar service measure; h is a habit stock; k is the capital stock; i is investment; d is an endowment/technology shock; b is a **preference shock** (bliss-point shifter, distinct from the bond/debt variable b_t used above); $\epsilon^* \sim N(0, I)$ is the baseline shock; and w^* is a **distortion** to the conditional mean of ϵ^* chosen by a minimizing agent.

The penalty parameter $\theta > 0$ governs the consumer's concern about robustness.

We use the transformation

$$\sigma = -\theta^{-1} \leq 0$$

so $\sigma = 0$ corresponds to no robustness concern and $\sigma < 0$ to an increasing concern.

When $\lambda > 0$ and $\delta_h \in (0, 1)$, the technology (82.5) accommodates **habit persistence** (positive λ) or durability.

The stock h_t is a geometric weighted average of current and past consumption.

Equation $c_t + k_t = Rk_{t-1} + d_t$ with $R = \delta_k + \gamma$ combines capital accumulation with a linear production technology.

R is the physical gross return on capital.

Let $x_t^\top = [h_{t-1},\, k_{t-1},\, z_t^\top]$.

The state transition equations are:

$$x_{t+1} = A\, x_t + B\, u_t + C(\epsilon_{t+1} + w_{t+1}) \tag{82.6}$$

where $u_t = c_t$ and w_{t+1} is the distortion to the conditional mean of ϵ_{t+1}.

HST estimated the model on U.S. quarterly data (1970Q1-1996Q3) using nondurables plus services for consumption and durable consumption plus gross private investment for investment.

Key estimates are summarised in the following table (reported in Appendix A of HST):

Parameter	Habit	No Habit
Risk-free rate	0.025	0.025
β	0.997	0.997
δ_h	0.682	—
λ	2.443	0
α_1	0.813	0.900
α_2	0.189	0.241
ϕ_1	0.998	0.995
ϕ_2	0.704	0.450
$2 \times \log L$	779.05	762.55

HST imposed $\beta R = 1$ and $\delta_k = 0.975$, so γ is pinned down once β is estimated.

An annual real interest rate of 2.5% corresponds to $\beta = 0.997$.

82.3.3 Solution when $\sigma = 0$

When $\sigma = 0$ the objective reduces to

$$\mathbb{E}_0 \sum_{t=0}^{\infty} \beta^t \{-(s_t - b_t)^2\} \tag{82.7}$$

Formulating a Lagrangian and deriving first-order conditions yields:

$$\begin{aligned} \mu_{st} &= b_t - s_t \\ \mu_{ct} &= (1+\lambda)\mu_{st} + (1-\delta_h)\mu_{ht} \\ \mu_{ht} &= \beta \mathbb{E}_t[\delta_h \mu_{h,t+1} - \lambda \mu_{s,t+1}] \\ \mu_{ct} &= \beta R\, \mathbb{E}_t \mu_{c,t+1} \end{aligned} \tag{82.8}$$

Here μ_{st} is the **marginal valuation of consumption services**, which summarises the endogenous state variables h_{t-1} and k_{t-1}.

Equation (82.8) (last line) implies $\mathbb{E}_t \mu_{c,t+1} = (\beta R)^{-1} \mu_{ct}$, so μ_{st} is a martingale when $\beta R = 1$:

$$\mu_{st} = \mu_{s,t-1} + \nu^\top \epsilon_t \tag{82.9}$$

for some vector ν.

Solving forward and substituting gives

$$\mu_{st} = \Psi_1 k_{t-1} + \Psi_2 h_{t-1} + \Psi_3 \sum_{j=0}^{\infty} R^{-j} \mathbb{E}_t b_{t+j} + \Psi_4 \sum_{j=0}^{\infty} R^{-j} \mathbb{E}_t d_{t+j} \tag{82.10}$$

where

$$\Psi_1 = -(1+\lambda) R (1 - R^{-2}\beta^{-1}) \left[\frac{1 - R^{-1}\tilde{\delta}_h}{1 - R^{-1}\tilde{\delta}_h + \lambda(1 - \tilde{\delta}_h)} \right], \quad \Psi_4 = R^{-1}\Psi_1 \tag{82.11}$$

and $\tilde{\delta}_h = (\delta_h + \lambda)/(1+\lambda)$.

In the widely-studied special case $\lambda = \delta_h = 0$, so $s_t = c_t$ and $\mu_{st} = b_t - c_t$, the marginal propensity to consume out of **non-human wealth** Rk_{t-1} equals that out of **human wealth** $\sum_{j=0}^{\infty} R^{-j}\mathbb{E}_t d_{t+j}$, a well-known feature of the LQ model.

The formula for μ_{st} can be written as $\mu_{st} = M_s x_t$ where x_t follows (82.6).

It follows that

$$\nu^\top = M_s C, \qquad \alpha = \sqrt{\nu^\top \nu} = \sqrt{M_s C C^\top M_s^\top} \tag{82.12}$$

The scalar α plays a central role in the observational equivalence result below.

82.3.4 Observational equivalence

HST state an observational-equivalence theorem.

Theorem 82.3.1 (Observational Equivalence, I)

Fix all parameters except (σ, β) and suppose $\beta R = 1$ when $\sigma = 0$.

There exists $\underline{\sigma} < 0$ such that for any $\sigma \in (\underline{\sigma}, 0)$, the optimal consumption-investment plan for $(0, \beta)$ is also chosen by a robust decision maker with parameters $(\sigma, \hat{\beta}(\sigma))$, where

$$\hat{\beta}(\sigma) = \frac{1}{R} + \frac{\sigma\alpha^2}{R-1} \tag{82.13}$$

and $\hat{\beta}(\sigma) < \beta$.

Since $R > 1$ and $\alpha^2 > 0$, a more negative σ (stronger robustness concern) lowers $\hat{\beta}$.

A robust consumer wants to save more because his alter ego, a utility-minimizing agent, makes future income look worse than the approximating model predicts.

A lower discount factor makes a consumer less patient and therefore reduces saving.

When these two forces are balanced according to (82.13), consumption plans are identical across $(\sigma, \hat{\beta}(\sigma))$ pairs.

Proof. When $\beta R = 1$ and $\sigma = 0$, the marginal utility μ_{st} obeys the martingale

$$\mu_{st} = \mu_{s,t-1} + \alpha \tilde{\epsilon}_t \tag{82.14}$$

where $\tilde{\epsilon}_t$ is scalar IID with mean zero and unit variance.

Activating a concern about robustness ($\sigma < 0$) implies the utility minimizing alter ego sets

$$\tilde{w}_t = K(\sigma, \hat{\beta})\, \mu_{s,t-1}$$

making the worst-case model for μ_{st}:

$$\mu_{st} = (1 + \alpha\, K(\sigma, \hat{\beta}))\, \mu_{s,t-1} + \alpha\, \tilde{\epsilon}_t \tag{82.15}$$

For the allocation to remain the same, we require the robust Euler equation $\hat{\beta} R\, \hat{\mathbb{E}}_t \mu_{s,t+1} = \mu_{st}$ to hold under the worst-case model, which gives

$$(\hat{\beta} R)^{-1} = 1 + \alpha\, K(\sigma, \hat{\beta}) \tag{82.16}$$

The minimizing agent's Bellman equation, a pure forecasting problem, yields

$$\hat{\zeta}(\hat{\beta}) \equiv 1 + \alpha K(\sigma, \hat{\beta}) = \frac{1}{1 - \sigma\alpha^2 P(\hat{\beta})} \tag{82.17}$$

where $P(\hat{\beta})$ solves the scalar Bellman equation:

$$-P(\hat{\beta}) = \frac{\hat{\beta} - 1 + \sigma\alpha^2 + \sqrt{(\hat{\beta} - 1 + \sigma\alpha^2)^2 + 4\sigma\alpha^2}}{-2\sigma\alpha^2} \tag{82.18}$$

Solving (82.16)-(82.18) for $\hat{\beta}$ gives exactly (82.13).

Equation (82.13) is the useful numerical object because it gives a straight-line map from the robustness parameter to the observationally equivalent discount factor.

82.3.5 Precautionary savings interpretation

The consumer's concern about model misspecification activates the precautionary savings motive that underlies the observational-equivalence theorem.

A concern about robustness makes the consumer save *more*.

Decreasing β makes the consumer save *less*.

The observational-equivalence theorem says that these two forces can be made to offset each other exactly.

In the special case $\lambda = \delta_h = 0$, $s_t = c_t$ and the consumption rule is

$$c_t = (1 - R^{-2}\beta^{-1}) \left[R k_{t-1} + \mathbb{E}_t \sum_{j=0}^{\infty} R^{-j} d_{t+j} \right] + \left(\frac{(R\beta)^{-1} - 1}{R - 1} \right) b \tag{82.19}$$

The **marginal propensity to consume** out of non-human wealth $R k_{t-1}$ *equals* that out of human wealth $\mathbb{E}_t \sum R^{-j} d_{t+j}$.

This equal-propensity property is a hallmark of the LQ model and *persists* when a concern for robustness is present, in contrast to usual precautionary-savings models with convex marginal utility.

Theorem 82.3.1 says that with $\sigma < 0$, the observationally equivalent $\hat{\beta}$ satisfies $\hat{\beta} < \beta$.

If the starting point has $\beta R = 1$, then $\hat{\beta} R < 1$.

For a non-robust consumer with discount factor $\hat{\beta}$ at the same interest rate, the Euler equation implies $\mathbb{E}_t c_{t+1} < c_t$: expected consumption declines over time.

This downward drift is the impatience offset in *Theorem 82.3.1*.

It cancels the robust consumer's precautionary-savings motive, leaving the consumption and investment quantities unchanged.

The upward-drift comparison appears in *Theorem 82.3.2*, which asks the reverse observational-equivalence question.

The classical precautionary motive arises because:

$$u'''(c) > 0 \;\Rightarrow\; \mathbb{E}_t u'(c_{t+1}) > u'(\mathbb{E}_t c_{t+1}) \;\Rightarrow\; \mathbb{E}_t c_{t+1} > c_t$$

This channel requires *convexity of marginal utility* and is absent with quadratic preferences.

In contrast, the robustness-based precautionary motive operates through distortions of **conditional means** of shocks, shifting the first moment of the innovation to non-financial income.

82.3.6 Observational equivalence and distorted expectations

The observational-equivalence result can be interpreted using a **Stackelberg multiplier game**.

After the minimizing agent has committed to a distortion process $\{w_{t+1}\}$, the maximizing consumer faces the following worst-case law of motion for the state X_t:

$$\begin{aligned} X_{t+1} &= (A - BF(\sigma, \hat{\beta}) + CK(\sigma, \hat{\beta}))X_t + C\tilde{\epsilon}_{t+1} \\ \begin{pmatrix} b_t \\ d_t \end{pmatrix} &= SX_t \end{aligned} \tag{82.20}$$

A robust consumer with concerns about possible misspecification of the approximating model's stochastic process for non-financial income forms expectations of future income using the **distorted transition matrix** $A - BF + CK$ rather than the approximating transition matrix $A - BF$.

The distorted expectations operator $\hat{\mathbb{E}}_t$ satisfies

$$\hat{\mathbb{E}}_t X_{t+j} = (A - BF(\sigma, \hat{\beta}) + CK(\sigma, \hat{\beta}))^j X_t$$

Observational equivalence requires that the modified human-wealth formula

$$\hat{\Psi}_4 \sum_{j=0}^{\infty} R^{-j} \hat{\mathbb{E}}_t d_{t+j}$$

equals its benchmark counterpart $\Psi_4 \sum_{j=0}^{\infty} R^{-j} \mathbb{E}_t d_{t+j}$.

This is achieved by a mutual adjustment of the coefficients $\hat{\Psi}_j$ through $\hat{\beta}$ and the distorted expectation operator $\hat{\mathbb{E}}_t$ through σ.

The worst-case eigenvalue of $A - BF + CK$ exceeds that of $A - BF$ in modulus, so the worst-case distortions make the income process *more persistent* than under the approximating model.

This is the precautionary motive in state-space form: the minimizing agent makes future income look more risky by introducing low-frequency persistence.

82.3.7 Frequency domain interpretation

The LQ permanent income framework has a natural frequency-domain interpretation.

The consumer's concave utility makes him dislike **high-frequency** fluctuations in consumption, which he smooths by adjusting savings.

High-frequency fluctuations are easier to smooth, so the consumer is automatically robust to misspecification of high-frequency features of the income process.

Low-frequency fluctuations are harder to smooth because they are more persistent.

In the frequency-domain notation of HST, the transfer function from shocks ϵ_t to the target $s_t - b_t$ is $G(\zeta)$, and the frequency decomposition of the H_2 criterion is

$$H_2 = -\frac{1}{2\pi}\int_{-\pi}^{\pi} \text{trace}\left[G(\sqrt{\beta}\, e^{i\omega})^\top G(\sqrt{\beta}\, e^{i\omega})\right] d\omega$$

The integrand $G^\top G$ is *largest at low frequencies* $\omega \approx 0$, where the consumer's welfare is most sensitive to income variability.

Recognizing this, the minimizing agent concentrates the worst-case distortions at low frequencies.

The distortion process has spectral density $W(\zeta)^\top W(\zeta)$ that is concentrated near $\omega = 0$.

The variance of the worst-case shocks grows as $|\sigma|$ increases.

82.3.8 Detection error probabilities

A natural way to discipline the choice of σ (or θ) is to ask: **how difficult would it be to statistically distinguish the approximating model from the worst-case model?**

For a sample of length T, one can use a **log-likelihood ratio test** to compare the two hypotheses.

The **detection error probability** (DEP) is the probability of making the wrong decision using the log-likelihood ratio statistic when one does not know which model generated the data.

Specifically:

$$\text{DEP}(\sigma) = \frac{1}{2}\left[\mathbb{P}\{\text{prefer approx.} \mid \text{worst-case is true}\} + \mathbb{P}\{\text{prefer worst-case} \mid \text{approx. is true}\}\right]$$

When $\sigma = 0$ the two models are identical and DEP $= 0.5$.

As $|\sigma|$ increases the models diverge and the DEP falls toward zero.

The full DEP calculation requires a specified approximating model, its worst-case counterpart, and the sample length used in the likelihood-ratio experiment.

We compute such a DEP for a robust Bewley model below.

> **Note**
>
> HST suggested that a DEP above 0.2 is "plausible", meaning the models are still hard enough to distinguish statistically that a concern for robustness is warranted.
>
> Values of σ corresponding to DEP ≥ 0.2 define a set of plausible worst-case models.

82.3.9 Robustness of decision rules

To evaluate whether robust decision rules perform better than the non-robust rule when the data are generated by a distorted model, define the **payoff** when the decision rule is designed for robustness parameter σ_2 and the data are generated by the distorted model associated with σ_1:

$$\pi(\sigma_1;\sigma_2) = -\mathbb{E}_{0,\sigma_1} \sum_{t=0}^{\infty} \beta^t \, x_t^\top H(\sigma_2)^\top H(\sigma_2) \, x_t \tag{82.21}$$

where the state evolves under decision rule $F(\sigma_2)$ and worst-case shocks $K(\sigma_1)$:

$$x_{t+1} = (A - BF(\sigma_2) + CK(\sigma_1))x_t + C\epsilon_{t+1} \tag{82.22}$$

For $\sigma_1 = 0$ (approximating model generates data), the non-robust rule ($\sigma_2 = 0$) is optimal by construction.

As σ_1 decreases (the data are generated by increasingly distorted models), the payoff of the $\sigma_2 = 0$ rule deteriorates faster than that of robust rules.

Computing the payoff comparison requires solving the full HST matrix problem for $F(\sigma_2)$ and $K(\sigma_1)$.

82.3.10 Another observational equivalence result

Theorem 82.3.2 (Observational Equivalence, II)

Fix all parameters except (σ, β) and consider a consumption-investment allocation for $(\hat{\sigma}, \hat{\beta})$ where $\hat{\beta}R = 1$ and $\hat{\sigma} < 0$.

Then there exists $\tilde{\beta} > \hat{\beta}$ such that the $(\hat{\sigma}, \hat{\beta})$ allocation also solves the $(0, \tilde{\beta})$ problem.

Theorem 82.3.1 showed that starting from a benchmark with $\beta R = 1$, activating robustness ($\sigma < 0$) is equivalent to *reducing* β.

Theorem 82.3.2 goes in the opposite direction: it shows that the effects of activating a concern for robustness from a starting point with $\beta R = 1$ are replicated by *increasing* β while setting $\sigma = 0$.

In other words, when $\beta R = 1$, a concern for robustness operates like an *increase* in the discount factor, pushing $\beta R > 1$ and imparting an *upward drift* to the expected consumption profile.

Proof. With $\hat{\beta}R = 1$ and $\hat{\sigma} < 0$, the robust Euler equation implies

$$\hat{\mathbb{E}}_t \mu_{c,t+1} = \mu_{ct}$$

One seeks $\tilde{\beta} > \hat{\beta}$ and $\sigma = 0$ such that the same allocation solves the non-robust problem with discount factor $\tilde{\beta}$.

The key step is to observe that the worst-case distortion $K(\hat{\sigma}, \hat{\beta})$ introduces a drift in the marginal utility process that is equivalent to the drift produced by raising the discount factor above $\hat{\beta}$.

Equating the two drifts and solving the scalar Bellman equation for K yields

$$\tilde{\beta}(\hat{\sigma}) = \frac{\hat{\beta}(1+\hat{\beta})}{2(1+\hat{\sigma}\alpha^2)} \left[1 + \sqrt{1 - 4\hat{\beta}\,\frac{1+\hat{\sigma}\alpha^2}{(1+\hat{\beta})^2}} \right] \tag{82.23}$$

The solution satisfies $\tilde{\beta} > \hat{\beta}$ when $\hat{\sigma} < 0$.

The map (82.23) is a closed form, so we can plot it directly.

The next figure compares the two observational-equivalence loci for the two-factor calibration, using $\alpha^2 = \sigma_1^2 + (1-\beta)^2\sigma_2^2$ (derived below in (82.27)).

We start from a benchmark with $\hat{\beta}R = 1$, so $\hat{\beta} = \beta$.

```
β_bench = β  # benchmark with β R = 1
α2 = σ1**2 + (1 - β)**2 * σ2**2  # two-factor α^2 (see eq:bew_alpha2)

σ_hat_vals = np.linspace(0.0, -0.16, 60)

# Locus I (eq:obseq / eq:bew_locus): non-robust agent fixed at βR=1;
# report the robust twin's discount factor β̂(σ) < β
β_hat = β_bench + σ_hat_vals * α2 * β_bench / (1 - β_bench)

# Locus II (eq:obsequivn2): robust agent fixed at βR=1;
# report the non-robust twin's discount factor β̃(σ̂) > β
disc = 1 - 4 * β_bench * (1 + σ_hat_vals * α2) / (1 + β_bench)**2
β_tilde = (β_bench * (1 + β_bench)) / (2 * (1 + σ_hat_vals * α2)) \
          * (1 + np.sqrt(disc))

fig, ax = plt.subplots()
ax.plot(-σ_hat_vals, β_hat, lw=2, color='C3',
        label=r'locus I: robust twin $\hat\beta(\sigma)<\beta$'
              '\n(non-robust agent fixed at $\\beta R=1$)')
ax.plot(-σ_hat_vals, β_tilde, lw=2, color='C0',
        label=r'locus II: non-robust twin $\tilde\beta(\hat\sigma)>\beta$'
              '\n(robust agent fixed at $\\beta R=1$)')
ax.axhline(β_bench, color='k', linestyle=':', lw=1,
           label=r'benchmark $\beta$ ($\beta R = 1$)')
ax.set_xlabel(r'robustness concern $|\sigma|$')
ax.set_ylabel('discount factor of the equivalent agent')
ax.legend(fontsize=8.5)
plt.show()

print(f"at σ̂ = {σ_hat_vals[-1]:.3f}:  β̃ = {β_tilde[-1]:.4f} > β = {β_bench}")
print(f"                        β̂ = {β_hat[-1]:.4f} < β = {β_bench}")
```

```
at σ̂ = -0.160:  β̃ = 0.9905 > β = 0.95
                        β̂ = 0.8809 < β = 0.95
```

Both loci pass through the benchmark β at $\sigma = 0$ and separate as the robustness concern grows.

The key to reading the figure is that the two loci hold *different* agents fixed, so the discount factor plotted on the vertical axis refers to a different agent on each curve.

Locus I, from *Theorem 82.3.1*, holds the **non-robust** agent fixed at the benchmark $(\sigma = 0, \beta)$ with $\beta R = 1$ and reports the discount factor $\hat{\beta}(\sigma) < \beta$ of the **robust** agent that mimics it.

This is the sense in which HST call a concern for robustness observationally equivalent to a *lower* discount factor: because robustness already makes the agent save more, its discount factor must be lowered to hold the allocation at the benchmark.

Because the non-robust benchmark has $\beta R = 1$, its optimal consumption is a martingale, $\mathbb{E}_t c_{t+1} = c_t$.

The robust twin chooses the identical consumption process, so it too satisfies $\mathbb{E}_t c_{t+1} = c_t$.

The lower $\hat{\beta}$, which has $\hat{\beta}R < 1$, would on its own impart a downward drift, but the robust agent's precautionary saving offsets it exactly, leaving expected consumption flat.

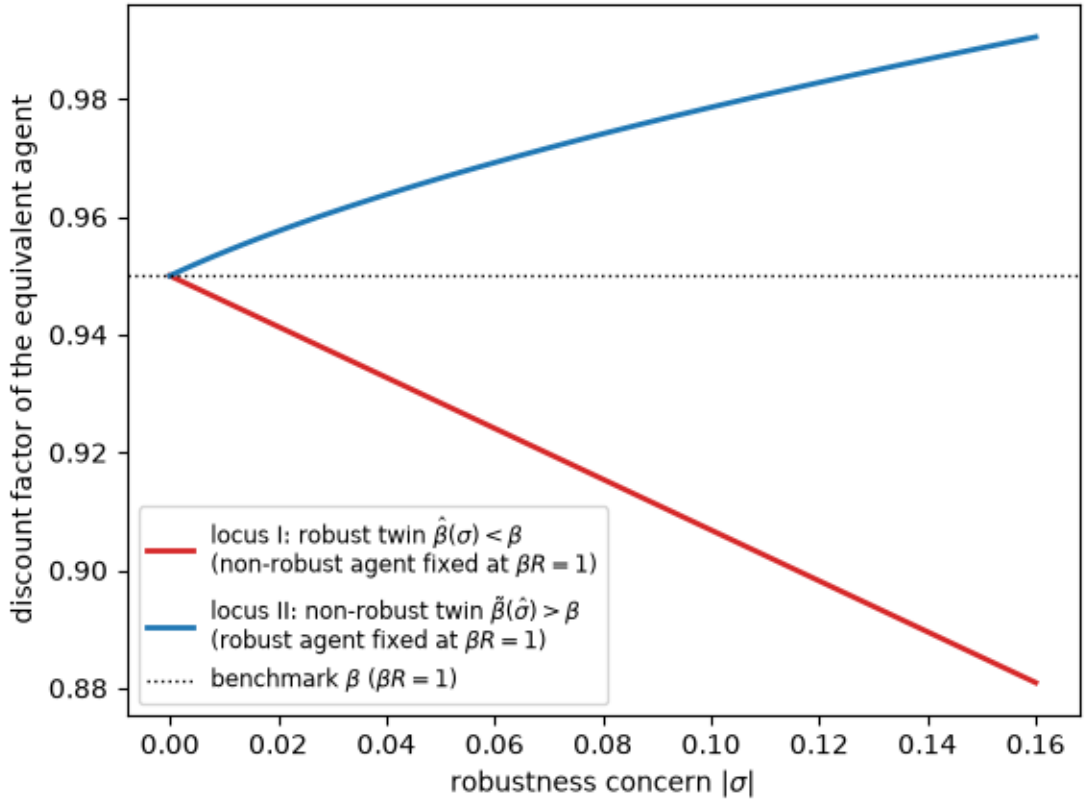


Fig. 82.1: Two observational-equivalence experiments. Locus I (below β) holds the *non-robust* agent fixed at $\beta R = 1$ and reports the *robust* twin's discount factor $\widehat{\beta}(\sigma)$; locus II (above β) holds the *robust* agent fixed at $\beta R = 1$ and reports the *non-robust* twin's discount factor $\widetilde{\beta}(\hat{\sigma})$.

Locus II, from *Theorem 82.3.2*, instead holds the **robust** agent fixed at $(\hat{\sigma}, \beta)$ with $\beta R = 1$ and reports the discount factor $\tilde{\beta}(\hat{\sigma}) > \beta$ of the **non-robust** agent that mimics it.

Here there is no impatience offset, so the common allocation inherits the robust agent's precautionary *upward* drift, which the non-robust twin reproduces through $\tilde{\beta} R > 1$.

The two experiments encode the *same* economics: a concern for robustness adds precautionary saving that acts like extra patience.

They differ only in which agent is anchored at $\beta R = 1$, and hence in whether the common saving motive shows up as an exactly-offsetting impatience adjustment (locus I, expected consumption flat) or as an upward drift in expected consumption (locus II).

82.3.11 A robust LQ Bewley model

We now synthesise the lecture by embedding the Bewley economy of *Consumption Smoothing with Incomplete and Complete Markets* into the HST framework and applying the observational-equivalence theorem.

We shall construct a family of **robust Bewley economies**, parameterised by a robustness level $\sigma \leq 0$, whose equilibrium quantities are identical to those of the plain vanilla Bewley model.

We first map the Bewley economy into HST notation, specialising the robust model to $\lambda = \delta_h = 0$ (no habits, no durable goods) and to a pure endowment economy (no physical capital, $k_t = 0$).

In this case:

Services equal consumption: $s_t = c_t$.

The only traded security is the one-period risk-free bond, and we write the household's net asset position as $a_t = -b_t$ so that positive a_t denotes wealth rather than debt.

The endowment process follows the state-space representation (82.1).

The household's augmented state vector is $x_t = [a_t,\ z_t^\top]^\top$, and the law of motion (82.6) specialises to

$$\begin{pmatrix} a_{t+1} \\ z_{t+1} \end{pmatrix} = \underbrace{\begin{pmatrix} R & R\check{G} \\ 0 & \check{A} \end{pmatrix}}_{A} \begin{pmatrix} a_t \\ z_t \end{pmatrix} + \underbrace{\begin{pmatrix} -R \\ 0 \end{pmatrix}}_{B} c_t + \underbrace{\begin{pmatrix} 0 \\ \check{C} \end{pmatrix}}_{C} \epsilon_{t+1} \tag{82.24}$$

The objective is $\mathbb{E}_0 \sum_{t=0}^{\infty} \beta^t [-(c_t - \gamma)^2/2]$, which is the HST criterion (82.7) with $\sigma = 0$ and $b_t \equiv \gamma$ (a fixed bliss level).

The robust Bellman equation (82.4) with $\sigma = 0$ therefore reduces exactly to the LQ problem of *The LQ Permanent Income Model*, confirming that the HST framework nests the Bewley model.

We next compute the robustness parameter α^2.

From the (c_t, z_t) representation (82.2), the consumption innovation is

$$c_{t+1} - c_t = h\, w_{t+1}, \qquad h = (1-\beta)\, \check{G}(I - \beta\check{A})^{-1}\check{C} \tag{82.25}$$

The vector h plays the role of $\nu^\top = M_s C$ in the HST scalar α formula (82.12).

Consequently,

$$\alpha^2 = hh^\top = (1-\beta)^2\, \check{G}(I - \beta\check{A})^{-1}\check{C}\check{C}^\top (I - \beta\check{A}^\top)^{-1}\check{G}^\top \tag{82.26}$$

For the two-factor model (82.3) with $\check{A} = \mathrm{diag}(1, 0)$ and $\check{C} = \mathrm{diag}(\sigma_1, \sigma_2)$ this simplifies to

$$\alpha^2 = \sigma_1^2 + (1-\beta)^2\, \sigma_2^2 \tag{82.27}$$

The permanent shock variance σ_1^2 enters with coefficient 1 because a unit permanent shock is *fully* capitalised into consumption.

The transitory shock variance σ_2^2 enters with the small coefficient $(1-\beta)^2$ because only its annuity value is consumed.

Applying *Theorem 82.3.1* (82.13) with equilibrium interest rate $R = \beta_0^{-1}$ and α^2 from (82.27) gives the **Bewley observational equivalence locus**:

$$\hat{\beta}(\sigma) = \beta_0 + \frac{\sigma\,\alpha^2\,\beta_0}{1-\beta_0} \tag{82.28}$$

For $\sigma < 0$, we have $\hat{\beta}(\sigma) < \beta_0$.

An agent with the pair $(\sigma, \hat{\beta}(\sigma))$ on this locus is more concerned about model misspecification (lower σ) but also more impatient (lower $\hat{\beta}$); the two forces cancel exactly, leaving the consumption decision rule unchanged.

These ingredients combine into a robust Bewley equilibrium.

Proposition 82.3.1

Suppose all agents in the Bewley economy share a common pair $(\sigma, \hat{\beta}(\sigma))$ lying on the locus (82.28), with $R = \beta_0^{-1}$.

Then every agent's optimal consumption plan is identical to that of the plain vanilla $(\sigma = 0,\ \beta_0)$ economy, and the equilibrium interest rate remains $R = \beta_0^{-1}$.

Proof. By *Theorem 82.3.1*, each agent's consumption-saving rule is identical to the benchmark.

The goods-market clearing condition $\int c_t^i\, di = Y$ is therefore satisfied at $R = \beta_0^{-1}$ for the same reason as in the benchmark Bewley economy.

Heterogeneous (β_i, σ_i) preferences

A richer extension populates the economy with a **continuum of types**, each indexed by a robustness parameter $\sigma_i \in [\underline{\sigma}, 0]$, with discount factor

$$\beta_i = \hat{\beta}(\sigma_i) = \beta_0 + \frac{\sigma_i\,\alpha^2\,\beta_0}{1-\beta_0} \tag{82.29}$$

Since all pairs (σ_i, β_i) lie on (82.28), every agent adopts the **same consumption rule** as the benchmark.

Aggregate dynamics are unchanged because the cross-section mean of consumption equals Y and the cross-section variance grows at rate α^2 per period.

The equilibrium interest rate is unchanged: $R = \beta_0^{-1}$.

Agents are observationally indistinguishable to an outside econometrician because data on (c_t^i, a_t^i) cannot reveal whether agent i has $\sigma_i = 0$ or $\sigma_i < 0$.

Agents differ in their internal model because an agent with $\sigma_i < 0$ applies a worst-case distortion $w_{t+1}^i = K(\sigma_i, \beta_i)\,\mu_{s,t}^i$ to her conditional expectations, while an agent with $\sigma_i = 0$ takes the approximating model at face value.

This sets the stage for a Bewley model with **heterogeneous ambiguity aversion**: although every agent acts identically in terms of observable choices, they hold different subjective models of the income process and have different attitudes toward model uncertainty.

Computation

```
# Bewley parameters
β0_bew = β       # 0.95
σ1_bew = σ1      # 0.15
σ2_bew = σ2      # 0.30
R_bew = 1.0 / β0_bew

# Two-factor Bewley α^2
α2_bew = σ1_bew**2 + (1 - β0_bew)**2 * σ2_bew**2

print(f"α^2  (Bewley, two-factor)        = {α2_bew:.6f}")
print(f"  permanent component  σ1^2     = {σ1_bew**2:.6f}  "
      f"({100*σ1_bew**2/α2_bew:.1f} % of α^2)")
print(f"  transitory component (1-β)^2σ2^2= {(1-β0_bew)**2*σ2_bew**2:.6f}  "
      f"({100*(1-β0_bew)**2*σ2_bew**2/α2_bew:.1f} % of α^2)")
```

```
α^2  (Bewley, two-factor)        = 0.022725
  permanent component  σ1^2     = 0.022500  (99.0 % of α^2)
  transitory component (1-β)^2σ2^2= 0.000225  (1.0 % of α^2)
```

The calculation shows why permanent shocks dominate α^2 in this calibration.

We now solve the scalar robust forecasting problem attached to this α^2.

The solution selects the Bellman-equation root that satisfies the observational-equivalence Euler equation.

```
def robust_scalar_solution(σ, β0, α2):
    """
    Solve the scalar robust marginal-utility problem on the
    observational-equivalence locus.
    """
    α = np.sqrt(α2)
    R = 1.0 / β0

    if np.isclose(σ, 0.0):
        return β0, np.nan, 1.0, 0.0

    β_hat = β0 + σ * α2 * β0 / (1 - β0)
    disc = (β_hat - 1 + σ * α2)**2 + 4 * σ * α2
    root_disc = np.sqrt(max(disc, 0.0))
    target_ζ = 1 / (β_hat * R)

    candidates = []
    for sign in (1.0, -1.0):
        P = (β_hat - 1 + σ * α2 + sign * root_disc) / (-2 * σ * α2)
        ζ = 1 / (1 - σ * α2 * P)
        K = (ζ - 1) / α
        candidates.append((abs(ζ - target_ζ), P, ζ, K))

    _, P, ζ, K = min(candidates, key=lambda x: x[0])
    return β_hat, P, ζ, K


def log_likelihood_ratio(paths, ζ, α):
    """
    Return log p_worst(path) - log p_approx(path).
    """
```

(continues on next page)

(continued from previous page)

```
    lag = paths[:, :-1]
    lead = paths[:, 1:]
    ll_worst = -0.5 * np.sum(((lead - ζ * lag) / α)**2, axis=1)
    ll_approx = -0.5 * np.sum(((lead - lag) / α)**2, axis=1)
    return ll_worst - ll_approx


def simulate_scalar_paths(ζ, α, T, n_paths, seed):
    rng = np.random.default_rng(seed)
    paths = np.zeros((n_paths, T + 1))
    shocks = rng.standard_normal((n_paths, T))

    for t in range(T):
        paths[:, t + 1] = ζ * paths[:, t] + α * shocks[:, t]

    return paths


def detection_error_probability(ζ, α, T=40, n_paths=10_000, seed=1234):
    """
    Finite-sample DEP for the approximating and worst-case scalar laws.
    """
    if np.isclose(ζ, 1.0):
        return 0.5

    approx_paths = simulate_scalar_paths(1.0, α, T, n_paths, seed)
    worst_paths = simulate_scalar_paths(ζ, α, T, n_paths, seed + 1)

    llr_approx = log_likelihood_ratio(approx_paths, ζ, α)
    llr_worst = log_likelihood_ratio(worst_paths, ζ, α)

    return 0.5 * (np.mean(llr_worst < 0) + np.mean(llr_approx > 0))
```

The next figure reports worst-case dynamics and model-detection probabilities implied by this solved scalar problem.

```
α_bew = np.sqrt(α2_bew)
β_min = 0.88
σ_min = (β_min - β0_bew) * (1 - β0_bew) / (α2_bew * β0_bew)
σ_vals = np.linspace(0.0, σ_min, 31)

solutions = np.array([robust_scalar_solution(σ, β0_bew, α2_bew) for σ in σ_vals])
β_hat_vals = solutions[:, 0]
ζ_vals = solutions[:, 2]
K_vals = solutions[:, 3]
dep_vals = np.array([
    detection_error_probability(ζ, α_bew)
    for ζ in ζ_vals
])

fig, axes = plt.subplots(1, 2, figsize=(11.2, 4.1))

horizons = np.arange(31)
for σ in [0.0, σ_vals[10], σ_vals[20]]:
    β_hat, P, ζ, K = robust_scalar_solution(σ, β0_bew, α2_bew)
    label = rf'$\sigma={σ:.3f}$, $\zeta={ζ:.3f}$'
    axes[0].plot(horizons, ζ**horizons, lw=2, label=label)
```

(continues on next page)

(continued from previous page)

```
axes[0].set_xlabel('horizon')
axes[0].set_ylabel(r'response of $\mu_{s,t+h}$')
axes[0].set_title('worst-case impulse response')
axes[0].legend(fontsize=8.5)

axes[1].plot(-σ_vals, dep_vals, lw=2, color='C0')
axes[1].axhline(0.2, color='C3', linestyle='--', lw=1.2,
                label='DEP = 0.2')
axes[1].set_xlabel(r'robustness concern $-\sigma$')
axes[1].set_ylabel('detection error probability')
axes[1].set_ylim(0.0, 0.52)
axes[1].set_title('finite-sample detectability')
axes[1].legend(fontsize=8.5)

fig.tight_layout()
plt.show()
```

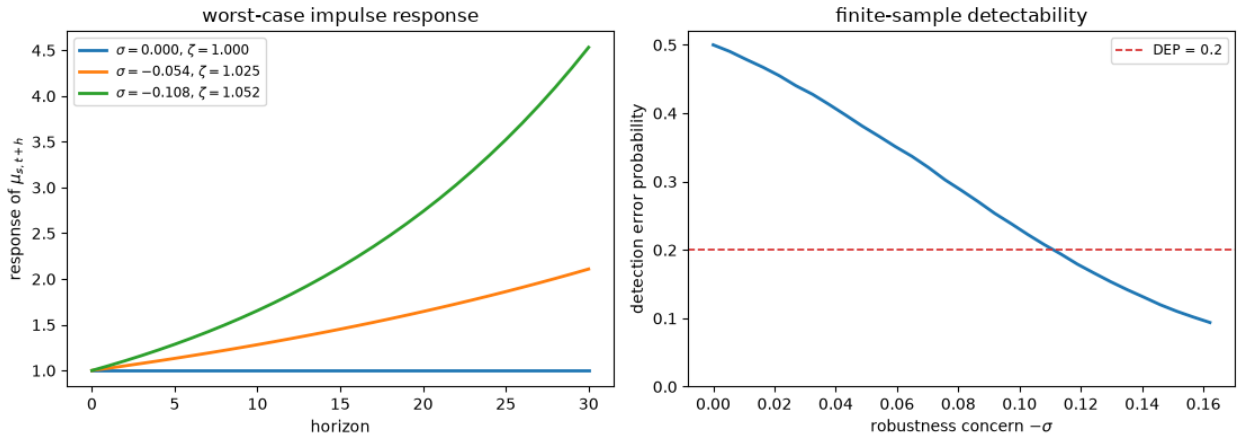


Fig. 82.2: Solved robust scalar model

The left panel shows that the solved worst-case law makes marginal utility more persistent as σ becomes more negative.

The right panel computes the DEP from the exact likelihood ratio between the approximating scalar law $\mu_{t+1} = \mu_t + \alpha\epsilon_{t+1}$ and the solved worst-case law $\mu_{t+1} = \zeta(\sigma)\mu_t + \alpha\epsilon_{t+1}$.

82.3.12 Concluding remarks

We close with a summary of the key messages from all three lectures.

The LQ permanent income model, a rational-expectations version of Friedman's permanent income hypothesis, has two complementary state-space representations:

1. (b_t, z_t) **representation**: emphasises that the consumer's optimal borrowing is history dependent and cointegrated with consumption.
2. (c_t, z_t) **representation**: emphasises that consumption is a martingale (random walk) and that assets b_t are encoded in consumption, so the impulse response function of consumption is "box-shaped": a permanent shift in the level.

We embedded this single-agent model in a Bewley equilibrium with a continuum of ex-post heterogeneous consumers.

The equilibrium gross interest rate $R = \beta^{-1}$ is supported by constant average consumption, though the cross-section variance of consumption grows linearly with age.

A complete-markets version of the same model achieves full risk sharing and a time-invariant consumption distribution at the cost of more complex financial arrangements (Arrow securities).

A concern for model misspecification, parameterised by $\sigma = -\theta^{-1} \leq 0$, alters the permanent income model.

A concern for robustness generates a precautionary savings motive even under quadratic preferences by distorting the conditional means of income shocks.

The distorted worst-case model makes the income process **more persistent**, shifting power toward low frequencies where the permanent income consumer is most vulnerable.

The observational equivalence theorem *Theorem 82.3.1* shows that for quantities (c_t, i_t) alone, a concern for robustness is indistinguishable from a reduction in β.

The reverse theorem *Theorem 82.3.2* shows that, starting from $\beta R = 1$, robustness is observationally equivalent to an *increase* in β, which imparts an upward drift to expected consumption.

Detection error probabilities provide a principled way to calibrate σ: choose $|\sigma|$ small enough that the approximating and worst-case models remain difficult to distinguish statistically.

The observationally equivalent $(\sigma, \hat{\beta})$ pairs **do** have different implications for asset prices, a point explored further by HST in the asset-pricing context.

The robust Bewley economy shows how agents can have the same consumption decision rule and support the same equilibrium interest rate $R = \beta_0^{-1}$ while differing in their worst-case subjective income dynamics.

82.4 Exercises

Exercise 82.4.1

We translate from the benchmark Bewley economy to HST notation.

Specialise the robust-control setup to the no-habit, no-capital LQ Bewley environment ($\lambda = \delta_h = 0$, $k_t = 0$), and let the endowment process be the two-factor model in (82.3).

1. Write the household state as $x_t = [a_t, z_t^\top]^\top$, where $a_t = -b_t$ is net assets, and derive matrices (A, B, C) for the law of motion (82.6).
2. Show that when $\sigma = 0$, the Bellman problem coincides with the LQ permanent-income problem.
3. Derive α^2 and verify

$$\alpha^2 = \sigma_1^2 + (1-\beta)^2\sigma_2^2.$$

Interpret economically why the permanent and transitory components enter with different weights.

Solution

Here is one solution:

1. With $x_t = [a_t, z_t^\top]^\top$ and budget law $a_{t+1} = R(a_t + y_t - c_t)$, $y_t = \check{G} z_t$, and $z_{t+1} = \check{A} z_t + \check{C}\epsilon_{t+1}$, the stacked law is

$$\begin{pmatrix} a_{t+1} \\ z_{t+1} \end{pmatrix} = \underbrace{\begin{pmatrix} R & R\check{G} \\ 0 & \check{A} \end{pmatrix}}_{A} \begin{pmatrix} a_t \\ z_t \end{pmatrix} + \underbrace{\begin{pmatrix} -R \\ 0 \end{pmatrix}}_{B} c_t + \underbrace{\begin{pmatrix} 0 \\ \check{C} \end{pmatrix}}_{C} \epsilon_{t+1}.$$

The sign of B is negative because higher c_t reduces asset accumulation a_{t+1}.

2. At $\sigma = 0$, the robust Bellman problem collapses to the ordinary LQ objective with no minimizing distortion term, so the planner/consumer problem is exactly the permanent-income problem with quadratic utility and linear constraints.
3. From the (c_t, z_t) representation,

$$\Delta c_{t+1} = h\,\epsilon_{t+1}, \qquad h = (1-\beta)\check{G}(I - \beta\check{A})^{-1}\check{C}.$$

In HST notation, $\alpha^2 = hh^\top$, and for the two-factor calibration $\check{A} = \mathrm{diag}(1, 0)$ and $\check{C} = \mathrm{diag}(\sigma_1, \sigma_2)$, so

$$\alpha^2 = \sigma_1^2 + (1-\beta)^2\sigma_2^2.$$

Permanent shocks get unit weight because they shift lifetime resources one-for-one, while transitory shocks are annuitised and therefore scaled by $(1-\beta)$ in consumption growth.

Exercise 82.4.2

This exercise studies a continuum of robust but observationally equivalent Bewley consumers.

Fix a benchmark pair $(\beta_0, \sigma = 0)$ with $R = \beta_0^{-1}$ and define

$$\beta(\sigma) = \beta_0 + \frac{\sigma\alpha^2\beta_0}{1-\beta_0}, \qquad \sigma \in [-\bar{\sigma}, 0].$$

Suppose a unit interval of consumers is indexed by i with type $\sigma_i \in [-\bar{\sigma}, 0]$ and discount factor $\beta_i = \beta(\sigma_i)$.

1. Use *Theorem 82.3.1* to show that each type has the same consumption rule as the benchmark $(\beta_0, 0)$ agent.
2. Prove that aggregate consumption and bond-market clearing imply the same equilibrium interest rate $R = \beta_0^{-1}$ as in the plain-vanilla Bewley model.
3. Explain why agents can be observationally equivalent in quantities while still holding different worst-case subjective models.

Solution

Here is one solution:

1. *Theorem 82.3.1* implies that if (σ_i, β_i) lies on

$$\beta_i = \beta_0 + \frac{\sigma_i\alpha^2\beta_0}{1-\beta_0},$$

then type i chooses the same decision rule as the benchmark $(0, \beta_0)$ agent and all types share the same consumption policy function $c_t = \mathcal{C}(a_t, z_t)$.

2. Since all individual policy rules coincide with benchmark Bewley policies, aggregating over consumers gives the same goods- and bond-market clearing conditions and supports the same equilibrium $R = \beta_0^{-1}$.
3. Observational equivalence concerns quantities generated by optimal rules, so distinct (σ_i, β_i) can generate the same $\{c_t^i, a_t^i\}$ while implying different internal worst-case beliefs.

Exercise 82.4.3

This exercise separates quantities from beliefs without introducing an additional calibration.

Consider two agents a and b in the robust Bewley economy with $\sigma^a < \sigma^b \leq 0$ and $\beta^j = \beta_0 + \sigma^j \alpha^2 \beta_0/(1-\beta_0)$ for $j \in \{a, b\}$.

1. Use (82.25) and (82.28) to show that the two agents have the same consumption innovation $h\epsilon_{t+1}$.
2. Show that if the two agents start from the same (a_t, z_t) and observe the same shock ϵ_{t+1}, then their next-period choices of consumption and assets coincide.
3. Explain why the two agents can nevertheless disagree about the worst-case conditional mean of ϵ_{t+1}.

Summarise what is and is not identified by data on quantities alone.

Solution

Here is one solution:

1. Equation (82.28) places both agents on the observational-equivalence locus, so *Theorem 82.3.1* implies that both use the benchmark consumption rule and therefore the same innovation vector h in (82.25).
2. With a common state and common shock, both agents apply the same policy function and the same law of motion, so $c^a_{t+1} = c^b_{t+1}$ and $a^a_{t+1} = a^b_{t+1}$.
3. The minimizing feedback $K(\sigma^j, \beta^j)$ can differ across j, so the agents can attach different worst-case conditional means to the same shock process even though their observable choices coincide.

Conclusion: quantities identify the equilibrium decision rule but not the decomposition between impatience (β) and robustness (σ) along the observational-equivalence locus.

CHAPTER

EIGHTYTHREE

PRODUCTION SMOOTHING VIA INVENTORIES

Contents

- *Production Smoothing via Inventories*
  - *Overview*
  - *Example 1*
  - *Inventories Not Useful*
  - *Inventories Useful but are Hardwired to be Zero Always*
  - *Example 2*
  - *Example 3*
  - *Example 4*
  - *Example 5*
  - *Example 6*
  - *Exercises*

In addition to what's in Anaconda, this lecture employs the following library:

```
!pip install quantecon
```

83.1 Overview

This lecture can be viewed as an application of this *quantecon lecture* about linear quadratic control theory.

It formulates a discounted dynamic program for a firm that chooses a production schedule to balance

- minimizing costs of production across time, against
- keeping costs of holding inventories low

In the tradition of a classic book by Holt, Modigliani, Muth, and Simon [Holt *et al.*, 1960], we simplify the firm's problem by formulating it as a linear quadratic discounted dynamic programming problem of the type studied in this *quantecon lecture*.

Because its costs of production are increasing and quadratic in production, the firm holds inventories as a buffer stock in order to smooth production across time, provided that holding inventories is not too costly.

But the firm also wants to make its sales out of existing inventories, a preference that we represent by a cost that is quadratic in the difference between sales in a period and the firm's beginning of period inventories.

We compute examples designed to indicate how the firm optimally smooths production while keeping inventories close to sales.

To introduce components of the model, let

- S_t be sales at time t
- Q_t be production at time t
- I_t be inventories at the beginning of time t
- $\beta \in (0,1)$ be a discount factor
- $c(Q_t) = c_1 Q_t + c_2 Q_t^2$, be a cost of production function, where $c_1 > 0, c_2 > 0$, be an inventory cost function
- $d(I_t, S_t) = d_1 I_t + d_2 (S_t - I_t)^2$, where $d_1 > 0, d_2 > 0$, be a cost-of-holding-inventories function, consisting of two components:
  - a cost $d_1 I_t$ of carrying inventories, and
  - a cost $d_2 (S_t - I_t)^2$ of having inventories deviate from sales
- $p_t = a_0 - a_1 S_t + v_t$ be an inverse demand function for a firm's product, where $a_0 > 0, a_1 > 0$ and v_t is a demand shock at time t
- $\pi_t = p_t S_t - c(Q_t) - d(I_t, S_t)$ be the firm's profits at time t
- $\sum_{t=0}^{\infty} \beta^t \pi_t$ be the present value of the firm's profits at time 0
- $I_{t+1} = I_t + Q_t - S_t$ be the law of motion of inventories
- $z_{t+1} = A_{22} z_t + C_2 \epsilon_{t+1}$ be a law of motion for an exogenous state vector z_t that contains time t information useful for predicting the demand shock v_t
- $v_t = G z_t$ link the demand shock to the information set z_t
- the constant 1 be the first component of z_t

To map our problem into a linear-quadratic discounted dynamic programming problem (also known as an optimal linear regulator), we define the **state** vector at time t as

$$x_t = \begin{bmatrix} I_t \\ z_t \end{bmatrix}$$

and the **control** vector as

$$u_t = \begin{bmatrix} Q_t \\ S_t \end{bmatrix}$$

The law of motion for the state vector x_t is evidently

$$\begin{bmatrix} I_{t+1} \\ z_t \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & A_{22} \end{bmatrix} \begin{bmatrix} I_t \\ z_t \end{bmatrix} + \begin{bmatrix} 1 & -1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} Q_t \\ S_t \end{bmatrix} + \begin{bmatrix} 0 \\ C_2 \end{bmatrix} \epsilon_{t+1}$$

or

$$x_{t+1} = A x_t + B u_t + C \epsilon_{t+1}$$

(At this point, we ask that you please forgive us for using Q_t to be the firm's production at time t, while below we use Q as the matrix in the quadratic form $u_t' Q u_t$ that appears in the firm's one-period profit function)

We can express the firm's profit as a function of states and controls as

$$\pi_t = -(x_t' R x_t + u_t' Q u_t + 2 u_t' N x_t)$$

To form the matrices R, Q, N in an LQ dynamic programming problem, we note that the firm's profits at time t function can be expressed

$$
\begin{aligned}
\pi_t =& p_t S_t - c\left(Q_t\right) - d\left(I_t, S_t\right) \\
=& \left(a_0 - a_1 S_t + v_t\right) S_t - c_1 Q_t - c_2 Q_t^2 - d_1 I_t - d_2 \left(S_t - I_t\right)^2 \\
=& a_0 S_t - a_1 S_t^2 + G z_t S_t - c_1 Q_t - c_2 Q_t^2 - d_1 I_t - d_2 S_t^2 - d_2 I_t^2 + 2 d_2 S_t I_t \\
=& - \left(\underbrace{d_1 I_t + d_2 I_t^2}_{x_t' R x_t} + \underbrace{a_1 S_t^2 + d_2 S_t^2 + c_2 Q_t^2}_{u_t' Q u_t} \underbrace{- a_0 S_t - G z_t S_t + c_1 Q_t - 2 d_2 S_t I_t}_{2 u_t' N x_t} \right) \\
=& - \Bigg(\begin{bmatrix} I_t & z_t' \end{bmatrix} \underbrace{\begin{bmatrix} d_2 & \frac{d_1}{2} S_c \\ \frac{d_1}{2} S_c' & 0 \end{bmatrix}}_{\equiv R} \begin{bmatrix} I_t \\ z_t \end{bmatrix} + \begin{bmatrix} Q_t & S_t \end{bmatrix} \underbrace{\begin{bmatrix} c_2 & 0 \\ 0 & a_1 + d_2 \end{bmatrix}}_{\equiv Q} \begin{bmatrix} Q_t \\ S_t \end{bmatrix} + 2 \begin{bmatrix} Q_t & S_t \end{bmatrix} \underbrace{\begin{bmatrix} 0 & \frac{c_1}{2} S_c \\ -d_2 & -\frac{a_0}{2} S_c - \frac{G}{2} \end{bmatrix}}_{\equiv N} \begin{bmatrix} I \\ z \end{bmatrix}
\end{aligned}
$$

where $S_c = [1, 0]$.

Remark on notation: The notation for cross product term in the QuantEcon library is N.

The firms' optimum decision rule takes the form

$$u_t = -F x_t$$

and the evolution of the state under the optimal decision rule is

$$x_{t+1} = (A - BF) x_t + C \epsilon_{t+1}$$

The firm chooses a decision rule for u_t that maximizes

$$E_0 \sum_{t=0}^{\infty} \beta^t \pi_t$$

subject to a given x_0.

This is a stochastic discounted LQ dynamic program.

Here is code for computing an optimal decision rule and for analyzing its consequences.

```
import matplotlib.pyplot as plt
import numpy as np
import quantecon as qe
```

```
class SmoothingExample:
    """
    Class for constructing, solving, and plotting results for
    inventories and sales smoothing problem.
    """

    def __init__(self,
                 β=0.96,            # Discount factor
                 c1=1,              # Cost-of-production
                 c2=1,
                 d1=1,              # Cost-of-holding inventories
                 d2=1,
                 a0=10,             # Inverse demand function
                 a1=1,
                 A22=[[1,   0],     # z process
```

(continues on next page)

(continued from previous page)

```
                    [1, 0.9]],
              C2=[[0], [1]],
              G=[0, 1]):

        self.β = β
        self.c1, self.c2 = c1, c2
        self.d1, self.d2 = d1, d2
        self.a0, self.a1 = a0, a1
        self.A22 = np.atleast_2d(A22)
        self.C2 = np.atleast_2d(C2)
        self.G = np.atleast_2d(G)

        # Dimensions
        k, j = self.C2.shape        # Dimensions for randomness part
        n = k + 1                   # Number of states
        m = 2                       # Number of controls

        Sc = np.zeros(k)
        Sc[0] = 1

        # Construct matrices of transition law
        A = np.zeros((n, n))
        A[0, 0] = 1
        A[1:, 1:] = self.A22

        B = np.zeros((n, m))
        B[0, :] = 1, -1

        C = np.zeros((n, j))
        C[1:, :] = self.C2

        self.A, self.B, self.C = A, B, C

        # Construct matrices of one period profit function
        R = np.zeros((n, n))
        R[0, 0] = d2
        R[1:, 0] = d1 / 2 * Sc
        R[0, 1:] = d1 / 2 * Sc

        Q = np.zeros((m, m))
        Q[0, 0] = c2
        Q[1, 1] = a1 + d2

        N = np.zeros((m, n))
        N[1, 0] = - d2
        N[0, 1:] = c1 / 2 * Sc
        N[1, 1:] = - a0 / 2 * Sc - self.G / 2

        self.R, self.Q, self.N = R, Q, N

        # Construct LQ instance
        self.LQ = qe.LQ(Q, R, A, B, C, N, beta=β)
        self.LQ.stationary_values()

    def simulate(self, x0, T=100):

        c1, c2 = self.c1, self.c2
```

(continues on next page)

(continued from previous page)

```
        d1, d2 = self.d1, self.d2
        a0, a1 = self.a0, self.a1
        G = self.G

        x_path, u_path, w_path = self.LQ.compute_sequence(x0, ts_length=T)

        I_path = x_path[0, :-1]
        z_path = x_path[1:, :-1]
        ν_path = (G @ z_path)[0, :]

        Q_path = u_path[0, :]
        S_path = u_path[1, :]

        revenue = (a0 - a1 * S_path + ν_path) * S_path
        cost_production = c1 * Q_path + c2 * Q_path ** 2
        cost_inventories = d1 * I_path + d2 * (S_path - I_path) ** 2

        Q_no_inventory = (a0 + ν_path - c1) / (2 * (a1 + c2))
        Q_hardwired = (a0 + ν_path - c1) / (2 * (a1 + c2 + d2))

        fig, ax = plt.subplots(2, 2, figsize=(15, 10))

        ax[0, 0].plot(range(T), I_path, label="inventories")
        ax[0, 0].plot(range(T), S_path, label="sales")
        ax[0, 0].plot(range(T), Q_path, label="production")
        ax[0, 0].legend(loc=1)
        ax[0, 0].set_title("inventories, sales, and production")

        ax[0, 1].plot(range(T), (Q_path - S_path), color='b')
        ax[0, 1].set_ylabel("change in inventories", color='b')
        span = max(abs(Q_path - S_path))
        ax[0, 1].set_ylim(0-span*1.1, 0+span*1.1)
        ax[0, 1].set_title("demand shock and change in inventories")

        ax1_ = ax[0, 1].twinx()
        ax1_.plot(range(T), ν_path, color='r')
        ax1_.set_ylabel("demand shock", color='r')
        span = max(abs(ν_path))
        ax1_.set_ylim(0-span*1.1, 0+span*1.1)

        ax1_.plot([0, T], [0, 0], '--', color='k')

        ax[1, 0].plot(range(T), revenue, label="revenue")
        ax[1, 0].plot(range(T), cost_production, label="cost_production")
        ax[1, 0].plot(range(T), cost_inventories, label="cost_inventories")
        ax[1, 0].legend(loc=1)
        ax[1, 0].set_title("profits decomposition")

        ax[1, 1].plot(range(T), Q_path, label="production")
        ax[1, 1].plot(range(T), Q_hardwired, label='production when  $I_t$ \
            forced to be zero')
        ax[1, 1].plot(range(T), Q_no_inventory, label='production when \
            inventories not useful')
        ax[1, 1].legend(loc=1)
        ax[1, 1].set_title('three production concepts')

        plt.show()
```

Notice that the above code sets parameters at the following default values

- discount factor $\beta = 0.96$,
- inverse demand function: $a0 = 10, a1 = 1$
- cost of production $c1 = 1, c2 = 1$
- costs of holding inventories $d1 = 1, d2 = 1$

In the examples below, we alter some or all of these parameter values.

83.2 Example 1

In this example, the demand shock follows AR(1) process:

$$\nu_t = \alpha + \rho \nu_{t-1} + \epsilon_t,$$

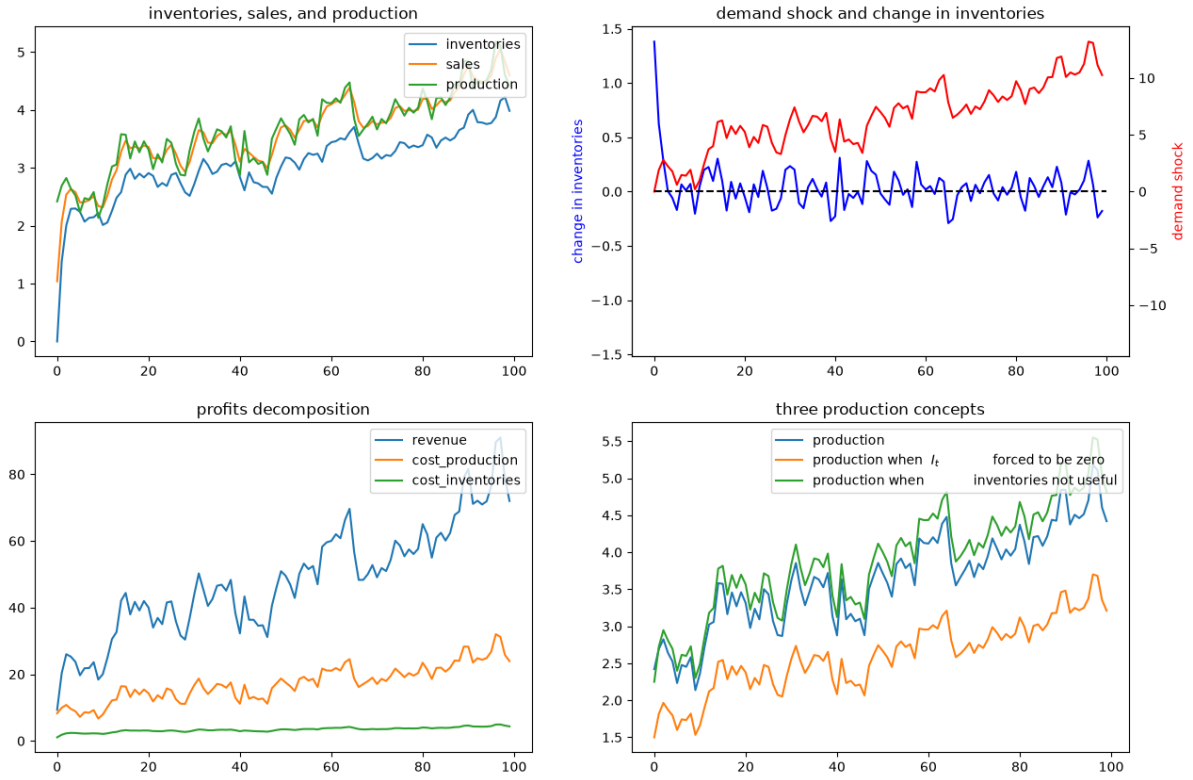
which implies

$$z_{t+1} = \begin{bmatrix} 1 \\ v_{t+1} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ \alpha & \rho \end{bmatrix} \underbrace{\begin{bmatrix} 1 \\ v_t \end{bmatrix}}_{z_t} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} \epsilon_{t+1}.$$

We set $\alpha = 1$ and $\rho = 0.9$, their default values.

We'll calculate and display outcomes, then discuss them below the pertinent figures.

```
ex1 = SmoothingExample()

x0 = [0, 1, 0]
ex1.simulate(x0)
```



The figures above illustrate various features of an optimal production plan.

Starting from zero inventories, the firm builds up a stock of inventories and uses them to smooth costly production in the face of demand shocks.

Optimal decisions evidently respond to demand shocks.

Inventories are always less than sales, so some sales come from current production, a consequence of the cost, $d_1 I_t$ of holding inventories.

The lower right panel shows differences between optimal production and two alternative production concepts that come from altering the firm's cost structure – i.e., its technology.

These two concepts correspond to these distinct altered firm problems.

- a setting in which inventories are not needed
- a setting in which they are needed but we arbitrarily prevent the firm from holding inventories by forcing it to set $I_t = 0$ always

We use these two alternative production concepts in order to shed light on the baseline model.

83.3 Inventories Not Useful

Let's turn first to the setting in which inventories aren't needed.

In this problem, the firm forms an output plan that maximizes the expected value of

$$\sum_{t=0}^{\infty} \beta^t \{p_t Q_t - C(Q_t)\}$$

It turns out that the optimal plan for Q_t for this problem also solves a sequence of static problems $\max_{Q_t} \{p_t Q_t - c(Q_t)\}$.

When inventories aren't required or used, sales always equal production.

This simplifies the problem and the optimal no-inventory production maximizes the expected value of

$$\sum_{t=0}^{\infty} \beta^t \left\{p_t Q_t - C\left(Q_t\right)\right\}.$$

The optimum decision rule is

$$Q_t^{ni} = \frac{a_0 + \nu_t - c_1}{c_2 + a_1}.$$

83.4 Inventories Useful but are Hardwired to be Zero Always

Next, we turn to a distinct problem in which inventories are useful – meaning that there are costs of $d_2(I_t - S_t)^2$ associated with having sales not equal to inventories – but we arbitrarily impose on the firm the costly restriction that it never hold inventories.

Here the firm's maximization problem is

$$\max_{\{I_t, Q_t, S_t\}} \sum_{t=0}^{\infty} \beta^t \left\{p_t S_t - C\left(Q_t\right) - d\left(I_t, S_t\right)\right\}$$

subject to the restrictions that $I_t = 0$ for all t and that $I_{t+1} = I_t + Q_t - S_t$.

The restriction that $I_t = 0$ implies that $Q_t = S_t$ and that the maximization problem reduces to

$$\max_{Q_t} \sum_{t=0}^{\infty} \beta^t \left\{ p_t Q_t - C\left(Q_t\right) - d\left(0, Q_t\right) \right\}$$

Here the optimal production plan is

$$Q_t^h = \frac{a_0 + \nu_t - c_1}{c_2 + a_1 + d_2}.$$

We introduce this I_t **is hardwired to zero** specification in order to shed light on the role that inventories play by comparing outcomes with those under our two other versions of the problem.

The bottom right panel displays a production path for the original problem that we are interested in (the blue line) as well with an optimal production path for the model in which inventories are not useful (the green path) and also for the model in which, although inventories are useful, they are hardwired to zero and the firm pays cost $d(0, Q_t)$ for not setting sales $S_t = Q_t$ equal to zero (the orange line).

Notice that it is typically optimal for the firm to produce more when inventories aren't useful. Here there is no requirement to sell out of inventories and no costs from having sales deviate from inventories.

But "typical" does not mean "always".

Thus, if we look closely, we notice that for small t, the green "production when inventories aren't useful" line in the lower right panel is below optimal production in the original model.

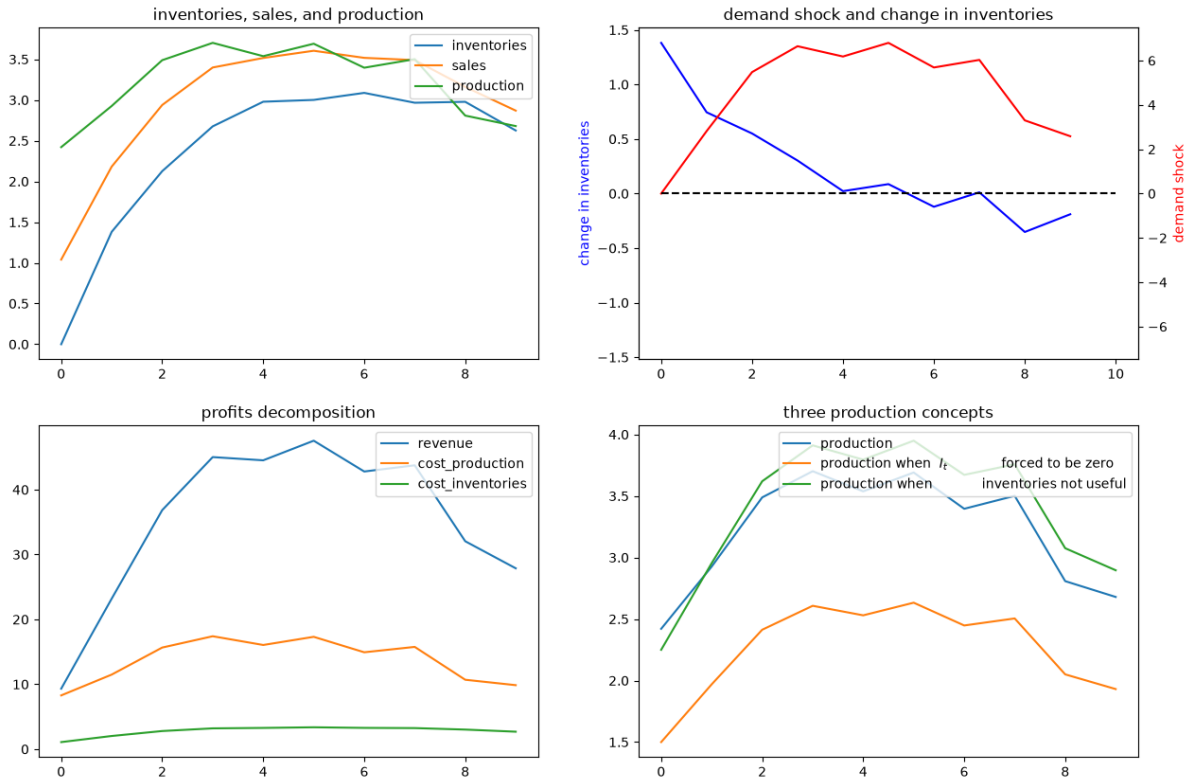
High optimal production in the original model early on occurs because the firm wants to accumulate inventories quickly in order to acquire high inventories for use in later periods.

But how the green line compares to the blue line early on depends on the evolution of the demand shock, as we will see in a deterministically seasonal demand shock example to be analyzed below.

In that example, the original firm optimally accumulates inventories slowly because the next positive demand shock is in the distant future.

To make the green-blue model production comparison easier to see, let's confine the graphs to the first 10 periods:

```
ex1.simulate(x0, T=10)
```



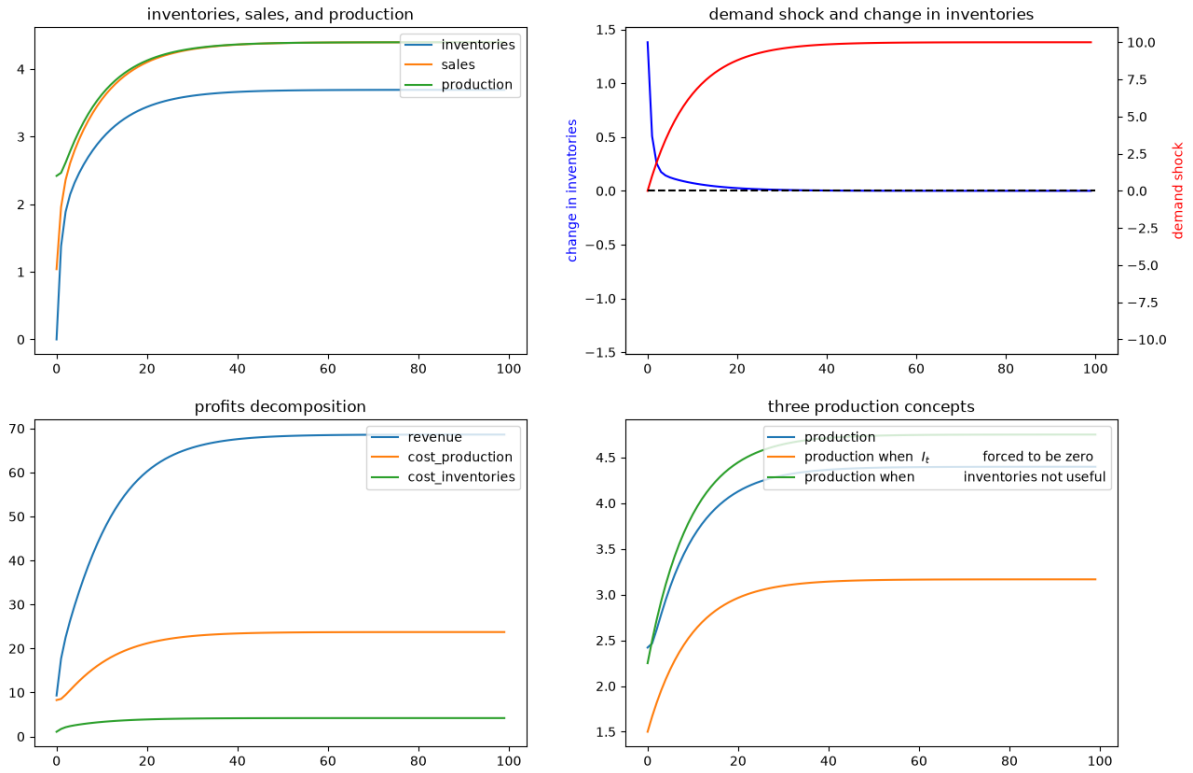
83.5 Example 2

Next, we shut down randomness in demand and assume that the demand shock ν_t follows a deterministic path:

$$\nu_t = \alpha + \rho \nu_{t-1}$$

Again, we'll compute and display outcomes in some figures

```
ex2 = SmoothingExample(C2=[[0], [0]])

x0 = [0, 1, 0]
ex2.simulate(x0)
```



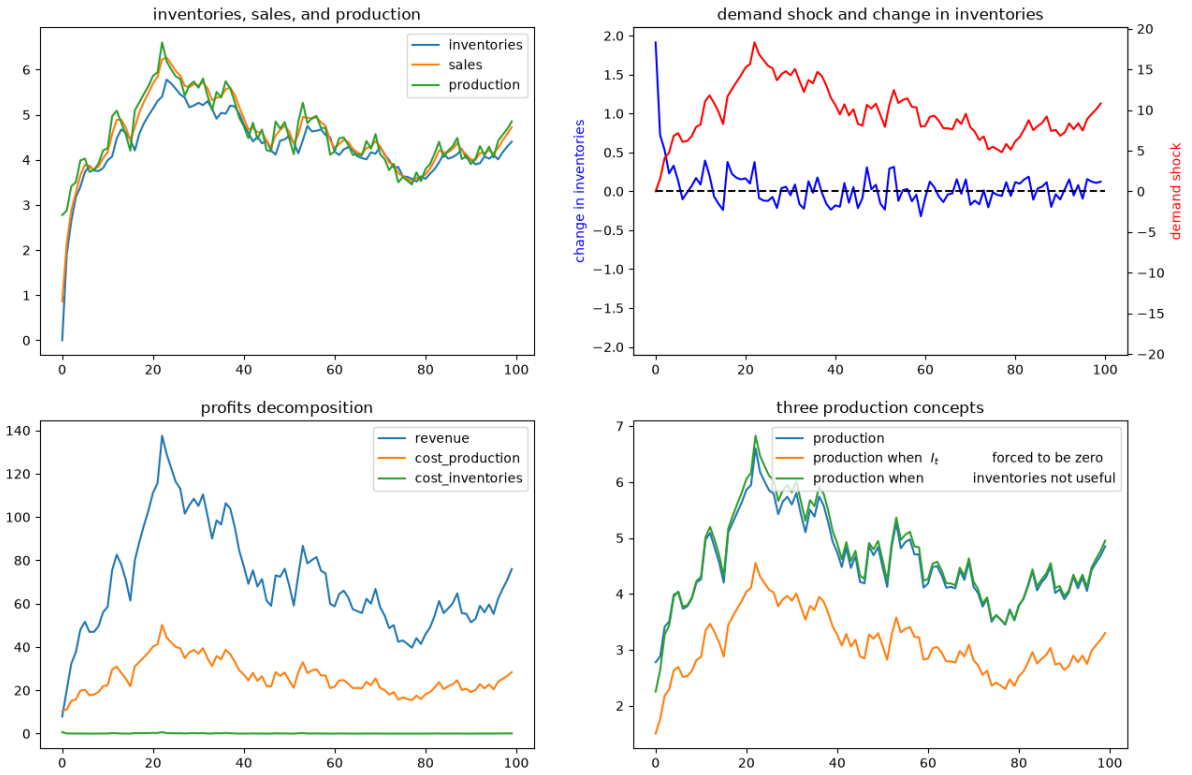
83.6 Example 3

Now we'll put randomness back into the demand shock process and also assume that there are zero costs of holding inventories.

In particular, we'll look at a situation in which $d_1 = 0$ but $d_2 > 0$.

Now it becomes optimal to set sales approximately equal to inventories and to use inventories to smooth production quite well, as the following figures confirm

```
ex3 = SmoothingExample(d1=0)

x0 = [0, 1, 0]
ex3.simulate(x0)
```



83.7 Example 4

To bring out some features of the optimal policy that are related to some technical issues in linear control theory, we'll now temporarily assume that it is costless to hold inventories.

When we completely shut down the cost of holding inventories by setting $d_1 = 0$ and $d_2 = 0$, something absurd happens (because the Bellman equation is opportunistic and very smart).

(Technically, we have set parameters that end up violating conditions needed to assure **stability** of the optimally controlled state.)

The firm finds it optimal to set $Q_t \equiv Q^* = \frac{-c_1}{2c_2}$, an output level that sets the costs of production to zero (when $c_1 > 0$, as it is with our default settings, then it is optimal to set production negative, whatever that means!).

Recall the law of motion for inventories

$$I_{t+1} = I_t + Q_t - S_t$$

So when $d_1 = d_2 = 0$ so that the firm finds it optimal to set $Q_t = \frac{-c_1}{2c_2}$ for all t, then

$$I_{t+1} - I_t = \frac{-c_1}{2c_2} - S_t < 0$$

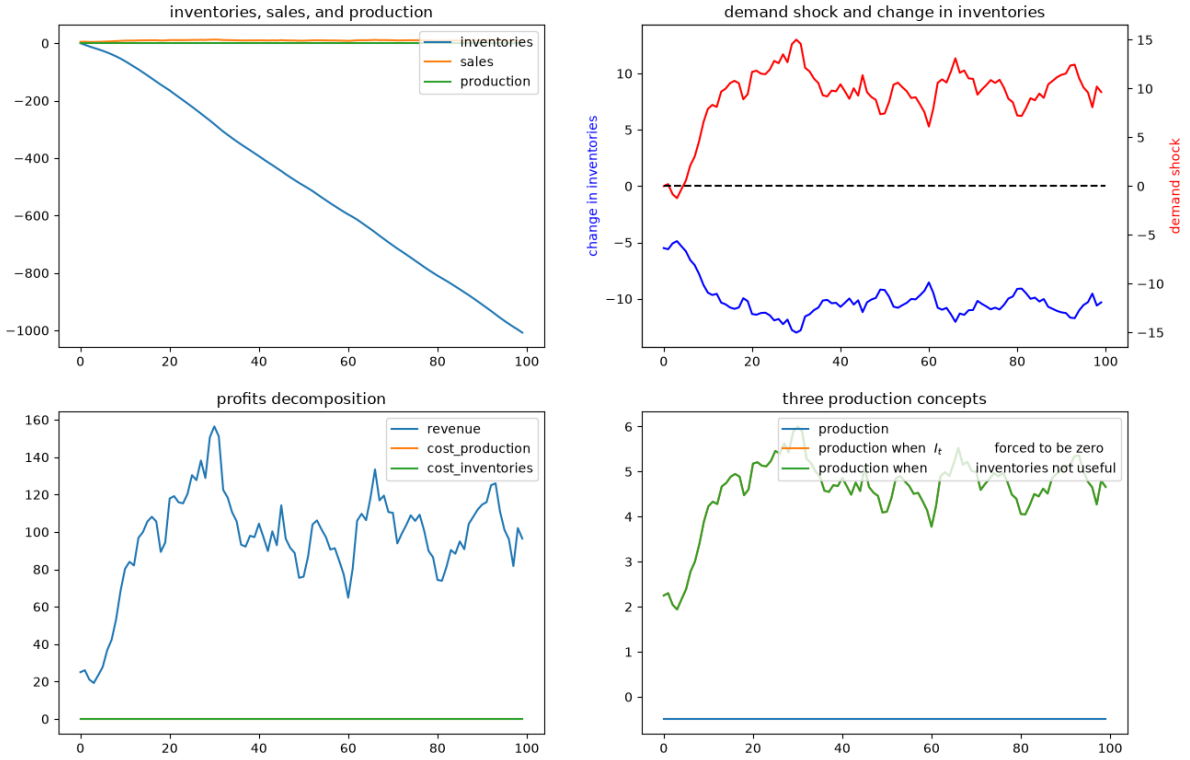
for almost all values of S_t under our default parameters that keep demand positive almost all of the time.

The dynamic program instructs the firm to set production costs to zero and to **run a Ponzi scheme** by running inventories down forever.

(We can interpret this as the firm somehow **going short in** or **borrowing** inventories)

The following figures confirm that inventories head south without limit

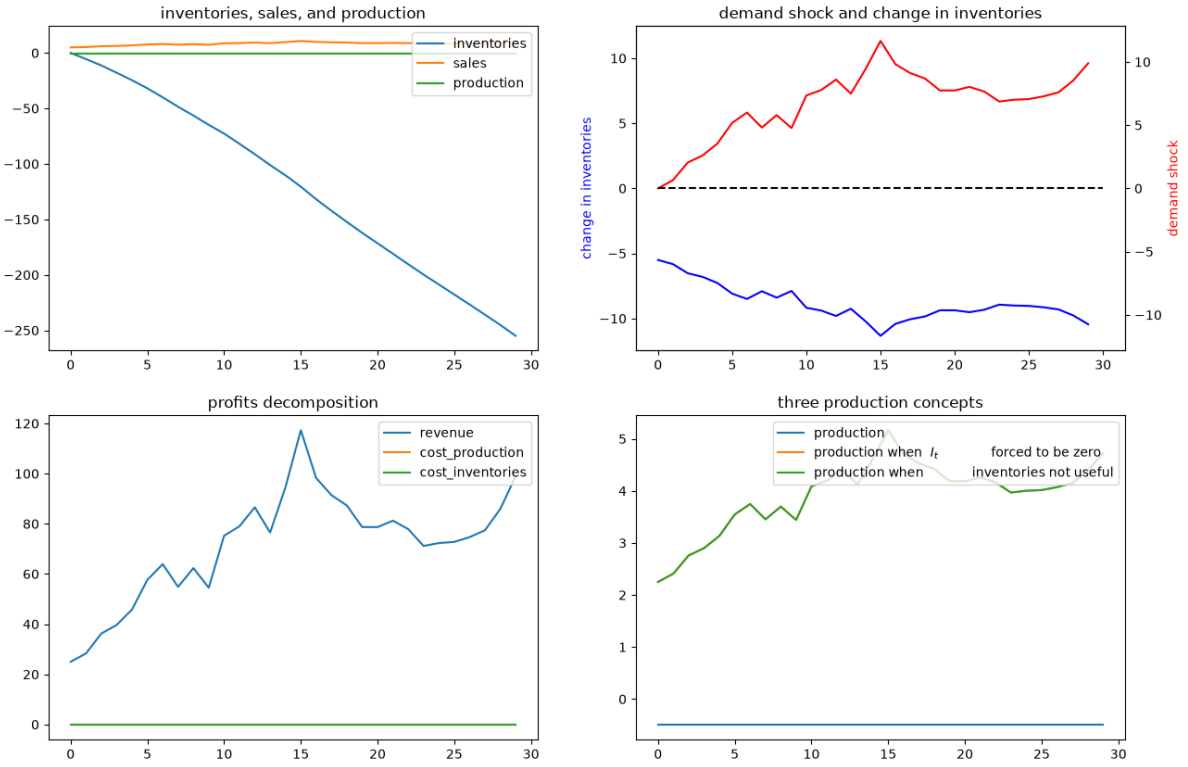
```
ex4 = SmoothingExample(d1=0, d2=0)

x0 = [0, 1, 0]
ex4.simulate(x0)
```



Let's shorten the time span displayed in order to highlight what is going on.

We'll set the horizon $T = 30$ with the following code

```
# shorter period
ex4.simulate(x0, T=30)
```



83.8 Example 5

Now we'll assume that the demand shock that follows a linear time trend

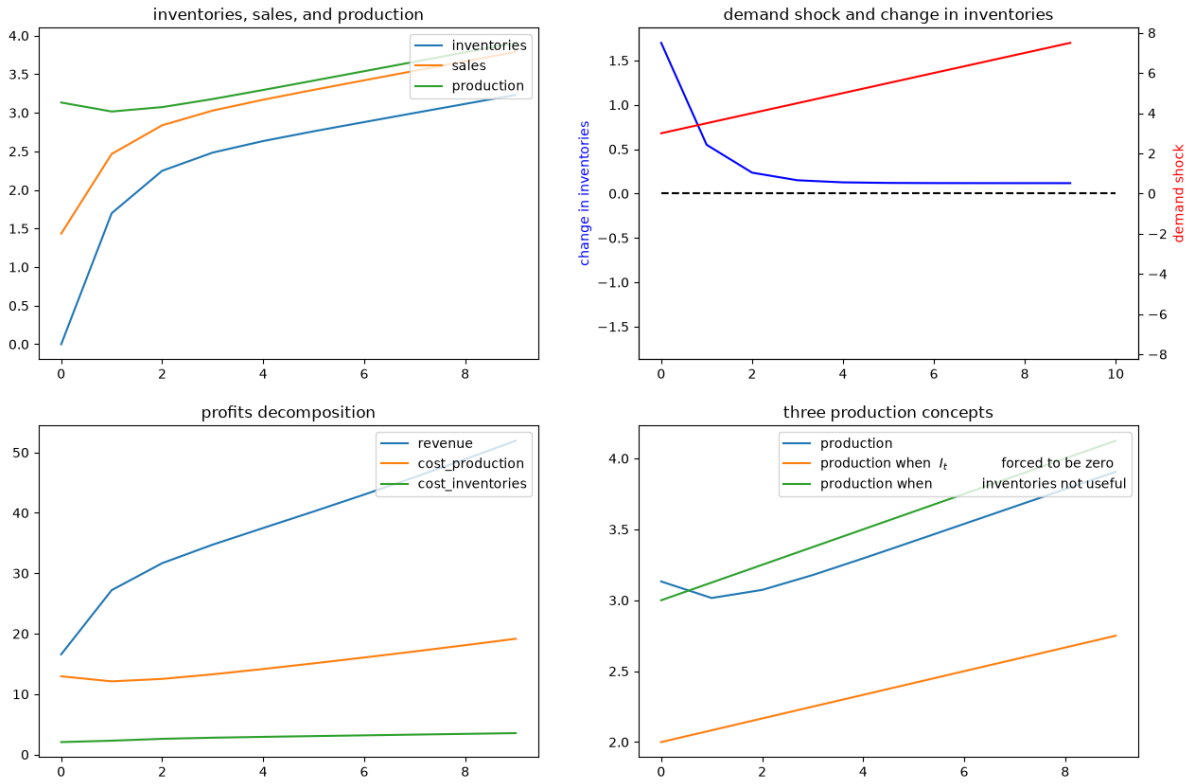
$$v_t = b + at, a > 0, b > 0$$

To represent this, we set $C_2 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$ and

$$A_{22} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}, x_0 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, G = \begin{bmatrix} b & a \end{bmatrix}$$

```
# Set parameters
a = 0.5
b = 3.
```

```
ex5 = SmoothingExample(A22=[[1, 0], [1, 1]], C2=[[0], [0]], G=[b, a])

x0 = [0, 1, 0] # set the initial inventory as 0
ex5.simulate(x0, T=10)
```



83.9 Example 6

Now we'll assume a deterministically seasonal demand shock.

To represent this we'll set

$$A_{22} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}, C_2 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, G' = \begin{bmatrix} b \\ a \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

where $a > 0, b > 0$ and

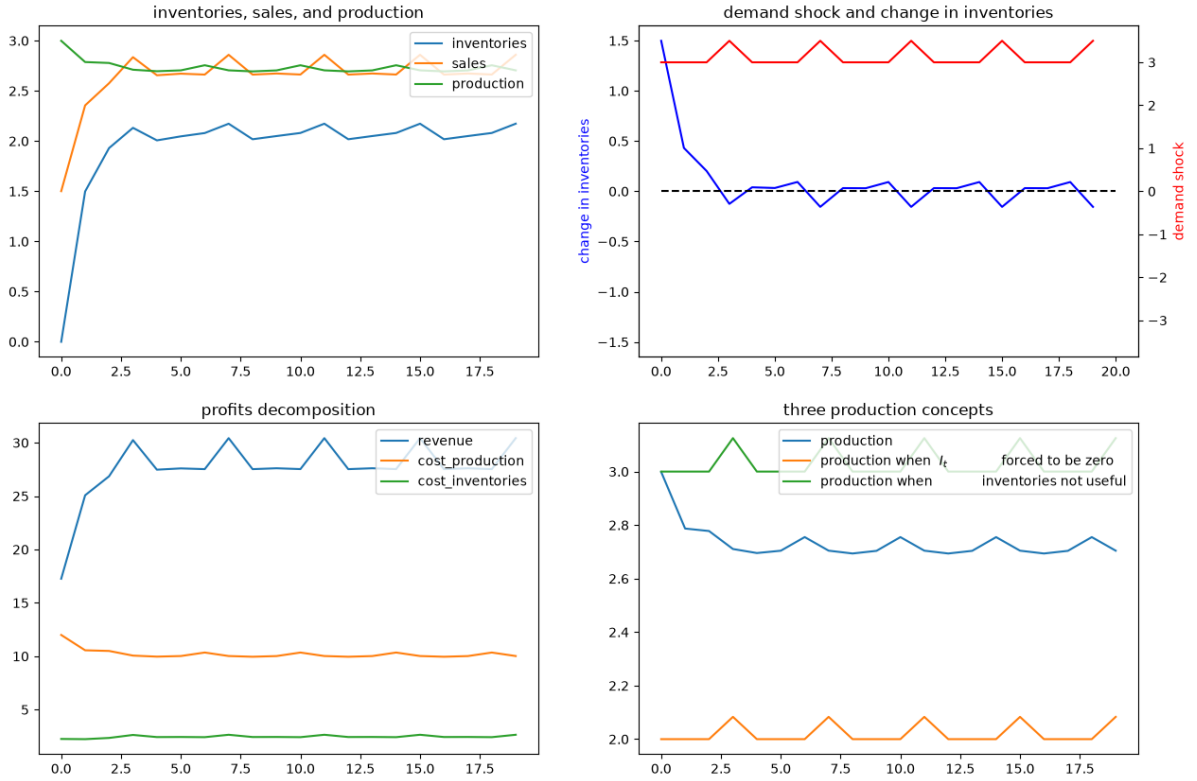
$$x_0 = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

```
ex6 = SmoothingExample(A22=[[1, 0, 0, 0, 0],
                           [0, 0, 0, 0, 1],
                           [0, 1, 0, 0, 0],
                           [0, 0, 1, 0, 0],
                           [0, 0, 0, 1, 0]],
                       C2=[[0], [0], [0], [0], [0]],
                       G=[b, a, 0, 0, 0])
```

(continues on next page)

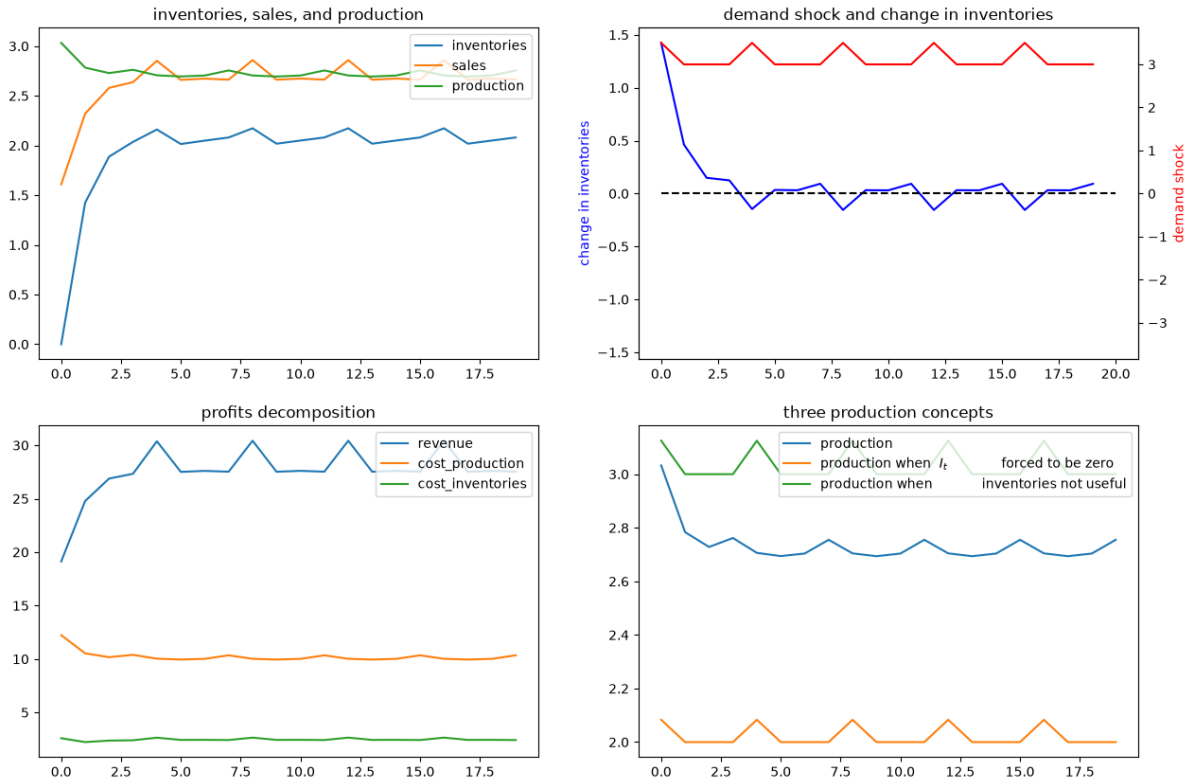
(continued from previous page)

```
x00 = [0, 1, 0, 1, 0, 0] # Set the initial inventory as 0
ex6.simulate(x00, T=20)
```

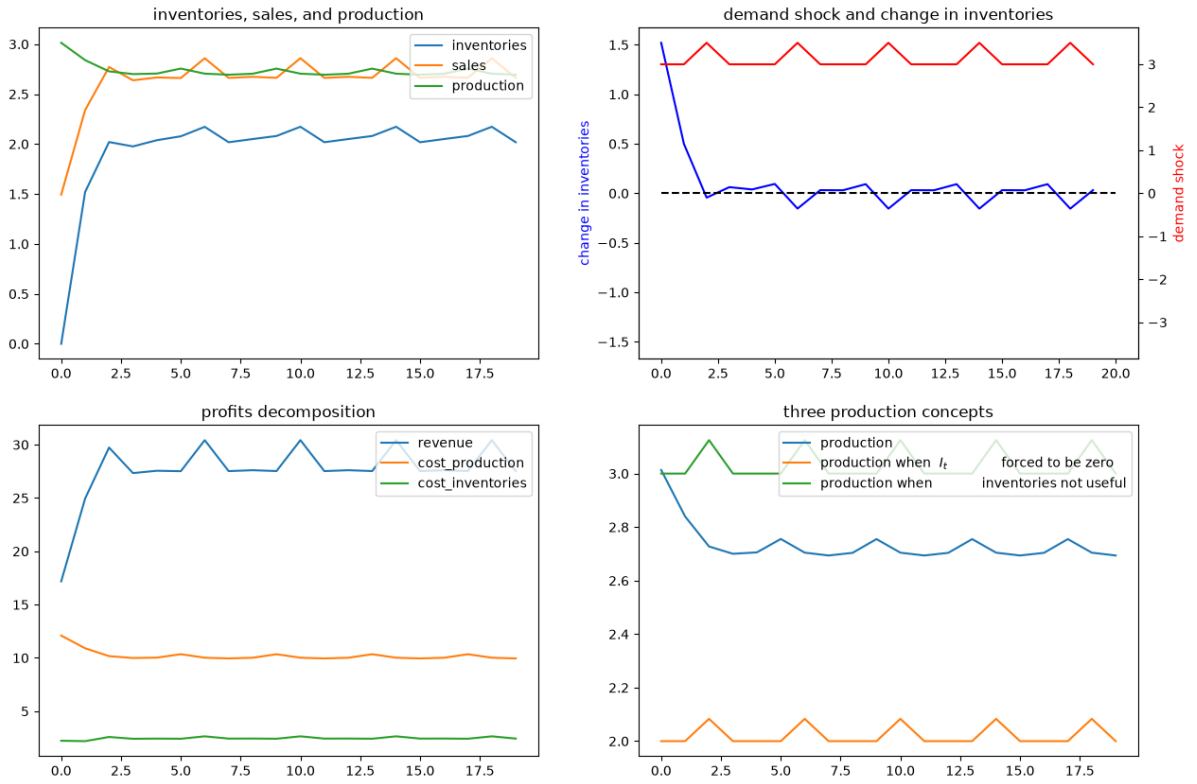


Now we'll generate some more examples that differ simply from the initial **season** of the year in which we begin the demand shock

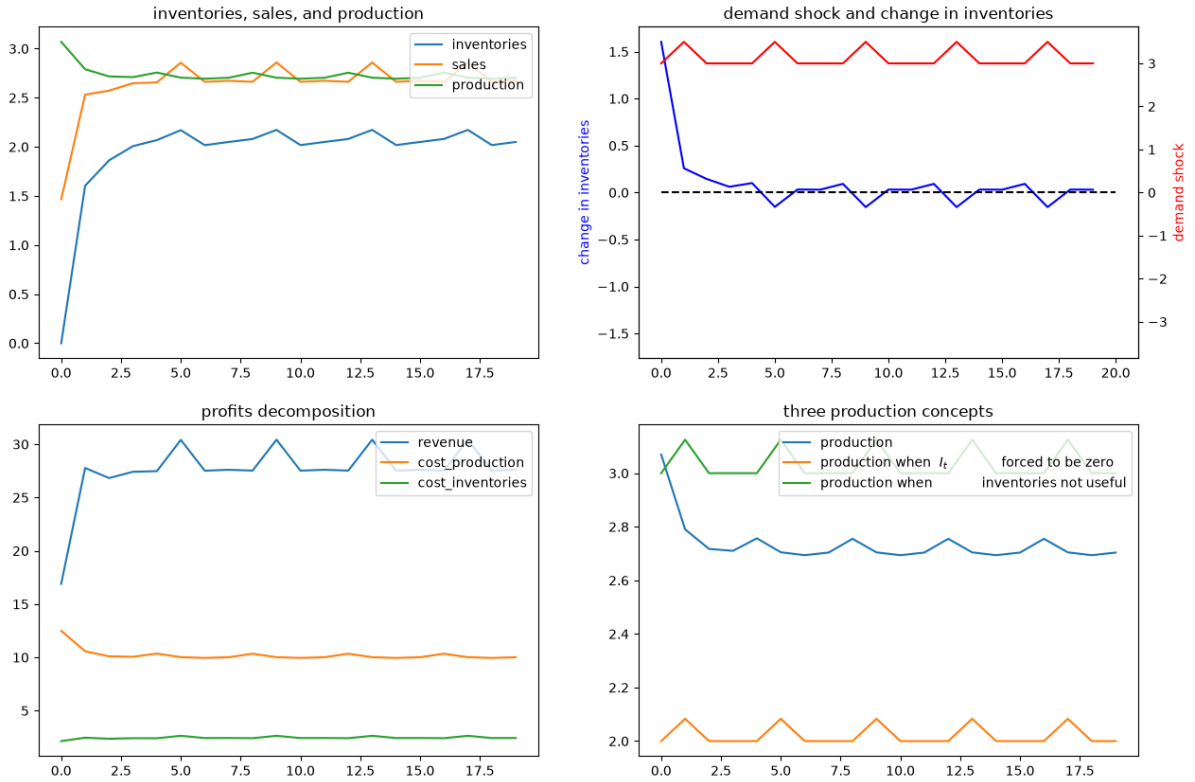
```
x01 = [0, 1, 1, 0, 0, 0]
ex6.simulate(x01, T=20)
```




```
x02 = [0, 1, 0, 0, 1, 0]
ex6.simulate(x02, T=20)
```



```
x03 = [0, 1, 0, 0, 0, 1]
ex6.simulate(x03, T=20)
```



83.10 Exercises

Please try to analyze some inventory sales smoothing problems using the `SmoothingExample` class.

Exercise 83.10.1

Assume that the demand shock follows AR(2) process below:

$$\nu_t = \alpha + \rho_1 \nu_{t-1} + \rho_2 \nu_{t-2} + \epsilon_t.$$

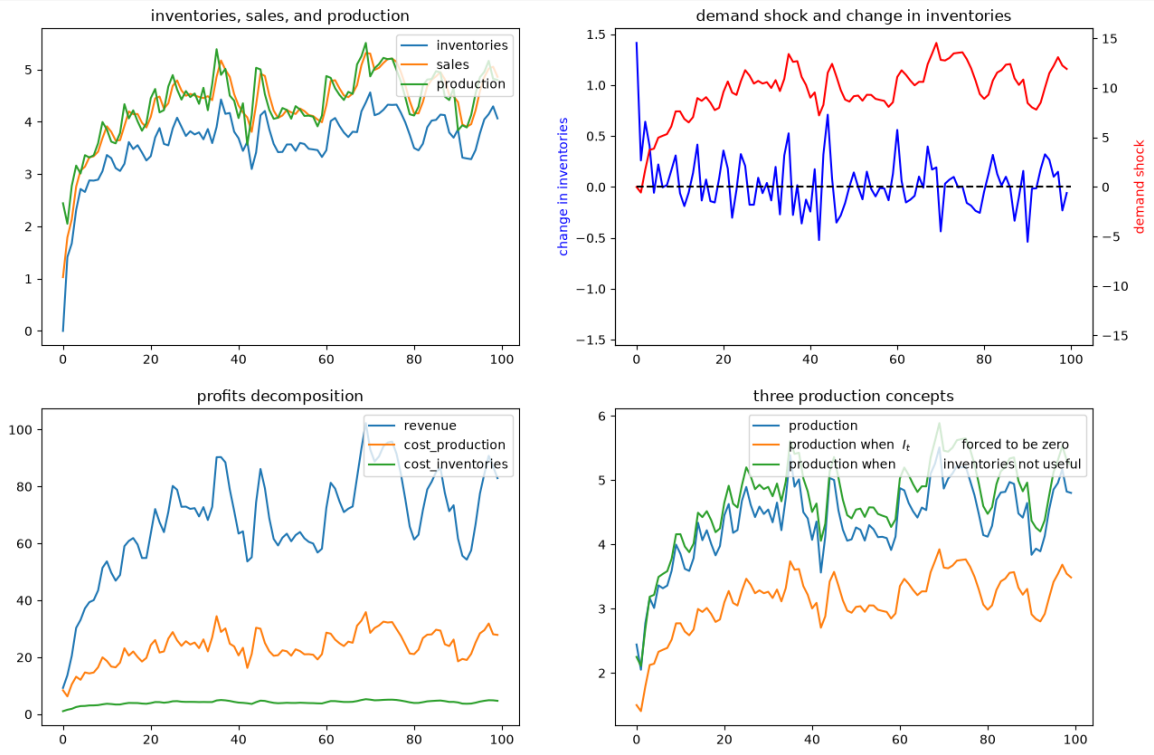
where $\alpha = 1$, $\rho_1 = 1.2$, and $\rho_2 = -0.3$. You need to construct $A22$, C, and G matrices properly and then to input them as the keyword arguments of `SmoothingExample` class. Simulate paths starting from the initial condition $x_0 = [0, 1, 0, 0]'$.

After this, try to construct a very similar `SmoothingExample` with the same demand shock process but exclude the randomness ϵ_t. Compute the stationary states $\bar{x}$ by simulating for a long period. Then try to add shocks with different magnitude to $\bar{\nu}_t$ and simulate paths. You should see how firms respond differently by staring at the production plans.

Solution

```
# set parameters
a = 1
```

```
ρ1 = 1.2
ρ2 = -.3
# construct matrices
A22 =[[1,  0,  0],
      [1, ρ1, ρ2],
      [0,  1,  0]]
C2 = [[0], [1], [0]]
G = [0, 1, 0]
ex1 = SmoothingExample(A22=A22, C2=C2, G=G)

x0 = [0, 1, 0, 0] # initial condition
ex1.simulate(x0)
```

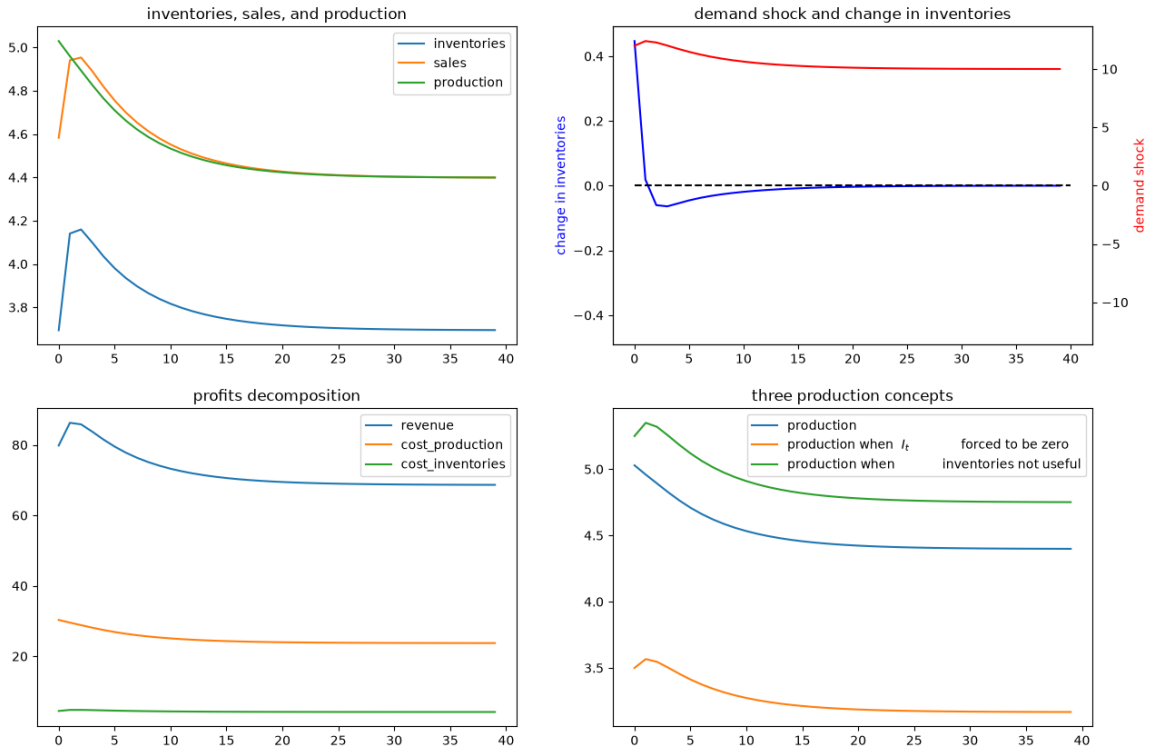



```
# now silence the noise
ex1_no_noise = SmoothingExample(A22=A22, C2=[[0], [0], [0]], G=G)

# initial condition
x0 = [0, 1, 0, 0]

# compute stationary states
x_bar = ex1_no_noise.LQ.compute_sequence(x0, ts_length=250)[0][:, -1]
x_bar
```

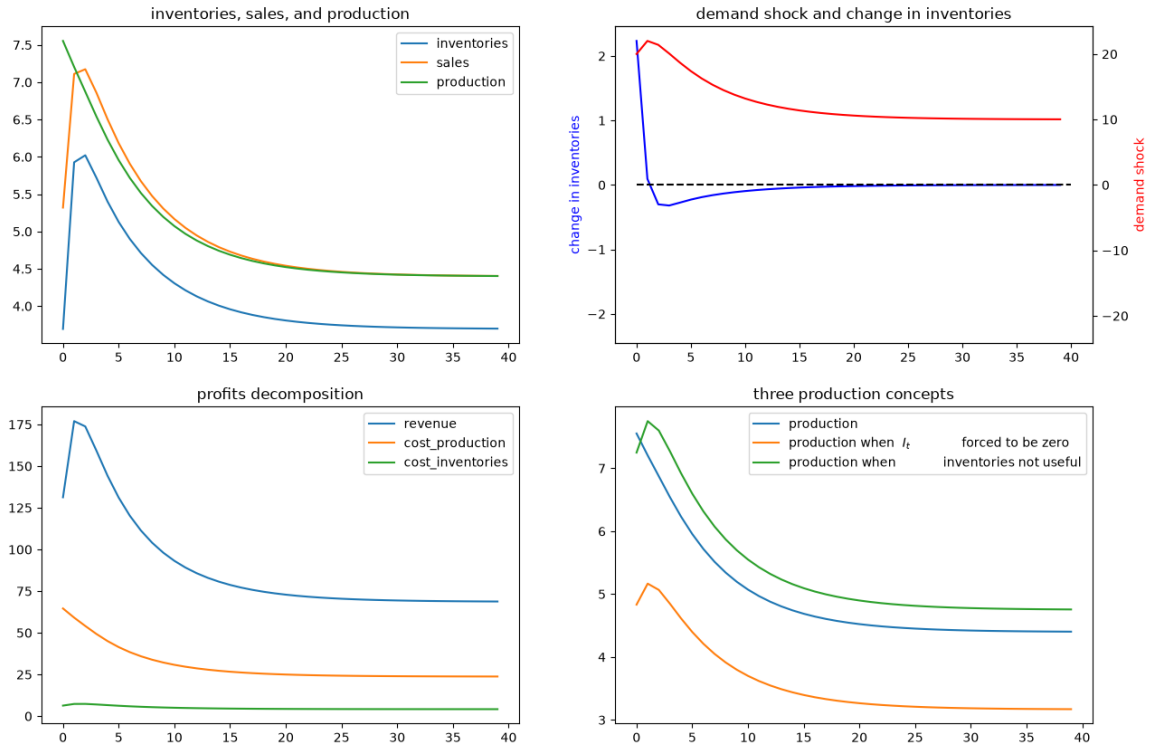
```
array([ 3.69387755,  1.        , 10.        , 10.        ])
```

In the following, we add small and large shocks to $\bar{\nu}_t$ and compare how firm responds differently in quantity. As the shock is not very persistent under the parameterization we are using, we focus on a short period response.

```
T = 40
# small shock
x_bar1 = x_bar.copy()
x_bar1[2] += 2
ex1_no_noise.simulate(x_bar1, T=T)
```




```
# large shock
x_bar1 = x_bar.copy()
x_bar1[2] += 10
ex1_no_noise.simulate(x_bar1, T=T)
```



Exercise 83.10.2

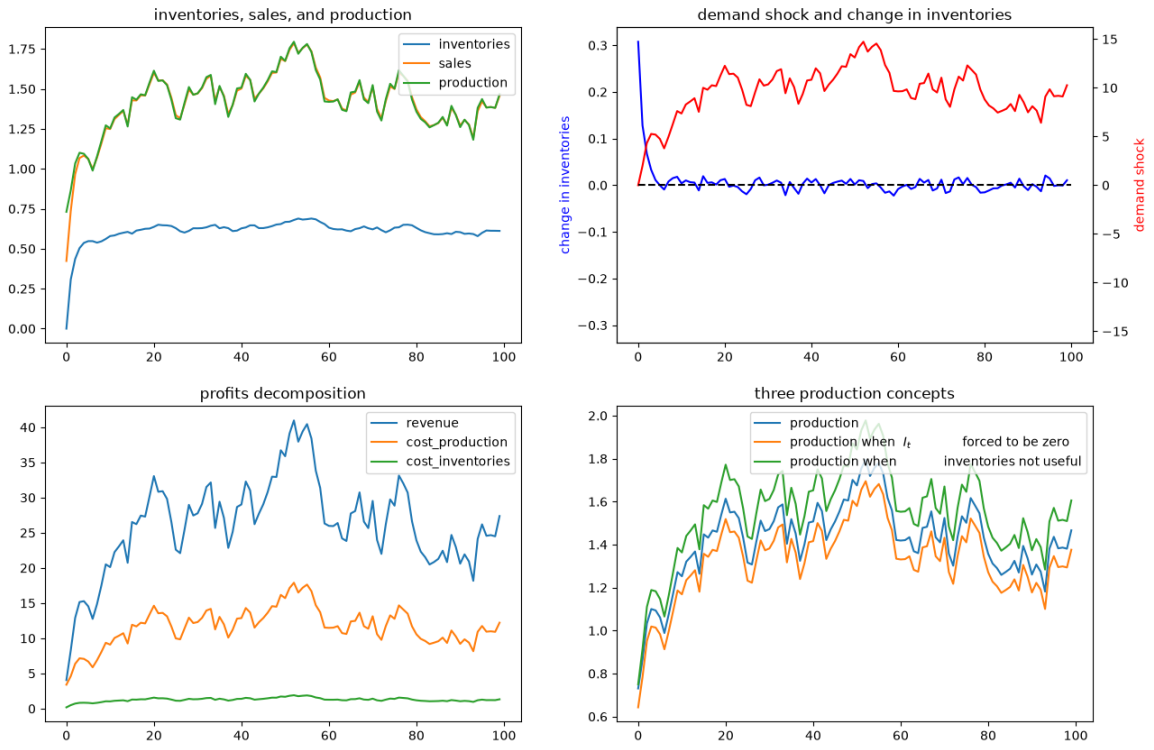
Change parameters of $C(Q_t)$ and $d(I_t, S_t)$.

1. Make production more costly, by setting $c_2 = 5$.
2. Increase the cost of having inventories deviate from sales, by setting $d_2 = 5$.

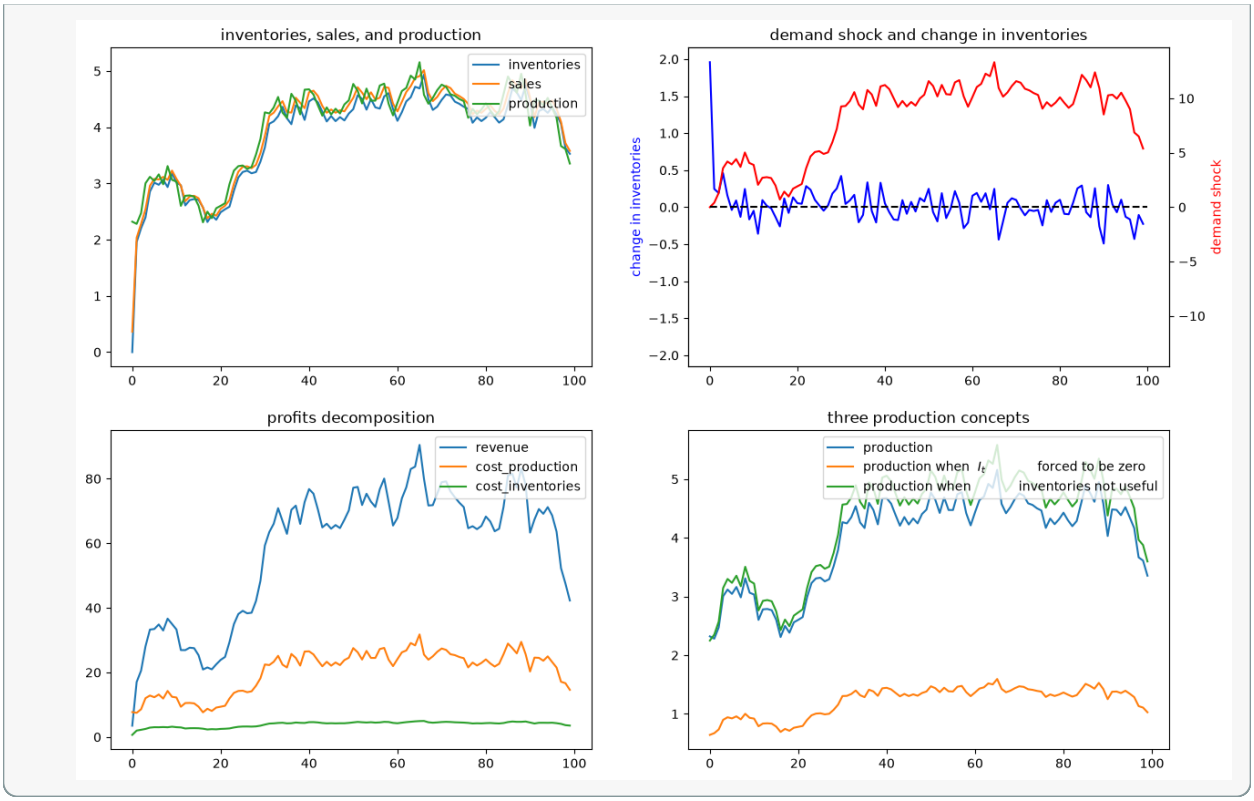
Solution

```
x0 = [0, 1, 0]
```

```
SmoothingExample(c2=5).simulate(x0)
```




```
SmoothingExample(d2=5).simulate(x0)
```


inventories, sales, and production
inventories
sales
production
demand shock and change in inventories
change in inventories
demand shock
profits decomposition
revenue
cost_production
cost_inventories
three production concepts
production
production when I_t
forced to be zero
production when
inventories not useful

Part XII

Bounded Rationality in Macroeconomics

CHAPTER

EIGHTYFOUR

A PECULIAR DEFINITION OF BOUNDED RATIONALITY

Contents

- *A Peculiar Definition of Bounded Rationality*
  - *Overview*
  - *Rational expectations as a fixed point*
  - *Money and prices*
  - *Two currencies*
  - *Where this leaves us*
  - *Exercises*

84.1 Overview

This lecture opens a series based on Sargent [1993] about Bounded Rationality in Macroeconomics, from one point of view in the late 1980s and early 1990s.

Those were years when the Soviet Union ended and formerly Warsaw Pact countries were rearranging their political economies.

For those countries, those were thus truly *regime changes* in the technical sense of modern dynamic macroeconomics.

Two generations of work on economic dynamics — in game theory, macroeconomics, and general equilibrium theory — had produced theories that embraced the rational expectations assumption, models that were designed to understand settings in which people face recurrent situations they have lived through many times before.

The transitions across regimes underway in Eastern Europe in the early 1990s were not like that.

People there were confronted with unprecedented opportunities, new and ill-defined rules, and a daily struggle to figure out the mechanism that would eventually govern trade and production.

Economists with good models of a market economy had ample *equilibrium theories* describing how a system behaves once it has fully adjusted to a new and coherent set of rules and expectations.

They knew much less about the dynamics of *transitions* from a Soviet system to a market economy.

They might hold prejudices and anecdotes about how to manage such a transition, but no empirically confirmed formal theory of it.

Against this background, some economists ventured into what Sims [1980] called the "wilderness" of irrational expectations and bounded rationality.

The aim was partly to build theories of transition dynamics, partly to understand the properties of equilibrium dynamics themselves, and partly to study systems that never settle down.

This series follows Sargent [1993] a little way into that wilderness.

84.1.1 The knowledge that rational expectations imputes

To see what bounded rationality retreats *from*, start with rational expectations, which imposes **two** requirements:

1. **Individual rationality** — each artificial agent's behavior maximizes an objective function subject to perceived constraints.
2. **Mutual consistency** — the constraints perceived by everybody in the system agree with one another.

The second requirement is Muth's *rational expectations* assumption.

In an economy one person's decisions are part of another person's constraints, so consistency requires each person to hold correct beliefs about everyone else's decisions, decision processes, and beliefs.

Consistency is also what gives rational expectations its power: without some restriction on perceptions, a model in which behavior depends on arbitrary assumptions about subjective beliefs can produce almost any outcome at all.

But look at what that requirement imputes to people once a model is taken to data.

The agents inside a rational expectations model evaluate their Euler equations using *equilibrium* probability distributions.

Those are the very distributions that the econometrician studying them is still struggling to estimate.

The agents, in other words, have somehow already solved the inference problem that the economist is only part way through.

84.1.2 Sargent's formulation of bounded rationality

Sargent's **bounded rationality** program keeps individual rationality and retreats from mutual consistency in a particular way that was motivated by his love of time series econometrics.

Sargent's version of a bounded rationality program is:

> I interpret a proposal to build models with 'boundedly rational' agents as a call to retreat from the second piece of rational expectations (mutual consistency of perceptions) by expelling rational agents from our model environments and replacing them with 'artificially intelligent' agents who behave like econometricians. These 'econometricians' theorize, estimate, and adapt in attempting to learn about probability distributions which, under rational expectations, they already know.

The agents are made more like the people who build the models: they gather data, form theories, estimate, and adapt.

Note

After he saw Sargent's manuscript, Carnegie-Mellon's Herbert Simon wrote Sargent a letter saying that he objected to Sargent's formulation and recommended that Sargent not call what he was doing "bounded rationality".

Simon particularly disliked Sargent's making the agents inside his model act like econometricians, a pretense that Simon found preposterous.

Sargent's proposal makes work harder for the model builder, not easier.

Withdrawing the assumption of a commonly understood environment means we must put something in its place, and there are many plausible somethings to choose among:

> This area is wilderness because the researcher faces so many choices after he decides to forgo the discipline provided by equilibrium theorizing. The commitment to equilibrium theorizing made many choices for him by requiring that people be modelled as optimal decision-makers within a commonly understood environment. When we withdraw the assumption of a commonly understood environment, we have to replace it with something, and there are so many plausible possibilities.

84.1.3 What the program is good for

The payoffs the book pursues are of three kinds.

Sometimes a collection of adaptive agents learns to behave *as if* it had rational expectations, which lends the equilibrium a plausibility it lacked as a mere assumption.

Sometimes adaptive agents converge to a *particular* equilibrium among many, turning learning into a device for **selecting** among rational expectations equilibria, and, relatedly, into a way of **computing** equilibria too complicated to solve by hand.

And, more ambitiously, adaptive dynamics hold out the hope of a theory of the *transition* itself, the out-of-equilibrium adjustment that the Eastern European reformers had to manage blind.

That last promise is the least fulfilled, as the series will acknowledge; the selection and computation payoffs are the surer ones.

This lecture takes up selection, through the sharpest case: models with **too many equilibria**.

When a rational expectations model has a continuum of equilibria, the physical description of the economy plus the equilibrium concept fail to pin down what happens.

Something else must choose, and a plausible account of how people grope toward equilibrium is a natural candidate for that something.

We build the two monetary examples that the rest of the series returns to repeatedly, both with a continuum of rational expectations equilibria:

- a quantity-theory model of money and prices in which the price level is determined only up to an arbitrary **bubble** term, and
- a two-currency version in which the *exchange rate is completely unrestricted*.

Along the way we set up the machinery — the fixed-point view of equilibrium, the relaxation algorithm, adaptive expectations, and Muth's inverse optimal-prediction problem — that later lectures use to expel the rational agents and put adaptive ones in their place.

84.1.4 The rest of the series

- *Adaptive Agents in an Overlapping Generations Monetary Economy* — adaptive households in Samuelson's overlapping generations monetary model.
  - Least squares learning selects the low-inflation equilibrium that the rational expectations dynamics reject, and laboratory subjects go the same way.
  - A government learning a Phillips curve closes the lecture.
- *Learning, Approximation, and Equilibrium Computation* — what an agent must give up when the state is continuous and a separate response for each contingency is out of the question, which brings in approximate equilibria and the resemblance between learning algorithms and equilibrium computation algorithms.

- *Exchange Rate Indeterminacy, Learning, and Experiments* — the two-currency model of this lecture with adaptive agents, where learning pins the exchange rate down only by making it depend on history, while a genetic-algorithm economy instead produces volatility that never dies.
- *Genetic Algorithms and Classifier Systems* — a catalogue of candidate brains, from Holland and the connectionists: perceptrons, associative memories, genetic algorithms, classifier systems.
- *Money as a Medium of Exchange among Artificially Intelligent Agents* — populations of classifier systems that discover, from scratch, which good will serve as money.
- *1993 Prospects for Bounded Rationality in Macroeconomics* — the 1993 ledger of what the program achieved and what it did not, and a postscript on the three decades since.

Let's start with some imports.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

84.2 Rational expectations as a fixed point

84.2.1 A static market

Take a competitive industry with a large number n of identical firms.

Each firm chooses output x to maximize

$$R(x, p) = px - c(x),$$

where c is an increasing, convex cost function, and price is determined by a downward-sloping inverse demand curve

$$p = p(nX)$$

in which X is the output of the *average* firm.

Each firm is a price-taker and an X-taker: it takes p as given and sets marginal cost equal to price, $p = c'(x)$.

Write the solution as $x = g(p)$.

Substituting the demand curve gives the **best-response map**

$$x = g(p(nX)) \equiv h(X), \tag{84.1}$$

which sends a conjectured industry average X to the individual output an optimizing firm would choose against it.

That is the first component of rational expectations, and it is all that individual rationality delivers.

The second component — consistency — requires that what each firm chooses coincides with what it assumed the average firm was choosing:

$$X = h(X). \tag{84.2}$$

A static rational expectations equilibrium is a **fixed point of the best-response map**.

For a concrete example, take a quadratic cost function $c(x) = \frac{\gamma}{2}x^2$ and a linear inverse demand curve $p = a - bnX$.

Then $p = c'(x) = \gamma x$ gives $x = p/\gamma$, so

$$h(X) = \frac{a - bnX}{\gamma}, \qquad \text{with fixed point} \qquad X^* = \frac{a}{\gamma + bn}.$$

```
a, b, n, γ = 10.0, 0.3, 5, 1.0

def h(X):
    "Best response of an individual firm to an industry average X."
    return (a - b * n * X) / γ

X_star = a / (γ + b * n)
print(f"X*     = {X_star}")
print(f"h(X*)  = {h(X_star)}")
print(f"h'(X)  = {-b * n / γ}")
```

```
X*     = 4.0
h(X*)  = 4.0
h'(X)  = -1.5
```

84.2.2 Computing the fixed point

Now suppose we want to *find* X^* without solving for it in closed form.

A natural starting point is the **relaxation algorithm**: carry an estimate X_k^* of the equilibrium, compute the best response to it, and move part of the way toward that best response,

$$X_k^* = X_{k-1}^* + \lambda(h(X_{k-1}^*) - X_{k-1}^*), \tag{84.3}$$

where $\lambda \in (0, 1]$ is a **relaxation parameter**.

With $\lambda = 1$ this is simple iteration on h, the classic cobweb.

```
def relax(λ, X0=1.0, n_iter=12):
    "Iterate the relaxation algorithm, returning the whole path."
    path = np.empty(n_iter + 1)
    path[0] = X0
    for k in range(n_iter):
        path[k + 1] = path[k] + λ * (h(path[k]) - path[k])
    return path


fig, axes = plt.subplots(1, 2, figsize=(11, 4))

axes[0].plot(relax(1.0), 'o-', ms=4, lw=1, color='C0',
             label=r"$\lambda = 1.0$ (cobweb)")
axes[0].set_title("undamped")
axes[0].set_ylabel("$X^*_k$")

for i, λ in enumerate((0.8, 0.5, 0.3)):
    axes[1].plot(relax(λ), 'o-', ms=4, lw=1, color=f'C{i+1}',
                 label=fr"$\lambda = {λ}$")
axes[1].set_title("damped")
axes[1].set_ylim(-1, 9)

for ax in axes:
    ax.axhline(X_star, color='k', lw=0.8, ls='--', label="$X^*$")
    ax.set_xlabel("$k$")
    ax.legend(frameon=False, fontsize=9)
plt.tight_layout()
plt.show()
```

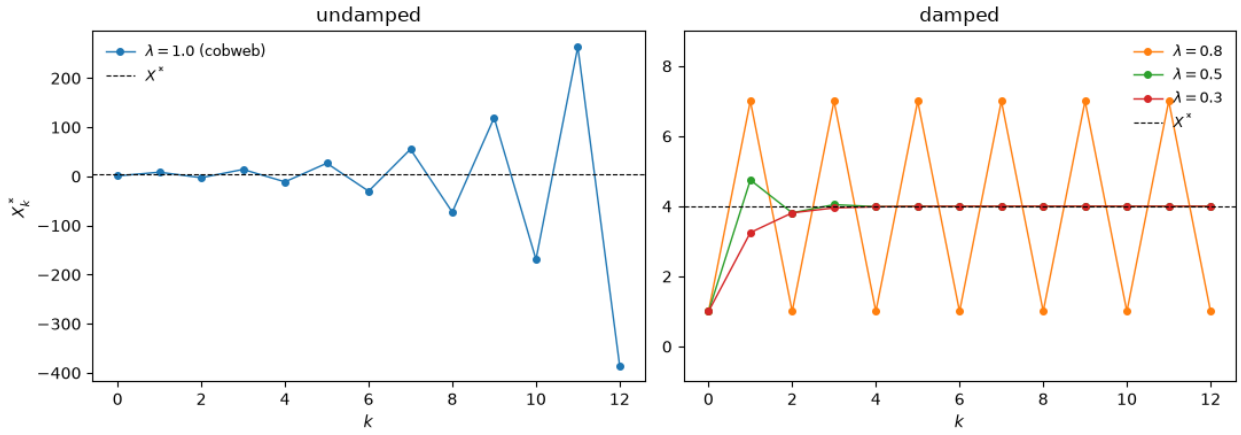


Fig. 84.1: Relaxation algorithm paths, undamped and damped

The naive cobweb $\lambda = 1$ **diverges**, oscillating ever further away from the equilibrium it is trying to find; note the scale on the left panel.

Since h is affine, iteration (84.3) has multiplier $1 + \lambda(h' - 1)$, so it converges if and only if

$$\left|1 + \lambda(h' - 1)\right| < 1 \qquad \iff \qquad \lambda < \frac{2}{1 + bn/\gamma}.$$

```
λ_max = 2 / (1 + b * n / γ)
print(f"converges iff λ < {λ_max}")
print(f"λ = 1.0 : multiplier = {1 + 1.0 * (-b*n/γ - 1):+.2f}   (diverges)")
print(f"λ = 0.8 : multiplier = {1 + 0.8 * (-b*n/γ - 1):+.2f}   (period-2 cycle)")
print(f"λ = 0.5 : multiplier = {1 + 0.5 * (-b*n/γ - 1):+.2f}   (converges)")
```

```
converges iff λ < 0.8
λ = 1.0 : multiplier = -1.50   (diverges)
λ = 0.8 : multiplier = -1.00   (period-2 cycle)
λ = 0.5 : multiplier = -0.25   (converges)
```

At $\lambda = 0.8$ the multiplier is exactly -1: the scheme maps $X \mapsto 8 - X$ and so bounces forever between 1 and 7 without either converging or diverging, which is the undamped zigzag in the right panel.

Damping the adjustment enough makes the algorithm find the equilibrium; not damping it enough makes the algorithm chase its own tail.

Hold on to that observation.

A recurring theme of this series is that *how* you adapt determines where, and whether, you end up.

84.2.3 Adaptive expectations

Equation (84.3) was introduced as an algorithm running in iteration count k.

Reinterpret k as **calendar time** t, read X_t^* as the value people *expect*, and $X_t = h(X_{t-1}^*)$ as what actually happens, and it becomes a theory of expectation formation:

$$X_t^* = (1-\lambda)X_{t-1}^* + \lambda X_t = \lambda \sum_{j=0}^{\infty} (1-\lambda)^j X_{t-j}. \tag{84.4}$$

This is the **adaptive expectations** scheme that Cagan [Cagan, 1956] used to study hyperinflations and that Friedman used to study consumption.

Expectations are a geometrically declining distributed lag of past observations, with a single free parameter λ describing beliefs.

Cagan and Friedman took λ as a free parameter and left open the question of *why* anyone would form expectations this way.

84.2.4 Muth's inverse problem

Muth [1960] set out to eliminate λ as a free parameter by turning the question around.

Instead of asking what forecasts a given environment implies, he asked: **for what environment would exponential smoothing be the optimal forecast?**

His answer is that (84.4) is the least-squares forecast of X_{t+k} at every horizon k if and only if X_t follows

$$X_t = X_{t-1} + \epsilon_t - \theta \epsilon_{t-1}, \tag{84.5}$$

with $\{\epsilon_t\}$ a martingale difference sequence, and the smoothing weight tied to the moving-average coefficient by

$$\lambda = 1 - \theta.$$

Note

Sargent writes both the smoothing weight in (84.4) and the moving-average coefficient in (84.5) as λ.

We use θ for the second to keep the relationship $\lambda = 1 - \theta$ visible.

Two other symbols work double shifts in this lecture, both following the book.

In the money model below, λ is no longer a gain but the gross growth rate w_1/w_2 of the bubble term, and γ is no longer the curvature of a cost function but the coefficient linking the price level to the money supply.

The code distinguishes them as `λ_m` and `γ_m`.

The forecasting rule inherits its one parameter from the stochastic process being forecast.

Let's confirm this numerically: simulate (84.5), run exponential smoothing at a range of weights, and see which weight forecasts best.

```
def smoothing_mse(x, λ):
    "One-step-ahead forecast MSE of exponential smoothing with weight λ."
    f = np.empty_like(x)
    f[0] = x[0]
```

(continues on next page)

(continued from previous page)

```
    for i in range(1, len(x)):
        f[i] = λ * x[i] + (1 - λ) * f[i - 1]
    return np.mean((x[1:] - f[:-1]) ** 2)


rng = np.random.default_rng(0)
T = 200_000
ε = rng.standard_normal(T + 1)            # one shock path, shared across all λ
grid = np.linspace(0.02, 0.98, 97)

fig, ax = plt.subplots(figsize=(7, 4))
for θ in (0.2, 0.5, 0.8):
    X = np.cumsum(ε[1:] - θ * ε[:-1])     # Muth's process
    mse = np.array([smoothing_mse(X, λ) for λ in grid])
    line, = ax.plot(grid, mse, lw=1.2, label=fr"$\theta = {θ}$")
    ax.axvline(1 - θ, color=line.get_color(), lw=0.8, ls='--')
    print(f"θ = {θ}:  best λ = {grid[mse.argmin()]:.2f},  1 - θ = {1 - θ:.2f},"
          f"  min MSE = {mse.min():.4f}")
ax.set_xlabel(r"smoothing weight $\lambda$")
ax.set_ylabel("forecast MSE")
ax.set_ylim(0.9, 3)
ax.legend(frameon=False)
plt.show()
```

```
θ = 0.2:  best λ = 0.80,  1 - θ = 0.80,  min MSE = 1.0024
θ = 0.5:  best λ = 0.50,  1 - θ = 0.50,  min MSE = 1.0024
θ = 0.8:  best λ = 0.20,  1 - θ = 0.20,  min MSE = 1.0024
```

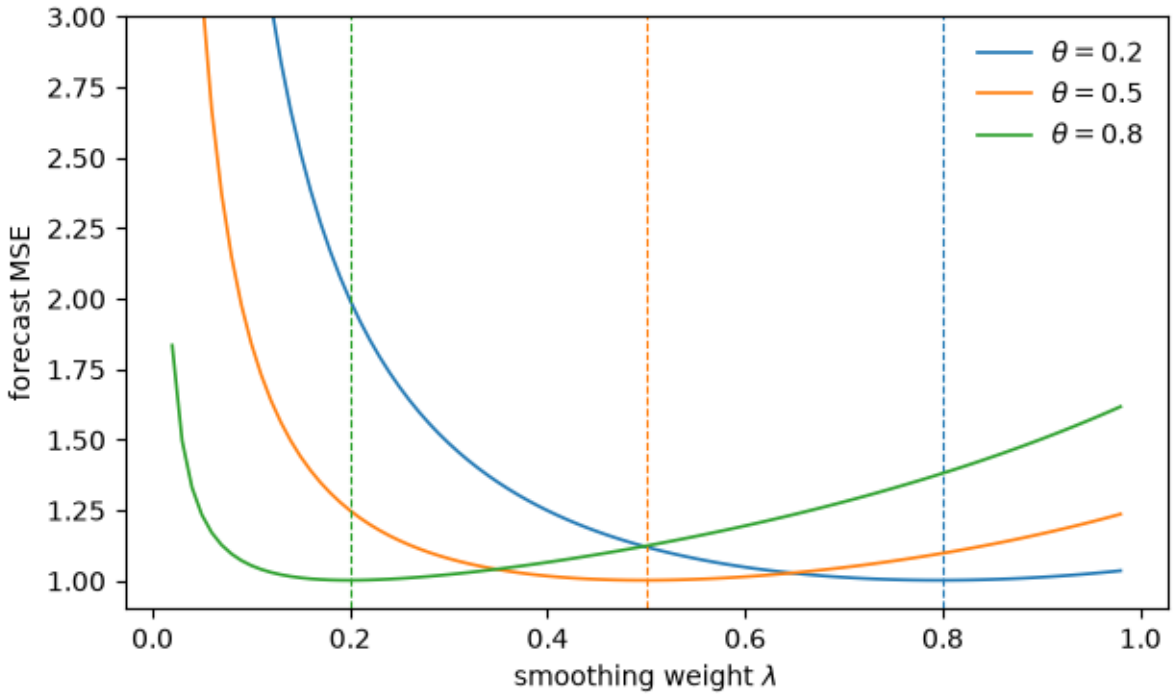


Fig. 84.2: Forecast MSE of exponential smoothing

Each curve bottoms out exactly at its dashed line $\lambda = 1 - \theta$, and the minimized mean squared error is $\mathbb{V}[\epsilon_t] = 1$ up to

simulation noise — exponential smoothing at that weight *is* the conditional expectation, and nothing can beat it.

Muth's exercise was the first application of the rational expectations idea in the form that became standard: find the restrictions that link a forecasting scheme to the environment in which it is used.

It also pushed later researchers to treat the **forecasting scheme itself** as the object in terms of which equilibrium is defined.

84.2.5 The dynamic analogue

That is exactly what happens when we move from static to dynamic models.

Let each agent choose a sequence rather than a single action, taking as given a **perceived law of motion** for the aggregate state,

$$X_t = H(X_{t-1}, u_t),$$

where $\{u_t\}$ is IID

Solving the agent's dynamic program yields an individual decision rule $x_t = h(x_{t-1}, X_{t-1}, u_t)$.

Imposing that the representative agent is representative, $x_t = X_t$, delivers the **actual law of motion**

$$X_t = h(X_{t-1}, X_{t-1}, u_t) \equiv H^*(X_{t-1}, u_t),$$

and hence a map from perceived to actual laws of motion,

$$H^* = T(H). \tag{84.6}$$

A dynamic rational expectations equilibrium is a fixed point $H = T(H)$, the same idea as (84.2), but the fixed point now lives in a space of *functions* rather than a space of numbers.

The relaxation algorithm carries over unchanged,

$$H^*_k = H^*_{k-1} + \lambda(T(H^*_{k-1}) - H^*_{k-1}), \tag{84.7}$$

except that it now revises entire expectations-generating functions in response to the gap between what they predicted and what happened.

Every learning model in this series is some version of (84.7) with the gain λ made to decline over time and the revision driven by data rather than by an exact evaluation of T.

See also

Rational Expectations Equilibrium develops the T map in detail for a Lucas–Prescott industry model and computes its fixed point.

Least Squares Learning in Self-Referential Models studies what happens when agents estimate the perceived law of motion by least squares while living inside the system their estimates help determine.

84.3 Money and prices

We now build the first of the two models that motivate everything that follows.

A representative money-holder chooses nominal balances m_t to carry from t to $t+1$ to maximize

$$\ln\left(2w_1 - \frac{m_t}{p_t}\right) + \ln\left(2w_2 + \frac{m_t}{p^*_{t+1}}\right), \qquad w_1 > w_2 > 0, \tag{84.8}$$

where p_t is the current price level and p^*_{t+1} is the price level expected next period.

The first term is consumption today, reduced by the real resources m_t/p_t given up to acquire currency; the second is consumption tomorrow, augmented by the goods m_t/p^*_{t+1} that the currency is expected to command.

Differentiating (84.8) with respect to m_t and rearranging gives the demand for money

$$\frac{m_t}{p_t} = w_1 - w_2\frac{p^*_{t+1}}{p_t}. \tag{84.9}$$

Real balances fall when currency is expected to lose value faster.

This is a version of the demand function Cagan [Cagan, 1956] used to study hyperinflations, and it also arises in Samuelson's [Samuelson, 1958] overlapping generations model.

To close the model we need a theory of p^*_{t+1}.

Suppose the money supply grows at a constant rate,

$$M_{t+1} = \mu M_t, \tag{84.10}$$

and that the household believes the price level is related to the money supply by

$$p_t = \gamma M_t + \lambda^t c, \tag{84.11}$$

for constants (γ, λ, c) that summarize its beliefs, all positive so that the price level stays positive.

Knowing μ, the household forecasts $p^*_{t+1} = \gamma\mu M_t + \lambda^{t+1}c$.

Substituting the forecast and (84.11) into (84.9) gives money demand as a function of the current money supply,

$$m_t = \gamma(w_1 - w_2\mu)M_t + \lambda^t(w_1 - w_2\lambda)c. \tag{84.12}$$

Notice a feature common to models in which expectations matter: the *demand* for money depends on its *supply*, because today's demand depends on tomorrow's expected price level, which is believed to depend on tomorrow's money supply.

84.3.1 Equilibrium

Setting demand equal to supply, $m_t = M_t$, turns (84.12) into a functional equation,

$$M_t = \gamma(w_1 - w_2\mu)M_t + \lambda^t(w_1 - w_2\lambda)c,$$

which must hold at every date.

Matching the two terms gives

$$\gamma = (w_1 - \mu w_2)^{-1}, \qquad \lambda = \frac{w_1}{w_2}, \qquad c \geq 0 \text{ arbitrary}, \tag{84.13}$$

so the equilibrium price level is

$$p_t = (w_1 - \mu w_2)^{-1} M_t + \left(\frac{w_1}{w_2}\right)^t c. \tag{84.14}$$

The parameters γ and λ are pinned down.

The constant c is not.

Every $c \geq 0$ is a rational expectations equilibrium, and expectations formed from (84.11) are always exactly right in each of them.

Let's verify that the residual really does vanish for any c we care to try.

```
w1, w2, μ, M0 = 2.0, 1.0, 1.5, 1.0
γ_m, λ_m = 1 / (w1 - μ * w2), w1 / w2

t = np.arange(12)
M = M0 * μ ** t

def price_path(c):
    "Equilibrium price level for bubble constant c."
    return γ_m * M + λ_m ** t * c

for c in (0.0, 0.5, 3.0, 25.0):
    p = price_path(c)
    p_star = γ_m * μ * M + λ_m ** (t + 1) * c      # forecast of next period's price
    m = w1 * p - w2 * p_star                       # money demand
    print(f"c = {c:5}:  max |demand - supply| = {np.max(np.abs(m - M)):.2e}")
```

```
c =   0.0:  max |demand - supply| = 0.00e+00
c =   0.5:  max |demand - supply| = 0.00e+00
c =   3.0:  max |demand - supply| = 0.00e+00
c =  25.0:  max |demand - supply| = 0.00e+00
```

Money demand equals money supply exactly, at every date, for every c.

84.3.2 The bubble

Since $w_1 > w_2$, the second term in (84.14) grows at the gross rate $\lambda = w_1/w_2 > 1$.

In the equilibrium with $c = 0$ the price level is proportional to the money supply: the quantity theory in its textbook form.

In every other equilibrium the price level carries a component that has nothing to do with the money supply and grows exponentially: a purely speculative **bubble**.

```
fig, axes = plt.subplots(1, 2, figsize=(11, 4))

for c in (0.0, 0.5, 3.0):
    lab = f"$c = {c}$" + (" (quantity theory)" if c == 0 else "")
    axes[0].plot(t, price_path(c), 'o-', ms=3, lw=1, label=lab)
    axes[1].plot(t, M / price_path(c), 'o-', ms=3, lw=1, label=lab)

axes[0].set_yscale('log')
axes[0].set_ylabel("$p_t$  (log scale)")
axes[1].set_ylabel("real balances  $M_t / p_t$")
axes[1].axhline(w1 - μ * w2, color='k', lw=0.8, ls='--')
axes[1].set_ylim(0, 0.6)
```

(continues on next page)

(continued from previous page)

```
for ax in axes:
    ax.set_xlabel("$t$")
    ax.legend(frameon=False, fontsize=9)
plt.tight_layout()
plt.show()
```

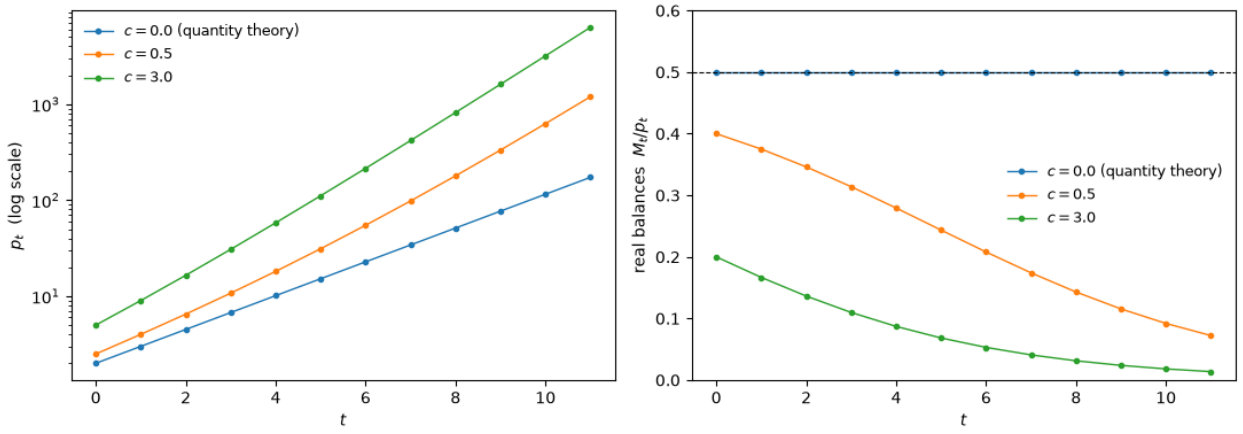


Fig. 84.3: Price levels and real balances under bubbles

The left panel shows the price level diverging further and further from the quantity-theory path as c rises.

The right panel shows what that does to the real value of the money stock: with $c = 0$ real balances are constant at $w_1 - \mu w_2$, while with $c > 0$ the bubble drives them steadily toward zero.

Along a bubble path the gross inflation rate climbs toward w_1/w_2, which from (84.9) is precisely the rate at which the demand for real balances vanishes.

The economy demonetizes itself, purely because everyone expects it to.

```
infl = pd.DataFrame(
    {f"c = {c}": price_path(c)[1:] / price_path(c)[:-1] for c in (0.0, 0.5, 3.0)},
    index=pd.Index(t[1:], name="t"),
).round(3)
infl
```

t	c = 0.0	c = 0.5	c = 3.0
1	1.5	1.600	1.800
2	1.5	1.625	1.833
3	1.5	1.654	1.864
4	1.5	1.686	1.890
5	1.5	1.721	1.913
6	1.5	1.757	1.932
7	1.5	1.792	1.947
8	1.5	1.826	1.959
9	1.5	1.857	1.969
10	1.5	1.885	1.976
11	1.5	1.908	1.982

There is nothing in the model — no preference, no technology, no policy — that says which c we are in.

Rational expectations, as the paper puts it, "is not a sufficiently restrictive principle to determine outcomes."

84.4 Two currencies

The indeterminacy gets worse when we allow more than one currency.

Following Kareken and Wallace [1981], keep (84.9) as the demand for currency *in total*, and suppose there are two fiat currencies in supplies M_{1t} and M_{2t} that are perfect substitutes as long as their rates of return are equal:

$$\frac{p^*_{1,t+1}}{p_{1t}} = \frac{p^*_{2,t+1}}{p_{2t}}. \tag{84.15}$$

This indifference about *which* currency to hold is what makes the exchange rate indeterminate.

Let people believe the price levels are given by

$$p_{1t} = \gamma_1 M_{1t} + \gamma_2 e M_{2t} + c\lambda^t, \qquad p_{2t} = e^{-1} p_{1t}, \tag{84.16}$$

where e is a constant exchange rate.

Requiring that the demand for currency, valued in units of currency 1, equal the total supply $M_{1t} + eM_{2t}$ gives

$$\gamma_1 = (w_1 - \mu_1 w_2)^{-1}, \quad \gamma_2 = (w_1 - \mu_2 w_2)^{-1}, \quad \lambda = \frac{w_1}{w_2}, \quad c \geq 0, \quad e \in [0, \infty). \tag{84.17}$$

These equations are remarkable for what they leave out.

The exchange rate e is *entirely unrestricted* — if the equations have a solution for one e, they have a solution for every other — and the formulas for γ_1 and γ_2 do not involve e at all.

Take the simplest case: two currencies in fixed supply, $\mu_1 = \mu_2 = 1$, and set $c = 0$.

Then $\gamma_1 = \gamma_2 = (w_1 - w_2)^{-1}$ and the price levels are constant.

```
w1, w2 = 2.0, 1.0
H1, H2 = 100.0, 120.0                  # fixed supplies of the two currencies
y_e = 1 / (w1 - w2)

def two_currency(e):
    "Price levels and the real allocation at exchange rate e."
    p1 = y_e * (H1 + e * H2)
    p2 = p1 / e
    supply = H1 + e * H2               # total currency, in units of currency 1
    demand = (w1 - w2) * p1            # money demand, with p* = p since prices are␣
 ↪constant
    return p1, p2, supply / p1, demand - supply

pd.DataFrame(
    [two_currency(e) for e in (0.25, 0.5, 1.0, 2.0, 4.0)],
    index=pd.Index([0.25, 0.5, 1.0, 2.0, 4.0], name="e"),
    columns=["$p_1$", "$p_2$", "real balances", "excess demand"],
).round(4)
```

e	p_1	p_2	real balances	excess demand
0.25	130.0	520.0	1.0	0.0
0.50	160.0	320.0	1.0	0.0
1.00	220.0	220.0	1.0	0.0
2.00	340.0	170.0	1.0	0.0
4.00	580.0	145.0	1.0	0.0

Every row is an equilibrium.

The nominal price levels move around a great deal as e varies.

Since e is the value of a unit of currency 2 in units of currency 1, a larger e means currency 2 is worth more, which raises p_1 and lowers p_2.

But the last two columns tell the real story: total real balances are $w_1 - w_2 = 1$ regardless of e, and markets clear exactly.

The real allocation is identical in every one of these equilibria. The model determines what people consume and how much purchasing power the currency stock commands; it says nothing whatever about the rate at which the two monies exchange.

This is a sharp version of a problem that has haunted international monetary theory, and it is not a knife-edge case: it is a continuum.

84.5 Where this leaves us

We now have two models in which rational expectations is silent about something we would very much like to predict.

There are three ways to respond.

The first is to add restrictions to the environment until the equilibrium becomes unique.

The second is to declare the indeterminacy a genuine feature of the world.

The third — the one this series pursues — is to ask what happens when we **replace the rational agents with adaptive ones** and watch where the system goes.

This is a substantive change, not a technicality.

An adaptive agent is not endowed with the equilibrium; it has beliefs, a rule for revising them, and initial conditions.

Those extra objects are exactly what the rational expectations equilibrium conditions failed to pin down, so a system of adaptive agents can select an outcome where rational expectations could not.

The rest of this series takes that idea seriously, and the results are mixed in an instructive way.

In the overlapping generations monetary economy, least squares learning selects the *opposite* equilibrium from the one the rational expectations dynamics converge to, and human experimental subjects side with the adaptive model.

In the two-currency model above, adaptive agents do pin the exchange rate down, but only by making it depend on initial conditions.

The rest points of the learning algorithm reproduce the indeterminacy exactly, and what selects an outcome is the dead hand of history.

In a Kiyotaki–Wright search economy, adaptive agents learn to use a medium of exchange and select the *fundamental* equilibrium over the speculative one, even at parameters where theory says the speculative equilibrium is the only one.

In each case the algorithm supplies what the equilibrium concept did not.

Whether that is a discovery about economies or an artifact of the algorithm is the question the series keeps returning to, and *1993 Prospects for Bounded Rationality in Macroeconomics* renders a verdict.

84.6 Exercises

Exercise 84.6.1

The relaxation algorithm (84.3) converges for the static market if and only if $\lambda < 2/(1 + bn/\gamma)$.

The ratio bn/γ measures the slope of demand relative to the curvature of costs, so a market with steep demand and near-linear costs is one in which the naive cobweb ($\lambda = 1$) is badly behaved.

Verify the stability boundary numerically: for a grid of values of bn/γ, find the largest λ (on a fine grid) for which the algorithm converges, and compare with the analytical prediction.

Solution

```
def converges(slope, λ, n_iter=400, tol=1e-8):
    """Does the relaxation algorithm converge when h'(X) = -slope?"""
    a_, X = 10.0, 1.0
    for _ in range(n_iter):
        X_new = X + λ * ((a_ - slope * X) - X)
        if not np.isfinite(X_new) or abs(X_new) > 1e12:
            return False
        X, X_prev = X_new, X
    return abs(X - X_prev) < tol


λ_grid = np.linspace(0.01, 1.5, 300)
rows = []
for slope in (0.5, 1.0, 1.5, 2.0, 4.0):
    ok = [λ for λ in λ_grid if converges(slope, λ)]
    rows.append((slope, max(ok) if ok else np.nan, 2 / (1 + slope)))

pd.DataFrame(rows, columns=["$bn/\\gamma$", "largest $\\lambda$ found",
                            "$2/(1 + bn/\\gamma)$"]).round(3)
```

	bn/γ	largest λ found	$2/(1 + bn/\gamma)$
0	0.5	1.296	1.333
1	1.0	0.972	1.000
2	1.5	0.777	0.800
3	2.0	0.648	0.667
4	4.0	0.389	0.400

The numerical boundary sits consistently a little *below* the analytical one, and that gap is not grid spacing; it is a real feature of the test.

As λ approaches the boundary the multiplier approaches -1, so convergence becomes arbitrarily slow, and a test with a fixed iteration count and tolerance declares failure just before the true boundary.

The size of the gap is predictable: with 400 iterations and a tolerance of 10^{-8}, the test passes only while $|1 - \lambda(1 + bn/\gamma)|^{400} \lesssim 10^{-8}$, i.e. while the multiplier is below about $\exp(-18.4/400) = 0.955$ in absolute value.

```
predicted = [(1 + 0.955) / (1 + s_) for s_ in (0.5, 1.0, 1.5, 2.0, 4.0)]
pd.DataFrame({"$bn/\\gamma$": [0.5, 1.0, 1.5, 2.0, 4.0],
              "found": [r[1] for r in rows],
              "predicted by the tolerance": predicted,
              "true boundary": [r[2] for r in rows]}).round(3)
```

	bn/γ	found	predicted by the tolerance	true boundary
0	0.5	1.296	1.303	1.333
1	1.0	0.972	0.978	1.000
2	1.5	0.777	0.782	0.800
3	2.0	0.648	0.652	0.667
4	4.0	0.389	0.391	0.400

Note the first row: when $bn/\gamma < 1$ the boundary exceeds one, so the naive cobweb converges on its own and no damping is needed.

Damping is what buys convergence in the steep markets, and the steeper the market the more damping is required.

Exercise 84.6.2

The equilibrium (84.13) was derived without checking that it makes economic sense.

Show that a monetary equilibrium requires $\mu < w_1/w_2$, by finding what goes wrong with the demand for real balances when money grows faster than that.

Solution

Along the $c = 0$ equilibrium, real balances are constant at $M_t/p_t = \gamma^{-1} = w_1 - \mu w_2$.

This is positive only when $\mu < w_1/w_2$.

If money grows faster, (84.13) still "solves" the functional equation, but it asks the household to hold a negative quantity of currency.

```
w1, w2 = 2.0, 1.0
μ_grid = np.array([0.5, 1.0, 1.5, 1.9, 2.0, 2.5])

with np.errstate(divide='ignore'):        # γ is infinite exactly at μ = w1/w2
    table = pd.DataFrame({
        "$\\mu$": μ_grid,
        "$\\gamma = (w_1 - \\mu w_2)^{-1}$": 1 / (w1 - μ_grid * w2),
        "real balances $w_1 - \\mu w_2$": w1 - μ_grid * w2,
        "monetary equilibrium?": np.where(μ_grid < w1 / w2, "yes", "no"),
    }).round(3)
table
```

	μ	$\gamma = (w_1 - \mu w_2)^{-1}$	real balances $w_1 - \mu w_2$	monetary equilibrium?
0	0.5	0.667	1.5	yes
1	1.0	1.000	1.0	yes
2	1.5	2.000	0.5	yes
3	1.9	10.000	0.1	yes
4	2.0	inf	0.0	no
5	2.5	-2.000	-0.5	no

At $\mu = w_1/w_2 = 2$ the demand for real balances hits zero and γ blows up; beyond it, both are negative.

The intuition runs through (84.9): the household holds currency only if the expected loss of purchasing power, p^*_{t+1}/p_t, is smaller than w_1/w_2.

Money growing at rate μ produces inflation at rate μ, so $\mu \geq w_1/w_2$ drives the demand for money to zero, the same boundary that the *bubble* equilibria approach asymptotically from below.

Exercise 84.6.3

In the two-currency example we set $\mu_1 = \mu_2 = 1$ and found that the real allocation was the same at every exchange rate.

That is special.

Repeat the calculation with $\mu_1 \neq \mu_2$ — say $\mu_1 = 1.0$ and $\mu_2 = 1.3$, with $w_1 = 2$, $w_2 = 1$, $M_{1,0} = M_{2,0} = 100$, and $c = 0$ — and compute total real balances at several exchange rates over the first several periods.

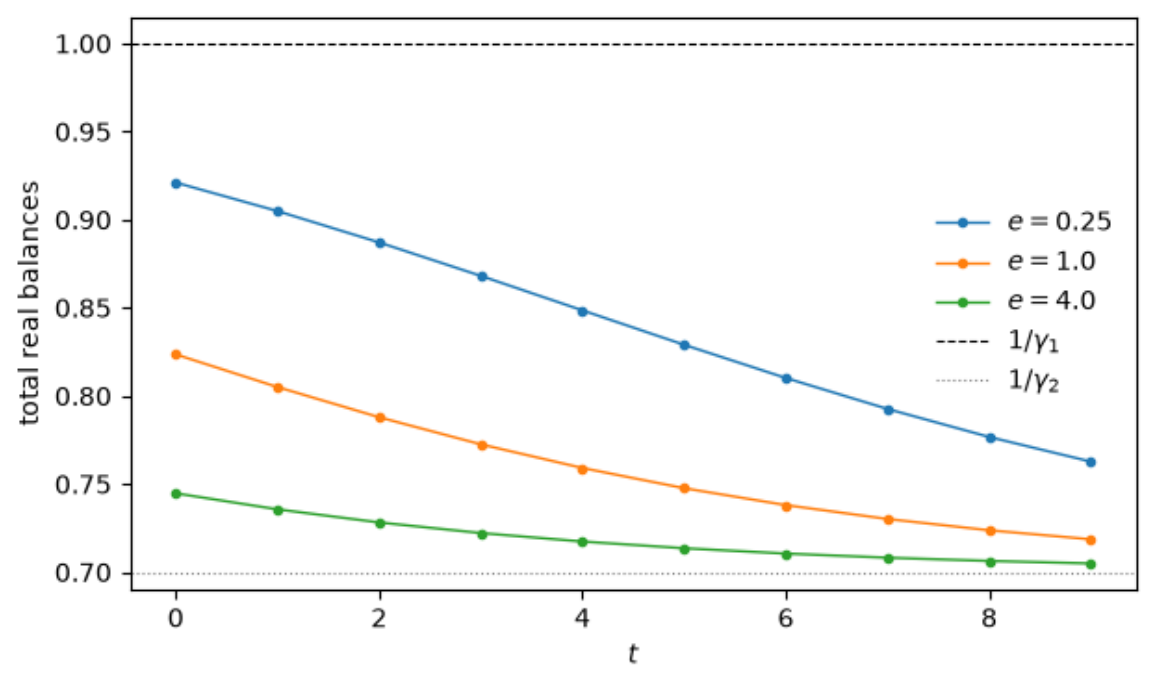
Does the choice of e still leave the real allocation untouched?

Solution

```
w1, w2 = 2.0, 1.0
μ1, μ2 = 1.0, 1.3
γ1, γ2 = 1 / (w1 - μ1 * w2), 1 / (w1 - μ2 * w2)
t = np.arange(10)
M1, M2 = 100.0 * μ1 ** t, 100.0 * μ2 ** t

def real_balances(e):
    p1 = γ1 * M1 + γ2 * e * M2
    return (M1 + e * M2) / p1

pd.DataFrame({f"e = {e}": real_balances(e) for e in (0.25, 1.0, 4.0)},
             index=pd.Index(t, name="t")).round(4)
```

t	e = 0.25	e = 1.0	e = 4.0
0	0.9211	0.8235	0.7447
1	0.9049	0.8050	0.7356
2	0.8871	0.7879	0.7282
3	0.8681	0.7725	0.7221
4	0.8485	0.7591	0.7173
5	0.8290	0.7476	0.7135
6	0.8101	0.7380	0.7105
7	0.7926	0.7301	0.7081
8	0.7767	0.7237	0.7063
9	0.7627	0.7186	0.7049

```
fig, ax = plt.subplots(figsize=(7, 4))
for e in (0.25, 1.0, 4.0):
    ax.plot(t, real_balances(e), 'o-', ms=3, lw=1, label=f"$e = {e}$")
ax.axhline(1 / γ1, color='k', lw=0.8, ls='--', label=r"$1/\gamma_1$")
ax.axhline(1 / γ2, color='gray', lw=0.8, ls=':', label=r"$1/\gamma_2$")
ax.set_xlabel("$t$")
ax.set_ylabel("total real balances")
ax.legend(frameon=False)
plt.show()
```



No. With unequal money growth rates the exchange rate affects the real allocation, and the allocation is no longer even constant over time.

The reason is that $\gamma_1 \neq \gamma_2$: the two currencies are valued differently because they are expected to be diluted at different rates, so the *composition* of the currency stock matters, and e is what fixes that composition.

Since currency 2 grows faster, it comes to dominate the stock whatever e we choose, and real balances converge to $1/\gamma_2$ from wherever e starts them.

So the pure nominal indeterminacy of the fixed-supply case is a knife-edge, but the indeterminacy of e itself is not: every e in the table is still an equilibrium.

CHAPTER

EIGHTYFIVE

ADAPTIVE AGENTS IN AN OVERLAPPING GENERATIONS MONETARY ECONOMY

Contents

- *Adaptive Agents in an Overlapping Generations Monetary Economy*
  - *Overview*
  - *The environment*
  - *Part 1: a stochastic deficit*
  - *Part 2: a constant deficit and two steady states*
  - *A government learning the Phillips curve*
  - *Concluding remarks*
  - *Exercises*

85.1 Overview

A Peculiar Definition of Bounded Rationality presented some models of monetary economies with multiple rational expectations equilibria.

This lecture takes one of them — Samuelson's overlapping generations model of fiat money, used by Bryant, Wallace and others to study inflationary finance — and replaces Samuelson's agents who forecast according to the equilibrium law of motion for the price level with "adaptive" ones who do not.

We do this twice, for two different purposes.

Part 1 puts a *stochastic* government deficit into the model and asks whether successive generations, groping their way by trial and error, can somehow as a society converge to a rational expectations equilibrium.

They can.

Each generation observes what happened to its predecessors and adjusts their saving decision in a utility-improving direction, using a **Robbins–Monro** recursion.

Given enough time the economy converges to the stationary equilibrium we compute independently.

This exercise also shows how sensitive learning is to the complexity of what must be learned: a deficit state that occurs rarely is learned about slowly, because the observations arrive slowly.

Part 2 removes the randomness and turns to a sharper question.

With a *constant* deficit the model has **two** stationary equilibria, one with low inflation and one with high inflation.

The low-inflation equilibrium Pareto-dominates the other.

Under the rational expectations dynamics, the low-inflation equilibrium is **unstable** and the high-inflation one is stable: theory pushes the economy toward the bad outcome.

Under least squares learning, the stability is **reversed**.

And when Marimon and Sunder [1993] ran the economy as a laboratory experiment with paid human subjects, the subjects behaved like the adaptive model, not like the rational expectations model.

That is the strongest evidence in Sargent [1993] that adaptive dynamics are doing real work as a selection device, tempered by a counterexample, due to Benjamin Bental, warning against concluding that adaptation reliably selects the *good* equilibrium.

We then close with a **third application**, in which the adaptive agent is a government learning a Phillips curve.

Now least squares learning selects the *bad*, high-inflation equilibrium, but a government that doubts its own model can **escape** toward the good one, a phenomenon that seeded Sargent's later work on *The Conquest of American Inflation*.

Let's start with some imports.

```
import numpy as np
import pandas as pd
import quantecon as qe
import matplotlib.pyplot as plt
from scipy.optimize import fsolve
```

85.2 The environment

The economy consists of overlapping generations of two-period-lived agents.

At each date $t \geq 1$, N identical agents are born, endowed with w_1 units of a single consumption good when young and w_2 units when old.

A young agent's preferences over lifetime consumption (c_1, c_2) are ordered by the expected value of $u(c_1) + u(c_2)$.

Throughout we use logarithmic utility, $u(c) = \ln c$.

A government prints currency to finance expenditures G_t, subject to the budget constraint

$$G_t = \frac{H_t - H_{t-1}}{p_t}, \tag{85.1}$$

where H_t is the stock of currency carried by the young at t into $t+1$, and p_t is the price level.

Currency is the only store of value, so a young agent's saving s_t is entirely in the form of real balances, and market clearing requires

$$\frac{H_t}{p_t} = s_t N. \tag{85.2}$$

Write $R_t = p_t / p_{t+1}$ for the gross rate of return on currency between t and $t+1$.

Combining (85.1) and (85.2) gives a relation we will use constantly: the return on currency is determined entirely by saving behavior and the deficit,

$$R_t = \frac{N s_{t+1} - G_{t+1}}{N s_t}. \tag{85.3}$$

Currency holds its value only if the next generation is willing to absorb the outstanding stock plus the new issue.

Note

(85.3) is worth pausing on, because it is what makes this a *self-referential* system, the property studied at length in *Least Squares Learning in Self-Referential Models*.

The return that today's young earn on their saving depends on how much tomorrow's young choose to save, which in turn depends on what tomorrow's young expect to earn.

Nobody's beliefs are about an exogenous process.

85.3 Part 1: a stochastic deficit

Now let government expenditures follow a Markov chain,

$$\pi(i,j) = \text{Prob}\{G_{t+1} = \bar{G}_j \mid G_t = \bar{G}_i\},$$

over a finite list of levels $G = [\bar{G}_1, \dots, \bar{G}_n]$.

85.3.1 Stationary rational expectations equilibrium

In a stationary equilibrium, saving depends on the current deficit state, so it is described by an $n \times 1$ vector $s = [s_1, \dots, s_n]$, and the return on currency by an $n \times n$ matrix $R(i,j)$.

A young agent who observes $G_t = \bar{G}_i$ chooses s_i to maximize

$$V(s_i) = u(w_1 - s_i) + \sum_j u(w_2 + s_i R(i,j))\pi(i,j),$$

with first-order condition

$$u'(w_1 - s_i) = \sum_{j=1}^{n} u'(w_2 + s_i R(i,j))R(i,j)\pi(i,j). \tag{85.4}$$

Substituting (85.3) — which in the stationary equilibrium reads $R(i,j) = (s_j N - \bar{G}_j)/(s_i N)$ — turns (85.4) into a system of n nonlinear equations in the saving rates alone:

$$u'(w_1 - s_i) = \sum_{j=1}^{n} u'\left(w_2 + \frac{s_j N - \bar{G}_j}{N}\right) \cdot \frac{s_j N - \bar{G}_j}{s_i N} \cdot \pi(i,j). \tag{85.5}$$

A stationary equilibrium exists if and only if (85.5) has a solution with $s_i \in (0, w_1)$ for every i.

```
w1, w2, N = 20.0, 10.0, 1.0

def u_prime(c):
    return 1 / c                          # log utility

def equilibrium_residual(s, G, P):
    "Left minus right side of the equilibrium condition, one entry per deficit state."
    n = len(s)
    out = np.empty(n)
    for i in range(n):
```

(continues on next page)

(continued from previous page)

```
        rhs = sum(u_prime(w2 + (s[j]*N - G[j])/N)
                  * (s[j]*N - G[j])/(s[i]*N)
                  * P[i, j] for j in range(n))
        out[i] = u_prime(w1 - s[i]) - rhs
    return out

def stationary_equilibrium(G, P, guess=4.0):
    s = fsolve(equilibrium_residual, np.full(len(G), guess), args=(G, P))
    R = np.array([[(s[j]*N - G[j])/(s[i]*N) for j in range(len(s))]
                  for i in range(len(s))])
    return s, R
```

Following Sargent [1993], we study two economies that differ only in the process for government expenditures, with $w_1 = 20$, $w_2 = 10$ and $N = 1$, so that $\bar{G}$ is measured per young person.

In the first, the deficit is always zero and the chain is a fair coin.

```
G1, P1 = np.array([0.0, 0.0]), np.full((2, 2), 0.5)
s1, R1 = stationary_equilibrium(G1, P1)
print(f"saving rates      {np.round(s1, 4)}")
print(f"returns\n{np.round(R1, 4)}")
```

```
saving rates      [5. 5.]
returns
[[1. 1.]
 [1. 1.]]
```

With no deficit to finance, currency simply holds its value: agents save 5 in either state and $R \equiv 1$.

In the second economy the deficit is 0.8 in state 1 and zero in state 2, and the chain is persistent.

```
G2 = np.array([0.8, 0.0])
P2 = np.array([[0.75, 0.25],
               [0.50, 0.50]])
s2, R2 = stationary_equilibrium(G2, P2)
print(f"saving rates      {np.round(s2, 4)}      (Sargent reports 4.211, 4.364)")
print(f"returns\n{np.round(R2, 4)}")
print("Sargent reports  [[0.81, 1.0362], [0.7817, 1.00]]")
```

```
saving rates      [4.2111 4.3637]      (Sargent reports 4.211, 4.364)
returns
[[0.81   1.0362]
 [0.7817 1.    ]]
Sargent reports  [[0.81, 1.0362], [0.7817, 1.00]]
```

Both match the values reported in the text.

Notice the pattern in R: the return on currency is poor whenever the government is about to run a deficit (the first column), because new issue dilutes the outstanding stock.

85.3.2 Learning by successive generations

Now withdraw the equilibrium from the agents.

They are still assumed to know their own utility function and to remember the experience of earlier agents like themselves, but they are *not* told the distribution of returns, which is exactly the object that (85.4) requires.

Instead, each generation adjusts its predecessors' saving decision in the direction that *realized* experience suggests would have been better.

The realized lifetime utility of an agent who saved s and earned R is

$$U(s) = u(w_1 - s) + u(w_2 + sR),$$

with derivatives

$$U'(s) = -u'(w_1 - s) + u'(w_2 + sR)R, \qquad U''(s) = u''(w_1 - s) + u''(w_2 + sR)R^2.$$

The connection to equilibrium is that in a rational expectations equilibrium $V'(s_i) = E_t U'(s_i)$ and $V''(s_i) = E_t U''(s_i)$, where E_t conditions on $G_t = \bar{G}_i$.

So an agent who could compute $E_t U'$ would just set it to zero and be done.

Our agents cannot, and must estimate it from experience instead.

They use a **Robbins–Monro** algorithm, applied state by state.

Let τ_i count the number of times the deficit state $\bar{G}_i$ has been visited so far, and let γ_τ be a decreasing gain sequence.

The rules are

$$\begin{aligned} M(i, \tau_i + 1) &= M(i, \tau_i) + \gamma_{\tau_i}\big(U''(s(i, \tau_i)) - M(i, \tau_i)\big), \\ s(i, \tau_i + 1) &= s(i, \tau_i) - \gamma_{\tau_i} M(i, \tau_i + 1)^{-1} U'(s(i, \tau_i)). \end{aligned} \tag{85.6}$$

M accumulates a running estimate of $E_t U''$, and the saving rule takes a Newton step against it.

If $\tau_i \to \infty$ then $M(i, \tau_i) \to E_i U''$ and $s(i, \tau_i)$ approaches a solution of $E_i U'(s_i) = 0$, which is exactly the equilibrium condition (85.4).

Two features of the setup deserve comment.

Two classes of agent. To evaluate a saving decision we must wait until *two* periods of that agent's consumption are known, so the population is split into an "odd" and an "even" subsequence.

Odd agents reset their rule in odd periods and learn only from earlier odd agents; even agents likewise.

This is the same device used in the laboratory experiments discussed in Part 2.

A projection facility. Nothing in (85.6) prevents a Newton step from pushing saving below the deficit, at which point (85.2) has no positive price level and the economy ceases to exist.

We therefore confine s to a region where an equilibrium price level exists.

Devices of this kind are needed for the convergence theorems too; *Least Squares Learning in Self-Referential Models* discusses the projection facility in detail.

```
def U_prime(s, R):
    return -1/(w1 - s) + R/(w2 + s*R)

def U_double(s, R):
    return -1/(w1 - s)**2 - R**2/(w2 + s*R)**2
```

(continues on next page)

(continued from previous page)

```
def simulate(G, P, T=50_000, s0=None, seed=0, tau0=20, band=0.4):
    """
    Overlapping generations learning a state-contingent saving rule.

    Returns the history of saving rules (shape (T, 2, n): time, class, state)
    and the realized returns.
    """
    rng = np.random.default_rng(seed)
    n = len(G)
    s = np.array(s0, float)
    M = np.array([[U_double(s[j, k], 1.0) for k in range(n)] for j in range(2)])
    τ = np.zeros((2, n), int)
    floor = G/N + band                  # projection facility: saving must cover the␣
↪deficit

    hist = np.empty((T, 2, n))
    R_hist = np.full(T, np.nan)
    i, prev = 0, None

    for t in range(T):
        j = t % 2                       # which class is young this period
        s_t = s[j, i]

        if prev is not None:            # last period's young now learn their return
            j_p, i_p, s_p = prev
            R = (N*s_t - G[i]) / (N*s_p)          # the return on currency
            R_hist[t-1] = R
            τ[j_p, i_p] += 1
            g = 1 / (τ[j_p, i_p] + tau0)
            M[j_p, i_p] += g * (U_double(s_p, R) - M[j_p, i_p])
            step = g * U_prime(s_p, R) / M[j_p, i_p]
            s[j_p, i_p] = np.clip(s[j_p, i_p] - step, floor[i_p], w1 - 0.4)

        prev = (j, i, s_t)
        hist[t] = s
        i = rng.choice(n, p=P[i])

    return hist, R_hist
```

Note

The simulation never computes a price level.

Because (85.3) expresses the return on currency purely in terms of saving rates and the deficit, the *real* side of the model is self-contained.

This is more than a convenience: the nominal money stock in these economies grows without bound whenever the government runs a deficit, so a simulation that tracked H_t and p_t directly would overflow long before the learning converged.

85.3.3 Do they find it?

```
start = [[8.0, 2.0],
         [2.0, 8.0]]                    # deliberately far from equilibrium, and␣
↪asymmetric

hist1, _ = simulate(G1, P1, s0=start)
hist2, _ = simulate(G2, P2, s0=start)

fig, axes = plt.subplots(1, 2, figsize=(11, 4), sharex=True)
for ax, hist, s_star, title in [(axes[0], hist1, s1, "Economy 1:  $\\bar G = (0, 0)$
↪"),
                                (axes[1], hist2, s2, "Economy 2:  $\\bar G = (0.8, 0)$
↪")]:
    for k, colour in enumerate(['C0', 'C1']):
        ax.plot(hist[:, 0, k], color=colour, lw=1, label=f"state {k+1}, even")
        ax.plot(hist[:, 1, k], color=colour, lw=1, ls=':', label=f"state {k+1}, odd")
        ax.axhline(s_star[k], color='k', lw=0.8, ls='--')
    ax.set_xscale('log')
    ax.set_xlabel("$t$  (log scale)")
    ax.set_title(title)
axes[0].set_ylabel("saving rate")
axes[0].legend(frameon=False, fontsize=8)
plt.tight_layout()
plt.show()
```

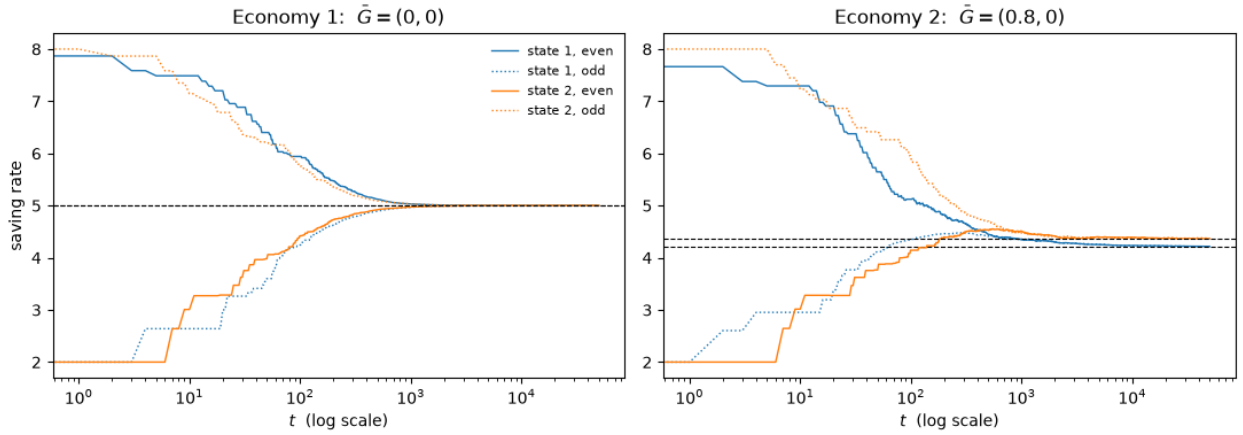


Fig. 85.1: Saving rates converge in both economies

Both economies converge, and both classes of agent converge to the same rule even though neither learns from the other.

The dashed black lines are the equilibrium saving rates computed earlier from (85.5), objects that no agent in the simulation has access to.

```
rows = []
for T in (5_000, 50_000, 200_000):
    h1, _ = simulate(G1, P1, T=T, s0=start)
    h2, _ = simulate(G2, P2, T=T, s0=start)
    rows.append([T, *h1[-1].mean(axis=0), *h2[-1].mean(axis=0)])
```

(continues on next page)

(continued from previous page)

```
pd.DataFrame(rows, columns=["T", "econ 1 state 1", "econ 1 state 2",
                            "econ 2 state 1", "econ 2 state 2"]).set_index("T").
↪round(4)
```

T	econ 1 state 1	econ 1 state 2	econ 2 state 1	econ 2 state 2
5000	5.0004	5.0005	4.2551	4.3938
50000	5.0000	5.0000	4.2172	4.3708
200000	5.0000	5.0000	4.2106	4.3635

```
print(f"rational expectations:  economy 1 = {np.round(s1, 4)},  economy 2 = {np.
↪round(s2, 4)}")
```

```
rational expectations:  economy 1 = [5. 5.],  economy 2 = [4.2111 4.3637]
```

Convergence is slow — that is what a $1/\tau$ gain buys you — but it is convergence, and to the right place.

85.3.4 How much do the agents have to be told?

The specification above lets agents learn a *separate* saving rate for each level of the deficit.

In effect they are learning the policy function $s = f(G)$ **non-parametrically**, state by state.

That is feasible here because there are only two states.

It has two drawbacks that Sargent emphasizes.

First, states that are rarely visited are learned about slowly, because the observations arrive slowly.

Exercise 85.7.1 makes this concrete.

Second, when the number of states is large, one parameter per state becomes unmanageable.

An econometrician's response would be to impose a **parametric** form $s = f(G, \theta)$ with θ of low dimension and use every observation to estimate it, replacing (85.6) with a recursion in θ driven by $\partial U/\partial\theta$.

This buys speed at the cost of *approximation*: a parametric learning scheme can converge to a rational expectations equilibrium only if some member of the family $f(\cdot, \theta)$ supports one.

Otherwise the best it can do is converge to an approximate equilibrium.

Note

This is the point at which models of learning and algorithms for *computing* equilibria become hard to tell apart.

Marcet's method of parameterized expectations posits a parametric form for a conditional expectation, simulates, regresses realized outcomes on the parametric form to update the coefficients, and iterates.

Written recursively, that algorithm is a nonlinear-least-squares recursion of the same shape as (85.6).

Sargent's summary: "Learning algorithms and equilibrium computation algorithms look like each other."

An equilibrium computation is a centralized learning algorithm run by the modeller; a learning economy is a decentralized equilibrium computation run by the agents.

85.4 Part 2: a constant deficit and two steady states

Now strip out the randomness.

Let the government finance a *constant* deficit G per young person, and let utility be $\ln c_1 + \ln c_2$ as before.

With perfect foresight over the price level, the young save

$$s_t = \frac{w_1 - w_2 \pi_t}{2}, \qquad \pi_t \equiv \frac{p_{t+1}}{p_t}, \tag{85.7}$$

where π_t is the gross inflation rate, the reciprocal of the return on currency.

The government's budget constraint in real terms is $h_t = h_{t-1}/\pi_{t-1} + G$, where $h_t = m_t/p_t$, and equilibrium requires $h_t = s_t$.

Eliminating h between these gives an autonomous difference equation in the inflation rate,

$$\pi_{t+1} = g(\pi_t) \equiv A_1 - \frac{A_2}{\pi_t}, \qquad A_1 = \frac{w_1 + w_2 - 2G}{w_2}, \qquad A_2 = \frac{w_1}{w_2}. \tag{85.8}$$

Stationary equilibria are the fixed points of g, and since g is a hyperbola there are generically two of them.

We use the parameters of Marimon and Sunder [1993]'s "Economy 7C": $w_1 = 6$, $w_2 = 1$, $G = 1$.

```
w1_d, w2_d, G_d = 6.0, 1.0, 1.0
A1 = (w1_d + w2_d - 2*G_d) / w2_d
A2 = w1_d / w2_d

def g_map(π):
    return A1 - A2/π

π_low, π_high = np.sort(np.roots([1, -A1, A2]))
print(f"A1 = {A1},  A2 = {A2}")
print(f"stationary gross inflation rates: {π_low} and {π_high}")
print(f"   i.e. net inflation of {100*(π_low-1):.0f}% and {100*(π_high-1):.0f}% per␣
↪period")
print(f"\nslope of g at the low  rate: {A2/π_low**2:.4f}   -> unstable under RE␣
↪dynamics")
print(f"slope of g at the high rate: {A2/π_high**2:.4f}   -> stable under RE dynamics
↪")
```

```
A1 = 5.0,  A2 = 6.0
stationary gross inflation rates: 2.0 and 3.0
   i.e. net inflation of 100% and 200% per period

slope of g at the low  rate: 1.5000   -> unstable under RE dynamics
slope of g at the high rate: 0.6667   -> stable under RE dynamics
```

There it is: the *low*-inflation stationary equilibrium is **unstable** under the rational expectations dynamics, and the high-inflation one is stable.

```
π_grid = np.linspace(1.35, 4.2, 400)

fig, ax = plt.subplots(figsize=(6.5, 5.5))
ax.plot(π_grid, g_map(π_grid), lw=1.6, label=r"$g(\pi) = A_1 - A_2/\pi$")
ax.plot(π_grid, π_grid, 'k--', lw=0.8, label="45 degree line")

# cobweb the RE dynamics away from the low steady state
```

(continues on next page)

(continued from previous page)

```
π = π_low + 0.05
for _ in range(7):
    nxt = g_map(π)
    ax.plot([π, π], [π, nxt], color='C3', lw=0.9)
    ax.plot([π, nxt], [nxt, nxt], color='C3', lw=0.9)
    π = nxt

for p, name in [(π_low, "low"), (π_high, "high")]:
    ax.plot(p, p, 'ko', ms=6)
    ax.annotate(f"{name}\n$\\pi = {p:.0f}$", (p, p), textcoords="offset points",
                xytext=(10, -22), fontsize=9)
ax.set_xlabel(r"$\pi_t$")
ax.set_ylabel(r"$\pi_{t+1}$")
ax.legend(frameon=False, loc='upper left')
plt.show()
```

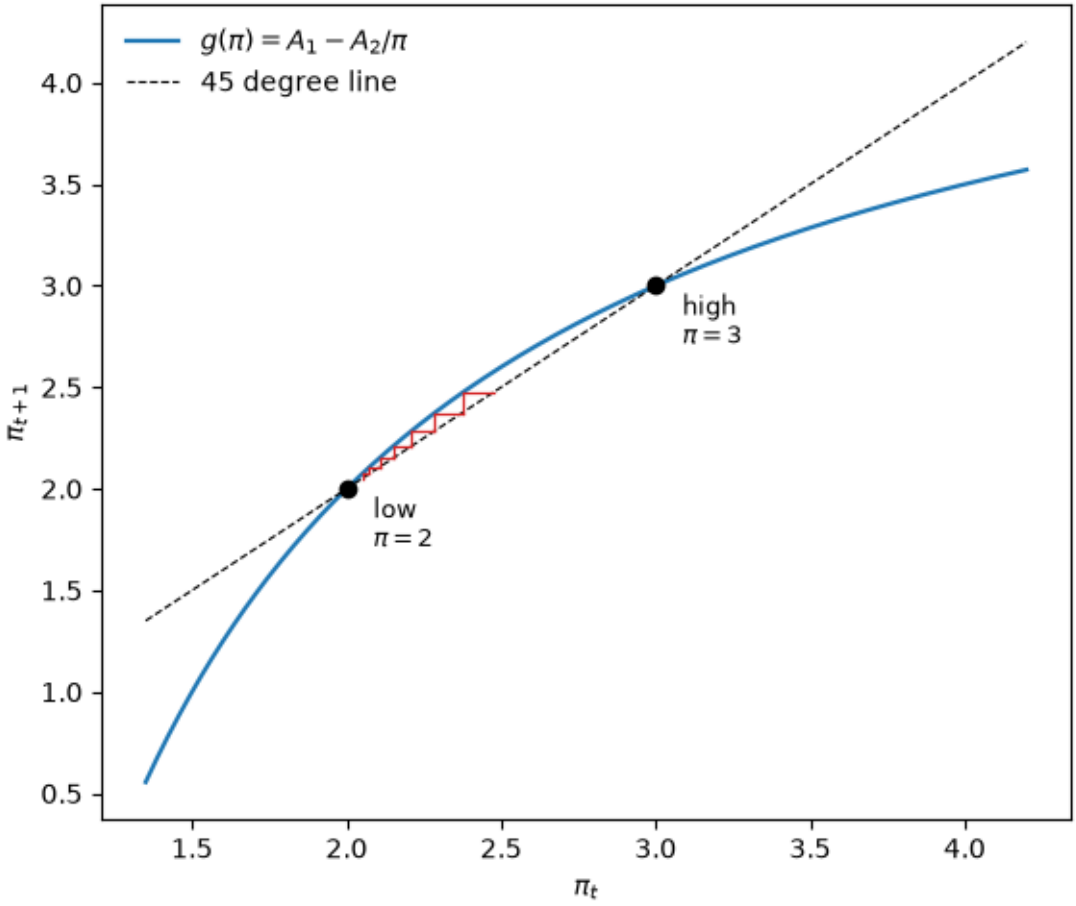


Fig. 85.2: Rational expectations inflation dynamics

The red staircase starts just above the low-inflation steady state and marches away from it, straight to the high-inflation one.

```
def re_path(π0, T=25):
    path = [π0]
    for _ in range(T):
        nxt = g_map(path[-1])
        path.append(nxt if nxt > 0 else np.nan)
    return np.array(path)

pd.DataFrame({f"$\\pi_0 = {p}$": re_path(p) for p in (1.99, 2.01, 2.5, 4.0)}
             ).rename_axis("t").round(4).iloc[[0, 1, 2, 5, 10, 25]]
```

t	$\pi_0 = 1.99$	$\pi_0 = 2.01$	$\pi_0 = 2.5$	$\pi_0 = 4.0$
0	1.9900	2.0100	2.5000	4.0000
1	1.9849	2.0149	2.6000	3.5000
2	1.9772	2.0222	2.6923	3.2857
5	1.9187	2.0712	2.8836	3.0705
10	0.6693	2.3681	2.9830	3.0087
25	NaN	2.9961	3.0000	3.0000

Started fractionally below the low steady state the economy collapses (the price level implied by (85.8) goes negative); started fractionally above it, the economy converges to $\pi = 3$.

This matters because of how the two equilibria rank.

The comparative statics of the low-inflation equilibrium are "classical": raising the deficit G lowers A_1, which raises the low stationary inflation rate.

The comparative statics of the high-inflation equilibrium are the reverse, because the economy is on the wrong side of an inflation-tax **Laffer curve**; see Exercise 85.7.3.

And the low-inflation equilibrium Pareto-dominates every other equilibrium of this model, stationary or not.

So the rational expectations dynamics repel the economy from the Pareto-optimal outcome, and most classical doctrine in monetary theory depends on selecting the equilibrium that those dynamics reject.

85.4.1 Least squares dynamics

Marcet and Sargent [1989] studied a version of this economy in which agents do not have perfect foresight but instead **run a regression**.

Each period they regress the price level on its own lagged value and use the fitted slope as their forecast of gross inflation,

$$\beta_t = \frac{\sum_{s<t} p_s p_{s-1}}{\sum_{s<t} p_{s-1}^2}, \qquad \pi_t^e = \beta_t,$$

and then save $s_t = (w_1 - w_2\beta_t)/2$ according to (85.7).

Since β_t uses data through $t-1$ only, there is no simultaneity: beliefs determine saving, saving determines the price level, and the new price level enters next period's regression.

Writing β_t as

$$\beta_t = \frac{\sum_{s<t} \pi_s\, p_{s-1}^2}{\sum_{s<t} p_{s-1}^2}$$

shows what the estimator is doing: it is a weighted average of *past realized inflation rates*, with weights proportional to p_{s-1}^2.

That form is also what makes the recursion easy to compute.

The price level grows geometrically, so the two sums overflow quickly, but if we rescale both by the newest weight at each step everything stays of order one.

```
def ls_dynamics(π_init, T=3000):
    """
    Least squares dynamics.  Carries the two regression sums rescaled by the
    newest p^2, so nothing overflows even though the price level does not stop
    growing.  Returns (beliefs, realized inflation, survived?).
    """
    s_prev = (w1_d - w2_d*π_init) / 2
    num, den, π_last = π_init, 1.0, π_init
    betas, infl = [], []

    for t in range(T):
        β = num / den
        s_t = (w1_d - w2_d*β) / 2
        if s_t <= G_d or not np.isfinite(s_t):
            return np.array(betas), np.array(infl), False   # no positive price level
        π_t = s_prev / (s_t - G_d)                          # realized gross inflation
        num = num/π_last**2 + π_t
        den = den/π_last**2 + 1.0
        π_last, s_prev = π_t, s_t
        betas.append(β)
        infl.append(π_t)

    return np.array(betas), np.array(infl), True


rows = []
for π0 in (1.2, 1.5, 1.9, 2.0, 2.1, 2.5, 2.9, 3.0):
    b, infl, ok = ls_dynamics(π0)
    rows.append([π0, b[-1], infl[-1], "yes" if ok else "no"])

pd.DataFrame(rows, columns=["initial belief $\\pi_0$", "belief $\\beta_T$",
                            "realized $\\pi_T$", "survived"]).round(5)
```

	initial belief π_0	belief β_T	realized π_T	survived
0	1.2	2.0	2.0	yes
1	1.5	2.0	2.0	yes
2	1.9	2.0	2.0	yes
3	2.0	2.0	2.0	yes
4	2.1	2.0	2.0	yes
5	2.5	2.0	2.0	yes
6	2.9	2.0	2.0	yes
7	3.0	2.0	2.0	yes

85.4.2 The stability reversal

Least squares learning converges to the **low**-inflation equilibrium from every initial belief in the table, including from $\pi_0 = 3.0$, which is the steady state that the rational expectations dynamics converge *to*.

```
fig, ax = plt.subplots(figsize=(8, 4.5))

for π0 in (1.5, 2.5, 3.0):
    _, infl, _ = ls_dynamics(π0, T=400)
    ax.plot(infl, color='C0', lw=1.2,
```

(continues on next page)

(continued from previous page)

```
            label="least squares dynamics" if π0 == 1.5 else None)

for π0 in (2.05, 2.5, 4.0):
    ax.plot(re_path(π0, 400), color='C3', lw=1.2, ls='--',
            label="rational expectations dynamics" if π0 == 2.05 else None)

ax.axhline(π_low, color='k', lw=0.8, ls=':')
ax.axhline(π_high, color='k', lw=0.8, ls=':')
ax.annotate("low", (330, π_low + 0.04), fontsize=9)
ax.annotate("high", (330, π_high + 0.04), fontsize=9)
ax.set_xlim(0, 400)
ax.set_ylim(1.2, 4.2)
ax.set_xlabel("$t$")
ax.set_ylabel("gross inflation")
ax.legend(frameon=False)
plt.show()
```

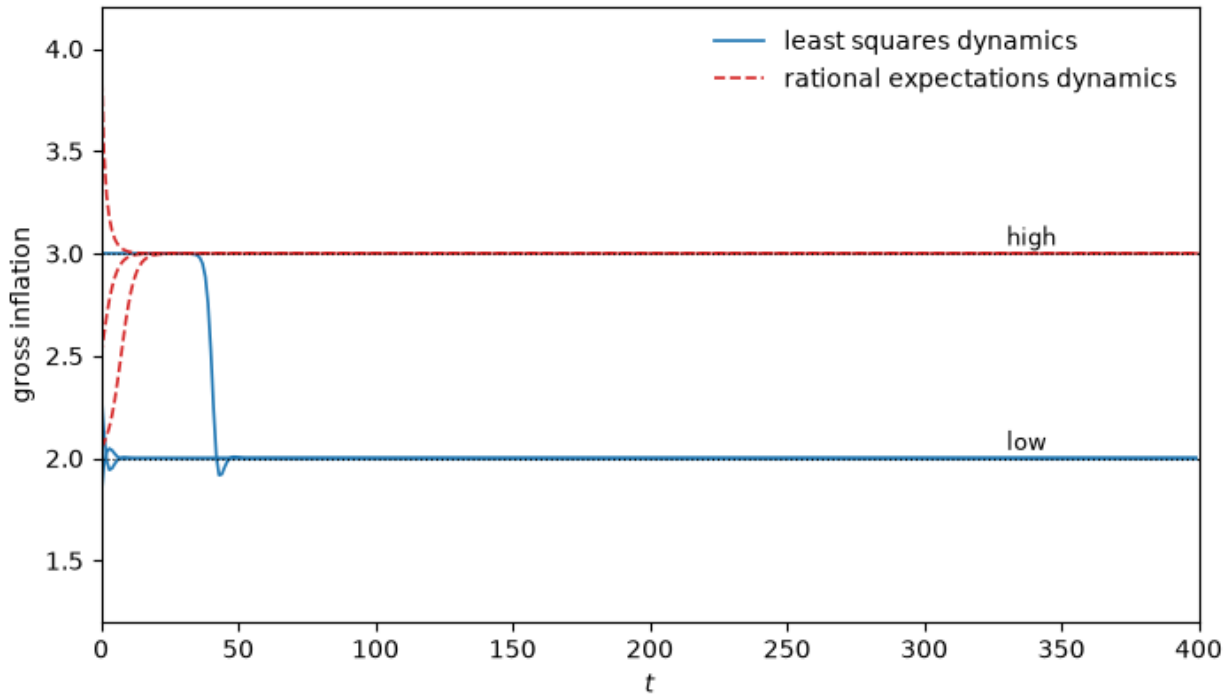


Fig. 85.3: Least squares and rational expectations inflation paths

The two families of paths run in opposite directions.

The boundary is exact: least squares learning converges to $\pi = 2$ from every initial belief up to and including $\pi_0 = 3$, and above that the economy has no equilibrium price level at all.

```
lo, hi = 3.0, 3.2
for _ in range(40):                          # bisect for the edge of the basin
    mid = (lo + hi) / 2
    lo, hi = (mid, hi) if ls_dynamics(mid)[2] else (lo, mid)
print(f"least squares dynamics survive for initial beliefs up to {lo:.6f}")
```

(continues on next page)

(continued from previous page)

```
print(f"the rational-expectations-stable steady state is      {π_high}")
```

```
least squares dynamics survive for initial beliefs up to 3.000000
the rational-expectations-stable steady state is      3.0
```

The steady state that the rational expectations dynamics single out as *the* stable one turns out to be precisely the edge of the basin of attraction of the learning dynamics toward the other one.

Why the reversal?

Under perfect foresight, a belief slightly above π_{low} is *validated and amplified* — the map g has slope 1.5 there.

Under least squares, beliefs are a weighted average of a long history of realized inflation rates, and that averaging drags the forecast back down.

The high steady state is stable for the forward-looking dynamics and repelling for the backward-looking ones.

Bruno and Fischer [1990] found the same reversal in a closely related model using Friedman-style adaptive expectations instead of least squares, so the result does not hinge on the details of the estimator.

85.4.3 Marimon and Sunder's experiment

Which set of dynamics describes people?

Marimon and Sunder [1993] built a laboratory economy to find out.

A fixed number of paying subjects were assigned roles as "young" and "old" in an implementation of exactly this model.

Each period the young submitted a schedule saying how much output they were willing to supply to the old for money at each price, and the experimenters cleared the market against the demand from the old and from the government.

Subjects temporarily on the sidelines played a **forecasting game** with its own cash prize, so the experiment produced a direct record of the price expectations driving the economy.

The session length was unknown to the subjects but governed by publicly announced rules that made the economy equivalent, from their point of view, to one that never ends.

Two findings matter here.

First, the experimental inflation paths were much better approximated by the least squares dynamics converging to the **low** stationary rate than by the rational expectations dynamics converging to the high one.

That pattern held across all of their economies.

Second, the sideliners' forecast errors were comparable in size to those a least squares forecaster would have made using the same data.

The subjects were not doing something exotic, they were doing roughly what the adaptive model says.

The equilibrium that theory calls unstable is the one that both the algorithm and the humans select.

> **Note**
>
> Experimental evidence is not the only way to discriminate.
>
> İmrohoroğlu [1993] estimated a rational expectations version of this model on German hyperinflation data and found the opposite: his estimated equilibrium slides along the *bad* side of the Laffer curve toward the high stationary rate, which is inconsistent with the equilibrium the adaptive dynamics select.

85.4.4 A warning

It is tempting to conclude that adaptive dynamics reliably select the Pareto-superior equilibrium.

Sargent offers a counterexample, shown to him by Benjamin Bental, built from a special case of Brock [1974]'s money-in-the-utility-function model.

An infinitely lived household maximizes $\sum_t \beta^t (\ln c_t + \gamma \ln(m_t/p_t))$ subject to a sequence of budget constraints, while the government finances a fixed expenditure $g > 0$ by printing currency.

Under the parameter restriction $\gamma(y-g) > g$ the model has a unique stationary equilibrium, with gross inflation

$$\pi = \frac{(y-g)\gamma - \beta g}{(y-g)\gamma - g},$$

which has the classical property of rising with the deficit.

```
γ_b, β_b, y_b = 1.0, 1/1.05, 11.0

def brock_π(g):
    return ((y_b - g)*γ_b - β_b*g) / ((y_b - g)*γ_b - g)

pd.DataFrame({
    "$g$": [0.5, 1.0, 1.5, 2.0],
    "restriction $\\gamma(y-g) > g$": [γ_b*(y_b - g) > g for g in (0.5, 1.0, 1.5, 2.
 ↪0)],
    "stationary $\\pi$": [brock_π(g) for g in (0.5, 1.0, 1.5, 2.0)],
    "return on currency $1/\\pi$": [1/brock_π(g) for g in (0.5, 1.0, 1.5, 2.0)],
}).round(5)
```

```
   $g$  restriction $\gamma(y-g) > g$  stationary $\pi$  \
0  0.5                          True           1.00238
1  1.0                          True           1.00529
2  1.5                          True           1.00893
3  2.0                          True           1.01361

   return on currency $1/\pi$
0                     0.99762
1                     0.99474
2                     0.99115
3                     0.98658
```

But the model *also* has a continuum of non-stationary equilibria, indexed by initial price levels below the stationary one, in all of which the gross inflation rate converges to β — that is, the economy ends in **deflation**, with the return on currency approaching $1/\beta = 1.05$ and real balances exploding.

These exist because the demand for currency implied by $\gamma \ln(m/p)$ is so elastic with respect to the rate of return that the government can raise enough seigniorage to finance g even while paying interest on currency through deflation.

Applying least squares learning here selects the *classical stationary* equilibrium, just as in the overlapping generations model.

The difference is the welfare ranking: in the Brock model, the non-stationary equilibria all Pareto-dominate the classical stationary one that learning picks.

So least squares dynamics reliably select the *classical* equilibrium.

Whether that equilibrium is the good one is a separate question, and the answer depends on the model.

85.5 A government learning the Phillips curve

The examples so far put adaptive *households* into a monetary economy.

We close with an application in which the adaptive agent is the **government**, learning an econometric relationship it can act on.

The environment is different from the overlapping generations model — it is a Phillips curve economy — but the machinery is the same: an agent runs a regression, acts on it, and thereby helps generate the very data it is fitting.

This example matters for a further reason.

It is the one in Sargent [1993] that most directly seeded later work: Sims [1988] and Chung [1990] studied it, and it led Sargent to the escape-dynamics model of Sargent [1999], *The Conquest of American Inflation*, which the *Phillips curve lectures* study in depth.

85.5.1 Two stories about post-war inflation

There is a well-known account of the rise and fall of U.S. inflation after World War II.

In it, the private sector always had rational expectations and understood the natural-rate hypothesis, while the government, for a time, wrongly believed there was an *exploitable* Phillips curve, a lasting tradeoff between inflation and unemployment.

The government estimated Phillips curves on 1960s data, saw a tradeoff, and tried to buy lower unemployment with higher inflation.

The result was higher inflation with no lasting gain in employment: the Phillips curve shifted up adversely, as the natural-rate hypothesis says it must.

But there is a counter-story, told by economists who fit Phillips curves at the time.

They argued that their procedures were *adaptive*, that they detected the adverse shift quickly enough to give sound advice, and so need not have led the government astray.

The virtue of the model below is that a single specification nests both stories, with a parameter deciding which one plays out.

85.5.2 The model

Inflation is decomposed into an expected and an unexpected part,

$$\pi_t = g_{t-1} + \eta_t, \tag{85.9}$$

where g_{t-1} is the inflation the public expects (and which the government controls) and η_t is the public's forecast error, orthogonal to earlier information.

The *true* Phillips curve is the natural-rate one: only the **surprise** η_t moves unemployment:

$$U_t = U^* - \theta(\pi_t - g_{t-1}) + u_t = U^* - \theta\eta_t + u_t, \qquad \theta > 0. \tag{85.10}$$

The government does not know this.

It believes unemployment depends on the *level* of inflation, and fits the non-expectational regression

$$U_t = \alpha_0 + \alpha_1 \pi_t + \epsilon_t. \tag{85.11}$$

Each period it minimizes $\frac{1}{2}\mathbb{E}(U_t^2 + \pi_t^2)$ subject to its perceived model, which yields the myopic target

$$g_{t-1} = -\frac{\alpha_0\,\alpha_1}{1+\alpha_1^2}. \tag{85.12}$$

The system is **self-referential**: the government's beliefs α set its policy g through (85.12); the policy shapes the joint distribution of (π, U); and that distribution is what the government's regression (85.11) estimates.

```
U_star, θ_pc = 5.0, 1.0              # natural rate 5%, Phillips slope
σ_η, σ_u = 0.3, 0.3                  # inflation-surprise and unemployment shocks

def govt_target(α):
    "Government's myopic optimal target inflation."
    α0, α1 = α
    return -α0 * α1 / (1 + α1**2)
```

85.5.3 The consistent equilibrium

A self-confirming, or **consistent**, equilibrium is a belief vector α whose induced policy generates data whose regression returns the same α.

Under a constant policy g, the surprise η_t is the only thing moving inflation around its mean, so the population regression of U on π has slope exactly $-\theta$ and intercept $U^* + \theta g$.

Solving that fixed point with the decision rule (85.12) gives Kydland and Prescott's time-consistent outcome,

$$\alpha_0 = (1 + \theta^2)U^*, \qquad \alpha_1 = -\theta, \qquad g \to \theta U^*.$$

```
α_consistent = np.array([(1 + θ_pc**2) * U_star, -θ_pc])
g_consistent = θ_pc * U_star
print(f"consistent equilibrium beliefs α = {α_consistent},  target inflation g = {g_
↪consistent}")
print(f"optimal (Ramsey) outcome:                                    target inflation g␣
↪= 0")
```

```
consistent equilibrium beliefs α = [10. -1.],  target inflation g = 5.0
optimal (Ramsey) outcome:                                    target inflation g = 0
```

This is the **inflation bias**.

The government inflates at rate θU^* and gets nothing for it: because only surprises move unemployment, average unemployment is U^* either way.

Had the government understood the true model (85.10)**, it would have chosen** $g = 0$
the same unemployment with none of the inflation.

85.5.4 Constant-coefficient beliefs: learning the inflation bias

Suppose the government believes its coefficients are constant and estimates them by least squares, updating a little each period.

The deterministic path that least squares learning follows on average — its **mean dynamics** — moves beliefs toward the regression their current policy would induce.

```
def induced_beliefs(α):
    "The OLS fit that the current beliefs' policy would induce in a stationary sample.
↪"
    g = govt_target(α)
    return np.array([U_star + θ_pc * g, -θ_pc])          # slope is exactly -θ
```

(continues on next page)

(continued from previous page)

```
def mean_dynamics(α0=(0.0, 0.0), lr=0.2, T=60):
    α = np.array(α0, float)
    path = np.empty((T, 3))
    for t in range(T):
        α = α + lr * (induced_beliefs(α) - α)
        path[t] = [α[0], α[1], govt_target(α)]
    return path

path = mean_dynamics()

fig, ax = plt.subplots(figsize=(7.5, 4))
ax.plot(path[:, 2], 'C0', lw=1.6, label="target inflation $g$")
ax.plot(path[:, 0], 'C1', lw=1.2, label=r"belief $\alpha_0$")
ax.plot(path[:, 1], 'C2', lw=1.2, label=r"belief $\alpha_1$")
ax.axhline(g_consistent, color='C0', ls='--', lw=0.8)
ax.set_xlabel("iteration")
ax.legend(frameon=False)
plt.show()
```

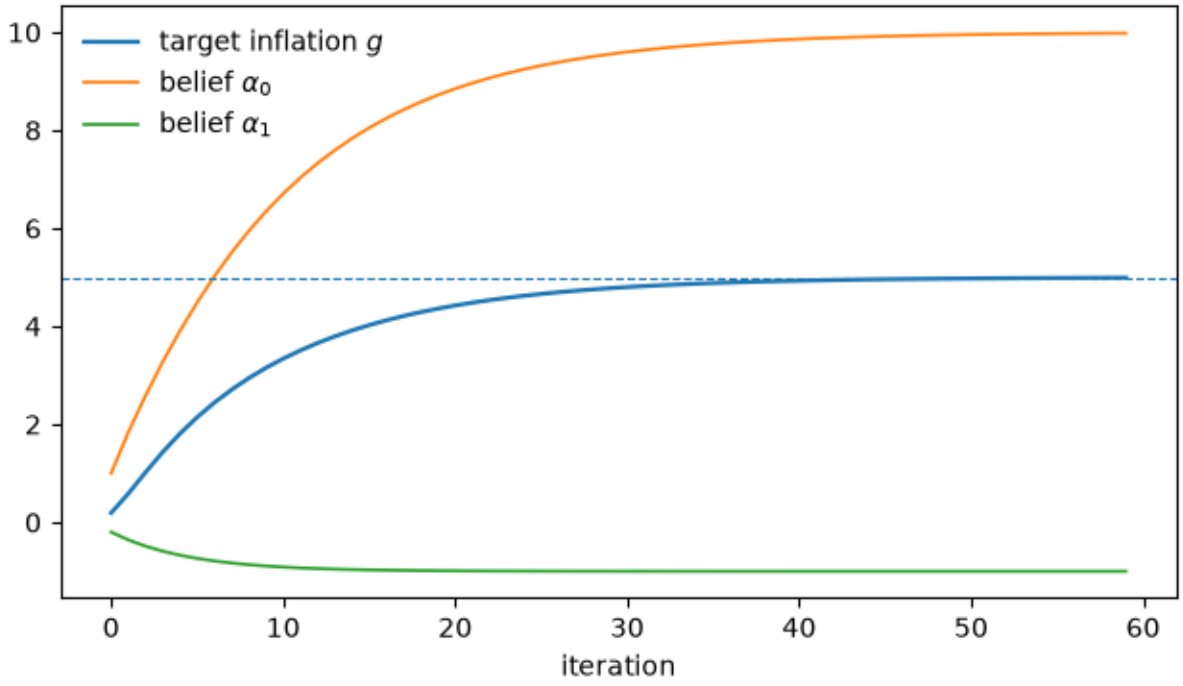


Fig. 85.4: Least squares learning converges to the inflation bias

Least squares learning walks the government straight into the consistent equilibrium: beliefs settle at $\alpha = ((1 + \theta^2)U^*, -\theta)$ and target inflation at $\theta U^* = 5$.

The government keeps inflating, period after period, at a rate that buys it nothing, because its own policy keeps producing data that confirm the tradeoff it believes in.

85.5.5 Random-coefficient beliefs: escaping toward the optimum

Sims [1988] and Chung [1990] then asked what happens if the government is *less* sure its coefficients are constant.

They let the government suspect the coefficients drift — a random walk in α — and estimate them with a Kalman filter, which discounts old data in favor of recent data.

That is a **constant-gain** algorithm: instead of a $1/t$ gain that eventually freezes the estimates, it keeps a fixed gain forever, so the government never stops paying attention to what just happened.

```
def constant_gain(T=40_000, gain=0.02, seed=1):
    "Government estimates a drifting-coefficient Phillips curve; returns target␣
 ↪inflation g_t."
    rng = np.random.default_rng(seed)
    α = α_consistent.copy()                                  # start at the inflation bias
    R = np.eye(2)
    g_path = np.empty(T)
    for t in range(T):
        g = govt_target(α)
        η = σ_η * rng.standard_normal()
        π = g + η
        U = U_star - θ_pc * η + σ_u * rng.standard_normal()    # true Phillips curve
        x = np.array([1.0, π])
        R = R + gain * (np.outer(x, x) - R)
        α = α + gain * np.linalg.solve(R, x * (U - x @ α))
        g_path[t] = g
    return g_path

g_path = constant_gain()

fig, ax = plt.subplots(figsize=(9, 4))
ax.plot(g_path, lw=0.5)
ax.axhline(g_consistent, color='C3', ls='--', lw=1, label="consistent equilibrium␣
 ↪(inflation bias)")
ax.axhline(0, color='C2', ls=':', lw=1, label="Ramsey / zero inflation")
ax.set_xlabel("$t$")
ax.set_ylabel("target inflation $g_t$")
ax.legend(frameon=False)
plt.show()
```

The constant-gain government does not settle at the inflation bias.

It climbs toward it, then abruptly **escapes** toward zero inflation, then drifts back up, then escapes again, in a recurring sawtooth.

The mechanism is the one Sims and Chung identified.

Because the government discounts old data, a run of observations in which inflation surprises are small and unemployment stays near U^* regardless of inflation quickly persuades it that the tradeoff is not there, and it stops exploiting a tradeoff it no longer believes in, dropping inflation toward zero.

The willingness to entertain drifting coefficients lets the government learn the truth of the natural-rate hypothesis *without having to sit through a long inflation to do it*.

How often the escapes happen depends on the gain — the government's degree of doubt about its own model — which is exactly the parameter Sims [1988] found to select between the two stories.

A small gain keeps the economy near the inflation bias; a larger gain sends it toward the optimum more often (Exercise 85.7.4).

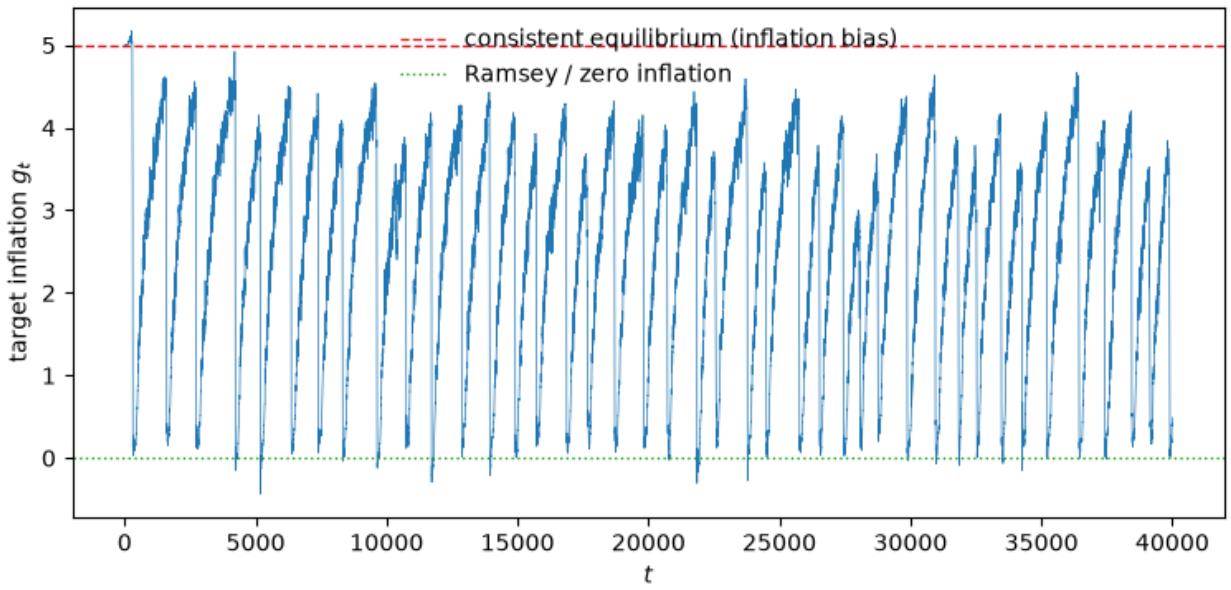


Fig. 85.5: Constant-gain learning and recurrent escapes to zero inflation

85.5.6 From escape dynamics to the *Conquest of American Inflation*

The two regimes are formalizations of the two stories we began with.

The consistent-equilibrium path is the natural-rate story: a government that trusts its mis-specified model inflates and stays stuck.

The escaping path is the counter-story: a government alert to model change detects the poor tradeoff and backs away from it.

This is the seed of a large literature.

Sims [1988] and Chung [1990] showed both that the model could nest the two stories and — in Chung's case — that it could be *econometrically estimated* on post-war U.S. data, imputing a genuine estimation procedure to the government inside the model.

Their work led Sargent [1999] to the **escape-route** model of *The Conquest of American Inflation*, in which recurrent escapes from the high-inflation Nash equilibrium toward the Ramsey outcome offer a theory of *how* U.S. inflation was actually conquered in the early 1980s.

The QuantEcon lectures *Escaping Nash Inflation* and *Adaptive Learning and Escape Dynamics* develop that model and its escape dynamics in full; the sawtooth above is a first glimpse of what they study.

85.6 Concluding remarks

Part 1 showed that a system of adaptive agents can find a rational expectations equilibrium it was never told about, and made visible how much the *complexity of the environment* governs the speed: a rare deficit state is learned about slowly, and a large state space forces a parametric shortcut that may put the equilibrium out of reach entirely.

Part 2 showed something stronger.

Where the model has two equilibria, the learning dynamics do not merely find one; they systematically pick out the one that the rational expectations dynamics reject, and laboratory subjects go the same way.

That is the best case in Sargent [1993] for treating adaptive dynamics as an equilibrium selection device.

The Brock counterexample marks its limit.

The selection is a fact about the algorithm, not a welfare theorem, and Sargent is candid about the resulting discomfort:

> I know that it is inconsistent to doubt the real-time dynamics but keep the equilibria selected by them. I confess that my affection for the selection performed in the monetary models described in Chapter 6 is partly driven by my prior conviction that the selected equilibria seem sensible to me.

The Phillips curve application sharpened the point in two ways.

It showed that *which* learning scheme finds the good outcome is model-dependent: here least squares walks into the bad equilibrium, and it is the constant-gain government — the one that never stops doubting — that escapes toward the good one.

And it turned the *gain* into the object of interest, foreshadowing the escape-dynamics research of Sargent [1999] taken up in *Escaping Nash Inflation* and *Adaptive Learning and Escape Dynamics*.

Money as a Medium of Exchange among Artificially Intelligent Agents takes the selection idea in a different direction, into a setting where the competing equilibria are not high and low inflation but different *monetary institutions*: which good becomes money.

85.7 Exercises

Exercise 85.7.1

Because agents learn a separate saving rate for each deficit state, they can only learn about a state as fast as that state occurs.

Compare two economies with $\bar{G} = (0.8, 0)$ that differ only in the transition matrix: one where the two states are equally likely, and one where the high-deficit state is rare.

For each, compute the stationary equilibrium, run the learning simulation, and report how far each state's learned saving rate is from its target along with how often that state was visited.

Solution

```
G_ex = np.array([0.8, 0.0])
cases = {"equally likely": np.array([[0.5, 0.5], [0.5, 0.5]]),
         "high deficit rare": np.array([[0.5, 0.5], [0.03, 0.97]])}

rows = []
for name, P in cases.items():
    s_star, _ = stationary_equilibrium(G_ex, P)
    ergodic = qe.MarkovChain(P).stationary_distributions[0]
    hist, _ = simulate(G_ex, P, T=30_000, s0=start)
    learned = hist[-1].mean(axis=0)
    for k in (0, 1):
        rows.append([name, k+1, ergodic[k], s_star[k], learned[k],
                     abs(learned[k] - s_star[k])])

pd.DataFrame(rows, columns=["chain", "state", "ergodic prob.",
                            "equilibrium $s_i$", "learned $s_i$", "error"]).
↪round(4)
```

```
                chain  state  ergodic prob.  equilibrium $s_i$  learned $s_i$  \
0     equally likely      1         0.5000             4.4965         4.5031
1     equally likely      2         0.5000             4.4965         4.5006
2  high deficit rare      1         0.0566             4.6828         4.6994
3  high deficit rare      2         0.9434             4.9646         4.9662

    error
0  0.0066
1  0.0041
2  0.0167
3  0.0016
```

When the two states are equally likely the two saving rules are learned about equally well.

When the high-deficit state occurs only about six percent of the time, the error in its rule is an order of magnitude larger than the error in the rule for the common state, even though both have had 30,000 periods of calendar time to settle.

Sargent's comment is worth keeping in mind: in terms of *unconditional expected utility*, failing to learn what to do in a rarely visited state may cost very little.

The cost of slow learning is not proportional to the size of the error.

Note also that changing the transition matrix changes the equilibrium itself — beliefs about the persistence of deficits feed straight into (85.5) — so the comparison has to be against a separately recomputed target for each chain.

Exercise 85.7.2

The gain sequence $\gamma_\tau = 1/\tau$ is what makes (85.6) converge.

An agent who suspected the environment might be drifting would use a **constant gain** instead, weighting recent experience more heavily forever.

Modify the simulation to use a constant gain and describe what happens to the limiting behavior of the saving rules.

Compare gains of 0.02 and 0.005 against the $1/\tau$ benchmark.

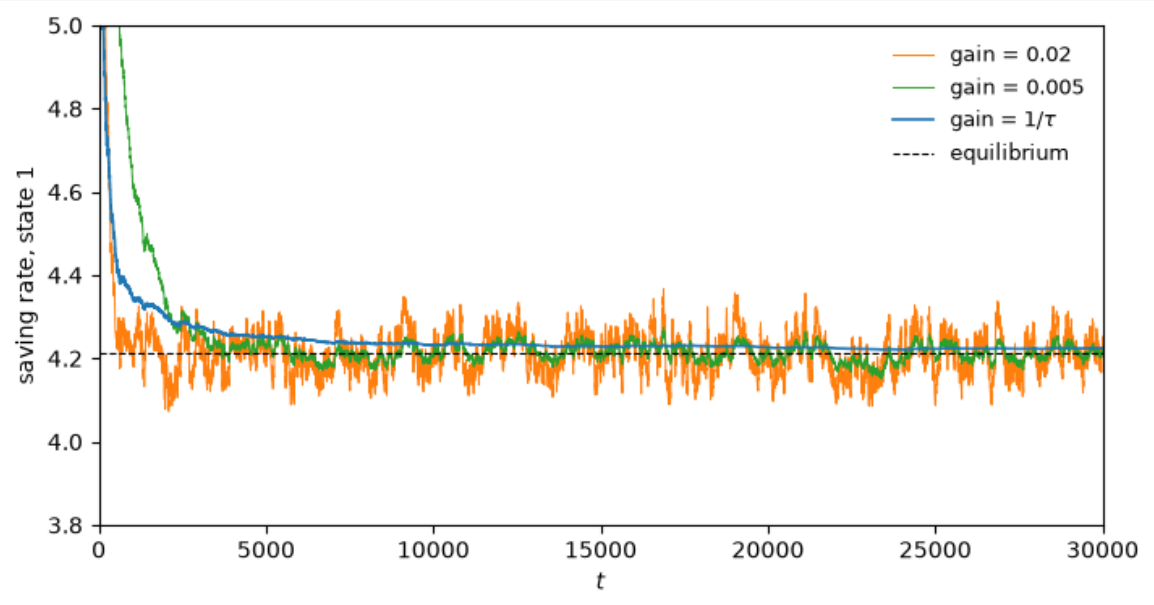
Solution

```
def simulate_constant_gain(G, P, gain, T=30_000, s0=None, seed=0, band=0.4):
    rng = np.random.default_rng(seed)
    n = len(G)
    s = np.array(s0, float)
    M = np.array([[U_double(s[j, k], 1.0) for k in range(n)] for j in range(2)])
    floor = G/N + band
    hist = np.empty((T, 2, n))
    i, prev = 0, None
    for t in range(T):
        j = t % 2
        s_t = s[j, i]
        if prev is not None:
            j_p, i_p, s_p = prev
            R = (N*s_t - G[i]) / (N*s_p)
            M[j_p, i_p] += gain * (U_double(s_p, R) - M[j_p, i_p])
            step = gain * U_prime(s_p, R) / M[j_p, i_p]
            s[j_p, i_p] = np.clip(s[j_p, i_p] - step, floor[i_p], w1 - 0.4)
```

```
        prev = (j, i, s_t)
        hist[t] = s
        i = rng.choice(n, p=P[i])
    return hist

rows = []
for label, hist in [("gain = 0.02", simulate_constant_gain(G2, P2, 0.02,␣
↪s0=start)),
                    ("gain = 0.005", simulate_constant_gain(G2, P2, 0.005,␣
↪s0=start)),
                    ("gain = 1/τ", simulate(G2, P2, T=30_000, s0=start)[0])]:
    tail = hist[-5_000:].mean(axis=1)        # average the two classes
    rows.append([label, *tail.mean(axis=0), *tail.std(axis=0)])

pd.DataFrame(rows, columns=["", "mean $s_1$", "mean $s_2$",
                            "s.d. $s_1$", "s.d. $s_2$"]).set_index("").round(5)
```

	mean s_1	mean s_2	s.d. s_1	s.d. s_2
gain = 0.02	4.20862	4.36167	0.04089	0.04015
gain = 0.005	4.20937	4.34929	0.01391	0.02159
gain = 1/τ	4.22312	4.37010	0.00057	0.00098

```
fig, ax = plt.subplots(figsize=(8, 4))
for gain, colour in [(0.02, 'C1'), (0.005, 'C2')]:
    h = simulate_constant_gain(G2, P2, gain, s0=start)
    ax.plot(h[:, 0, 0], color=colour, lw=0.7, label=f"gain = {gain}")
h, _ = simulate(G2, P2, T=30_000, s0=start)
ax.plot(h[:, 0, 0], color='C0', lw=1.2, label=r"gain = $1/\tau$")
ax.axhline(s2[0], color='k', lw=0.8, ls='--', label="equilibrium")
ax.set_xlim(0, 30_000)
ax.set_ylim(3.8, 5.0)
ax.set_xlabel("$t$")
ax.set_ylabel("saving rate, state 1")
ax.legend(frameon=False, fontsize=9)
plt.show()
```



The constant-gain rules do not converge.

They settle into a *neighborhood* of the equilibrium and then keep rattling around inside it forever, because each new observation is always given the same weight and so never stops moving the estimate.

Shrinking the gain tightens the band but does not eliminate it.

Only a gain that declines to zero — like $1/\tau$ — delivers convergence to a point, which is why the convergence theorems for these systems all require it.

What the constant-gain agent buys in exchange is adaptability: if the deficit process were to change, the $1/\tau$ agent would barely notice, while the constant-gain agent would track the change.

That trade-off reappears throughout this literature.

Exercise 85.7.3

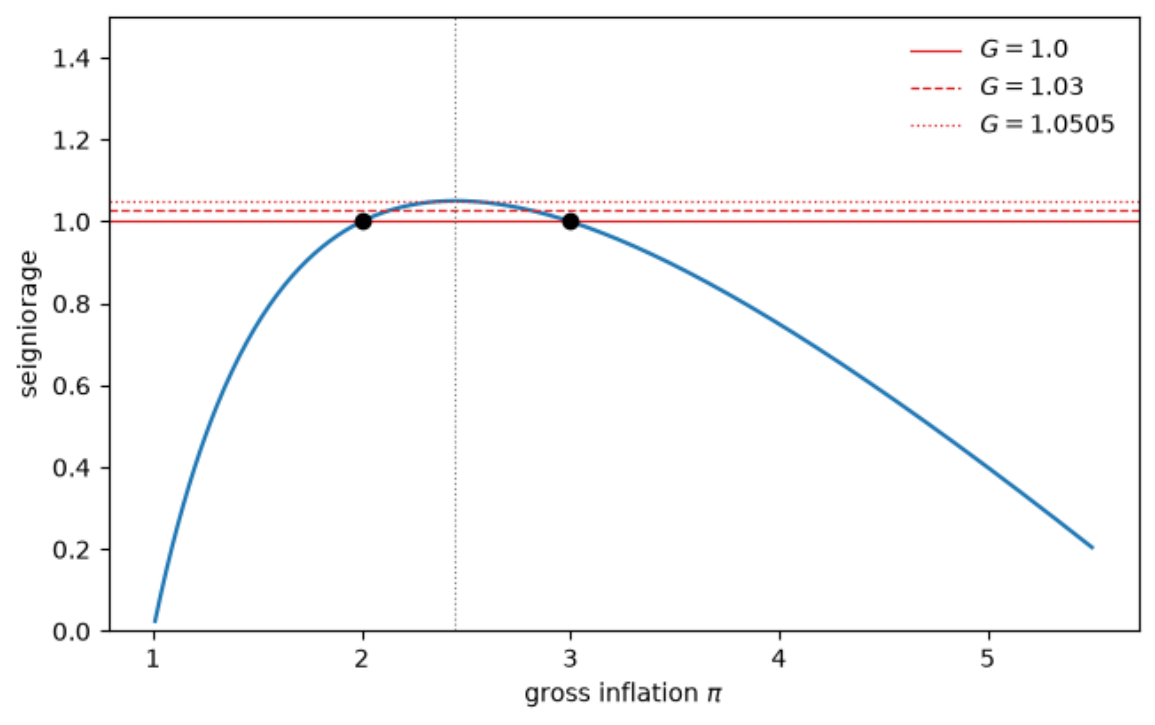
The claim that the high-inflation steady state sits on "the wrong side of a Laffer curve" can be made precise.

In a stationary equilibrium with gross inflation π, real balances are $h = (w_1 - w_2\pi)/2$ and the government collects seigniorage $h(1 - 1/\pi)$.

Plot seigniorage as a function of π, mark the two stationary equilibria, and use the picture to explain why raising G raises the low stationary inflation rate but *lowers* the high one, and what happens when G gets too large.

Solution

```
def seigniorage(π):
    return (w1_d - w2_d*π)/2 * (1 - 1/π)

π_grid = np.linspace(1.01, 5.5, 500)
peak = π_grid[np.argmax(seigniorage(π_grid))]

fig, ax = plt.subplots(figsize=(7.5, 4.5))
ax.plot(π_grid, seigniorage(π_grid), lw=1.6)
for G_try, style in [(1.0, '-'), (1.03, '--'), (1.0505, ':')]:
    ax.axhline(G_try, color='C3', lw=0.9, ls=style, label=f"$G = {G_try}$")
ax.plot([π_low, π_high], [seigniorage(π_low), seigniorage(π_high)], 'ko', ms=6)
ax.axvline(peak, color='gray', lw=0.8, ls=':')
ax.set_xlabel(r"gross inflation $\pi$")
ax.set_ylabel("seigniorage")
ax.set_ylim(0, 1.5)
ax.legend(frameon=False)
plt.show()

print(f"peak of the Laffer curve at π = {peak:.4f}, raising {seigniorage(peak):.4f}
 ↪")
print(f"the two steady states raise {seigniorage(π_low):.4f} and {seigniorage(π_
 ↪high):.4f}")
```




```
peak of the Laffer curve at π = 2.4497, raising 1.0505
the two steady states raise 1.0000 and 1.0000
```

The two stationary equilibria are the two inflation rates at which the inflation-tax Laffer curve crosses the required revenue G, one on each side of the peak.

The low one sits on the rising branch, so financing a larger deficit there requires *more* inflation: the classical comparative static.

The high one sits on the falling branch, where more inflation raises *less* revenue, so a larger deficit is financed by a *lower* stationary inflation rate, the anti-classical comparative static that makes the high equilibrium so awkward.

As G rises the horizontal line moves up and the two intersections converge.

Past the peak of the curve there is no stationary equilibrium at all: the deficit exceeds the maximum revenue the inflation tax can raise.

```
rows = []
for G_try in (0.8, 1.0, 1.04, 1.05, 1.0505, 1.06):
    A1_try = (w1_d + w2_d - 2*G_try)/w2_d
    disc = A1_try**2 - 4*A2
    if disc < 0:
        rows.append([G_try, np.nan, np.nan, "none"])
    else:
        r = np.sort(np.roots([1, -A1_try, A2]))
        rows.append([G_try, r[0], r[1], "two"])

pd.DataFrame(rows, columns=["$G$", "low $\\pi$", "high $\\pi$",
                            "stationary equilibria"]).round(4)
```

	G	low π	high π	stationary equilibria
0	0.8000	1.5642	3.8358	two
1	1.0000	2.0000	3.0000	two
2	1.0400	2.2328	2.6872	two
3	1.0500	2.4000	2.5000	two
4	1.0505	2.4424	2.4566	two
5	1.0600	NaN	NaN	none

The two roots move toward each other as G rises and collide at $G \approx 1.0505$, after which the model has no stationary equilibrium at all.

That number is not a coincidence.

Setting the discriminant of (85.8)'s fixed-point equation to zero gives a double root at $\pi = \sqrt{A_2} = \sqrt{w_1/w_2}$, and the deficit at which it occurs is exactly the peak of the Laffer curve computed above.

```
G_max = (w1_d + w2_d - 2*np.sqrt(A2)*w2_d) / 2
print(f"roots collide at G       = {G_max:.6f},  where π = √(w1/w2) = {np.
 ↪sqrt(A2):.6f}")
print(f"peak of the Laffer curve = {seigniorage(np.sqrt(A2)):.6f}  at π = {np.
 ↪sqrt(A2):.6f}")
```

```
roots collide at G       = 1.050510,  where π = √(w1/w2) = 2.449490
peak of the Laffer curve = 1.050510  at π = 2.449490
```

The largest deficit the government can finance is the maximum of the inflation tax, and at that deficit the two stationary equilibria have merged into one.

Exercise 85.7.4

In the Phillips curve application, the constant-gain government escapes toward zero inflation, and the lecture claims that *how often* it escapes depends on the gain, the government's degree of doubt about its own model.

Make the claim quantitative.

For a range of gains, simulate the constant-gain economy and measure both the average target inflation and the fraction of time spent near the Ramsey outcome (say, $g_t < 1$).

Which way does more doubt push the economy, and how does this connect to the two stories about post-war inflation?

Solution

```
rows = []
for gain in (0.005, 0.01, 0.02, 0.04, 0.08):
    tails = [constant_gain(T=30_000, gain=gain, seed=s)[-20_000:] for s in␣
 ↪range(4)]
    mean_g = np.mean([g.mean() for g in tails])
    near_ramsey = np.mean([np.mean(g < 1) for g in tails])
    rows.append([gain, mean_g, near_ramsey])

pd.DataFrame(rows, columns=["gain", "mean target inflation", "fraction of time $g
 ↪< 1$"]
             ).set_index("gain").round(3)
```

gain	mean target inflation	fraction of time $g < 1$
0.005	3.221	0.121
0.010	2.886	0.148
0.020	2.485	0.174
0.040	2.174	0.189
0.080	1.820	0.210

More doubt — a larger gain — pushes the economy away from the inflation bias and toward the optimum: mean inflation falls and the economy spends a larger fraction of its time near the Ramsey outcome.

The intuition is that a higher gain discounts old data more heavily, so the government reacts faster to the run of observations that reveals the tradeoff to be illusory, and escapes sooner and more often.

This is exactly the parameter Sims [1988] found to select between the two stories.

A government confident in its constant-coefficient model (small gain) is the natural-rate story's government, stuck inflating; a government alert to model change (large gain) is the counter-story's, detecting the poor tradeoff and stepping away from it.

The same gain reappears as the central object in Sargent [1999], where the *rate* of escape from the high-inflation equilibrium governs how quickly an inflation can be conquered.

CHAPTER

EIGHTYSIX

LEARNING, APPROXIMATION, AND EQUILIBRIUM COMPUTATION

Contents

- *Learning, Approximation, and Equilibrium Computation*
  - *Overview*
  - *A continuous-state functional equation*
  - *A benchmark solution*
  - *Parameterized expectations*
  - *Learning is equilibrium computation*
  - *Learning without a functional form*
  - *Concluding remarks*
  - *Exercises*

86.1 Overview

In *Adaptive Agents in an Overlapping Generations Monetary Economy* a system of adaptive agents found a rational expectations equilibrium it was never told about.

The agents there learned a *separate* saving rate for each level of the government deficit — a **non-parametric** rule, one number per state.

That works when there are two states.

It runs into two walls otherwise, both noted in Sargent [1993]:

- a state that occurs rarely is learned about slowly, because its observations arrive slowly;
- when the number of states is large — and especially when the state is *continuous* — one parameter per state is hopeless.

The econometrician's response is to impose a **parametric** form on the decision rule, $s = f(G, \theta)$, with θ of low dimension, and use every observation to estimate θ.

This lecture works through what that buys and what it costs.

The costs and benefits are two sides of one idea.

A parametric family can be learned quickly because every observation informs every state.

But a learning scheme confined to a family of functions can converge to a rational expectations equilibrium *only if some member of the family supports one*.

Otherwise the best it can reach is an **approximate equilibrium**.

Working this out surfaces a theme that runs through the whole bounded-rationality program:

> Learning algorithms and equilibrium computation algorithms look like each other.

We will make that concrete.

Marcet's method of parameterized expectations — a standard tool for *computing* rational expectations equilibria — turns out, when written recursively, to be exactly a model of adaptive agents learning.

An equilibrium computation is a centralized learning algorithm run by the modeller; a learning economy is a decentralized equilibrium computation run by the agents.

The plan:

1. Put a *continuous* deficit into the overlapping generations model of the previous lecture, turning the equilibrium condition into a **functional equation** in the saving rule $f(G)$.
2. Compute a benchmark solution, so we have ground truth to measure against.
3. Solve it by **parameterized expectations** with families of increasing richness, and watch approximate equilibria give way to the exact one.
4. Show that the recursive, real-time version of the same algorithm **is** a learning economy.
5. Learn the saving rule **non-parametrically** with a kernel estimator, and weigh the trade-off against the parametric route.

Let's start with some imports.

```
import numpy as np
import pandas as pd
import quantecon as qe
import matplotlib.pyplot as plt
from scipy.optimize import brentq
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[1], line 3
      1 import numpy as np
      2 import pandas as pd
----> 3 import quantecon as qe
      4 import matplotlib.pyplot as plt
      5 from scipy.optimize import brentq

ModuleNotFoundError: No module named 'quantecon'
```

86.2 A continuous-state functional equation

We keep the overlapping generations monetary economy of *Adaptive Agents in an Overlapping Generations Monetary Economy* — two-period lived agents, log utility, endowments w_1 when young and w_2 when old, currency the only store of value, and a government financing a deficit by printing money.

The one change: the deficit G_t now follows a **continuous** Markov process, with transition kernel $F(G', G) = \text{Prob}\{G_{t+1} \le G' \mid G_t = G\}$.

We look for a stationary equilibrium in which saving is a *function* of the current deficit, $s_t = f(G_t)$.

The household's first-order condition, evaluated at $s_t = f(G_t)$, is

$$u'(w_1 - f(G_t)) = \mathbb{E}_t\big[u'(w_2 + f(G_t)R_t)\, R_t\big], \qquad \mathbb{E}_t(\cdot) = \mathbb{E}(\cdot \mid G_t),$$

and the government budget constraint with market clearing gives the return on currency in terms of the saving rule, exactly as before (with $N = 1$),

$$R_t = \frac{f(G_{t+1}) - G_{t+1}}{f(G_t)}.$$

Substituting R_t, and using $f(G_t)R_t = f(G_{t+1}) - G_{t+1}$ for old-age consumption, turns the first-order condition into a **functional equation** in f:

$$u'(w_1 - f(G_t)) = \mathbb{E}_t\left[u'(w_2 + f(G_{t+1}) - G_{t+1}) \cdot \frac{f(G_{t+1}) - G_{t+1}}{f(G_t)}\right]. \tag{86.1}$$

Under log utility, $u'(c) = 1/c$, and because $f(G_t)$ is known at t it pulls out of the expectation.

Writing

$$\psi(G_t) \equiv \mathbb{E}_t\big[u'(w_2 + f(G_{t+1}) - G_{t+1})\,(f(G_{t+1}) - G_{t+1})\big], \tag{86.2}$$

equation (86.1) becomes

$$\frac{1}{w_1 - f(G_t)} = \frac{\psi(G_t)}{f(G_t)}, \qquad \text{i.e.} \qquad f(G_t) = \frac{w_1\,\psi(G_t)}{1 + \psi(G_t)}.$$

So the whole problem reduces to finding the **conditional expectation** $\psi(G)$: once we have it, the saving rule follows in closed form.

That observation is the hinge of the entire lecture.

Every method below — parametric or not — is a way of estimating the one object $\psi(G)$.

```
w1, w2 = 20.0, 10.0

def s_from_psi(ψ):
    "Recover the saving rule from the conditional expectation ψ."
    return w1 * ψ / (1 + ψ)
```

86.2.1 The deficit process

We take $\log G_t$ to follow an AR(1), $\log G_{t+1} = (1-\rho)\log \bar{G} + \rho \log G_t + \sigma\varepsilon_{t+1}$, and discretize it with the method of Tauchen [1986] to get a fine grid for the benchmark.

The parameters keep the deficit modest enough that a monetary equilibrium exists at every state.

```
ρ, σ, G_bar = 0.7, 0.30, 0.6

mc = qe.tauchen(21, ρ, σ, mu=np.log(G_bar) * (1 - ρ), n_std=2.5)
G, P = np.exp(mc.state_values), mc.P
ergodic = mc.stationary_distributions[0]      # where the deficit spends its time

print(f"deficit grid: {len(G)} points from {G.min():.3f} to {G.max():.3f}")
print(f"mean deficit under the ergodic distribution: {ergodic @ G:.3f}")
```

86.3 A benchmark solution

To measure any learning scheme, we first need the true $f(G)$.

On the discrete grid, (86.1) is a system of equations that we solve by **iterating the perceived-to-actual map**, precisely the T map of *A Peculiar Definition of Bounded Rationality*.

Guess a saving rule; use it on the right-hand side to compute the actual saving each state would call forth; repeat until the two agree.

Because ψ is monotone we can invert the first-order condition state by state with a bracketing root-finder, which makes the iteration robust.

```
def solve_benchmark(G, P, damp=0.5, tol=1e-12, max_iter=5000):
    "Fixed point of the perceived-to-actual saving map on the deficit grid."
    n = len(G)
    s = np.full(n, (w1 - w2) / 2)
    for it in range(max_iter):
        s_new = np.empty(n)
        for i in range(n):
            # E_t[ u'(c2) (s' - G') ] under the perceived rule s
            expect = sum((s[j] - G[j]) / (w2 + (s[j] - G[j])) * P[i, j] for j in␣
↪range(n))
            # solve  1/(w1 - s_i) = expect / s_i  for s_i in (G_i, w1)
            foc = lambda si: 1/(w1 - si) - expect/si
            s_new[i] = brentq(foc, G[i] + 1e-9, w1 - 1e-9)
        if np.max(np.abs(s_new - s)) < tol:
            return s, it
        s = damp * s_new + (1 - damp) * s
    return s, it


s_benchmark, iters = solve_benchmark(G, P)
print(f"converged in {iters} iterations")


def foc_residual(s_rule):
    "Max |first-order-condition error| of a saving rule under the true kernel P."
    n = len(G)
    err = []
    for i in range(n):
        rhs = sum((s_rule[j] - G[j]) / (w2 + (s_rule[j] - G[j])) / s_rule[i] * P[i, j]
                  for j in range(n))
        err.append(1/(w1 - s_rule[i]) - rhs)
    return np.max(np.abs(err))
```

(continues on next page)

(continued from previous page)

```
print(f"benchmark FOC residual: {foc_residual(s_benchmark):.2e}")
print(f"saving rule f(G) runs from {s_benchmark.max():.3f} (low deficit) "
      f"to {s_benchmark.min():.3f} (high deficit)")
```

```
fig, ax = plt.subplots(figsize=(6.5, 4))
ax.plot(G, s_benchmark, 'k-', lw=2)
ax.set_xlabel("deficit $G$")
ax.set_ylabel("saving $f(G)$")
plt.show()
```

Saving falls as the deficit rises: a larger deficit means faster money creation, a poorer return on currency, and less saving.

The rule is gently curved, which will matter in a moment.

This $f(G)$ is the object no adaptive agent gets to see.

Everything below tries to recover it.

86.4 Parameterized expectations

Marcet's **method of parameterized expectations** attacks (86.1) by putting a parametric form on the conditional expectation $\psi(G)$ from (86.2),

$$\psi(G) \approx \psi(G, \theta) = \phi(G)^\top \theta,$$

where $\phi(G)$ is a vector of basis functions and θ a short vector of coefficients.

The algorithm is a fixed-point iteration on θ:

1. Given θ, the saving rule is $f(G) = w_1 \psi(G, \theta)/(1 + \psi(G, \theta))$.
2. Simulate a long path of the deficit and compute the implied saving each period.
3. Form the *realized* value of the object inside (86.2), $y_t = u'(w_2 + s_{t+1} - G_{t+1})(s_{t+1} - G_{t+1})$, and **regress** it on $\phi(G_t)$ to get a new θ.
4. Iterate to convergence.

The regression in step 3 is doing the work of the conditional expectation: least squares projects the realized y_t onto functions of the current state, which is exactly what $\mathbb{E}_t[\cdot \mid G_t]$ is.

We use monomials in a rescaled $\log G$ as the basis, and vary the degree.

```
G_min, G_max = G.min(), G.max()

def simulate_deficit(T, seed=0):
    "Simulate the AR(1) deficit, clipped to the benchmark support."
    rng = np.random.default_rng(seed)
    μ = np.log(G_bar) * (1 - ρ)
    lg = np.empty(T)
    lg[0] = np.log(G_bar)
    for t in range(1, T):
        lg[t] = μ + ρ * lg[t-1] + σ * rng.standard_normal()
    return np.clip(np.exp(lg), G_min, G_max)

def basis(G_vals, degree):
```

(continues on next page)

(continued from previous page)

```
    "Monomials in log-deficit rescaled to [-1, 1]."
    x = 2 * (np.log(G_vals) - np.log(G_min)) / (np.log(G_max) - np.log(G_min)) - 1
    return np.vstack([x**k for k in range(degree + 1)]).T


def parameterized_expectations(degree, T=40_000, n_iter=200, seed=0):
    "Batch parameterized expectations; returns the implied saving rule on the grid."
    G_sim = simulate_deficit(T + 1, seed)
    Φ = basis(G_sim, degree)
    θ = np.zeros(degree + 1)
    θ[0] = 2.0
    for _ in range(n_iter):
        ψ = np.clip(Φ @ θ, 0.05, 50)
        s = np.clip(s_from_psi(ψ), G_sim + 0.05, w1 - 0.05)
        y = (s[1:] - G_sim[1:]) / (w2 + (s[1:] - G_sim[1:]))      # realized regressand
        θ_new = np.linalg.lstsq(Φ[:-1], y, rcond=None)[0]
        if np.max(np.abs(θ_new - θ)) < 1e-11:
            θ = θ_new
            break
        θ = 0.4 * θ + 0.6 * θ_new
    ψ_grid = np.clip(basis(G, degree) @ θ, 0.05, 50)
    return s_from_psi(ψ_grid)
```

86.4.1 Approximate versus exact equilibria

Now run it for families of increasing richness: a constant ψ (saving independent of the deficit), a linear one, and a quadratic one.

```
rules = {deg: parameterized_expectations(deg) for deg in (0, 1, 2)}

fig, ax = plt.subplots(figsize=(7, 4.5))
ax.plot(G, s_benchmark, 'k-', lw=2.2, label="benchmark $f(G)$")
labels = {0: "degree 0 (constant)", 1: "degree 1 (linear)", 2: "degree 2 (quadratic)"}
for deg, colour in [(0, 'C0'), (1, 'C1'), (2, 'C2')]:
    ax.plot(G, rules[deg], '--', color=colour, lw=1.4, label=labels[deg])
ax.set_xlabel("deficit $G$")
ax.set_ylabel("saving $s$")
ax.legend(frameon=False)
plt.show()
```

The **constant** family cannot represent a saving rule that depends on the deficit at all, so the best it can do is a flat line through the middle of the data.

That flat line *is* an approximate equilibrium — a fixed point of the learning scheme — but it satisfies the first-order condition only on average, not state by state.

The **linear** family does much better but bends the wrong way at the extremes.

The **quadratic** family is visually indistinguishable from the benchmark: because the true $f(G)$ is very nearly quadratic here, a three-parameter rule essentially nails it.

We can quantify "how close" two ways: by the distance to the benchmark, and by the residual in the first-order condition under the true kernel, the honest measure of how far an approximate equilibrium is from an exact one.

```
rows = []
for deg in (0, 1, 2):
```

(continues on next page)

(continued from previous page)

```
    s_hat = rules[deg]
    sup = np.max(np.abs(s_hat - s_benchmark))
    rms = np.sqrt(ergodic @ (s_hat - s_benchmark)**2)
    rows.append([labels[deg], sup, rms, foc_residual(s_hat)])

pd.DataFrame(rows, columns=["family", "sup $|s - f|$",
                            "ergodic RMS", "FOC residual"]).set_index("family").
↪round(4)
```

Every column falls as the family grows richer.

The first-order-condition residual — the quantity that is exactly zero at a rational expectations equilibrium and positive at an approximate one — drops by more than an order of magnitude from the constant family to the quadratic.

Note

Richer is not *always* better with finite data.

Push the degree higher and the extra terms start chasing sampling noise in the tails of the deficit distribution, where observations are scarce, and the sup-norm error can tick back up even as the fit improves where agents actually spend their time.

That is the same phenomenon as the slowly-learned rare state in *Adaptive Agents in an Overlapping Generations Monetary Economy*, seen from the approximation side: a scheme fits well where the data is, and the cost of misfitting a rarely-visited region is small in terms of expected utility.

Exercise 86.8.1 explores this.

86.5 Learning is equilibrium computation

So far parameterized expectations is an algorithm *we* run to compute an equilibrium.

Now comes the point of the whole lecture.

Write the same algorithm **recursively** — updating θ once per period as a single new observation arrives, rather than re-regressing a whole simulated panel — and it becomes a model of *adaptive agents learning in real time*.

The recursive form is ordinary recursive least squares:

$$\begin{aligned} R_{t+1} &= R_t + \tfrac{1}{t}(\phi(G_t)\phi(G_t)^\top - R_t), \\ \theta_{t+1} &= \theta_t + \tfrac{1}{t} R_{t+1}^{-1}\phi(G_t)(y_t - \phi(G_t)^\top \theta_t), \end{aligned} \tag{86.3}$$

where y_t is the same realized regressand as before and R_t tracks the second moment of the regressors.

This is the identical object we met in *Adaptive Agents in an Overlapping Generations Monetary Economy*: a stochastic-approximation recursion with a $1/t$ gain.

The only difference from the state-by-state learning there is that θ indexes a *parametric* rule, so a single observation updates the saving rule everywhere at once.

```
def online_pea(degree=2, T=500_000, seed=0, ridge=1e-3):
    """
    Recursive-least-squares parameterized expectations: the learning economy.

    Young agents at t save according to the current θ; next period the Euler
```

(continues on next page)

(continued from previous page)

```
    equation's realized value arrives and θ is updated once.  Returns the θ
    path and the final implied saving rule.
    """
    G_sim = simulate_deficit(T, seed)
    θ = np.array([2.0] + [0.0] * degree)
    R = ridge * np.eye(degree + 1)

    def saving(g, θ):
        ψ = np.clip(basis(np.array([g]), degree)[0] @ θ, 0.05, 50)
        return np.clip(s_from_psi(ψ), g + 0.05, w1 - 0.05)

    x_prev = None
    θ_path = np.empty((T, degree + 1))
    for t in range(T):
        s_t = saving(G_sim[t], θ)
        if x_prev is not None:                                  # date t-1 Euler␣
↪realizes now
            y = (s_t - G_sim[t]) / (w2 + (s_t - G_sim[t]))
            gain = 1.0 / (t + 1)
            R += gain * (np.outer(x_prev, x_prev) - R)
            θ = θ + gain * np.linalg.solve(R, x_prev * (y - x_prev @ θ))
        x_prev = basis(np.array([G_sim[t]]), degree)[0]
        θ_path[t] = θ

    s_grid = np.array([saving(g, θ) for g in G])
    return θ_path, s_grid


θ_path, s_online = online_pea()
print(f"online PEA: sup error = {np.max(np.abs(s_online - s_benchmark)):.4f},  "
      f"ergodic RMS = {np.sqrt(ergodic @ (s_online - s_benchmark)**2):.4f}")
```

```
fig, axes = plt.subplots(1, 2, figsize=(11, 4.2))

axes[0].plot(θ_path[:, 0], label=r"$\theta_0$", lw=2)
axes[0].plot(θ_path[:, 1], label=r"$\theta_1$", lw=2)
axes[0].plot(θ_path[:, 2], label=r"$\theta_2$", lw=2)
axes[0].set_xscale('log')
axes[0].set_xlabel("$t$  (log scale)")
axes[0].set_ylabel(r"$\theta_t$")
axes[0].set_title("coefficients learned in real time")
axes[0].legend(frameon=False, fontsize=9)

axes[1].plot(G, s_benchmark, 'k-', lw=2.2, label="benchmark $f(G)$")
axes[1].plot(G, s_online, 'C3--', lw=1.5, label="online PEA")
axes[1].set_xlabel("deficit $G$")
axes[1].set_ylabel("saving $s$")
axes[1].set_title("saving rule after learning")
axes[1].legend(frameon=False)
plt.tight_layout()
plt.show()
```

The coefficients settle down, and the saving rule they imply tracks the benchmark closely.

The small remaining gap is the signature of a scheme that has not quite finished converging.

Convergence is *slow*: the $1/t$ gain guarantees it but does not hurry it, and the online scheme takes hundreds of thousands of periods to reach an accuracy the batch algorithm attains in a handful of passes.

That gap is exactly the difference between an agent constrained to learn from experience as it arrives and a modeller free to re-use a whole simulated history at once.

But the limit is the same object.

Sargent's summary:

> Learning algorithms and equilibrium computation algorithms look like each other. Equilibrium computation algorithms often have interpretations as centralized learning algorithms whereby the model builder, acting in a role of 'social planner,' gropes for a set of pricing functions for markets and decision rules for agents that will satisfy all of the individual optimum conditions and market-clearing conditions. We have also seen that learning systems with boundedly rational agents sometimes have interpretations as decentralized equilibrium computation algorithms.

This is why Marcet arrived at parameterized expectations as a *computational* method by way of earlier work on the *dynamics of least squares learning*.

The two are the same recursion read in two directions.

86.6 Learning without a functional form

The parametric route is fast but stakes everything on the family.

If no member of $\{f(\cdot,\theta)\}$ supports an equilibrium, the scheme converges to an approximate equilibrium and stops.

The **non-parametric** alternative imposes no functional form.

Following the recursive kernel estimators of Chen and White [1998], an agent estimates the conditional expectation $\psi(G)$ directly from a kernel-weighted average of past realized values, letting the data choose the shape.

The recursive kernel density estimator is itself a stochastic-approximation recursion,

$$\hat{F}_t(x) = \hat{F}_{t-1}(x) + \tfrac{1}{t}\left[K\left(\tfrac{x-x_t}{h_t}\right) - \hat{F}_{t-1}(x)\right],$$

with a bandwidth $h_t \searrow 0$, the same $1/t$-gain shape as everything else in this lecture, now updating an entire estimated density rather than a finite parameter vector.

What we implement below is the **batch** counterpart of that recursion: a **Nadaraya–Watson** estimate of $\psi(G)$, formed as a kernel-weighted average of the realized regressand y over a whole simulated history, with weights that fall off with distance in $\log G$, and iterated to a fixed point.

Reading it in batch form keeps the comparison clean, because the parametric scheme we are measuring it against is also in batch form.

The recursive display above stands to this smoother exactly as (86.3) stands to the batch algorithm of the previous section: same estimator, read as a real-time learning rule rather than as a computation.

```
def kernel_smoother(T=400_000, seed=0, h=0.06, n_iter=40, damp=0.5):
    """
    Non-parametric parameterized expectations.  ψ(G) is estimated by a
    Nadaraya-Watson smoother of the realized regressand -- no functional form.
    The rule is iterated to a self-consistent fixed point.
    """
    G_sim = simulate_deficit(T, seed)
    log_sim = np.log(G_sim[:-1])
    # kernel weights of each grid point against the whole simulated history
    W = np.array([np.exp(-0.5 * ((np.log(g) - log_sim) / h)**2) for g in G])
    W /= W.sum(axis=1, keepdims=True)
```

(continues on next page)

(continued from previous page)

```
    s_grid = np.full(len(G), (w1 - w2) / 2)
    for _ in range(n_iter):
        s = np.clip(np.interp(np.log(G_sim), np.log(G), s_grid), G_sim + 0.05, w1 - 0.
↪05)
        y = (s[1:] - G_sim[1:]) / (w2 + (s[1:] - G_sim[1:]))
        ψ = W @ y
        s_new = np.clip(s_from_psi(ψ), G + 0.05, w1 - 0.05)
        if np.max(np.abs(s_new - s_grid)) < 1e-8:
            s_grid = s_new
            break
        s_grid = damp * s_new + (1 - damp) * s_grid
    return s_grid


s_kernel = kernel_smoother()
print(f"kernel smoother: sup error = {np.max(np.abs(s_kernel - s_benchmark)):.4f},  "
      f"ergodic RMS = {np.sqrt(ergodic @ (s_kernel - s_benchmark)**2):.4f}")
```

```
fig, ax = plt.subplots(figsize=(6.5, 4.2))
ax.plot(G, s_benchmark, 'k-', lw=2.2, label="benchmark $f(G)$")
ax.plot(G, s_kernel, 'C3o', ms=4, label="kernel smoother")
ax.set_xlabel("deficit $G$")
ax.set_ylabel("saving $s$")
ax.legend(frameon=False)
plt.show()
```

With no functional form imposed, the kernel learner recovers the benchmark to about the same accuracy as the quadratic parametric rule.

That the two do equally well here is not a coincidence to lean on.

It is because the true $f(G)$ *happens* to be almost exactly quadratic, so the parametric family we chose was very nearly correct.

When it is not, the two part ways, and the trade-off is the one familiar from econometrics:

- the **parametric** scheme learns fast and extrapolates gracefully, but is only as good as its family; a poorly chosen family delivers an approximate equilibrium and no warning;
- the **non-parametric** scheme cannot be led astray by a bad functional form, but pays for its flexibility with slower learning and with bias where the data is thin, at the edges of the deficit's range, where the kernel has few neighbours to average.

Both are recursions of the same shape.

Which one an adaptive agent should use is, once again, one of the modelling choices that the bounded-rationality program forces into the open and that rational expectations quietly made for us.

86.7 Concluding remarks

The overlapping generations model gave us a clean laboratory for a general lesson.

An adaptive agent in an environment with a large or continuous state cannot learn a separate response for every contingency.

It must **generalize**, imposing some structure that lets a finite amount of experience inform behavior across the whole state space.

Whether that structure is a low-order polynomial, a kernel bandwidth, or something else is a modelling choice with real consequences: it fixes whether the achievable limit is a rational expectations equilibrium or merely an approximation to one.

That same act of generalization is what makes learning and equilibrium computation two faces of one object.

Marcet's algorithm computes an equilibrium by parameterizing an expectation and regressing; an adaptive agent learns by parameterizing an expectation and regressing; the recursions coincide.

The next lecture, *Money as a Medium of Exchange among Artificially Intelligent Agents*, carries the generalization problem to its sharpest form.

There the state and action spaces are large enough that enumerating rules is out of the question, and agents must **discover** a compact set of good rules from scratch, using John Holland's classifier systems and genetic algorithm in place of the least squares and kernel machinery of this lecture.

86.8 Exercises

Exercise 86.8.1

The note above claimed that pushing the polynomial degree too high can *hurt* the sup-norm fit, because high-order terms chase sampling noise in the sparsely-visited tails of the deficit distribution.

Verify it.

Run the batch parameterized-expectations algorithm for degrees 0 through 5 and report two error measures against the benchmark: the sup norm over the whole grid, and the RMS error weighted by the ergodic distribution of the deficit (i.e. weighted by where agents actually operate).

Which measure is monotone in the degree, and which is not?

Interpret.

Solution

```
rows = []
for deg in range(6):
    s_hat = parameterized_expectations(deg, T=150_000)
    sup = np.max(np.abs(s_hat - s_benchmark))
    rms = np.sqrt(ergodic @ (s_hat - s_benchmark)**2)
    rows.append([deg, sup, rms])

pd.DataFrame(rows, columns=["degree", "sup error (whole grid)",
                            "ergodic RMS (where agents live)"]).set_index("degree
 ↪").round(4)
```

The ergodic-weighted error falls monotonically and then flattens: once the family is rich enough to capture the curvature of f where the deficit actually spends its time, extra terms add nothing.

The sup-norm error falls at first but then rises again.

The high-order polynomials fit the center well but oscillate at the ends of the grid, where the ergodic distribution puts almost no mass and so the regression has almost no data to discipline them.

This is the approximation-side image of the rare-state problem from *Adaptive Agents in an Overlapping Generations Monetary Economy*.

A learning scheme allocates its accuracy to where the observations are.

Misfitting a rarely-visited region shows up starkly in the sup norm but costs almost nothing in expected utility, which is why the ergodic-weighted measure is the economically relevant one.

Exercise 86.8.2

The constant family (degree 0) delivers an *approximate* equilibrium: a single saving rate, independent of the deficit, that satisfies the first-order condition only on average.

There is a natural benchmark for it.

Compute the single saving rate $\bar{s}$ that a degree-0 parameterized-expectations agent converges to, and compare it with the saving rate of the economy in which the deficit is fixed forever at its mean $\bar{G}$.

Are they the same?

Should they be?

Solution

```
# the degree-0 approximate equilibrium
s_const = parameterized_expectations(0)[0]        # constant across states

# the deterministic economy with G fixed at its mean.  Like the constant-deficit
# model of the previous lecture it has two steady states; we take the high-saving
# (low-inflation) one, found by iterating the saving map to its stable fixed point.
G_mean = ergodic @ G

def deterministic_saving(G_fixed, damp=0.5):
    s = (w1 - w2) / 2
    for _ in range(1000):
        ψ = (s - G_fixed) / (w2 + (s - G_fixed))
        s = damp * s + (1 - damp) * s_from_psi(ψ)
    return s

s_fixed = deterministic_saving(G_mean)

print(f"degree-0 approximate-equilibrium saving rate : {s_const:.4f}")
print(f"mean deficit  E[G]                           : {G_mean:.4f}")
print(f"saving in the economy with G fixed at E[G]   : {s_fixed:.4f}")
```

They are close but not equal.

The degree-0 agent picks the single saving rate that best fits the first-order condition *across the stochastic economy*, an average over the realized distribution of returns.

The fixed-deficit economy replaces that whole distribution with a single degenerate one at the mean deficit.

By a Jensen-type argument these need not coincide: the first-order condition is nonlinear in the return, so its expectation over a spread-out distribution of deficits differs from its value at the mean deficit.

They would agree only if the deficit were deterministic to begin with.

The lesson is that an approximate equilibrium is not a naive "certainty-equivalent" object.

Even the crudest one-parameter learner is responding to the *whole* distribution of outcomes it experiences, not to a point forecast.

Exercise 86.8.3

The non-parametric kernel learner has one free tuning constant: the bandwidth h.

Explore its effect.

Run the kernel learner for a range of bandwidths and report the sup and ergodic-weighted errors against the benchmark.

Explain the shape of the trade-off, and relate the two failure modes to the bias–variance decomposition.

Solution

```
rows = []
for h in (0.03, 0.05, 0.08, 0.12, 0.20):
    s_h = kernel_smoother(h=h)
    sup = np.max(np.abs(s_h - s_benchmark))
    rms = np.sqrt(ergodic @ (s_h - s_benchmark)**2)
    rows.append([h, sup, rms])

table = pd.DataFrame(rows, columns=["bandwidth $h$", "sup error",
                                    "ergodic RMS"]).set_index("bandwidth $h$")
table.round(4)
```

```
fig, ax = plt.subplots(figsize=(7, 4.2))
for h, colour in [(0.03, 'C0'), (0.08, 'C1'), (0.20, 'C2')]:
    ax.plot(G, kernel_smoother(h=h), 'o-', ms=3, lw=0.8, color=colour, label=f"$h␣
 ↪= {h}$")
ax.plot(G, s_benchmark, 'k-', lw=2, label="benchmark")
ax.set_xlabel("deficit $G$")
ax.set_ylabel("saving $s$")
ax.legend(frameon=False)
plt.show()
```

There is an interior optimum.

A **small** bandwidth averages over few neighbours, so the estimate is noisy: it wiggles around the benchmark, and the wiggle is worst at the edges where neighbours are scarce.

This is the **variance** end of the trade-off.

A **large** bandwidth averages over a wide window, smoothing the estimate but flattening the genuine curvature of $f(G)$, so the rule is pulled toward a straight line.

This is the **bias** end.

The bandwidth is the non-parametric counterpart of the polynomial degree in Exercise 86.8.1: both control how much structure the learner imposes, and both have a sweet spot that trades misfit-from-too-little-flexibility against noise-from-too-much.

The bounded-rationality program does not tell us where that spot is; it is one more choice that replaces the single discipline of rational expectations with a menu of plausible alternatives.

CHAPTER

EIGHTYSEVEN

EXCHANGE RATE INDETERMINACY, LEARNING, AND EXPERIMENTS

Contents

- *Exchange Rate Indeterminacy, Learning, and Experiments*
  - *Overview*
  - *The two-currency economy*
  - *Newton–Raphson learning*
  - *Evidence from the laboratory*
  - *A genetic algorithm economy*
  - *Concluding remarks*
  - *Exercises*

87.1 Overview

In *A Peculiar Definition of Bounded Rationality* we met a model in which rational expectations cannot pin down the exchange rate at all.

Two fiat currencies, perfect substitutes as long as their rates of return are equal, leave the exchange rate e *completely unrestricted*: if the equilibrium conditions have a solution for one e, they have a solution for every other, and the real allocation is identical across all of them.

This lecture, following Sargent [1993], expels the rational agents from that economy and puts adaptive ones in their place.

Two questions organize what follows.

First, does learning pin the exchange rate down?

An adaptive agent has beliefs, a rule for revising them, and initial conditions, and those initial conditions are exactly what the equilibrium conditions failed to determine.

So a system of adaptive agents *can* select an exchange rate where rational expectations could not.

We shall see that it does, but in a very particular way.

Newton–Raphson learners converge to a **determinate** exchange rate that depends entirely on where they started.

The "dead hand of history" does the pinning that fundamentals refused to do.

Second, does a ghost of the indeterminacy survive?

It does.

The rest points of the learning algorithm reproduce the indeterminacy exactly: the condition that stops the algorithm is the same arbitrage condition that made the exchange rate free in the first place.

Sargent calls a regime that leaves the exchange rate history-dependent "a weak reed" on which to base exchange rate determination.

Then we turn to evidence.

Arifovic [1996] ran this economy as a laboratory experiment with paying human subjects.

Their exchange rate never settled down.

And when she replaced the Newton–Raphson learners with a **genetic algorithm** — a population of binary-string agents bred by selection, crossover, and mutation — she got an economy whose exchange rate wanders persistently, with a spectrum resembling real floating exchange rates.

That contrast — a learning rule that converges versus one that generates permanent volatility — is where this lecture lands, and it is the on-ramp to *Money as a Medium of Exchange among Artificially Intelligent Agents*, where genetic algorithms and classifier systems take over entirely.

Let's start with some imports.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from typing import NamedTuple
```

87.2 The two-currency economy

We use the overlapping generations incarnation of the Kareken and Wallace [1981] indeterminacy.

At each date t, N two-period-lived agents are born, endowed with w_1 when young and w_2 when old, with $w_1 > w_2$.

There are two fiat currencies in fixed supplies H_1 and H_2.

A young agent makes two decisions: how much to save, s_t, and what fraction λ_t of that saving to hold in currency 1 (the rest going to currency 2).

The realized lifetime utility of an agent who chooses (s_t, λ_t) is

$$U(s_t, \lambda_t) = u(w_1 - s_t) + u\left(w_2 + \lambda_t s_t \frac{p_{1t}}{p_{1,t+1}} + (1-\lambda_t) s_t \frac{p_{2t}}{p_{2,t+1}}\right), \tag{87.1}$$

where p_{it} is the price level in currency i and $p_{it}/p_{i,t+1}$ is the gross return on holding currency i.

We take $u(c) = \ln c$ throughout.

Equating each currency's supply to the demand for it gives the price levels

$$p_{1t} = \frac{H_1}{\sum_i \lambda_{it} s_{it}}, \qquad p_{2t} = \frac{H_2}{\sum_i (1-\lambda_{it}) s_{it}}, \tag{87.2}$$

and the exchange rate is $e_t = p_{1t}/p_{2t}$.

87.2.1 The indeterminacy, recalled

Consider a stationary equilibrium with constant prices.

Then each currency returns $p_{it}/p_{i,t+1} = 1$, so both returns are equal, and the portfolio return in (87.1) is 1 regardless of λ.

The saving decision then solves $\max_s \ln(w_1 - s) + \ln(w_2 + s)$, giving

$$s^\star = \frac{w_1 - w_2}{2},$$

but the portfolio share λ is *completely undetermined*: utility does not depend on it when the two returns are equal.

And λ is what fixes the exchange rate.

From (87.2), with a common (s, λ),

$$e = \frac{p_1}{p_2} = \frac{H_1}{H_2} \cdot \frac{1-\lambda}{\lambda}.$$

Every $\lambda \in (0, 1)$ is an equilibrium, so every $e \in (0, \infty)$ is an equilibrium.

This is the indeterminacy of *A Peculiar Definition of Bounded Rationality*, now expressed through an undetermined portfolio choice.

This lecture runs two economies with different parameters — Sargent's and Arifovic's — so we carry the parameters in a container and pass them explicitly rather than leaving them lying about as globals.

```
class Params(NamedTuple):
    w1: float           # endowment when young
    w2: float           # endowment when old
    H1: float           # fixed supply of currency 1
    H2: float           # fixed supply of currency 2

    @property
    def s_star(self):
        "Rational expectations saving rate, when both currencies return 1."
        return (self.w1 - self.w2) / 2


kw = Params(w1=20.0, w2=15.0, H1=100.0, H2=120.0)

print(f"rational expectations saving rate s* = {kw.s_star}")
print(f"any portfolio share λ is an equilibrium, and e = (H1/H2)(1-λ)/λ is free")
```

```
rational expectations saving rate s* = 2.5
any portfolio share λ is an equilibrium, and e = (H1/H2)(1-λ)/λ is free
```

87.3 Newton–Raphson learning

Now expel the rational agents.

Following Sargent [1993], we split the population into two classes — call them "even" and "odd" — because a cohort's lifetime utility cannot be evaluated until it is old.

Each class carries its own rule (s, λ), updated only from the experience of earlier agents of the same class.

An agent revises (s, λ) by a **Newton–Raphson** step against realized utility: move in the direction that a second-order expansion of (87.1) says would have raised utility, given the returns the agent actually experienced.

Writing g for the gradient and R for a running estimate of the (negative-definite) Hessian, the recursion is

$$\begin{aligned} R_{\tau+1} &= R_\tau + \gamma_\tau (H_\tau - R_\tau), \\ \begin{bmatrix} s \\ \lambda \end{bmatrix}_{\tau+1} &= \begin{bmatrix} s \\ \lambda \end{bmatrix}_\tau - \gamma_\tau R_{\tau+1}^{-1} \, g_\tau, \end{aligned} \tag{87.3}$$

with H_τ the realized Hessian and γ_τ a gain.

Here is the gradient and Hessian of (87.1) in the returns $R_1 = p_{1t}/p_{1,t+1}$ and $R_2 = p_{2t}/p_{2,t+1}$.

```
def grad_hess(p, s, lam, R1, R2):
    "Gradient and Hessian of realized utility U(s, λ)."
    A = lam*R1 + (1 - lam)*R2               # portfolio gross return
    c2 = p.w2 + s*A                         # old-age consumption
    dR = R1 - R2
    g = np.array([-1/(p.w1 - s) + A/c2,
                  s*dR/c2])
    H_ss = -1/(p.w1 - s)**2 - A**2/c2**2
    H_ll = -s**2 * dR**2 / c2**2
    H_sl = dR/c2 - s*A*dR/c2**2
    return g, np.array([[H_ss, H_sl], [H_sl, H_ll]])
```

87.3.1 The indeterminacy shows up as a singular Hessian

Look at the λ block of the Hessian.

When the two returns are equal, $R_1 = R_2$, every λ-derivative vanishes: the gradient component $s(R_1 - R_2)/c_2$ is zero, and so is the curvature $-s^2(R_1 - R_2)^2/c_2^2$.

Utility is **flat in** λ at equal returns.

That is the indeterminacy, seen locally: there is no force pushing λ in any direction, so a Newton step — which divides the gradient by the curvature — is $0/0$ in the λ direction.

For the algorithm to move at all we must keep R invertible.

We do that with a small **prior curvature** κ in the λ direction.

Economically this is exactly the sluggishness that Sargent [1993] invokes: it is the "dead hand of history" that lets an otherwise-free exchange rate settle down.

```
def newton_learning(p, s0, lam0, T=400, gain=0.3, κ=0.5):
    """
    Two-currency OLG with Newton-Raphson learning of (s, λ), one rule per class
    (even/odd).  κ is a prior curvature in the indeterminate λ direction that
    keeps the second-moment matrix R invertible.
    """
    s = np.array(s0, float)
    lam = np.array(lam0, float)
    R = [np.array([[-1.0, 0.0], [0.0, -κ]]) for _ in range(2)]   # prior curvature
    p1 = np.empty(T)
    p2 = np.empty(T)
    e = np.empty(T)
    s_hist = np.empty((T, 2))
    lam_hist = np.empty((T, 2))

    for t in range(T):
        j = t % 2                                       # class that is young at t
```

(continues on next page)

(continued from previous page)

```
        p1[t] = p.H1 / (lam[j]*s[j])
        p2[t] = p.H2 / ((1 - lam[j])*s[j])
        e[t] = p1[t] / p2[t]
        if t >= 1:                                    # class young at t-1 is now old
            jp = 1 - j
            R1, R2 = p1[t-1]/p1[t], p2[t-1]/p2[t]
            g, H = grad_hess(p, s[jp], lam[jp], R1, R2)
            H[1, 1] -= κ                              # retain prior λ-curvature
            R[jp] = R[jp] + gain*(H - R[jp])
            step = gain * np.linalg.solve(R[jp], g)
            s[jp] = np.clip(s[jp] - step[0], 0.1, p.w1 - 0.1)
            lam[jp] = np.clip(lam[jp] - step[1], 0.02, 0.98)
        s_hist[t] = s
        lam_hist[t] = lam

    return e, s_hist, lam_hist
```

87.3.2 Convergence to a history-dependent exchange rate

Run two experiments that are identical except for the initial portfolio shares.

```
e1, s1, l1 = newton_learning(kw, [3.5, 2.0], [0.35, 0.40])
e2, s2, l2 = newton_learning(kw, [2.0, 3.5], [0.62, 0.58])

fig, axes = plt.subplots(1, 3, figsize=(14, 4))

axes[0].plot(np.log(e1), 'C0', lw=1.3, label="experiment 1")
axes[0].plot(np.log(e2), 'C1', lw=1.3, label="experiment 2")
axes[0].set_title("log exchange rate")
axes[0].set_xlabel("$t$")
axes[0].legend(frameon=False)

for s_h, c in [(s1, 'C0'), (s2, 'C1')]:
    axes[1].plot(s_h[:, 0], color=c, lw=1.0)
    axes[1].plot(s_h[:, 1], color=c, lw=1.0, ls=':')
axes[1].axhline(kw.s_star, color='k', lw=0.8, ls='--')
axes[1].set_title("saving  (solid = even, dotted = odd)")
axes[1].set_xlabel("$t$")

for l_h, c in [(l1, 'C0'), (l2, 'C1')]:
    axes[2].plot(l_h[:, 0], color=c, lw=1.0)
    axes[2].plot(l_h[:, 1], color=c, lw=1.0, ls=':')
axes[2].set_title(r"portfolio share $\lambda$")
axes[2].set_xlabel("$t$")
plt.tight_layout()
plt.show()
```

Both economies converge.

Saving climbs or falls to the rational expectations rate $s^\star = 2.5$ from either side, and the two classes' portfolio shares merge to a common value.

But the two experiments converge to *different exchange rates*.

The only thing that differed between them was where the portfolios started.

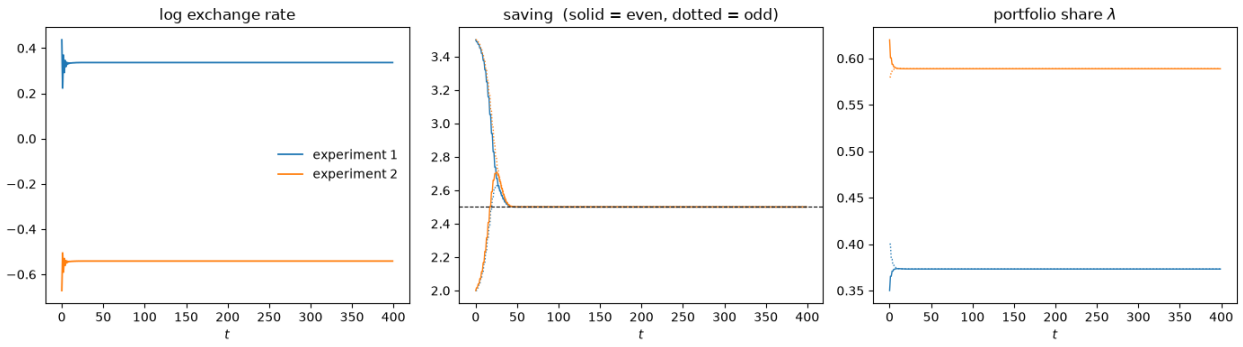


Fig. 87.1: Newton-Raphson learning from two initial conditions

```
for name, e, l in [("experiment 1", e1, l1), ("experiment 2", e2, l2)]:
    print(f"{name}: e → {e[-1]:.4f},  s → {s1[-1][0]:.3f},  λ → {l[-1][0]:.3f}")
```

```
experiment 1: e → 1.3998,  s → 2.500,  λ → 0.373
experiment 2: e → 0.5819,  s → 2.500,  λ → 0.589
```

The saving rate is pinned down by fundamentals; the exchange rate is pinned down by history.

87.3.3 The dead hand of history

To see how completely history governs the outcome, sweep the initial portfolio share and record the exchange rate each economy settles on.

```
λ0_grid = np.linspace(0.15, 0.85, 15)
e_limits = [newton_learning(kw, [4.0, 4.0], [λ0, λ0])[0][-1] for λ0 in λ0_grid]

fig, ax = plt.subplots(figsize=(7, 4.5))
ax.plot(λ0_grid, e_limits, 'C0o-', ms=5, label="limiting $e$ from learning")
ax.plot(λ0_grid, (kw.H1/kw.H2)*(1 - λ0_grid)/λ0_grid, 'k--', lw=1,
        label=r"$(H_1/H_2)(1-\lambda)/\lambda$")
ax.set_xlabel(r"initial portfolio share $\lambda_0$")
ax.set_ylabel("limiting exchange rate $e$")
ax.legend(frameon=False)
plt.show()
```

The limiting exchange rate traces out the *entire rational expectations continuum*.

Every point on the curve is a valid rational expectations equilibrium; learning selects among them purely by initial condition.

This is the sense in which learning renders the exchange rate determinate.

It does not add a fundamental that fundamentals were missing.

It converts an *indeterminacy of levels* into a *dependence on history*: the exchange rate the economy reaches is whatever the initial portfolio beliefs implied, frozen in place.

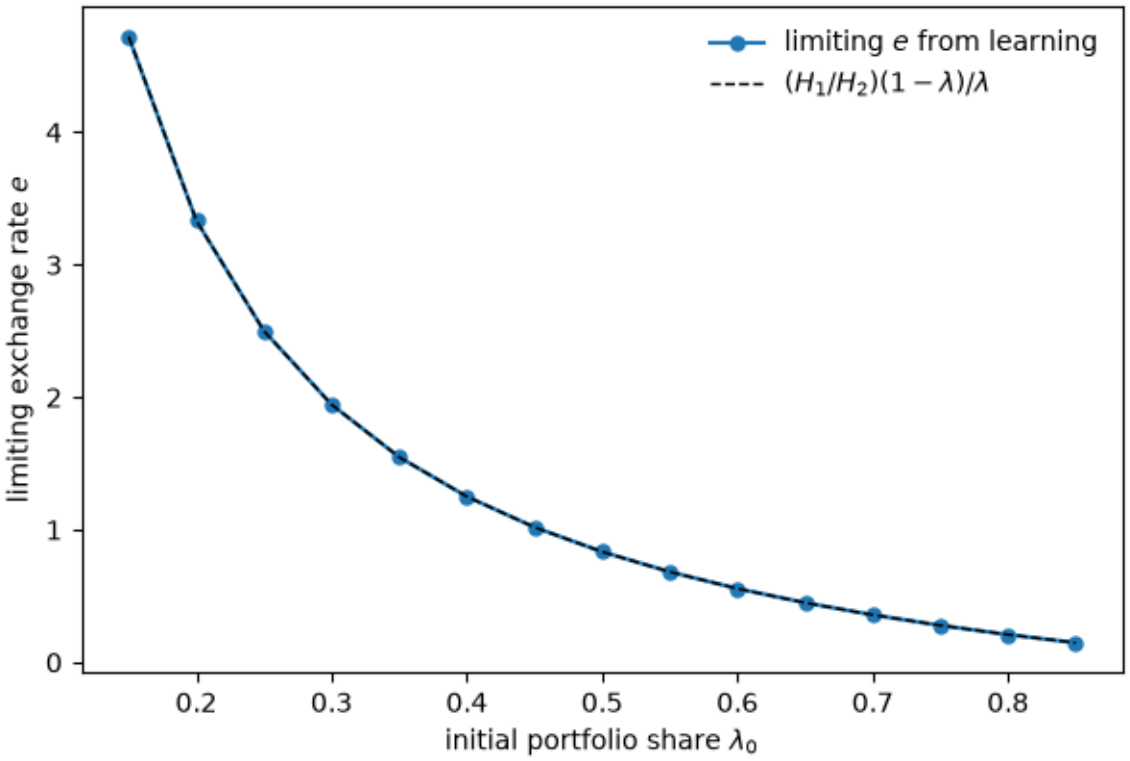


Fig. 87.2: Limiting exchange rate against initial portfolio share

87.3.4 The ghost of indeterminacy

Why does the exchange rate freeze rather than move to some particular value?

Because the algorithm comes to rest wherever the gradient vanishes, and the portfolio gradient $s(R_1 - R_2)/c_2$ vanishes precisely when $R_1 = R_2$, the **arbitrage condition** that made the exchange rate free under rational expectations.

```
# at the limit the classes have merged, so prices are constant and R1 = R2 = 1
s_lim, lam_lim = s1[-1], l1[-1]
p1_lim = kw.H1 / (lam_lim*s_lim)
p2_lim = kw.H2 / ((1 - lam_lim)*s_lim)
print(f"classes converged to a common rule:  s = {s_lim.round(4)},  λ = {lam_lim.
 ↪round(4)}")
print(f"→ prices constant across periods, so R1 = R2 = 1 (arbitrage holds at the rest␣
 ↪point)")
print(f"→ the λ-gradient is zero for *any* λ, so learning cannot move the exchange␣
 ↪rate off "
      f"wherever history left it")
```

```
classes converged to a common rule:  s = [2.5 2.5],  λ = [0.3732 0.3732]
→ prices constant across periods, so R1 = R2 = 1 (arbitrage holds at the rest␣
 ↪point)
→ the λ-gradient is zero for *any* λ, so learning cannot move the exchange rate␣
 ↪off wherever history left it
```

The rest points of the learning dynamics *are* the rational expectations equilibria, exchange rate and all.

The indeterminacy is not resolved; it is displaced into the initial conditions.

Sargent is blunt about how much weight this can bear:

> Put differently, a regime that allows the exchange rate to be history-dependent seems to be an ill-formed mechanism.

An exchange rate determined by nothing but the accident of initial beliefs is a weak reed.

If the economy is a shade different — if the learning rule keeps experimenting rather than settling — the whole construction can come apart.

That is exactly what the next two exhibits show.

87.4 Evidence from the laboratory

Arifovic [1996] implemented this two-currency economy as an experiment with paying human subjects, using $w_1 = 11$, $w_2 = 1$, $H_1 = H_2 = 10$.

Each young subject chose a saving rate and a fraction to allocate between the two currencies; the experimenters cleared the two money markets against those choices, exactly as in (87.2).

The result did not look like the Newton–Raphson simulations at all.

The exchange rate fluctuated persistently, roughly within a band from 0.5 to 2, and showed no sign of settling down.

If anything the amplitude grew across sessions.

The simple Newton–Raphson model, which always converges to a constant, does a poor job of matching that.

So Arifovic built a different model of the same economy, one in which the agents are a **population** bred by a genetic algorithm.

87.5 A genetic algorithm economy

Now each class is a population of $N = 30$ agents, and each agent is a **binary string** of length 30: the first 20 bits encode its saving rate, the last 10 its portfolio share.

Every period the young population's strings are decoded into (s_i, λ_i) pairs, the two money markets clear against the aggregates via (87.2), and the exchange rate is read off.

A generation later, when the cohort is old, each string's realized utility (87.1) is its **fitness**, and the genetic algorithm breeds the next generation.

```
N, L_s, L_lam = 30, 20, 10          # population size; bits for saving and portfolio

arifovic = Params(w1=11.0, w2=1.0, H1=10.0, H2=10.0)

def decode(p, pop):
    "Binary strings → (s, λ), with s in (0, w1) and λ in (0, 1)."
    ints_s = pop[:, :L_s] @ (1 << np.arange(L_s)[::-1])
    ints_l = pop[:, L_s:] @ (1 << np.arange(L_lam)[::-1])
    s = 0.05 + (p.w1 - 0.10) * ints_s / (2**L_s - 1)
    lam = 0.02 + 0.96 * ints_l / (2**L_lam - 1)
    return s, lam

def market_prices(p, pop):
    s, lam = decode(p, pop)
    return p.H1 / np.sum(lam*s), p.H2 / np.sum((1 - lam)*s)
```

(continues on next page)

(continued from previous page)

```
def fitness(p, pop, R1, R2):
    "Realized lifetime utility of each string given the returns it experienced."
    s, lam = decode(p, pop)
    c2 = p.w2 + s*(lam*R1 + (1 - lam)*R2)
    return np.log(p.w1 - s) + np.log(np.maximum(c2, 1e-9))
```

The genetic algorithm has the classic three operators — fitness-proportional **selection** of parents, single-point **crossover**, and bit-flip **mutation** — plus one that Arifovic introduced, the **election operator**: a child is admitted to the next generation only if it would have been fitter than its parent on the most recent returns, otherwise the parent survives.

```
def genetic_step(p, pop, R1, R2, rng, p_mut=0.033, election=True):
    "Breed one new generation from the current one."
    fit = fitness(p, pop, R1, R2)
    weight = fit - fit.min() + 1e-6                    # shift to positive for roulette␣
 ↪wheel
    new = np.empty_like(pop)
    for k in range(0, N, 2):
        i, j = np.searchsorted(np.cumsum(weight), rng.random(2) * weight.sum())
        parents = np.array([pop[i], pop[j]])
        cut = rng.integers(1, L_s + L_lam)             # single-point crossover
        kids = parents.copy()
        kids[0, cut:], kids[1, cut:] = parents[1, cut:], parents[0, cut:]
        for c in kids:                                 # mutation
            c[rng.random(L_s + L_lam) < p_mut] ^= 1
        if election:                                   # Arifovic's election operator
            f_kids = fitness(p, kids, R1, R2)
            f_par = fitness(p, parents, R1, R2)
            for m in range(2):
                new[k + m] = kids[m] if f_kids[m] > f_par[m] else parents[m]
        else:
            new[k], new[k + 1] = kids
    return new


def genetic_economy(p, T=3000, seed=0, election=True):
    rng = np.random.default_rng(seed)
    pops = [rng.integers(0, 2, (N, L_s + L_lam)) for _ in range(2)]   # even, odd
    p1 = np.empty(T)
    p2 = np.empty(T)
    e = np.empty(T)
    for t in range(T):
        j = t % 2
        p1[t], p2[t] = market_prices(p, pops[j])
        e[t] = p1[t] / p2[t]
        if t >= 1:
            jp = 1 - j
            R1, R2 = p1[t-1]/p1[t], p2[t-1]/p2[t]
            pops[jp] = genetic_step(p, pops[jp], R1, R2, rng, election=election)
    return e
```

87.5.1 Volatility that never dies

```
e_ga = genetic_economy(arifovic, T=4000, seed=1)
log_e = np.log(e_ga[500:])                              # drop a burn-in

fig, ax = plt.subplots(figsize=(9, 4))
ax.plot(log_e, lw=0.5)
ax.set_xlabel("$t$")
ax.set_ylabel("$\\log e_t$")
plt.show()
```

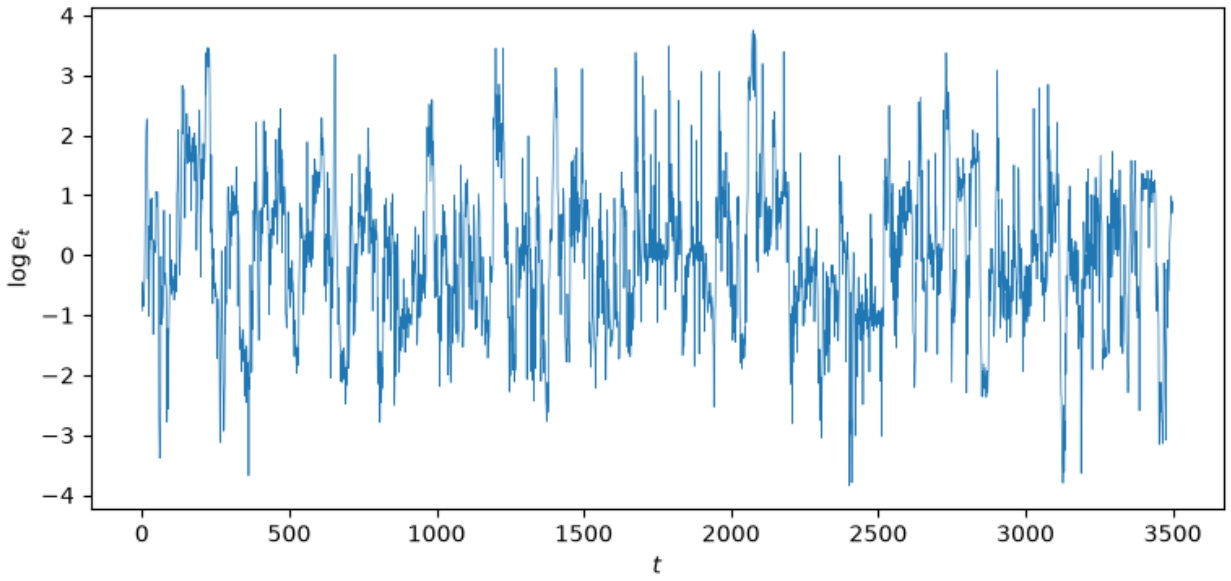


Fig. 87.3: Exchange rate in the genetic-algorithm economy

The exchange rate wanders and keeps wandering.

Unlike the Newton–Raphson learner, the genetic population never settles: mutation keeps injecting new strings, and the market keeps repricing them.

The volatility is a permanent feature, not a transient.

```
early = np.log(e_ga[500:1500]).std()
late = np.log(e_ga[3000:]).std()
print(f"standard deviation of log e, early window: {early:.3f}")
print(f"standard deviation of log e, late window:  {late:.3f}")
print("→ the volatility does not damp out over time")
```

```
standard deviation of log e, early window: 1.279
standard deviation of log e, late window:  1.194
→ the volatility does not damp out over time
```

87.5.2 The shape of the volatility

Arifovic reported that the exchange rate in her genetic economy behaves almost like a random walk — but with **mean reversion**, showing up as a dip in the spectrum of its first difference at zero frequency.

```
def averaged_spectrum(x, n_seg=16):
    "Bartlett-averaged periodogram, for a readable spectral estimate."
    x = x - x.mean()
    seg = len(x) // n_seg
    acc = np.zeros(seg // 2 + 1)
    for k in range(n_seg):
        acc += np.abs(np.fft.rfft(x[k*seg:(k+1)*seg]))**2 / seg
    return np.fft.rfftfreq(seg), acc / n_seg


d_log_e = np.diff(np.log(e_ga[500:]))
d_log_e = np.clip(d_log_e, *np.percentile(d_log_e, [1, 99]))    # winsorize tails
freq, spec = averaged_spectrum(d_log_e)

def acf(x, K):
    x = x - x.mean()
    return np.array([np.sum(x[k:]*x[:len(x)-k]) / np.sum(x*x) for k in range(K)])


fig, axes = plt.subplots(1, 2, figsize=(12, 4))
axes[0].plot(freq[1:], spec[1:], lw=1.2)
axes[0].axhline(spec[1:].mean(), color='k', ls=':', lw=0.8)
axes[0].set_title(r"spectrum of $\Delta \log e$")
axes[0].set_xlabel("frequency")

axes[1].bar(range(25), acf(np.log(e_ga[500:]), 25))
axes[1].set_title(r"autocorrelation of $\log e$")
axes[1].set_xlabel("lag")
plt.tight_layout()
plt.show()
```

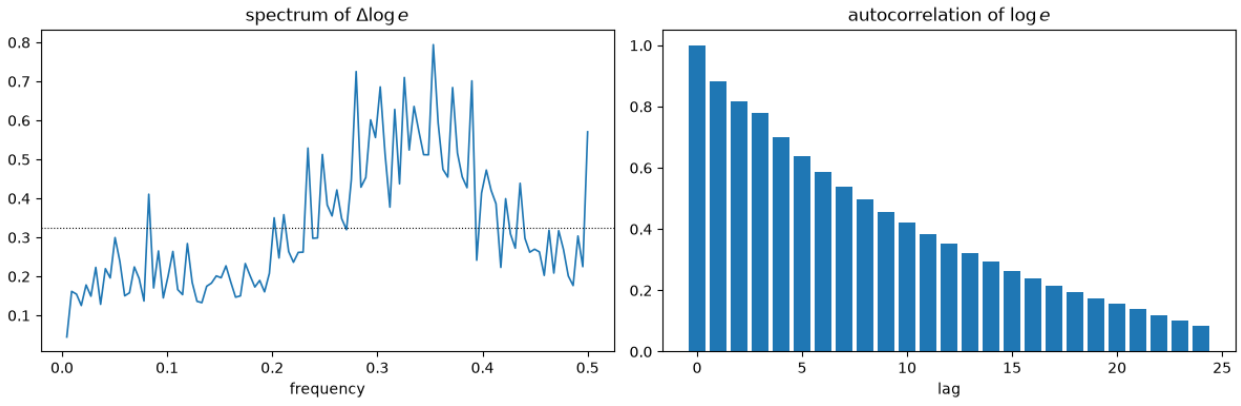


Fig. 87.4: Spectrum and autocorrelation of the exchange rate

The spectrum of the first difference is low near zero frequency and rises toward the higher frequencies, the signature of a series whose *level* is close to a random walk, but with enough mean reversion to pull the low-frequency power down.

The autocorrelation of the level decays slowly, as a near-unit-root series would, but it *does* decay, which a pure random walk's would not.

```
band = slice(1, len(spec)//2)
print(f"spectral power of Δlog e near zero frequency: {spec[1]:.4f}")
print(f"average spectral power over low-to-mid band:  {spec[band].mean():.4f}")
print(f"→ pronounced dip at zero frequency (mean reversion): {spec[1] < spec[band].
↪mean()}")
```

```
spectral power of Δlog e near zero frequency: 0.0446
average spectral power over low-to-mid band:  0.2174
→ pronounced dip at zero frequency (mean reversion): True
```

The book's assessment: real floating exchange rates for hard-currency pairs have spectra of log differences that look much like this, except *without* the dip at zero frequency; actual exchange rates are even closer to pure random walks.

Arifovic's genetic economy, with no fundamental shocks at all, manufactures most of the way there out of nothing but a population of adapting agents repricing two intrinsically identical currencies.

87.6 Concluding remarks

Two models of the same indeterminate economy gave two very different verdicts.

The Newton–Raphson learner **converges** — and in converging, exposes the indeterminacy rather than curing it.

Its rest point is an arbitrage condition that leaves the exchange rate free, so the limiting exchange rate is whatever history's initial portfolio beliefs implied.

Determinacy by dead hand, which Sargent judges a weak reed.

The genetic-algorithm economy **does not converge**.

A whole population of adapting strings, constantly refreshed by mutation and repriced by the market, generates exchange rate volatility that never dies and that mimics the low-frequency behavior of real floating rates — from an economy with no fundamental disturbances whatsoever.

The gap between them is not about the economics, which is identical, but about the *adaptive machinery*.

A device that stabilizes a single agent's learning — the election operator, which keeps only improving offspring — turns out, inside a self-referential market, to sustain volatility rather than damp it (Exercise 87.7.2).

Which learning model an economist reaches for is, once again, one of the choices that the bounded-rationality program forces into the open.

That machinery — Holland's genetic algorithm, and its richer cousin the classifier system — is the subject of *Money as a Medium of Exchange among Artificially Intelligent Agents*, where a population of adaptive agents must not merely tune a portfolio but *discover from scratch* which good will serve as money.

87.7 Exercises

Exercise 87.7.1

The Newton–Raphson economy converges to a rest point where the two classes share a common rule and the arbitrage condition $R_1 = R_2$ holds.

Verify the claim that the limiting exchange rate depends on the *whole* initial condition, not just the initial λ, by starting the two classes **asymmetrically**.

Run the learner from several initial conditions in which the even and odd classes begin with different portfolio shares, and confirm that (a) the classes' shares merge to a common value, (b) saving converges to $s^\star$, and © the limiting exchange rate depends on the starting point.

Does the common limiting λ equal the average of the two initial shares?

Solution

```
rows = []
starts = [([3.0, 2.2], [0.30, 0.50]),
          ([2.2, 3.0], [0.50, 0.30]),
          ([3.5, 2.0], [0.35, 0.55]),
          ([2.5, 2.5], [0.40, 0.60])]
for s0, lam0 in starts:
    e, s_h, l_h = newton_learning(kw, s0, lam0, T=600)
    rows.append([str(lam0), l_h[-1][0], l_h[-1][1], s_h[-1].mean(),
                 e[-1], np.mean(lam0)])

pd.DataFrame(rows, columns=["initial λ", "λ even (limit)", "λ odd (limit)",
                            "s (limit)", "e (limit)", "mean of initial λ"]).
↪round(4)
```

	initial λ	λ even (limit)	λ odd (limit)	s (limit)	e (limit)	\
0	[0.3, 0.5]	0.4117	0.4117	2.5	1.1908	
1	[0.5, 0.3]	0.3666	0.3666	2.5	1.4397	
2	[0.35, 0.55]	0.4433	0.4433	2.5	1.0463	
3	[0.4, 0.6]	0.5208	0.5208	2.5	0.7668	

	mean of initial λ
0	0.40
1	0.40
2	0.45
3	0.50

The two classes' shares always merge (columns 2 and 3 agree), saving always converges to $s^\star = 2.5$, and the limiting exchange rate differs across starting points — so it is genuinely history-dependent.

But the common limiting λ is **not** simply the average of the two initial shares: compare the merged value with the last column.

The transient path matters, because while the classes still differ the returns differ, and those transient return gaps push λ around before the arbitrage condition shuts the motion off.

The exchange rate records the whole history of the adjustment, not just its starting average.

Exercise 87.7.2

Arifovic's **election operator** admits a child to the next generation only if it beats its parent on the latest returns.

In single-agent optimization such a filter can only help — it never lets a worse rule replace a better one.

But this economy is *self-referential*: the returns against which fitness is measured are themselves determined by the population's choices.

Investigate what the election operator does to exchange rate volatility.

Run the genetic economy with and without it across several seeds, and report the volatility of the log exchange rate.

Which way does the operator push volatility, and can you explain why?

Solution

```
rows = []
for election in (True, False):
    sds = [np.log(genetic_economy(arifovic, T=2500, seed=sd,
                                  election=election)[500:]).std()
           for sd in range(6)]
    rows.append([election, np.mean(sds), np.min(sds), np.max(sds)])

pd.DataFrame(rows, columns=["election operator", "mean s.d. of log e",
                            "min", "max"]).round(3)
```

```
   election operator  mean s.d. of log e    min    max
0               True               1.220  1.081  1.350
1              False               0.522  0.473  0.553
```

The election operator **raises** volatility — the opposite of its stabilizing role in single-agent search.

The mechanism is the self-reference.

With the filter on, the population keeps only offspring that beat their parents *on last period's returns*, so it chases the recently profitable portfolio.

But "recently profitable" depends on the exchange rate, which the population's own chasing moves — so the whole population piles into the recent winner, overshoots, and the exchange rate lurches to make a different portfolio profitable, whereupon the population chases that.

Without the filter, mutation keeps the population diffuse, and diverse portfolios partly cancel in the aggregates (87.2), damping the swings.

A device that unambiguously improves an isolated learner can destabilize a market of learners.

It is a compact warning about reading single-agent intuitions into multi-agent systems — a theme that returns in *Money as a Medium of Exchange among Artificially Intelligent Agents*.

Exercise 87.7.3

The lecture claimed the genetic economy's exchange rate is "close to a random walk, but with mean reversion."

Make the claim quantitative.

Treating $\log e_t$ as data, estimate the first-order autoregressive coefficient in $\log e_t = \mu + \phi \log e_{t-1} + \varepsilon_t$, and compare it with 1 (a pure random walk).

Do this for several seeds.

Is ϕ close to but below 1, consistent with a near-unit-root, mean-reverting series?

Solution

```
def ar1_coefficient(x):
    "OLS estimate of ψ in x_t = μ + ψ x_{t-1} + ε."
    x0, x1 = x[:-1], x[1:]
    X = np.column_stack([np.ones_like(x0), x0])
    μ, ψ = np.linalg.lstsq(X, x1, rcond=None)[0]
    return ψ

rows = []
for sd in range(6):
    le = np.log(genetic_economy(arifovic, T=3000, seed=sd)[500:])
    rows.append([sd, ar1_coefficient(le)])

table = pd.DataFrame(rows, columns=["seed", "AR(1) coefficient ψ"])
print(table.round(4).to_string(index=False))
print(f"\nmean ψ across seeds: {table['AR(1) coefficient ψ'].mean():.4f}")
```

```
 seed  AR(1) coefficient ψ
    0               0.9165
    1               0.8849
    2               0.8974
    3               0.8739
    4               0.8726
    5               0.8979

mean ψ across seeds: 0.8905
```

The estimated ϕ is consistently close to 1 but strictly below it — a highly persistent series that nonetheless reverts to its mean rather than drifting forever.

That is precisely the "near random walk with mean reversion" the spectrum showed: a pure random walk would have $\phi = 1$ exactly (and no dip at zero frequency), while a value a little under 1 produces the slow autocorrelation decay and the low-frequency dip we saw.

The genetic economy lands in that near-unit-root region on its own, with no fundamental shocks driving it — the persistence is manufactured entirely by the population dynamics repricing two identical currencies.

CHAPTER

EIGHTYEIGHT

GENETIC ALGORITHMS AND CLASSIFIER SYSTEMS

Contents

- *Genetic Algorithms and Classifier Systems*
  - *Overview*
  - *The perceptron: an agent as a discriminant function*
  - *Associative memory: the Hopfield network*
  - *The genetic algorithm*
  - *Classifier systems*
  - *Evolutionary programming*
  - *Concluding remarks*
  - *Exercises*

88.1 Overview

The lectures so far gave adaptive agents a fairly econometric brain.

In *Adaptive Agents in an Overlapping Generations Monetary Economy* and *Learning, Approximation, and Equilibrium Computation* they ran recursive least squares; in *Exchange Rate Indeterminacy, Learning, and Experiments* they took Newton steps against realized utility.

In each case an agent held a *parametric* rule and tuned its coefficients.

This lecture surveys a different toolbox, the one Sargent [1993] draws from John Holland's work on artificial intelligence and from the connectionist literature on neural networks.

These devices do not tune the coefficients of a fixed rule.

They **discover the rule itself**, out of a large space of possibilities, from experience.

Sargent's framing is that the bounded-rationality program asks us to populate models with agents who "behave like econometricians," and that the artificial-intelligence literature is a catalogue of candidate brains for them:

> It is from the store of methods built up in these literatures that we shall select the 'brains' to give our boundedly rational agents.

We tour four of them:

1. the **perceptron**, the simplest neural network, which turns out to be a linear discriminant function, a direct link to the "agents as econometricians" reading;
2. the **Hopfield network**, an associative memory that recovers stored patterns from corrupted inputs by rolling downhill on an energy function;
3. the **genetic algorithm**, Holland's population-based search, which we set loose on Axelrod's iterated prisoner's dilemma and watch discover cooperation;
4. the **classifier system**, Holland's "brain as a competitive economy" of if-then rules, whose simplest instance — a two-armed bandit — already exposes a subtle limitation.

We close with **evolutionary programming**: the idea that a population of adaptive agents, plodding toward an equilibrium, can be used to *compute* equilibria that we cannot solve for directly.

That is the idea *Money as a Medium of Exchange among Artificially Intelligent Agents* carries out in full, so this lecture is the machinery on which the next one is built.

Let's start with some imports.

```
import numpy as np
import matplotlib.pyplot as plt
```

88.2 The perceptron: an agent as a discriminant function

The simplest neural network is a single **perceptron**: k inputs x_i, weights w_i, and one output

$$y = S\left(\sum_{i=1}^{k} w_i x_i\right) = S(w^\top x),$$

where S is a "squasher" mapping $\mathbb{R}$ onto $[0,1]$: a step function, or the sigmoid $S(z) = 1/(1+e^{-z})$, or any cumulative distribution function.

For fixed weights the perceptron is a **classifier**: it fires ($y=1$) when $w^\top x > 0$ and stays quiet otherwise, so the boundary $w^\top x = 0$ is a hyperplane separating two classes.

Training the perceptron — choosing w to minimize $\sum_t (y_t - S(w^\top x_t))^2$ — is a nonlinear least squares problem, solved by the familiar stochastic-gradient recursion

$$w_t = w_{t-1} + \gamma_t \nabla S(w_{t-1}, x_t)\,(y_t - S(w_{t-1}^\top x_t)),$$

which is the same $1/t$-flavored update that ran the learning models of the previous lectures.

Take the book's example: separate football players from economists on two standardized features.

```
rng = np.random.default_rng(0)
n = 100
economists = rng.multivariate_normal([-1.0, -0.3], [[0.5, 0.1], [0.1, 0.5]], n)
players    = rng.multivariate_normal([1.2,  0.8], [[0.6, 0.0], [0.0, 0.6]], n)
X = np.vstack([economists, players])
y = np.r_[np.zeros(n), np.ones(n)]
X_aug = np.column_stack([np.ones(len(X)), X])          # prepend an intercept

def sigmoid(z):
    return 1 / (1 + np.exp(-z))

w = np.zeros(3)
```

(continues on next page)

(continued from previous page)

```
for _ in range(200):                                    # train by stochastic gradient
    for t in rng.permutation(len(X)):
        w += 0.1 * (y[t] - sigmoid(X_aug[t] @ w)) * X_aug[t]

print(f"training accuracy: {np.mean((sigmoid(X_aug @ w) > 0.5) == y):.3f}")
```

```
training accuracy: 0.930
```

The point Sargent [1993] stresses is that this is nothing exotic to an econometrician: the perceptron's decision boundary is essentially a **linear discriminant function**.

Compare the perceptron's boundary normal to Fisher's linear discriminant direction.

```
m0, m1 = X[y == 0].mean(0), X[y == 1].mean(0)
S_within = np.cov(X[y == 0].T) * n + np.cov(X[y == 1].T) * n
lda_direction = np.linalg.solve(S_within, m1 - m0)
lda_direction /= np.linalg.norm(lda_direction)
perceptron_direction = w[1:] / np.linalg.norm(w[1:])

print(f"perceptron boundary normal : {perceptron_direction.round(3)}")
print(f"Fisher discriminant direction: {lda_direction.round(3)}")
print(f"cosine similarity: {abs(perceptron_direction @ lda_direction):.4f}")
```

```
perceptron boundary normal : [0.946 0.324]
Fisher discriminant direction: [0.925 0.38 ]
cosine similarity: 0.9982
```

```
fig, ax = plt.subplots(figsize=(6.5, 5))
ax.scatter(*economists.T, s=18, c='C0', label="economists ($y=0$)")
ax.scatter(*players.T, s=18, c='C1', label="football players ($y=1$)")
xs = np.linspace(X[:, 0].min(), X[:, 0].max(), 2)
ax.plot(xs, -(w[0] + w[1]*xs) / w[2], 'k-', lw=1.5, label="perceptron boundary")
ax.set_xlabel("weight (standardized)")
ax.set_ylabel("salary (standardized)")
ax.legend(frameon=False)
plt.show()
```

The two directions are all but identical.

Minsky and Papert's famous critique [Minsky and Papert, 1969] was precisely that a single perceptron can represent *only* a linear discriminant, and so cannot separate classes that a line cannot.

The revival of the field came with the recognition that **layering** perceptrons — feeding their outputs into further perceptrons — approximates any nonlinear discriminant, the subject of *Introduction to Artificial Neural Networks*.

For our purposes the perceptron is the entry in the catalogue closest to what the previous lectures already did: a parametric rule, trained by gradient descent, that an econometrician would recognize on sight.

The remaining three brains are stranger.

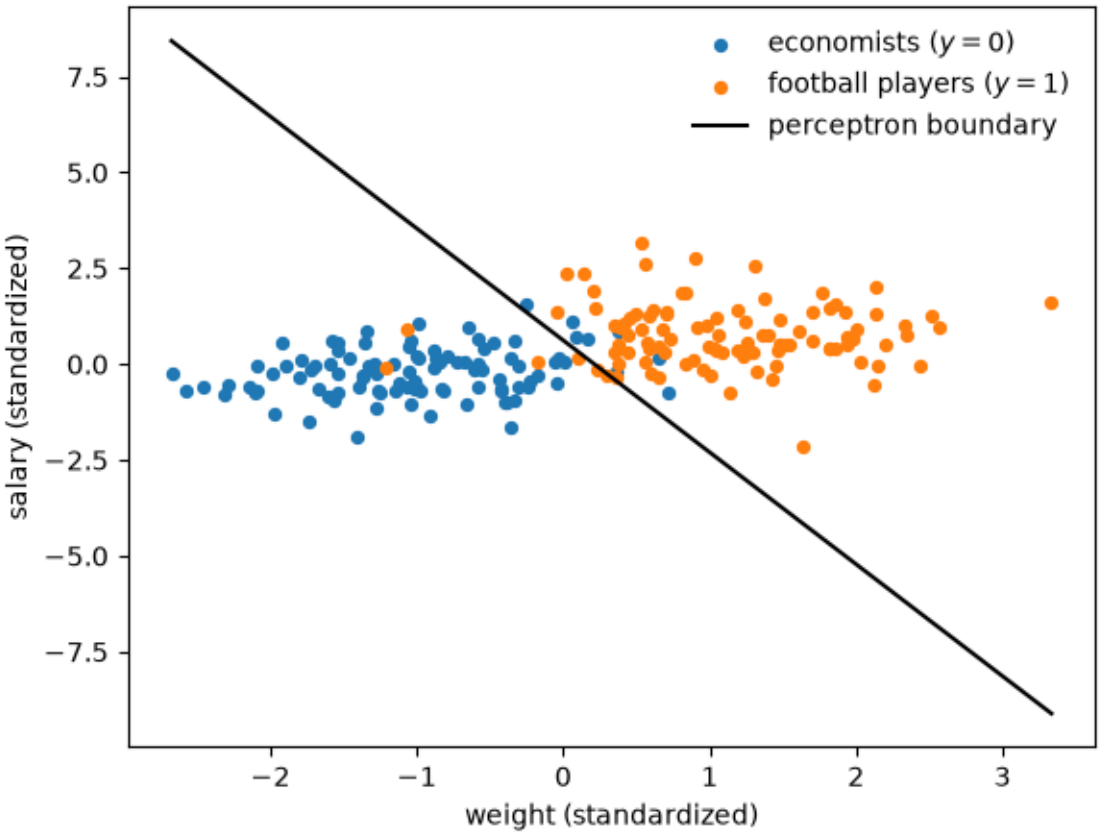


Fig. 88.1: Perceptron boundary separating the two groups

88.3 Associative memory: the Hopfield network

The second device stores patterns and recalls them from corrupted fragments.

Represent a pattern as a vector of ± 1 of length N.

We want to store p patterns $\sigma^1, \dots, \sigma^p$ so that each is a fixed point of a dynamical system, and so that feeding in a corrupted version converges quickly to the nearest stored pattern.

The **Hopfield network** does this with the dynamics

$$s(t+1) = \operatorname{sgn}(w\, s(t)),$$

and a weight matrix built from the patterns themselves.

When the patterns are orthogonal, Hebb's rule $w = \frac{1}{N}\sigma\sigma^\top$ suffices; for merely linearly independent (correlated) patterns one uses the projection rule $w = \frac{1}{N}\sigma V^{-1}\sigma^\top$ with $V = \frac{1}{N}\sigma^\top\sigma$.

Both make each stored pattern an exact fixed point.

We store five letters on a 5×5 pixel grid, and because letters share many pixels, they are correlated, so the projection rule is the right one.

```
PATTERNS = {
    'A': ["01110", "10001", "11111", "10001", "10001"],
    'E': ["11111", "10000", "11110", "10000", "11111"],
    'I': ["11111", "00100", "00100", "00100", "11111"],
    'O': ["01110", "10001", "10001", "10001", "01110"],
    'T': ["11111", "00100", "00100", "00100", "00100"],
}

def to_vector(rows):
    return np.array([1 if c == '1' else -1 for r in rows for c in r])

letters = list(PATTERNS)
σ = np.array([to_vector(PATTERNS[c]) for c in letters])      # (p, N)

def projection_rule(patterns):
    "Weight matrix making each (correlated) pattern an exact fixed point."
    N = patterns.shape[1]
    Σ = patterns.T                                           # (N, p)
    V = Σ.T @ Σ / N
    return Σ @ np.linalg.inv(V) @ Σ.T / N

def energy(w, s):
    "Hopfield energy; stored patterns are local minima."
    return -0.5 * s @ w @ s

def recall(w, s0, max_iter=30):
    "Iterate sgn(w s) to a fixed point."
    s = s0.copy()
    for _ in range(max_iter):
        s_new = np.sign(w @ s)
        s_new[s_new == 0] = 1
        if np.array_equal(s_new, s):
            break
        s = s_new
    return s
```

(continues on next page)

(continued from previous page)

```
w_hop = projection_rule(σ)
fixed = all(np.array_equal(np.sign(w_hop @ σ[i]), σ[i]) for i in range(len(letters)))
print(f"all stored letters are fixed points: {fixed}")
print(f"energy of each stored letter: {[round(energy(w_hop, σ[i]), 1) for i in␣
↪range(len(letters))]}")
```

```
all stored letters are fixed points: True
energy of each stored letter: [np.float64(-12.5), np.float64(-12.5), np.float64(-
↪12.5), np.float64(-12.5), np.float64(-12.5)]
```

Every stored letter sits at the same energy $-N/2$ and is a fixed point.

Now corrupt a few pixels and let the network fall back to the nearest memory.

```
rng = np.random.default_rng(3)
show = ['E', 'O', 'T']

fig, axes = plt.subplots(len(show), 3, figsize=(6, 6))
for row, letter in enumerate(show):
    i = letters.index(letter)
    corrupt = σ[i].copy()
    corrupt[rng.choice(25, 4, replace=False)] *= -1             # flip 4 pixels
    recovered = recall(w_hop, corrupt)
    for col, (img, title) in enumerate([(σ[i], "stored"),
                                        (corrupt, "corrupted"),
                                        (recovered, "recovered")]):
        axes[row, col].imshow(img.reshape(5, 5), cmap='binary', vmin=-1, vmax=1)
        axes[row, col].set_xticks([])
        axes[row, col].set_yticks([])
        if row == 0:
            axes[row, col].set_title(title)
plt.tight_layout()
plt.show()
```

The corrupted letters snap back to their originals.

How reliably depends on how much is corrupted and on how correlated the patterns are.

A corrupted letter can occasionally land in a *different* letter's basin, or in a spurious mixture the designer never stored.

```
for n_flip in (3, 5, 7):
    correct = 0
    for i in range(len(letters)):
        for _ in range(40):
            corrupt = σ[i].copy()
            corrupt[rng.choice(25, n_flip, replace=False)] *= -1
            if np.array_equal(recall(w_hop, corrupt), σ[i]):
                correct += 1
    print(f"{n_flip} pixels flipped: recovered exactly {correct}/{len(letters)*40}")
```

```
3 pixels flipped: recovered exactly 179/200
5 pixels flipped: recovered exactly 138/200
7 pixels flipped: recovered exactly 77/200
```

The mechanism is worth naming because it recurs throughout this program.

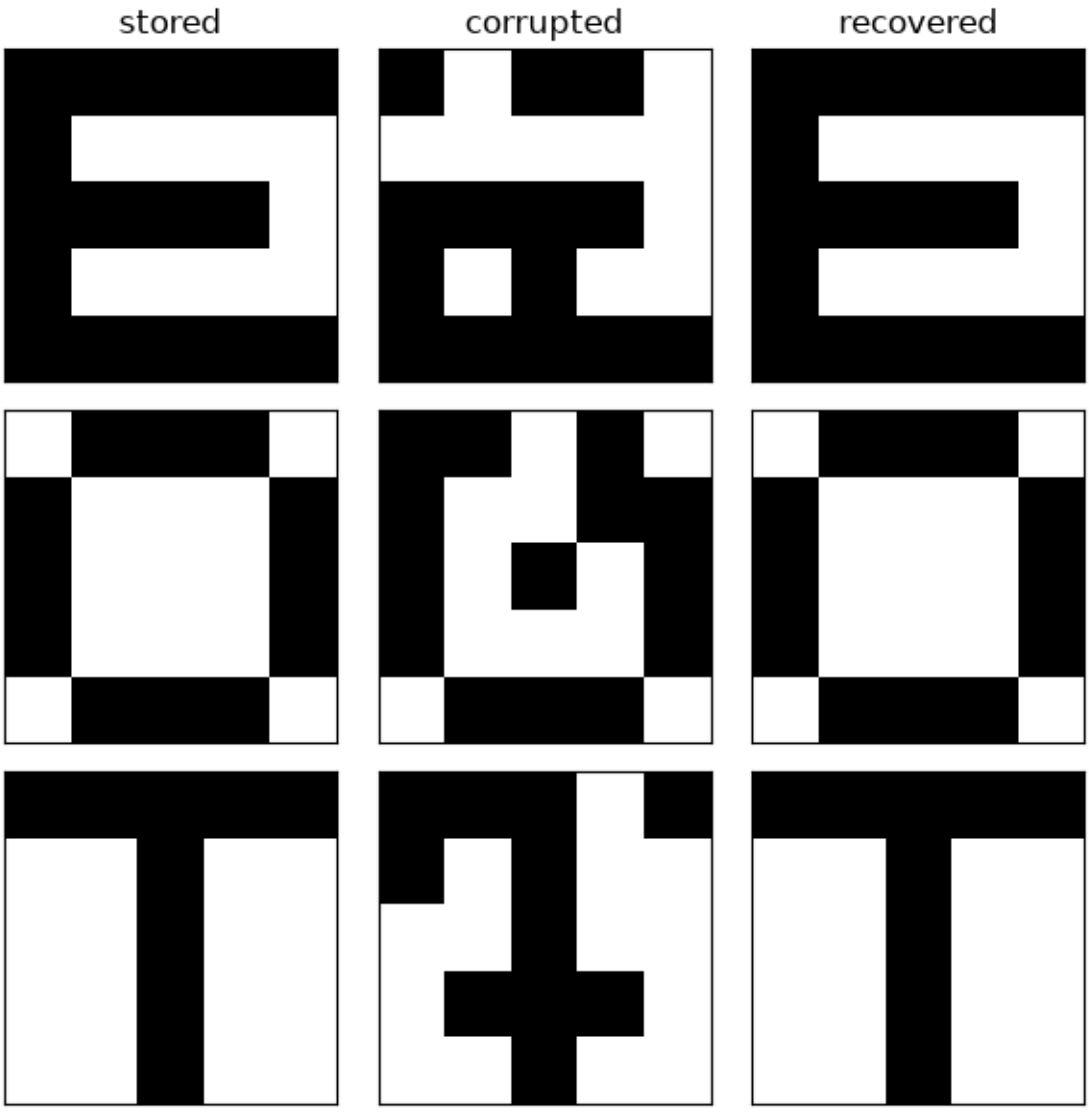


Fig. 88.2: Hopfield recall of letters from corrupted inputs

Recall is **descent on an energy function** whose local minima are the stored patterns.

Feeding in a corrupted pattern places the system on the energy surface near the right minimum, and the dynamics roll it down.

This is exactly the geometry behind **simulated annealing**, the method the book invokes for escaping *unwanted* local minima: add a controlled amount of random "shaking," large at first and declining over time, so the system can hop out of shallow spurious basins before settling into a deep one.

It is the stochastic cousin of Newton's method, and it reappears as the "genetic" experimentation in classifier systems.

88.4 The genetic algorithm

The third brain does not descend a smooth surface at all.

Holland's **genetic algorithm** searches "rugged landscapes" that lack the smoothness Newton's method needs, by evolving a *population* of candidate solutions encoded as binary strings.

Given a fitness function f to maximize, and a population of N strings of length S, one generation applies four operators:

1. **Evaluation**: compute each string's fitness $f(x_i)$.
2. **Reproduction**: copy strings into the next generation with probability proportional to fitness (a "biased roulette wheel").
3. **Crossover**: pair strings up and swap their tails at a random cut point, forming children that recombine their parents' segments.
4. **Mutation**: flip each bit independently with a small probability p_m.

Reproduction concentrates the population on what already works; crossover and mutation inject new candidates.

Crossover is the workhorse: it "preserves long sections of genetic structure" — Holland's *schemata* — while still exploring, provided the population is diverse enough to recombine.

Crucially, the individual strings do not learn: each lives one generation and dies.

Only the *society*, the sequence of populations, learns.

That is what makes the genetic algorithm awkward as a model of one brain, and a better fit as a model of a population or a market, a point that matters when we reach *Money as a Medium of Exchange among Artificially Intelligent Agents*.

88.4.1 Axelrod's iterated prisoner's dilemma

Axelrod [1987] used the genetic algorithm to evolve strategies for the **iterated prisoner's dilemma**, the game whose round-robin tournament, famously, was won by tit-for-tat.

Two players each choose to Cooperate or Defect; the payoffs reward defection against a cooperator but punish mutual defection.

```
# 1 = Cooperate, 0 = Defect;  T > R > P > S is the prisoner's dilemma ranking
T_pay, R_pay, P_pay, S_pay = 5, 3, 1, 0

def payoff(a, b):
    "My payoff when I play a against opponent's b."
    if a and b:       return R_pay        # both cooperate
    if not a and not b: return P_pay      # both defect
    if not a and b:   return T_pay        # I defect, they cooperate
    return S_pay                          # I cooperate, they defect
```

Following Axelrod, a strategy is a **70-bit string**: a strategy conditions on the outcomes of the last three rounds.

Each round is one of four outcomes (my move, opponent's move), so three rounds give $4^3 = 64$ possible histories, and 64 bits specify the action in each.

The remaining 6 bits encode a presumed pre-game history to get the first moves going.

```
def play(gene, opponent, n_rounds=150):
    "Play a 70-bit genetic strategy against an opponent; return average payoffs."
    action, premise = gene[:64], gene[64:]
    hist = list(premise)                        # last 3 rounds as [my, opp] pairs
    my_moves, opp_moves = [], []
    my_total = opp_total = 0
    for t in range(n_rounds):
        idx = 0
        for bit in hist:
            idx = (idx << 1) | bit              # 6 history bits -> index 0..63
        a = action[idx]                         # my move from the lookup table
        b = opponent(my_moves, opp_moves, t)
        my_total += payoff(a, b)
        opp_total += payoff(b, a)
        my_moves.append(a)
        opp_moves.append(b)
        hist = [a, b] + hist[:4]                # roll the 3-round window
    return my_total / n_rounds, opp_total / n_rounds
```

The strategy is bred to do well against a fixed panel of opponents.

Each panel member is a function of the histories *as that member sees them*: its first argument is what its opponent has played and its second is what it has played itself.

```
panel_rng = np.random.default_rng(2024)

def all_cooperate(opp_moves, own_moves, t): return 1
def all_defect(opp_moves, own_moves, t):    return 0
def tit_for_tat(opp_moves, own_moves, t):   return 1 if t == 0 else opp_moves[-1]
def grudger(opp_moves, own_moves, t):       return 0 if 0 in opp_moves else 1
def random_play(opp_moves, own_moves, t):   return int(panel_rng.random() < 0.5)

PANEL = {'AllC': all_cooperate, 'AllD': all_defect, 'TFT': tit_for_tat,
         'Grudger': grudger, 'Random': random_play}

def fitness(gene):
    "Average payoff against the whole panel."
    return np.mean([play(gene, opp)[0] for opp in PANEL.values()])
```

Tit-for-tat copies what its opponent last did, and the grudger defects forever once its opponent has defected once.

Now evolve.

```
def evolve(N=60, generations=80, p_mut=0.01, seed=0):
    rng = np.random.default_rng(seed)
    pop = rng.integers(0, 2, (N, 70))
    best_hist, mean_hist = [], []
    for _ in range(generations):
        fit = np.array([fitness(pop[i]) for i in range(N)])
        best_hist.append(fit.max())
        mean_hist.append(fit.mean())
        weight = fit - fit.min() + 1e-6                    # shift positive for␣
```

(continues on next page)

(continued from previous page)

```
↪roulette
        nxt = np.empty_like(pop)
        for k in range(0, N, 2):
            i, j = np.searchsorted(np.cumsum(weight), rng.random(2) * weight.sum())
            cut = rng.integers(1, 70)                       # single-point crossover
            c1 = np.concatenate([pop[i, :cut], pop[j, cut:]])
            c2 = np.concatenate([pop[j, :cut], pop[i, cut:]])
            for c in (c1, c2):
                c[rng.random(70) < p_mut] ^= 1              # mutation
            nxt[k], nxt[k+1] = c1, c2
        pop = nxt
    fit = np.array([fitness(pop[i]) for i in range(N)])
    return pop[fit.argmax()], np.array(best_hist), np.array(mean_hist)


panel_rng = np.random.default_rng(0)      # reset the panel's randomizer
champion, best_hist, mean_hist = evolve()
print(f"generation 0: best fitness {best_hist[0]:.3f}")
print(f"generation {len(best_hist)-1}: best fitness {best_hist[-1]:.3f}")
```

```
generation 0: best fitness 2.504
generation 79: best fitness 2.672
```

```
fig, ax = plt.subplots(figsize=(7.5, 4))
ax.plot(best_hist, label="best in population", lw=2)
ax.plot(mean_hist, label="population mean", lw=2)
ax.set_xlabel("generation")
ax.set_ylabel("panel fitness")
ax.legend(frameon=False)
plt.show()
```

Fitness climbs, then holds.

What kind of strategy did evolution find?

```
print("evolved champion's average payoff against each opponent:")
for name, opp in PANEL.items():
    me, them = play(champion, opp)
    print(f"  vs {name:8s}:  me = {me:.2f},  opponent = {them:.2f}")
```

```
evolved champion's average payoff against each opponent:
  vs AllC    :  me = 5.00,  opponent = 0.00
  vs AllD    :  me = 1.00,  opponent = 1.00
  vs TFT     :  me = 2.98,  opponent = 2.98
  vs Grudger :  me = 1.01,  opponent = 1.07
  vs Random  :  me = 2.80,  opponent = 1.33
```

The evolved strategy is *nice and retaliatory*, in Axelrod's language.

It reaches mutual cooperation with the cooperative opponents (AllC, TFT, Grudger), refuses to be suckered by AllD (settling into mutual defection near the punishment payoff), and exploits Random.

Nobody told it to cooperate; cooperation emerged because it pays against a panel that contains cooperators.

And it does all this *better than tit-for-tat itself* would against the same panel.

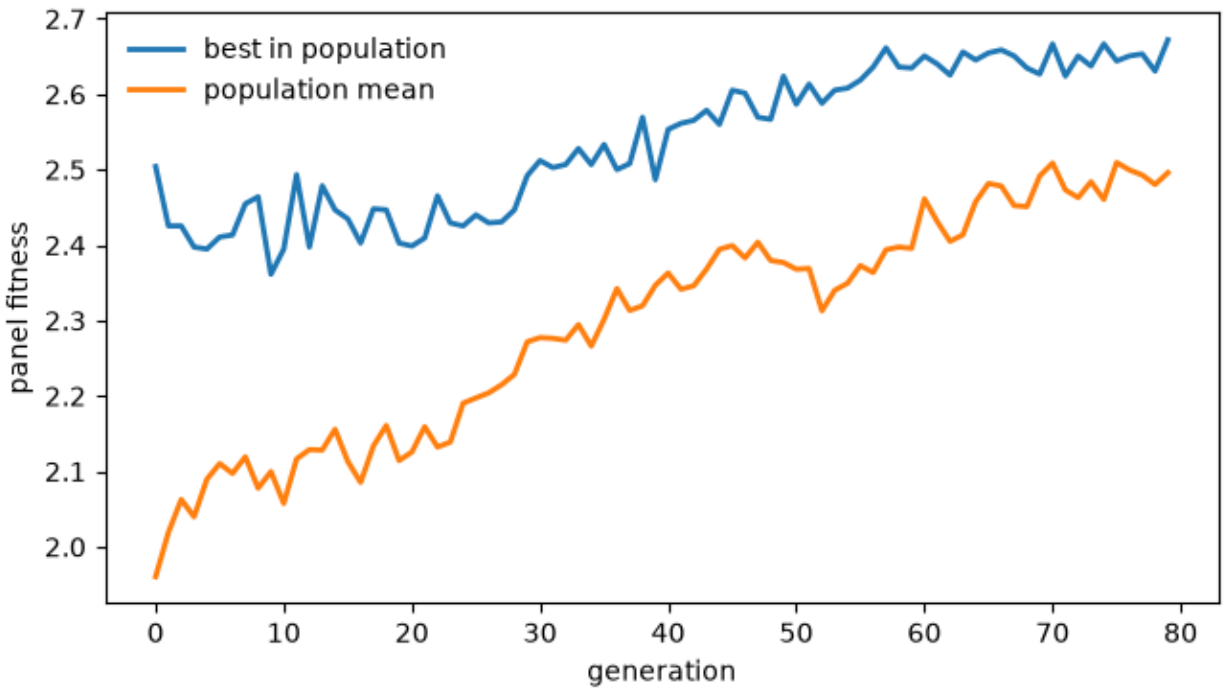


Fig. 88.3: Population fitness across generations

```
def play_function(strategy, opponent, n=150):
    "Average payoff of a stateful strategy function against an opponent."
    mine, theirs, total = [], [], 0
    for t in range(n):
        # each side is passed its opponent's history first, then its own
        a, b = strategy(theirs, mine, t), opponent(mine, theirs, t)
        total += payoff(a, b)
        mine.append(a)
        theirs.append(b)
    return total / n

panel_rng = np.random.default_rng(0)
tft_fitness = np.mean([play_function(tit_for_tat, opp) for opp in PANEL.values()])
print(f"tit-for-tat's own panel fitness:  {tft_fitness:.3f}")
print(f"evolved champion's panel fitness: {best_hist[-1]:.3f}")
```

```
tit-for-tat's own panel fitness:  2.432
evolved champion's panel fitness: 2.672
```

This reproduces Axelrod's headline finding: the genetic algorithm "produced a strategy that would have won the tournament, and in particular would have outplayed the 'tit-for-tat' strategy that won the tournament."

It behaves much like tit-for-tat against cooperators, but its extra memory lets it squeeze more out of exploitable opponents that tit-for-tat leaves alone.

The margin is not large, and it should not be.

Tit-for-tat is a strong strategy against this panel; what the extra memory buys is a modest edge against the opponents that tit-for-tat handles adequately but not optimally.

88.5 Classifier systems

The genetic algorithm evolves a population in which no individual learns.

Holland's **classifier system** puts the same evolutionary machinery *inside a single agent*, as a model of one brain.

Sargent describes it as Holland's vision of the mind as a **competitive economy**:

> The statements compete with one another for the opportunity to decide. The classifier system incorporates elements of the genetic algorithm with other aspects in a way that represents a brain in terms that Holland describes as a competitive economy.

A classifier system has:

- **Classifiers**, if-then rules encoded as strings over the trinary alphabet $\{0, 1, \#\}$ whose condition part matches states and whose action part prescribes a move, with $\#$ a wildcard ("I don't care") that lets general rules coexist with specific ones.
- **A decoder** that, given the current state, finds which classifiers' conditions match.
- **An auction** that selects one matching classifier to act: the strongest, or one chosen with probability proportional to strength.
- **An accounting system** that updates each classifier's **strength** — a running average of the net rewards its decisions earn — and, in sequential problems, passes reward *backward* from rules that collect payoffs to the rules that set them up.
- **Genetic operators** that create new classifiers, generalize (add $\#$'s) and specialize (remove them), so the system's very vocabulary evolves.

88.5.1 A two-armed bandit

The simplest classifier system, due to Brian Arthur and Carl Simon, plays a **two-armed bandit**.

Arm i pays a random reward with mean μ_i, and $\mu_1 > \mu_2$, but the player knows neither distribution.

The classifier system holds two rules — "pull arm 1" and "pull arm 2" — whose conditions are always met.

Each rule's strength is the running average of the payoff that arm has delivered, and the arm to pull is chosen with probability proportional to strength.

The two rules start with **equal** strengths: the system is told nothing about either arm and must build its estimates out of the rewards it experiences.

```
def two_armed_bandit(μ, σ=0.5, T=20_000, seed=0):
    "Strength = running average of an arm's payoff; pull ∝ strength."
    rng = np.random.default_rng(seed)
    S = np.ones(2)                              # equal strengths: no prior knowledge
    τ = np.array([1, 1])                       # pull counters
    pulls = np.empty(T, int)
    for t in range(T):
        w = np.maximum(S, 1e-9)                 # strengths can dip below zero early on
        i = 0 if rng.random() < w[0] / w.sum() else 1
        reward = μ[i] + σ * rng.standard_normal()
        τ[i] += 1
        S[i] += (reward - S[i]) / τ[i]          # running average
        pulls[t] = i
    return pulls

pulls = two_armed_bandit([3.0, 1.0])
```

(continues on next page)

(continued from previous page)

```
frac_best = np.cumsum(pulls == 0) / np.arange(1, len(pulls) + 1)

fig, ax = plt.subplots(figsize=(7.5, 4))
ax.plot(frac_best, lw=1)
ax.axhline(0.75, color='k', ls='--', label="probability match  $\\mu_1/(\\mu_1+\\mu_2)
↪$")
ax.axhline(1.0, color='C3', ls=':', label="optimal (always best arm)")
ax.set_xlabel("$t$")
ax.set_ylabel("fraction of pulls on the best arm")
ax.set_ylim(0.5, 1.05)
ax.legend(frameon=False)
plt.show()
```

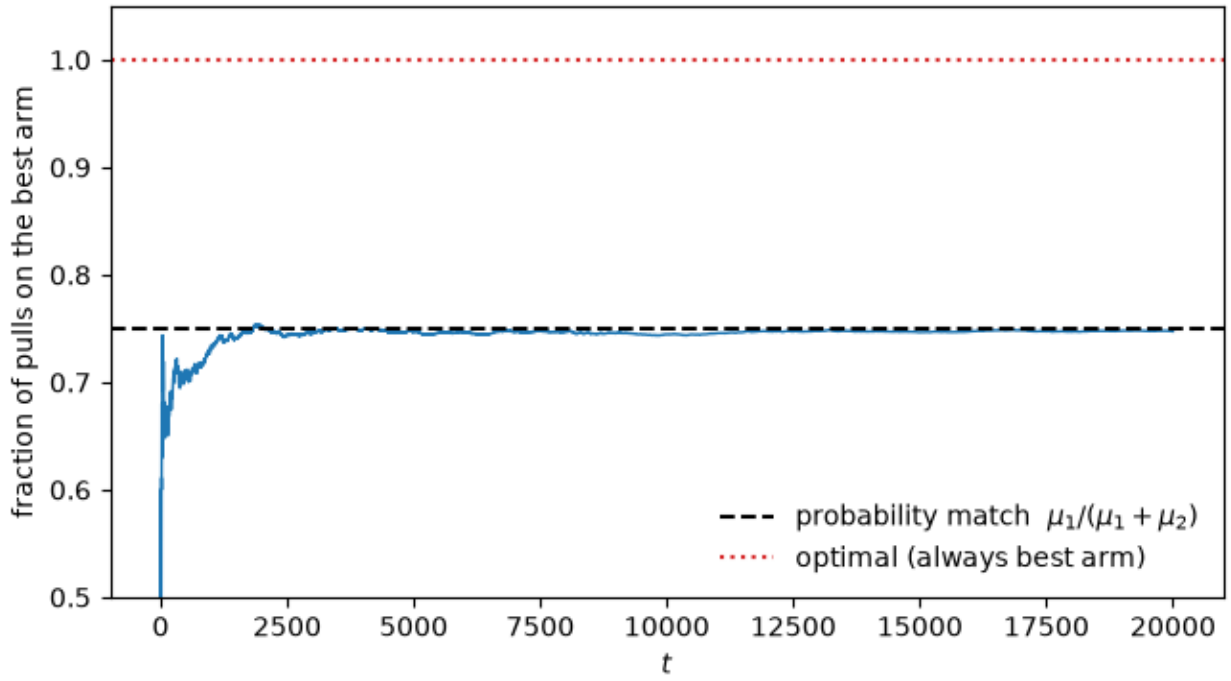


Fig. 88.4: The classifier bandit converges to probability matching

The fraction of pulls on the better arm converges, but **not to one**.

It converges to $\mu_1/(\mu_1 + \mu_2)$, the arm's share of total expected reward.

```
for μ in ([1.0, 0.5], [2.0, 1.0], [3.0, 1.0]):
    frac = np.mean(two_armed_bandit(μ)[-5000:] == 0)
    print(f"μ = {μ}: fraction on best arm = {frac:.3f}, "
          f"probability match = {μ[0]/(μ[0]+μ[1]):.3f}  (optimal = 1.0)")
```

```
μ = [1.0, 0.5]: fraction on best arm = 0.655, probability match = 0.667  (optimal␣
↪= 1.0)
μ = [2.0, 1.0]: fraction on best arm = 0.659, probability match = 0.667  (optimal␣
↪= 1.0)
μ = [3.0, 1.0]: fraction on best arm = 0.750, probability match = 0.750  (optimal␣
```

(continues on next page)

(continued from previous page)

```
    ↪= 1.0)
```

Arthur and Simon proved this: the strength-proportional classifier **probability-matches**.

It plays the arms in proportion to their expected rewards rather than concentrating on the best one, so it leaves reward on the table forever.

That is not a bug to paper over; it is a lesson about accounting.

A classifier system is only as good as the scheme by which strength is assigned and passed around.

The naive rule here matches probabilities; better rules do better.

And in a *sequential* problem — where a rule's payoff comes only much later, through a chain of intermediate decisions — the accounting must do something harder still: reward a rule for merely *setting up* a profitable future decision.

Holland's device for this is the **bucket brigade**: each acting classifier pays part of its strength to the classifier that acted just before it, the one that moved the system into the state where the current rule could act.

Reward paid at the end of a chain seeps backward, rule by rule, until early setup rules that never touch a payoff directly acquire strength for enabling it.

Designing that backward flow is the central craft of building a classifier system, and it is exactly what *Money as a Medium of Exchange among Artificially Intelligent Agents* must get right to make agents learn to accept money today for the sake of a trade tomorrow.

88.6 Evolutionary programming

We have surveyed four brains.

The last idea is about what to *do* with them.

A recurring finding of this whole section is that systems of adaptive agents, however plodding, tend to converge on rational expectations equilibria.

Adaptive Agents in an Overlapping Generations Monetary Economy and *Learning, Approximation, and Equilibrium Computation* showed least squares learners finding one; *Exchange Rate Indeterminacy, Learning, and Experiments* showed Newton learners settling on one (of many).

Evolutionary programming turns that tendency into a tool.

If a population of adaptive agents reliably converges to an equilibrium, we can run the population as a *method for computing* the equilibrium, especially in models too complicated to solve by hand.

Sargent is careful about what is and isn't going on:

> The adaptive agents are 'teaching' the economist in the same sense that any numerical algorithm for solving nonlinear equations 'teaches' a mathematician. When these agents can 'teach' us something, it is because we designed them to do so.

This is the same duality that ran through *Learning, Approximation, and Equilibrium Computation*: a learning economy is a decentralized equilibrium computation, and an equilibrium computation is a centralized learning algorithm.

The genetic algorithm and the classifier system are simply richer computational engines than recursive least squares, able to search rugged landscapes and to discover the structure of a good rule, not just tune the coefficients of a fixed one.

The showcase is Kiyotaki and Wright [1989]'s search-theoretic model of money, in which the medium of exchange is not assumed but must **emerge** from how agents choose to trade.

The equilibrium is a set of trading strategies and matching probabilities, and for enriched versions of the model it is hard to characterize analytically.

Marimon *et al.* [1990] put populations of Holland classifier systems into that environment and watched them converge to an equilibrium, then built a five-good version, with no known analytical solution, and let the classifier systems suggest what the equilibrium looked like.

88.7 Concluding remarks

The four brains in this catalogue differ in what they take as given.

The perceptron is handed a functional form and asked only for its coefficients, which is why an econometrician recognizes it immediately.

The Hopfield network is handed the patterns themselves and asked only to recall them.

The genetic algorithm is handed nothing but a fitness function and must search a space in which it cannot compute a gradient.

The classifier system is handed a vocabulary in which rules can be written and must discover which rules are worth holding.

Moving down that list, we hand the agent less and ask it to find more, which is precisely the direction the bounded-rationality program pushes us.

Two cautions emerged along the way, and both return in the next lecture.

The genetic algorithm's individuals do not learn: only the population does, which makes it a model of a society rather than of a mind.

And the bandit showed that a classifier system's performance is decided by its *accounting* — by how strength is assigned, bid, and passed along — rather than by the fact of having classifiers at all.

Money as a Medium of Exchange among Artificially Intelligent Agents assembles this machinery into agents who learn, from scratch, to use money, and both cautions bear directly on what those agents turn out to be able to learn.

88.8 Exercises

Exercise 88.8.1

The Hopfield network's recall is descent on the energy $E(s) = -\frac{1}{2} s^\top w s$.

Verify the descent directly.

Take a stored letter, corrupt several pixels, and record the energy at each step of the recall dynamics.

Confirm that energy never increases and that recall halts at (or below) the stored pattern's energy.

Then corrupt patterns more heavily and classify the failures: does recall land on a *different* stored letter, or on a *spurious* state that was never stored?

Compare the energies in each case, and use them to explain why the error happens.

Solution

```
def recall_with_energy(w, s0, max_iter=30):
    s = s0.copy()
    trace = [energy(w, s)]
```

```
    for _ in range(max_iter):
        s_new = np.sign(w @ s)
        s_new[s_new == 0] = 1
        trace.append(energy(w, s_new))
        if np.array_equal(s_new, s):
            break
        s = s_new
    return s, trace

rng = np.random.default_rng(1)
i = letters.index('E')
corrupt = σ[i].copy()
corrupt[rng.choice(25, 5, replace=False)] *= -1
final, trace = recall_with_energy(w_hop, corrupt)

print(f"energy along the recall path: {[round(e, 2) for e in trace]}")
print(f"monotonically non-increasing: {all(np.diff(trace) <= 1e-9)}")
print(f"recovered the intended letter 'E': {np.array_equal(final, σ[i])}")
```

```
energy along the recall path: [np.float64(-6.38), np.float64(-12.5), np.
↪float64(-12.5)]
monotonically non-increasing: True
recovered the intended letter 'E': True
```

The energy falls monotonically along every recall path: the dynamics can only move downhill, which is why the network always halts at a local minimum.

Now corrupt more heavily and sort the outcomes into three kinds: the intended letter, a *different* stored letter, and a spurious state that is a fixed point but was never taught.

```
def classify_outcome(final, i):
    if np.array_equal(final, σ[i]):
        return "intended"
    if any(np.array_equal(final, σ[k]) for k in range(len(letters))):
        return "wrong letter"
    return "spurious"

counts = {"intended": 0, "wrong letter": 0, "spurious": 0}
wrong_energy, spurious_energy = [], []
for i in range(len(letters)):
    for seed in range(200):
        r = np.random.default_rng(1000*i + seed)
        corrupt = σ[i].copy()
        corrupt[r.choice(25, 7, replace=False)] *= -1
        final = recall(w_hop, corrupt)
        kind = classify_outcome(final, i)
        counts[kind] += 1
        if kind == "wrong letter":
            wrong_energy.append(energy(w_hop, final))
        elif kind == "spurious":
            spurious_energy.append(energy(w_hop, final))

print(f"7-pixel corruptions ({sum(counts.values())} trials): {counts}")
print(f"stored patterns all have energy {energy(w_hop, σ[0]):.1f}")
print(f"wrong-letter results: energy in [{min(wrong_energy):.1f}, {max(wrong_
↪energy):.1f}]")
print(f"spurious results:     energy in [{min(spurious_energy):.1f}, {max(spurious_
↪energy):.1f}]")
```

```
7-pixel corruptions (1000 trials): {'intended': 404, 'wrong letter': 62,
↪'spurious': 534}
```

```
stored patterns all have energy -12.5
wrong-letter results: energy in [-12.5, -12.5]
spurious results:     energy in [-12.5, -10.4]
```

Both failure modes occur, and at this level of corruption the spurious states are the more common of the two by some margin.

A **wrong-letter** result sits at exactly the stored energy $-N/2$: the corruption pushed the starting point clear across a basin boundary, into the domain of attraction of a *different* stored memory that is just as deep.

Energy descent converges to *a* minimum but cannot guarantee the *nearest* one when correlated patterns have interlocking basins.

A **spurious** result is a local minimum the designer never intended, typically a blend of stored patterns, created as an artifact of the storage rule.

Its energy ranges from shallower than the stored patterns' all the way down to exactly their depth, so depth alone does not identify a memory as one we asked for.

Some of the deepest spurious states are sign reversals: because $\text{sgn}(w(-s)) = -\text{sgn}(ws)$, the vector $-\sigma$ is a fixed point of the same energy whenever σ is, and the network stores each letter's photographic negative whether we wanted it or not.

Because all of these are genuine local minima, downhill dynamics cannot escape any of them.

Both are why the network is imperfect, and both are why **simulated annealing** matters: adding declining random shaking lets the system hop out of a shallow spurious basin, or across a basin boundary, before settling, something pure energy descent can never do.

Exercise 88.8.2

The genetic algorithm's exploration comes from two operators, crossover and mutation.

The book argues that crossover "lies at the heart of the algorithm," while mutation alone "is a poor mechanism for injecting diversity."

Test the claim on Axelrod's game.

Write a mutation-only variant (each child is a mutated copy of one selected parent, with no crossover) and compare the fitness it reaches against the full algorithm.

The book is careful to say *when* mutation alone is weak: "when the mutation rate is set to a very low value, mutation alone is a poor mechanism for injecting diversity."

So run the comparison at a **low** mutation rate, where crossover has to carry the exploration.

Solution

```
def evolve_no_crossover(N=60, generations=60, p_mut=0.005, seed=0):
    "Mutation-only variant: each child is a mutated copy of one selected parent."
    rng = np.random.default_rng(seed)
    pop = rng.integers(0, 2, (N, 70))
    best_hist = []
    for _ in range(generations):
        fit = np.array([fitness(pop[i]) for i in range(N)])
        best_hist.append(fit.max())            # recorded exactly as in `evolve`
        weight = fit - fit.min() + 1e-6
```

```
        parents = np.searchsorted(np.cumsum(weight), rng.random(N) * weight.sum())
        nxt = pop[parents].copy()
        nxt[rng.random((N, 70)) < p_mut] ^= 1
        pop = nxt
    return best_hist[-1]

rows = []
for seed in range(5):
    panel_rng = np.random.default_rng(seed)
    with_x = evolve(N=60, generations=60, p_mut=0.005, seed=seed)[1][-1]
    panel_rng = np.random.default_rng(seed)
    without_x = evolve_no_crossover(seed=seed)
    rows.append((seed, with_x, without_x))

print(f"{'seed':>4} {'with crossover':>16} {'mutation only':>16}")
for sd, a, b in rows:
    print(f"{sd:>4} {a:>16.3f} {b:>16.3f}")
print(f"{'mean':>4} {np.mean([r[1] for r in rows]):>16.3f} "
      f"{np.mean([r[2] for r in rows]):>16.3f}")
```

```
seed   with crossover    mutation only
   0            2.643            2.572
   1            2.540            2.583
   2            2.664            2.547
   3            2.657            2.616
   4            2.669            2.556
mean            2.635            2.575
```

At a low mutation rate crossover clearly wins: it reaches a higher fitness than the mutation-only variant on almost every seed.

The reason is the one the book gives.

With little mutation, a mutation-only population can only inch forward one rare bit-flip at a time, and quickly loses diversity as selection copies its few best strings.

Crossover instead recombines whole *segments* that have already proved useful in different strings: a good way of handling one opponent spliced onto a good way of handling another.

It injects large, structured variation while preserving the schemata that fitness has already favored, which is why Holland put it at the center of the algorithm.

Note

The margin narrows, and can even vanish, at higher mutation rates: when mutation is generating plenty of diversity on its own, crossover's contribution is less pivotal.
Try re-running the comparison at `p_mut=0.02` to see the effect shrink.
Whether crossover is decisive depends on the mutation rate *and* on whether the encoding lines up useful building blocks with contiguous bit segments, which, for this history-indexed strategy table, it only partly does.

Exercise 88.8.3

The two-armed bandit classifier probability-matches, which is suboptimal: it keeps pulling the worse arm a fixed fraction of the time forever.

A natural fix is to make the auction **greedier** as the system gains confidence.

Replace the strength-proportional choice rule with a softmax,

$$\pi_1 = \frac{e^{\beta S_1}}{e^{\beta S_1} + e^{\beta S_2}},$$

where β controls greediness ($\beta \to \infty$ always picks the stronger arm).

Implement it and show how the long-run fraction on the best arm depends on β.

How large must β be to move the classifier from probability matching toward the optimal policy?

Solution

```
def bandit_softmax(μ, β, σ=0.5, T=20_000, seed=0):
    rng = np.random.default_rng(seed)
    S = np.ones(2)                                  # equal strengths, as before
    τ = np.array([1, 1])
    pulls = np.empty(T, int)
    for t in range(T):
        p1 = 1 / (1 + np.exp(-β * (S[0] - S[1])))
        i = 0 if rng.random() < p1 else 1
        reward = μ[i] + σ * rng.standard_normal()
        τ[i] += 1
        S[i] += (reward - S[i]) / τ[i]
        pulls[t] = i
    return np.mean(pulls[-5000:] == 0)

μ = [3.0, 1.0]
print(f"probability match target = {μ[0]/(μ[0]+μ[1]):.3f},  optimal = 1.000\n")
for β in (0.5, 1.0, 2.0, 4.0, 8.0):
    frac = bandit_softmax(μ, β)
    print(f"β = {β:>4}: fraction on best arm = {frac:.3f}")
```

```
probability match target = 0.750,  optimal = 1.000

β =  0.5: fraction on best arm = 0.732
β =  1.0: fraction on best arm = 0.888
β =  2.0: fraction on best arm = 0.980
β =  4.0: fraction on best arm = 1.000
β =  8.0: fraction on best arm = 1.000
```

As β rises the classifier abandons probability matching and concentrates on the better arm, approaching the optimal policy of always pulling it.

The exercise makes concrete why the *accounting and auction rules* — not just the fact of having classifiers — determine how well a classifier system performs.

The strength-proportional rule of Arthur and Simon is one point on a spectrum; a greedy rule sits at the other end.

Real classifier systems, including the one in *Money as a Medium of Exchange among Artificially Intelligent Agents*, choose their auction and strength-update rules deliberately, precisely because the choice is what separates a system that merely matches probabilities from one that learns to act well.

CHAPTER

EIGHTYNINE

MONEY AS A MEDIUM OF EXCHANGE AMONG ARTIFICIALLY INTELLIGENT AGENTS

Contents

- *Money as a Medium of Exchange among Artificially Intelligent Agents*
  - *Overview*
  - *The Kiyotaki-Wright environment*
  - *Classifier systems*
  - *Implementation*
  - *Economy A1.1: does a medium of exchange emerge?*
  - *Economy A2.1: when theory predicts speculation*
  - *Economy B.1: a different production pattern*
  - *The genetic algorithm*
  - *Economy A1.2: learning from random rules*
  - *Economy C: fiat money*
  - *Economy D: five goods, five types*
  - *Concluding remarks*
  - *Exercises*

89.1 Overview

Kiyotaki and Wright [Kiyotaki and Wright, 1989] studied an economy in which there is no double coincidence of wants.

Trade can occur only if some good is accepted not because it is wanted, but because it can be passed along later.

A good that plays that role is a **medium of exchange**.

Kiyotaki and Wright characterized *stationary Nash equilibria* of such an economy under the assumption that agents are fully rational: agents know the distribution of goods across their trading partners and best-respond to it.

Marimon, McGrattan and Sargent [Marimon *et al.*, 1990] asked a different question.

Suppose we drop rationality altogether and put in its place a collection of **artificially intelligent agents** who begin with arbitrary, even random, rules of thumb, and who adapt those rules only by keeping score of what has paid off in the past.

Will such agents *learn* to use a medium of exchange?

And when the rational-expectations model has more than one equilibrium, which one, if any, will emerge?

The learning device that Marimon, McGrattan and Sargent used is John Holland's **classifier system** [Holland, 1975, Holland *et al.*, 1986]: a population of if-then rules, an auction that decides which rule acts, and an accounting system that credits rules that lead to good outcomes and debits rules that lead to bad ones.

Optionally, a **genetic algorithm** breeds new rules and retires old ones.

This lecture rebuilds their computational experiments.

The main findings that we shall reproduce are:

1. Starting from a complete enumeration of rules with equal strengths, or even from randomly generated rules, holdings and trading patterns converge to a stationary Nash equilibrium of the Kiyotaki-Wright model.
2. When the Kiyotaki-Wright model has both a *fundamental* and a *speculative* equilibrium, the artificially intelligent agents select the fundamental one – the one in which the good with the lowest storage cost circulates as money.
3. An intrinsically worthless, costlessly stored object – **fiat money** – is accepted in trade by agents who have to discover its usefulness for themselves.
4. The same machinery works in an economy with five goods and five types for which the authors had no analytical characterization of equilibrium: the algorithm is used as an *equilibrium-discovery device*.

Let's start with some imports.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from dataclasses import dataclass
```

89.2 The Kiyotaki-Wright environment

There are three types of agents, indexed by $i = 1, 2, 3$, and three goods, indexed by $k = 1, 2, 3$.

A type i agent derives utility only from consuming good i.

He has a technology for producing good i^*, where $i^* \neq i$.

In Kiyotaki and Wright's *model A*, the production pattern is the "Wicksell triangle"

type i	produces i^*	consumes
1	2	1
2	3	2
3	1	3

so there is no double coincidence of wants: the agent who has what you want never wants what you have.

All goods are indivisible and each agent can store exactly one unit of exactly one good from one period to the next.

Storing good k for one period costs s_k, with

$$s_3 > s_2 > s_1 > 0.$$

There are equal numbers A_i of agents of each type, so $A = 3A_i$ in total.

Each period every agent is randomly matched with exactly one other agent, without regard to type.

Write x_{at} for the good that agent a carries into date t and $\rho_t(a)$ for the agent with whom a is matched.

The **pre-trade state** of agent a is the pair

$$z_{at} = (x_{at},\ x_{\rho_t(a)t}).$$

Each period each agent makes two decisions in sequence.

First, having seen z_{at}, he decides whether to propose a trade,

$$\lambda_{at} = \begin{cases} 1 & \text{propose to trade } x_{at} \text{ for } x_{\rho_t(a)t} \\ 0 & \text{refuse} \end{cases}$$

Trade occurs if and only if $\lambda_{at}\lambda_{\rho_t(a)t} = 1$, so post-trade holdings are

$$x^+_{at} = (1 - \lambda_{at}\lambda_{\rho_t(a)t})x_{at} + \lambda_{at}\lambda_{\rho_t(a)t}x_{\rho_t(a)t}. \tag{89.1}$$

Second, he decides whether to consume what he is left holding,

$$\gamma_{at} = \begin{cases} 1 & \text{consume } x^+_{at} \\ 0 & \text{carry } x^+_{at} \text{ into } t+1 \end{cases}$$

If he consumes he immediately produces good $f(a) = i^*$ and carries that into $t + 1$.

Hence

$$x_{a,t+1} = \gamma_{at} f(a) + (1 - \gamma_{at})x^+_{at}. \tag{89.2}$$

The one-period net payoff is

$$U_a(\gamma_{at}) = \gamma_{at}\left[u_i(x^+_{at}) - s(f(a))\right] - (1 - \gamma_{at})\, s(x^+_{at}), \tag{89.3}$$

where $u_i(k) = u_i > 0$ if $k = i$ and $u_i(k) = 0$ otherwise.

Note that an agent is *permitted* to consume a good he does not want; he simply gets zero utility from it while still producing and paying to store $f(a)$.

Not consuming is not free either – it costs $s(x^+_{at})$.

Learning which of these to do is part of the problem.

Note

Kiyotaki and Wright ranked payoff streams by expected discounted utility.

Marimon, McGrattan and Sargent instead assume that each agent cares about his **long-run average** utility.

This matters: as we shall see, it is the reason the accounting system inside a classifier system is built out of running averages.

89.2.1 Two equilibria

Since agents' payoffs depend on what other agents do, the model can have more than one stationary equilibrium.

Kiyotaki and Wright characterize equilibria by a set of probabilities, of which the most useful for us is

$$\pi_{it}^h(k) = \text{probability that a type } i \text{ agent holds good } k \text{ at } t.$$

In the **fundamental equilibrium**, good 1 – the cheapest to store – is the general medium of exchange:

- type 1 agents always hold good 2 (which they produce) and trade it for good 1;
- type 3 agents always hold good 1 and trade it for good 3;
- type 2 agents hold good 1 half the time and good 3 half the time.

So the equilibrium holding probabilities are

	$k=1$	$k=2$	$k=3$
$i=1$	0	1	0
$i=2$	0.5	0	0.5
$i=3$	1	0	0

Type 2 agents accept good 1 even though they never consume it: they use it as money.

In the **speculative equilibrium**, type 1 agents additionally accept good 3 – the *most* expensive good to store – because they expect to trade it away quickly for good 1.

Kiyotaki and Wright show that, in the limit as the discount rate goes to zero, the fundamental equilibrium is the unique stationary equilibrium if

$$s_3 - s_2 > (\pi_1^h(3) - \pi_1^h(2))\tfrac{1}{3}u_1, \tag{89.4}$$

and the speculative equilibrium is the unique one when the inequality is reversed.

Raising u_1 enough therefore flips the model's prediction from fundamental to speculative.

Economy A2 below does exactly that, and it is a good test of whether adaptive agents track the rational-expectations prediction.

89.3 Classifier systems

An agent is not endowed with a strategy.

He is endowed with a **population of candidate rules** and a way of keeping score.

A *classifier* is a string in the trinary alphabet $\{0, 1, \#\}$ split into a **condition** and an **action**, where $\#$ means "don't care".

[Goldberg, 1989] is a book-length treatment of classifier systems and genetic algorithms.

Goods are encoded in binary and conditions in trinary, so with three goods two positions suffice:

code	meaning
1 0	good 1
0 1	good 2
0 0	good 3
0 #	not good 1
# 0	not good 2
# #	any good

An **exchange classifier** is a string of length 7: two positions for own holding, two for the partner's holding, and a final binary digit for the action (1 = propose trade, 0 = refuse).

For example

```
1 0 0 0 1   ->  1     "if I hold good 1 and my partner holds good 3, propose to trade"
1 0 # #     ->  0     "if I hold good 1, refuse to trade with anyone"
```

There are $6 \times 6 \times 2 = 72$ distinct exchange classifiers, which is a *complete enumeration* of all rules definable on the state z_{at}.

A **consumption classifier** is a string of length 4: two positions for the post-trade holding x_{at}^{+} and one action digit (1 = consume).

There are $6 \times 2 = 12$ of these.

89.3.1 The auction

Attached to classifier e at date t is a **strength** $S_e^a(t)$.

Given the state z_{at}, let

$$M_e(z_{at}) = \{e : z_{at} \text{ matches the condition part of } e\}$$

be the set of classifiers whose conditions are satisfied.

The classifier that acts is the strongest matched one,

$$e_t(z_{at}) = \arg\max \{S_e^a(t) : e \in M_e(z_{at})\}, \tag{89.5}$$

and the consumption classifier is chosen the same way from $M_c(z_{at})$.

89.3.2 The bucket brigade

Strengths are updated by a system of internal payments that Holland calls a *bucket brigade*.

Only the classifier that wins an auction changes its strength, so we attach to each classifier a counter $\tau_e^a(t)$ recording the number of auctions it has won up to t, initialized at 1.

A matched classifier e bids the fraction

$$b_1(e) = b_{11} + b_{12}\sigma_e, \qquad \sigma_e = \frac{1}{1 + \text{number of } \#\text{'s in } e}$$

of its strength, and similarly $b_2(c) = b_{21} + b_{22}\sigma_c$ for consumption classifiers.

Because σ_e rises with specificity, specific rules outbid general ones of equal strength.

Payments flow as follows.

- The external payoff $U_a(\gamma_t)$ goes to the winning **consumption** classifier at t.
- The winning consumption classifier at t pays its bid to the winning **exchange** classifier at t, which created the state that gave it a chance to act.
- The winning exchange classifier at t pays its bid to the winning **consumption** classifier at $t-1$, which set up the state z_{at}.

This chain is what transmits the reward for consuming backwards to the trades that made consumption possible.

The resulting laws of motion are

$$S^a_{c,\tau_c(t)} = S^a_{c,\tau_c(t)-1} - \frac{1}{\tau_c(t)-1}\Big[(1+b_2(c))S^a_{c,\tau_c(t)-1} - \sum_e I^a_e(t) b_1(e) S^a_{e,\tau_e(t)} - U_a(\gamma_{ct})\Big] \tag{89.6}$$

$$S^a_{e,\tau_e(t)+1} = S^a_{e,\tau_e(t)} - \frac{1}{\tau_e(t)}\Big[(1+b_1(e))S^a_{e,\tau_e(t)} - \sum_c I^a_c(t) b_2(c) S^a_{c,\tau_c(t)}\Big] \tag{89.7}$$

where $I^a_e(t)$ and $I^a_c(t)$ are indicators for winning the auction at t.

The timing in (89.6) is worth reading carefully, because it is what carries reward across periods.

The consumption classifier updated at date t is the one that won at $t-1$: it collects the external payoff its own decision earned, and it collects the bid $b_1(e)S_e$ from the exchange classifier winning *now*, because that is the classifier whose opportunity to act it created.

The exchange classifier, in turn, is paid within the period by the consumption classifier that follows it.

So each bid travels one step back along the chain

$$\cdots \;\to\; c_{t-1} \;\to\; e_t \;\to\; c_t \;\to\; e_{t+1} \;\to\; \cdots$$

and a payoff collected at consumption seeps backwards, one link per payment, to the trades that made it possible.

An exchange classifier that proposes a trade which is *not* reciprocated is not charged and its counter does not advance: the agent learns nothing from an offer that was refused.

Note

Equations (89.6)-(89.7) make strength a **cumulative average** of past net receipts rather than a cumulative total, which is what Holland's original specification used.

This is the innovation that makes strengths converge.

With $1/\tau$ gains these are stochastic approximation recursions, so any limit point must satisfy

$$\mathbb{E}\Big[(1+b_2(c))S_c - \sum_e I_e b_1(e) S_e - U(\gamma_c)\Big] = 0, \qquad \mathbb{E}\Big[(1+b_1(e))S_e - \sum_c I_c b_2(c) S_c\Big] = 0.$$

Marimon, McGrattan and Sargent define a set of strengths solving these equations to be *stationary*, and a stationary Nash equilibrium to be supported when the rules that win auctions at stationary strengths are exactly the rules that support equilibrium behavior.

All agents of a given type share one classifier system, as in the paper; this economizes on computation at the cost of making all type i agents experiment simultaneously.

89.4 Implementation

We store a population of classifiers as a set of parallel NumPy arrays rather than as a list of objects, which lets us find all matched rules with a single vectorized comparison.

The wildcard # is represented by -1.

```
WILD = -1

# binary codes for goods, one row per good
CODES = {
    3: np.array([[1, 0], [0, 1], [0, 0]]),
    4: np.array([[1, 0], [0, 1], [0, 0], [1, 1]]),          # good 4 = fiat money
    5: np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1],
                 [1, 1, 0], [1, 0, 1]]),
}

# the six conditions expressible with two trits (see the table above)
CONDS_2 = np.array([[1, 0], [0, 1], [0, 0],
                    [0, WILD], [WILD, 0], [WILD, WILD]])


def rule_string(cond, action):
    "Print a classifier the way the paper does."
    body = ''.join('#' if b == WILD else str(int(b)) for b in cond)
    return f"{body} -> {int(action)}"
```

```
class Rules:
    """
    A population of classifiers held as parallel arrays.

    cond[i]     condition part of rule i, entries in {0, 1, WILD}
    action[i]   action part of rule i, in {0, 1}
    strength[i] S_i, a running average of net receipts
    used[i]     the counter tau_i, initialized at 1
    traded[i]   number of times rule i actually executed a trade

    """

    def __init__(self, cond, action, strength=None):
        self.cond = np.asarray(cond, dtype=np.int64)
        self.action = np.asarray(action, dtype=np.int64)
        n = len(self.action)
        self.strength = (np.zeros(n) if strength is None
                         else np.asarray(strength, float).copy())
        self.used = np.ones(n, dtype=np.int64)
        self.traded = np.zeros(n, dtype=np.int64)

    @property
    def n(self):
        return len(self.action)

    @property
    def length(self):
        return self.cond.shape[1]

    def matched(self, state):
```

(continues on next page)

(continued from previous page)

```
        "Boolean mask of rules whose condition is satisfied by state."
        return np.all((self.cond == WILD) | (self.cond == state), axis=1)

    def specificity(self, i):
        "sigma_i = 1 / (1 + number of wildcards)."
        return 1.0 / (1.0 + np.count_nonzero(self.cond[i] == WILD))

    def replace(self, i, cond, action, strength, used=1, traded=0):
        "Overwrite rule i."
        self.cond[i] = cond
        self.action[i] = action
        self.strength[i] = strength
        self.used[i] = used
        self.traded[i] = traded
```

A complete enumeration pairs every condition with both actions.

A random population draws conditions and actions uniformly, which is how the incomplete-enumeration economies start.

```
def complete_rules(pair):
    "All two-trit rules; pair=True for exchange rules, False for consumption rules."
    if pair:
        conds = np.array([np.concatenate([a, b])
                          for a in CONDS_2 for b in CONDS_2])
    else:
        conds = CONDS_2.copy()
    return Rules(np.repeat(conds, 2, axis=0), np.tile([0, 1], len(conds)))


def random_rules(n, length, rng):
    return Rules(rng.integers(-1, 2, size=(n, length)),
                 rng.integers(0, 2, size=n),
                 rng.random(n) * 0.1)
```

The auction (89.5) picks the strongest matched rule, with ties broken by position.

```
def auction(rules, state):
    "Index of the strongest rule matching state, or -1 if none matches."
    idx = np.flatnonzero(rules.matched(state))
    if idx.size == 0:
        return -1, idx
    return idx[np.argmax(rules.strength[idx])], idx
```

Two operators are needed even without a genetic algorithm.

Creation fires when the current state matches no rule at all: a redundant or weak rule is overwritten by one whose condition is exactly the state just observed, with a randomly drawn action.

Diversification fires when every matched rule calls for the same action: a rule with the opposite action is planted so that the alternative can be tried and scored.

Both keep the population size constant.

```
def create(rules, state, rng):
    "No rule matches state, so overwrite the weakest of the most redundant rules."
    groups = {}
    for i in range(rules.n):
        groups.setdefault(tuple(rules.cond[i]), []).append(i)
```

(continues on next page)

(continued from previous page)

```
    biggest = max(groups.values(), key=len)
    group = biggest if len(biggest) > 1 else range(rules.n)
    j = min(group, key=lambda i: rules.strength[i])
    rules.replace(j, state, rng.integers(0, 2), rules.strength.mean())
    return j


def diversify_simple(rules, matches):
    "If every matched rule takes the same action, plant the opposite action."
    if len(set(rules.action[matches])) > 1:
        return
    weak = matches[np.argmin(rules.strength[matches])]
    rules.replace(weak, rules.cond[weak], 1 - rules.action[matches[0]],
                  rules.strength[matches].mean())
```

89.4.1 Describing an economy

An `Economy` collects the parameters of the physical environment together with the settings of the learning algorithm.

The field `method` selects one of the three programs used in the paper:

- `'enumerate'` – start from a complete enumeration of rules with zero strengths, and use no genetic algorithm;
- `'ga3'` – start from random rules and evolve them with single-point crossover and mutation;
- `'ga4'` – start from random rules and evolve them with a *generalizing* crossover, used for the largest economy.

Fiat money, when present, is the last good: it has zero storage cost, gives no utility, and cannot be consumed.

```
@dataclass
class Economy:
    name: str
    produces: np.ndarray            # produces[i] = good produced by type i
    storage_costs: np.ndarray       # one entry per good
    u: float = 100.0                # utility from own consumption good
    n_agents_per_type: int = 50
    method: str = 'enumerate'       # 'enumerate' | 'ga3' | 'ga4'
    n_trade_rules: int = 72
    n_consume_rules: int = 12
    b_trade: tuple = (0.025, 0.025)     # (b11, b12)
    b_consume: tuple = (0.25, 0.25)     # (b21, b22)
    n_fiat: int = 0                 # units of fiat money injected at t = 0
    start: str = 'random'           # initial holdings: 'random' | 'production'
    pcross: float = 0.6
    pmutation: float = 0.01

    @property
    def n_types(self):
        return len(self.produces)

    @property
    def n_goods(self):
        return len(self.storage_costs)

    @property
    def n_agents(self):
        return self.n_types * self.n_agents_per_type
```

(continues on next page)

(continued from previous page)

```
    @property
    def fiat(self):
        return self.n_goods > self.n_types

    @property
    def n_bits(self):
        return CODES[self.n_goods].shape[1]

    def code(self, good):
        return CODES[self.n_goods][good]
```

89.4.2 The agent

An `Agent` represents one *type*: it owns an exchange population and a consumption population, and remembers which consumption rule won last period so that it can be paid by this period's exchange rule.

```
class Agent:

    def __init__(self, i, econ, rng):
        self.i, self.econ, self.rng = i, econ, rng
        if econ.method == 'enumerate':
            self.trade = complete_rules(pair=True)
            self.consume = complete_rules(pair=False)
        else:
            self.trade = random_rules(econ.n_trade_rules, 2 * econ.n_bits, rng)
            self.consume = random_rules(econ.n_consume_rules, econ.n_bits, rng)
        self.pending = None             # last period's consumption rule, awaiting␣
↪settlement

    def decide(self, rules, state, specialize=False):
        "Run the auction on `state`, applying the operators the method calls for."
        win, matches = auction(rules, state)
        if win < 0:                                 # creation
            j = create(rules, state, self.rng)
            return rules.action[j], j
        if self.econ.method == 'enumerate':
            diversify_simple(rules, matches)
        elif self.econ.method == 'ga4':
            diversify_clone(rules, matches, win)
            if specialize:
                specialize_winner(rules, win, self.econ.pmutation, self.rng)
        if self.econ.method != 'ga3':
            win, matches = auction(rules, state)     # the population may have changed
        return rules.action[win], win

    def trade_decision(self, own, partner, specialize=False):
        state = np.concatenate([self.econ.code(own), self.econ.code(partner)])
        return self.decide(self.trade, state, specialize)

    def consume_decision(self, good, specialize=False):
        return self.decide(self.consume, self.econ.code(good), specialize)

    def update(self, e, c, payoff, active):
        """
```

(continues on next page)

(continued from previous page)

```
        The bucket brigade laws of motion for strengths.

        `e` and `c` index the winning exchange and consumption rules and `active`
        records whether the exchange rule's action was actually carried out.

        """
        b11, b12 = self.econ.b_trade
        b21, b22 = self.econ.b_consume
        T, C = self.trade, self.consume
        b1 = b11 + b12 * T.specificity(e)
        b2 = b21 + b22 * C.specificity(c)

        if active:                          # exchange rule: pays b1, receives b2 * S_c
            τ = T.used[e]
            T.used[e] += 1
            T.traded[e] += int(T.action[e] == 1)
            T.strength[e] -= ((1 + b1) * T.strength[e] - b2 * C.strength[c]) / τ

        # Now settle last period's consumption rule.  Its update waits a period
        # because only now is the second of its two receipts known: it collects
        # the payoff its own decision earned *and* the bid of the exchange rule
        # winning today, whose chance to act it created.
        if self.pending is not None:
            p, u_prev = self.pending
            inflow = b1 * T.strength[e] if active else 0.0
            b2p = b21 + b22 * C.specificity(p)
            tau_p = C.used[p]
            C.used[p] += 1
            C.strength[p] -= ((1 + b2p) * C.strength[p] - inflow - u_prev) / tau_p

        self.pending = (c, payoff)
```

89.4.3 The simulation

Each period, all A agents are shuffled into $A/2$ pairs; each pair trades, consumes, and updates strengths; then, in the incomplete-enumeration economies, the genetic operators run.

A small proportional tax on exchange strengths, present in the authors' code, keeps strengths from locking in permanently.

```
class Simulation:

    def __init__(self, econ, seed=0):
        self.econ = econ
        self.rng = np.random.default_rng(seed)
        self.agents = [Agent(i, econ, self.rng) for i in range(econ.n_types)]
        self.types = np.repeat(np.arange(econ.n_types), econ.n_agents_per_type)

        if econ.start == 'random':
            self.holdings = self.rng.integers(0, econ.n_types, size=econ.n_agents)
        else:
            self.holdings = econ.produces[self.types].copy()
        if econ.n_fiat:
            who = self.rng.choice(econ.n_agents, size=econ.n_fiat, replace=False)
            self.holdings[who] = econ.n_goods - 1
```

(continues on next page)

(continued from previous page)

```
        self.hold_hist, self.exch_hist, self.cons_hist = [], [], []
        self.trades, self.eaten = [], []

    def run(self, T, verbose=False):
        econ, rng = self.econ, self.rng
        evolving = econ.method != 'enumerate'

        if evolving:
            # the genetic algorithm fires on even dates with probability 1/sqrt(t/2)
            p = 1.0 / np.sqrt(np.arange(1, T // 2 + 1))
            even = np.arange(1, T, 2)
            ga_dates = np.zeros(T, dtype=bool)
            ga_dates[even] = p[:len(even)] > rng.random(len(even))
            spec_dates = np.zeros(T, dtype=bool)
            spec_dates[even] = p[:len(even)] > rng.random(len(even))

        for t in range(1, T + 1):
            spec = evolving and econ.method == 'ga4' and spec_dates[t - 1]
            n_trades = n_eaten = 0
            exch = np.zeros((econ.n_types, econ.n_goods, econ.n_goods))
            cons = np.zeros((econ.n_types, econ.n_goods, 2))

            order = rng.permutation(econ.n_agents)
            for k in range(econ.n_agents // 2):
                a, b = order[2 * k], order[2 * k + 1]
                ia, ib = self.types[a], self.types[b]
                ga, gb = self.holdings[a], self.holdings[b]
                A, B = self.agents[ia], self.agents[ib]

                # --- exchange ---
                la, ea = A.trade_decision(ga, gb, spec)
                lb, eb = B.trade_decision(gb, ga, spec)
                swap = (la == 1) and (lb == 1)
                if swap:
                    self.holdings[a], self.holdings[b] = gb, ga
                    n_trades += 1
                    exch[ia, ga, gb] += 1
                    exch[ib, gb, ga] += 1

                # --- consumption ---
                pa, pb = self.holdings[a], self.holdings[b]
                ca, wa = A.consume_decision(pa, spec)
                cb, wb = B.consume_decision(pb, spec)
                cons[ia, pa, 0] += 1
                cons[ib, pb, 0] += 1

                ua = self.consume_and_produce(a, ia, pa, ca)
                if ca == 1:
                    cons[ia, pa, 1] += 1
                    n_eaten += pa == ia
                ub = self.consume_and_produce(b, ib, pb, cb)
                if cb == 1:
                    cons[ib, pb, 1] += 1
                    n_eaten += pb == ib

                # --- accounting ---
                A.update(ea, wa, ua, swap or la == 0)
```

(continues on next page)

(continued from previous page)

```
                B.update(eb, wb, ub, swap or lb == 0)

            if evolving:
                if ga_dates[t - 1]:
                    self.evolve()
                if econ.method == 'ga3':
                    for A in self.agents:
                        specialize_all(A.trade, t, rng)
                        specialize_all(A.consume, t, rng)
            self.tax()

            self.hold_hist.append(self.distribution())
            self.exch_hist.append(exch)
            self.cons_hist.append(cons)
            self.trades.append(n_trades)
            self.eaten.append(n_eaten)
            if verbose and t % max(1, T // 10) == 0:
                print(f"  period {t:5d}:  trades = {n_trades:3d},"
                      f"  consumptions = {n_eaten:3d}")

    def consume_and_produce(self, a, i, good, action):
        """
        Carry out the consumption decision and return the external payoff.

        Consuming fiat money is not allowed.  If the agent does consume, he
        immediately produces his own good, so this updates his holding.

        """
        econ = self.econ
        if action == 1 and not (econ.fiat and good == econ.n_goods - 1):
            new = econ.produces[i]
            self.holdings[a] = new
            u = econ.u if good == i else 0.0
            return u - econ.storage_costs[new]
        return -econ.storage_costs[good]

    def distribution(self):
        "The matrix of holding frequencies pi^h_it(k)."
        econ = self.econ
        d = np.zeros((econ.n_types, econ.n_goods))
        for i in range(econ.n_types):
            h = self.holdings[self.types == i]
            for k in range(econ.n_goods):
                d[i, k] = np.mean(h == k)
        return d

    def tax(self):
        if self.econ.method == 'ga4':
            for A in self.agents:
                T, C = A.trade, A.consume
                P = np.where(T.action == 1, T.traded, T.used) + 1
                T.strength -= (T.strength + 1.0) / P
                C.strength -= (C.strength + 1.0) / (C.used + 1)
        else:
            for A in self.agents:
                A.trade.strength -= 1e-4 * np.abs(A.trade.strength)
```

(continues on next page)

(continued from previous page)

```
    def pick_types(self):
        "Send one type to the genetic algorithm, a second and a third each w.p. 0.33."
        rng, n = self.rng, self.econ.n_types
        chosen = [int(rng.integers(n))]
        rest = [i for i in range(n) if i not in chosen]
        while rest and len(chosen) < 3 and rng.random() < 0.33:
            pick = rest[int(rng.integers(len(rest)))]
            chosen.append(pick)
            rest.remove(pick)
        return chosen

    def evolve(self):
        econ = self.econ
        gen = econ.method == 'ga4'
        for i in self.pick_types():
            genetic_algorithm(self.agents[i].trade, self.rng, generalize=gen,
                              pcross=econ.pcross, pmutation=econ.pmutation,
                              crowd_factor=8)
        for i in self.pick_types():
            genetic_algorithm(self.agents[i].consume, self.rng, generalize=gen,
                              pcross=econ.pcross, pmutation=econ.pmutation,
                              crowd_factor=4)
```

Note

`Agent` and `Simulation` refer to four functions – `genetic_algorithm`, `specialize_all`, `diversify_clone` and `specialize_winner` – that belong to the genetic algorithm and are therefore deferred to *The genetic algorithm* below, where they can be motivated by the problem they solve.

Deferring them costs nothing.

Python looks names up when a function runs rather than when it is defined, and the complete-enumeration economies we study first never call any of the four.

Readers who prefer to see the machinery before it is used can run that section's cells first.

89.4.4 Reporting

The following helpers put simulation output into the format of the paper's tables.

We follow the paper in reporting ten-period moving averages.

```
def good_names(econ):
    names = [f"good {k+1}" for k in range(econ.n_types)]
    return names + ["fiat"] if econ.fiat else names


def type_names(econ):
    return [f"type {i+1}" for i in range(econ.n_types)]


def holdings(sim, t=None, window=10):
    r"Table of $\pi^h_{it}(j)$, averaged over the `window` periods ending at `t`."
    h = np.array(sim.hold_hist)
```

(continues on next page)

(continued from previous page)

```
    t = len(h) if t is None else t
    d = h[max(0, t - window):t].mean(axis=0)
    return pd.DataFrame(d, index=type_names(sim.econ),
                        columns=good_names(sim.econ)).round(3)


def exchanges(sim, t=None, window=10):
    r"""
    Table of $\pi^e_{it}(jk)$: the frequency with which a type $i$ agent holds
    good $j$, meets an agent holding good $k$, and trades.  Row $j$ of column
    $i$ holds the triple over $k$.
    """
    econ = sim.econ
    e = np.array(sim.exch_hist)
    t = len(e) if t is None else t
    f = e[max(0, t - window):t].mean(axis=0) / econ.n_agents_per_type
    cols = {type_names(econ)[i]:
            ["(" + ", ".join(f"{f[i, j, k]:.2f}" for k in range(econ.n_goods)) + ")"
             for j in range(econ.n_goods)]
            for i in range(econ.n_types)}
    return pd.DataFrame(cols, index=good_names(econ)).T


def winning_actions(sim):
    r"""
    Table of $\tilde\pi^e_{it}(jk|j)$: the action chosen by the winning exchange
    rule in each state, whether or not that state is ever visited.
    """
    econ = sim.econ
    cols = {}
    for i, A in enumerate(sim.agents):
        col = []
        for j in range(econ.n_goods):
            acts = []
            for k in range(econ.n_goods):
                w, _ = auction(A.trade, np.concatenate([econ.code(j), econ.code(k)]))
                acts.append('-' if w < 0 else str(int(A.trade.action[w])))
            col.append("(" + ",".join(acts) + ")")
        cols[type_names(econ)[i]] = col
    return pd.DataFrame(cols, index=good_names(econ)).T


def consume_actions(sim):
    r"Table of the winning consumption action for each post-trade holding."
    econ = sim.econ
    cols = {}
    for i, A in enumerate(sim.agents):
        col = []
        for j in range(econ.n_goods):
            w, _ = auction(A.consume, econ.code(j))
            col.append('-' if w < 0 else int(A.consume.action[w]))
        cols[type_names(econ)[i]] = col
    return pd.DataFrame(cols, index=good_names(econ)).T


def strongest(rules, n=5):
    "The n highest-strength classifiers in a population."
```

(continues on next page)

(continued from previous page)

```
    order = np.argsort(-rules.strength)[:n]
    return pd.DataFrame({
        'classifier': [rule_string(rules.cond[i], rules.action[i]) for i in order],
        'strength': rules.strength[order].round(2),
        'times used': rules.used[order]})
```

Two figures: the time path of holdings, which corresponds to the paper's figures 6-9, and a diagram of the exchange pattern that the system discovers, which corresponds to its figures 2, 4, 9 and 11.

```
def plot_holdings(sim, title=None):
    econ = sim.econ
    h = np.array(sim.hold_hist)
    names = good_names(econ)
    fig, axes = plt.subplots(1, econ.n_types,
                             figsize=(3.2 * econ.n_types, 3.2), sharey=True)
    axes = np.atleast_1d(axes)
    for i, ax in enumerate(axes):
        for k in range(econ.n_goods):
            ax.plot(h[:, i, k], lw=1.0, label=names[k])
        ax.set_title(f"type {i+1}")
        ax.set_xlabel("$t$")
        ax.set_ylim(-0.03, 1.03)
    axes[0].set_ylabel(r"$\pi^h_{it}(j)$")
    axes[-1].legend(frameon=False, fontsize=8, loc='center right')
    if title:
        fig.suptitle(title)
    plt.tight_layout()
    plt.show()


def plot_flows(sim, window=100, cutoff=0.02, title=None):
    """
    One panel per type.  An arrow from good j to good k means that agents of that
    type give up j and receive k in trade; its width is the frequency with which
    that exchange occurs over the last `window` periods.
    """
    econ = sim.econ
    f = np.array(sim.exch_hist)[-window:].mean(axis=0) / econ.n_agents_per_type
    names, n = good_names(econ), econ.n_goods
    ang = np.pi / 2 + 2 * np.pi * np.arange(n) / n
    xy = np.column_stack([np.cos(ang), np.sin(ang)])

    fig, axes = plt.subplots(1, econ.n_types, figsize=(2.9 * econ.n_types, 3.1))
    axes = np.atleast_1d(axes)
    for i, ax in enumerate(axes):
        for k in range(n):
            ax.plot(*xy[k], 'o', ms=22, mfc='white', mec='black', zorder=2)
            ax.annotate(names[k].replace(' ', '\n'), xy[k], ha='center',
                        va='center', fontsize=6.5, zorder=3)
        for j in range(n):
            for k in range(n):
                if j == k or f[i, j, k] < cutoff:
                    continue
                a, b = xy[j], xy[k]
                d = b - a
                ax.annotate("", xy=b - 0.24 * d, xytext=a + 0.24 * d, zorder=1,
                            arrowprops=dict(arrowstyle="-|>", color="C0",
```

(continues on next page)

(continued from previous page)

```
                                            lw=1 + 8 * f[i, j, k], alpha=0.7,
                                            connectionstyle="arc3,rad=0.15"))
        ax.set_title(f"type {i+1}", fontsize=10)
        ax.set_xlim(-1.45, 1.45)
        ax.set_ylim(-1.45, 1.45)
        ax.set_aspect('equal')
        ax.axis('off')
    if title:
        fig.suptitle(title)
    plt.tight_layout()
    plt.show()
```

89.5 Economy A1.1: does a medium of exchange emerge?

Our first economy is the Wicksell triangle with

$$s_1 = 0.1, \quad s_2 = 1, \quad s_3 = 20, \quad u_i = 100,$$

fifty agents of each type, and a complete enumeration of the 72 exchange and 12 consumption classifiers, all with strength zero.

Since all strengths start equal, the initial auction winners are effectively arbitrary: agents begin with no idea what to do.

Condition (89.4) holds comfortably here, so the fundamental equilibrium is the Kiyotaki-Wright prediction.

```
economy_a11 = Economy(
    name='A1.1',
    produces=np.array([1, 2, 0]),               # type 1 -> good 2, etc. (0-indexed)
    storage_costs=np.array([0.1, 1.0, 20.0]),
    u=100.0,
    method='enumerate',
)

sim_a11 = Simulation(economy_a11, seed=42)
sim_a11.run(1000, verbose=True)
```

```
  period   100:  trades =  19,  consumptions =  27
  period   200:  trades =  29,  consumptions =  41
  period   300:  trades =  21,  consumptions =  26
  period   400:  trades =  19,  consumptions =  22
  period   500:  trades =  17,  consumptions =  24
  period   600:  trades =  19,  consumptions =  27
  period   700:  trades =  15,  consumptions =  19
  period   800:  trades =  16,  consumptions =  18
  period   900:  trades =  16,  consumptions =  21
  period  1000:  trades =  22,  consumptions =  29
```

Here are holdings at $t = 500$ and $t = 1000$.

```
holdings(sim_a11, t=500)
```

```
        good 1  good 2  good 3
type 1   0.000     1.0   0.000
type 2   0.522     0.0   0.478
type 3   1.000     0.0   0.000
```

```
holdings(sim_a11)
```

```
        good 1  good 2  good 3
type 1    0.00     1.0    0.00
type 2    0.53     0.0    0.47
type 3    1.00     0.0    0.00
```

Compare these with the fundamental equilibrium tabulated above: type 1 holds good 2 with probability one, type 3 holds good 1 with probability one, and type 2 splits evenly between goods 1 and 3.

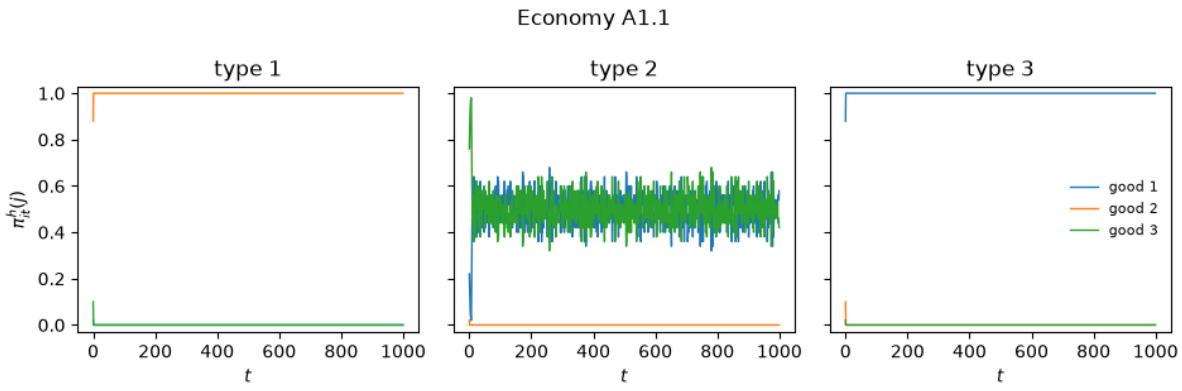
Rather than assert the comparison, let us compute it.

```
fundamental = np.array([[0.0, 1.0, 0.0],
                        [0.5, 0.0, 0.5],
                        [1.0, 0.0, 0.0]])
paper_a11 = np.array([[0.0, 1.0, 0.0],          # the paper's table at t = 1000
                      [0.506, 0.0, 0.494],
                      [1.0, 0.0, 0.0]])

simulated = holdings(sim_a11).to_numpy()
print(f"max |simulated - fundamental equilibrium| = "
      f"{np.abs(simulated - fundamental).max():.3f}")
print(f"max |simulated - paper's table|           = "
      f"{np.abs(simulated - paper_a11).max():.3f}")
```

```
max |simulated - fundamental equilibrium| = 0.030
max |simulated - paper's table|           = 0.024
```

The convergence is essentially immediate, as the time paths show.

```
plot_holdings(sim_a11, title="Economy A1.1")
```



Only type 2 agents carry any randomness, and the reason is instructive: they are the agents who use good 1 as money, so which good they are holding depends on where they are in the cycle "acquire money, spend money".

Now let us look at the trades themselves.

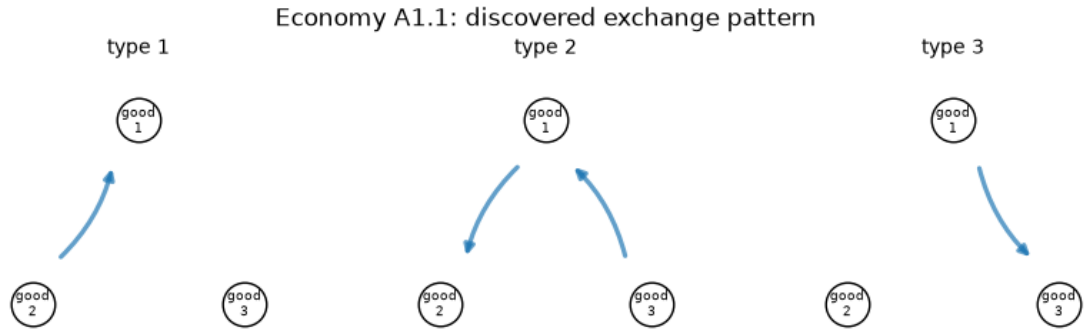
The entry in row i, column j is the triple over k of the frequency with which a type i agent holds j, meets someone holding k, and trades.

```
exchanges(sim_a11)
```

```
                    good 1              good 2              good 3
type 1   (0.00, 0.00, 0.00)  (0.15, 0.00, 0.00)  (0.00, 0.00, 0.00)
type 2   (0.09, 0.15, 0.00)  (0.00, 0.00, 0.00)  (0.16, 0.00, 0.00)
type 3   (0.00, 0.00, 0.16)  (0.00, 0.00, 0.00)  (0.00, 0.00, 0.00)
```

The exchange pattern is easier to read as a picture.

```
plot_flows(sim_a11, title="Economy A1.1: discovered exchange pattern")
```



This is exactly the fundamental equilibrium triangle: type 1 gives up good 2 for good 1, type 3 gives up good 1 for good 3, and type 2 runs both legs, giving up good 3 for good 1 and later good 1 for good 2.

Good 1 circulates as money.

We can also ask what the winning rules would do in states that are *never* visited, which is what Kiyotaki and Wright's strategies specify and what the paper reports.

```
winning_actions(sim_a11)
```

```
         good 1   good 2   good 3
type 1   (0,0,0)  (1,0,0)  (1,0,0)
type 2   (1,1,0)  (0,0,0)  (1,1,0)
type 3   (0,0,1)  (0,0,0)  (0,0,0)
```

Reading row "type 1", the entry in column "good 2" is the triple $(\tilde{\pi}_1^e(21|2), \tilde{\pi}_1^e(22|2), \tilde{\pi}_1^e(23|2))$.

A type 1 agent holding good 2 accepts good 1 and refuses good 3: he does not speculate.

Finally, we can look inside a classifier system and read off the rules that won the competition.

```
strongest(sim_a11.agents[0].trade)
```

```
  classifier  strength  times used
0  0110 -> 1     30.04        8343
1  1010 -> 0     29.27           5
2  1001 -> 0     27.54           8
3  1000 -> 0     16.80           3
4  1001 -> 1      0.00           1
```

```
strongest(sim_a11.agents[0].consume)
```

```
  classifier  strength  times used
0    10 -> 1     66.20        8354
1    10 -> 0     -0.10           2
2    0# -> 1     -0.55       41625
3    #0 -> 1     -0.68           6
4    00 -> 1     -0.75           3
```

The strongest exchange rule for a type 1 agent is `0110 -> 1`, that is, "if I am holding good 2 (`01`) and my partner holds good 1 (`10`), trade" – and it is used thousands of times, while the runners-up are essentially never used.

The strongest consumption rule is `10 -> 1`: "if I am holding good 1, eat it".

These are exactly the classifiers $e^1_{2,1,1}$ and $c^1_{1,1}$ that the paper shows support the fundamental equilibrium.

89.6 Economy A2.1: when theory predicts speculation

Economy A2 differs from A1 only in that $u_i = 500$ instead of 100.

That is enough to violate the Kiyotaki-Wright inequality (89.4), so for patient agents the unique stationary rational-expectations equilibrium is now the **speculative** one, in which type 1 agents accept good 3 in the expectation of trading it for good 1.

Do our adaptive agents find it?

```
economy_a21 = Economy(
    name='A2.1',
    produces=np.array([1, 2, 0]),
    storage_costs=np.array([0.1, 1.0, 20.0]),
    u=500.0,
    method='enumerate',
)

sim_a21 = Simulation(economy_a21, seed=42)
sim_a21.run(1000)

holdings(sim_a21)
```

```
        good 1  good 2  good 3
type 1    0.00     1.0    0.00
type 2    0.53     0.0    0.47
type 3    1.00     0.0    0.00
```

Here is the speculative equilibrium that Kiyotaki and Wright's theory predicts for these parameters, for comparison.

```
speculative = pd.DataFrame([[0, 0.707, 0.293],
                            [0.586, 0, 0.414],
                            [1, 0, 0]],
                           index=type_names(economy_a21),
                           columns=good_names(economy_a21))
speculative
```

```
        good 1  good 2  good 3
type 1   0.000   0.707   0.293
type 2   0.586   0.000   0.414
type 3   1.000   0.000   0.000
```

The answer is no.

The simulated holdings are those of the *fundamental* equilibrium: type 1 agents hold good 2 essentially always and never accumulate good 3, whereas speculation requires them to hold good 3 nearly thirty percent of the time.

It is worth looking at why.

```
winning_actions(sim_a21)
```

```
         good 1   good 2   good 3
type 1   (0,0,0)  (1,0,0)  (0,1,0)
type 2   (1,1,0)  (0,0,0)  (1,1,0)
type 3   (0,0,1)  (0,0,0)  (0,0,0)
```

Reading row "type 1", column "good 2", the winning exchange rule refuses good 3: type 1 agents will not make the speculative swap.

Now look at what they would do with good 3 if they had it.

```
consume_actions(sim_a21)
```

```
        good 1  good 2  good 3
type 1       1       1       1
type 2       0       1       0
type 3       1       1       1
```

```
strongest(sim_a21.agents[0].consume, n=4)
```

```
  classifier  strength  times used
0    10 -> 1    333.70        8161
1    ## -> 1      0.17       41809
2    10 -> 0     -0.10           2
3    #0 -> 1     -0.66          17
```

This is the paper's diagnosis, visible in the output.

One consumption rule, `10 -> 1`, is specific and enormously strong: "if I hold good 1, eat it".

The rule that decides everything else is `## -> 1` – *eat whatever you are holding* – a maximally general rule with two wildcards and a strength barely above zero, yet it is the strongest rule matching any state other than "holding good 1".

So a type 1 agent who acquired good 3 would immediately consume it for zero utility rather than carry it and trade it for good 1.

As the paper puts it, the winning consumption classifiers of type 1 agents are too general – they have too many #'s to distinguish among stored goods – so type 1 agents overconsume good 3, and the information that would make speculation pay never reaches the exchange classifiers.

The authors' diagnosis is worth quoting:

> *Patience requires experience.* The transfer system inside the classifier system is designed to converge to a set of long-run average strengths. In the limit, the artificially intelligent agents should behave as long-run average payoff maximizers… It takes time, however, for optimal rules to achieve the desired strengths. The behavior of our artificially intelligent agents can be very myopic at the beginning… The present algorithm seems defective in that it has too little experimentation to support the speculative equilibrium even in the long simulations we have run.

Early on, before any rule has accumulated a meaningful average, agents behave myopically, and a myopic agent never accepts an expensive good.

By the time strengths have settled, everyone else has already stopped speculating, so speculation no longer pays.

The equilibrium selected is a consequence of the *learning dynamics*, not of the payoffs alone.

89.7 Economy B.1: a different production pattern

Economy B changes the production technology – type 1 produces good 3, type 2 produces good 1, type 3 produces good 2 – and compresses storage costs to $s = (1, 4, 9)$ with $u_i = 100$.

Both a fundamental and a speculative equilibrium exist.

The paper reports something striking: at $t = 500$ the economy looks speculative, but by $t = 1000$ it has moved to the fundamental equilibrium.

```
economy_b1 = Economy(
    name='B.1',
    produces=np.array([2, 0, 1]),               # type 1 -> good 3, type 2 -> good 1,␣
 ↪...
    storage_costs=np.array([1.0, 4.0, 9.0]),
    u=100.0,
    method='enumerate',
    b_trade=(0.25, 0.25),
)

sim_b1 = Simulation(economy_b1, seed=42)
sim_b1.run(1000)
```

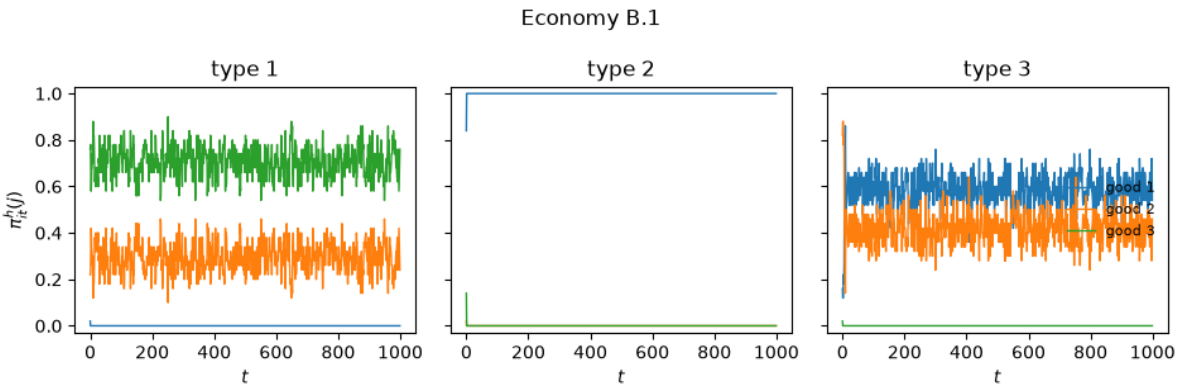
```
holdings(sim_b1, t=500)
```

	good 1	good 2	good 3
type 1	0.00	0.274	0.726
type 2	1.00	0.000	0.000
type 3	0.56	0.440	0.000

```
holdings(sim_b1)
```

	good 1	good 2	good 3
type 1	0.000	0.324	0.676
type 2	1.000	0.000	0.000
type 3	0.626	0.374	0.000

```
plot_holdings(sim_b1, title="Economy B.1")
```




```
paper_b1 = np.array([[0.0, 0.28, 0.72],          # the paper's table at t = 1000
                     [0.994, 0.0, 0.006],
                     [0.526, 0.474, 0.0]])

print(f"max |simulated - paper's table| = "
      f"{np.abs(holdings(sim_b1).to_numpy() - paper_b1).max():.3f}")
```

```
max |simulated - paper's table| = 0.100
```

The end state reproduces the paper's qualitative pattern, with the largest single discrepancy about a tenth.

Type 2 agents hold good 1 always, and type 3 agents split their holdings between good 1 and their own production good 2.

The paper reports one further feature that our run does not reproduce.

In the authors' simulation, type 3 agents held good 2 with probability one at $t = 500$ – refusing the cheaper good 1, which is the speculative pattern – and only later shifted into good 1:

> Economy B.1 displays an interesting pattern of evolution. At iteration 500 the distribution of holdings and, especially, the trading patterns correspond to the speculative equilibrium. However, the economy moves away from this state and by iteration 1000 has practically converged to the fundamental equilibrium.

In our implementation the shift happens within the first fifty periods, so the speculative phase is over before $t = 500$.

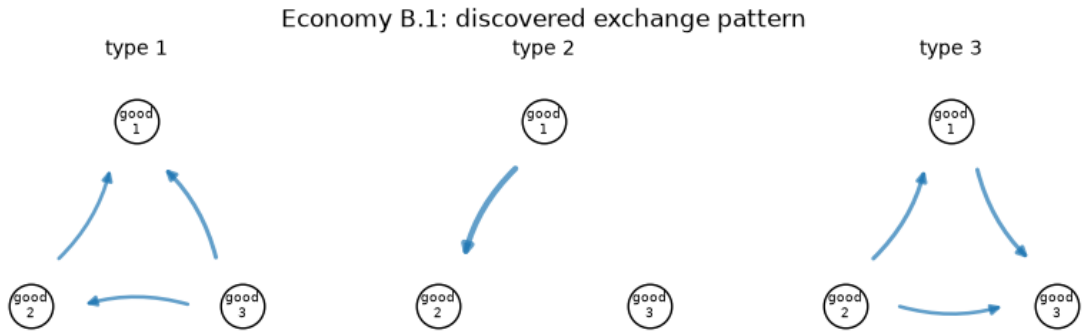
The transient is evidently sensitive to details of the implementation in a way that the limit is not.

The economics of the authors' observation is nevertheless worth stating, because it bears on how we should read Economy A2.

There, one might suspect that the fundamental equilibrium was selected merely because myopic agents refuse expensive goods from the very beginning.

In Economy B the system *starts* in the speculative pattern and *moves away from it* as strengths accumulate, which suggests that the selection of the fundamental equilibrium is something the learning dynamics do, not just an artifact of where they start.

```
plot_flows(sim_b1, title="Economy B.1: discovered exchange pattern")
```



89.8 The genetic algorithm

Complete enumeration is only feasible because the Kiyotaki-Wright state space is tiny.

In any larger problem the list of all conceivable rules is far too long to carry around, and an agent must instead work with a limited population of rules that is continually revised.

Marimon, McGrattan and Sargent add four operations for this case.

Creation and **diversification** we have already implemented; they handle unforeseen states and guarantee that both actions get tried.

For the five-good economy we shall use a variant of diversification that clones the winning rule with the opposite action, thereby preserving the winner's level of generality rather than manufacturing a fully specific rule.

```
def diversify_clone(rules, matches, winner, ufitness=0.5):
    "Copy the winner with the opposite action over a rarely used rule."
    if len(set(rules.action[matches])) > 1:
        return
    losers = np.flatnonzero(rules.used / (rules.used.max() + 1) < ufitness)
    if losers.size == 0:
        return
    j = losers[np.argmin(rules.strength[losers])]
    rules.replace(j, rules.cond[winner].copy(), 1 - rules.action[winner],
                  rules.strength[winner], used=rules.used[winner])
```

Specialization turns wildcards into specific bits, so that a rule which has been serving several states can split off a sharper version of itself.

It is called with a probability $f_s(t) = 1/(2\sqrt{t})$ that declines over time: experimentation is cheap early and expensive late.

```
def specialize_all(rules, t, rng):
    "Replace each wildcard by a bit with probability 1 / (2 sqrt(t))."
    hit = (rules.cond == WILD) & (rng.random(rules.cond.shape)
                                  < 1.0 / (2.0 * np.sqrt(t)))
    if hit.any():
        rules.cond[hit] = rng.integers(0, 2, size=hit.sum())


def specialize_winner(rules, winner, pmutation, rng, ufitness=0.5):
```

(continues on next page)

(continued from previous page)

```
    "Plant a sharpened copy of a heavily used winner over a rarely used rule."
    cond = rules.cond[winner]
    if not np.any(cond == WILD):
        return
    if rules.used[winner] / (rules.used.max() + 1) <= ufitness:
        return
    pick = (cond == WILD) & (rng.random(cond.shape) < pmutation)
    losers = np.flatnonzero(rules.used / (rules.used.max() + 1) < ufitness)
    if not pick.any() or losers.size == 0:
        return
    j = losers[np.argmin(rules.strength[losers])]
    new = cond.copy()
    new[pick] = rng.integers(0, 2, size=pick.sum())
    rules.replace(j, new, rules.action[winner], rules.strength[winner],
                  used=rules.used[winner])
```

Generalization is the genetic algorithm proper.

Rules that are weak or rarely used become candidates for replacement.

Parents are drawn in two stages – first a subset weighted by how often rules have been used, then a roulette wheel on strength within that subset – so that a rule must be both successful and *relevant* to reproduce.

Two children are formed by crossover and, in the `'ga3'` variant, mutation.

Each child then displaces the rule it most closely resembles among the replaceable ones, a device known as *crowding* that preserves diversity by making children compete against their own kind.

The `'ga4'` variant replaces single-point crossover with a *generalizing* crossover: inside a randomly drawn interval, positions where the parents disagree become wildcards.

This is the operator described in section 6 of the paper and illustrated in its figure 5, and it manufactures general rules rather than recombining specific ones.

```
def roulette(weights, rng):
    total = weights.sum()
    if total <= 0:
        return int(rng.integers(0, len(weights)))
    return int(np.searchsorted(np.cumsum(weights), rng.random() * total))


def crowding_victim(rules, cond, action, cankill, rng, crowd_factor, crowd_subpop):
    "De Jong crowding: the child displaces the replaceable rule it resembles most."
    size = max(1, int(crowd_subpop * len(cankill)))
    best, best_sim = cankill[0], -1
    for _ in range(crowd_factor):
        pool = (cankill if size >= len(cankill)
                else list(rng.choice(cankill, size=size, replace=False)))
        cand = min(pool, key=lambda i: rules.strength[i])
        sim = (np.count_nonzero(cond == rules.cond[cand])
               + (action != rules.action[cand]))
        if sim > best_sim:
            best, best_sim = cand, sim
    return best


def genetic_algorithm(rules, rng, generalize=False, pcross=0.6, pmutation=0.01,
                      propselect=0.2, propused=0.7, crowd_factor=8,
```

(continues on next page)

(continued from previous page)

```
                      crowd_subpop=0.5, uratio=(0.0, 0.2)):
    n, L = rules.n, rules.length
    if n < 4:
        return

    # rules that are weak or seldom used may be replaced
    max_used = max(rules.used.max() + (1 if generalize else 0), 1)
    cankill = list(np.flatnonzero((rules.strength < uratio[0]) |
                                  (rules.used / max_used < uratio[1])))
    if not cankill:
        return

    fitness = rules.strength - min(rules.strength.min(), 0.0) + 1e-6
    n_pairs = min(max(1, round(propselect * n * 0.5)), (len(cankill) + 1) // 2)
    n_called = int(propused * n)

    for _ in range(n_pairs):
        if not cankill:
            break

        # stage 1: pre-select a pool with probability proportional to usage
        if n_called < n:
            avail = rules.used.astype(float) + 1.0
            pool = []
            for _ in range(n_called):
                if avail.sum() <= 0:
                    break
                k = roulette(avail, rng)
                pool.append(k)
                avail[k] = 0.0
            if len(pool) < 2:
                pool = list(range(n))
        else:
            pool = list(range(n))
        pool = np.array(pool)

        # stage 2: roulette wheel on fitness within the pool
        mum, dad = pool[roulette(fitness[pool], rng)], pool[roulette(fitness[pool],␣
↪rng)]
        kids = [rules.cond[mum].copy(), rules.cond[dad].copy()]
        acts = [rules.action[mum], rules.action[dad]]
        avg = 0.5 * (rules.strength[mum] + rules.strength[dad])

        if generalize:
            # two-point crossover in which disagreements become wildcards
            lo, hi = np.sort(rng.integers(0, L + 1, size=2))
            inside = rng.random() > 0.5
            region = (np.arange(lo, hi) if inside else
                      np.concatenate([np.arange(0, lo), np.arange(hi, L)]))
            for j in region:
                a, b = kids[0][j], kids[1][j]
                if a >= 0 and b >= 0 and a != b:
                    kids[0][j] = kids[1][j] = WILD
        else:
            # single-point crossover with ternary mutation
            jc = 1 + int((L - 1) * rng.random()) if rng.random() < pcross else L
            kids = [np.concatenate([rules.cond[mum][:jc], rules.cond[dad][jc:]]),
```

(continues on next page)

(continued from previous page)

```
                    np.concatenate([rules.cond[dad][:jc], rules.cond[mum][jc:]])]
            for k in range(2):
                flip = rng.random(L) < pmutation
                if flip.any():
                    shift = rng.integers(1, 3, size=flip.sum())
                    kids[k][flip] = ((kids[k][flip] + 1 + shift) % 3) - 1
                if rng.random() < pmutation:
                    acts[k] = 1 - acts[k]

        for k, parent in zip(range(2), (mum, dad)):
            if not cankill:
                break
            j = crowding_victim(rules, kids[k], acts[k], cankill, rng,
                                crowd_factor, crowd_subpop)
            rules.replace(j, kids[k], acts[k], avg,
                          used=rules.used[parent], traded=rules.traded[parent])
            cankill.remove(j)
```

89.9 Economy A1.2: learning from random rules

Economy A1.2 has the same parameters as A1.1, but agents now begin with 72 exchange rules and 12 consumption rules drawn *at random*.

Most of these rules are nonsense, and there is no guarantee that the population even contains the rules needed to support an equilibrium.

The genetic operators must manufacture them.

```
economy_a12 = Economy(
    name='A1.2',
    produces=np.array([1, 2, 0]),
    storage_costs=np.array([0.1, 1.0, 20.0]),
    u=100.0,
    method='ga3',
)

sim_a12 = Simulation(economy_a12, seed=2)
sim_a12.run(2000, verbose=True)
```

```
  period   200:  trades =  33,  consumptions =  14
  period   400:  trades =   4,  consumptions =   0
  period   600:  trades =  37,  consumptions =  18
  period   800:  trades =  44,  consumptions =   0
  period  1000:  trades =  30,  consumptions =  15
  period  1200:  trades =  32,  consumptions =  17
  period  1400:  trades =  31,  consumptions =  14
  period  1600:  trades =  48,  consumptions =  17
  period  1800:  trades =  40,  consumptions =  41
  period  2000:  trades =  63,  consumptions =  41
```

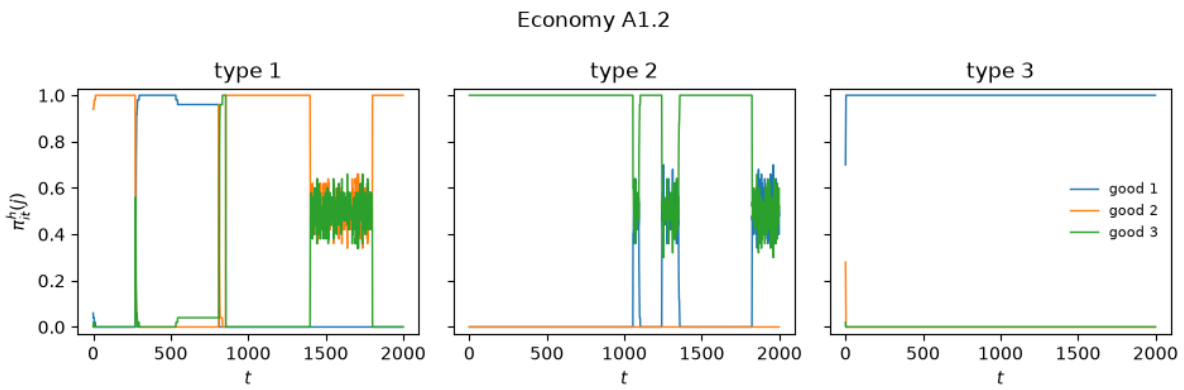
```
holdings(sim_a12, t=1000)
```

```
        good 1  good 2  good 3
type 1     0.0     1.0     0.0
type 2     0.0     0.0     1.0
type 3     1.0     0.0     0.0
```

```
holdings(sim_a12)
```

```
        good 1  good 2  good 3
type 1   0.000     1.0   0.000
type 2   0.478     0.0   0.522
type 3   1.000     0.0   0.000
```

```
plot_holdings(sim_a12, title="Economy A1.2")
```



The fundamental equilibrium is reached again, though it takes noticeably longer and the early transition is much less orderly than under complete enumeration.

Let us see which rules survived.

```
strongest(sim_a12.agents[0].trade, n=6)
```

```
  classifier  strength  times used
0  0110 -> 1     30.20       15642
1  1010 -> 1     28.19          32
2  0010 -> 1     27.74        7933
3  1010 -> 1     27.41          32
4  0110 -> 1     27.41        6422
5  1010 -> 1     27.05          32
```

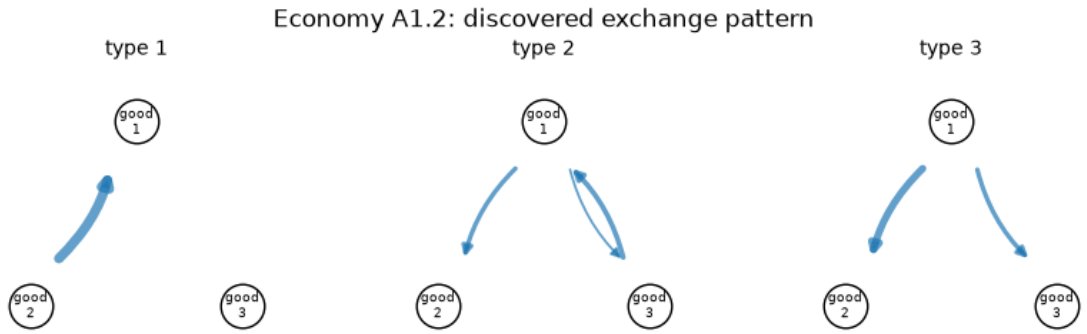
The population has converged onto near-copies of a single rule.

`0110 -> 1` – "holding good 2, meeting good 1, trade" – is exactly the rule that complete enumeration selected in Economy A1.1, but here the genetic algorithm had to build it, and several copies of it now occupy the population.

The other entries are its offspring: `0111 -> 1` has the same own-holding condition but a partner condition of `11`, which is not the code of any of the three goods, so that rule can never fire.

It carries a high strength and a large usage count only because children inherit both from their parents.

```
plot_flows(sim_a12, title="Economy A1.2: discovered exchange pattern")
```



Warning

Convergence is not guaranteed run by run.

With random initial rules the population sometimes fails to manufacture the rules needed for the fundamental equilibrium within 2000 periods, and the economy settles into a pattern with little trade.

Try changing the seed above to see this.

The paper is explicit that the algorithm has "too little experimentation" and that improving it is unfinished business.

89.10 Economy C: fiat money

Now add a fourth object, good 0, which

- costs nothing to store, $s_0 = 0$;
- yields no utility to anybody; and
- cannot be consumed.

It is intrinsically worthless.

It is introduced by handing 48 units of it to 48 randomly chosen agents at $t = 0$, and commodity storage costs are raised to $s = (9, 14, 29)$ so that no commodity is nearly as cheap to store as money.

If a good like this circulates, it can only be because agents have discovered that other agents will take it.

```
economy_c = Economy(
    name='C',
    produces=np.array([1, 2, 0]),
    storage_costs=np.array([9.0, 14.0, 29.0, 0.0]),    # last good is fiat money
    u=100.0,
    method='ga3',
    n_trade_rules=150,
    n_consume_rules=20,
    b_consume=(0.025, 0.25),
    n_fiat=48,
    start='production',
)
```

(continues on next page)

(continued from previous page)

```
sim_c = Simulation(economy_c, seed=2)
sim_c.run(2000, verbose=True)
```

```
  period   200:  trades =  53,  consumptions =  32
  period   400:  trades =  35,  consumptions =  29
  period   600:  trades =  61,  consumptions =  40
  period   800:  trades =  53,  consumptions =  26
  period  1000:  trades =  50,  consumptions =  37
  period  1200:  trades =  49,  consumptions =  32
  period  1400:  trades =  45,  consumptions =  23
  period  1600:  trades =  55,  consumptions =  38
  period  1800:  trades =  63,  consumptions =  36
  period  2000:  trades =  52,  consumptions =  26
```

```
holdings(sim_c, t=750)
```

```
         good 1  good 2  good 3   fiat
type 1    0.000   0.514   0.212  0.274
type 2    0.228   0.000   0.318  0.454
type 3    0.768   0.000   0.000  0.232
```

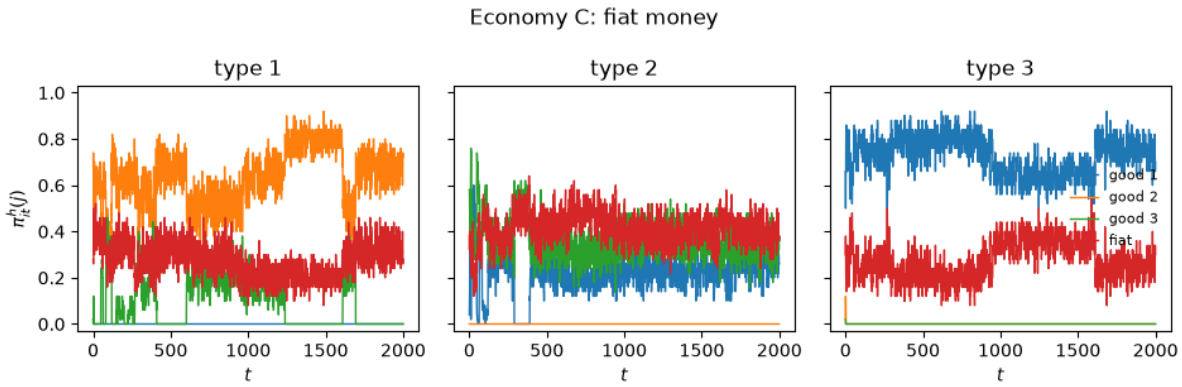
```
holdings(sim_c, t=1250)
```

```
         good 1  good 2  good 3   fiat
type 1    0.000   0.782   0.000  0.218
type 2    0.242   0.000   0.388  0.370
type 3    0.628   0.000   0.000  0.372
```

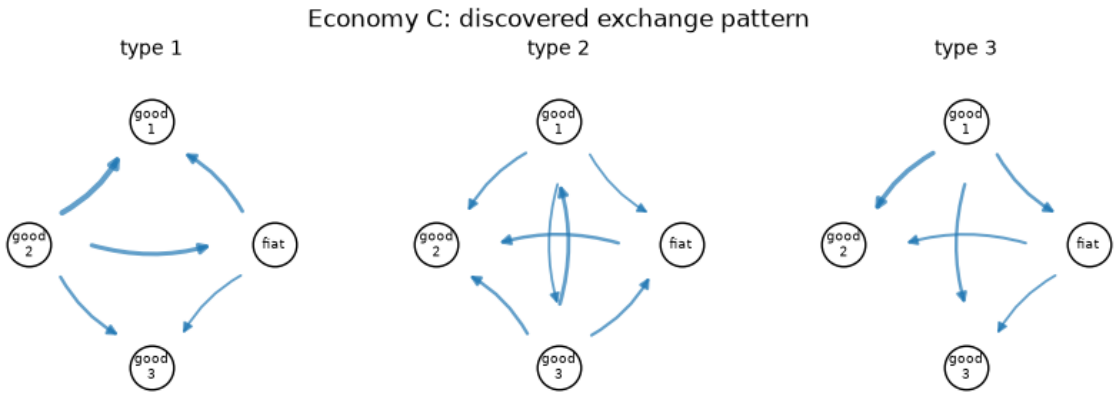
Compare with the paper's table at $t = 1250$, where the columns are (good 1, good 2, good 3, fiat): $(0, 0.54, 0, 0.46)$ for type 1, $(0.18, 0, 0.53, 0.28)$ for type 2, and $(0.77, 0, 0, 0.21)$ for type 3.

Every type holds fiat money a substantial fraction of the time.

```
plot_holdings(sim_c, title="Economy C: fiat money")
```




```
plot_flows(sim_c, title="Economy C: discovered exchange pattern")
```




```
winning_actions(sim_c)
```

```
           good 1      good 2      good 3       fiat
type 1  (1,-,-,-)   (1,1,1,1)   (1,-,-,-)  (1,0,1,1)
type 2  (1,1,1,1)   (-,1,-,1)   (1,1,1,1)  (0,1,0,1)
type 3  (1,1,1,1)   (-,-,1,-)   (-,-,1,-)  (0,1,1,1)
```

The arrows into and out of the fiat node show money changing hands in both directions for every type: agents give up commodities to acquire it and give it up to acquire the commodity they consume.

Nothing in the environment told them to do this.

Each agent discovered only that rules which end up holding the costless object earn more than rules that do not – and, because everyone was discovering this at once, the belief became self-confirming.

This is a genuinely social arrangement built by agents who individually understand nothing about it.

89.11 Economy D: five goods, five types

The last economy has five types and five goods, with production

type i	produces	consumes
1	good 3	good 1
2	good 4	good 2
3	good 5	good 3
4	good 1	good 4
5	good 2	good 5

storage costs $s = (1, 4, 9, 16, 30)$ and $u_i = 200$.

Goods are now encoded in three bits, and each agent carries 180 exchange rules and 20 consumption rules – a small fraction of the possible rules, so the genetic algorithm is essential.

The authors emphasize that they had *no analytical characterization of equilibrium* for this economy before running it.

The simulation is being used as a device to *discover* what an equilibrium might look like, which could then be verified analytically.

```
economy_d = Economy(
    name='D',
    produces=np.array([2, 3, 4, 0, 1]),          # type 1 -> good 3, type 2 -> good 4,␣
 ↪...
    storage_costs=np.array([1.0, 4.0, 9.0, 16.0, 30.0]),
    u=200.0,
    method='ga4',
    n_trade_rules=180,
    n_consume_rules=20,
    start='production',
)

sim_d = Simulation(economy_d, seed=3)
sim_d.run(2000, verbose=True)
```

```
  period   200:  trades =  41,  consumptions =  21
  period   400:  trades =  65,  consumptions =  26
  period   600:  trades =  38,  consumptions =  22
  period   800:  trades =  28,  consumptions =  20
  period  1000:  trades =  36,  consumptions =  26
  period  1200:  trades =  42,  consumptions =  13
  period  1400:  trades =  34,  consumptions =   5
  period  1600:  trades =  36,  consumptions =  25
  period  1800:  trades =  34,  consumptions =  19
  period  2000:  trades =  40,  consumptions =  23
```

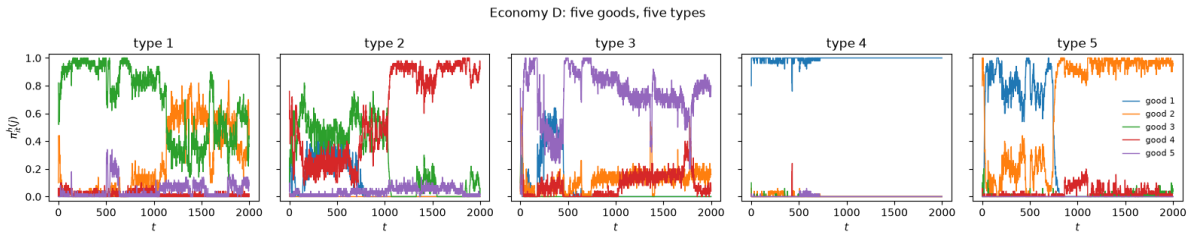
```
holdings(sim_d, t=500)
```

	good 1	good 2	good 3	good 4	good 5
type 1	0.000	0.000	0.972	0.012	0.016
type 2	0.274	0.000	0.504	0.212	0.010
type 3	0.000	0.000	0.000	0.000	1.000
type 4	0.994	0.000	0.006	0.000	0.000
type 5	0.972	0.024	0.000	0.004	0.000

```
holdings(sim_d)
```

	good 1	good 2	good 3	good 4	good 5
type 1	0.0	0.588	0.324	0.008	0.080
type 2	0.0	0.000	0.062	0.926	0.012
type 3	0.0	0.170	0.000	0.074	0.756
type 4	1.0	0.000	0.000	0.000	0.000
type 5	0.0	0.952	0.046	0.002	0.000

```
plot_holdings(sim_d, title="Economy D: five goods, five types")
```



Two features stand out.

First, the diagonal is zero: no type ever ends a period holding the good it consumes, because it consumes it.

Second, each type accumulates its own production good together with goods that are *cheaper* to store, never more expensive.

Type 3, for instance, produces good 5, the most expensive good in the economy, and holds it only part of the time, having traded some of it away for the much cheaper good 2.

Type 4 produces good 1, the cheapest good of all, and simply holds it.

Let us classify the trades that actually take place.

```
def trade_composition(sim, window=200):
    """
    Classify realized trades by what the agent acquires: its own consumption
    good, a good that is cheaper to store than the one given up, or neither.
    """
    econ = sim.econ
    f = np.array(sim.exch_hist)[-window:].sum(axis=0)
    s = econ.storage_costs
    counts = {'own consumption good': 0.0, 'a cheaper good': 0.0, 'neither': 0.0}
    for i in range(econ.n_types):
        for j in range(econ.n_goods):
            for k in range(econ.n_goods):
                if j == k:
                    continue
                key = ('own consumption good' if k == i else
                       'a cheaper good' if s[k] < s[j] else 'neither')
                counts[key] += f[i, j, k]
    total = sum(counts.values())
    return pd.Series({k: round(v / total, 3) for k, v in counts.items()},
                     name='share of realized trades')


trade_composition(sim_d)
```

```
own consumption good    0.364
a cheaper good          0.301
neither                 0.335
Name: share of realized trades, dtype: float64
```

Roughly two thirds of realized trades acquire either the agent's own consumption good or a good that is cheaper to store, which is the pattern the paper describes.

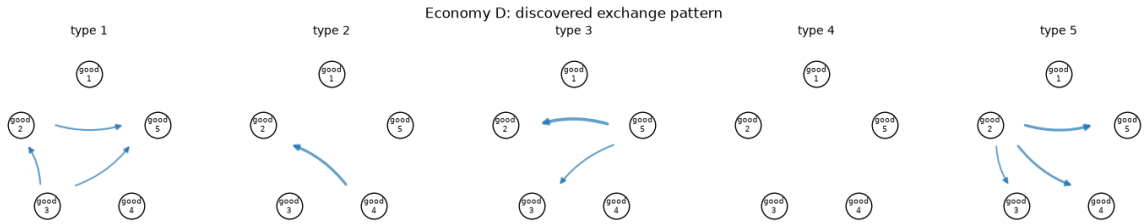
The remaining third deserves a comment, because it is not evidence against that pattern.

Consider a type 1 agent, who produces good 3, swapping good 3 for the far more expensive good 5.

He then consumes good 5 – getting no utility from it – and produces good 3 again, so he ends the period holding good 3 and paying s_3, exactly as he would have had he refused.

Such trades are payoff-neutral, so nothing in the accounting system pushes the rules that generate them out of the population.

```
plot_flows(sim_d, window=200, cutoff=0.03,
           title="Economy D: discovered exchange pattern")
```



The paper summarizes what it sees in these patterns:

> From the simulation results, we can see that the trading patterns nearly seem to describe a fundamental equilibrium in which agents are only willing to trade for commodities of lower cost than the one currently in storage, except that they always accept the commodity of their type. Some speculative moves can be detected.

An example of such a speculative move is a type 2 agent accepting good 3 for good 1, not because good 3 is cheap but because type 3 agents will take it in exchange for good 2.

89.12 Concluding remarks

Marimon, McGrattan and Sargent's agents know very little.

They do not know their own utility functions, they do not know storage costs, they do not know the distribution of goods across the population, and they certainly do not solve dynamic programs.

They recognize utility when they experience it and costs when they bear them, and they keep running averages.

Out of this the following emerges:

- **Nash-Markov behavior is learnable.** In most of the economies simulated, holdings and trading patterns converge to a stationary Nash equilibrium of the Kiyotaki-Wright model.
- **Learning selects among equilibria.** Where the rational-expectations model admits both a fundamental and a speculative equilibrium, the classifier systems always found the fundamental one.
  - Economy A2 shows they found it even where theory says it should not be an equilibrium at all for patient agents.
  - Economy B shows this is not simply early myopia, since that economy moves *away* from a speculative pattern as it learns.
- **Institutions can be discovered.** Economy C's agents built a fiat monetary system out of nothing but their own experience of storage costs.
- **The method scales.** Economy D produced a credible description of equilibrium in a model its authors had not solved.

The paper is candid about what is missing.

There are no convergence theorems, only a sketch of how stochastic approximation arguments might supply them; and the authors judge their genetic algorithm to provide "too little experimentation" – which is precisely why the speculative equilibrium never appears.

That diagnosis, that an adaptive system's selection among equilibria is governed by how much it explores and when, has proved durable.

89.13 Exercises

Exercise 89.13.1

Economy A2.2 is Economy A2 – with $u_i = 500$ – but started from randomly generated rules and evolved with the `'ga3'` genetic algorithm, exactly as A1.2 relates to A1.1.

The paper's summary table lists its equilibrium type as speculative, and the authors report that after 1000 iterations the economy had not converged, with trading patterns closer to the fundamental equilibrium at $t = 1000$ than at $t = 500$.

Simulate it for 2000 periods and report holdings at $t = 500$, $t = 1000$ and $t = 2000$.

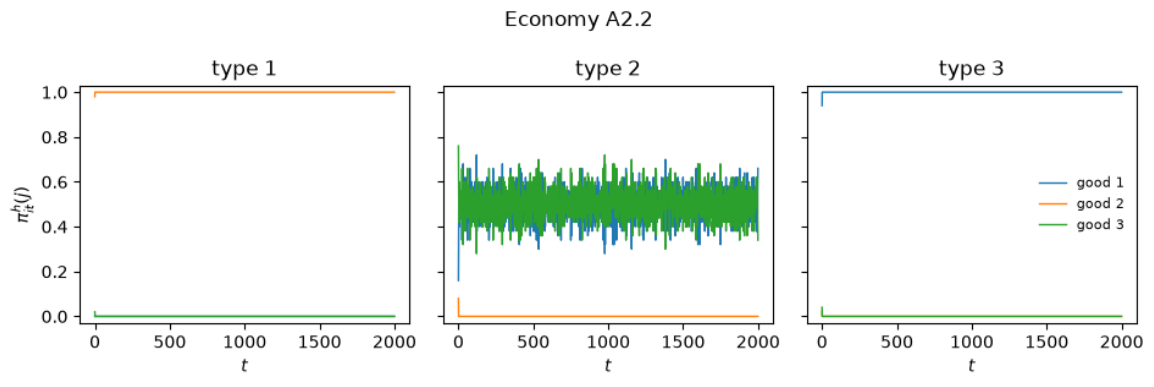
Does the extra experimentation supplied by the genetic algorithm produce speculation?

Solution

```
economy_a22 = Economy(
    name='A2.2',
    produces=np.array([1, 2, 0]),
    storage_costs=np.array([0.1, 1.0, 20.0]),
    u=500.0,
    method='ga3',
)

sim_a22 = Simulation(economy_a22, seed=3)
sim_a22.run(2000)

for t in (500, 1000, 2000):
    print(f"\nt = {t}")
    print(holdings(sim_a22, t=t))
```

```
t = 500
        good 1  good 2  good 3
type 1    0.00     1.0    0.00
type 2    0.47     0.0    0.53
type 3    1.00     0.0    0.00

t = 1000
        good 1  good 2  good 3
type 1   0.000     1.0   0.000
type 2   0.492     0.0   0.508
type 3   1.000     0.0   0.000

t = 2000
        good 1  good 2  good 3
type 1   0.000     1.0   0.000
type 2   0.494     0.0   0.506
type 3   1.000     0.0   0.000
```

```
plot_holdings(sim_a22, title="Economy A2.2")
```



The economy again settles on the fundamental pattern: type 1 holds good 2, type 3 holds good 1, and type 2 alternates between goods 1 and 3.

Type 1 agents are not holding good 3, so they are not speculating.

Randomizing the initial rules and letting the genetic algorithm run does not by itself generate enough experimentation to sustain speculation – which is the paper's own conclusion.

Exercise 89.13.2

Economy B.2 is Economy B started from random rules with the `'ga3'` genetic algorithm.

The paper reports that it "had not converged after 2000 periods" but was "moving towards the fundamental equilibrium", with holdings at $t = 2000$ of $(0, 0.354, 0.646)$ for type 1, $(0.996, 0, 0.004)$ for type 2 and $(0.268, 0.732, 0)$ for type 3.

Simulate it and compare with the complete-enumeration Economy B.1 that we ran above.

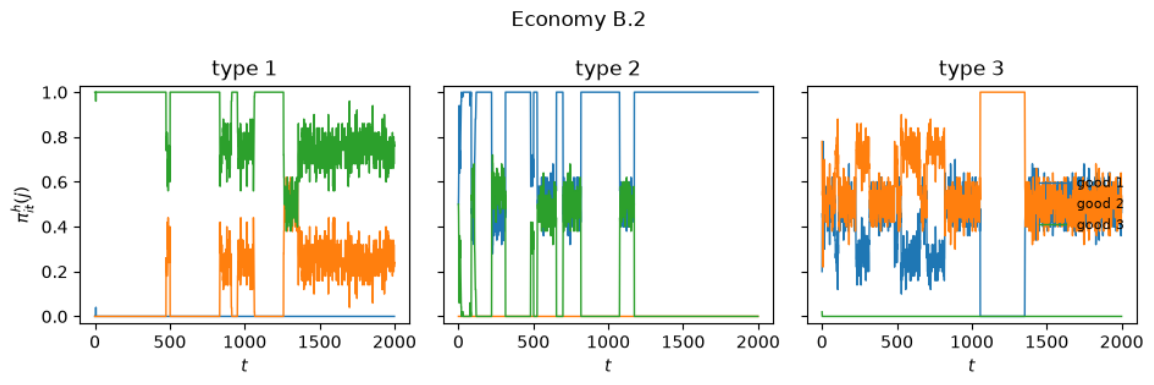
Solution

```
economy_b2 = Economy(
    name='B.2',
    produces=np.array([2, 0, 1]),
    storage_costs=np.array([1.0, 4.0, 9.0]),
    u=100.0,
    method='ga3',
)

sim_b2 = Simulation(economy_b2, seed=1)
sim_b2.run(2000)

for t in (500, 1000, 2000):
    print(f"B.2 at t = {t}")
    print(holdings(sim_b2, t=t))
    print()
print("B.1 (complete enumeration) at t = 1000, for comparison")
print(holdings(sim_b1))
```

```
B.2 at t = 500
        good 1  good 2  good 3
type 1   0.000   0.304   0.696
```

```
type 2    0.676    0.000    0.324
type 3    0.336    0.664    0.000

B.2 at t = 1000
         good 1   good 2   good 3
type 1    0.000    0.252    0.748
type 2    1.000    0.000    0.000
type 3    0.506    0.494    0.000

B.2 at t = 2000
         good 1   good 2   good 3
type 1    0.000    0.262    0.738
type 2    1.000    0.000    0.000
type 3    0.472    0.528    0.000

B.1 (complete enumeration) at t = 1000, for comparison
         good 1   good 2   good 3
type 1    0.000    0.324    0.676
type 2    1.000    0.000    0.000
type 3    0.626    0.374    0.000
```

```
plot_holdings(sim_b2, title="Economy B.2")
```



Follow the two types that move.

Type 2 agents are still split at $t = 500$ and have locked onto good 1 by $t = 1000$.

Type 3 agents start out mostly holding their own production good 2 and shift a substantial part of their holdings into the cheaper good 1 over the same interval, which is the movement toward the fundamental equilibrium that the paper describes.

After that the type 3 split stops moving in one direction and rattles around a half-and-half mix from one reading to the next, so this economy has not converged in the sense that B.1 has.

That is the paper's own verdict on it.

Compared with the complete-enumeration run, the genetic algorithm gets type 1 and type 2 to much the same place and leaves type 3 less far along.

Building the needed rules from random material costs time that the enumerated economy never had to spend.

Exercise 89.13.3

The bid functions $b_1(e) = b_{11} + b_{12}\sigma_e$ favor specific rules over general ones, since σ_e falls with the number of wildcards.

What happens if this tilt is removed?

Rerun Economy A1.1 with $b_{12} = 0$, holding $b_{11} + b_{12}$ fixed at 0.05, and compare the number of wildcards in the winning rules with those in the baseline.

A single run will not settle the question, so average over several seeds.

Solution

```
economy_flat = Economy(
    name='A1.1 with no specificity premium',
    produces=np.array([1, 2, 0]),
    storage_costs=np.array([0.1, 1.0, 20.0]),
    u=100.0,
    method='enumerate',
    b_trade=(0.05, 0.0),
)

sim_flat = Simulation(economy_flat, seed=42)
sim_flat.run(1000)

print(holdings(sim_flat))
```

```
         good 1  good 2  good 3
type 1     0.00     1.0     0.00
type 2     0.53     0.0     0.47
type 3     1.00     0.0     0.00
```

The holdings are unchanged: the fundamental equilibrium is reached either way.

What changes is the *kind of rule* that gets there.

```
def wildcards_in_winners(sim):
    "Average number of wildcards in the exchange rule that wins in each state."
    econ = sim.econ
    counts = []
    for A in sim.agents:
        for j in range(econ.n_goods):
            for k in range(econ.n_goods):
                w, _ = auction(A.trade, np.concatenate([econ.code(j), econ.
 ↪code(k)]))
                if w >= 0:
                    counts.append(np.count_nonzero(A.trade.cond[w] == WILD))
    return np.mean(counts)


def average_wildcards(b_trade, seeds=range(8)):
    out = []
    for seed in seeds:
        econ = Economy(name='sweep', produces=np.array([1, 2, 0]),
                       storage_costs=np.array([0.1, 1.0, 20.0]), u=100.0,
                       method='enumerate', b_trade=b_trade)
        sim = Simulation(econ, seed=seed)
        sim.run(1000)
        out.append(wildcards_in_winners(sim))
    return np.array(out)


base = average_wildcards((0.025, 0.025))
flat = average_wildcards((0.05, 0.0))

pd.DataFrame({"baseline $b_{12} = 0.025$": base.round(2),
```

```
      baseline $b_{12} = 0.025$  no premium $b_{12} = 0$
seed
0                          0.30                       0.19
1                          0.00                       0.00
2                          0.19                       0.30
3                          0.33                       0.33
4                          0.22                       0.41
5                          0.07                       0.22
6                          0.07                       0.11
7                          0.22                       0.37
```

```
print(f"mean wildcards, baseline           : {base.mean():.3f}")
print(f"mean wildcards, no specificity bid : {flat.mean():.3f}")
print(f"higher without the premium in {np.sum(flat > base)} of {len(base)} seeds")
```

```
mean wildcards, baseline           : 0.176
mean wildcards, no specificity bid : 0.241
higher without the premium in 5 of 8 seeds
```

Removing the specificity premium raises the average generality of the winning rules, but the effect is modest and does not show up in every run.

That is worth knowing rather than glossing over.

With a complete enumeration the specific rules are all present from the start and accumulate strength on their own, so the bid premium is only one of the forces tilting the auction toward them; the counter is a genuine tendency visible in the average rather than a law obeyed run by run.

The tilt matters more where it is harder to see.

It is the mechanism the paper blames for the failure of Economy A2: general consumption classifiers cannot tell stored goods apart, so they cannot transmit to the exchange classifiers the information that would make speculation profitable.

CHAPTER

NINETY

1993 PROSPECTS FOR BOUNDED RATIONALITY IN MACROECONOMICS

Contents

- *1993 Prospects for Bounded Rationality in Macroeconomics*
  - *Overview*
  - *The quest, restated*
  - *The debit side: how much we must hard-wire*
  - *Why the econometricians have not returned the compliment*
  - *The credit side: selection, computation, and a returned gift*
  - *Reading the ledger against the quest*
  - *Postscript: what became of the program*

90.1 Overview

This is the closing lecture of the series, and it is different in kind from the others.

It carries no new model.

Instead it gathers the judgments that Sargent [1993] recorded in 1993 — the opinions, reservations, and hopes with which he ended *Bounded Rationality in Macroeconomics* — and loops them back to the quest that opened the book, and that opened *A Peculiar Definition of Bounded Rationality*, the first lecture here.

That quest had a destination: a theory of **transition dynamics**, the out-of-equilibrium adjustment that the Eastern European reformers of 1989 had to manage with no map.

The route was to expel the rational agents from our models and replace them with "artificially intelligent" agents who behave like **econometricians**, who gather data, form theories, estimate, and adapt.

We can now ask how far that route carried us, by 1993, toward the destination.

Sargent's own answer is a ledger, with entries on both sides.

On the credit side: adaptive dynamics as a device for **selecting** among equilibria, and as a tool — evolutionary programming — for **computing** them.

On the debit side: the original prize, a theory of transition dynamics, is largely unclaimed; and a striking asymmetry has emerged.

The program set out to make the agents in our models behave more like econometricians.

The econometricians, Sargent observes, have not returned the compliment.

This lecture explains that asymmetry, gives it a concrete numerical face, and reads the ledger against the opening quest.

The lecture then closes with a postscript on what became of the program after 1993.

Let's start with some imports.

```
import numpy as np
import matplotlib.pyplot as plt
```

90.2 The quest, restated

It is worth restating the argument of the first lecture, because everything below is measured against it.

Rational expectations imposes two requirements: individual rationality, and mutual consistency of perceptions.

The second is the demanding one, and — this is the crux — when a rational expectations model is taken to data, it imputes far more knowledge to the agents inside it than to the econometrician studying them.

The agents evaluate their Euler equations using the *equilibrium* probability distributions, the very distributions the econometrician is still struggling to estimate.

The bounded rationality program proposes to close that gap by demoting the agents to the econometrician's own level: they too must learn the distributions, from data, as they go.

The hope was that this would deliver something rational expectations cannot: a description of the system *while it is still adjusting*, before beliefs and outcomes have settled into mutual consistency.

That is what a theory of transition dynamics would be.

We now take stock, following Sargent [1993]'s own accounting: first the debits, then the central asymmetry, then the credits.

90.3 The debit side: how much we must hard-wire

The first reservation is about **arbitrariness**.

Bounded rationality is most easily defined by what it is *not* — rational expectations — and that very malleability is a liability.

Once we stop insisting that agents know the equilibrium, we must decide, case by case, exactly what they *do* know and how they learn it.

Do they know their own utility and profit functions, or must they learn those too?

Do they know calculus and dynamic programming, or only trial and error?

Do they learn from their own experience alone, or from others'?

To what class of approximating functions do we confine what they learn about?

Every model in this series answered these questions by **hard-wiring**, by prompting the agents heavily, with an eye on the outcome we hoped they would reach.

Bray's agents, in *A Peculiar Definition of Bounded Rationality*'s adaptive-expectations model and in the least squares learning literature, know the correct supply curve and need only estimate one conditional expectation to plug into it.

The Marcet–Sargent agents know dynamic programming, know their return function, and know the parametric form of the law of motion — they lack only its coefficients, which they update by vector autoregression.

Even the classifier agents of *Money as a Medium of Exchange among Artificially Intelligent Agents*, which are prompted far less — they are never told their utility functions, and recognize utility only when they experience it — are still told *when* to choose and *what* information to condition on, and the entire apparatus of their accounting system and genetic operators is designed by hand, with the Kiyotaki–Wright equilibrium in view.

The second reservation is about **simplicity**.

The learning tasks we set our agents are trivial next to those in a first econometrics course, let alone those that real firms and households are implicitly solving.

We ask an agent to learn a single time-invariant decision rule, or a fixed collection of conditional expectations.

We do not ask it to learn the parameters of a simultaneous-equations system, or to infer a mapping from policy regimes to distributions, the tasks that make econometrics hard.

Put together, these reservations bear directly on the original prize.

The environments into which we have cast our adaptive agents are far more stable and hospitable than the transitions we actually care about.

Convergence-rate results are scarce, and tractability forces us to restrict the distribution of agents' beliefs severely.

So the literature on adaptive processes, Sargent judged in 1993, falls well short of a secure foundation for a theory of real-time transition dynamics, the very thing the quest set out to find.

He declines to end on that failure, though, and the baseball metaphor he reaches for is deliberately modest:

> It would not be wise or fair to end this essay by dwelling on the failure of adaptive methods so far to have 'hit a home run' by giving us a good theory of transition dynamics. The problem of transition dynamics is difficult and long-standing. So maybe it should count as a single, or at least a sacrifice fly, that these methods have sharpened our appreciation of the problem.

90.4 Why the econometricians have not returned the compliment

Here is the asymmetry that gives this lecture its theme.

The bounded rationality program is, at bottom, a movement to make the agents in our models behave more like the econometricians who build and estimate those models.

Imitation is the sincerest form of flattery.

We might therefore have expected macroeconometricians to rush to fit these models to data.

There was no rush.

Chung [1990]'s estimation of the Sims policy-maker model — the application in *Adaptive Agents in an Overlapping Generations Monetary Economy* — was, Sargent noted, close to the *only* econometrically serious macroeconomic implementation of bounded rationality he knew of.

Why the reluctance?

The reasons are worth spelling out, because they are not about taste.

The governing dictum among applied econometricians is Lucas's: *beware of theorists bearing free parameters*.

Replacing a rational agent with a boundedly rational one **adds** parameters: parameters describing beliefs and how beliefs move.

Take the simplest case, Bray's model.

Relative to its rational expectations version, the adaptive version adds at least the initial belief and a parameter setting the gain sequence, and one might want more parameters to describe the gain's shape.

That would already give Lucas's warning something to bite on.

But there is a deeper problem, and it is the reason the added parameters are not just unwelcome but genuinely hard to estimate.

Because the adaptive system **converges** to the rational expectations equilibrium, the extra parameters influence only the *transient*.

The asymptotic distribution of the data contains no information about them.

Let us see this directly.

90.4.1 The nuisance parameter, made concrete

Bray's cobweb economy [Bray, 1982] sets the market price by

$$p_t = a + b\,\beta_t + u_t,$$

where β_t is the price agents expect, formed by averaging past prices,

$$\beta_t = \beta_{t-1} + \gamma_t\,(p_{t-1} - \beta_{t-1}), \qquad \gamma_t = \frac{1}{t + t_0},$$

and u_t is an IID shock.

The constant t_0 is the weight the agents attach to the belief they start with, measured in observations: with $t_0 = 50$ they treat β_0 as though it summarized fifty prior prices, so that after t periods β_0 still carries weight $t_0/(t + t_0)$.

Some such weight is needed for the exercise to have any content.

With $t_0 = 0$ the first gain is $\gamma_1 = 1$, so $\beta_1 = p_0$ exactly and the initial belief is erased after a single period — there would be no transient left for the econometrician to try to estimate.

When $b < 1$ the belief converges to the rational expectations value $\beta^\star = a/(1 - b)$, whatever value it started from.

```
a, b, sigma_u = 5.0, 0.7, 1.0
t0 = 50                                      # weight on the initial belief
β_star = a / (1 - b)                         # rational expectations belief

def simulate(β0, u):
    "Bray's cobweb under least squares learning, given a shock path u."
    T = len(u)
    β = np.empty(T)
    p = np.empty(T)
    β[0] = β0
    for t in range(T):
        p[t] = a + b * β[t] + u[t]
        if t + 1 < T:
            β[t + 1] = β[t] + (1 / (t + 1 + t0)) * (p[t] - β[t])
    return p, β

print(f"rational expectations belief β* = {β_star:.3f}")
```

```
rational expectations belief β* = 16.667
```

The initial belief β_0 is the extra "bounded rationality" parameter.

Watch three economies with very different initial beliefs forget where they started.

```
rng = np.random.default_rng(0)
u = rng.standard_normal(400)

fig, ax = plt.subplots(figsize=(7.5, 4))
for β0, colour in [(2.0, 'C0'), (16.667, 'C1'), (40.0, 'C2')]:
    _, β = simulate(β0, u)
    ax.plot(β, color=colour, lw=1.3, label=fr"$\beta_0 = {β0}$")
ax.axhline(β_star, color='k', ls='--', lw=0.8, label=r"$\beta^\star$")
ax.set_xlabel("$t$")
ax.set_ylabel(r"belief $\beta_t$")
ax.legend(frameon=False)
plt.show()
```

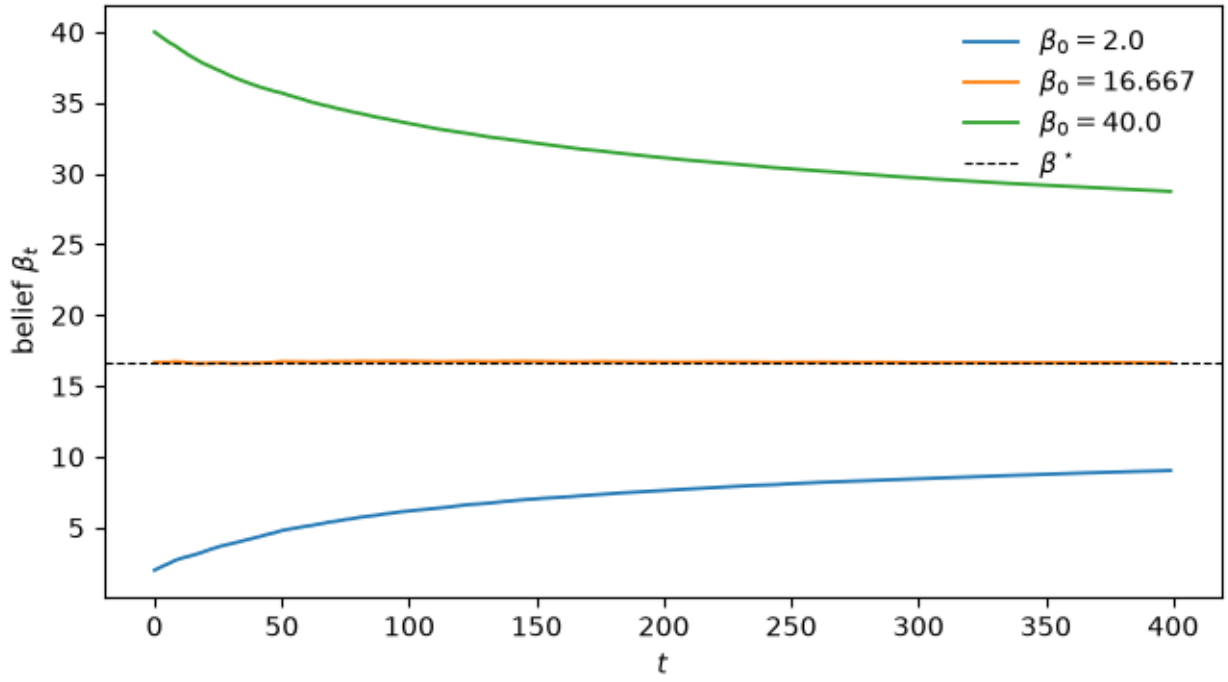


Fig. 90.1: The belief forgets its starting point

All three converge to $\beta^\star$.

Now put on the econometrician's hat.

Given a sample of prices and the structural parameters (a, b), the model implies a shock $\hat{u}_t = p_t - a - b\,\beta_t(\beta_0)$ for any candidate initial belief β_0, because the belief path is pinned down by β_0 and the observed prices.

The sum of squared implied shocks measures how well a given β_0 fits the data.

```
def belief_path(β0, p):
    "Belief sequence implied by an initial belief and an observed price path."
    T = len(p)
    β = np.empty(T)
```

(continues on next page)

(continued from previous page)

```
    β[0] = β0
    for t in range(1, T):
        β[t] = β[t - 1] + (1 / (t + t0)) * (p[t - 1] - β[t - 1])
    return β

def ssr(β0, p):
    "Sum of squared implied shocks, as a function of the belief parameter β0."
    β = belief_path(β0, p)
    return np.sum((p - a - b * β) ** 2)
```

Whether the data can pin β_0 down is a question of how fast *information* about it accumulates as the sample grows.

The natural way to read that off is the **excess** sum of squares — how much worse a wrong β_0 fits than the best one.

For an ordinary, well-identified parameter every new observation adds to the penalty for being wrong, so the excess grows in proportion to T and the confidence interval shrinks like $1/\sqrt{T}$.

Watch what happens here.

```
grid = np.linspace(2, 40, 80)

def excess_curve(T, seed=1):
    "SSR(β₀) - min SSR across the β₀ grid, for a sample of length T."
    u = np.random.default_rng(seed).standard_normal(T)
    p, _ = simulate(β_star, u)
    curve = np.array([ssr(b0, p) for b0 in grid])
    return curve - curve.min()

fig, ax = plt.subplots(figsize=(7.5, 4))
for T, colour in zip((50, 500, 5000), ('C0', 'C1', 'C2')):
    ax.plot(grid, excess_curve(T), color=colour, lw=1.5, label=f"$T = {T}$")
ax.axvline(β_star, color='k', ls='--', lw=0.8)
ax.set_xlabel(r"belief parameter $\beta_0$")
ax.set_ylabel("excess sum of squares")
ax.legend(frameon=False)
plt.show()
```

The curves stack on top of one another instead of getting steeper.

To see that this is not an artifact of the range plotted, compare the penalty for a wrong β_0 with the penalty for a wrong *slope* b, an ordinary structural parameter of the same model, estimated from the same data.

```
rows = []
for T in (50, 500, 5_000, 50_000):
    u = np.random.default_rng(1).standard_normal(T)
    p, _ = simulate(β_star, u)
    β = belief_path(β_star, p)                    # belief path at the true β0
    wrong_β0 = ssr(2.0, p) - ssr(β_star, p)
    wrong_slope = (np.sum((p - a - 0.75 * β) ** 2)
                   - np.sum((p - a - b * β) ** 2))
    rows.append([T, wrong_β0, wrong_slope])

for T, e_b0, e_b in rows:
    print(f"T = {T:6d}:  penalty for β₀ = 2 : {e_b0:10.1f}     "
          f"penalty for b = 0.75 : {e_b:12.1f}")
```

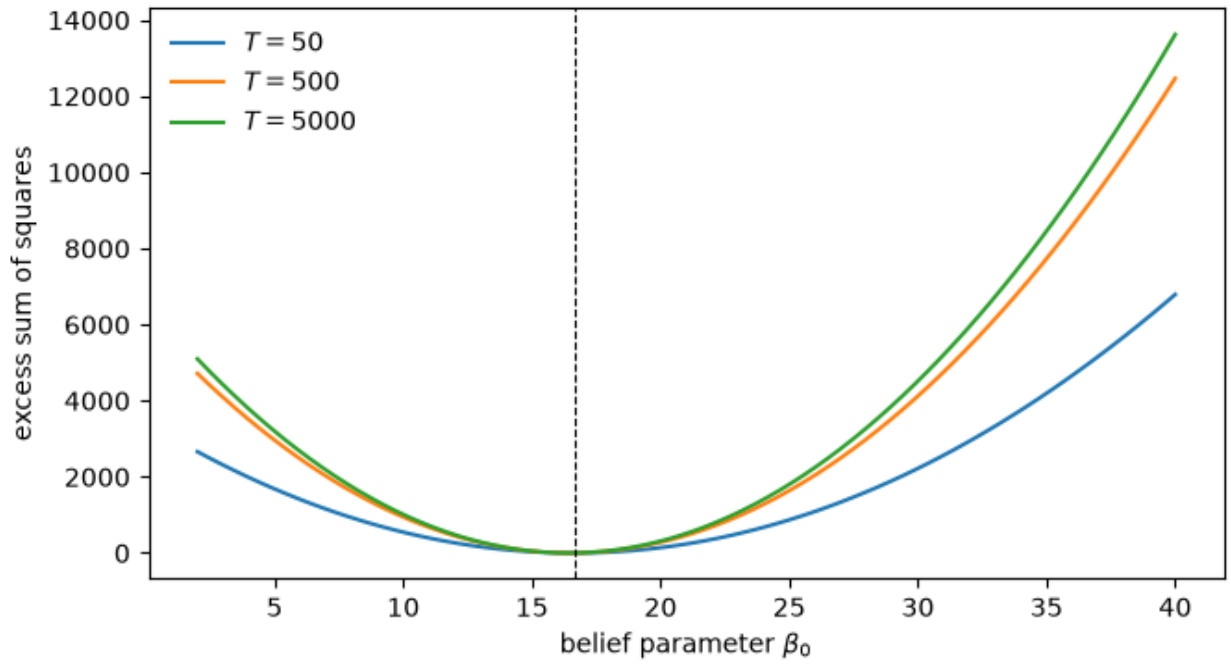


Fig. 90.2: Information about the belief parameter stops accumulating

```
T =     50:  penalty for β₀ = 2 :     2662.3     penalty for b = 0.75 :          37.
↪8
T =    500:  penalty for β₀ = 2 :     4714.1     penalty for b = 0.75 :         373.
↪4
T =   5000:  penalty for β₀ = 2 :     5097.1     penalty for b = 0.75 :        3558.
↪9
T =  50000:  penalty for β₀ = 2 :     5127.5     penalty for b = 0.75 :       35194.
↪0
```

The two columns behave completely differently.

The penalty for getting the slope wrong grows in proportion to the sample: a hundredfold more data makes a hundredfold stronger case against the wrong value, which is what identification looks like.

The penalty for getting the initial belief wrong stops growing.

Past a few thousand observations it is pinned at a constant, and every further observation is uninformative about β_0.

The confidence interval for β_0 never shrinks; the parameter is not consistently estimable at all.

This is the technical heart of the matter.

The parameters that bounded rationality adds live entirely in the transient.

A transient contributes a fixed, finite amount of information no matter how long we watch the economy afterwards, so those parameters become a **nuisance to estimate** — they enter the likelihood, and the data have only ever a bounded amount to say about them.

And there is a final, decisive reason for the econometricians' cool response.

Many applied macroeconometricians are in the market for methods that *reduce* the number of parameters needed to explain the data.

A reduction is precisely what bounded rationality does not offer.

It offers more parameters, most of them weakly identified.

So the flattery ran one way.

The theorists remade their agents in the econometrician's image; the econometricians, offered models full of extra, poorly-identified parameters, and warned by Lucas against exactly that, declined the gift.

90.5 The credit side: selection, computation, and a returned gift

The ledger is not one-sided.

Set against those debits, Sargent lists three genuine successes, and the last of them quietly undoes some of the asymmetry just described.

Equilibrium selection.

Where a rational expectations model has many equilibria, a system of adaptive agents often converges to a *particular* one, turning learning into a device for selecting among them.

We saw it repeatedly: the low-inflation equilibrium chosen in *Adaptive Agents in an Overlapping Generations Monetary Economy*, the history-dependent exchange rate of *Exchange Rate Indeterminacy, Learning, and Experiments*, the fundamental monetary equilibrium of *Money as a Medium of Exchange among Artificially Intelligent Agents*.

Sargent is candid that his affection for this use sits in some tension with his doubts about the dynamics that perform the selection:

> I know that it is inconsistent to doubt the real-time dynamics but keep the equilibria selected by them. I confess that my affection for the selection performed in the monetary models described in Chapter 6 is partly driven by my prior conviction that the selected equilibria seem sensible to me.

Evolutionary programming.

If a population of adaptive agents reliably converges to an equilibrium, we can run the population as a *method of computing* the equilibrium, especially in models too complicated to solve by hand.

That is exactly what *Money as a Medium of Exchange among Artificially Intelligent Agents* did with the five-good Kiyotaki–Wright economy, for which no analytical characterization was in hand.

Sargent expected this use to be applied often.

New tools for the econometrician.

Here the asymmetry bends back on itself.

The literatures on parallel and genetic algorithms have handed econometricians new computational gadgets — genetic algorithms and stochastic Gauss–Newton procedures among them — for solving their *own* estimation and optimization problems.

McGrattan, for instance, used genetic algorithms to search for the neighborhood of a maximum before switching to a Newton method, a use noted back in *Genetic Algorithms and Classifier Systems*.

So the econometricians took up the adaptive algorithms after all, not as *models* of the agents they study, but as *tools* in their own hands.

The compliment was returned, but through a side door: the algorithms crossed over, the models did not.

90.6 Reading the ledger against the quest

Line the two columns up against the destination we set out for.

The **debit** column holds the prize itself.

A theory of real-time transition dynamics — the map the Eastern European reformers lacked — remained, in 1993, largely unclaimed, blocked by arbitrariness, prompting, over-simple learning tasks, and a shortage of empirical traction.

The **credit** column holds what the journey delivered along the way: a principled way to select among multiple equilibria, a practical way to compute equilibria that resist analysis, and a transfer of computational technique into econometrics itself.

The organizing image of the whole book is the gap between the econometrician and the agents inside the model.

Rational expectations closes that gap by lifting the agents up to a knowledge the econometrician lacks.

Bounded rationality proposed to close it from the other side, by bringing the agents down to the econometrician's level and making them learn.

By 1993 the program had not, on Sargent's own accounting, delivered the transition dynamics that motivated it, and the econometricians it sought to imitate had kept the models at arm's length — for the sound reason that those models add parameters the data cannot identify, when what the econometrician wants is fewer.

But it had sharpened the questions, selected equilibria, computed them, and lent its tools to the very econometricians who declined its models.

That was the state of the prospects for bounded rationality in macroeconomics, as they stood in 1993.

90.7 Postscript: what became of the program

Three decades is long enough to see which of the 1993 worries were permanent and which were about a field that had not yet found its footing.

90.7.1 The arbitrariness was disciplined

The first reservation was that once we stop insisting agents know the equilibrium, nothing tells us what to put in its place.

The discipline that emerged is **expectational stability**.

Evans and Honkapohja [Evans and Honkapohja, 2001] showed that whether a rational expectations equilibrium is learnable is governed by a condition on the map from perceived to actual laws of motion — the very T map of *A Peculiar Definition of Bounded Rationality* — and that the condition is largely *independent* of the details of the learning algorithm.

That is exactly what was missing in 1993.

Selection is no longer an artifact of whichever recursion the modeller happened to write down; a large class of reasonable learning rules select the same equilibria, and one can check which those are without simulating anything.

The stability reversal of *Adaptive Agents in an Overlapping Generations Monetary Economy* is a case in point.

It looked in 1993 like a fact about least squares, propped up by Bruno and Fischer [1990] having found the same thing with a different estimator.

E-stability explains why the two agreed.

90.7.2 The transition dynamics arrived, in a narrower form than hoped

The prize was a theory of out-of-equilibrium adjustment.

What the program delivered instead was a theory of *departures from* equilibrium: escape dynamics.

The sawtooth we simulated at the end of *Adaptive Agents in an Overlapping Generations Monetary Economy* was, in 1993, a numerical curiosity.

Cho *et al.* [2002] characterized it analytically with large-deviations theory, computing the most likely escape path and the rate at which escapes occur, and Williams [2019] extended the characterization considerably.

The QuantEcon lectures *Escaping Nash Inflation* and *Priors, Escapes, and Learning Cycles* work through both.

This is less than the original quest asked for.

It describes recurrent excursions away from a self-confirming equilibrium, not the arrival of a market economy in a country that never had one.

But it is a genuine theory of a system that does not settle down, derived rather than simulated, and in 1993 there was none.

90.7.3 The econometricians did return the compliment

Here the 1993 assessment was simply overtaken.

The obstacle, we saw above, was that the added parameters live in a vanishing transient.

But that argument applies only to a learning scheme with a $1/t$ gain, which converges and then stops moving.

Constant-gain learning has no such transient. Beliefs never settle; they keep moving forever, and their movement is part of the stationary distribution of the data.

So the gain is identified the way an ordinary structural parameter is — from the whole sample, at the usual rate — and not, like β_0, from a bounded initial episode.

Let us check that on the model we have been using.

```
def simulate_constant_gain(β0, gain, u):
    "Bray's cobweb when agents discount old prices at a fixed rate."
    T = len(u)
    β, p = np.empty(T), np.empty(T)
    β[0] = β0
    for t in range(T):
        p[t] = a + b * β[t] + u[t]
        if t + 1 < T:
            β[t + 1] = β[t] + gain * (p[t] - β[t])
    return p, β

def ssr_gain(g_hat, p, β0):
    "Fit criterion for a candidate gain, given observed prices."
    T = len(p)
    β = np.empty(T)
    β[0] = β0
    for t in range(1, T):
        β[t] = β[t - 1] + g_hat * (p[t - 1] - β[t - 1])
    return np.sum((p - a - b * β) ** 2)

gain_true = 0.05
for T in (500, 5_000, 50_000):
```

(continues on next page)

(continued from previous page)

```
    penalties = []
    for seed in range(20):                      # average out sampling noise
        u = np.random.default_rng(seed).standard_normal(T)
        p, _ = simulate_constant_gain(β_star, gain_true, u)
        penalties.append(ssr_gain(0.08, p, β_star) - ssr_gain(gain_true, p, β_star))
    print(f"T = {T:6d}:  mean penalty for using gain 0.08 instead of 0.05 : "
          f"{np.mean(penalties):9.1f}")
```

```
T =    500:  mean penalty for using gain 0.08 instead of 0.05 :       1.1
T =   5000:  mean penalty for using gain 0.08 instead of 0.05 :       9.8
T =  50000:  mean penalty for using gain 0.08 instead of 0.05 :     123.6
```

The penalty grows in proportion to the sample, exactly as it did for the slope b and exactly as it did *not* for the initial belief.

That is why the econometric work that eventually materialized uses constant gain.

Sargent *et al.* [2006] estimated a constant-gain learning model of the Federal Reserve on post-war U.S. data — imputing to the government inside the model a genuine recursive estimation procedure, and asking the data which gain it used.

Sargent and Williams [2005] studied how the government's prior about drifting coefficients shapes what it converges to, and *Priors, Escapes, and Learning Cycles* develops that.

Drifts and Volatilities fits a drifting-coefficient VAR to the same episode and asks whether it was bad policy or bad luck.

So the flattery did run both ways in the end.

It took a change in the learning technology — from a scheme that converges to one that never does — to make the models estimable, and that change was made for reasons of economics rather than econometrics.

90.7.4 A second retreat, made differently

The program in this book keeps individual rationality and gives up mutual consistency: agents optimize, but against beliefs they are still estimating.

A parallel literature retreats along the other axis.

In the **robustness** work of Hansen and Sargent [Hansen and Sargent, 2008], agents do not estimate their model at all.

They admit that they cannot know it, and optimize against the worst case among the models they cannot rule out.

The two are complements rather than rivals, and both are answers to the question that opens *A Peculiar Definition of Bounded Rationality*: what do we do about the knowledge that rational expectations imputes?

One answer is that agents should learn what the econometrician is learning.

The other is that they should behave well without ever learning it.

90.7.5 The algorithms kept crossing over

The 1993 ledger noted that econometricians had adopted the adaptive algorithms as computational tools even while declining the models.

That traffic increased, and reversed direction again.

Holland's bucket brigade of *Genetic Algorithms and Classifier Systems* — pay part of your reward backward to whatever set you up — is *temporal-difference learning*, which became the organizing idea of modern reinforcement learning [Sutton and Barto, 2018].

The classifier systems of *Money as a Medium of Exchange among Artificially Intelligent Agents* are recognizable, in retrospect, as reinforcement learners with a hand-built function approximator; and the perceptrons of *Genetic Algorithms and Classifier Systems* became the deep networks of *Introduction to Artificial Neural Networks*, which now serve as the function approximators.

Sargent's artificially intelligent agents were not, it turned out, a metaphor borrowed from a neighboring field.

They were an early instance of what that field went on to build.

90.7.6 Reading the ledger again

The 1993 debits have not all been paid.

The choices remain many, the learning tasks we set our agents remain simple next to the ones real firms solve, and no one has produced the theory of transition dynamics that the Eastern European reforms called for.

But the entries have moved.

Selection acquired a theory instead of a set of examples; non-convergence acquired an analytical characterization instead of a simulation; and the econometricians, offered a version of the models whose extra parameters the data could actually speak to, took them up.

The gap between the econometrician and the agents inside the model is still there.

It is narrower, and we now know a good deal about its width.

Part XIII

Phillips Curve Tradeoffs

CHAPTER

NINETYONE

THE RISE AND FALL OF U.S. INFLATION

Contents

- *The Rise and Fall of U.S. Inflation*
  - *Overview*
  - *Facts*
  - *The Phillips curve in the data*
  - *Two interpretations*
  - *Ignoring the Lucas Critique*
  - *A premature summary: triumph or vindication?*
  - *Data patterns after 1999*
  - *Exercises*

In addition to what's in Anaconda, this lecture will use the following libraries to download and filter macroeconomic data:

```
!pip install pandas_datareader
```

91.1 Overview

This is the first lecture in a suite based on Thomas Sargent's *The Conquest of American Inflation* [Sargent, 1999].

It combines chapters 1 and 2 of that book.

The suite asks a question about post-war U.S. macroeconomic history:

> If we take for granted that inflation is under the control of the Federal Reserve, how can we explain the rise of U.S. inflation into the 1970s and its abrupt fall under Paul Volcker in the early 1980s?

The essay evaluates two interpretations, both based on policy makers' *beliefs* about the Phillips curve.

In both stories, the Federal Reserve learns the natural-rate-of-unemployment theory from a combination of experience and *a priori* reasoning.

The stories differ in how that theory is cast:

- **The triumph of natural-rate theory.** Academic economists discovered the natural-rate hypothesis, taught that any inflation-unemployment tradeoff is temporary, and eventually persuaded policy makers to pursue low inflation.

- **The vindication of econometric policy evaluation.** Policy makers never abandoned the methods that Robert Lucas criticized in his famous Critique.
  - Recurrently re-estimating a Phillips curve and using it to choose a target, they were led toward lower inflation by the *data itself*, an adversely shifting empirical Phillips curve.

This lecture presents the facts that motivate both stories, sketches the two interpretations, and reviews the Lucas Critique that chapter 2 both invokes and modifies.

The remaining lectures in the suite build the models:

- *The Credibility Problem* — the one-period Kydland-Prescott credibility problem (chapter 3).
- *Credible Government Policies* — reputation in the repeated economy, and why the theory of credible policy replaces pessimism with agnosticism (chapter 4).
- *Adaptive Expectations and the Phelps Problem* — adaptive expectations and the Phelps problem (chapter 5).
- *Optimal Misspecified Beliefs* — equilibrium under optimal misspecified beliefs (chapter 6).
- *Self-Confirming Equilibria* — self-confirming equilibria (chapter 7).
- *Adaptive Learning and Escape Dynamics* — adaptive learning, escape dynamics, and simulated Volcker stabilizations (chapter 8).
- *Escaping Nash Inflation* — the escape dynamics characterized analytically ([Cho *et al.*, 2002]).
- *Priors, Escapes, and Learning Cycles* — how the government's prior about drifting coefficients shapes convergence, cycles, and escapes ([Sargent and Williams, 2005]).
- *The Lost Conquest: Fed Policy in the 2020s* — the same tools turned on the 2020s inflation and the Fed's slow response ([Sargent and Williams, 2025]).
- *Drifts and Volatilities* — an empirical postscript that fits a drifting-coefficient, stochastic-volatility VAR to the data and asks whether the Great Inflation was bad policy or bad luck ([Cogley and Sargent, 2005]).

91.1.1 A note on notation

The lectures follow the notation of the source each one is based on, and those sources do not agree with one another.

Rather than impose a single scheme and diverge from the papers a reader may want to consult, we keep each lecture faithful to its source and record the translations here.

object	symbol	where
inflation	y	*The Credibility Problem* – *Self-Confirming Equilibria*
	π	*Escaping Nash Inflation* onward
public's expected inflation	x	*The Credibility Problem*, *Adaptive Expectations and the Phelps Problem*
government's systematic inflation	x	*Escaping Nash Inflation*, *Priors, Escapes, and Learning Cycles*
natural rate of unemployment	U^*	*The Credibility Problem* – *Adaptive Learning and Escape Dynamics*
	u	*Escaping Nash Inflation*, *Priors, Escapes, and Learning Cycles*
Phillips-curve slope	θ	*The Credibility Problem* – *Escaping Nash Inflation*
government's beliefs	γ	*Adaptive Expectations and the Phelps Problem* – *Priors, Escapes, and Learning Cycles*
	θ	*The Lost Conquest: Fed Policy in the 2020s*, *Drifts and Volatilities*
discount factor	δ	*Credible Government Policies* – *Adaptive Learning and Escape Dynamics*
	β	*The Lost Conquest: Fed Policy in the 2020s*, *Drifts and Volatilities*
learning gain	λ, g_t	*Adaptive Expectations and the Phelps Problem*, *Adaptive Learning and Escape Dynamics*
	ε	*Escaping Nash Inflation*, *Priors, Escapes, and Learning Cycles*

Three collisions are worth flagging in advance, because a reader carrying symbols forward will otherwise be caught by them.

The letter x changes sides: in *The Credibility Problem* it is what the *public* expects, while from *Escaping Nash Inflation* onward it is what the *government* sets.

The two coincide in equilibrium, which is exactly what makes the switch easy to miss.

The letter θ is the slope of the Phillips curve in the early lectures and the government's whole belief vector in the last two.

And λ works four shifts: the Cagan–Friedman adaptation parameter in *Adaptive Expectations and the Phelps Problem*, a forgetting factor in *Adaptive Learning and Escape Dynamics*, a prior-tightening parameter in *Priors, Escapes, and Learning Cycles*, and a measured persistence root in *The Lost Conquest: Fed Policy in the 2020s*.

Let's start with some imports:

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import datetime
from pandas_datareader import data as web
from statsmodels.tsa.filters.bk_filter import bkfilter
```

Note

The figures in the next two sections reproduce the ones in chapters 1 and 2 of [Sargent, 1999], which were drawn from data available in the late 1990s.

We download the underlying series from FRED and restrict attention to the same historical window.

The section *Data patterns after 1999* then carries the most enlightening of these figures through to the present and asks what the additional quarter-century of data means for the two stories.

91.2 Facts

We begin with the single fact that the whole essay seeks to explain: the hump-shaped path of U.S. inflation since World War II.

We measure inflation by the annualized monthly change in the consumer price index (all items), smoothed with a 13-month centered moving average to remove seasonal and high-frequency noise.

```
start, end = datetime.datetime(1948, 1, 1), datetime.datetime(1999, 1, 1)

cpi = web.DataReader('CPIAUCNS', 'fred', start, end)['CPIAUCNS']

# annualized monthly inflation, then a 13-month centered moving average
inflation = 1200 * np.log(cpi).diff()
inflation_ma = inflation.rolling(13, center=True).mean()
```

```
fig, ax = plt.subplots(figsize=(9, 5))
ax.plot(inflation_ma, lw=1.2)
ax.axhline(0, color='k', lw=0.5)
ax.set_xlabel('year')
ax.set_ylabel('inflation (percent, annualized)')
ax.set_title('Figure 1.1: Monthly inflation, CPI all items, '
             '13-month centered moving average')
plt.show()
```

Inflation was low during the late 1950s and early 1960s, swept upward into the 1970s, and then fell abruptly with Volcker's stabilization in the early 1980s.

Any explanation that treats inflation as under the Federal Reserve's control must account for this rise and fall.

91.3 The Phillips curve in the data

Despite its disrepute in some academic and policy-making circles, the Phillips curve persists in U.S. data, and simple procedures detect it.

To coax the Phillips curve from the data, we follow the book in two ways.

First, we use the unemployment rate for a single demographic group — white men 20 years and over — rather than the aggregate rate, which is contaminated by slow-moving shifts in the demographic mix.

Second, we look at *business-cycle* frequencies, filtering out slowly moving components so that the eye can spot the inverse relationship between inflation and unemployment.

```
u = web.DataReader('LNS14000028', 'fred', start, end)['LNS14000028']  # white men 20+

data = pd.concat([inflation.rename('inflation'),
                  u.rename('unemployment')], axis=1).dropna()
data.head()
```

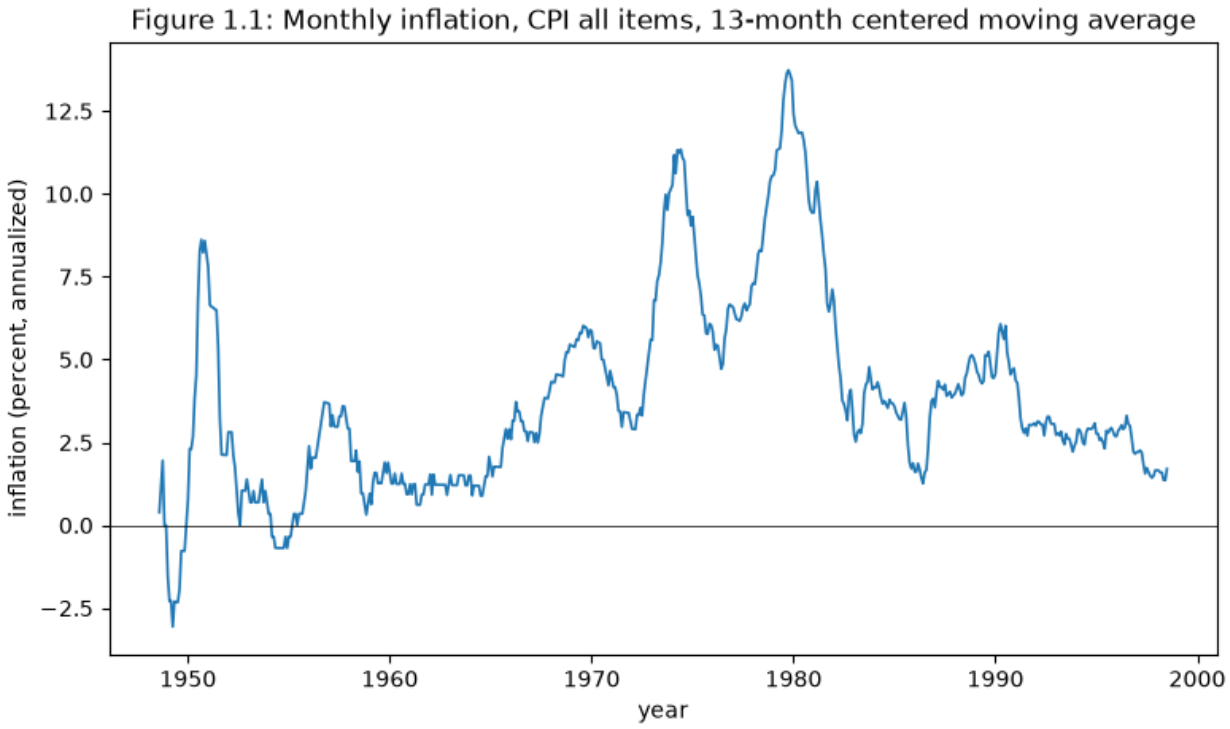


Fig. 91.1: Monthly CPI inflation, 13-month centered moving average, 1948-1999

```
            inflation  unemployment
DATE
1954-01-01   0.000000           3.6
1954-02-01   0.000000           3.8
1954-03-01   0.000000           4.1
1954-04-01  -4.469279           4.7
1954-05-01   4.469279           4.6
```

Figure 1.2 plots the two raw series together.

```
fig, ax = plt.subplots(figsize=(9, 5))
ax.plot(data.index, data['inflation'], 'C0', lw=1, label='inflation (CPI)')
ax.plot(data.index, data['unemployment'], 'C1:', lw=1.2,
        label='unemployment (white men 20+)')
ax.axhline(0, color='k', lw=0.5)
ax.set_xlabel('year')
ax.set_ylabel('percent')
ax.set_title('Figure 1.2: Monthly unemployment and inflation rates')
ax.legend()
plt.show()
```

To isolate the business-cycle relationship, we apply the finite-lag bandpass filter of Baxter and King [Baxter and King, 1999].

Following the book, we keep fluctuations with periods between 24 and 84 months and use a lead-lag truncation of 84 months.

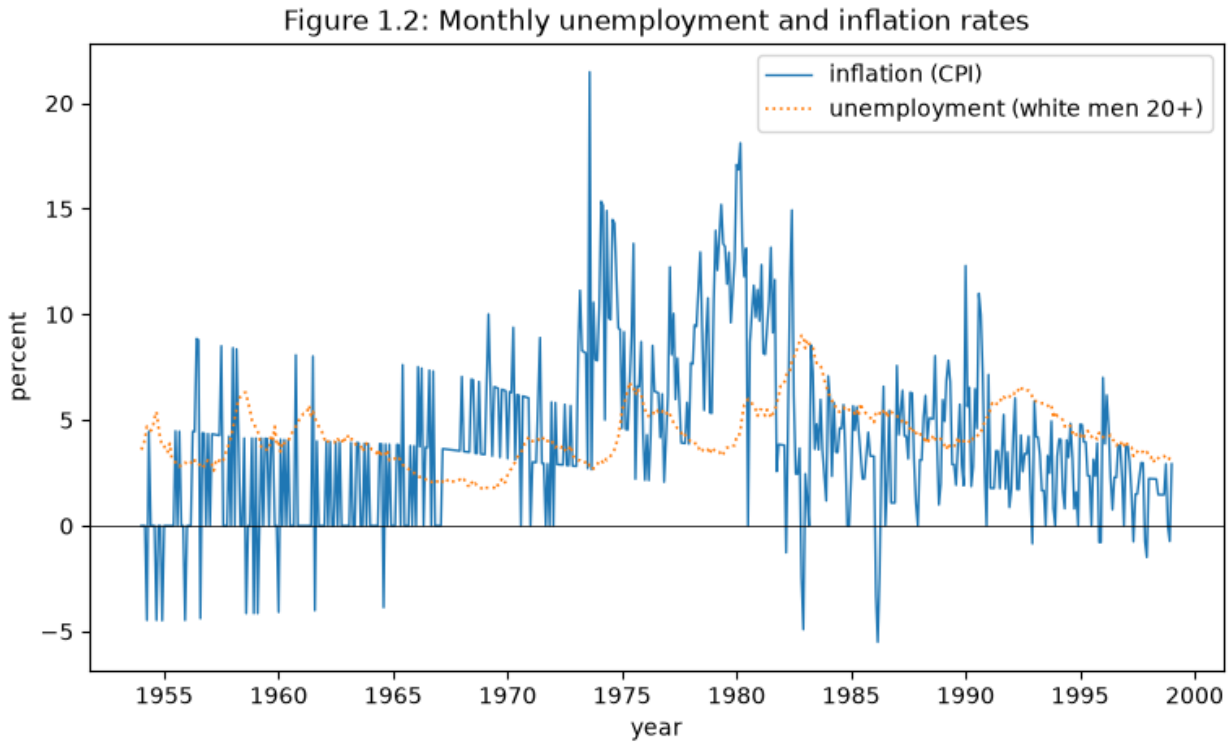


Fig. 91.2: Monthly unemployment (white men 20+) and inflation rates

```
# Baxter-King bandpass: periods between 24 and 84 months, truncation 84
bk = bkfilter(data, low=24, high=84, K=84)
bk.columns = ['inflation_cycle', 'unemployment_cycle']
```

```
fig, ax = plt.subplots(figsize=(9, 5))
ax.plot(bk.index, bk['inflation_cycle'], 'C0', lw=1, label='inflation')
ax.plot(bk.index, bk['unemployment_cycle'], 'C1:', lw=1.2,
        label='unemployment')
ax.axhline(0, color='k', lw=0.5)
ax.set_xlabel('year')
ax.set_ylabel('deviation from trend (percent)')
ax.set_title('Figure 1.3: Business-cycle components '
             '(Baxter-King bandpass filter)')
ax.legend()
plt.show()
```

The filtered components tend to move in opposite directions: a business-cycle Phillips curve.

We can see the tradeoff more directly in a scatter plot for the subperiod that interests us most, 1960-1982.

Figure 1.4 plots the raw series against each other, and Figure 1.5 the business-cycle components.

```
sub = slice('1960', '1982')

fig, axes = plt.subplots(1, 2, figsize=(12, 5))
```

(continues on next page)

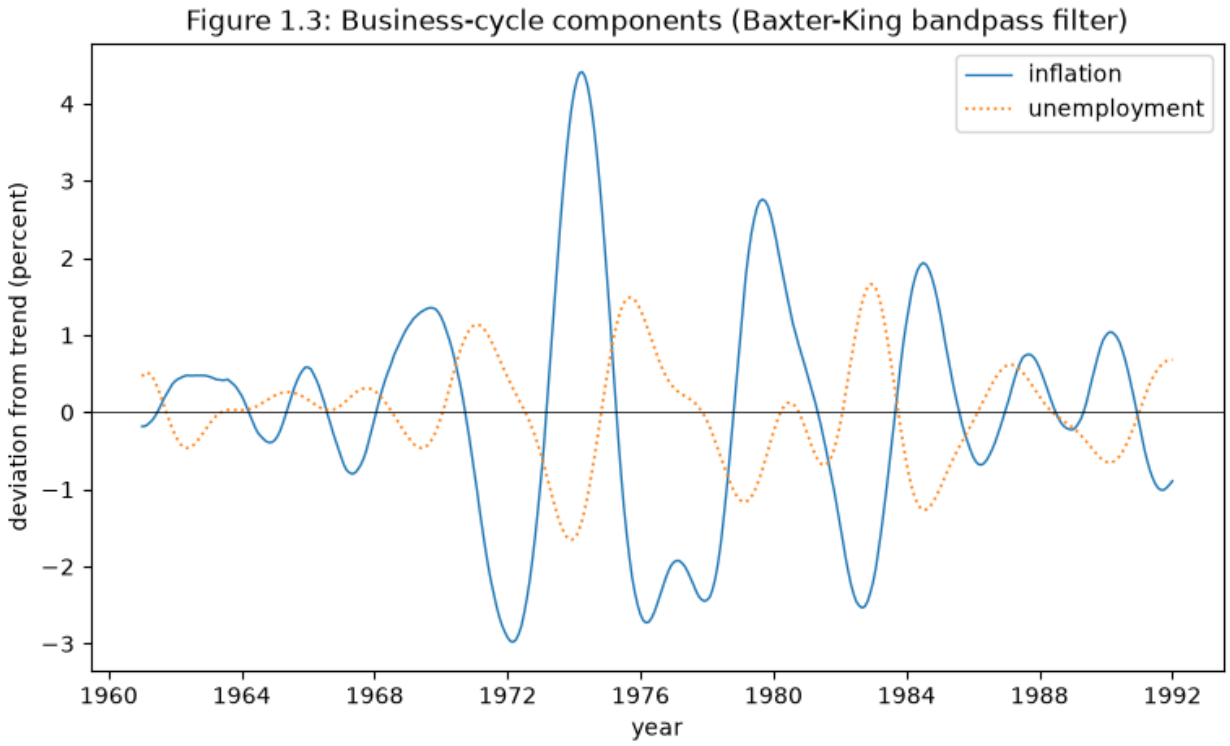


Fig. 91.3: Business-cycle components of inflation and unemployment, Baxter-King bandpass filter

(continued from previous page)

```
axes[0].scatter(data.loc[sub, 'inflation'],
                data.loc[sub, 'unemployment'], s=8, alpha=0.6)
axes[0].set_xlabel('inflation')
axes[0].set_ylabel('unemployment (white men 20+)')
axes[0].set_title('Figure 1.4: Raw series, 1960-1982')

axes[1].scatter(bk.loc[sub, 'inflation_cycle'],
                bk.loc[sub, 'unemployment_cycle'], s=8, alpha=0.6)
axes[1].set_xlabel('inflation (business-cycle component)')
axes[1].set_ylabel('unemployment (business-cycle component)')
axes[1].set_title('Figure 1.5: Business-cycle components, 1960-1982')

plt.tight_layout()
plt.show()
```

Focusing on the business-cycle components sharpens the apparent Phillips curve.

Figure 1.5 reveals **Phillips loops**: inflation and unemployment trace out counter-clockwise loops rather than a single stable curve, a signature of the shifting expectations that the natural-rate theory places at the center of the story.

Note

The book adjusts for demographic change by choosing a single unemployment series.

A broader definition of unemployment would inject additional low-frequency demographic components, which one

Fig. 91.4: Inflation against unemployment, 1960-1982: raw series and business-cycle components

might model with a unit-root process.

The essay instead puts a unit root into the inflation-unemployment process from a different source: the *drifting beliefs* of a monetary authority cut loose from the discipline of Bretton Woods.

91.4 Two interpretations

Both stories start from the same initial conditions — the history of inflation and unemployment, and the state of expectations, inherited by policy makers around 1960 — and both assume that the data conform to the natural-rate hypothesis, whether or not policy makers realized it.

91.4.1 The triumph of natural-rate theory

Adherence to the gold standard and then to Bretton Woods gave the U.S. low inflation and low expected inflation.

In 1960, Paul Samuelson and Robert Solow [Samuelson and Solow, 1960] found a Phillips curve in U.S. data and taught that it was *exploitable* — that policy could raise inflation to reduce unemployment.

Within a decade this recommendation was widely endorsed and implemented.

To everyone's dismay, the Phillips curve then shifted adversely: inflation rose, but unemployment on average did not fall.

Meanwhile Edmund Phelps [Phelps, 1967], Milton Friedman [Friedman, 1968], and Robert Lucas [Lucas, 1972] created and refined the concept of the natural rate of unemployment, which assigns a central role to expectations of inflation in locating the Phillips curve.

The natural-rate theory allowed only a *temporary* tradeoff and explained the adverse shifts; its rational expectations version implied that policy makers should ignore the temporary tradeoff and strive only for low inflation.

In this story, these ideas diffused from academics to policy makers and ultimately produced the lower inflation of the 1980s and 1990s.

Events were shaped by policy makers' beliefs — some false, others true — and the actions those beliefs inspired.

91.4.2 The vindication of econometric policy evaluation

The alternative story ascribes Volcker's conquest partly to the *success* of the very econometric and policy-making procedures that Lucas challenged.

Policy makers accepted the Samuelson-Solow Phillips curve as an exploitable tradeoff and adopted their methods for learning from data and deducing policy.

Recurrently, they re-estimated a distributed-lag Phillips curve and used it to reset a target inflation-unemployment pair.

Interpreted mechanically — without identifying expectations as the hidden state variable — the adversely shifting empirical Phillips curve eventually led policy makers to pursue lower inflation.

This story is told with an *adaptive* theory of policy that departs minimally from rational expectations.

It connects a sequence of ideas that the rest of the suite develops: drifting coefficients, self-confirming equilibria, least squares and other recursive learning algorithms, convergence of least squares learners to self-confirming equilibria, and recurrent dynamics along *escape routes* from those equilibria.

The key idea, taken from Christopher Sims [Sims, 1988], is that an adaptive model lets a government *learn* from its past attempts to exploit the Phillips curve, and eventually discover a version of the natural-rate hypothesis that instructs it to reduce inflation.

91.5 Ignoring the Lucas Critique

Chapter 2 of the book confronts the obvious objection to the vindication story: doesn't the Lucas Critique forbid the mechanical, exploitable Phillips curve on which it rests?

The essay resurrects the econometric and policy-evaluation procedures that Lucas decisively criticized [Lucas, 1976], and emphasizes a neglected aspect of his Critique: **drifting coefficients**.

91.5.1 The Critique

An econometric model is a collection of stochastic difference equations, some of which describe private agents' decision rules.

Econometric policy evaluation in the Tinbergen-Theil tradition holds those private decision rules *fixed* while the government optimizes its own rule against an objective function.

Lucas noted that if private agents solve intertemporal optimization problems, their decision rules *depend on* the government's rule.

By missing this dependence, the Tinbergen-Theil method mistranslates the government's preferences over outcomes into an ordering over decision rules, and so gives unreliable policy advice.

91.5.2 The appeal to drifting coefficients

Lucas conceded the impressive forecasting record of Keynesian models but argued that good forecasting is no evidence for the *invariance under intervention* that Tinbergen-Theil assumes.

He stressed that forecasters routinely adjusted the constant terms of key equations, and interpreted these adjustments as an adaptive-coefficients model in the spirit of Cooley and Prescott [Cooley and Prescott, 1973].

The intertemporal instability of estimated relationships — coefficient drift — undermined treating them as invariant to systematic changes in policy rules.

Yet Lucas left the drift *unexplained*, and neither the macroeconomic theory nor the rational expectations econometrics built after the Critique accounts for it: both focus on environments with time-invariant transition functions.

91.5.3 Parameter drift as a point of departure

The essay starts from parameter drift, treating it as a *smoking gun* — the key evidence that the government's beliefs about the economy, and hence its policy toward inflation, have evolved over time.

It builds a model from two components:

1. a Tinbergen-Theil theory of government decision making — the Phelps problem of *Adaptive Expectations and the Phelps Problem*; and
2. a drifting-coefficients econometric procedure for the government, featuring the constant adjustments that Lucas wrote about.

Within a **self-confirming equilibrium** (developed in *Self-Confirming Equilibria*), some of the force of the Lucas Critique vanishes.

Although the government's invariance assumption is wrong, it is not disappointed in outcomes, because those outcomes are statistically consistent with its beliefs.

In a self-confirming equilibrium the government's beliefs are correct *along the equilibrium path*, so its forecasts satisfy the same cross-equation restrictions as a rational expectations equilibrium; but relative to the fully structural models Lucas used, the government's model carries *fewer* free parameters — and precisely those missing parameters would be needed to represent regime changes.

To admit regime changes and drifting coefficients, convergence to a self-confirming equilibrium must be *resisted*.

The essay arrests convergence by replacing the government's least squares estimator with a constant-gain, adaptive-coefficients algorithm that overweights recent data — endowing the government with a suspicion that the environment is unstable.

This weakens the pull toward a self-confirming equilibrium and sustains dynamics along an escape route, along which regime changes occur.

Ironically, as we shall see in *Adaptive Learning and Escape Dynamics*, the procedures that *violate* the Lucas Critique can yield better outcomes than ones that respect it.

91.6 A premature summary: triumph or vindication?

The rest of this suite builds models.

Before we start, it is worth previewing where they lead and the tension they leave unresolved — a premature summary of the journey, adapted from the concluding chapter of [Sargent, 1999].

Everything can be organized around two *benchmark* models.

In the first, due to [Phelps, 1967], the public forms expectations *adaptively* while the government chooses policy *optimally*, taking the public's rule as given.

In the second, the rational-expectations natural-rate model, the public is *rational* while the government's policy is treated as *exogenous and arbitrary*.

Lucas recommended replacing the first benchmark with the second.

Coming to grips with the two stories drives us to propose models that make various *compromises* between these poles — and the remaining lectures are those compromises.

91.6.1 The road ahead

We begin, in *The Credibility Problem*, by imposing rationality on *both* sides.

The one-period [Kydland and Prescott, 1977] model delivers a pessimistic prediction — the high-inflation time-consistent (Nash) outcome — but a repeated-economy version of the theory of credible policy replaces that pessimism with *agnosticism*: so many outcomes become sustainable that the theory yields only weak predictions.

Credible Government Policies establishes this, computing the whole set of sustainable values with the recursive methods of [Abreu *et al.*, 1990] and exhibiting three quite different equilibria that deliver an identical payoff.

That weakness is the first reason to hesitate before declaring the triumph of natural-rate theory.

We then turn back from the Lucas Critique and start again from the Phelps benchmark, but with one change: the government's model of the private sector is no longer arbitrary — it is *fit to historical data*.

Varying the details of that fitting problem generates the rest of the suite:

- self-confirming equilibria (*Self-Confirming Equilibria*),
- equilibria with optimally *misspecified* forecasting functions (*Optimal Misspecified Beliefs*), and
- adaptive, "anticipated-utility" learning models (*Adaptive Learning and Escape Dynamics*, *Escaping Nash Inflation*, and *Priors, Escapes, and Learning Cycles*).

These adaptive models are a *disciplined* retreat from rational expectations, not an abandonment of it.

They carry no free parameters governing expectations; period by period they impose the same cross-equation restrictions as a rational expectations model; and — because a self-confirming equilibrium is the attractor of their *mean dynamics* — under tranquil conditions they converge back to it, and hence, on the equilibrium path, to rational expectations, satisfying a desideratum of [Kreps, 1998].

But, following [Sims, 1988], our real interest is in the *recurrent* dynamics that adaptation adds.

Suspecting that the Phillips curve is prone to wander, the government uses a constant-gain algorithm, which is the sensible choice when coefficients drift.

The payoff is a striking one: the adaptive models produce abrupt *stabilizations* of inflation that defy the inferior self-confirming outcome toward which the mean dynamics point.

These regime shifts arise not from any change in the government's procedures, nor from large shocks, but from *changes in beliefs created by the government's own econometrics* — the mathematics of escape routes in the space of approximating models.

91.6.2 The induction hypothesis: villain and hero

At the center of the story sits the **induction hypothesis** — the restriction that the weights on lagged inflation in an expectations equation sum to one, so that a permanently higher inflation rate is eventually fully expected.

It was built almost without comment into the adaptive expectations hypothesis of [Friedman, 1957] and [Cagan, 1956], and it was the basis of Solow's and Tobin's early tests of the natural-rate hypothesis [Solow, 1968, Tobin, 1968].

Cast as a *villain* in Lucas's Critique — a naive restriction that rational expectations does not imply — the induction hypothesis re-emerges as the *hero* of the adaptive models: activating it is exactly what makes the government's Phelps problem call for near-Ramsey, low-inflation policy.

The escape routes our simulations follow are precisely the paths along which the government's estimated model, by using a unit root to approximate a constant, stumbles into believing the induction hypothesis.

Wrestling with such approximation problems, with several models simultaneously in play, is what led [Sims, 1980] to call bounded rationality a *wilderness*, set apart from the tidy one-model world of rational expectations.

91.6.3 The reservation

Which story, then — triumph or vindication?

The contest is not rational expectations versus an alternative, because *both* stories selectively apply and withdraw from rational expectations.

And the vindication story, however well it fits, is an exercise in *positive* economics, not *normative* economics.

It is tempting to read its long stretches of near-Ramsey inflation as an endorsement of the adaptive policy-making procedures that produce them — but that temptation should be resisted.

For the same mean dynamics that permit a stabilization also guarantee that it is temporary: once beliefs drift close enough to the induction hypothesis, the mean dynamics begin to point *away* from it, back toward the region where the Phelps problem recommends resuscitating inflation.

The simulations contain long episodes that look like Paul Volcker — and others that look like Arthur Burns.

Theoretical work after [Kydland and Prescott, 1977], and [Rogoff, 1985], insists that durable low inflation must rest on *commitment mechanisms* that keep a monetary authority from choosing sequentially — not on the hope that an adaptive government, armed with an approximate model, will by chance eventually learn to do approximately the right thing.

So the book ends on a hope rather than a verdict: we *hope* that the triumph story is the right one — that policy makers have learned a correct rational expectations version of the natural-rate hypothesis and have found devices to commit themselves to low inflation.

Because if instead the vindication story is closer to the truth, then the same mean dynamics that carried inflation down are always waiting, eventually, to carry it back up.

The quarter-century of data since 1999 — a long, quiet *Great Moderation* interrupted by a sudden surge in 2021-2022 — is a running test of exactly this hope, and we turn to it next.

91.7 Data patterns after 1999

[Sargent, 1999] was written at the end of the long disinflation that began with Volcker.

We now have another quarter-century of data.

This section carries the book's most enlightening figures through to the present and asks what the new observations mean for the two stories.

Let's extend the sample to the latest available month.

```
end_recent = datetime.datetime(2026, 7, 1)

cpi_full = web.DataReader('CPIAUCNS', 'fred', start, end_recent)['CPIAUCNS']
u_full = web.DataReader('LNS14000028', 'fred', start, end_recent)['LNS14000028']

# the book's measure (annualized monthly change, 13-month centered MA)
inflation_full = 1200 * np.log(cpi_full).diff()
inflation_ma_full = inflation_full.rolling(13, center=True).mean()

# year-over-year inflation: smoother, and the measure usually quoted today
inflation_yoy = 100 * (cpi_full / cpi_full.shift(12) - 1)
```

91.7.1 The rise and fall, extended

Figure 1.1 showed inflation rising into the 1970s and falling under Volcker.

Extending it to the present adds three chapters the book could not see: the *Great Moderation* of low, stable inflation from the mid-1980s; a long spell near — and briefly below — zero after the 2008 financial crisis; and a sudden surge in 2021-2022 to the highest rate since 1981, followed by a rapid decline.

```
fig, ax = plt.subplots(figsize=(11, 5))
ax.plot(inflation_ma_full, lw=1)
ax.axhline(0, color='k', lw=0.5)
ax.axvspan(pd.Timestamp('1948-01-01'), pd.Timestamp('1999-01-01'),
           color='C0', alpha=0.06, label="the book's window")
for date, y, txt in [('1980-03-01', 13.7, '1970s\nacceleration'),
                     ('1983-06-01', 3.0, 'Volcker'),
                     ('2009-07-01', -1.5, '2009\ndeflation scare'),
                     ('2022-06-01', 8.9, '2021-22\nsurge')]:
    ax.annotate(txt, (pd.Timestamp(date), y), ha='center', fontsize=9,
                color='C3')
ax.set_xlabel('year')
ax.set_ylabel('inflation (percent, annualized)')
ax.set_title('Inflation, CPI all items, 13-month centered moving average, '
             '1948-2026')
ax.legend(loc='upper right')
plt.show()
```

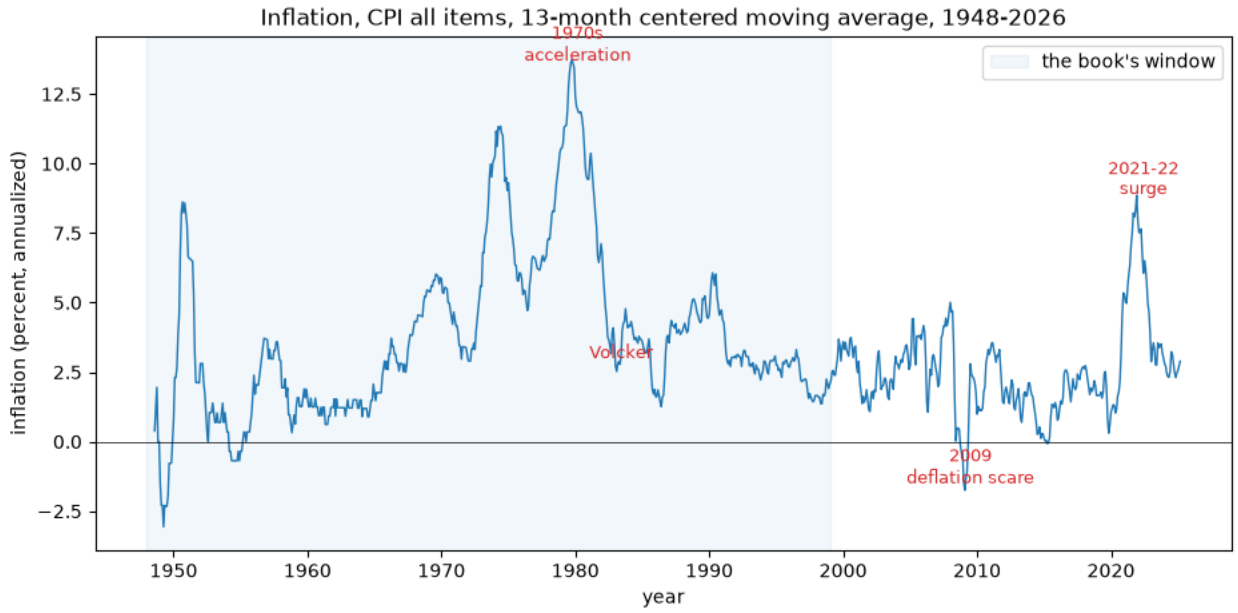


Fig. 91.5: Inflation extended to the present, with the book's window shaded

The hump that the book set out to explain is now one of *two*.

The second, in 2021-2022, is a genuinely new episode — a fast acceleration and an almost-as-fast disinflation, all compressed into about three years.

91.7.2 Unemployment and inflation to the present

Figure 1.2 plotted the two series together for the post-war period.

Extending it shows the two most dramatic macroeconomic events of the new data: the COVID unemployment spike of 2020 — briefly the highest since the Great Depression — and the inflation surge that followed.

```
recent = slice('1990', None)

fig, ax = plt.subplots(figsize=(11, 5))
ax.plot(inflation_yoy[recent], 'C0', lw=1, label='inflation (CPI, year-over-year)')
ax.plot(u_full[recent], 'C1:', lw=1.2, label='unemployment (white men 20+)')
ax.axhline(0, color='k', lw=0.5)
ax.set_xlabel('year')
ax.set_ylabel('percent')
ax.set_title('Unemployment and inflation, 1990-2026')
ax.legend()
plt.show()
```

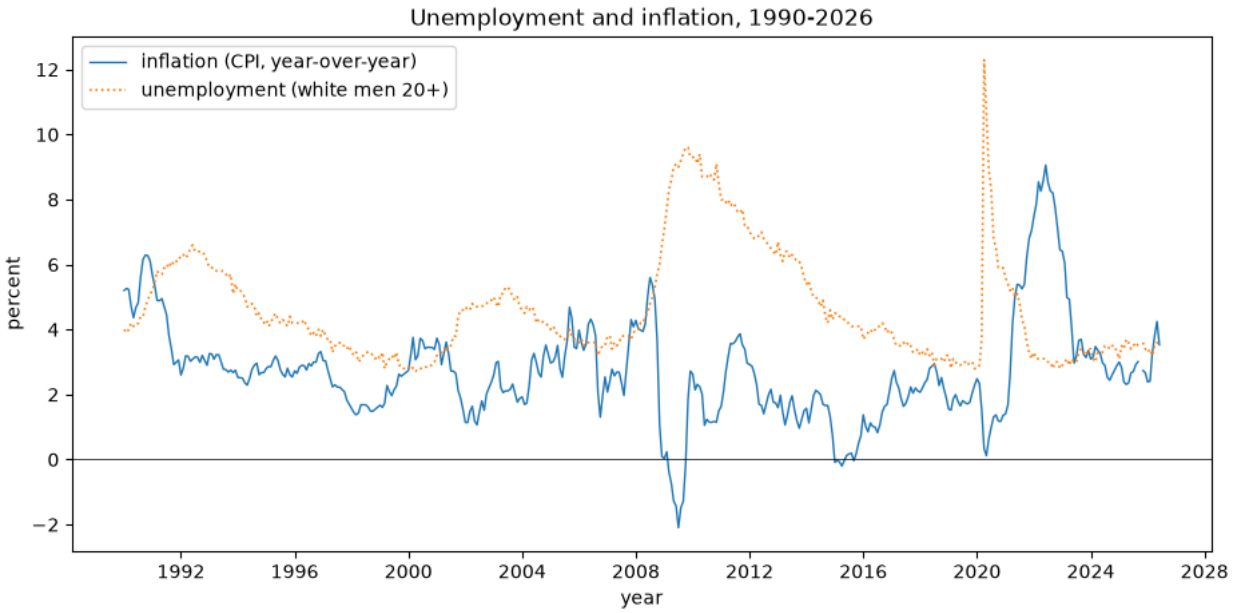


Fig. 91.6: Unemployment and inflation since 1990

Two features stand out.

From the mid-1990s to 2020, inflation stayed remarkably quiet even as unemployment swung widely — falling to historic lows in the late 1990s and 2019, and doubling in the 2008 recession.

Then, after the COVID spike, unemployment fell back quickly and inflation surged — a pattern with the fingerprints of a *supply* disturbance rather than the demand-driven tradeoff of the classic Phillips curve.

91.7.3 The Phillips curve across three eras

The book coaxed a Phillips curve from 1960-1982 data.

The most striking post-1999 pattern is how *unstable* the inflation-unemployment scatter has been across eras.

We split the sample into the book's acceleration era, the Great Moderation, and the post-2008 period, and plot inflation against unemployment in each.

```
scatter_data = pd.concat([inflation_yoy.rename('inflation'),
                          u_full.rename('unemployment')], axis=1).dropna()

eras = [('1960', '1983', '1960–1983 (acceleration)'),
        ('1984', '2007', '1984–2007 (Great Moderation)'),
        ('2008', None, '2008–2026 (crisis, COVID, surge)')]

fig, axes = plt.subplots(1, 3, figsize=(14, 4.5), sharex=True, sharey=True)
for ax, (lo, hi, title) in zip(axes, eras):
    era_data = scatter_data.loc[lo:hi]
    ax.scatter(era_data['unemployment'], era_data['inflation'], s=8, alpha=0.5)
    ax.axhline(0, color='k', lw=0.5)
    ax.set_xlabel('unemployment')
    ax.set_title(title, fontsize=10)
axes[0].set_ylabel('inflation (year-over-year)')
plt.tight_layout()
plt.show()
```

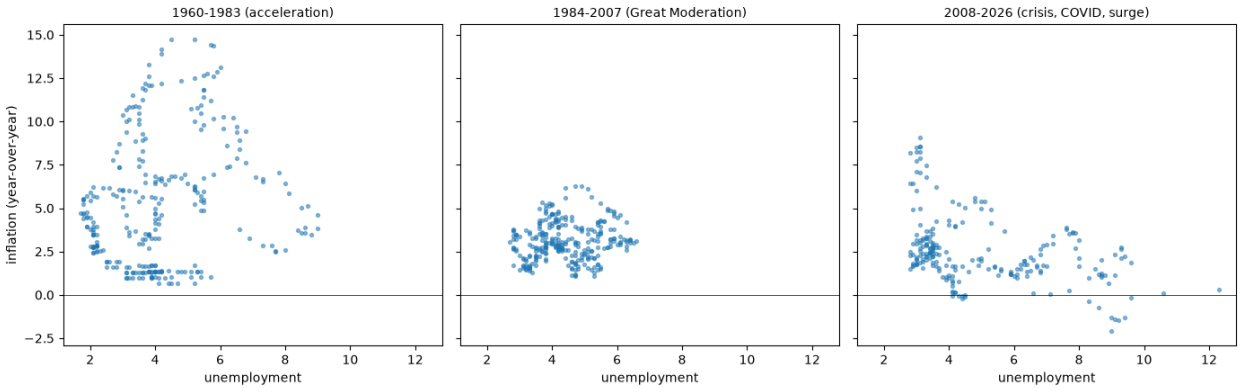


Fig. 91.7: The inflation-unemployment scatter across three eras

In 1960-1983 the points sprawl across a wide range of inflation rates — the era of shifting expectations and Phillips *loops*.

In 1984-2007 they collapse into a tight, low, nearly flat cloud — the Great Moderation, in which inflation barely responded to unemployment at all.

Since 2008 the relationship dissolves into a scatter dominated by two outliers: the COVID recession, with double-digit unemployment and still-low inflation, and the 2021-2022 surge, with high inflation and low unemployment.

Whatever the Phillips curve is, it is not a stable structural relationship — exactly the instability that motivates the book's account of *drifting beliefs*.

91.7.4 What the new data mean for the two stories

The additional quarter-century neither refutes nor confirms either story cleanly, but it sharpens both.

For the triumph of natural-rate theory. The Great Moderation reads as the triumph completed: with the natural-rate consensus entrenched and central-bank independence established, inflation stayed low and expectations stayed *anchored* for a generation.

In the book's language, the economy settled into a low-inflation self-confirming equilibrium and stayed there.

Even the 2021-2022 surge, on this reading, supports the triumph: once the Federal Reserve moved decisively, inflation came down quickly and long-run expectations never came unmoored — the low-inflation equilibrium survived a large shock.

For the vindication of econometric policy evaluation. The surge is also a reminder that the monetary authority's *model* can still mislead it: the widely-held 2021 view that inflation would be "transitory" was a model that the data falsified, and policy adjusted only after beliefs did.

The decade of near-zero inflation before 2020 — the apparently *flat* Phillips curve, with neither the "missing disinflation" of 2009-2013 nor the "missing inflation" of 2015-2019 fitting a stable curve — is precisely the kind of drifting empirical relationship whose changing slope and intercept the book's adaptive government tracks in real time.

Note

A caveat the book itself would insist on: its mechanisms assume that the *fundamentals* — the true data-generating process — are stable, so that all the action comes from the government's evolving beliefs.

The 2021-2022 episode involved genuine supply shocks (pandemic disruptions, energy prices), which lie outside that assumption.

Disentangling shifting beliefs from shifting fundamentals is exactly the identification problem that makes this history so hard, and so interesting.

The tools built in the rest of this suite — self-confirming equilibria, drifting coefficients, and escape dynamics — remain a natural language for asking the question the new data pose: will a credible low-inflation equilibrium keep re-anchoring after each shock, or can a sequence of surprises still set beliefs drifting, as they did after 1965?

The Lost Conquest: Fed Policy in the 2020s turns exactly these tools on the 2021-2022 surge, and asks why the Federal Reserve was so slow to respond, while the closing lecture *Drifts and Volatilities* asks the data directly whether the coefficients of post-war macroeconomic dynamics really drifted.

91.8 Exercises

Exercise 91.8.1

The claim that "focusing on the business-cycle components sharpens the apparent Phillips curve" can be made quantitative.

Compute the correlation between inflation and unemployment over 1960-1982 for

- the raw series (as in Figure 1.4), and
- the Baxter-King business-cycle components (as in Figure 1.5).

By how much does bandpass filtering sharpen the negative Phillips correlation?

Solution

```
corr_raw = data.loc[sub].corr().iloc[0, 1]
corr_cycle = bk.loc[sub].corr().iloc[0, 1]

print(f"raw correlation, 1960-1982        : {corr_raw:+.2f}")
print(f"business-cycle correlation, 1960-82: {corr_cycle:+.2f}")
```

```
raw correlation, 1960-1982        : +0.03
business-cycle correlation, 1960-82: -0.64
```

In the raw series the correlation is close to zero: the adverse *shifts* of the Phillips curve — the slowly moving expectational component that the natural-rate theory emphasizes — swamp the business-cycle tradeoff.

Once those low-frequency shifts are filtered out, a strong negative relationship emerges, confirming that the Phillips tradeoff operates at business-cycle frequencies.

CHAPTER

NINETYTWO

THE CREDIBILITY PROBLEM

Contents

- *The Credibility Problem*
  - *Overview*
  - *A one-period economy*
  - *Nash and Ramsey outcomes*
  - *Best response dynamics*
  - *Least squares learning converges to Nash*
  - *More foresight*
  - *Appendix: stochastic approximation*
  - *Exercises*

92.1 Overview

This lecture describes a basic expectational Phillips curve model of the sort studied by [Kydland and Prescott, 1977] and Robert Barro and David Gordon.

It is the first *modeling* lecture in a suite based on chapters of [Sargent, 1999], following *The Rise and Fall of U.S. Inflation*, which lays out the two stories and reviews the Lucas Critique.

Those chapters formalize

- the temptation to inflate that is unleashed by the discovery of a Phillips curve,
- the value of a commitment technology for resisting that temptation, and
- the fragility of reputational mechanisms as substitutes for commitment.

Throughout, rational expectations is the only equilibrium concept that we use.

Alterations in the *timing* of decisions by a government and a private sector induce different economies with distinct outcomes.

A government faces a **credibility problem** whenever it wishes to make decisions sooner than it must.

We shall compare outcomes under two timing protocols:

- In one, the government chooses inflation *before* the private sector sets its expectations, so the government takes into account how its choice shapes those expectations.
- In the other, the government chooses inflation *after* the private sector has set its expectations.

The deterioration in outcomes under the second protocol measures the loss from an inability to commit.

We also study two *out-of-equilibrium* dynamics that converge to the no-commitment (Nash) outcome:

- best response dynamics, and
- least squares learning.

Let's start with some standard imports:

```
import matplotlib.pyplot as plt
import numpy as np
```

92.2 A one-period economy

Although credibility problems are intrinsically dynamic, it is possible to describe them in a one-period model under different patterns of within-period timing.

This prepares the way for the multi-period analyses in subsequent lectures.

We describe a version of the one-period model of [Kydland and Prescott, 1977] in the terms used by Nancy Stokey [Stokey, 1989].

Let (U, y, x) be the unemployment rate, the inflation rate, and the public's expectation of the inflation rate, respectively.

The government's one-period payoff is

$$-\frac{1}{2}\left(U^2 + y^2\right). \tag{92.1}$$

Unemployment is determined by an expectations-augmented Phillips curve

$$U = U^* - \theta(y - x), \qquad \theta > 0. \tag{92.2}$$

The equation asserts that unemployment deviates from U^*, the natural rate, only when there is surprise inflation or deflation.

Substituting (92.2) into (92.1) expresses the government's payoff as a function $r(x, y)$:

$$r(x, y) = -\frac{1}{2}\left[\left(U^* - \theta(y - x)\right)^2 + y^2\right]. \tag{92.3}$$

We work with the following objects.

Rational expectations equilibrium: a triple (U, x, y) satisfying (92.2) and $y = x$.

Government (one-period) best response: given the public's expectation x, a decision rule $B(x) = \text{argmax}_y\, r(x, y)$ for setting y.

Nash equilibrium: a pair (x, y) satisfying (i) $x = y$, and (ii) $y = B(x)$.

Ramsey problem: $\max_y r(y, y)$.

The *Ramsey outcome* is the value of y that attains the maximum.

Best response dynamics: the dynamical system $y_t = B(y_{t-1})$, y_0 given.

A rational expectations equilibrium is a (U, x, y) triple that lies on the Phillips curve and for which private agents are not fooled, given x.

Substituting $x = y$ into the Phillips curve (92.2) shows that $U = U^*$ in any rational expectations equilibrium.

This identifies U^* as the natural rate of unemployment.

92.3 Nash and Ramsey outcomes

A Nash equilibrium builds in a best response by the government, taking the state of expectations x as given, together with a response $x = y$ by the market, i.e., rational expectations for a given y.

Maximizing (92.3) with respect to y for a fixed x gives the government's best response function

$$y = B(x) = \frac{\theta}{\theta^2 + 1} U^* + \frac{\theta^2}{\theta^2 + 1} x. \tag{92.4}$$

The Nash equilibrium sets $x = y = B(x)$, which gives

$$y^N = x^N = \theta U^*, \qquad U = U^*.$$

The Ramsey problem instead imposes $x = y$ *before* maximizing, so it maximizes $r(y, y) = -\frac{1}{2}(U^{*2} + y^2)$, which yields the Ramsey outcome

$$y^R = x^R = 0, \qquad U = U^*.$$

Thus $r(x^R, y^R) = -\frac{1}{2} U^{*2}$ while $r(x^N, y^N) = -\frac{1}{2}(1 + \theta^2) U^{*2}$.

Both outcomes deliver the natural rate U^*, but the Nash equilibrium delivers it with positive inflation and hence a strictly lower payoff.

Let's collect these formulas in a class.

```
class CredibilityModel:
    """
    A one-period expectational Phillips curve economy.
    """

    def __init__(self, θ=1.0, U_star=5.0):
        self.θ, self.U_star = θ, U_star

    def phillips(self, y, x):
        "Unemployment implied by inflation y and expected inflation x."
        return self.U_star - self.θ * (y - x)

    def r(self, x, y):
        "Government one-period payoff."
        U = self.phillips(y, x)
        return -0.5 * (U**2 + y**2)

    def B(self, x):
        "Government best response to expected inflation x."
        θ = self.θ
        return θ / (θ**2 + 1) * self.U_star + θ**2 / (θ**2 + 1) * x

    def nash(self):
        "Nash equilibrium inflation (= expected inflation)."
```

(continues on next page)

(continued from previous page)

```
        return self.θ * self.U_star

    def ramsey(self):
        "Ramsey inflation (= expected inflation)."
        return 0.0
```

```
cm = CredibilityModel()

y_N, y_R = cm.nash(), cm.ramsey()
print(f"Nash inflation      y^N = {y_N:.2f}")
print(f"Ramsey inflation    y^R = {y_R:.2f}")
print(f"Nash payoff         r   = {cm.r(y_N, y_N):.3f}")
print(f"Ramsey payoff       r   = {cm.r(y_R, y_R):.3f}")
```

```
Nash inflation      y^N = 5.00
Ramsey inflation    y^R = 0.00
Nash payoff         r   = -25.000
Ramsey payoff       r   = -12.500
```

The Nash payoff is worse than the Ramsey payoff.

The government would prefer the Ramsey outcome, but it cannot attain it without a technology for committing to $y = 0$ before the public forms its expectations.

The Nash equilibrium is supported by a timing protocol in which the government decides *after* the private sector sets its expectations.

The Ramsey outcome is associated with a timing protocol in which the government chooses *first*, knowing that in a rational expectations equilibrium the public's expectations will move with its choice because $y = x$.

92.3.1 A picture of the two outcomes

The government's indifference curves are circles centered at the origin in (U, y) space, because the payoff (92.1) depends only on $U^2 + y^2$.

For a given expectation x, the Phillips curve (92.2) is a downward-sloping line in (U, y) space.

The government's best response for y, given x, occurs where an indifference curve is tangent to the Phillips curve indexed by x.

```
fig, ax = plt.subplots(figsize=(7, 6))

U_grid = np.linspace(0, 12, 200)

# a family of Phillips curves indexed by expected inflation x
for x in [0.0, y_N / 2, y_N]:
    # U = U_star - θ (y - x)  =>  y = x + (U_star - U) / θ
    y_line = x + (cm.U_star - U_grid) / cm.θ
    ax.plot(U_grid, y_line, 'C0', lw=1)

# government indifference curves (circles U^2 + y^2 = const)
ξ = np.linspace(0, 2 * np.pi, 200)
for R in [np.hypot(cm.U_star, y_R), np.hypot(cm.U_star, y_N)]:
    if R > 0:
        ax.plot(R * np.cos(ξ), R * np.sin(ξ), 'C1--', lw=1)
```

(continues on next page)

(continued from previous page)

```
ax.plot(cm.U_star, y_N, 'ko')
ax.annotate('Nash', (cm.U_star, y_N), (cm.U_star + 0.4, y_N + 0.4))
ax.plot(cm.U_star, y_R, 'ko')
ax.annotate('Ramsey', (cm.U_star, y_R), (cm.U_star + 0.4, y_R + 0.4))

ax.set_xlim(0, 12)
ax.set_ylim(0, 10)
ax.set_xlabel('unemployment $U$')
ax.set_ylabel('inflation $y$')
plt.show()
```

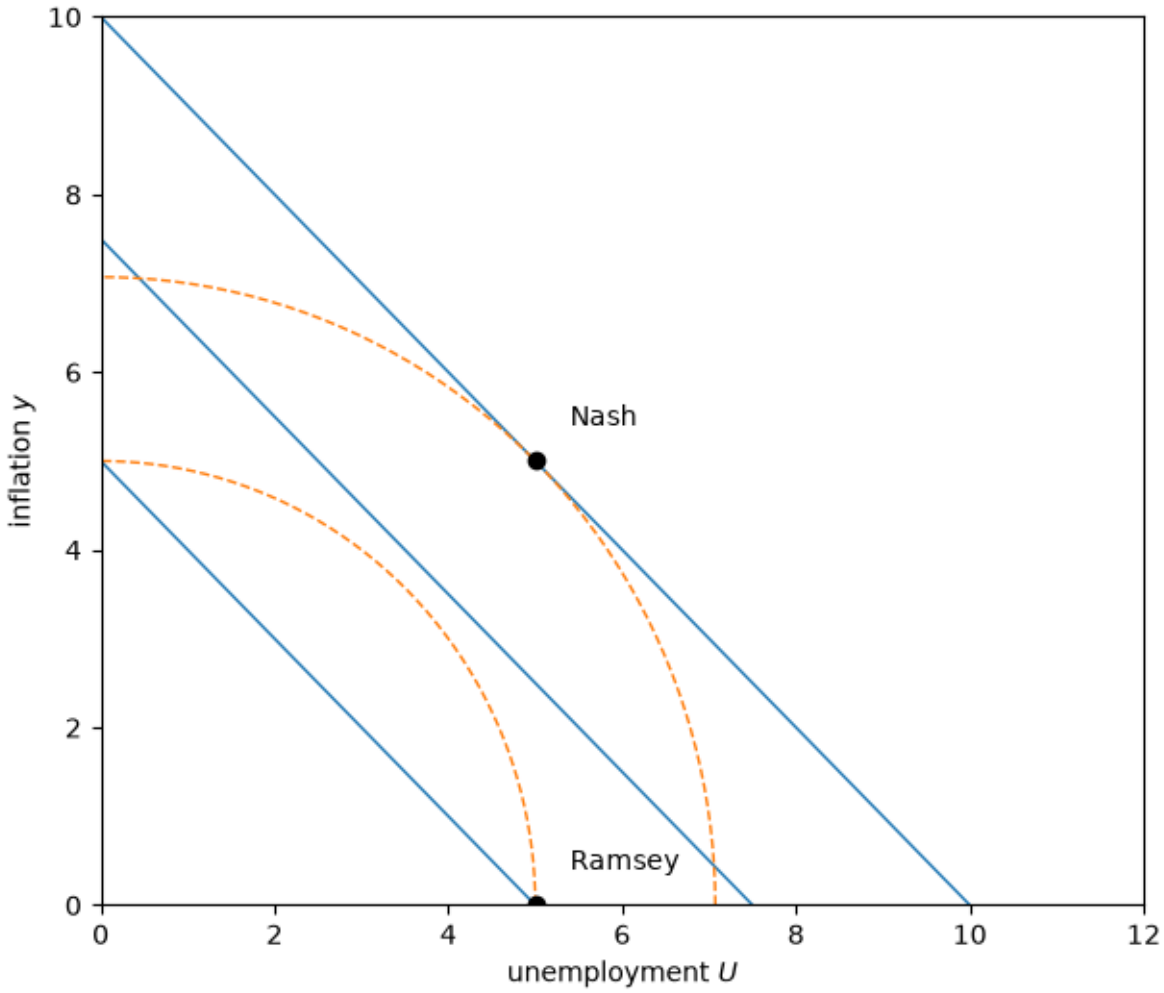


Fig. 92.1: Phillips curves indexed by expected inflation, government indifference curves, and the Nash and Ramsey outcomes

Solid lines are Phillips curves for expected inflation $x \in \{0, y^N/2, y^N\}$; dashed circles are government indifference curves.

The Nash outcome (U^*, y^N) lies on a larger circle (lower payoff) than the Ramsey outcome $(U^*, 0)$.

92.4 Best response dynamics

Best response dynamics convert the one-period model into a dynamic one by positing an adaptive mechanism in which expected inflation equals last period's inflation, $x_t = y_{t-1}$.

This leads to the dynamics

$$y_t = B(y_{t-1}), \qquad y_0 \text{ given.}$$

Because B is an affine map with slope $\theta^2/(\theta^2+1) \in (0,1)$, iterating on it converges to the fixed point $y^N = \theta U^*$ from any starting point.

Let's plot the best response function against the 45-degree line and simulate the dynamics.

```
def best_response_path(cm, y0, T=20):
    "Iterate y_{t+1} = B(y_t)."
    y = np.empty(T + 1)
    y[0] = y0
    for t in range(T):
        y[t + 1] = cm.B(y[t])
    return y

y_path = best_response_path(cm, y0=0.0, T=20)
```

```
fig, ax = plt.subplots(figsize=(6, 6))

x_grid = np.linspace(0, y_N + 1, 100)
ax.plot(x_grid, cm.B(x_grid), 'C0', label='$B(x)$')
ax.plot(x_grid, x_grid, 'k--', lw=1, label='45 degrees')

# cobweb of the best response dynamics
for t in range(len(y_path) - 1):
    ax.plot([y_path[t], y_path[t]], [y_path[t], y_path[t + 1]], 'C1', lw=0.8)
    ax.plot([y_path[t], y_path[t + 1]], [y_path[t + 1], y_path[t + 1]],
            'C1', lw=0.8)

ax.plot(y_N, y_N, 'ko')
ax.annotate('Nash', (y_N, y_N), (y_N - 1.5, y_N + 0.3))
ax.set_xlabel('$x$')
ax.set_ylabel('$B(x)$')
ax.legend()
plt.show()
```

Starting from $x = 0$ (the Ramsey inflation rate), the government sets $y = B(0) > 0$.

This provokes the public to raise its expectation, which leads the government to raise inflation further.

The limit of this process is the Nash outcome $y = x = y^N$, a self-confirming situation.

Thus best response dynamics converge to the Nash equilibrium and reinforce (U^*, y^N) as the prediction of the model without a commitment technology.

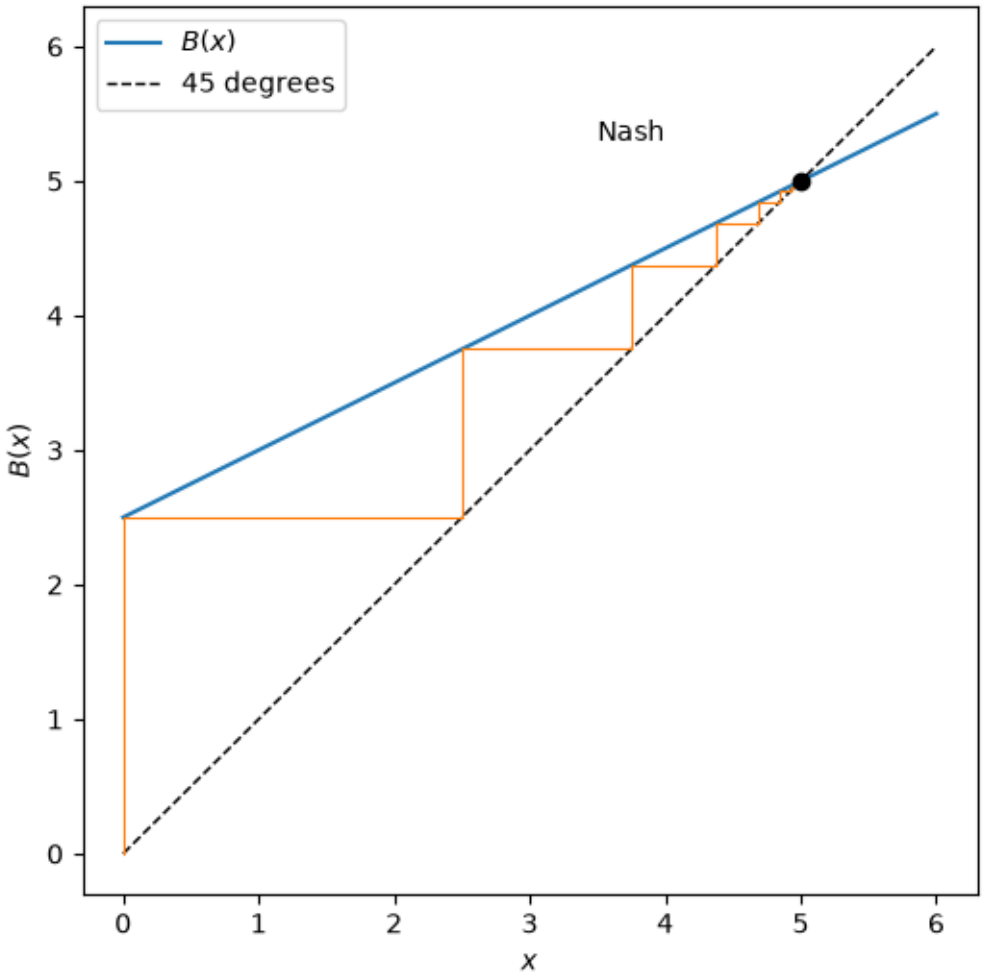


Fig. 92.2: Best response dynamics cobwebbing up to the Nash inflation rate

92.5 Least squares learning converges to Nash

A version of the best response dynamics also emerges from least squares learning.

Least squares learning plays a key role throughout this suite of lectures, and this simple example introduces analytical elements that reappear later in more complex settings.

Following [Bray, 1982], assume that expected inflation x_t is the average of past inflation rates,

$$x_t = \frac{1}{t-1}\sum_{s=1}^{t-1} y_s,$$

which can be represented recursively as

$$x_t = x_{t-1} + \frac{1}{t-1}(y_{t-1} - x_{t-1}), \qquad x_1 = 0. \tag{92.5}$$

Actual inflation is a disturbed version of the best response mapping evaluated at x_t,

$$y_t = B(x_t) + \eta_t, \tag{92.6}$$

where η_t is an IID mean-zero term that represents the government's imperfect control of inflation.

Substituting (92.6) into (92.5) gives the stochastic recursion

$$x_t = x_{t-1} + \frac{1}{t-1}\left[B(x_{t-1}) - x_{t-1} + \eta_t\right]. \tag{92.7}$$

By the theory of stochastic approximation, the limiting behavior of x_t is described by the associated ordinary differential equation (ODE)

$$\frac{dx}{dt} = B(x) - x. \tag{92.8}$$

The rest point of this ODE satisfies $x = B(x)$, which is the Nash equilibrium inflation rate $x = \theta U^*$.

Because the map B is affine, the ODE is linear with slope

$$\mathcal{M} = \frac{d}{dx}\left(B(x) - x\right) = B'(x) - 1 = -\frac{1}{\theta^2 + 1}.$$

Since $\mathcal{M} < 0$, the ODE is stable about its rest point, and theorems of [Marcet and Sargent, 1989] give conditions under which x_t converges to y^N globally.

Let's simulate the recursion (92.7) and confirm convergence to the Nash outcome.

```
def ls_learning(cm, T=2000, σ_η=1.0, seed=0):
    "Simulate least squares learning of expected inflation."
    rng = np.random.default_rng(seed)
    x = np.empty(T + 1)
    y = np.empty(T + 1)
    x[0] = 0.0
    y[0] = cm.B(x[0])
    for t in range(1, T + 1):
        η = σ_η * rng.standard_normal()
        y[t] = cm.B(x[t - 1]) + η
        gain = 1.0 / t                # the 1/(t-1) gain of equation (7), reindexed
        x[t] = x[t - 1] + gain * (cm.B(x[t - 1]) - x[t - 1] + η)
    return x, y
```

```
x, y = ls_learning(cm, T=2000, σ_η=1.0)

fig, ax = plt.subplots(figsize=(9, 5))
ax.plot(x, 'C0', lw=1, label='expected inflation $x_t$')
ax.axhline(y_N, color='k', ls='--', lw=1, label='Nash $y^N$')
ax.axhline(y_R, color='C2', ls=':', lw=1, label='Ramsey $y^R$')
ax.set_xlabel('$t$')
ax.set_ylabel('inflation')
ax.legend()
plt.show()
```

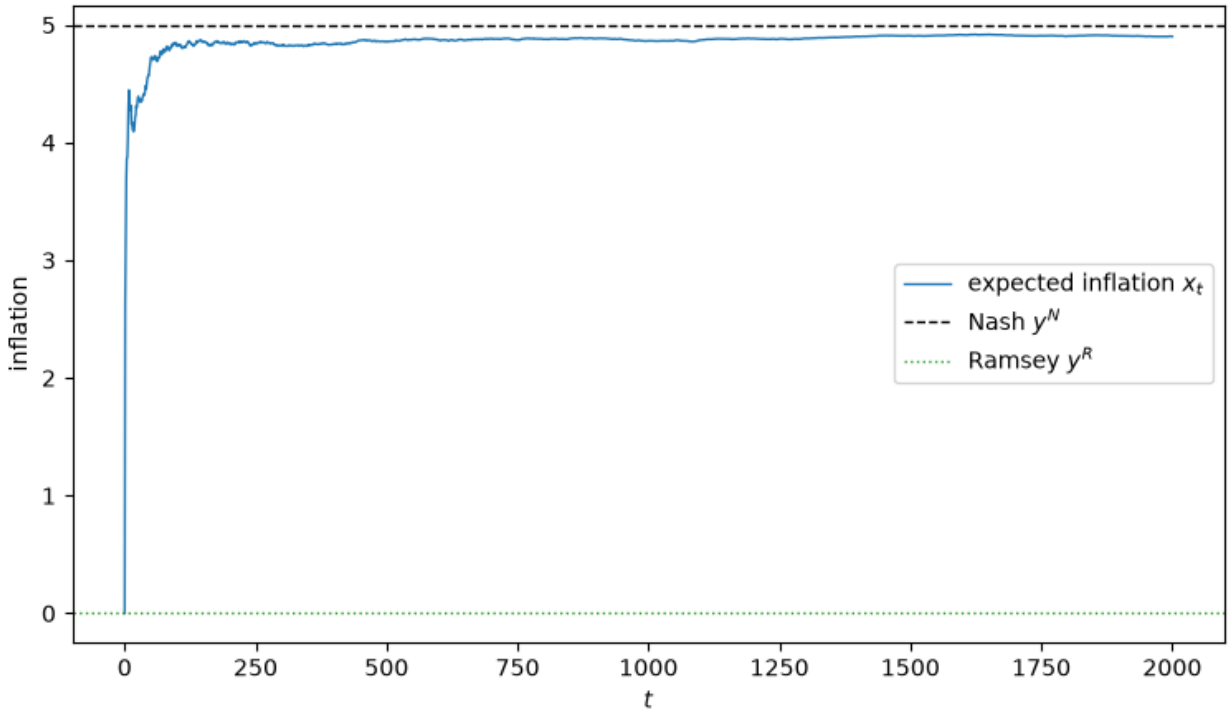


Fig. 92.3: Least squares learning of expected inflation converges to the Nash outcome

The least squares dynamics confirm the pessimism of the best response dynamics.

Given an initial condition in the form of a low, gold-standard value of x, the best response or least squares dynamics can explain an *acceleration* of inflation.

But they cannot explain a Volcker-style *stabilization* that brings inflation back down.

Later lectures reformulate versions of least squares learning in ways designed to moderate this pessimism.

92.6 More foresight

Best response and least squares learning are out-of-equilibrium dynamics tacked onto a one-period economy.

They force all movement through expectation formation: in choosing inflation, the government forgets that the economy lasts more than one period.

Better outcomes can occur if the government plans for the future.

Subsequent lectures describe three ways of modeling foresight, which impute varying amounts of rationality and predict different qualities of outcomes:

1. A reputational approach that attributes rational expectations to both the government and the public, the subject of *Credible Government Policies*.
   - Many outcomes are sustainable, ranging from repetition of the Ramsey outcome to paths worse than repetition of the Nash outcome, a multiplicity that turns out to be the theory's chief lesson.
2. An approach that keeps the government rational but gives the public *adaptive* expectations in the original Cagan-Friedman sense, the subject of *Adaptive Expectations and the Phelps Problem*.
   - Depending on a comparison between a discount factor and an adaptation parameter, this setup can improve outcomes and possibly sustain repetition of the Ramsey outcome.
3. An approach that attributes adaptive behavior to both the government and the public, the subject of *Optimal Misspecified Beliefs* and *Self-Confirming Equilibria*.

92.7 Appendix: stochastic approximation

Here we sketch why the ODE (92.8) governs the tail behavior of the stochastic difference equation (92.7).

The argument, due to [Kushner and Clark, 1978], has two key components: a shift in time scale and a liberal application of averaging.

Let $\{a_n\}_{n\geq 0}$ be a positive sequence of gains satisfying

$$\lim_{n\to\infty} a_n = 0, \qquad \sum_n a_n = +\infty, \qquad \sum_n a_n^2 < +\infty.$$

The choice $a_n = 1/(n+1)$ satisfies these assumptions.

Rewrite the recursion as

$$x_{n+1} = x_n + a_n \left[B(x_n) - x_n + \eta_n \right], \tag{92.9}$$

where η_n is IID with mean zero and finite variance.

Introduce the transformed time scale $t_0 = 0$, $t_n = \sum_{i=0}^{n-1} a_i$, and interpolate the discrete sequence $\{x_n\}$ into a continuous-time process $x^0(t)$.

Kushner and Clark show that on this transformed time scale the interpolated process is well approximated by the integral equation

$$x^0(t) = x^0(0) + \int_0^t \left[B(x^0(s)) - x^0(s) \right] ds + R(t),$$

where the approximation error $R(t)$ has two components — one from approximating a distributed lag in $B(x) - x$ by an integral, and one from a distributed lag in η_s.

Studying a sequence of left-shifted versions of the process, they show that both error components can be driven to zero as $n \to \infty$.

The key step for the noise component is to note that the relevant partial sums form a martingale with variance proportional to $\sum_i a_i^2$, which converges because $\sum_i a_i^2 < \infty$.

The remaining error is sent to zero because $a_i \to 0$ shrinks the mesh of the Riemann sum used to approximate the integral.

In the limit, the stochastic difference equation (92.9) shares the behavior of the non-stochastic ODE

$$\frac{d}{dt}\tilde{x}(t) = B(\tilde{x}(t)) - \tilde{x}(t),$$

which is said to describe the *mean dynamics* of the original system.

Later lectures study systems like (92.9) in which a_i does *not* approach zero as i grows — so-called constant gain algorithms.

The mean dynamics (92.8) and these constant-gain algorithms become the central tools of *Adaptive Learning and Escape Dynamics*, *Escaping Nash Inflation*, and *Priors, Escapes, and Learning Cycles*, where the scalar expectation x studied here grows into a whole vector of drifting Phillips-curve coefficients.

92.8 Exercises

Exercise 92.8.1

The convergence rate of least squares learning depends on the slope $\mathcal{M} = -1/(\theta^2 + 1)$ of the associated ODE.

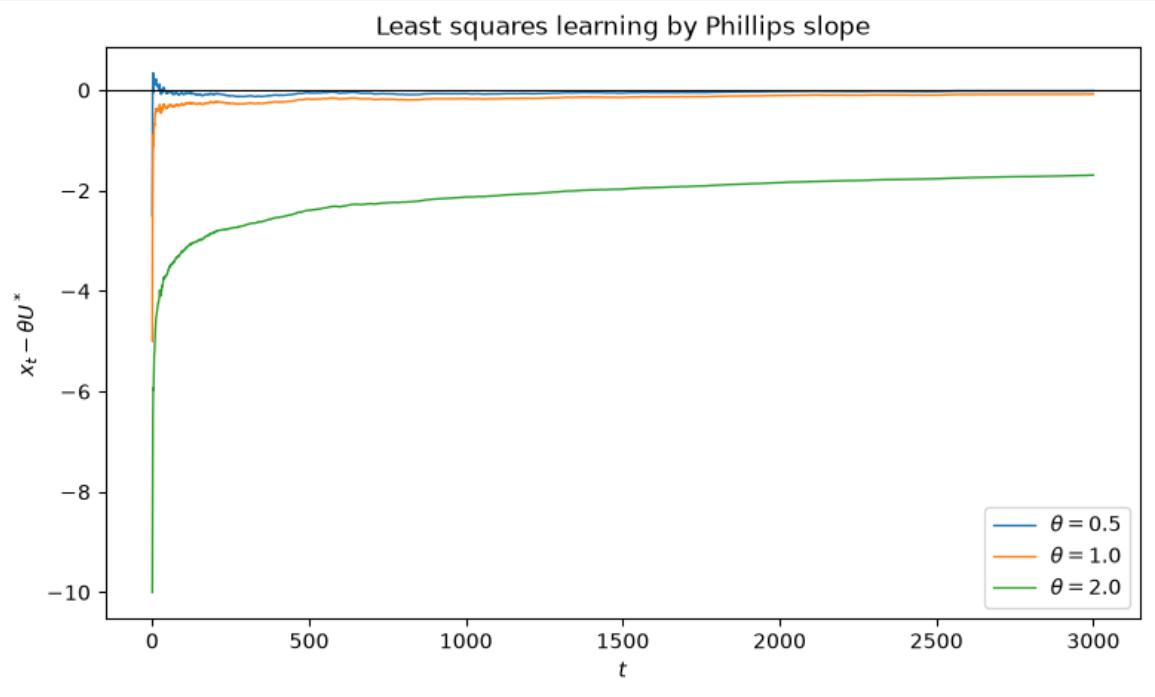
A necessary condition for convergence at the usual $\sqrt{t}$ rate is $\mathcal{M} < -1/2$, which requires $\theta < 1$.

Simulate least squares learning for $\theta \in \{0.5, 1.0, 2.0\}$ (holding U^* fixed) and compare how quickly x_t settles down near its Nash value θU^*.

Plot the three paths of $x_t - \theta U^*$ on one figure.

Solution

```
fig, ax = plt.subplots(figsize=(9, 5))

for θ in [0.5, 1.0, 2.0]:
    cm_θ = CredibilityModel(θ=θ, U_star=5.0)
    x, _ = ls_learning(cm_θ, T=3000, σ_η=1.0, seed=1)
    ax.plot(x - cm_θ.nash(), lw=1, label=rf'$\theta = {θ}$')

ax.axhline(0, color='k', lw=0.8)
ax.set_xlabel('$t$')
ax.set_ylabel('$x_t - \\theta U^*$')
ax.set_title('Least squares learning by Phillips slope')
ax.legend()
plt.show()
```



Smaller θ (a steeper $\mathcal{M}$) produces faster and tighter convergence to the Nash inflation rate.

Larger θ leaves x_t wandering more persistently around its limit.

CHAPTER

NINETYTHREE

CREDIBLE GOVERNMENT POLICIES

Contents

- *Credible Government Policies*
  - *Overview*
  - *The repeated economy*
  - *Recursive strategies and promised values*
  - *The Abreu–Pearce–Stacchetti method*
  - *The best and the worst equilibrium*
  - *Examples of recursive equilibria*
  - *Multiplicity*
  - *Numerical example*
  - *Interpretations*
  - *Exercises*

93.1 Overview

The Credibility Problem left the government in a trap.

Without a technology for committing itself, a government that re-optimizes every period ends at the Nash outcome, inflating at rate θU^* and getting nothing for it.

That was a one-period story, and it invited an obvious objection: a government that expects to face the same public again tomorrow has a *reputation* to protect.

This lecture, following chapter 4 of [Sargent, 1999], takes the objection seriously.

We repeat the Kydland–Prescott economy forever, let the government's current action depend on the whole history of outcomes, and ask which outcome paths can be supported as **subgame perfect equilibria**.

The answer is not the one the objection anticipated.

Reputation does not rescue the Ramsey outcome, and it does not confirm the Nash outcome either.

It delivers a *continuum* of equilibrium values, some better than Nash and some considerably worse, with no principle inside the model to select among them.

That is the conclusion this lecture exists to establish, and Sargent states it in one sentence:

> My conclusion is that the multiplicity of credible plans replaces pessimism with agnosticism.

It matters for the argument of the whole suite.

The Rise and Fall of U.S. Inflation rests part of its case against the *triumph of natural-rate theory* on exactly this weakness: a theory with so many equilibria makes few predictions, so a story that depends on policy makers having learned the right one is doing work the theory cannot do for it.

93.1.1 What we build

The machinery is the recursive method of Abreu, Pearce, and Stacchetti [Abreu, 1988, Abreu *et al.*, 1990], which computes not an optimal *value* but the whole *set* of equilibrium values.

Ordinary dynamic programming iterates an operator that maps continuation *values* into values.

APS iterate an operator that maps *sets* of continuation values into sets of values, and the set of subgame perfect equilibrium values is its largest fixed point.

Note

Making a promised value into a state variable, and then doing dynamic programming in that state, is the technique sometimes called **dynamic programming squared**.

It is the organizing idea of the QuantEcon lectures on Stackelberg plans and of the recursive treatments of Ramsey problems in [Ljungqvist and Sargent, 2018].

This lecture is a compact instance: the state is a promised value, and the object we compute is a set.

We then use the machinery three ways.

We construct particular equilibria by guess-and-verify — infinite repetition of Nash, infinite repetition of something better, and Abreu's stick-and-carrot, which is *worse* than Nash.

We compute the best and worst equilibrium values from two small programming problems, and check them against a direct iteration of the APS operator.

And we exhibit three quite different equilibria that all attain the same worst value, which is the sharpest illustration of how little the equilibrium concept pins down.

Note

Sargent's own advice to the reader is worth passing on: this chapter is difficult if you have not seen the theory before, and a reader willing to accept its verdict — that the theory of credible policy yields agnosticism rather than a prediction — can go straight to *Adaptive Expectations and the Phelps Problem* without losing the thread of the argument.

Let's start with our imports:

```
import matplotlib.pyplot as plt
import numpy as np
from typing import NamedTuple
```

93.2 The repeated economy

The one-period economy is the one from *The Credibility Problem*.

Let (U, y, x) be unemployment, inflation, and the public's expectation of inflation.

Unemployment obeys the expectations-augmented Phillips curve $U = U^* - \theta(y - x)$, and the government's one-period payoff, written as a function of (x, y), is

$$r(x, y) = -\frac{1}{2}\left[(U^* - \theta(y - x))^2 + y^2\right]. \tag{93.1}$$

The government's one-period best response to an expectation x is

$$B(x) = \frac{\theta\,(U^* + \theta x)}{\theta^2 + 1}, \tag{93.2}$$

the Nash outcome is $y^N = \theta U^*$, and the Ramsey outcome is $y^R = 0$.

Two things are new.

First, the economy repeats for $t = 1, 2, \ldots$, and the government ranks outcome paths $(x, y) = \{x_t, y_t\}_{t=1}^{\infty}$ by

$$V^g(x, y) = (1 - \delta)\sum_{t=1}^{\infty} \delta^{t-1} r(x_t, y_t), \qquad \delta \in (0, 1). \tag{93.3}$$

The factor $(1 - \delta)$ puts V^g in the same units as a one-period payoff, so values and one-period returns can be compared directly.

Second, inflation is confined to a bounded interval $Y = [0, y^{\#}]$.

The lower bound is the Ramsey outcome; the upper bound $y^{\#}$ is what makes the government's problem non-trivial, as we shall see when we look for the *worst* equilibrium.

```
class Model(NamedTuple):
    θ: float = 1.25          # slope of the Phillips curve
    U_star: float = 5.5      # natural rate of unemployment
    y_max: float = 10.0      # y^#, the highest admissible inflation rate
    δ: float = 0.95          # discount factor


def r(m, x, y):
    "One-period government payoff when the public expects x and inflation is y."
    U = m.U_star - m.θ * (y - x)
    return -0.5 * (U**2 + y**2)


def B(m, x):
    "The government's one-period best response to expected inflation x."
    return m.θ * (m.U_star + m.θ * x) / (m.θ**2 + 1)


def y_nash(m):
    return m.θ * m.U_star
```

Two payoff schedules do all the work below, both read as functions of a single inflation rate y that the public has come to expect.

The first is the **rational-expectations payoff** $r(y, y)$: what the government gets if it delivers the inflation the public expects.

The second is the **deviation payoff** $r(y, B(y))$: what it gets if the public expects y but the government yields to temptation and best-responds.

Both have closed forms worth recording, because they explain everything that follows:

$$r(y, y) = -\frac{1}{2}\left(U^{*2} + y^2\right), \qquad r(y, B(y)) = -\frac{\left(U^* + \theta y\right)^2}{2\left(1 + \theta^2\right)}. \tag{93.4}$$

```
def r_keep(m, y):
    "Payoff from delivering the expected inflation rate: r(y, y)."
    return -0.5 * (m.U_star**2 + y**2)


def r_cheat(m, y):
    "Payoff from best-responding to an expectation of y: r(y, B(y))."
    return -(m.U_star + m.θ * y)**2 / (2 * (1 + m.θ**2))


m = Model()
ys = np.linspace(0, m.y_max, 7)
print("closed forms agree with the definitions:",
      np.allclose(r_keep(m, ys), r(m, ys, ys)),
      np.allclose(r_cheat(m, ys), r(m, ys, B(m, ys))))
```

```
closed forms agree with the definitions: True True
```

Both schedules fall as y rises, but at different rates, and they cross exactly at the Nash rate, where the temptation to deviate vanishes because $B(y^N) = y^N$.

```
grid = np.linspace(0, m.y_max, 400)
yN = y_nash(m)
v_R, v_N = r_keep(m, 0.0), r_keep(m, yN)
v_lo = r_cheat(m, m.y_max)

fig, ax = plt.subplots(figsize=(7.5, 5.5))
ax.plot(grid, r_keep(m, grid), 'C0', lw=1.6, label='$r(y, y)$: deliver what is␣
↪expected')
ax.plot(grid, r_cheat(m, grid), 'C1', lw=1.6, label='$r(y, B(y))$: deviate')
ax.plot(0.0, v_R, 'ko', ms=6)
ax.annotate('$v^R$', (0.0, v_R), textcoords='offset points', xytext=(8, 4))
ax.plot(yN, v_N, 'ko', ms=6)
ax.annotate('$v^N$', (yN, v_N), textcoords='offset points', xytext=(8, 4))
ax.plot(m.y_max, v_lo, 'ko', ms=6)
ax.annotate('$v_{min}$', (m.y_max, v_lo), textcoords='offset points',
            xytext=(-34, 4))
ax.axvline(yN, color='k', lw=0.6, ls=':')
ax.set_xlabel('inflation rate $y$ expected by the public')
ax.set_ylabel('one-period payoff')
ax.set_title('Figure 4.1: the two payoff schedules and the worst equilibrium value')
ax.legend(loc='lower left')
plt.show()
```

The vertical gap between the curves is the government's one-period temptation to inflate less than expected.

It is zero at y^N, and it widens as expected inflation rises above the Nash rate — which is why the *worst* equilibrium will turn out to live at the top of Y.

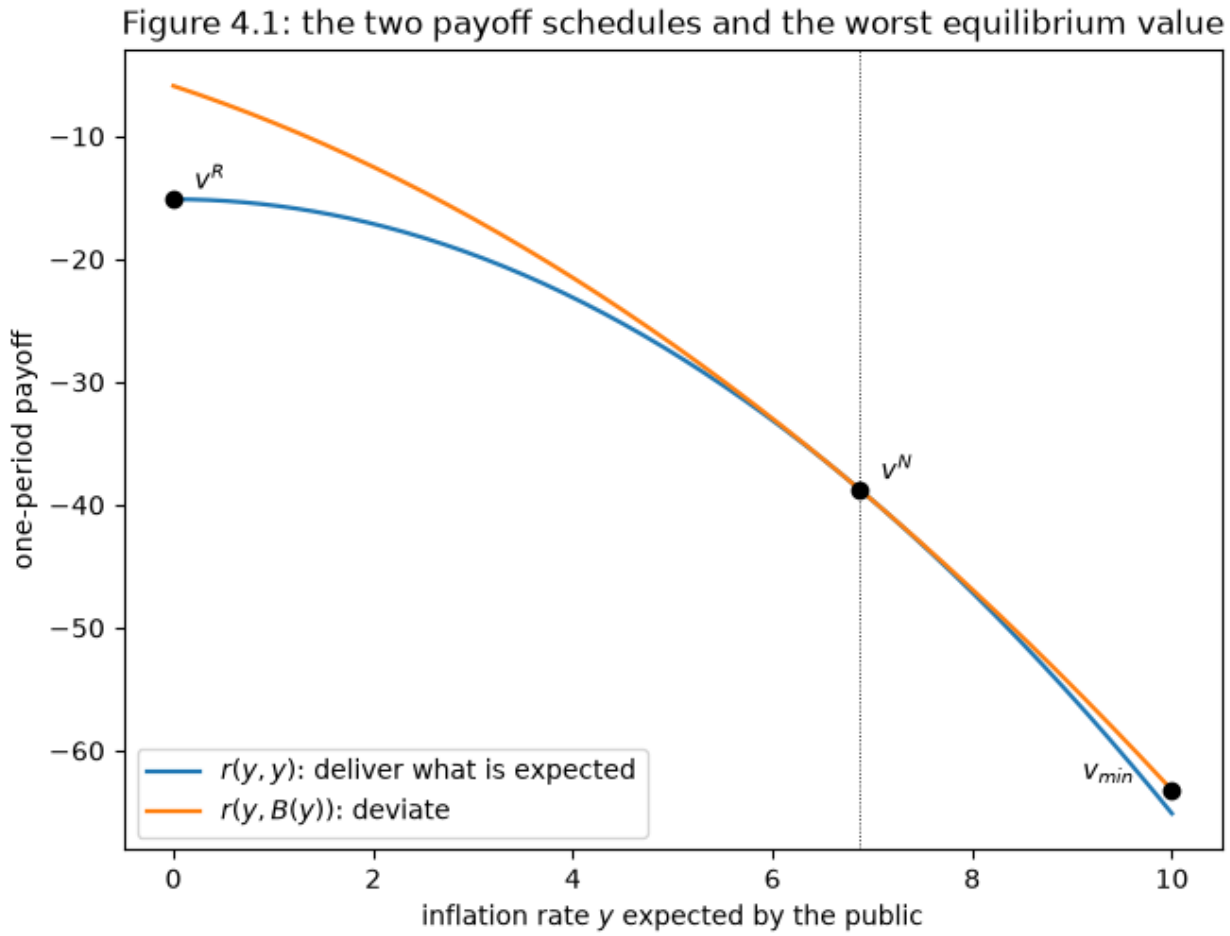


Fig. 93.1: The two payoff schedules: delivering expected inflation against deviating from it

93.3 Recursive strategies and promised values

A government strategy must be able to condition today's action on the entire past.

Carrying whole histories around is unmanageable, so we follow [Sargent, 1999] in restricting attention to strategies with a recursive representation — a restriction that costs nothing, since it excludes no equilibrium *values*.

Definition 93.3.1 (Recursive government strategy)

A recursive government strategy is a pair of functions $\sigma = (\sigma_1, \sigma_2)$ together with an initial condition v_1, with the structure

$$v_1 \in \mathbb{R} \text{ given}, \qquad y_t = \sigma_1(v_t), \qquad v_{t+1} = \sigma_2(v_t, x_t, y_t),$$

where v_t is a state variable that summarizes the history of outcomes before t.

Each member of the private sector knows v_t and knows σ, and so forecasts

$$x_t = \sigma_1(v_t). \tag{93.5}$$

Equation (93.5) builds rational expectations into the private sector: the public is never surprised along the equilibrium path.

A strategy (σ, v_1) generates an entire outcome path, and hence a value $V^g(\sigma, v_1)$ through (93.3).

The theory of credible policy now does something that at first looks like a trick.

It ties the past to the future by requiring the state variable to *be* the value it generates,

$$v = V^g(\sigma, v). \tag{93.6}$$

So v works two shifts at once.

In $v_{t+1} = \sigma_2(v_t, x_t, y_t)$ it is a bookkeeping device that records what has happened.

In (93.6) it is a **promised value** — the discounted future the government is owed on entering the period.

Reading it the second way is what makes the machinery go: the strategy hands the government current and future outcomes that make it *want* to do what is expected of it.

Note

Two readings of σ are available and, within an equilibrium, impossible to tell apart.

It can be a decision rule that the government chooses, or a description of a system of public expectations to which the government conforms.

We return to this ambiguity in the interpretation section, because it is where the theory's agnosticism comes from.

93.3.1 Historical antecedents

Formalizing credibility was an achievement of the 1980s, but a sophisticated grasp of the idea is much older.

In 1784 the finance minister Jacques Necker explained to Louis XVI why an absolute monarch, free to renege on debt payments, found it hard to borrow [Sargent and Velde, 1995]:

> Therefore one can rekindle or sustain public trust only by giving reassurances on the sovereign's intentions, and by proving that no motive can incite him to fail his obligations.

Every clause of that sentence is in the modern definition.

A king proves he will never fail his obligations by never *wanting* to fail them — by conforming to a system of expectations that carries its own incentive constraints, so that at every date and in every contingency his current payoff plus continuation value is higher if he confirms expectations than if he disappoints them.

Definition 93.3.2 (Subgame perfect equilibrium)

A recursive strategy with promised value v is a **subgame perfect equilibrium** (SPE) if and only if

(a) $\sigma_2(v, \sigma_1(v), \eta)$ is itself a value attainable by a subgame perfect equilibrium, for every $\eta \in Y$; and

(b) writing $y = \sigma_1(v)$,

$$v = (1-\delta)\, r(y,y) + \delta\, \sigma_2(v,y,y) \;\geq\; (1-\delta)\, r(y,\eta) + \delta\, \sigma_2(v,y,\eta), \qquad \forall\, \eta \in Y.$$

The definition attaches four objects to an equilibrium: a promised value v; a first-period outcome (y, y); a continuation value v' if the prescribed outcome is observed; and another continuation value $\tilde{v}$ if it is not.

In those terms condition (b) reads

$$v = (1-\delta) r(y,y) + \delta v' \;\geq\; (1-\delta) r(y,\eta) + \delta \tilde{v}, \qquad \forall \eta \in Y,$$

which simply says the government does better adhering than departing.

Condition (a) says the continuation values must themselves be equilibrium values.

The definition is circular — equilibrium values appear on both sides — and that circularity is exactly what recursivity buys, and what APS learned to exploit.

93.4 The Abreu–Pearce–Stacchetti method

Dynamic programming computes an optimal value function by iterating the Bellman equation, a map that turns tomorrow's value function into today's.

APS adapted the idea to equilibrium *sets*.

Start with a candidate set $W \subset \mathbb{R}$ of continuation values.

Pick a first-period rational-expectations outcome (y, y) and two continuation values from W: a w_1 to reward adherence and a w_2 to punish deviation.

If w_1 is high enough and w_2 low enough, the pair supports y, and delivers the value

$$w = (1-\delta) r(y,y) + \delta w_1 \;\geq\; (1-\delta) r(y,\eta) + \delta w_2, \qquad \forall \eta \in Y.$$

Definition 93.4.1 (Admissibility)

A pair (y, w) is **admissible** with respect to a set of continuation values W if there exist $w_1, w_2 \in W$ with $w = (1-\delta)r(y,y) + \delta w_1 \geq (1-\delta)r(y,\eta) + \delta w_2$ for all $\eta \in Y$.

Let $B(W)$ collect the w components of all admissible pairs.

This construction builds in condition (b) of *Definition 93.3.2* but ignores condition (a), because the continuation values were drawn from an arbitrary set.

Definition 93.4.2 (Self-generation)

A set W of prospective continuation values is **self-generating** if $W \subseteq B(W)$.

Every value in a self-generating set is supported by continuation values drawn from that same set — which is condition (a).

APS proved that the set V of SPE values is the largest self-generating set, that B maps compact sets into compact sets, that B is monotone ($W_2 \subseteq W_1$ implies $B(W_2) \subseteq B(W_1)$), and that starting from any W_0 with $B(W_0) \subseteq W_0$ the iteration $W_j = B(W_{j-1})$ converges monotonically to $V = B(V)$.

93.4.1 Computing the operator

Two simplifications make B easy to evaluate here.

Because the government's temptation is worst when it best-responds, the constraint in *Definition 93.4.1* binds hardest at $\eta = B(y)$, so we may replace "for all η" by that single deviation.

And because a lower punishment relaxes the constraint, we may always set w_2 to the smallest element of W.

Writing $W = [\underline{w}, \overline{w}]$, the outcome y is then admissible provided some $w_1 \in W$ satisfies

$$w_1 \;\geq\; \frac{(1-\delta)\,[r(y, B(y)) - r(y,y)]}{\delta} + \underline{w} \;\equiv\; \ell(y), \tag{93.7}$$

and the values it generates sweep out the interval $[(1-\delta)r(y,y) + \delta \max(\underline{w}, \ell(y)),\ (1-\delta)r(y,y) + \delta\overline{w}]$.

```
def B_operator(m, W, n_grid=4001):
    """
    One application of the APS operator to an interval W = [w_lo, w_hi].

    Returns the new interval, or None if no first-period outcome is admissible.
    """
    w_lo, w_hi = W
    y = np.linspace(0.0, m.y_max, n_grid)
    ℓ = (1 - m.δ) * (r_cheat(m, y) - r_keep(m, y)) / m.δ + w_lo    # equation (10)
    ok = ℓ <= w_hi                                                  # admissible␣
↪outcomes
    if not ok.any():
        return None
    keep = (1 - m.δ) * r_keep(m, y)
    lows = keep[ok] + m.δ * np.maximum(w_lo, ℓ[ok])
    highs = keep[ok] + m.δ * w_hi
    return lows.min(), highs.max()
```

(continues on next page)

(continued from previous page)

```
def solve_aps(m, W0=None, tol=1e-12, max_iter=10_000):
    "Iterate the APS operator to its largest fixed point."
    W = (r_keep(m, m.y_max), 0.0) if W0 is None else W0
    for it in range(max_iter):
        W_new = B_operator(m, W)
        if W_new is None:
            return None, it
        if max(abs(W_new[0] - W[0]), abs(W_new[1] - W[1])) < tol:
            return W_new, it
        W = W_new
    return W, max_iter
```

We start from $W_0 = [r(y^\#, y^\#),\, 0]$, which is large enough to contain every equilibrium value, and iterate.

```
W, iters = solve_aps(m)
print(f"converged in {iters} iterations")
print(f"set of SPE values V = [{W[0]:.4f}, {W[1]:.4f}]")
```

```
converged in 534 iterations
set of SPE values V = [-63.2195, -15.1250]
```

93.5 The best and the worst equilibrium

APS's iteration is general but tells us little about *why* the set ends where it does.

For this economy both endpoints can be found by hand, and the arguments are instructive.

93.5.1 The worst

The worst equilibrium value solves

$$\underline{v} = \min_{y \in Y,\, v_1 \in V} \left[(1-\delta) r(y,y) + \delta v_1\right] \quad \text{subject to} \quad (1-\delta) r(y,y) + \delta v_1 \geq (1-\delta) r(y, B(y)) + \delta \underline{v},$$

where the *worst* value is used as the continuation in the event of a deviation — the harshest punishment available.

The minimum is attained where the constraint binds, and at that point the two sides collapse to $\underline{v} = r(y, B(y))$ for some y.

So finding the worst equilibrium reduces to a one-dimensional problem:

$$\underline{v} = \min_{y \in Y} r(y, B(y)) = -\frac{\left(U^* + \theta y^\#\right)^2}{2\left(1+\theta^2\right)}, \tag{93.8}$$

where the closed form follows from (93.4), and the minimizing action is $y^\#$ because the deviation payoff falls monotonically in y.

Now the role of the upper bound $y^\#$ is clear.

It is what keeps the worst punishment finite; without it the government could be threatened with arbitrarily bad outcomes and almost anything could be supported.

Proposition 93.5.1 (The worst SPE is self-enforcing)

At the worst equilibrium the continuation value following a deviation equals the initial promised value, so a deviation simply restarts the equilibrium.

93.5.2 The best

Given $\underline{v}$ as the threat, the best value solves

$$\overline{v} = \max_{y \in Y} \; r(y, y) \quad \text{subject to} \quad r(y, y) \geq (1 - \delta) r(y, B(y)) + \delta \underline{v}, \tag{93.9}$$

where we have used the fact that the best value must reward adherence with itself, so that $\overline{v} = (1-\delta) r(y, y) + \delta \overline{v} = r(y, y)$.

Proposition 93.5.2 (The best SPE is self-rewarding)

At the best equilibrium the continuation value following adherence equals the promised value.

```
def worst_value(m):
    "The worst SPE value and the action that attains it."
    return r_cheat(m, m.y_max), m.y_max


def best_value(m, v_lo, n_grid=200_001):
    "The best SPE value, given the worst value as the punishment threat."
    y = np.linspace(0.0, m.y_max, n_grid)
    feasible = r_keep(m, y) >= (1 - m.δ) * r_cheat(m, y) + m.δ * v_lo
    if not feasible.any():
        return None, None
    vals = np.where(feasible, r_keep(m, y), -np.inf)
    k = vals.argmax()
    return vals[k], y[k]


v_lo, y_sharp = worst_value(m)
v_hi, y_best = best_value(m, v_lo)

print(f"worst value  v_lo = {v_lo:.4f}   attained with y = {y_sharp:.2f}")
print(f"best  value  v_hi = {v_hi:.4f}   attained with y = {y_best:.2f}")
print(f"APS iteration gave [{W[0]:.4f}, {W[1]:.4f}]")
print(f"the two agree: {np.allclose(W, (v_lo, v_hi), atol=1e-6)}")
```

```
worst value  v_lo = -63.2195   attained with y = 10.00
best  value  v_hi = -15.1250   attained with y = 0.00
APS iteration gave [-63.2195, -15.1250]
the two agree: True
```

The programming problems and the set iteration agree, and at this discount factor the best equilibrium is Ramsey itself.

93.6 Examples of recursive equilibria

We now construct particular equilibria by guess-and-verify, in the order in which the literature found them.

93.6.1 Infinite repetition of Nash

The easiest equilibrium repeats the one-period Nash outcome forever.

Take $v_1 = v^N = r(y^N, y^N)$, $\sigma_1(v) = y^N$ for every v, and $\sigma_2(v, x, y) = v^N$ for every (v, x, y).

Condition (a) holds by construction, and condition (b) collapses to $r(y^N, y^N) \geq r(y^N, B(y^N))$, which holds *with equality* because y^N is a fixed point of the best response map.

Nothing deters the government here, and nothing needs to: it is already doing what it most wants to do.

93.6.2 Infinite repetition of something better

Let $v^b = r(y^b, y^b) > v^N$ be a better-than-Nash value, and suppose

$$r(y^b, B(y^b)) - r(y^b, y^b) \;\leq\; \frac{\delta}{1-\delta}\,(v^b - v^N)\,. \tag{93.10}$$

The left side is the one-period gain from cheating; the right side is the discounted loss from reverting to Nash forever.

When (93.10) holds, the strategy that prescribes y^b while the promised value is v^b and reverts to Nash after any deviation is an SPE.

Barro and Gordon [1983] studied the case $y^b = y^R$: anticipated reversion to Nash supports Ramsey forever.

Whether it does depends on patience, and for this economy the threshold has a remarkably clean form.

Setting $y^b = 0$ in (93.10) and using the closed forms (93.4), every U^* and every scale factor cancels, leaving

$$\delta \;\geq\; \delta^\star = \frac{1}{2+\theta^2}. \tag{93.11}$$

```
def supports_ramsey_by_nash(m):
    "Does reversion to Nash sustain Ramsey forever?"
    gain = r_cheat(m, 0.0) - r_keep(m, 0.0)
    loss = m.δ / (1 - m.δ) * (r_keep(m, 0.0) - r_keep(m, y_nash(m)))
    return gain <= loss


δ_star = 1 / (2 + m.θ**2)
print(f"closed-form cutoff  δ* = 1/(2 + θ²) = {δ_star:.6f}\n")
for δ in (0.20, 0.27, δ_star - 1e-6, δ_star + 1e-6, 0.50, 0.95):
    md = m._replace(δ=δ)
    print(f"  δ = {δ:.8f}: Nash reversion supports Ramsey? "
          f"{supports_ramsey_by_nash(md)}")
```

```
closed-form cutoff  δ* = 1/(2 + θ²) = 0.280702

  δ = 0.20000000: Nash reversion supports Ramsey? False
  δ = 0.27000000: Nash reversion supports Ramsey? False
  δ = 0.28070075: Nash reversion supports Ramsey? False
  δ = 0.28070275: Nash reversion supports Ramsey? True
  δ = 0.50000000: Nash reversion supports Ramsey? True
  δ = 0.95000000: Nash reversion supports Ramsey? True
```

Above $\delta^\star \approx 0.28$ the threat of reverting to Nash is enough to hold the government at zero inflation; below it, the threat is too weak.

93.6.3 Something worse: Abreu's stick and carrot

When reversion to Nash is not a strong enough threat, Abreu [1988] asked whether some *worse* equilibrium could be used as the punishment.

His device is a stick-and-carrot strategy: the government is required to inflate at the maximum rate $y^{\#}$ for one period — the stick — after which it is rewarded with the Ramsey outcome forever.

The value is

$$\tilde{v} = (1-\delta)\, r(y^{\#}, y^{\#}) + \delta\, v^{R}, \tag{93.12}$$

and the punishment for refusing the stick is to *restart* the whole strategy, so the deviation continuation value is $\tilde{v}$ itself.

That self-referential choice is what makes the arithmetic so simple.

Substituting $\tilde{v}$ on both sides of the incentive constraint, the δ terms cancel and the strategy is an equilibrium precisely when $\tilde{v} \geq r(y^{\#}, B(y^{\#})) = \underline{v}$.

```
def abreu_value(m):
    "Value of the stick-and-carrot strategy: one period at y^#, then Ramsey."
    return (1 - m.δ) * r_keep(m, m.y_max) + m.δ * r_keep(m, 0.0)


for δ in (0.20, 0.95):
    md = m._replace(δ=δ)
    va, vn = abreu_value(md), r_keep(md, y_nash(md))
    print(f"δ = {δ}:  v_abreu = {va:8.4f}   v^N = {vn:8.4f}   "
          f"worse than Nash? {va < vn}   is an SPE? {va >= worst_value(md)[0]}")
```

```
δ = 0.2:  v_abreu = -55.1250   v^N = -38.7578   worse than Nash? True   is an SPE?␣
↪True
δ = 0.95:  v_abreu = -17.6250   v^N = -38.7578   worse than Nash? False   is an␣
↪SPE? True
```

At $\delta = 0.95$ the stick-and-carrot value is *better* than Nash, because a single bad period is heavily outweighed by an eternity of Ramsey.

At $\delta = 0.2$ it is far *worse* than Nash — and that is the point.

A punishment worse than Nash is a stronger deterrent than reversion to Nash, so it can support outcomes that Nash reversion cannot.

```
m2 = m._replace(δ=0.20)

def supports_ramsey_with_threat(m, threat):
    "Does the threat value sustain Ramsey forever?"
    return r_keep(m, 0.0) >= (1 - m.δ) * r_cheat(m, 0.0) + m.δ * threat

print(f"at δ = 0.2:")
print(f"  threat = Nash    ({r_keep(m2, y_nash(m2)):8.4f}): "
      f"supports Ramsey? {supports_ramsey_with_threat(m2, r_keep(m2, y_nash(m2)))}")
print(f"  threat = Abreu   ({abreu_value(m2):8.4f}): "
      f"supports Ramsey? {supports_ramsey_with_threat(m2, abreu_value(m2))}")
```

```
at δ = 0.2:
  threat = Nash    (-38.7578): supports Ramsey? False
  threat = Abreu   (-55.1250): supports Ramsey? True
```

At $\delta = 0.2$, an impatient government cannot be held at Ramsey by the prospect of Nash, but *can* be held there by the prospect of Abreu's stick.

Two definitions name the structures we have just met.

Definition 93.6.1 (Self-enforcing and self-rewarding)

A recursive SPE is **self-enforcing** if the continuation value following a deviation equals the initial promised value, and **self-rewarding** if the continuation value following adherence equals the promised value.

Abreu's stick-and-carrot is self-enforcing; infinite repetition of a better-than-Nash outcome is self-rewarding.

Proposition 93.5.1 and *Proposition 93.5.2* say that these two structures are not curiosities: they are exactly what the extreme equilibria look like.

93.7 Multiplicity

Two layers of multiplicity inhabit this theory.

There is a continuum of equilibrium *values*, and — the sharper point — many different outcome *paths* attain the same value.

To see the second, we construct three equilibria that all deliver the worst value $\underline{v}$.

All three begin the same way: the first-period promised value is $\underline{v}$, the prescribed action is $y^{\#}$, and the continuation value v_2 solves $\underline{v} = (1-\delta) r(y^{\#}, y^{\#}) + \delta v_2$.

They differ in what they do afterwards.

```
def next_value(m, v):
    "Solve v = (1-δ) r(y^#, y^#) + δ v' for the continuation value v'."
    return (v - (1 - m.δ) * r_keep(m, m.y_max)) / m.δ


def action_delivering(m, v):
    "The inflation rate ỹ with r(ỹ, ỹ) = v."
    return np.sqrt(max(-2.0 * v - m.U_star**2, 0.0))


def method_1(m, v_lo, v_hi, max_steps=500):
    "Climb at y^# until one more step would overshoot v_hi, then settle at v_hi."
    v, y = [v_lo], []
    for _ in range(max_steps):
        nxt = next_value(m, v[-1])
        if nxt > v_hi:
            break
        y.append(m.y_max)
        v.append(nxt)
    target = (v[-1] - m.δ * v_hi) / (1 - m.δ)      # r(ỹ, ỹ) for the switching period
    y.append(action_delivering(m, target))
    v.append(v_hi)
```

(continues on next page)

(continued from previous page)

```
    y.append(action_delivering(m, v_hi))          # stay at v_hi forever after
    return np.array(v), np.array(y)


def method_2(m, v_lo, max_steps=500):
    "Climb at y^# only until the promised value first exceeds v^N, then freeze."
    v_N = r_keep(m, y_nash(m))
    v, y = [v_lo], []
    for _ in range(max_steps):
        nxt = next_value(m, v[-1])
        y.append(m.y_max)
        v.append(nxt)
        if nxt > v_N:
            break
    y.append(action_delivering(m, v[-1]))          # freeze at v** forever
    return np.array(v), np.array(y)


def method_3(m, v_lo):
    "One period at y^#, then freeze immediately at v_2."
    v_2 = next_value(m, v_lo)
    return (np.array([v_lo, v_2, v_2]),
            np.array([m.y_max, action_delivering(m, v_2),
                      action_delivering(m, v_2)]))
```

```
paths = {'method 1': method_1(m, v_lo, v_hi),
         'method 2': method_2(m, v_lo),
         'method 3': method_3(m, v_lo)}

fig, axes = plt.subplots(2, 3, figsize=(13, 6.5), sharex='col')
for k, (name, (v, y)) in enumerate(paths.items()):
    axes[0, k].plot(range(1, len(v) + 1), v, 'C0o-', ms=3.5, lw=1)
    axes[0, k].axhline(v_lo, color='k', ls='--', lw=0.8)
    axes[0, k].axhline(v_hi, color='C2', ls=':', lw=0.8)
    axes[0, k].set_title(f'{name}: continuation values', fontsize=10)
    axes[1, k].plot(range(1, len(y) + 1), y, 'C1o-', ms=3.5, lw=1)
    axes[1, k].set_ylim(-0.4, m.y_max + 0.4)
    axes[1, k].set_xlabel('$t$')
    axes[1, k].set_title(f'{name}: inflation', fontsize=10)
axes[0, 0].set_ylabel('promised value $v_t$')
axes[1, 0].set_ylabel('inflation $y_t$')
fig.suptitle('Figures 4.2-4.4: three equilibria that all attain the worst '
             'value $v_{min}$')
plt.tight_layout()
plt.show()
```

```
print(f"Nash value {r_keep(m, y_nash(m)):.4f} at inflation {y_nash(m):.4f};  "
      f"worst value {v_lo:.4f}\n")
for name, (v, y) in paths.items():
    print(f"{name:9s}: {len(y):3d} periods before settling,  "
          f"first-period value {v[0]:.4f},  "
          f"terminal value {v[-1]:8.4f} at inflation {y[-1]:.4f}")
```

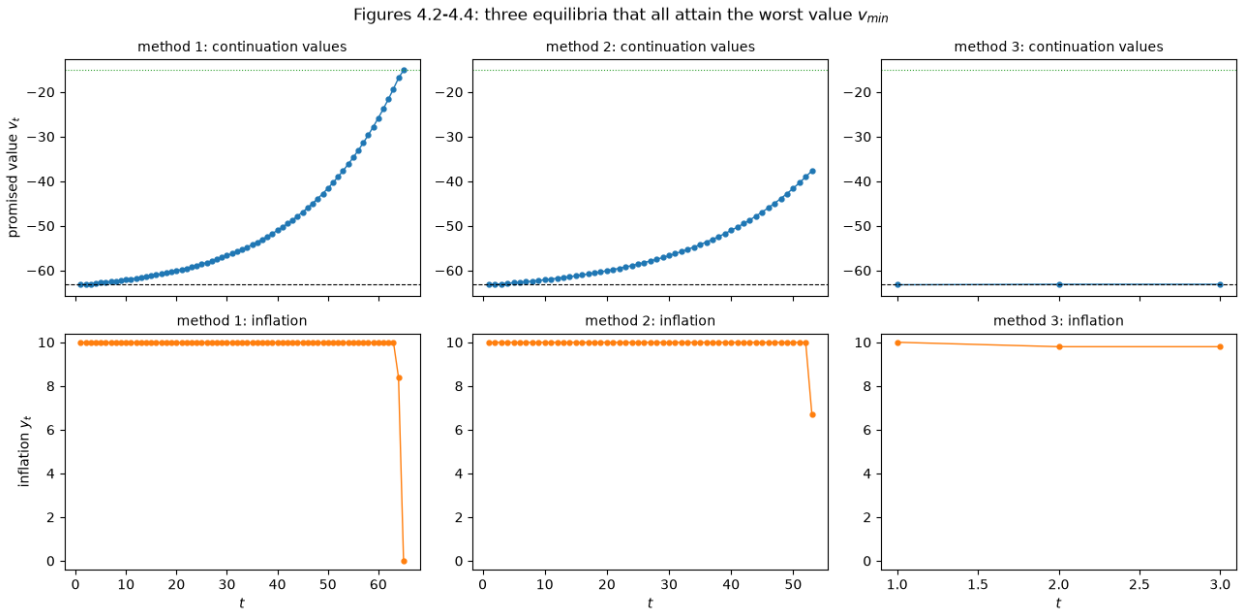


Fig. 93.2: Three subgame perfect equilibria that attain the same worst value

```
Nash value -38.7578 at inflation 6.8750;  worst value -63.2195

method 1 :  65 periods before settling,  first-period value -63.2195,  terminal␣
 ↪value -15.1250 at inflation 0.0000
method 2 :  53 periods before settling,  first-period value -63.2195,  terminal␣
 ↪value -37.6853 at inflation 6.7172
method 3 :   3 periods before settling,  first-period value -63.2195,  terminal␣
 ↪value -63.1192 at inflation 9.7974
```

The three paths could hardly be less alike.

The first climbs at maximum inflation for about sixty periods before easing down to Ramsey.

The second climbs for a shorter spell and then freezes forever at a value just *better* than Nash, sustained by an inflation rate just below the Nash rate.

The third inflates at the maximum for a single period and then settles immediately, at a value barely above the worst one.

All three are subgame perfect, and all three deliver exactly the same value to the government.

The equilibrium concept has nothing to say about which of them describes the world.

93.8 Numerical example

Here is the numerical example reported in chapter 4 of [Sargent, 1999], computed from scratch.

```
import pandas as pd

rows = [
    ("$\\theta$", m.θ, ""),
    ("$U^*$", m.U_star, ""),
    ("$y^\\#$", m.y_max, ""),
```

(continues on next page)

(continued from previous page)

```
    ("$\\delta$", m.δ, ""),
    ("$y^N$ (Nash inflation)", y_nash(m), "6.8750"),
    ("$y^R$ (Ramsey inflation)", 0.0, "0"),
    ("$v^R$", r_keep(m, 0.0), "-15.1250"),
    ("$v^N$", r_keep(m, y_nash(m)), "-38.7578"),
    ("$\\underline v$", v_lo, "-63.2195"),
    ("$v_{\\rm abreu}$", abreu_value(m), "-17.6250"),
]
pd.DataFrame(rows, columns=["object", "computed", "reported in the book"]).set_index(
 ↪"object")
```

object	computed	reported in the book
θ	1.250000	
U^*	5.500000	
$y^\#$	10.000000	
δ	0.950000	
y^N (Nash inflation)	6.875000	6.8750
y^R (Ramsey inflation)	0.000000	0
v^R	-15.125000	-15.1250
v^N	-38.757812	-38.7578
$\underline v$	-63.219512	-63.2195
$v_{\rm abreu}$	-17.625000	-17.6250

```
print(f"cutoff discount factor δ*        : {1 / (2 + m.θ**2):.4f}  (book reports 0.
 ↪2807)")
print(f"Abreu stick-and-carrot at δ = 0.2: {abreu_value(m._replace(δ=0.2)):.4f}"
      f"  (book reports -55.125)")
```

```
cutoff discount factor δ*        : 0.2807  (book reports 0.2807)
Abreu stick-and-carrot at δ = 0.2: -55.1250  (book reports -55.125)
```

Every entry reproduces the book.

93.9 Interpretations

The literature on credible plans bears mixed news for the *triumph of natural-rate theory* story of *The Rise and Fall of U.S. Inflation*.

It certainly rescues the government from the pessimism of *The Credibility Problem*: better outcomes than Nash are available, and at plausible discount factors Ramsey itself is an equilibrium.

But it rescues too much.

Values worse than Nash are equilibria too, and between the extremes lies a continuum.

The multitude of outcomes mutes the model empirically, and it undoes the very thing early rational-expectations researchers wanted from the hypothesis — the elimination of free parameters describing expectations.

93.9.1 Whose expectations are they?

In his 1979 review of an OECD report edited by Paul McCracken, Lucas protested the report's recommendation that "governments should try to promote good expectations," as though expectations were an extra set of policy instruments.

In 1979 that protest was well aimed: rational-expectations models then took government policy as exogenous and made expectations a *function* of it, linked by cross-equation restrictions.

The theory of credible policy changes the picture in a way that vindicates neither side cleanly.

It turns systems of expectations into free parameters that influence outcomes — but into parameters that nobody inside the model gets to choose.

The government complies with equilibrium expectations about its own behavior.

Within an equilibrium, the government's strategy is simultaneously a decision rule and a description of the public's expectations, and the two cannot be disentangled.

As Sargent puts it: the authors of the McCracken report believed in multiplicity and manipulation, while Lucas doubted both; the literature on credible plans supports multiplicity but not manipulation.

93.9.2 Remedies

Reputation alone is a weak foundation for anti-inflation policy, and that weakness has inspired proposals to change the game rather than to hope for a good equilibrium within it.

Rogoff [1985] proposed delegating monetary policy to someone who cares less about unemployment than society does.

Assigning it to someone unaware even of a *temporary* tradeoff would be equally effective, and Alan Blinder later suggested a related device: an authority who knows the natural rate and never wants unemployment to differ from it.

Maintaining a pool of potential central bankers with different inflation–unemployment preferences can also improve outcomes [Barro and Gordon, 1983].

93.9.3 Where this leaves us

Every remedy just listed changes the *institutions*, not the theory.

Within the theory, the multiplicity is irreducible, and Sargent locates its source precisely: it stems from the rationality imputed to *everyone* in the system.

Perfection is what generates the continuum, because a system of expectations sophisticated enough to support one equilibrium is sophisticated enough to support many.

So the book retreats from perfection.

The remaining lectures replace fully rational participants with ones whose understanding of the economy is limited — first the public in *Adaptive Expectations and the Phelps Problem*, then both sides in *Optimal Misspecified Beliefs* and *Self-Confirming Equilibria*, and finally a government that estimates its model in real time in *Adaptive Learning and Escape Dynamics*.

Those models are closer to what Lucas had in mind when he criticized the McCracken report, and, unlike the theory of this lecture, they make predictions.

93.10 Exercises

Exercise 93.10.1

The cutoff (93.11) claims that reversion to Nash sustains Ramsey forever if and only if $\delta \geq 1/(2+\theta^2)$ — a threshold that depends on the slope of the Phillips curve alone, and not on the natural rate U^* or the upper bound $y^\#$.

Verify both claims numerically.

For a grid of θ values, find the smallest δ on a fine grid at which reversion to Nash supports Ramsey, and compare it with $1/(2+\theta^2)$.

Then check that changing U^* leaves the answer untouched.

Solution

```
δ_grid = np.linspace(0.01, 0.99, 9801)

rows = []
for θ in (0.5, 1.0, 1.25, 2.0, 3.0):
    for U_star in (5.5, 11.0):
        md = Model(θ=θ, U_star=U_star)
        ok = [δ for δ in δ_grid
              if supports_ramsey_by_nash(md._replace(δ=δ))]
        rows.append((θ, U_star, min(ok), 1 / (2 + θ**2)))

pd.DataFrame(rows, columns=["$\\theta$", "$U^*$", "smallest $\\delta$ found",
                            "$1/(2+\\theta^2)$"]).round(4)
```

	θ	U^*	smallest δ found	$1/(2+\theta^2)$
0	0.50	5.5	0.4445	0.4444
1	0.50	11.0	0.4445	0.4444
2	1.00	5.5	0.3334	0.3333
3	1.00	11.0	0.3334	0.3333
4	1.25	5.5	0.2808	0.2807
5	1.25	11.0	0.2808	0.2807
6	2.00	5.5	0.1667	0.1667
7	2.00	11.0	0.1667	0.1667
8	3.00	5.5	0.0910	0.0909
9	3.00	11.0	0.0910	0.0909

The grid search matches the closed form to within the grid spacing, and doubling U^* changes nothing.

The reason both U^* and $y^\#$ drop out is visible in (93.4): the one-period temptation at $y = 0$ and the Nash–Ramsey value gap are both proportional to U^{*2}, so the scale cancels from the inequality, and $y^\#$ never enters because neither side involves the upper bound.

A steeper Phillips curve — a larger θ — makes the Nash outcome worse relative to Ramsey, which strengthens the threat and lets a *less* patient government be held at zero inflation.

Exercise 93.10.2

The set of equilibrium values shrinks as the government becomes impatient.

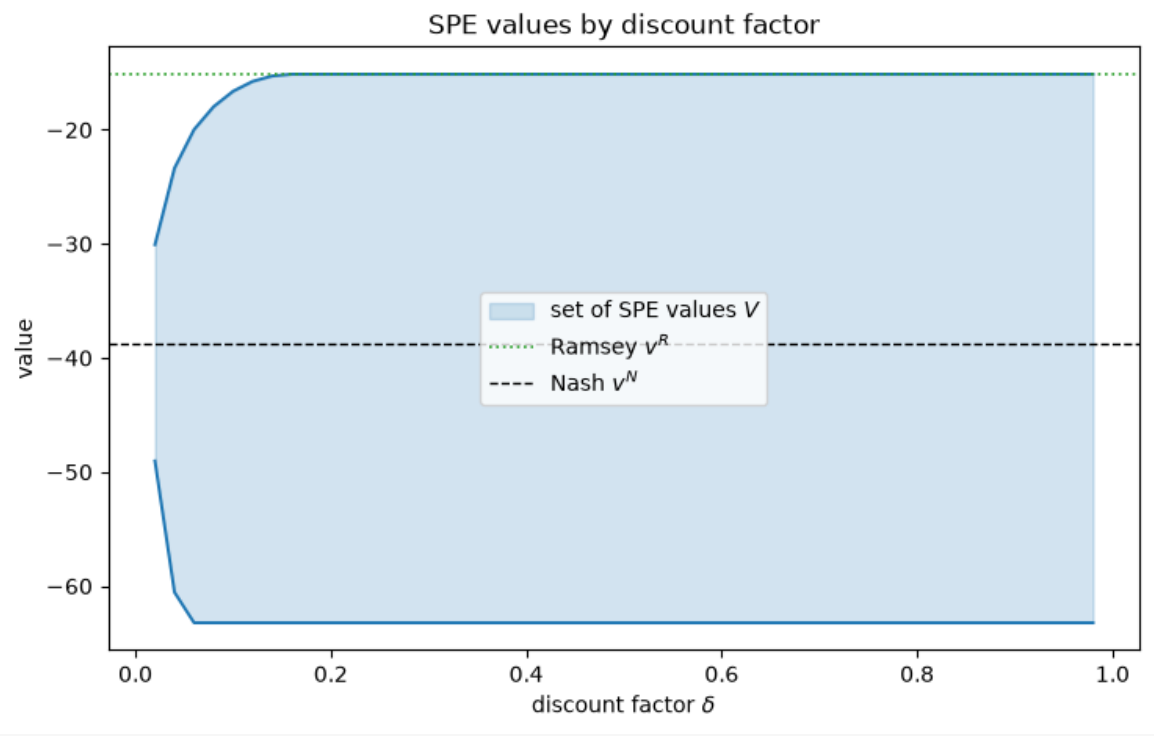
Use `solve_aps` to compute the set $V = [\underline{v}, \overline{v}]$ for a grid of discount factors, and plot both endpoints against δ, marking the Nash and Ramsey values.

At roughly what discount factor does the best equilibrium value stop being Ramsey?

What happens to the set as $\delta \to 0$, and why is that the answer you should expect?

Solution

```
δs = np.linspace(0.02, 0.98, 49)
lows, highs = [], []
for δ in δs:
    Wδ, _ = solve_aps(m._replace(δ=δ))
    lows.append(Wδ[0])
    highs.append(Wδ[1])

fig, ax = plt.subplots(figsize=(8.5, 5))
ax.fill_between(δs, lows, highs, alpha=0.2, color='C0', label='set of SPE values $V
↪$')
ax.plot(δs, highs, 'C0', lw=1.4)
ax.plot(δs, lows, 'C0', lw=1.4)
ax.axhline(r_keep(m, 0.0), color='C2', ls=':', lw=1.2, label='Ramsey $v^R$')
ax.axhline(r_keep(m, y_nash(m)), color='k', ls='--', lw=1, label='Nash $v^N$')
ax.set_xlabel(r'discount factor $\delta$')
ax.set_ylabel('value')
ax.set_title('SPE values by discount factor')
ax.legend()
plt.show()
```



```
best_is_ramsey = [δ for δ, h in zip(δs, highs)
                  if np.isclose(h, r_keep(m, 0.0), atol=1e-6)]
print(f"best value equals Ramsey for δ ≥ {min(best_is_ramsey):.3f}")
print(f"closed-form cutoff for Nash reversion: δ* = {1/(2+m.θ**2):.3f}")
```

```
best value equals Ramsey for δ ≥ 0.180
closed-form cutoff for Nash reversion: δ* = 0.281
```

The best equilibrium value equals Ramsey down to a discount factor well *below* the $\delta^\star \approx 0.28$ of (93.11), and the reason is Abreu.

The cutoff $\delta^\star$ was derived using reversion to *Nash* as the threat; the APS set uses the worst equilibrium as the threat, which is a good deal harsher, so Ramsey survives at lower discount factors than Barro and Gordon's argument alone would suggest.

As $\delta \to 0$ the set collapses toward the single point v^N.

That is what it must do: an entirely impatient government cares only about the current period, promises about the future carry no weight, and the only outcome that can be sustained is the one the government would choose myopically — the Nash outcome.

Exercise 93.10.3

Method 3 in the multiplicity section is the boldest of the three: it inflates at $y^\#$ for a single period and then freezes forever at a constant inflation rate.

Because it settles immediately, its incentive constraint is the tightest of the three, so it is worth checking rather than assuming.

Verify directly that method 3 is a subgame perfect equilibrium: confirm that its first-period promise is honoured, that its frozen continuation value lies inside V, and that the government prefers adherence to deviation in *both* phases, using $\underline{v}$ as the punishment.

Solution

```
v_2 = next_value(m, v_lo)
y_tilde = action_delivering(m, v_2)

# phase 1: promised v_lo, prescribed action y^#, continuation v_2 on adherence
lhs_1 = (1 - m.δ) * r_keep(m, m.y_max) + m.δ * v_2
rhs_1 = (1 - m.δ) * r_cheat(m, m.y_max) + m.δ * v_lo

# phase 2: promised v_2, prescribed action ỹ, continuation v_2 on adherence
lhs_2 = (1 - m.δ) * r_keep(m, y_tilde) + m.δ * v_2
rhs_2 = (1 - m.δ) * r_cheat(m, y_tilde) + m.δ * v_lo

print(f"v_2 = {v_2:.4f},  ỹ = {y_tilde:.4f}")
print(f"v_2 lies inside V = [{v_lo:.4f}, {v_hi:.4f}]: "
      f"{v_lo <= v_2 <= v_hi}")
print(f"phase 1 delivers the promise: {np.isclose(lhs_1, v_lo)}")
print(f"phase 1 incentive: {lhs_1:.4f} >= {rhs_1:.4f}  ->  {lhs_1 >= rhs_1}")
print(f"phase 2 delivers the promise: {np.isclose(lhs_2, v_2)}")
print(f"phase 2 incentive: {lhs_2:.4f} >= {rhs_2:.4f}  ->  {lhs_2 >= rhs_2}")
print(f"slack in phase 2: {lhs_2 - rhs_2:.6f}")
```

```
v_2 = -63.1192,  ỹ = 9.7974
v_2 lies inside V = [-63.2195, -15.1250]: True
phase 1 delivers the promise: True
phase 1 incentive: -63.2195 >= -63.2195  ->  True
phase 2 delivers the promise: True
phase 2 incentive: -63.1192 >= -63.1312  ->  True
slack in phase 2: 0.011955
```

All the conditions hold, and the first-period promise is delivered exactly.

The margin in the second phase is very thin, and that is not an accident.

The frozen value v_2 is only barely above $\underline{v}$, so the punishment for deviating is only barely worse than the equilibrium itself — which is precisely what it means to be near the bottom of the set of equilibrium values.

Push the construction any lower and the incentive constraint fails, which is another way of seeing why $\underline{v}$ is where it is.

CHAPTER

NINETYFOUR

ADAPTIVE EXPECTATIONS AND THE PHELPS PROBLEM

Contents

- *Adaptive Expectations and the Phelps Problem*
  - *Overview*
  - *Adaptive expectations*
  - *The Phelps problem*
  - *Disinflation paths*
  - *The general Phelps problem*
  - *Testing the natural-rate hypothesis*
  - *Sacrifice ratios and the subversion of the Phelps model*
  - *Exercises*

In addition to what's in Anaconda, this lecture will use the following library:

```
!pip install quantecon
```

94.1 Overview

> They cannot look out far. They cannot look in deep. But when was that ever a bar To any watch they keep?
>
> – Robert Frost

This lecture continues the study of Phillips curve tradeoffs begun in *The Credibility Problem*.

It follows chapter 5 of [Sargent, 1999].

Credible Government Policies pursued better-than-Nash outcomes while keeping *everyone* rational, and found too many of them: a continuum of sustainable values, with nothing inside the theory to choose among them.

Here we retreat from that perfection in the smallest way we can, by keeping the government rational and giving the public a mechanical forecasting rule.

We describe

- the Cagan-Friedman adaptive expectations hypothesis,
- how [Phelps, 1967] used it to formulate a government control problem, and

- the role of adaptive expectations in early econometric tests of the natural-rate hypothesis.

The key object is the **Phelps problem**: a government that is rational solves an optimal control problem while the public forecasts inflation with a fixed, mechanical adaptive rule.

Unlike the one-period model of *The Credibility Problem*, the government now takes into account that the economy lasts more than one period, and that today's inflation shapes tomorrow's expectations.

This intertemporal link can improve outcomes and, in a limiting case, even sustain the Ramsey outcome.

The Phelps problem is a linear-quadratic (LQ) control problem, so we solve it with the *LQ control* tools from QuantEcon.

Let's start with some imports:

```
import matplotlib.pyplot as plt
import numpy as np
import quantecon as qe
```

94.2 Adaptive expectations

[Phelps, 1967] formulated a control problem for a natural-rate model.

He dropped rationality for the public but not for the government, and assigned the public a particular mechanical forecasting rule that is known to the government.

The public uses the adaptive expectations scheme of Milton Friedman and [Cagan, 1956]:

$$x_t - x_{t-1} = (1-\lambda)(y_{t-1} - x_{t-1}), \qquad \lambda \in (0,1), \tag{94.1}$$

where x_t is the public's expectation of inflation and y_t is inflation.

Rearranging, $x_t = \lambda x_{t-1} + (1-\lambda) y_{t-1}$, so expected inflation is a geometric distributed lag of past inflation,

$$x_t = (1-\lambda)\sum_{i=1}^{\infty} \lambda^{i-1} y_{t-i}. \tag{94.2}$$

Notice that (94.1) is a *constant gain* version of the least squares learning algorithm from *The Credibility Problem*, with the constant $(1-\lambda)$ playing the role that the decreasing gain t^{-1} played there.

Equation (94.2) possesses an **induction property**: if the government keeps repeating a constant $y_t = \tilde{y}$ policy, eventually the public comes to set $x_t \approx \tilde{y}$.

The weights $(1-\lambda)\lambda^{i-1}$ sum to one, so a permanently maintained inflation rate is eventually expected.

Let's confirm the induction property numerically.

```
def adaptive_forecast(y, λ, x0=0.0):
    "Simulate x_t = λ x_{t-1} + (1-λ) y_{t-1}."
    T = len(y)
    x = np.empty(T)
    x[0] = x0
    for t in range(1, T):
        x[t] = λ * x[t - 1] + (1 - λ) * y[t - 1]
    return x

T = 60
y_const = np.full(T, 10.0)                # a constant inflation policy
```

(continues on next page)

(continued from previous page)

```
fig, ax = plt.subplots(figsize=(9, 4.5))
for λ in [0.7, 0.9]:
    x = adaptive_forecast(y_const, λ)
    ax.plot(x, lw=1.5, label=rf'$\lambda = {λ}$')
ax.axhline(10.0, color='k', ls='--', lw=1, label='policy $\\tilde y$')
ax.set_xlabel('$t$')
ax.set_ylabel('expected inflation $x_t$')
ax.legend()
plt.show()
```

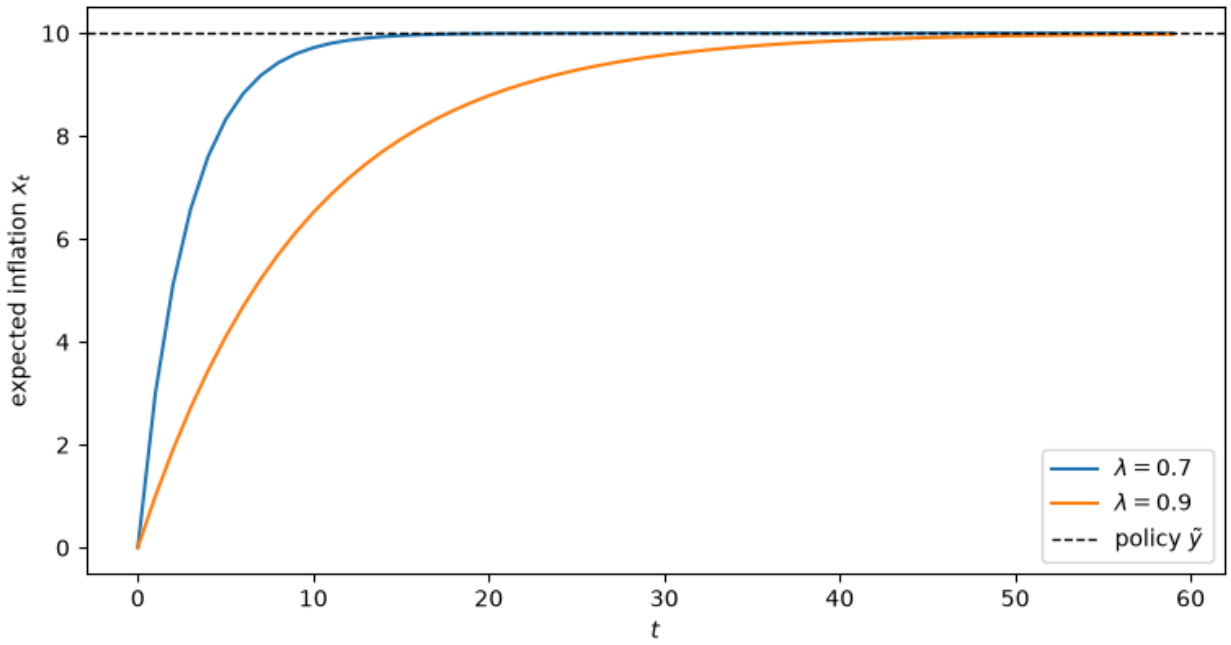


Fig. 94.1: Adaptive expectations converge to a constant inflation policy: the induction property

Under a constant policy the public's expectation converges to the policy, more slowly the larger is λ.

Solow and Tobin exploited this induction property when they tested the natural-rate hypothesis, as we discuss below.

94.3 The Phelps problem

The economy repeats forever, and the government evaluates outcome sequences with the discounted criterion

$$V^g = (1-\delta)\sum_{t=1}^{\infty}\delta^{t-1}p(U_t,y_t), \qquad p(U_t,y_t) = -\frac{1}{2}(U_t^2+y_t^2), \qquad \delta\in(0,1]. \tag{94.3}$$

When $\delta = 1$ we interpret (94.3) in the limit-of-means (Cesàro) sense.

The government maximizes (94.3) by choice of a rule for inflation y_t, subject to the adaptive expectations scheme (94.1) and the expectations-augmented Phillips curve

$$U_t = U^* - \theta(y_t - x_t). \tag{94.4}$$

Because expected inflation x_t is predetermined at the start of period t, it is the only endogenous state variable.

The government's problem is therefore a discounted LQ control problem with

- state $s_t = \begin{bmatrix} 1 & x_t \end{bmatrix}^\top$,
- control y_t, and
- transition $x_{t+1} = \lambda x_t + (1-\lambda) y_t$.

94.3.1 Casting the problem in LQ form

Write $U_t = a^\top s_t - \theta y_t$ with $a = \begin{bmatrix} U^* & \theta \end{bmatrix}^\top$.

The per-period loss $\frac{1}{2}(U_t^2 + y_t^2)$ then equals

$$\frac{1}{2}\left[s_t^\top (a a^\top) s_t + (\theta^2 + 1) y_t^2 - 2\theta\, y_t\, (a^\top s_t) \right].$$

Matching this to the QuantEcon LQ loss $s_t^\top R s_t + y_t^\top Q y_t + 2 y_t^\top N s_t$, and matching the transition to $s_{t+1} = A s_t + B y_t$, gives

$$R = \tfrac{1}{2} a a^\top, \quad Q = \tfrac{1}{2}(\theta^2 + 1), \quad N = -\tfrac{1}{2}\theta\, a^\top, \quad A = \begin{bmatrix} 1 & 0 \\ 0 & \lambda \end{bmatrix}, \quad B = \begin{bmatrix} 0 \\ 1 - \lambda \end{bmatrix}.$$

The discount factor is $\beta = \delta$; the scaling $(1-\delta)$ in (94.3) does not affect the optimal policy.

```
class PhelpsProblem:
    """
    The Phelps optimal-control problem: a rational government facing a
    public that forecasts inflation adaptively with parameter λ.
    """

    def __init__(self, θ=1.0, U_star=5.0, λ=0.7, δ=0.96):
        self.θ, self.U_star, self.λ, self.δ = θ, U_star, λ, δ

        a = np.array([[U_star], [θ]])
        R = 0.5 * (a @ a.T)
        Q = np.array([[0.5 * (θ**2 + 1)]])
        N = -0.5 * θ * a.T
        A = np.array([[1.0, 0.0], [0.0, λ]])
        B = np.array([[0.0], [1 - λ]])

        # δ = 1 (limit of means) is handled as the limit δ → 1
        β = min(δ, 1 - 1e-7)
        self.lq = qe.LQ(Q, R, A, B, N=N, beta=β)
        P, F, d = self.lq.stationary_values()

        # optimal rule y_t = f1 + f2 x_t
        self.f1, self.f2 = -F[0, 0], -F[0, 1]

    def simulate(self, x0=12.0, T=60):
        "Disinflation path (U_t, y_t) starting from expectation x0."
        θ, U_star, λ = self.θ, self.U_star, self.λ
        U, y, x = np.empty(T), np.empty(T), x0
        for t in range(T):
            y[t] = self.f1 + self.f2 * x
            U[t] = U_star - θ * (y[t] - x)
```

(continues on next page)

(continued from previous page)

```
            x = λ * x + (1 - λ) * y[t]
        return U, y
```

The optimal rule takes the form $y_t = f_1 + f_2 x_t$ with $f_1 \neq 0$ and $f_2 \neq 1$.

These inequalities reflect that the public does not use an optimal forecasting rule; if instead $f_1 = 0$ and $f_2 = 1$, we would have $y_t = x_t$ for all histories.

```
pp = PhelpsProblem(θ=1.0, U_star=5.0, λ=0.7, δ=0.96)
print(f"optimal rule:  y_t = {pp.f1:.3f} + {pp.f2:.3f} x_t")
```

```
optimal rule:  y_t = 0.406 + 0.334 x_t
```

94.3.2 A proposition

The reason the Phelps problem is interesting is the following result.

Proposition 94.3.1 (δ = 1 eventually sustains Ramsey)

In the absence of discounting ($\delta = 1$), the government drives y_t to 0, the Ramsey outcome.

When $\delta = 1$, λ governs the speed of convergence to the Ramsey outcome.

When $\delta < 1$, the limit point of y_t depends on a comparison of λ with δ.

For $\lambda < \delta$ and δ close to 1, the government's policy eventually approximates the Ramsey outcome.

The public's expectations are wrong along the transition path but are correct in the steady state, by virtue of the induction property.

94.4 Disinflation paths

We now reproduce the disinflation experiments of chapter 5 of [Sargent, 1999].

Set $\theta = 1$ and $U^* = 5$, and start the government's problem from late-1970s initial conditions $x_{-1} = y_{-1} = 12$, which imply $U = U^* = 5$.

The following tables record paths of unemployment U and inflation y at selected lags, for two discount factors $\delta \in \{0.96, 1\}$ and two adaptation parameters $\lambda \in \{0.7, 0.9\}$.

```
def disinflation_table(δ, lags=(1, 5, 20, 50)):
    rows = []
    for λ in [0.7, 0.9]:
        U, y = PhelpsProblem(λ=λ, δ=δ).simulate(x0=12.0, T=max(lags) + 1)
        for lag in lags:
            rows.append((λ, lag, U[lag - 1], y[lag - 1]))
    return rows

for δ in [0.96, 1.0]:
    print(f"\n δ = {δ}")
    print(f" {'λ':>4} {'lag':>4} {'U':>7} {'y':>7}")
    for λ, lag, U, y in disinflation_table(δ):
        print(f" {λ:>4} {lag:>4} {U:>7.1f} {y:>7.1f}")
```

```
δ = 0.96
   λ  lag       U       y
 0.7    1    12.6     4.4
 0.7    5     8.1     2.2
 0.7   20     5.1     0.7
 0.7   50     5.0     0.6
 0.9    1    11.9     5.1
 0.9    5    10.2     4.3
 0.9   20     6.9     2.5
 0.9   50     5.3     1.6

δ = 1.0
   λ  lag       U       y
 0.7    1    13.2     3.8
 0.7    5     8.3     1.5
 0.7   20     5.1     0.0
 0.7   50     5.0     0.0
 0.9    1    13.4     3.6
 0.9    5    11.3     2.7
 0.9   20     7.1     0.9
 0.9   50     5.2     0.1
```

For each parameter setting the government engineers a major recession and immediately brings inflation down more than halfway toward its eventual limiting value.

In the discounted case ($\delta = 0.96$), inflation settles at a positive level, and the government accepts a longer but milder recession when $\lambda = 0.9$ than when $\lambda = 0.7$.

In the undiscounted case ($\delta = 1$), inflation is driven all the way to the Ramsey value of zero, more slowly the larger is λ.

Let's plot the full disinflation paths.

```
fig, axes = plt.subplots(1, 2, figsize=(11, 4.5))

for δ, ax in zip([0.96, 1.0], axes):
    for λ in [0.7, 0.9]:
        U, y = PhelpsProblem(λ=λ, δ=δ).simulate(x0=12.0, T=60)
        ax.plot(y, lw=1.5, label=rf'$\lambda = {λ}$')
    ax.set_title(rf'$\delta = {δ}$')
    ax.set_xlabel('$t$')
    ax.set_ylabel('inflation $y_t$')
    ax.legend()

plt.tight_layout()
plt.show()
```

The undiscounted problem drives inflation to the Ramsey outcome; the discounted problem stops short of it.

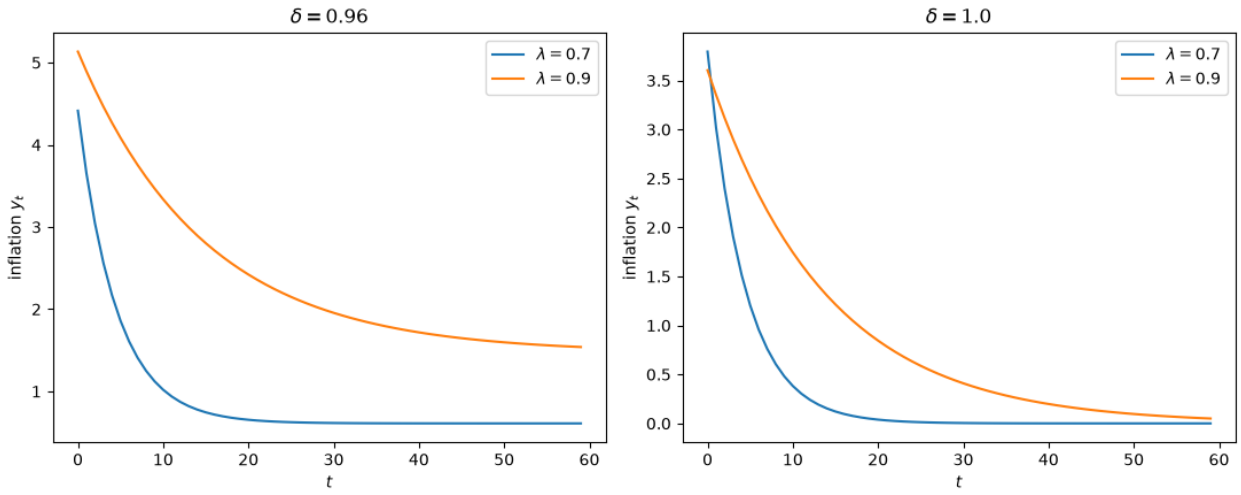


Fig. 94.2: Optimal disinflation paths from the Phelps problem, discounted and undiscounted

94.5 The general Phelps problem

For the government's control problem, what matters is the *reduced form* of the Phillips curve, not the underlying structure that identifies x_t.

It is useful to state a more general version of the Phelps problem in which the government's model is a reduced-form distributed lag Phillips curve.

Define the vectors

$$X_{U,t} = \begin{bmatrix} U_{t-1} & \cdots & U_{t-m_U} \end{bmatrix}^\top, \qquad X_{y,t} = \begin{bmatrix} y_{t-1} & \cdots & y_{t-m_y} \end{bmatrix}^\top,$$

and the state vector $X_t = \begin{bmatrix} X_{U,t}^\top & X_{y,t}^\top & 1 \end{bmatrix}^\top$, which collects information dated $t-1$ and earlier.

We can write two reduced-form Phillips curves that differ only in their *direction of fit*:

$$\text{Classical:} \quad U_t = \gamma^\top X_{C,t} + \varepsilon_{C,t}, \qquad X_{C,t} = \begin{bmatrix} y_t & X_{t-1}^\top \end{bmatrix}^\top,$$

$$\text{Keynesian:} \quad y_t = \beta^\top X_{K,t} + \varepsilon_{K,t}, \qquad X_{K,t} = \begin{bmatrix} U_t & X_{t-1}^\top \end{bmatrix}^\top.$$

The subscripts C and K stand for *Classical* (regress U on y) and *Keynesian* (regress y on U).

The general **Phelps problem** is to choose a control law $\hat{y}_t = hX_{t-1}$ to maximize the expected value of (94.3) subject to the government's believed Phillips curve and to $y_t = \hat{y}_t + v_{2t}$, where v_{2t} is a control error.

This induces a mapping from the government's beliefs γ to its decision rule h:

$$h = h(\gamma). \tag{94.5}$$

The specific problem solved above is the special case in which γ is restricted by substituting the adaptive expectations hypothesis (94.1) into the Phillips curve (94.4).

Once that substitution is made, the state variable x_t disappears from view.

These objects — γ, β, $h(\gamma)$, and the two directions of fit — are exactly the ingredients we will need to define **self-confirming equilibria** in *Self-Confirming Equilibria*.

Note

The **induction hypothesis** is the restriction that, in the Keynesian Phillips curve $y_t = \beta^\top X_{K,t} + \varepsilon_{K,t}$, the weights on lagged y's sum to unity (equivalently, in the classical form the weights on current and lagged y's sum to zero).

Under adaptive expectations this holds because the weights in (94.2) sum to one.

94.6 Testing the natural-rate hypothesis

Robert Solow and James Tobin [Solow, 1968, Tobin, 1968] exploited the induction hypothesis to test the natural-rate hypothesis.

Substituting the geometric distributed lag (94.2) into an inverted Phillips curve gives

$$y_t = (1-\lambda)\sum_{i=1}^{\infty}\lambda^{i-1}y_{t-i} + \theta^{-1}(U^* - U_t). \tag{94.6}$$

They proposed to test the natural-rate hypothesis by running the regression

$$y_t = b_0 + b_1(1-\lambda)\sum_{i=1}^{\infty}\lambda^{i-1}y_{t-i} + b_2 U_t + \varepsilon_t, \tag{94.7}$$

and interpreting a finding of $b_1 < 1$ as evidence of a long-run tradeoff between inflation and unemployment of slope $b_1 - 1$.

Early implementations found $b_1 < 1$ and so rejected the natural-rate hypothesis in favor of a long-run tradeoff.

[King and Watson, 1994] and others later argued that the pattern of rejections and non-rejections is consistent with the tendency of inflation to have a unit root after the 1960s but not before, so the unit-sum restriction $b_1 = 1$ is *compatible* with rational expectations when y_t has a unit root.

From the viewpoint of Phelps's control problem, it is incidental whether the natural-rate hypothesis holds: the Phelps problem imparts interesting dynamics to the inflation-unemployment choice whether or not $b_1 = 1$.

94.7 Sacrifice ratios and the subversion of the Phelps model

Despite its encouraging implications for sustaining the Ramsey outcome under the induction hypothesis, Phelps's control problem carries a tattered past.

In the Phelps problem, for a fixed $\delta < 1$ it is always possible to find a λ close enough to 1 that high inflationary expectations will make a government want to avoid disinflating.

In the late 1970s, models with long expectations-adjustment lags were used to recommend *against* reducing inflation.

Large *sacrifice ratios* — estimated amounts of foregone GDP required to bring inflation down one percentage point — circulated widely.

The lesson to carry forward is that activating the induction hypothesis can eventually lead to better outcomes, but in the form that Phelps, Tobin, and Solow used it, the induction hypothesis retreats from rational expectations.

In *Optimal Misspecified Beliefs* and *Self-Confirming Equilibria* we impute more symmetry to the government and the public, applying updating schemes like (94.1) to functions rather than to numbers, and we turn λ from a free parameter into an equilibrium outcome.

The LQ Phelps problem solved here returns again in *Adaptive Learning and Escape Dynamics*, *Escaping Nash Inflation*, and *Priors, Escapes, and Learning Cycles*, where a learning government re-solves it every period — and where activating the induction hypothesis is exactly what triggers the Volcker-like stabilizations.

94.8 Exercises

Exercise 94.8.1

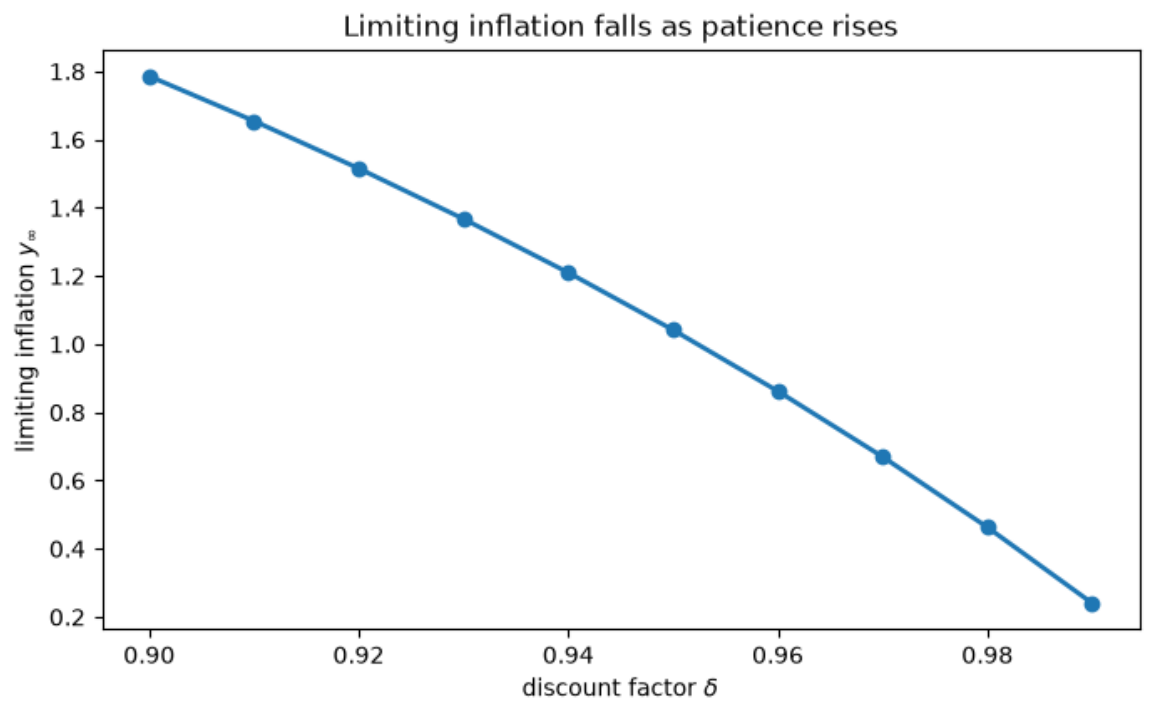
The proposition above states that as $\delta \to 1$ the government drives inflation to the Ramsey value 0.

For $\lambda = 0.8$, compute the limiting inflation rate y_∞ (the value that y_t settles at after a long simulation) for a grid of discount factors $\delta \in \{0.90, 0.92, \ldots, 0.99\}$.

Plot y_∞ against δ and confirm that it declines toward zero as $\delta \to 1$.

Solution

```
δ_grid = np.arange(0.90, 0.995, 0.01)
y_inf = []
for δ in δ_grid:
    U, y = PhelpsProblem(λ=0.8, δ=δ).simulate(x0=12.0, T=400)
    y_inf.append(y[-1])

fig, ax = plt.subplots(figsize=(8, 4.5))
ax.plot(δ_grid, y_inf, 'o-', lw=2)
ax.set_xlabel(r'discount factor $\delta$')
ax.set_ylabel(r'limiting inflation $y_\infty$')
ax.set_title('Limiting inflation falls as patience rises')
plt.show()
```



As δ rises toward one, the government becomes willing to accept the transitional recession needed to reap the long-run benefit of low expected inflation, so the limiting inflation rate falls toward the Ramsey value of zero.

Exercise 94.8.2

Verify the induction property of the optimal policy directly.

Take the discounted problem with $\delta = 0.96$ and $\lambda = 0.7$, and simulate a long disinflation path.

Check that in the steady state the public's expectation x_t equals actual inflation y_t (so the public is *not* fooled in the limit), even though expectations are wrong along the transition.

Solution

```
pp = PhelpsProblem(θ=1.0, U_star=5.0, λ=0.7, δ=0.96)
U, y = pp.simulate(x0=12.0, T=200)

# reconstruct the expectation path implied by the adaptive rule
x = adaptive_forecast(np.concatenate([[12.0], y]), λ=0.7, x0=12.0)[1:]

print(f"steady-state inflation   y_∞ = {y[-1]:.4f}")
print(f"steady-state expectation x_∞ = {x[-1]:.4f}")
print(f"gap                          = {y[-1] - x[-1]:.2e}")
```

```
steady-state inflation   y_∞ = 0.6098
steady-state expectation x_∞ = 0.6098
gap                          = -2.22e-16
```

In the steady state expected and actual inflation coincide, confirming that the induction property makes the public's forecast correct in the limit.

CHAPTER

NINETYFIVE

OPTIMAL MISSPECIFIED BELIEFS

Contents

- *Optimal Misspecified Beliefs*
  - *Overview*
  - *An experiment in Bray's lab*
  - *Optimal misspecification*
  - *Comparing the true and forecasting models*
  - *Lessons*
  - *Exercises*

95.1 Overview

This lecture continues the study of Phillips curve tradeoffs.

It follows chapter 6 of [Sargent, 1999].

In *Adaptive Expectations and the Phelps Problem* the public forecast inflation with a fixed adaptive rule whose parameter λ we simply chose.

That was unsatisfying in a way the *vindication* story of *The Rise and Fall of U.S. Inflation* cannot afford: a free parameter describing expectations is exactly what rational expectations was meant to eliminate.

Here we take the first step toward earning that parameter, by letting agents choose it to fit the data their own beliefs generate.

We describe three conceptual issues that recur throughout this suite of lectures:

1. how to formulate equilibria in which agents share a common *misspecified* least squares forecasting model,
2. how expectations can contribute independent dynamics within equilibria, and
3. how the classic adaptive expectations scheme can use second moments to approximate a first moment.

To expose these issues we temporarily set aside the Phillips curve and work with [Bray, 1982]'s simple model of the price of a single good, a workhorse for studying bounded rationality.

We alter Bray's model to illustrate an equilibrium concept that merges aspects of rational and adaptive expectations in a new way, and that we apply to the Phillips curve in *Self-Confirming Equilibria*.

The focus is on **market equilibrium with optimal but misspecified forecasts**.

- *Optimal* means the free parameters of the forecasting scheme are chosen by (nonlinear) least squares.
- *Misspecified* means the forecasting model is wrong in functional form.

A distinctive feature is that the true model *depends on* how the agents' model is misspecified: agents' beliefs affect their behavior, which shapes the data they then fit.

We work in the frequency domain, so let's import what we need:

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.optimize import minimize_scalar
```

95.2 An experiment in Bray's lab

Following [Bray, 1982], assume that

$$p_t = a + b\, p_{t+1}^e + u_t, \tag{95.1}$$

where u_t is IID with mean zero and variance σ_u^2, $a > 0$, $b \in (0, 1)$, p_t is the market price, and p_{t+1}^e is the market's expectation of next period's price.

The rational expectations equilibrium has $p_{t+1}^e = \frac{a}{1-b}$ and $p_t = \frac{a}{1-b} + u_t$.

Bray posited that p_{t+1}^e is the empirical average of past prices, and showed that when $0 < b < 1$ this decreasing-gain scheme converges almost surely to the rational expectation $\frac{a}{1-b}$.

During the transition, the state variable p_t^e contributes dynamics and makes the price serially correlated, but these dynamics are transitory: at the rational expectations equilibrium, p_t is a constant plus a serially uncorrelated shock.

95.2.1 Constant-gain adaptive expectations

To let expectations impart *persistent* serial correlation, we depart from Bray and assume that the market has **constant-gain** adaptive expectations,

$$p_{t+1}^e = C p_t + (1 - C) p_t^e, \qquad |C| < 1. \tag{95.2}$$

Bray's scheme replaces C by $\frac{1}{t}$, so that p_{t+1}^e becomes a sample average.

Fixing C instead *discounts* past observations, which arrests convergence to rational expectations and prevents p_{t+1}^e from converging to a constant.

Written as a distributed lag,

$$p_{t+1}^e = \frac{C}{1 - (1 - C)L} p_t, \tag{95.3}$$

where L is the lag operator.

Equation (95.2) would be the linear least squares forecast if the price followed the integrated moving-average process

$$p_t = p_{t-1} + \epsilon_t - (1 - C)\epsilon_{t-1}, \tag{95.4}$$

so the market perceives the price as composed of purely permanent and transitory components.

95.2.2 The actual law of motion

Substituting the belief (95.3) into (95.1) shows that when the market forecasts this way, its actions make the *actual* law of motion for the price

$$p_t = \frac{a}{1-b} + \frac{1}{1-bC}\left[\frac{1-(1-C)L}{1-\frac{1-C}{1-bC}L}\right] u_t = \nu + f(L)u_t, \tag{95.5}$$

where $\nu = \frac{a}{1-b}$ and $f(L)$ is defined to match.

The price has mean ν and spectral density

$$F(\omega) = f(e^{i\omega})f(e^{-i\omega})\,\sigma_u^2, \qquad \omega \in [-\pi, \pi].$$

Notice that F depends on C through f.

Let's encode the true process.

```
class BrayModel:
    """
    Bray's price model with constant-gain adaptive expectations.

    The perceived law of motion is an IMA(1,1) with unit root; to keep its
    spectral density well defined we approximate the unit root by a root ρ
    slightly below one, following Sargent (1999).
    """

    def __init__(self, a=1.0, b=0.5, σ_u=1.0, ρ=0.995, N=1024):
        self.a, self.b, self.σ_u, self.ρ, self.N = a, b, σ_u, ρ, N
        self.ν = a / (1 - b)
        ω = 2 * np.pi * np.arange(N) / N
        self.ω = ω
        self.z = np.exp(1j * ω)

    def true_spectrum(self, C):
        "Spectral density F(ω) of the actual price process, given belief C."
        b, z = self.b, self.z
        φ = (1 - C) / (1 - b * C)
        scale = 1 / (1 - b * C)
        f = scale * (1 - (1 - C) * z) / (1 - φ * z)
        return np.abs(f)**2 * self.σ_u**2

    def approx_spectrum(self, c, σ_ε2=1.0):
        "Spectral density G(ω) of the agent's approximating IMA model."
        g = (1 - (1 - c) * self.z) / (1 - self.ρ * self.z)
        return np.abs(g)**2 * σ_ε2
```

95.3 Optimal misspecification

Two facts about the actual law (95.5) motivate an equilibrium restriction on C:

1. Given that the price obeys (95.5), the true linear least squares one-step forecasting rule is *not* a geometric distributed lag like (95.2).
2. Even restricting the forecast to the form (95.2), the *best* such rule would make C solve a forecast-error-minimization problem, so C is an outcome, not a free parameter.

A rational expectations equilibrium would repair both features.

Following [Bray, 1982] we soften the equilibrium concept: we leave feature 1 untouched (agents keep the wrong functional form) while fixing feature 2 (they choose the best parameter within that form).

Think of putting a single individual into a market where everyone else (the "representative agent") uses C, so the price obeys (95.5).

The individual chooses c to fit the best model of the form

$$p_t = \frac{1-(1-c)L}{1-L}\epsilon_t = g(L)\epsilon_t, \tag{95.6}$$

by minimizing the one-step-ahead forecast error variance.

Because $g(L)$ has a unit root, its DC gain is infinite; this is exactly how the perceived model uses a unit root to *fit the constant mean* ν.

Numerically we replace the unit root by a root ρ slightly below one.

Definition 95.3.1 (Best-estimate map)

Given C and the consequent price process (95.5), the individual's best forecast parameter $c = B(C)$ is the nonlinear least squares estimator of c in (95.6), where the data are generated by (95.5).

Following the frequency-domain method of Hansen and Sargent [Hansen and Sargent, 1993], the best approximating (c, σ_ϵ^2) minimizes

$$A(c,\sigma_\epsilon^2) = \frac{1}{N}\sum_{j=0}^{N-1}\left\{\log G(\omega_j, c) + \frac{F(\omega_j)}{G(\omega_j,c)}\right\} + \frac{\nu^2}{G(0)}, \tag{95.7}$$

where $\omega_j = \frac{2\pi j}{N}$, and the term $\frac{\nu^2}{G(0)}$ makes the approximating model use its near-unit-root to fit the mean.

Concentrating out σ_ϵ^2 leaves a one-dimensional minimization over c.

Definition 95.3.2 (Equilibrium under forecast misspecification)

An equilibrium under forecast misspecification is a fixed point $C = B(C)$.

At such a fixed point the representative agent is representative: the single individual's best parameter equals the one everyone uses.

```
def best_estimate(model, C):
    "The best-estimate map c = B(C)."
    F = model.true_spectrum(C)
    z, ν, N = model.z, model.ν, model.N

    def neg_profile(c):
        H = np.abs((1 - (1 - c) * z) / (1 - model.ρ * z))**2   # |g|^2
        σ_ε2 = np.mean(F / H) + ν**2 / H[0]                    # concentrated
        return np.log(σ_ε2) + np.mean(np.log(H))               # profiled criterion

    res = minimize_scalar(neg_profile, bounds=(1e-4, 0.99), method='bounded')
    return res.x

def solve_equilibrium(model, C0=0.3, tol=1e-10, maxit=500):
```

(continues on next page)

(continued from previous page)

```
    "Iterate the best-estimate map to a fixed point."
    C = C0
    for _ in range(maxit):
        C_new = best_estimate(model, C)
        if abs(C_new - C) < tol:
            break
        C = C_new
    return C_new
```

```
bray = BrayModel(a=1.0, b=0.5, σ_u=1.0)
C_star = solve_equilibrium(bray)
print(f"equilibrium belief   C = {C_star:.4f}")
```

```
equilibrium belief   C = 0.0805
```

For these parameters the equilibrium belief is $C \approx 0.08$, reproducing the value reported in chapter 6 of [Sargent, 1999].

Let's also report the *actual* one-step-ahead forecast error standard deviation that agents incur by using their misspecified model.

```
def fitted_sigma2(model, C, c):
    "Concentrated innovation variance σ_ε^2 of the approximating model."
    F = model.true_spectrum(C)
    H = np.abs((1 - (1 - c) * model.z) / (1 - model.ρ * model.z))**2
    return np.mean(F / H) + model.ν**2 / H[0]

c_star = best_estimate(bray, C_star)
σ_bar = np.sqrt(fitted_sigma2(bray, C_star, c_star))
print(f"actual one-step forecast error std  σ̄_ε = {σ_bar:.4f}")
```

```
actual one-step forecast error std  σ̄_ε = 1.0579
```

95.4 Comparing the true and forecasting models

For the equilibrium C, we plot the equilibrium spectral densities of the true and approximating models.

```
F = bray.true_spectrum(C_star)
σ_ε2 = fitted_sigma2(bray, C_star, c_star)
G = bray.approx_spectrum(c_star, σ_ε2)

half = bray.N // 2
fig, ax = plt.subplots(figsize=(8, 5))
ax.plot(bray.ω[:half], np.log(F[:half]), 'C0', label='true model', lw=2)
ax.plot(bray.ω[:half], np.log(G[:half]), 'C1--', label='forecasting model', lw=2)
ax.set_xlabel(r'angular frequency $\omega$')
ax.set_ylabel('log spectral density')
ax.legend()
plt.show()
```

In minimizing (95.7), the approximating model uses its near-unit-root to fit the mean.

The large gap between the spectral densities at low frequencies reflects how the approximating model fits a *first* moment (the mean ν) with features of *second* moments (a spike in the spectral density at frequency zero).

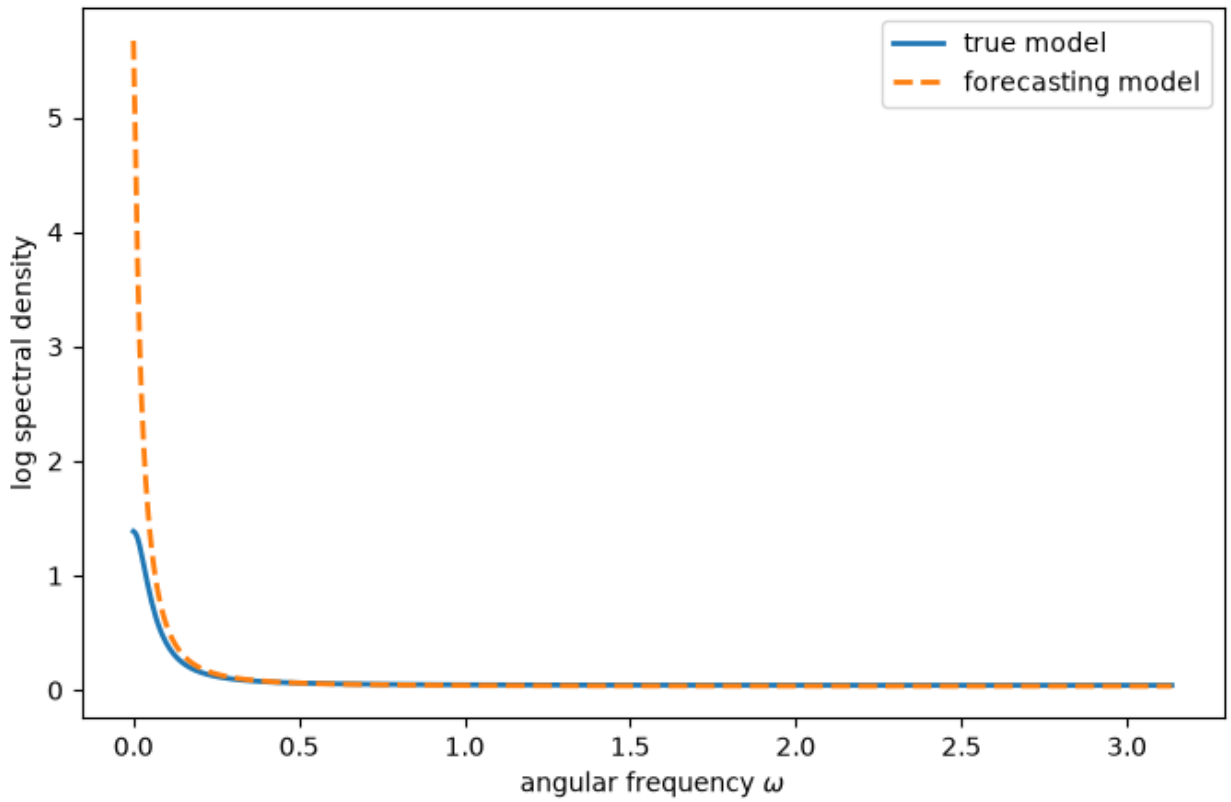


Fig. 95.1: Spectral densities of the true price process and of the agents' approximating model

The true spectral density decreases sharply with frequency — Granger's "typical spectral shape" [Granger, 1966] — revealing substantial positive serial correlation in the price, because agents' belief that the price is subject to permanent shocks makes shocks persist.

95.4.1 Impulse responses

We compare the impulse response functions of the two models by feeding a unit shock through each moving-average representation.

```
def impulse_response(num_roots, den_roots, T=25):
    "IRF of (1 - num L)/(1 - den L): coefficients of the ratio of lag polys."
    h = np.empty(T)
    h[0] = 1.0
    for k in range(1, T):
        h[k] = den_roots * h[k - 1]
    h[1:] -= num_roots * h[:-1]           # apply the numerator (1 - num L)
    return h

ψ = (1 - C_star) / (1 - bray.b * C_star)
scale = 1 / (1 - bray.b * C_star)
irf_true = scale * impulse_response(1 - C_star, ψ)          # f(L)
irf_approx = impulse_response(1 - c_star, bray.ρ)           # g(L)

fig, ax = plt.subplots(figsize=(8, 5))
ax.plot(irf_true, 'C0o-', ms=4, label='true model', lw=2)
ax.plot(irf_approx, 'C1s--', ms=4, label='approximating model', lw=2)
ax.set_xlabel('lag')
ax.set_ylabel('response')
ax.legend()
plt.show()
```

The impulse response of the true model affirms the serial correlation in the price.

The approximating model tends to under-predict the short-term consequences of a shock while over-predicting the long-term ones: its near-unit-root produces a response that does not die out.

95.5 Lessons

The agents in this model are **boundedly rational**: *rational* describes their use of least squares, and *bounded* describes their model misspecification.

Under rational expectations there is only one model in play.

Under bounded rationality there must be at least two: the one used by the boundedly rational agents, and the true one.

These mutually influence each other — the boundedly rational agents use their model to approximate the true one, and the true one reflects the decisions of the agents — and both differ from the rational expectations model.

The peculiar way that the adaptive expectations model uses a unit root to mimic a constant foreshadows a version of the Phillips curve model, developed in *Self-Confirming Equilibria*, that will help vindicate econometric policy evaluation.

This same trick — using a unit root to approximate a constant — turns out to be the engine of the *escape dynamics* of *Adaptive Learning and Escape Dynamics* and *Escaping Nash Inflation*, where a learning government's estimated Phillips curve drifts toward the induction hypothesis and, believing it, cuts inflation toward Ramsey.

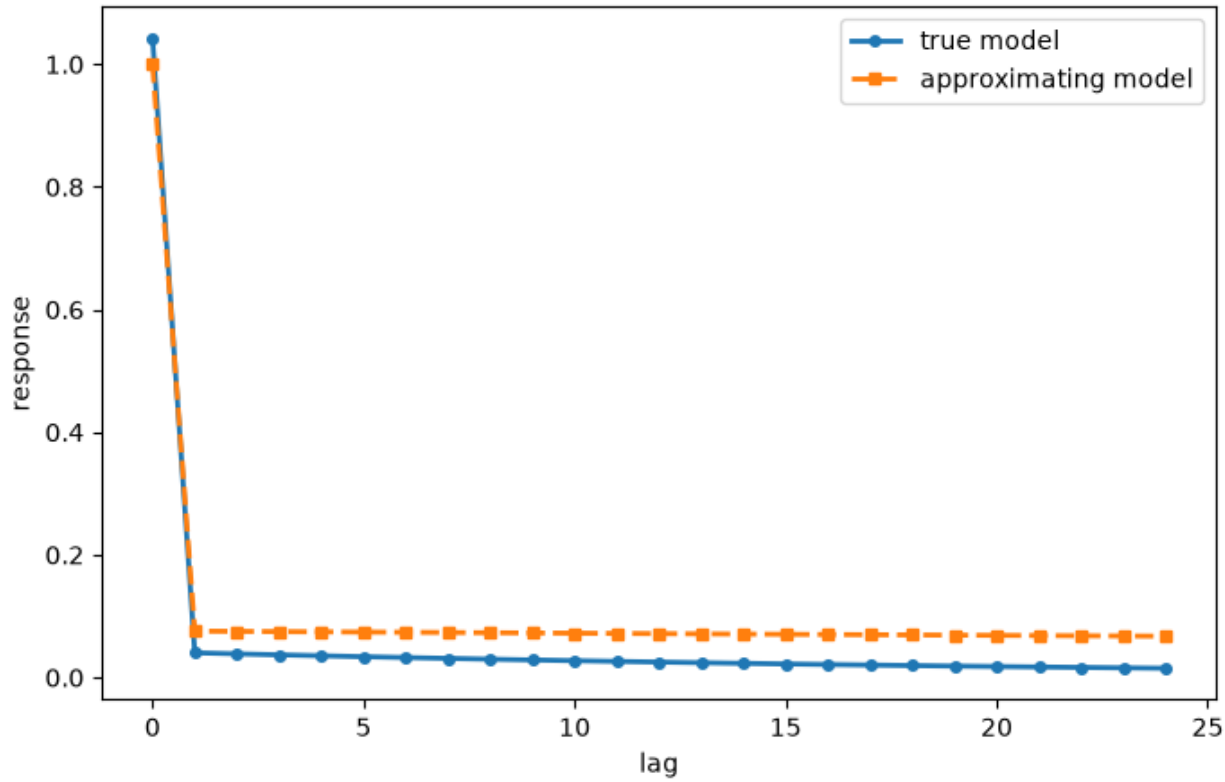


Fig. 95.2: Impulse responses of the true and approximating models

95.6 Exercises

Exercise 95.6.1

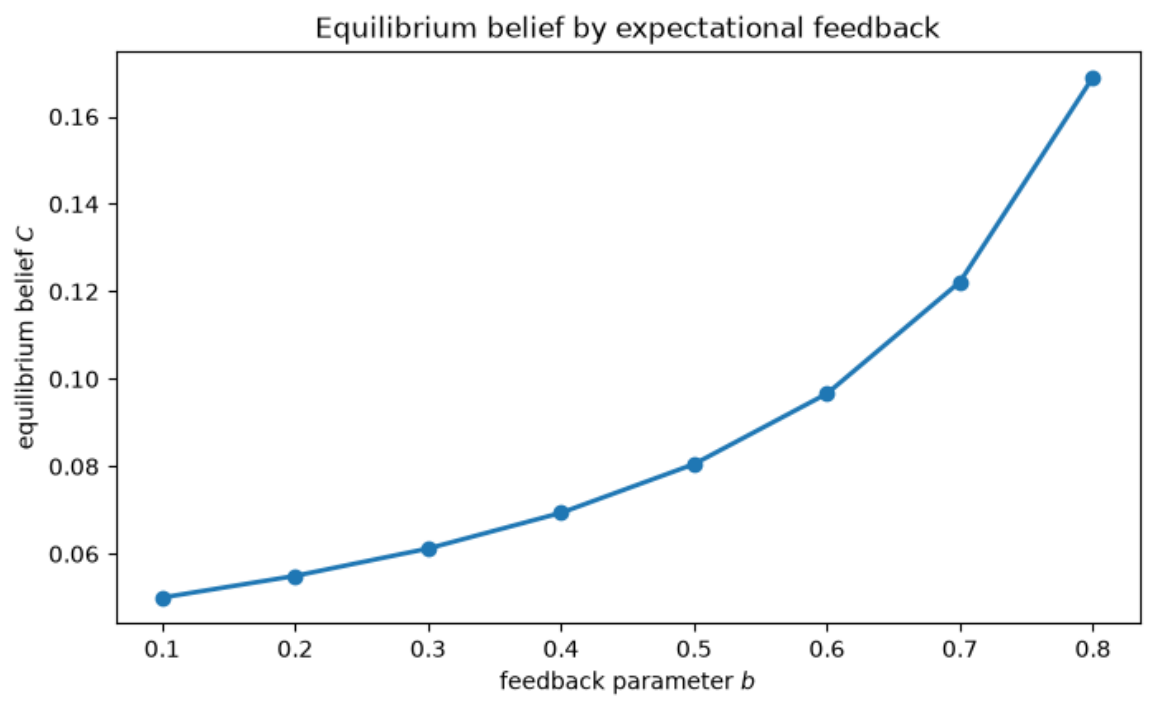
The equilibrium belief C depends on the feedback parameter b in (95.1).

Compute and plot the equilibrium C as a function of b over a grid $b \in \{0.1, 0.2, \ldots, 0.8\}$, holding $a = 1$ and $\sigma_u = 1$ fixed.

How does stronger expectational feedback (larger b) affect the equilibrium amount of discounting of past data?

Solution

```
b_grid = np.arange(0.1, 0.85, 0.1)
C_of_b = [solve_equilibrium(BrayModel(a=1.0, b=b, σ_u=1.0)) for b in b_grid]

fig, ax = plt.subplots(figsize=(8, 4.5))
ax.plot(b_grid, C_of_b, 'o-', lw=2)
ax.set_xlabel('feedback parameter $b$')
ax.set_ylabel('equilibrium belief $C$')
ax.set_title('Equilibrium belief by expectational feedback')
plt.show()
```



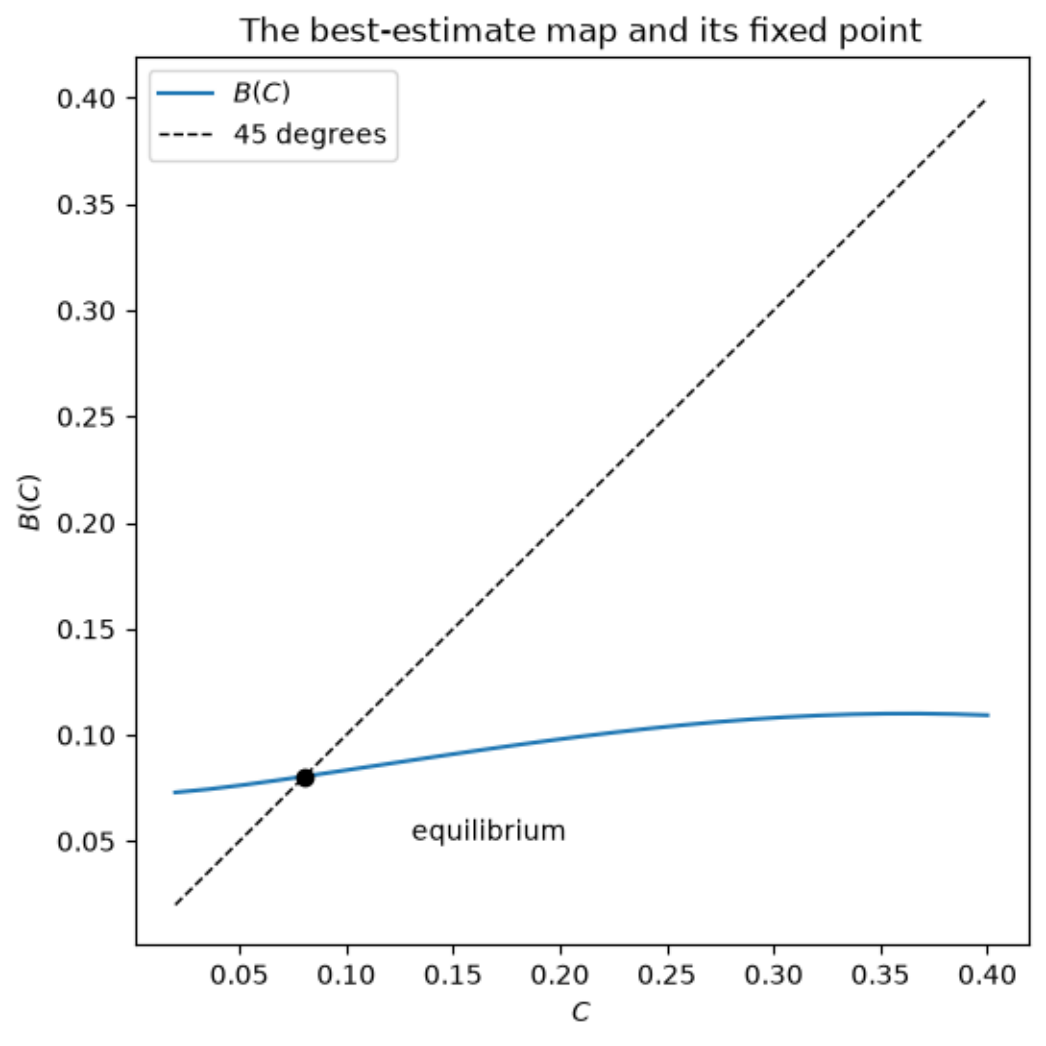
Stronger feedback raises the equilibrium gain C: agents put more weight on recent observations, so the price process they generate is less persistent than it would otherwise be.

Exercise 95.6.2

Verify that the equilibrium is a genuine fixed point by plotting the best-estimate map $c = B(C)$ against the 45-degree line, and mark the fixed point.

Solution

```
C_grid = np.linspace(0.02, 0.4, 25)
B_vals = [best_estimate(bray, C) for C in C_grid]

fig, ax = plt.subplots(figsize=(6, 6))
ax.plot(C_grid, B_vals, 'C0', label='$B(C)$')
ax.plot(C_grid, C_grid, 'k--', lw=1, label='45 degrees')
ax.plot(C_star, C_star, 'ko')
ax.annotate('equilibrium', (C_star, C_star),
            (C_star + 0.05, C_star - 0.03))
ax.set_xlabel('$C$')
ax.set_ylabel('$B(C)$')
ax.set_title('The best-estimate map and its fixed point')
ax.legend()
plt.show()
```



The best-estimate map crosses the 45-degree line at the equilibrium belief, confirming $C = B(C)$.

CHAPTER

NINETYSIX

SELF-CONFIRMING EQUILIBRIA

Contents

- *Self-Confirming Equilibria*
  - *Overview*
  - *Objects in the Phelps problem*
  - *The actual Phillips curve*
  - *Self-confirming equilibria*
  - *The special case solved by hand*
  - *Equilibrium with misspecified beliefs*
  - *Exercises*

In addition to what's in Anaconda, this lecture will use the following library:

```
!pip install quantecon
```

96.1 Overview

This lecture completes the *equilibrium* half of the study begun in *The Credibility Problem*.

It follows chapter 7 of [Sargent, 1999], after which the suite turns from fixed beliefs to beliefs that are learned in real time.

We seek models that depart minimally from the basic [Kydland and Prescott, 1977] model of *The Credibility Problem* but that also let a government's *beliefs* be shaped by the data its own policies generate.

The key equilibrium concept is a **self-confirming equilibrium**: the government has a possibly *wrong* model of the Phillips curve, but it fits that model to the data by least squares, and the data it observes confirm its beliefs.

We combine ideas from two literatures:

- [King and Watson, 1994] document how inferences about the Phillips curve depend on the *direction of fit* — whether one regresses unemployment on inflation (a *classical* identification) or inflation on unemployment (a *Keynesian* identification).
- A literature going back to Muth, Lucas, and Prescott [Lucas and Prescott, 1971, Muth, 1961] formulates rational expectations equilibria as fixed points of mappings from beliefs to the population moments of statistical models.

We build two self-confirming equilibria that are identical except for the direction of fit, and we find that the direction of minimization affects outcomes.

We then study an **equilibrium with misspecified beliefs**, combining the Phelps problem of *Adaptive Expectations and the Phelps Problem* with the optimal-misspecification machinery of *Optimal Misspecified Beliefs*, and find grounds for optimism: outcomes better than Nash, approaching Ramsey as the government becomes patient.

Let's import what we need:

```
import matplotlib.pyplot as plt
import numpy as np
import quantecon as qe
from scipy.optimize import minimize_scalar
```

96.2 Objects in the Phelps problem

Recall from *Adaptive Expectations and the Phelps Problem* the ingredients of the general Phelps problem.

The government believes in a reduced-form Phillips curve that it can fit in either direction:

$$\text{Classical:} \quad U_t = \gamma^\top X_{C,t} + \varepsilon_{C,t}, \qquad X_{C,t} = \begin{bmatrix} y_t & X_{t-1}^\top \end{bmatrix}^\top,$$

$$\text{Keynesian:} \quad y_t = \beta^\top X_{K,t} + \varepsilon_{K,t}, \qquad X_{K,t} = \begin{bmatrix} U_t & X_{t-1}^\top \end{bmatrix}^\top.$$

Solving the Phelps problem takes the government's beliefs γ as given and delivers a decision rule $h(\gamma)$ for inflation.

The two parameterizations are related by the inversion formulas

$$\gamma_1 = \beta_1^{-1}, \qquad \gamma_{-1} = -\beta_{-1}/\beta_1. \tag{96.1}$$

96.3 The actual Phillips curve

The *actual* Phillips curve extends the one used in earlier lectures to allow for serially correlated shocks:

$$U_t = U^* - \frac{\theta}{1-\rho_2 L}(y_t - x_t) + \frac{v_{1t}}{1-\rho_1 L}, \tag{96.2}$$

with $|\rho_1| < 1$, $|\rho_2| < 1$, and $v_t = (v_{1t}, v_{2t})^\top$ a vector white noise, where $v_{2t} \equiv y_t - x_t$ is the surprise in inflation.

For much of this lecture we set $\rho_1 = \rho_2 = 0$ to make theoretical points, which reduces (96.2) to

$$U_t = U^* - \theta(y_t - x_t) + v_{1t}.$$

96.4 Self-confirming equilibria

A self-confirming equilibrium reconciles the government's beliefs with the environment those beliefs generate.

Definition (Self-confirming equilibrium)

A self-confirming equilibrium is a fixed belief vector γ, a government decision rule $h = h(\gamma)$, and a stationary process for (y_t, U_t, x_t) such that

(a) inflation solves the Phelps problem, $y_t = hX_{t-1} + v_{2t}$;

(b) the public optimally forecasts inflation, $x_t = hX_{t-1}$;

© unemployment is generated by the actual Phillips curve (96.2); and

(d) the government's beliefs satisfy the least squares orthogonality conditions $\mathbb{E}\left[U_t - \gamma^\top X_{C,t}\right] X_{C,t}^\top = 0$ (the **classical** direction of fit).

Condition (d) makes the government's beliefs depend on moment matrices that, through (a)–©, themselves depend on the government's beliefs.

The government's beliefs imply behavior that produces data whose moments *confirm* those beliefs.

A distinct self-confirming equilibrium results from replacing (d) with the **Keynesian** direction of fit:

> (d′) the government fits the Keynesian Phillips curve, $\mathbb{E}\left[y_t - \beta^\top X_{K,t}\right] X_{K,t}^\top = 0$, then recovers γ from the inversion formulas (96.1).

Because the government's beliefs affect the whole probability distribution of the data, the direction of minimization affects outcomes.

Note

Computing a self-confirming equilibrium in general means finding a fixed point of a map $\gamma = T(h(\gamma))$ (classical) or $\beta = S(h(\gamma(\beta)))$ (Keynesian).

The moments in the orthogonality conditions are obtained from the state-space representation of the system by solving a discrete Lyapunov equation.

In practice one iterates a relaxation algorithm $\beta_{j+1} = \kappa\beta_j + (1-\kappa)S(\beta_j)$, which resembles the least squares learning recursion of *The Credibility Problem*.

96.5 The special case solved by hand

When $\rho_1 = \rho_2 = 0$, the government's problem collapses to a sequence of static problems, and each self-confirming equilibrium can be computed by hand.

With $X_{t-1} = 1$, the actual Phillips curve implies the second moments

$$\mathbb{V}[U_t] = \theta^2\sigma_2^2 + \sigma_1^2, \qquad \mathbb{V}[y_t] = \sigma_2^2, \qquad \operatorname{cov}(U_t, y_t) = -\theta\sigma_2^2. \tag{96.3}$$

Classical direction of fit (U on y): the slope is

$$\gamma_1 = \frac{\operatorname{cov}(U_t, y_t)}{\mathbb{V}[y_t]} = -\theta,$$

and the requirement that the means lie on the regression line gives the intercept $\gamma_{-1} = (\gamma_1^2 + 1)U^*$.

Keynesian direction of fit (y on U): the slope is

$$\beta_1 = \frac{\operatorname{cov}(U_t, y_t)}{\mathbb{V}[U_t]} = \frac{-\theta\sigma_2^2}{\sigma_1^2 + \theta^2\sigma_2^2},$$

with intercept $\beta_{-1} = -\frac{\beta_1^2+1}{\beta_1}U^*$, from which the implied classical coefficients follow by inversion, $\gamma_1 = \beta_1^{-1}$ and $\gamma_{-1} = \frac{\beta_1^2+1}{\beta_1^2}U^*$.

```
class SelfConfirmingStatic:
    """
    The two static self-confirming equilibria (ρ1 = ρ2 = 0), one for
    each direction of fit.
    """

    def __init__(self, θ=1.0, U_star=5.0, σ1=0.3, σ2=0.3):
        self.θ, self.U_star, self.σ1, self.σ2 = θ, U_star, σ1, σ2

    def classical(self):
        "Perceived Phillips curve U = y_{-1} + γ_1 y under classical fit."
        θ, U_star = self.θ, self.U_star
        γ1 = -θ
        γ_1 = (γ1**2 + 1) * U_star
        y_bar = -γ_1 * γ1 / (γ1**2 + 1)      # mean (= Nash) inflation
        return γ1, γ_1, y_bar

    def keynesian(self):
        "Perceived Phillips curve under Keynesian fit, inverted to γ."
        θ, U_star, σ1, σ2 = self.θ, self.U_star, self.σ1, self.σ2
        β1 = -θ * σ2**2 / (σ1**2 + θ**2 * σ2**2)
        β_1 = -(β1**2 + 1) / β1 * U_star
        γ1 = 1 / β1
        γ_1 = (β1**2 + 1) / β1**2 * U_star
        y_bar = β_1 / (β1**2 + 1)             # mean inflation
        return γ1, γ_1, y_bar
```

```
sce = SelfConfirmingStatic(θ=1.0, U_star=5.0, σ1=0.3, σ2=0.3)

γ1_C, γ0_C, y_C = sce.classical()
γ1_K, γ0_K, y_K = sce.keynesian()

print("Classical direction of fit")
print(f"  γ_1 = {γ1_C:.1f},  γ_(-1) = {γ0_C:.1f},  mean inflation = {y_C:.1f}")
print("Keynesian direction of fit")
print(f"  γ_1 = {γ1_K:.1f},  γ_(-1) = {γ0_K:.1f},  mean inflation = {y_K:.1f}")
```

```
Classical direction of fit
  γ_1 = -1.0,  γ_(-1) = 10.0,  mean inflation = 5.0
Keynesian direction of fit
  γ_1 = -2.0,  γ_(-1) = 25.0,  mean inflation = 10.0
```

These reproduce the numerical example of chapter 7 of [Sargent, 1999].

Under the classical direction of fit, mean inflation is the Nash value $\theta U^* = 5$.

Under the Keynesian direction of fit, the government estimates a *flatter* Phillips curve and believes the tradeoff is more favorable than it is, so it sets inflation twice as high, at 10.

Let's draw the two self-confirming Phillips curves, reproducing Figure 7.1.

```
fig, ax = plt.subplots(figsize=(7, 6))

U_grid = np.linspace(0, 12, 100)

# perceived Phillips curves U = y_{-1} + γ_1 y  =>  y = (U - y_{-1}) / γ_1
ax.plot(U_grid, (U_grid - γ0_C) / γ1_C, 'C0', label='P: classical fit', lw=2)
ax.plot(U_grid, (U_grid - γ0_K) / γ1_K, 'C1', label='Q: Keynesian fit', lw=2)
```

(continues on next page)

(continued from previous page)

```
ax.plot(sce.U_star, y_C, 'C0o')
ax.annotate('Nash', (sce.U_star, y_C), (sce.U_star + 0.4, y_C - 0.6))
ax.plot(sce.U_star, y_K, 'C1o')
ax.annotate('Keynesian mean', (sce.U_star, y_K),
            (sce.U_star + 0.4, y_K + 0.2))

ax.set_xlim(0, 12)
ax.set_ylim(0, 12)
ax.set_xlabel('unemployment $U$')
ax.set_ylabel('inflation $y$')
ax.legend()
plt.show()
```

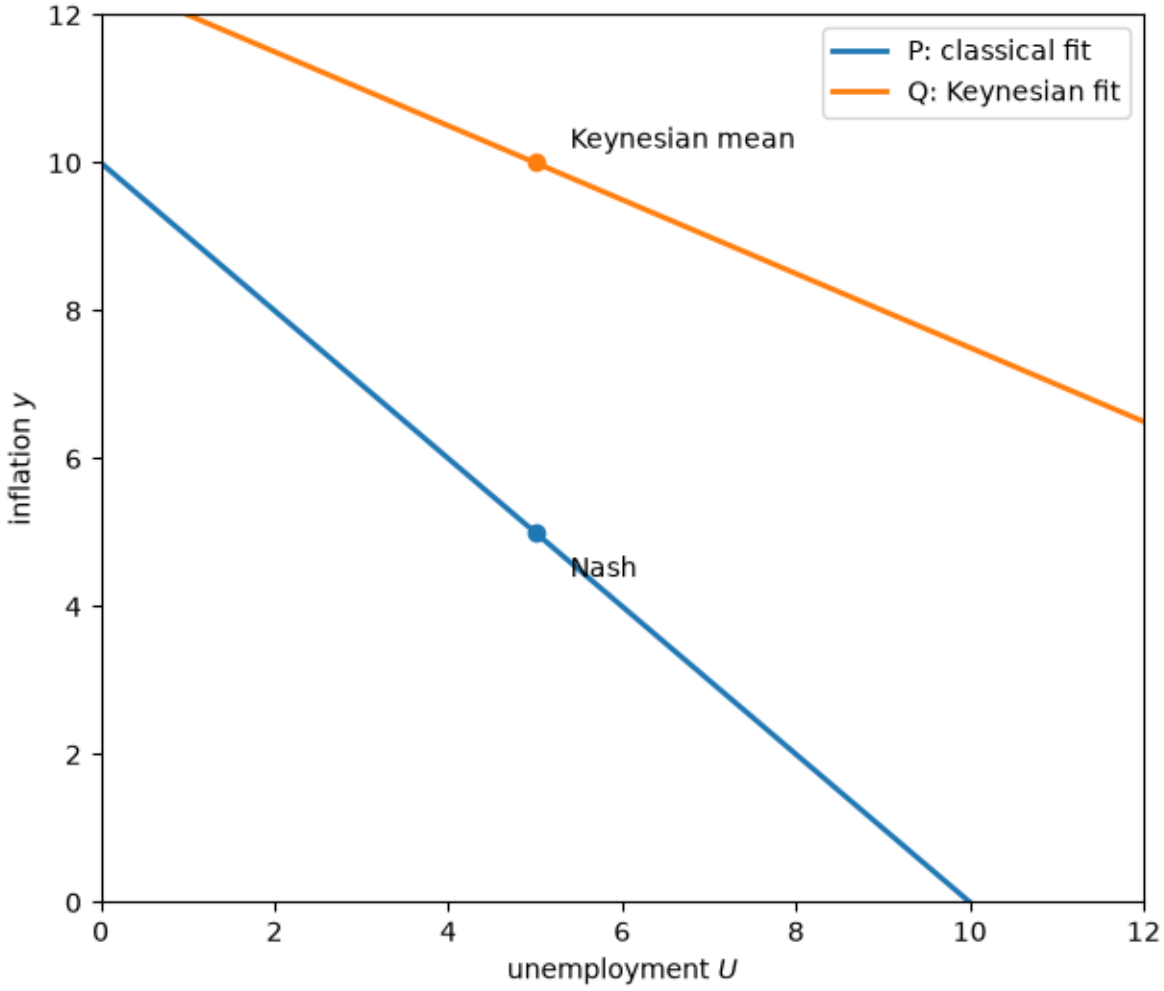


Fig. 96.1: The two self-confirming Phillips curves, one for each direction of fit

Curve P is the self-confirming Phillips curve estimated with the classical direction of fit; curve Q with the Keynesian direction.

Because the Keynesian fit makes the government believe the tradeoff is more exploitable, average inflation is higher.

96.5.1 Why not Ramsey?

Both self-confirming equilibria give a mean outcome *worse* than the Ramsey outcome.

This signifies a breakdown of the hypotheses of the proposition in *Adaptive Expectations and the Phelps Problem* that solving the Phelps problem eventually sustains nearly the Ramsey outcome.

What fails is the **induction hypothesis**.

Because $\rho_1 = \rho_2 = 0$, the self-confirming equilibria are serially uncorrelated, so lagged inflation drops out of the empirical Phillips curves.

Deactivating the induction hypothesis makes the government in effect solve a one-period problem and shuts down the intertemporal channel that promotes better outcomes.

96.6 Equilibrium with misspecified beliefs

To explore how imputing a *different* wrong model might *improve* on the Nash outcome, we now let the *public*, not the government, make a subtle specification error.

This links adaptive expectations, the induction hypothesis, and the Phelps problem.

The government knows the correct model,

$$U_t = U^* - \theta(y_t - x_t) + v_{1t}, \qquad x_t = Cy_{t-1} + (1-C)x_{t-1}, \quad C \in (0,1), \tag{96.4}$$

where the public has constant-gain adaptive expectations with a parameter C that it *tunes to fit the data*.

Taking x_t as a state variable, the government solves the Phelps problem: it maximizes $-\mathbb{E}_0 \sum_{t=0}^{\infty} \delta^t \left[(U^* - \theta(y_t - x_t))^2 + y_t^2\right]$ by choice of a feedback rule $y_t = f_1 + f_2 x_t + v_{2t}$.

This is exactly the LQ Phelps problem of *Adaptive Expectations and the Phelps Problem* with adaptation parameter $\lambda = 1 - C$.

```
def phelps_policy(C, δ, θ=1.0, U_star=5.0):
    "Government feedback rule y_t = f1 + f2 x_t, with public gain C."
    λ = 1 - C
    a = np.array([[U_star], [θ]])
    R = 0.5 * (a @ a.T)
    Q = np.array([[0.5 * (θ**2 + 1)]])
    N = -0.5 * θ * a.T
    A = np.array([[1.0, 0.0], [0.0, λ]])
    B = np.array([[0.0], [1 - λ]])
    lq = qe.LQ(Q, R, A, B, N=N, beta=min(δ, 1 - 1e-7))
    P, F, d = lq.stationary_values()
    return -F[0, 0], -F[0, 1]
```

The government's behavior makes the *actual* inflation rate

$$y_t = \frac{f_1}{1-f_2} + \frac{1-(1-C)L}{1-(1-C(1-f_2))L} v_{2t} = \nu + f(L)v_{2t}, \tag{96.5}$$

with mean $\nu = f_1/(1-f_2)$ and spectrum $F(\omega; C) = |f(e^{i\omega})|^2 \sigma_2^2$.

Given C, the public seeks the best-fitting misspecified model of the same integrated moving-average form as in *Optimal Misspecified Beliefs*,

$$y_t = \frac{1-(1-c)L}{1-L}\epsilon_t = g(L)\epsilon_t,$$

and the induced best-estimate map $c = B(C)$ defines an **equilibrium under forecast misspecification** as a fixed point $C = B(C)$.

This makes both the true and the approximating models equilibrium outcomes, and turns the adaptation parameter C into an outcome rather than a free parameter.

```
class MisspecifiedPhillips:
    """
    Equilibrium with misspecified public beliefs: a fixed point of the
    best-estimate map, with the government solving the Phelps problem.
    """

    def __init__(self, θ=1.0, U_star=5.0, σ2=0.3, δ=0.97, ρ=0.995, N=1024):
        self.θ, self.U_star, self.σ2 = θ, U_star, σ2
        self.δ, self.ρ, self.N = δ, ρ, N
        ω = 2 * np.pi * np.arange(N) / N
        self.z = np.exp(1j * ω)
        self.ω = ω

    def true_process(self, C):
        "Return the mean ν, spectrum F, and policy (f1, f2) given belief C."
        f1, f2 = phelps_policy(C, self.δ, self.θ, self.U_star)
        ν = f1 / (1 - f2)
        ψ = 1 - C * (1 - f2)
        f = (1 - (1 - C) * self.z) / (1 - ψ * self.z)
        F = np.abs(f)**2 * self.σ2**2
        return ν, F, (f1, f2)

    def best_estimate(self, C):
        "The best-estimate map c = B(C)."
        ν, F, _ = self.true_process(C)
        z, ρ = self.z, self.ρ

        def profile(c):
            H = np.abs((1 - (1 - c) * z) / (1 - ρ * z))**2
            σ_ε2 = np.mean(F / H) + ν**2 / H[0]
            return np.log(σ_ε2) + np.mean(np.log(H))

        return minimize_scalar(profile, bounds=(1e-4, 0.99),
                               method='bounded').x

    def solve(self, C0=0.1, tol=1e-10, maxit=500):
        "Iterate the best-estimate map to a fixed point."
        C = C0
        for _ in range(maxit):
            C_new = self.best_estimate(C)
            if abs(C_new - C) < tol:
                break
            C = C_new
        ν, _, (f1, f2) = self.true_process(C_new)
        return C_new, ν, (f1, f2)
```

```
mp = MisspecifiedPhillips(δ=0.97)
C_star, ν_star, (f1, f2) = mp.solve()

print(f"equilibrium gain     C = {C_star:.4f}")
print(f"policy rule          y = {f1:.4f} + {f2:.4f} x")
print(f"mean inflation       ν = {ν_star:.3f}")
```

(continues on next page)

(continued from previous page)

```
print(f"Nash inflation    θ U*  = {mp.θ * mp.U_star:.3f}")
```

```
equilibrium gain      C = 0.1138
policy rule           y = 0.7109 + 0.3348 x
mean inflation        ν = 1.069
Nash inflation    θ U*  = 5.000
```

The most important outcome is that the implied mean inflation rate is substantially *below* the Nash value of 5.

The induction hypothesis embedded in the adaptive expectations scheme, together with a high discount factor, delivers this improvement — and because C is now an equilibrium outcome, the mechanism is sharper than in *Adaptive Expectations and the Phelps Problem*, where C could be manipulated independently of δ.

96.6.1 Approaching Ramsey

As the government becomes more patient, the equilibrium mean inflation rate falls toward the Ramsey value of zero.

```
δ_grid = np.array([0.95, 0.96, 0.97, 0.98, 0.99, 0.995])
C_vals, ν_vals = [], []
for δ in δ_grid:
    C, ν, _ = MisspecifiedPhillips(δ=δ).solve()
    C_vals.append(C)
    ν_vals.append(ν)

fig, axes = plt.subplots(1, 2, figsize=(11, 4.5))
axes[0].plot(δ_grid, ν_vals, 'o-')
axes[0].axhline(mp.θ * mp.U_star, color='k', ls='--', lw=1, label='Nash')
axes[0].set_xlabel(r'discount factor $\delta$')
axes[0].set_ylabel('mean inflation')
axes[0].legend()

axes[1].plot(δ_grid, C_vals, 'o-', color='C1', lw=2)
axes[1].set_xlabel(r'discount factor $\delta$')
axes[1].set_ylabel('equilibrium gain $C$')

plt.tight_layout()
plt.show()
```

Mean inflation lies far below the Nash value at every discount factor and declines toward the Ramsey value of zero as $\delta \to 1$.

Note

The precise equilibrium values depend on the near–unit-root approximation ρ used to keep the perceived model's spectral density well defined, as discussed in *Optimal Misspecified Beliefs*.

The qualitative conclusion — better-than-Nash outcomes that approach Ramsey as $\delta \to 1$ — is robust.

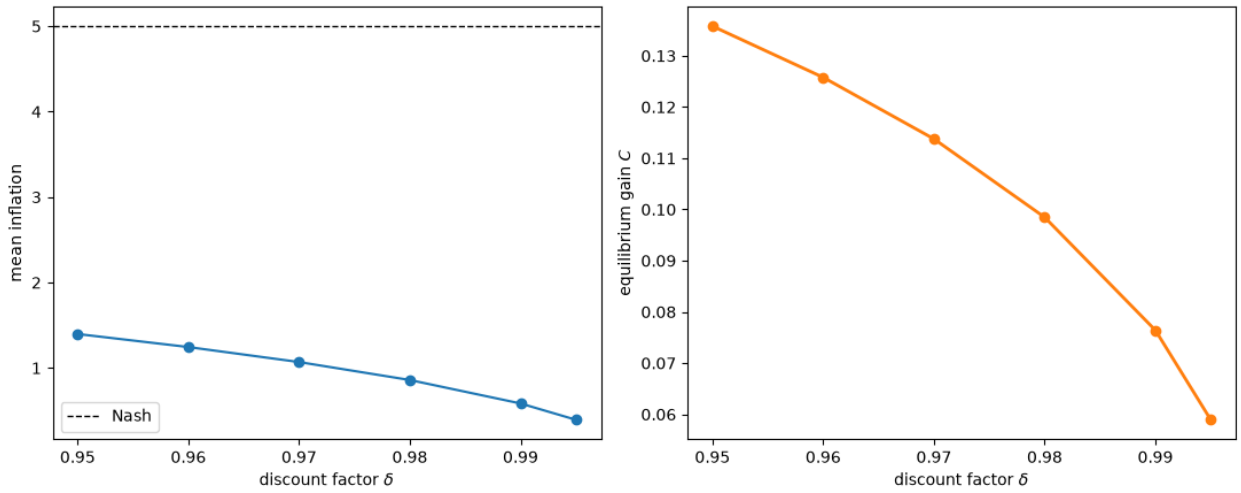


Fig. 96.2: Mean inflation and the equilibrium gain as the government becomes more patient

96.6.2 Spectra and impulse responses

Let's compare the true and approximating inflation processes at the equilibrium, as in Figures 7.2 and 7.3 of [Sargent, 1999].

```
ν_star, F, _ = mp.true_process(C_star)
c_star = mp.best_estimate(C_star)
H = np.abs((1 - (1 - c_star) * mp.z) / (1 - mp.ρ * mp.z))**2
σ_ε2 = np.mean(F / H) + ν_star**2 / H[0]
G = H * σ_ε2

half = mp.N // 2
fig, ax = plt.subplots(figsize=(8, 5))
ax.plot(mp.ω[:half], np.log(F[:half]), 'C0', label='true model', lw=2)
ax.plot(mp.ω[:half], np.log(G[:half]), 'C1--', label='approximating model', lw=2)
ax.set_xlabel(r'angular frequency $\omega$')
ax.set_ylabel('log spectral density')
ax.legend()
plt.show()
```

The true and approximating spectral densities match well at all but the lowest frequencies.

The true inflation rate is only moderately serially correlated, and — as in the Bray model of *Optimal Misspecified Beliefs* — the approximating model uses a unit root to simulate a mean, capturing first moments with second moments.

```
def ima_impulse(num, den, T=25):
    "IRF of (1 - num L)/(1 - den L)."
    h = np.empty(T)
    h[0] = 1.0
    for k in range(1, T):
        h[k] = den * h[k - 1]
    h[1:] -= num * h[:-1]
    return h

ψ = 1 - C_star * (1 - f2)
```

(continues on next page)

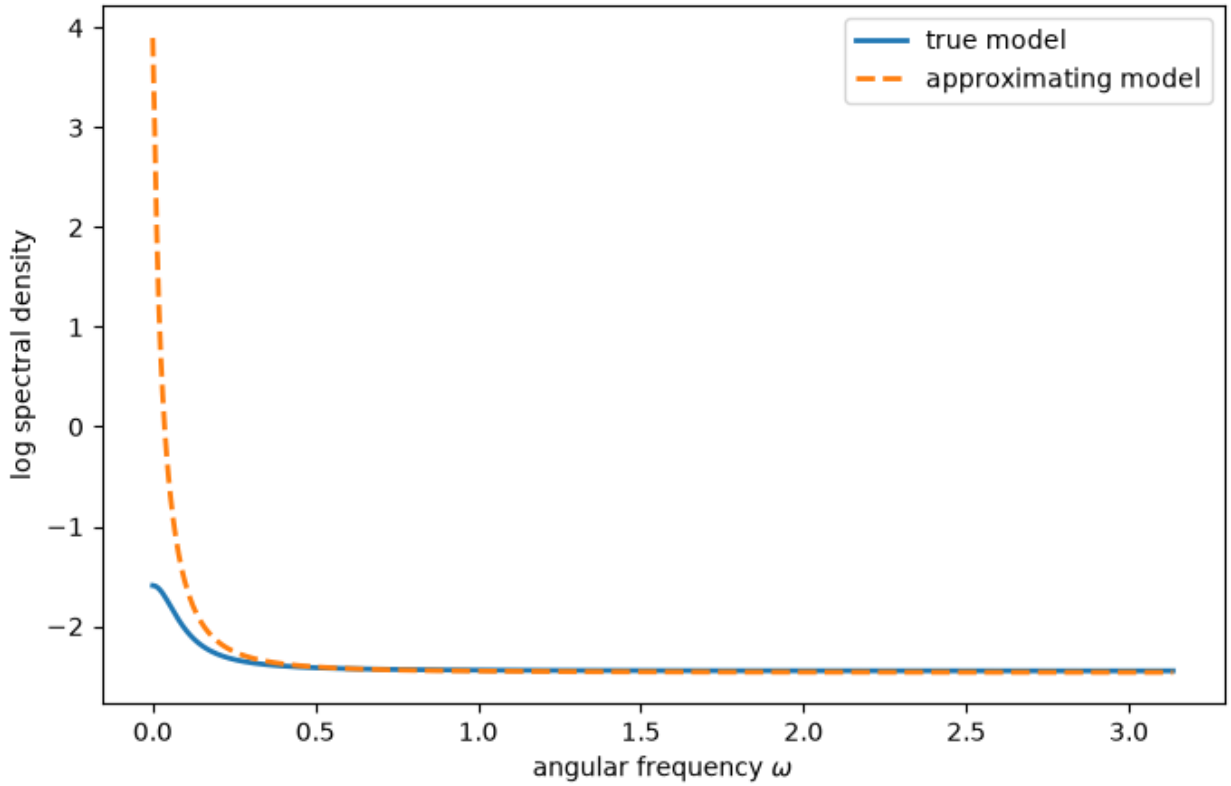


Fig. 96.3: Spectral densities of the true and approximating inflation processes at the equilibrium

(continued from previous page)

```
irf_true = ima_impulse(1 - C_star, ψ)
irf_approx = ima_impulse(1 - c_star, mp.ρ)

fig, ax = plt.subplots(figsize=(8, 5))
ax.plot(irf_true, 'C0o-', ms=4, label='true model', lw=2)
ax.plot(irf_approx, 'C1s--', ms=4, label='approximating model', lw=2)
ax.set_xlabel('lag')
ax.set_ylabel('response')
ax.legend()
plt.show()
```

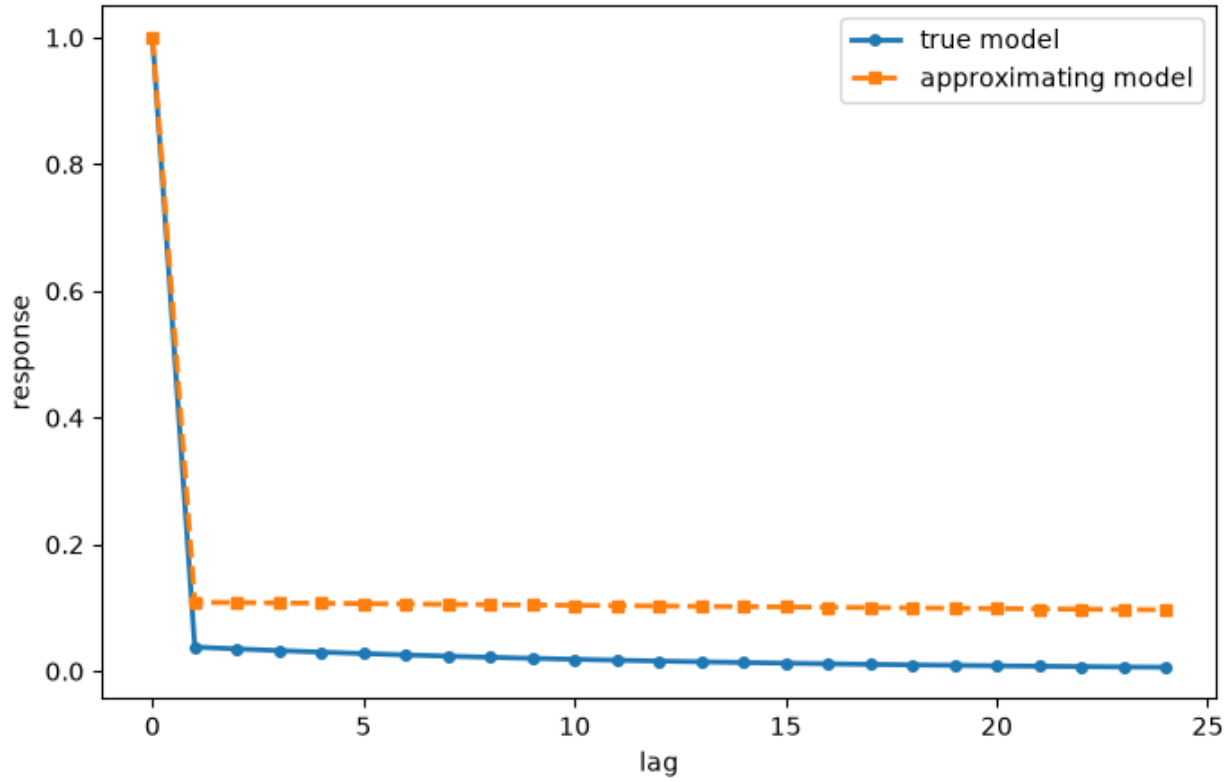


Fig. 96.4: Impulse responses of the true and approximating inflation processes

The unit root in the approximating model manifests itself as a nonzero asymptote in its impulse response.

96.6.3 Grounds for optimism

After the disappointments of the self-confirming equilibria, the equilibrium with forecast misspecification is heartening: it supports better-than-Nash outcomes.

The equilibrium concept is not self-confirming, but it has that spirit — it embodies a type of self-confirmation with a *wrong* model.

That the approximation error is small shows there is a nearly self-confirming model with much better than Nash outcomes.

The next lectures in this suite — *Adaptive Learning and Escape Dynamics*, *Escaping Nash Inflation*, and *Priors, Escapes, and Learning Cycles* — build adaptive, real-time versions of these models in which the government estimates its Phillips curve recursively from the most recent data.

There the self-confirming equilibrium computed here becomes the *attractor* of the learning dynamics, and the better-than-Nash outcomes reappear as recurrent *escapes* away from it.

96.7 Exercises

Exercise 96.7.1

The gap between the two static self-confirming equilibria is driven by the variances σ_1 (the Phillips curve shock) and σ_2 (the inflation surprise).

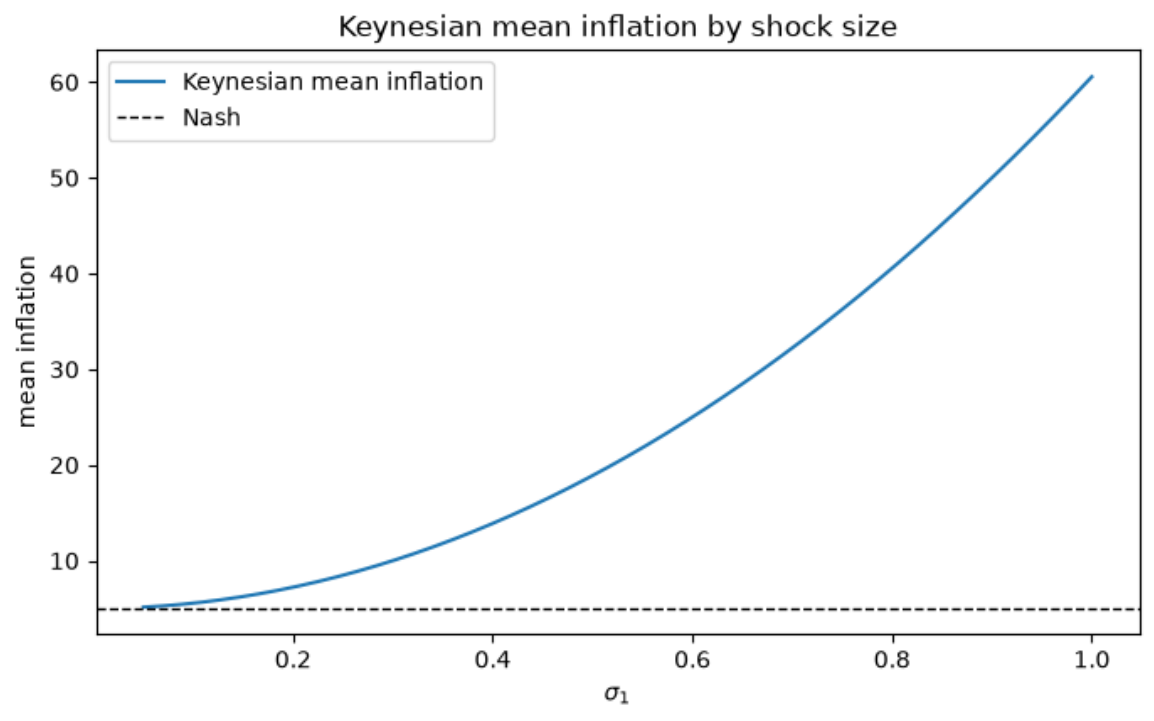
The classical equilibrium always delivers mean inflation equal to the Nash value θU^*, but the Keynesian mean inflation depends on the ratio σ_1/σ_2.

Fixing $\sigma_2 = 0.3$, $\theta = 1$, $U^* = 5$, plot Keynesian mean inflation as a function of $\sigma_1 \in [0.05, 1.0]$.

What happens as $\sigma_1 \to 0$, and why?

Solution

```
σ1_grid = np.linspace(0.05, 1.0, 50)
y_keynes = [SelfConfirmingStatic(σ1=σ1, σ2=0.3).keynesian()[2]
            for σ1 in σ1_grid]

fig, ax = plt.subplots(figsize=(8, 4.5))
ax.plot(σ1_grid, y_keynes, label='Keynesian mean inflation')
ax.axhline(5.0, color='k', ls='--', lw=1, label='Nash')
ax.set_xlabel(r'$\sigma_1$')
ax.set_ylabel('mean inflation')
ax.set_title('Keynesian mean inflation by shock size')
ax.legend()
plt.show()
```



As $\sigma_1 \to 0$ the Phillips curve becomes nearly deterministic, the Keynesian slope $\beta_1 \to -1/\theta$, and the two directions of fit agree, so Keynesian mean inflation approaches the Nash value.

As σ_1 grows, unemployment becomes noisier, the Keynesian regression of y on U flattens, and the government perceives a more exploitable tradeoff, raising mean inflation.

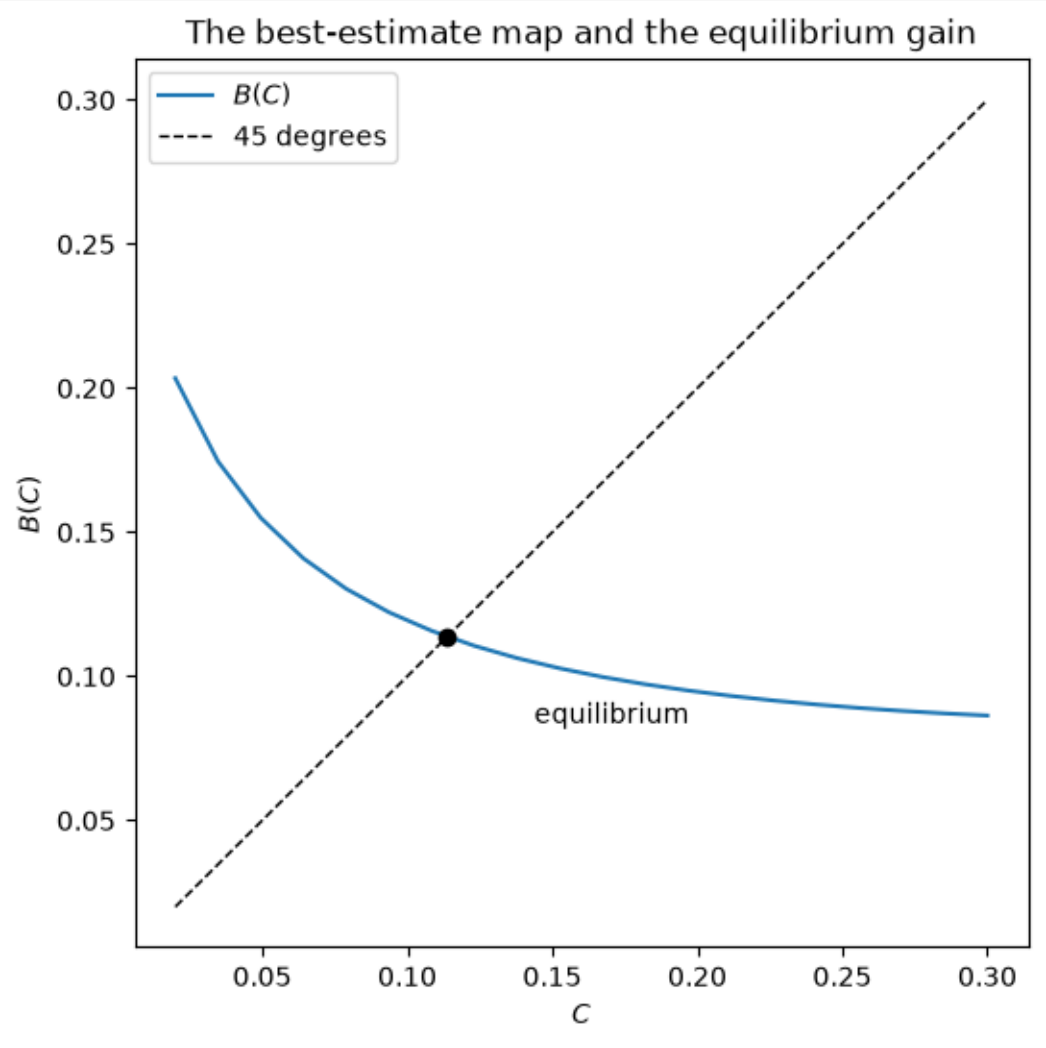
Exercise 96.7.2

Verify that the misspecified-beliefs equilibrium is a genuine fixed point.

For $\delta = 0.97$, plot the best-estimate map $C \mapsto B(C)$ against the 45-degree line and mark the fixed point.

Solution

```
mp = MisspecifiedPhillips(δ=0.97)
C_grid = np.linspace(0.02, 0.3, 20)
B_vals = [mp.best_estimate(C) for C in C_grid]
C_star, _, _ = mp.solve()

fig, ax = plt.subplots(figsize=(6, 6))
ax.plot(C_grid, B_vals, 'C0', label='$B(C)$')
ax.plot(C_grid, C_grid, 'k--', lw=1, label='45 degrees')
ax.plot(C_star, C_star, 'ko')
ax.annotate('equilibrium', (C_star, C_star), (C_star + 0.03, C_star - 0.03))
ax.set_xlabel('$C$')
ax.set_ylabel('$B(C)$')
ax.set_title('The best-estimate map and the equilibrium gain')
ax.legend()
plt.show()
```



The best-estimate map crosses the 45-degree line at the equilibrium gain, confirming $C = B(C)$.

CHAPTER

NINETYSEVEN

ADAPTIVE LEARNING AND ESCAPE DYNAMICS

Contents

- *Adaptive Learning and Escape Dynamics*
  - *Overview*
  - *A primer on recursive algorithms*
  - *The adaptive model*
  - *Least squares learning converges*
  - *Constant gain and escape dynamics*
  - *The escape route and the induction hypothesis*
  - *Relation to equilibria under forecast misspecification*
  - *Role of the discount factor*
  - *Anticipated utility*
  - *Conclusions*
  - *Exercises*

97.1 Overview

This lecture is the culmination of the theory built by [Sargent, 1999].

It follows chapter 8, the most ambitious chapter of the book, and everything after it in this suite either sharpens its analytics (*Escaping Nash Inflation*, *Priors, Escapes, and Learning Cycles*), carries its tools to a later episode (*The Lost Conquest: Fed Policy in the 2020s*), or puts its central claim to an empirical test (*Drifts and Volatilities*).

In *Self-Confirming Equilibria* a government held *fixed* beliefs about the Phillips curve — beliefs that were confirmed by the data those beliefs generated.

Here we make the government a real-time econometrician.

Each period it

- re-estimates its Phillips curve by recursive least squares from the data seen so far, and
- sets inflation at the first-period recommendation of the *Phelps problem* for its *current* estimate.

We ask whether such an adaptive government converges to a self-confirming equilibrium.

The answer depends on a single parameter — the **gain** that governs how fast old data are discounted:

- With a *decreasing* gain that implements least squares, the mean dynamics pull the economy to a self-confirming equilibrium, and we get nothing new: the system is stuck near the Nash outcome.
- With a *constant* gain, agents discount past data, convergence is arrested, and **new outcomes emerge** as the system recurrently *escapes* the self-confirming equilibrium toward the Ramsey (zero-inflation) outcome, in spontaneous stabilizations that resemble the arrival of Volcker.

These escapes are the heart of the *vindication of econometric policy evaluation* story from *The Rise and Fall of U.S. Inflation*: an adaptive government, learning a Solow-Tobin distributed-lag version of the natural-rate hypothesis, is led by chance observations to stabilize inflation.

This lecture replicates, analyzes, and reinterprets simulations like those of Christopher Sims and Heetaik Chung [Chung, 1990, Sims, 1988].

Here we study the escapes by *simulation*; *Escaping Nash Inflation* then characterizes them analytically as a second, deterministic ODE, and *Priors, Escapes, and Learning Cycles* asks how the government's prior about coefficient drift reshapes both forces.

Let's import what we need:

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.linalg import solve_discrete_are
```

97.2 A primer on recursive algorithms

This section is a self-contained primer.

It develops the two analytical objects — **mean dynamics** and **escape routes** — that organize everything that follows, and it explains the sense in which one and the same recursion can be read either as an *algorithm* for computing a self-confirming equilibrium or as a *model* of a government adapting in real time.

The material is technical, and readers who want the punchline can skip to the simulations below and refer back as needed.

97.2.1 Beliefs, moment conditions, and self-confirming equilibria

A self-confirming equilibrium under the classical identification is pinned down by the government's beliefs about some population moments and the regression coefficients they imply.

Under the classical identification these beliefs are measured by the triple $(\gamma,\ EX_C X_C^\top,\ EUX_C)$, where γ is the vector of Phillips-curve coefficients.

In the adaptive models of this lecture the *time-t values* of these objects are among the economy's state variables; they disappear as state variables in a self-confirming equilibrium only because there they are constants.

A self-confirming equilibrium under the classical identification satisfies the moment conditions

$$\begin{aligned} \mathbb{E}\, R_{XC}^{-1}(\gamma)\left[U_t X_{Ct}^\top - (X_{Ct}X_{Ct}^\top)\,\gamma\right] &= 0, \\ \mathbb{E}\, X_{Ct}X_{Ct}^\top - R_{XC}(\gamma) &= 0, \end{aligned} \tag{97.1}$$

where the mathematical expectation is taken with respect to a distribution of (U_t, X_{Ct}) that depends on γ through the solution $h(\gamma)$ of the Phelps problem.

Self-reference surfaces precisely in this dependence of the distribution on γ: the government's beliefs shape its policy, which shapes the data on which the beliefs are then checked.

The first line of (97.1) is the least squares normal equations for γ, pre-multiplied by the inverse second-moment matrix; the second line defines R_{XC} as the second-moment matrix of the regressors.

It is convenient to assemble all the unknowns into a single vector

$$\phi = \begin{bmatrix} \gamma \\ \mathrm{col}(R_{XC}) \end{bmatrix}, \tag{97.2}$$

where $\mathrm{col}(R_{XC})$ stacks the columns of R_{XC}.

The moment conditions (97.1) then take the compact form

$$\mathbb{E}\left[F(\phi, \zeta)\right] = 0, \qquad b(\phi) \equiv \mathbb{E}\left[F(\phi, \zeta)\right], \tag{97.3}$$

where ζ is a random vector and the expectation is over its distribution (which, again, depends on ϕ).

A self-confirming equilibrium is a set of beliefs ϕ_f that is a zero of b,

$$b(\phi_f) = 0. \tag{97.4}$$

The rest of this section describes recursive algorithms for finding such a zero, and a change of perspective that converts each computational algorithm into a model of real-time adaptation.

97.2.2 Iteration

The simplest algorithm computes a sequence $\{\phi_k\}$ of estimates from

$$\phi_{k+1} = \phi_k + a\, b(\phi_k), \tag{97.5}$$

where the distribution used to evaluate the expectation defining $b(\phi_k)$ in (97.3) is itself evaluated at the current estimate ϕ_k, and $a > 0$ is a step size.

This is the relaxation algorithm used to compute self-confirming equilibria in *Self-Confirming Equilibria*.

Each step requires evaluating the mathematical expectation $b(\phi) = \mathbb{E}[F(\phi, \zeta)]$ — which is exactly why we needed the moment (Lyapunov) formulas there.

97.2.3 Stochastic approximation

A random version of (97.5) is obtained by replacing the mean $b(\phi_n)$ by a single random draw $F(\phi_n, \zeta_n)$ and letting the *step size* do the averaging:

$$\phi_{n+1} = \phi_n + a_n F(\phi_n, \zeta_n), \qquad a_n > 0, \quad \sum_{n=0}^{\infty} a_n = +\infty. \tag{97.6}$$

To study the limiting behavior of (97.6), define **artificial time**

$$t_n = \sum_{k=0}^{n} a_k, \tag{97.7}$$

form the sampled process $\phi(t_n) = \phi_n$, and interpolate (typically piecewise-linearly) to obtain a continuous-time process $\phi^o(t)$.

One then approximates $\phi^o(t)$ by a continuous-time process as $n \to \infty$ and uses it to characterize the tail of the original sequence.

Different rates of decrease of the gain sequence $\{a_n\}$ produce different approximating processes, because they change the mapping (97.7) from real time n to artificial time t_n.

Note

Recursive stochastic approximation originates with [Robbins and Monro, 1951], who devised (97.6) to find the root of a regression function observed with noise, and with [Kiefer and Wolfowitz, 1952], who adapted it to find the maximum of a regression function (the "K-W" algorithms referred to below).

The "ODE method" for analyzing such recursions — approximating the interpolated process by the solution of a differential equation — is due to [Ljung, 1977]; book-length treatments are [Benveniste *et al.*, 1990] and [Kushner and Yin, 2003].

Its use to study learning in self-referential macroeconomic models is developed by [Marcet and Sargent, 1989] and, comprehensively, by [Evans and Honkapohja, 2001].

97.2.4 Mean dynamics

The classic stochastic approximation algorithms of [Kushner and Clark, 1978] and [Ljung, 1977] set the gain to decrease like $a_n \sim 1/n$ (at least for $t \geq N$ for some $N > 0$).

This permits strong statements about the almost sure convergence of (97.6) to a zero of $b(\phi)$.

For $a_n \sim 1/n$, as $n \to \infty$ the interpolated process $\phi^o(t)$ approaches the solution of the ordinary differential equation

$$\frac{d\phi^o(t)}{dt} = b\left(\phi^o(t)\right), \tag{97.8}$$

which we call the **mean dynamics** — the vector generalization of the scalar least-squares-learning ODE derived in the appendix of *The Credibility Problem*.

A law of large numbers makes the random term in the continuous-time approximation vanish fast enough that the mean dynamics (97.8) describe the *tail* behavior of the stochastic process (97.6).

Consequently:

- if the algorithm converges (almost surely), it converges to a zero of the mean dynamics, $b(\phi) = 0$ — a self-confirming equilibrium; and
- the ODE (97.8) carries the information about local and global stability of the algorithm.

Local stability is governed by the eigenvalues of the Jacobian of b at a rest point: if all eigenvalues have negative real parts the rest point is locally stable, and, under conditions on the gain, the eigenvalue of largest real part governs the *rate* of convergence (the usual $\sqrt{T}$ rate requires that eigenvalue to be below $-\frac{1}{2}$).

We will see below that for the classical model at our parameters this eigenvalue sits exactly at the $-\frac{1}{2}$ boundary, so convergence is marginal — a fact that turns out to matter for how readily the system escapes.

Note

[Brock and Hommes, 1997] build models whose global behavior is driven by stable mean dynamics far from rational expectations equilibria together with local instability of adaptation near them — a complementary mechanism for generating endogenous fluctuations from learning.

97.2.5 Constant gain and convergence in distribution

We are equally interested in versions of (97.6) with a *constant* gain $a_n = \epsilon > 0$ for all n.

Limit theorems for constant-gain algorithms use a weaker notion of convergence — convergence *in distribution* — than the almost-sure convergence available when $a_n \sim 1/n$.

They concern small-noise limits, taken as $\epsilon \to 0$ and $n\epsilon \to +\infty$ simultaneously.

Again using artificial time (97.7), form the family of processes

$$\phi^\epsilon_{n+1} = \phi^\epsilon_n + \epsilon\, F(\phi^\epsilon_n, \zeta_n), \tag{97.9}$$

interpolate to obtain $\phi^\epsilon(t)$, and study its small-ϵ limit.

Dupuis and Kushner, [Kushner and Yin, 2003], and others verified conditions under which, as $\epsilon \to 0$ and $\epsilon n \to \infty$, the process ϕ^ϵ_n converges *in distribution* to the zeros of the same mean dynamics (97.8); the restrictions on the mean dynamics needed for convergence match those from the classic $a_n \sim 1/n$ theory.

Unlike the decreasing-gain algorithm, a constant-gain algorithm does not settle down: (γ, R_{XC}) converges to a *stationary stochastic process*, perpetually fluctuating around — and occasionally far from — the self-confirming equilibrium.

97.2.6 Escape routes and the theory of large deviations

The feature of the constant-gain apparatus that matters most for this lecture is not convergence toward ϕ_f but the *excursions away from it*.

We are as interested in movements away from a self-confirming equilibrium as in those toward one, because the recurrent stabilizations in the simulations are precisely such excursions.

The **theory of large deviations** characterizes these excursions through three objects.

First, the log moment generating function of (an averaged version of) the innovation process $F(\phi_n, \zeta_n)$: for a vector θ conformable to F,

$$H(\theta, \phi) = \log \mathbb{E} \exp\left(\theta^\top F(\phi, \zeta)\right), \tag{97.10}$$

where the expectation is over the distribution of ζ.

Note

Equation (97.10) is a heuristic shorthand.

The object that actually enters the theory is a *time-averaged* limit; [Dupuis and Kushner, 1987] and [Kushner and Yin, 2003] assume that for each $\delta > 0$ the following limit exists uniformly in ϕ_i, α_i on any compact set: $$\sum_{i=0}^{T/\delta-1} \delta\, H(\alpha_i, \phi_i) = \lim_{N\to\infty} \tfrac{\delta}{N} \log \mathbb{E} \exp \sum_{i=0}^{T/\delta-1} \alpha_i^\top \sum_{j=iN}^{iN+N-1} F(\phi_i, \zeta_j).$$ The inner sum averages the innovations over a block of length N; the double limit lets us treat serially dependent innovations.

Second, the **Legendre transform** of H, which plays the role of a rate function:

$$L(\beta, \phi) = \sup_\theta \left[\theta^\top \beta - H(\theta, \phi)\right]. \tag{97.11}$$

Third, the **action functional**, which measures the "cost" of a candidate escape path $\phi(\cdot)$:

$$S(T, \phi) = \begin{cases} \displaystyle\int_0^T L\big(\tfrac{d}{ds}\phi(s),\, \phi(s)\big)\, ds & \text{if } \phi(s) \text{ is absolutely continuous and } \phi(0) = \phi_f, \\ \infty & \text{otherwise.} \end{cases} \tag{97.12}$$

Dupuis and Kushner turn the search for the most likely escape into a *deterministic control problem*.

Let D be a compact set containing ϕ_f, with boundary ∂D, and let $C[0,T]$ be the continuous functions on $[0,T]$.

The escape route is the path $\tilde{\phi}(\cdot)$ that solves

$$\inf_{T>0} \inf_{\phi \in A} S(T,\phi), \qquad A = \{\phi(\cdot) \in C[0,T] : \phi(T) \in \partial D\}. \tag{97.13}$$

Assuming the minimizer $\tilde{\phi}(\cdot)$ is unique, and letting t_D^ϵ be the first time the constant-gain process $\phi^\epsilon(t)$ leaves D, [Dupuis and Kushner, 1987] show that for every $\delta > 0$

$$\lim_{\epsilon \to 0} \text{Prob}\left(\left|\phi^\epsilon(t_D^\epsilon) - \tilde{\phi}(T)\right| > \delta\right) = 0. \tag{97.14}$$

In words: *conditional on escaping the set D, the system leaves it near the terminal point of the least-action path* — so the escape has a deterministic direction and shape, even though it is triggered by chance.

This is the sense in which the escapes below, though they require no large shock, "seem purposeful": they follow the least-action route dictated by (97.13).

A crucial contrast with the mean dynamics: the mean dynamics (97.8) do *not* depend on the noise around them, whereas the escape routes *do* — the noise not only adds random fluctuations around (97.8), it also carves out this second family of paths.

Note

The mathematical foundations are the large-deviation theory for randomly perturbed dynamical systems of [Freidlin and Wentzell, 1998] (especially their chapter 4), specialized to stochastic approximation by [Dupuis and Kushner, 1987] and [Dupuis and Kushner, 1989]; a weak-convergence treatment is [Dupuis and Ellis, 1997].

97.2.7 A tractable action functional

The escape-route calculation promises cheap information about central tendencies of the algorithm, but the action functional (97.12) is generally hard to compute.

An important special case simplifies it dramatically.

Suppose the innovation is additive and Gaussian,

$$F(\phi, \zeta) = b(\phi) + \sigma(\phi)\,\zeta,$$

where ζ_n is stationary and Gaussian but not necessarily serially uncorrelated, and define $R = \sum_j \mathbb{E}\,\zeta_t \zeta_{t-j}^\top$.

Then the action functional takes the quadratic form

$$S(T,\phi) = \frac{1}{2}\int_0^T \left(\tfrac{d}{ds}\phi - b(\phi)\right)^\top \left[\sigma(\phi)\,R\,\sigma(\phi)^\top\right]^+ \left(\tfrac{d}{ds}\phi - b(\phi)\right) h(s)\,ds, \tag{97.15}$$

where $(\cdot)^+$ is the Moore-Penrose generalized inverse (used to handle possible stochastic singularity of $\sigma R \sigma^\top$).

The weight $h(s)$ depends on the gain: $h(s) = \exp(s)$ when $\gamma = 1$ in $a_n = a_0/n^\gamma$, and $h(s) = 1$ when $\gamma < 1$.

Read (97.15) as a cost that penalizes departures of the *realized drift* $\frac{d}{ds}\phi$ from the *mean drift* $b(\phi)$, weighting each direction by the inverse of the local noise covariance $\sigma R \sigma^\top$.

The least-action escape therefore threads the beliefs through regions where the mean dynamics are weak and the noise is informative — which, in our model, is the direction of the *induction hypothesis*.

Note

This quadratic action functional is precisely the object minimized in [Cho *et al.*, 2002], the published treatment of the model of this lecture.

They solve the control problem (97.13) for the Nash self-confirming equilibrium and show, analytically, that the least-action escape drives the sum of weights on inflation toward the value that activates the induction hypothesis — that is, toward the Ramsey outcome.

[Sargent and Williams, 2005] study how the government's prior (equivalently, the covariance structure of the gain algorithm, our P_0 and forgetting factor) reshapes the escape, and [Kasa, 2004] applies the same large-deviation machinery to recurrent currency crises.

97.2.8 From computation to adaptation

The preceding recursions were introduced as *algorithms* to approximate a self-confirming equilibrium.

The same mathematics tells us what happens when we instead *modify* our self-confirming-equilibrium models to incorporate real-time adaptation — simply by reading ϕ_n as the government's time-n beliefs rather than as the n-th iterate of a solver.

Two facts organize everything below:

1. gain sequences that implement least squares (decreasing like $1/t$) make the mean dynamics pull the economy *toward* self-confirming equilibria; while
2. gain sequences that fall off more slowly — in the limit, constant gains that discount the past — *arrest* that pull and increase the frequency with which the escape dynamics influence outcomes.

Note

A brief intellectual history.

[Lucas and Prescott, 1971] dismissed iterating on the moment conditions (97.4) as a computational strategy, but [Townsend, 1983] used it.

[Woodford, 1990] and [Marcet and Sargent, 1989] used the mean dynamics (97.8) to establish conditions for the convergence of least squares learning to rational expectations in models with self-reference, both requiring continuity of $b(\phi)$.

In-Koo Cho studied problems with *discontinuous* $b(\phi)$ inherited from discontinuous decision rules (trigger strategies in credibility and search problems); to make least squares learning approach rational expectations he used gains satisfying $\frac{1}{\log n} < a_n < \frac{1}{\sqrt{n}}$, which yield a *diffusion* approximation to (97.6) that promotes enough experimentation to discover an equilibrium.

[Kandori *et al.*, 1993] use related mathematics to select long-run equilibria in games via mutation, and Roger Myerson applied an escape-route calculation to a voting problem.

The modern synthesis of these learning methods is [Evans and Honkapohja, 2001].

97.3 The adaptive model

We now build the classical adaptive model.

97.3.1 Government beliefs and behavior

The government believes in a distributed-lag Phillips curve

$$U_t = \gamma^\top X_{C,t} + \varepsilon_{C,t}, \qquad X_{C,t} = \begin{bmatrix} y_t & U_{t-1} & U_{t-2} & y_{t-1} & y_{t-2} & 1 \end{bmatrix}^\top. \tag{97.16}$$

Arriving at time t with an estimate γ_{t-1}, it sets the systematic part of inflation by solving the Phelps problem *as if* γ_{t-1} will govern the Phillips curve forever:

$$y_t = h(\gamma_{t-1}) X_{t-1} + v_{2t}, \qquad X_{t-1} = \begin{bmatrix} U_{t-1} & U_{t-2} & y_{t-1} & y_{t-2} & 1 \end{bmatrix}^\top. \tag{97.17}$$

It then updates its beliefs by **recursive least squares** (RLS):

$$\begin{aligned} \gamma_t &= \gamma_{t-1} + g_t R_{XC,t}^{-1} X_{C,t} \left(U_t - \gamma_{t-1}^\top X_{C,t} \right), \\ R_{XC,t} &= R_{XC,t-1} + g_t \left(X_{C,t} X_{C,t}^\top - R_{XC,t-1} \right), \end{aligned} \tag{97.18}$$

where $\{g_t\}$ is the gain sequence.

Least squares sets $g_t = 1/t$; a constant-gain algorithm sets $g_t = g_0 > 0$ and discounts past observations, which is sensible if the government suspects the Phillips curve wanders over time.

The public is assumed to know the government's rule, so its inflation forecast is $x_t = h(\gamma_{t-1}) X_{t-1}$, the systematic part of (97.17).

Unemployment is generated by the actual Phillips curve of *Self-Confirming Equilibria* with $\rho_1 = \rho_2 = 0$:

$$U_t = U^* - \theta (y_t - x_t) + v_{1t} = U^* - \theta v_{2t} + v_{1t}.$$

97.3.2 The Phelps problem with lags

Given a belief γ, the decision rule $h(\gamma)$ solves an LQ control problem.

Write the believed Phillips curve as $U_t = \gamma_0 y_t + c^\top s_t$, where γ_0 is the coefficient on current inflation and c collects the coefficients on the state $s_t = X_{t-1}$.

The government minimizes $\mathbb{E} \sum_t \delta^t (U_t^2 + y_t^2)$, so the per-period loss is $s_t^\top (c c^\top) s_t + (\gamma_0^2 + 1) y_t^2 + 2\gamma_0 \, y_t \, c^\top s_t$, and the state evolves as $s_{t+1} = A s_t + B y_t$ with

$$s_{t+1} = \begin{bmatrix} U_t \\ U_{t-1} \\ y_t \\ y_{t-1} \\ 1 \end{bmatrix}, \qquad U_t = c^\top s_t + \gamma_0 y_t.$$

We solve the discounted LQ problem with `scipy`'s discrete algebraic Riccati equation.

```
class AdaptivePhillips:
    """
    Classical adaptive Phillips curve model: the government re-estimates a
    distributed-lag Phillips curve by recursive least squares and each period
```

(continues on next page)

(continued from previous page)

```
    acts on the first-period recommendation of the Phelps problem.
    """

    def __init__(self, θ=1.0, U_star=5.0, σ1=0.3, σ2=0.3, δ=0.98):
        self.θ, self.U_star, self.σ1, self.σ2, self.δ = θ, U_star, σ1, σ2, δ

        # classical self-confirming belief: U = -θ y + (θ²+1)U*
        self.y_sce = np.array([-θ, 0.0, 0.0, 0.0, 0.0, (θ**2 + 1) * U_star])

        # self-confirming moment matrix M = E[X_C X_C'] and residual variance
        self.M = self._sce_moments()
        self.σC2 = σ1**2                        # var(U | X_C) at the SCE

    def _sce_moments(self):
        "E[X_C X_C'] at the serially-uncorrelated classical SCE."
        θ, σ1, σ2 = self.θ, self.σ1, self.σ2
        μU, μy = self.U_star, self.θ * self.U_star
        Σ = {('U', 'U'): θ**2 * σ2**2 + σ1**2, ('y', 'y'): σ2**2,
             ('U', 'y'): -θ * σ2**2, ('y', 'U'): -θ * σ2**2}
        # regressors: (time, type) for [y_t, U_{t-1}, U_{t-2}, y_{t-1}, y_{t-2}, 1]
        regs = [(0, 'y'), (1, 'U'), (2, 'U'), (1, 'y'), (2, 'y'), (None, 'c')]
        mean = {'U': μU, 'y': μy, 'c': 1.0}
        M = np.zeros((6, 6))
        for i, (ti, tyi) in enumerate(regs):
            for j, (tj, tyj) in enumerate(regs):
                if tyi == 'c' or tyj == 'c' or ti != tj:
                    M[i, j] = mean[tyi] * mean[tyj]
                else:
                    M[i, j] = Σ[(tyi, tyj)] + mean[tyi] * mean[tyj]
        return M

    def phelps_h(self, y):
        "Government decision rule ŷ_t = h(y)·X_{t-1} for belief y."
        δ, y0, c = self.δ, y[0], y[1:]
        R = np.outer(c, c)
        Q = np.array([[y0**2 + 1.0]])
        N = (y0 * c).reshape(1, -1)
        A = np.zeros((5, 5)); B = np.zeros((5, 1))
        A[0, :] = c; B[0, 0] = y0           # U_t
        A[1, 0] = 1.0                        # U_{t-1}
        B[2, 0] = 1.0                        # y_t
        A[3, 2] = 1.0                        # y_{t-1}
        A[4, 4] = 1.0                        # constant
        sb = np.sqrt(δ)
        Ad, Bd = sb * A, sb * B
        P = solve_discrete_are(Ad, Bd, R, Q, s=sb * N.T)
        F = np.linalg.solve(Q + Bd.T @ P @ Bd, Bd.T @ P @ Ad + N)
        return -F.ravel()
```

We use the Kalman-filter implementation of RLS from Appendix A of [Sargent, 1999].

A forgetting factor $\lambda \in (0, 1]$ maps to the gain: $\lambda = 1$ gives least squares ($g_t \to 1/t$), while $\lambda < 1$ gives a constant gain $g_0 = 1 - \lambda$.

The prior is initialized as if the government had already seen T periods of self-confirming-equilibrium data, through $P_0 = (\sigma_C^2 / T)\, M^{-1}$; larger T means a tighter prior.

```
def simulate(model, λ, T_prior, n=1000, seed=0):
    "Simulate the adaptive system. λ=1 is least squares; λ<1 is constant gain."
    rng = np.random.default_rng(seed)
    θ, U_star, σ1, σ2 = model.θ, model.U_star, model.σ1, model.σ2

    γ = model.γ_sce.copy()
    P = (model.σC2 / T_prior) * np.linalg.inv(model.M)
    R2 = model.σC2
    g0 = 1 - λ

    U1 = U2 = U_star
    y1 = y2 = θ * U_star
    y_path, U_path, sumweights, constant = (np.empty(n) for _ in range(4))

    for t in range(n):
        h = model.phelps_h(γ)
        X_lag = np.array([U1, U2, y1, y2, 1.0])
        yhat = h @ X_lag

        v2, v1 = σ2 * rng.standard_normal(), σ1 * rng.standard_normal()
        y = yhat + v2
        U = U_star - θ * (y - yhat) + v1

        ψ = np.array([y, U1, U2, y1, y2, 1.0])           # X_C,t
        denom = R2 + ψ @ P @ ψ
        gain = P @ ψ / denom
        γ = γ + gain * (U - γ @ ψ)
        R1 = (g0 / (1 - g0)) * P if λ < 1 else 0.0        # constant vs decreasing
        P = P - np.outer(P @ ψ, ψ @ P) / denom + R1

        y_path[t], U_path[t] = y, U
        sumweights[t] = γ[0] + γ[3] + γ[4]               # weights on current+lagged y
        constant[t] = γ[5]
        U2, U1 = U1, U
        y2, y1 = y1, y

    return dict(y=y_path, U=U_path, sumweights=sumweights, constant=constant)
```

```
model = AdaptivePhillips()
h_sce = model.phelps_h(model.γ_sce)
print("decision rule at the self-confirming belief:")
print(f"  h(γ_sce) = {np.round(h_sce, 3)}  (a constant rule of "
      f"{h_sce[-1]:.1f} = Nash inflation)")
```

```
decision rule at the self-confirming belief:
  h(γ_sce) = [-0. -0. -0. -0.  5.]  (a constant rule of 5.0 = Nash inflation)
```

At the self-confirming belief the Phelps rule is a constant equal to the Nash inflation rate, and the beliefs reproduce themselves — the adaptive system's rest point is the self-confirming equilibrium of *Self-Confirming Equilibria*.

97.4 Least squares learning converges

We follow [Sargent, 1999] in setting the true data-generating parameters to the classical example at the end of *Self-Confirming Equilibria*: $U^* = 5$, $\theta = 1$, $\sigma_1 = \sigma_2 = 0.3$, $\rho_1 = \rho_2 = 0$, $\delta = 0.98$.

The classical self-confirming equilibrium has serially uncorrelated (U, y) fluctuating around means $(5, 5)$.

First, least squares (a decreasing gain).

```
ls = simulate(model, λ=1.0, T_prior=5000, n=1000, seed=1)

fig, ax = plt.subplots(figsize=(9, 4.5))
ax.plot(ls['y'], lw=0.8)
ax.axhline(5, color='k', ls='--', lw=1, label='self-confirming (Nash)')
ax.axhline(0, color='C2', ls=':', lw=1, label='Ramsey')
ax.set_xlabel('$t$')
ax.set_ylabel('inflation $y_t$')
ax.legend()
plt.show()
```

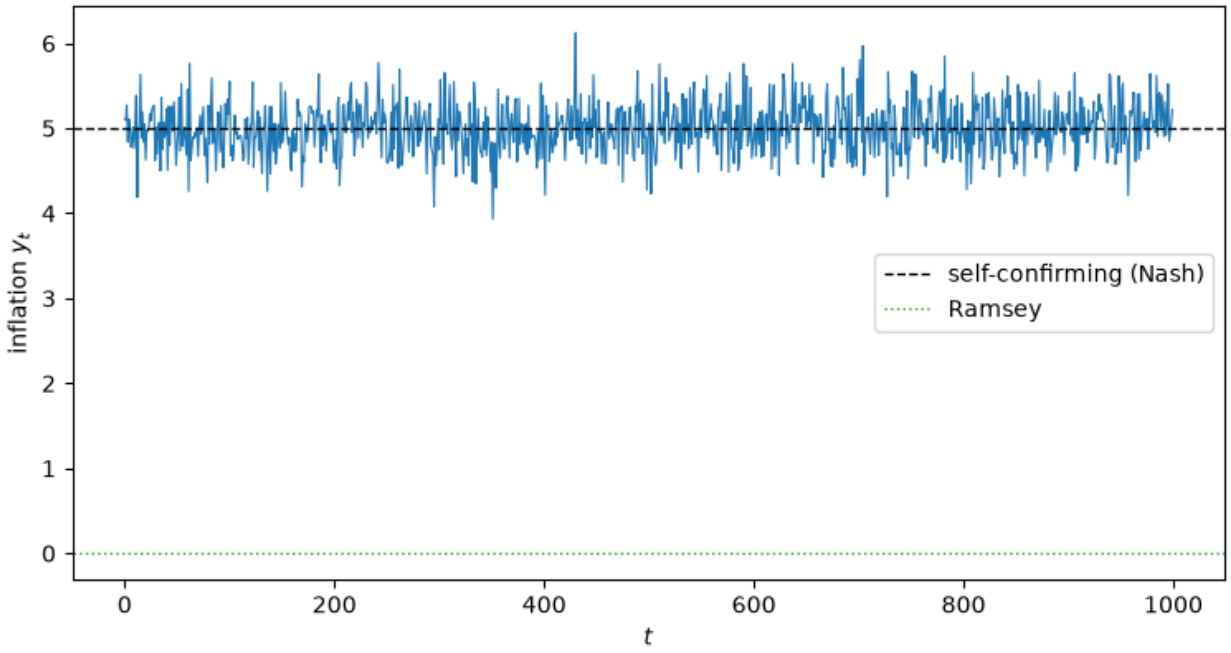


Fig. 97.1: Classical adaptive model under least squares: inflation hugs the self-confirming value

Under least squares the mean dynamics dominate: inflation hugs the self-confirming value of 5, and the simulation looks like a draw from the self-confirming equilibrium itself.

We get nothing new — the government is stuck near the Nash outcome.

97.5 Constant gain and escape dynamics

Now give the government a *constant* gain, $\lambda = 0.975$, so it discounts past data.

```
cg = simulate(model, λ=0.975, T_prior=300, n=1000, seed=1)

fig, ax = plt.subplots(figsize=(9, 4.5))
ax.plot(cg['y'], lw=0.8)
ax.axhline(5, color='k', ls='--', lw=1, label='self-confirming (Nash)')
ax.axhline(0, color='C2', ls=':', lw=1, label='Ramsey')
ax.set_xlabel('$t$')
ax.set_ylabel('inflation $y_t$')
ax.legend()
plt.show()
```

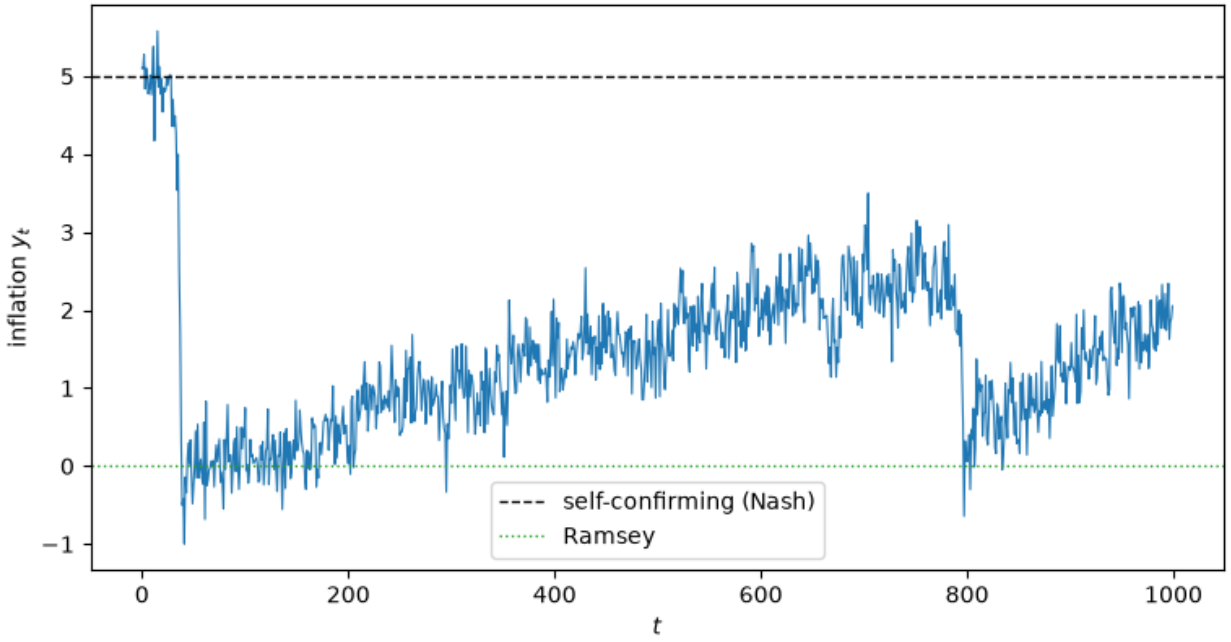


Fig. 97.2: Classical adaptive model under a constant gain: recurrent escapes toward Ramsey

Inflation starts near the self-confirming value of 5, then drops almost to zero and stays there for a long time, before slowly heading back toward 5 only to be propelled toward zero again.

The mean dynamics that pull the system toward the self-confirming equilibrium are opposed by a recurrent force that sends inflation close to the Ramsey outcome.

Crucially, no large shocks trigger these stabilizations: they are an *endogenous* feature of the constant-gain learning dynamics.

```
print(f"constant-gain inflation: mean {cg['y'].mean():.2f}, "
      f"fraction of periods near Ramsey (y<2): {(cg['y'] < 2).mean():.0%}")
```

```
constant-gain inflation: mean 1.41, fraction of periods near Ramsey (y<2): 77%
```

97.6 The escape route and the induction hypothesis

Why does the system escape toward Ramsey rather than in some other direction?

The answer is the **induction hypothesis** of *Adaptive Expectations and the Phelps Problem*: when the sum of the weights on current and lagged inflation in the estimated Phillips curve approaches zero, the Phelps problem advises the government to *reduce* inflation.

Let's plot inflation together with that sum of weights.

```
fig, axes = plt.subplots(2, 1, figsize=(9, 7), sharex=True)

axes[0].plot(cg['y'], lw=0.8)
axes[0].axhline(0, color='C2', ls=':', lw=1)
axes[0].set_ylabel('inflation $y_t$')

axes[1].plot(cg['sumweights'], lw=0.8, color='C1')
axes[1].axhline(-1, color='k', ls='--', lw=1, label='self-confirming value')
axes[1].axhline(0, color='C3', ls=':', lw=1, label='induction hypothesis')
axes[1].set_xlabel('$t$')
axes[1].set_ylabel('sum of weights on $y$')
axes[1].legend()

fig.suptitle('Escape route: stabilizations coincide with the sum of '
             'weights rising toward zero')
plt.tight_layout()
plt.show()
```

Every stabilization coincides with the sum of weights jumping from its self-confirming value of -1 toward zero.

When it reaches zero, the induction hypothesis is (temporarily) satisfied, the Phelps problem calls for near-Ramsey inflation, and the resulting data briefly *reinforce* the induction hypothesis — a situation that is not self-confirming in the technical sense but is nonetheless self-reinforcing.

We can see the escape route directly by plotting the joint path of the constant and the sum of weights in the estimated Phillips curve.

```
fig, ax = plt.subplots(figsize=(8, 6))
sc = ax.scatter(cg['constant'], cg['sumweights'], c=np.arange(len(cg['y'])),
                cmap='viridis', s=6)
ax.axhline(0, color='C3', ls=':', lw=1.5, label='induction hypothesis')
ax.axhline(-1, color='k', ls='--', lw=1, label='self-confirming value')
ax.set_xlabel('constant in estimated Phillips curve')
ax.set_ylabel('sum of weights on $y$')
ax.legend()
plt.colorbar(sc, label='time $t$')
plt.show()
```

The beliefs spend most of their time near the self-confirming value (sum of weights ≈ -1) but repeatedly shoot up toward the induction line (sum of weights $= 0$) — the escape route along which the government learns a Solow-Tobin version of the natural-rate hypothesis and stabilizes inflation.

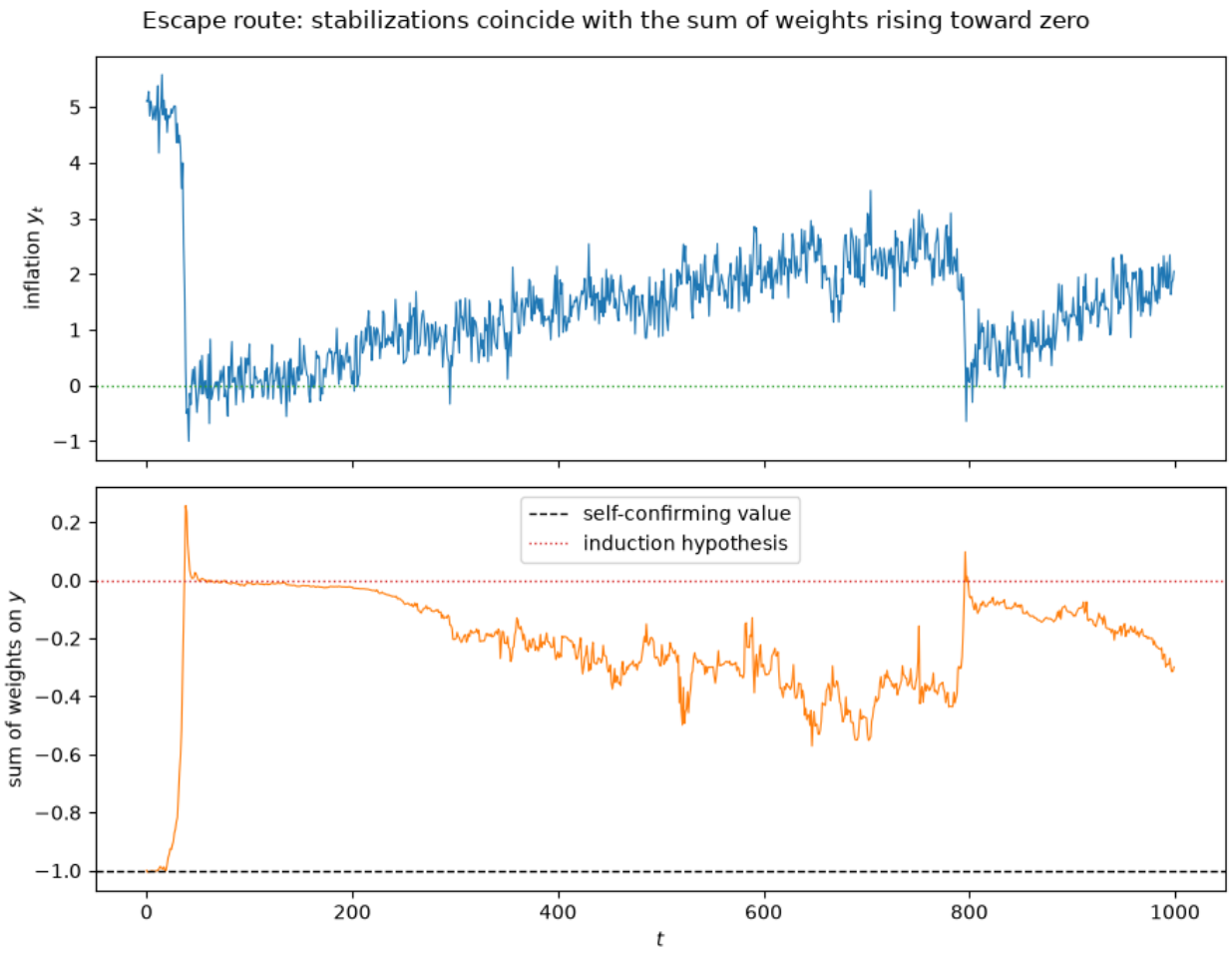


Fig. 97.3: Stabilizations coincide with the sum of weights on inflation rising toward zero

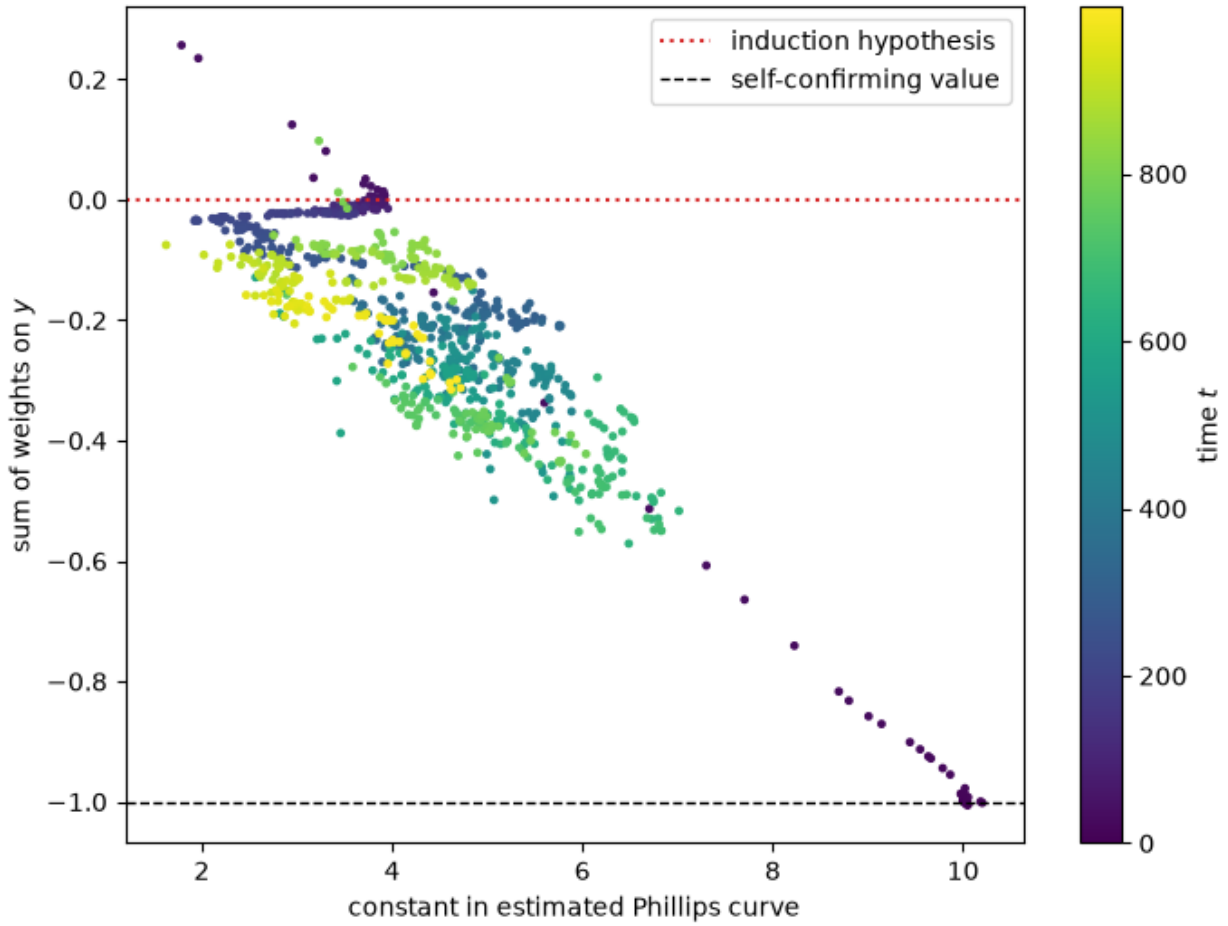


Fig. 97.4: The escape route in belief space, coloured by time

97.7 Relation to equilibria under forecast misspecification

The near-Ramsey episodes are reminiscent of the equilibria with optimal misspecified forecasts of *Optimal Misspecified Beliefs* and *Self-Confirming Equilibria*.

There, a forecasting model *without a constant but with a unit root* could closely approximate a true model that *includes* a constant.

Here an approximation with a similar flavor operates during the near-Ramsey episodes: the government's estimated Phillips curve, by driving the sum of weights toward zero, uses the induction hypothesis to approximate a constant — except that the approximated model is not fixed but changes as the government's own beliefs feed back through the Phelps problem.

97.8 Role of the discount factor

The recurrent stabilizations toward Ramsey depend on the discount factor δ being near one.

Lowering δ raises the inflation rate observed during the low-inflation episodes, consistent with the workings of the Phelps problem under the induction hypothesis.

```
fig, ax = plt.subplots(figsize=(9, 4.5))
for δ in [0.90, 0.95, 0.98]:
    m = AdaptivePhillips(δ=δ)
    sim = simulate(m, λ=0.975, T_prior=300, n=1000, seed=1)
    ax.plot(sim['y'], lw=0.7, label=rf'$\delta = {δ}$')
ax.axhline(0, color='k', lw=0.5)
ax.set_xlabel('$t$')
ax.set_ylabel('inflation $y_t$')
ax.legend()
plt.show()
```

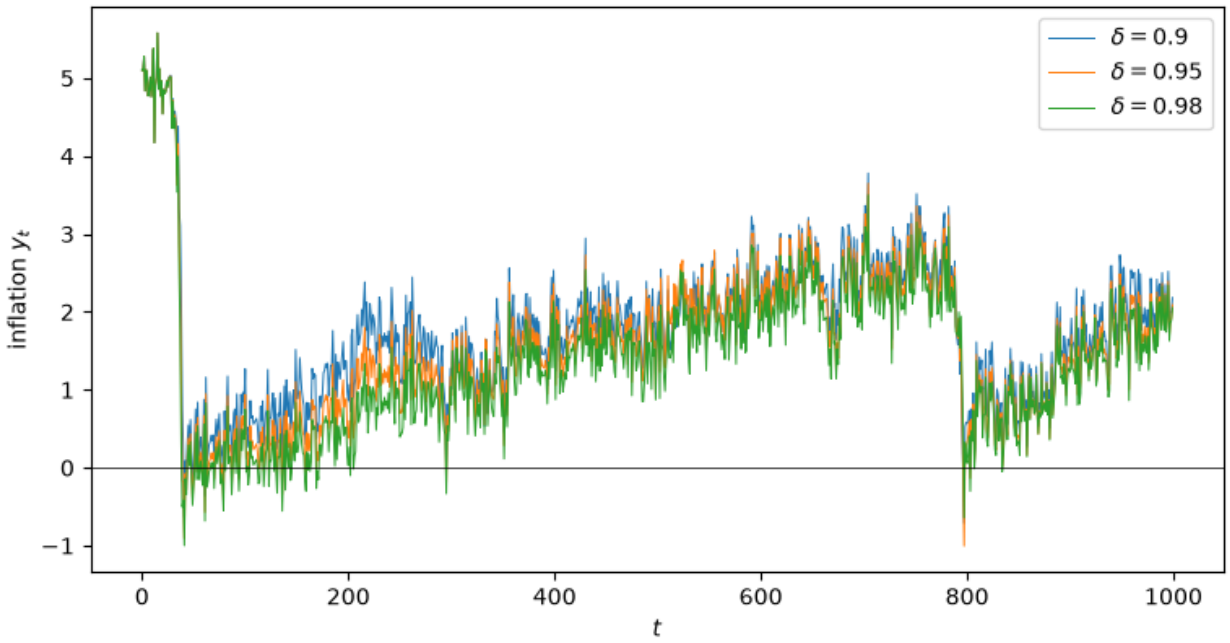


Fig. 97.5: Escapes toward Ramsey deepen as the government becomes more patient

97.9 Anticipated utility

The adaptive model is an example of what David Kreps calls an **anticipated utility** model [Kreps, 1998].

The government adapts a temporarily misspecified model — a Phillips curve with fixed coefficients — to incorporate the most recent observations, and reoptimizes along the way.

In forming decisions at t, it acts as if its current estimate γ_{t-1} will govern the Phillips curve forever, using the same policy functional $h(\cdot)$ that would be optimal if γ were truly time-invariant.

This is a small departure from rational expectations: calendar time enters only through the drifting beliefs γ_t.

Unlike Bayesian or robust decision makers, an anticipated-utility government ignores its period-by-period model misspecification — it does not entertain the possibility that its coefficients will drift, even as they do.

Yet, as the simulations show, this modest departure from rationality is enough to generate a rich account of the rise and fall of U.S. inflation, and to supply underpinnings for the vindication of econometric policy evaluation.

97.10 Conclusions

For long stretches, an adaptive government learns to generate *better than Nash* outcomes.

These results come from the recurrent dynamics of adaptation: the mean dynamics that under least squares pull the system toward a self-confirming equilibrium continue to operate, but under a constant gain the noise lets the system recurrently escape toward the Ramsey outcome.

Starting from a self-confirming equilibrium, the adaptive algorithm gradually makes the government put enough weight on the induction hypothesis that chance observations eventually promote a stabilization.

Adaptation makes the government's beliefs a hidden state that imparts serial correlation into inflation and unemployment — so that an outside forecaster would do well to use a random-coefficients model, or to make the constant adjustments that Lucas noted in his Critique [Lucas, 1976].

In this sense the adaptive models contain the underpinnings for vindicating econometric policy evaluation — the second of the two stories of *The Rise and Fall of U.S. Inflation*.

That last observation is a testable one, and it is worth recording what the model predicts before anyone looks.

If the government's beliefs really are a hidden drifting state, then a reduced-form description of post-war inflation and unemployment ought to show *drifting coefficients*, not merely drifting shock variances.

Drifts and Volatilities fits exactly such a model to the data and reports the verdict — including one prediction of the escape mechanism that the data decline to confirm.

97.11 Exercises

Exercise 97.11.1

Everything above hinges on the constant gain, so it is worth seeing how much.

Simulate the classical adaptive model with $\lambda \in \{0.99, 0.975, 0.95\}$ and compare how often inflation escapes toward Ramsey.

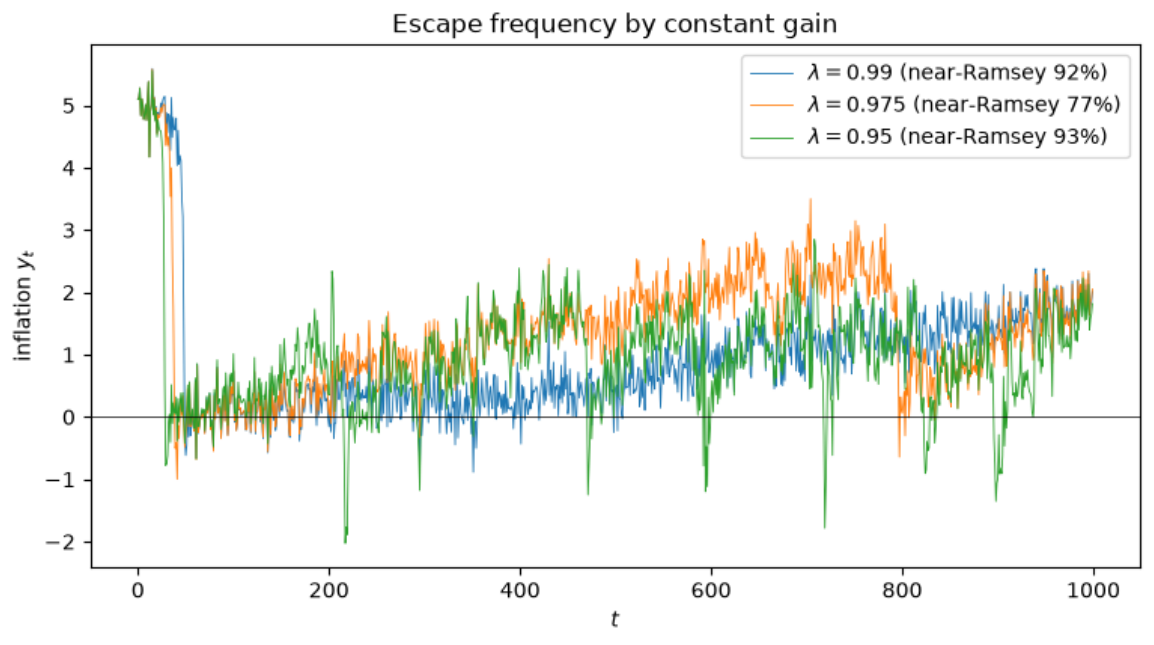
How does a larger gain (smaller λ, faster discounting of the past) affect the frequency of escapes?

Note

A more ambitious version of this exercise is to build the **Keynesian** adaptive model, in which the government fits the Phillips curve in the reverse direction, regressing inflation on unemployment.
The regressors are $X_{K,t} = \begin{bmatrix} U_t & U_{t-1} & U_{t-2} & y_{t-1} & y_{t-2} & 1 \end{bmatrix}^\top$; the government estimates β in $y_t = \beta^\top X_{K,t} + \varepsilon_{K,t}$, then inverts to γ using the inversion formulas $\gamma_1 = \beta_1^{-1}$, $\gamma_{-1} = -\beta_{-1}/\beta_1$ of *Self-Confirming Equilibria* before solving the Phelps problem.
[Sargent, 1999] reports that this variant escapes less readily than the classical one.

Solution

```
fig, ax = plt.subplots(figsize=(9, 4.5))
for λ in [0.99, 0.975, 0.95]:
    sim = simulate(model, λ=λ, T_prior=300, n=1000, seed=1)
    frac = (sim['y'] < 2).mean()
    ax.plot(sim['y'], lw=0.6,
            label=rf'$\lambda = {λ}$ (near-Ramsey {frac:.0%})')
ax.axhline(0, color='k', lw=0.5)
ax.set_xlabel('$t$')
ax.set_ylabel('inflation $y_t$')
ax.set_title('Escape frequency by constant gain')
ax.legend()
plt.show()
```



A larger constant gain (smaller λ) discounts the past more heavily, arresting convergence to the self-confirming equilibrium more forcefully and producing more frequent — though also noisier — escapes toward the Ramsey outcome.

Exercise 97.11.2

The contrast between Figures 8.1 and 8.2 hinges on the gain, but the least squares result also depends on the tightness of the prior.

Simulate the least squares system ($\lambda = 1$) for prior tightness $T \in \{500, 2000, 5000\}$, and report the mean inflation rate across several seeds.

Explain why a looser prior (smaller T) makes even the least squares system prone to escapes.

Solution

```
for T in [500, 2000, 5000]:
    means = [simulate(model, λ=1.0, T_prior=T, n=1000, seed=s)['y'].mean()
             for s in range(6)]
    print(f"T = {T:>4}:  mean inflation across seeds = "
          f"{np.round(means, 1)}")
```

```
T =  500:  mean inflation across seeds = [0.9 1.9 5.  2.3 1.3 0.6]
T = 2000:  mean inflation across seeds = [4.7 5.  5.  5.  3.7 4.9]
T = 5000:  mean inflation across seeds = [5. 5. 5. 5. 5. 5.]
```

With a looser prior the effective gain $1/(T + t)$ starts larger, so early updates are big enough to kick the beliefs off the self-confirming equilibrium.

Because that equilibrium is only marginally stable — the mean dynamics have an eigenvalue at the boundary of the region of fast convergence — the system can then drift toward the induction hypothesis and get stuck near Ramsey, mimicking the constant-gain escapes.

A tighter prior keeps the gain small throughout, so least squares reliably hugs the self-confirming equilibrium.

CHAPTER

NINETYEIGHT

ESCAPING NASH INFLATION

Contents

- *Escaping Nash Inflation*
  - *Overview*
  - *The model*
  - *The self-confirming equilibrium*
  - *Adaptation and the mean dynamics*
  - *Escape dynamics as a control problem*
  - *The dominant escape path*
  - *The race of four ODEs*
  - *Mean dynamics reinforce the escape*
  - *The experimentation trap*
  - *Escape frequency and model richness*
  - *Escaping volatile inflation*
  - *Exercises*

98.1 Overview

> If an unlikely event occurs, it is very likely to occur in the most likely way.
>
> – Michael Harrison

This lecture is the analytical completion of *Adaptive Learning and Escape Dynamics*.

It follows [Cho *et al.*, 2002] (CWS), which turns the *simulated* escape dynamics of chapter 8 of [Sargent, 1999] into a *deterministic characterization*.

In *Adaptive Learning and Escape Dynamics* we watched a constant-gain government recurrently escape the Nash self-confirming equilibrium and spend long spells near the Ramsey outcome — but we described those escapes only informally and by simulation.

CWS show that the escapes are governed by their own ordinary differential equation, obtained from the **theory of large deviations**.

The picture that emerges has two deterministic pieces:

- the **mean dynamics**, an ODE that pulls the government's beliefs *toward* the self-confirming equilibrium; and
- the **escape dynamics**, a second ODE that — driven by a "most likely unlikely" sequence of shocks — pushes beliefs *away* from it, toward the beliefs that support the Ramsey outcome.

The remarkable finding is that the escape has a *dominant path*: conditional on escaping, the government's beliefs follow a nearly deterministic route along which it temporarily learns a version of the natural-rate hypothesis and cuts inflation.

This lecture is a technical extension of *Adaptive Learning and Escape Dynamics*; the next lecture, *Priors, Escapes, and Learning Cycles*, builds on it by asking how the government's *prior* about parameter drift reshapes both the mean dynamics and these escape dynamics.

We work with the analytically tractable **static** model.

Let's start with our imports:

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.integrate import solve_ivp
```

98.2 The model

The true economy is the natural-rate model of [Kydland and Prescott, 1977],

$$U_n = u - \theta(\pi_n - \hat{x}_n) + \sigma_1 W_{1n}, \qquad \pi_n = x_n + \sigma_2 W_{2n}, \qquad \hat{x}_n = x_n, \tag{98.1}$$

with $\theta, u > 0$ and $W_n = (W_{1n}, W_{2n})^\top$ IID standard Gaussian.

The government does not know (98.1).

In the static model it fits a non-expectational Phillips curve by regressing unemployment on inflation and a constant,

$$U_n = \gamma_1 \pi_n + \gamma_{-1} + \eta_n, \tag{98.2}$$

with beliefs $\gamma = (\gamma_1, \gamma_{-1})$ (slope and intercept), and it treats η_n as exogenous.

> **Note**
>
> The slope comes first here, and the regressors are $\Phi = (\pi, 1)$, matching [Cho *et al.*, 2002] and the γ_1, γ_{-1} convention of *Self-Confirming Equilibria*.
>
> *Priors, Escapes, and Learning Cycles* studies the same model with the intercept first; see the warning there for the translation.

Believing (98.2), the government solves the *Phelps problem*, whose static best response sets inflation to the constant

$$x(\gamma) = -\frac{\gamma_{-1}\,\gamma_1}{1 + \gamma_1^2}. \tag{98.3}$$

Three beliefs are worth naming, following CWS:

- **Belief 1 (Nash):** $\gamma_1 = -\theta$ with an intercept that makes $x = \theta u$ — the time-consistent outcome of [Kydland and Prescott, 1977].
- **Belief 2 (Ramsey):** $\gamma_1 = 0$, so the government perceives no tradeoff and sets $x = 0$.

- **Belief 3 (induction):** coefficients on inflation summing to zero, which for a patient government also sends inflation to 0.

```
class EscapeModel:
    "CWS 2002 static model. γ = (γ₁ slope, γ₋₁ intercept), regressors Φ = (π, 1)."

    def __init__(self, θ=1.0, u=5.0, σ1=0.3, σ2=0.3):
        self.θ, self.u, self.σ1, self.σ2 = θ, u, σ1, σ2

    def x(self, γ):
        γ1, γm1 = γ
        return -γm1 * γ1 / (1 + γ1**2)

    def M(self, γ):
        "E[ΦΦ'] with Φ = (π, 1)."
        x = self.x(γ)
        return np.array([[x**2 + self.σ2**2, x], [x, 1.0]])

    def g_bar(self, γ):
        "Mean-dynamics forcing E[Φ(U - Φ'γ)] = M(T(γ) - γ)."
        x = self.x(γ)
        E_ΦU = np.array([x * self.u - self.θ * self.σ2**2, self.u])
        return E_ΦU - self.M(γ) @ γ
```

98.3 The self-confirming equilibrium

A self-confirming equilibrium is a belief that reproduces itself: the population regression coefficients of (98.2), computed on the data the government generates by acting on γ, equal γ.

Writing $T(\gamma)$ for those population coefficients, CWS show

$$\bar{g}(\gamma) \equiv \mathbb{E}\left[\Phi(U - \Phi^{\top}\gamma)\right] = \bar{M}\left(T(\gamma) - \gamma\right),$$

so a self-confirming equilibrium solves $\bar{g}(\gamma) = 0$.

For the static model the equilibrium is the intersection of the line $\gamma_1 = -\theta$ with the parabola $\gamma_{-1} = u(1 + \gamma_1^2)$ — a unique point that supports the Nash outcome of *The Credibility Problem*.

It is the same self-confirming equilibrium constructed in *Self-Confirming Equilibria*, written here in the static (constant-plus-slope) special case that *Priors, Escapes, and Learning Cycles* also uses.

```
model = EscapeModel(θ=1.0, u=5.0, σ1=0.3, σ2=0.3)
γ_sce = np.array([-model.θ, model.u * (1 + model.θ**2)])

print(f"self-confirming beliefs γ = {γ_sce}  (slope -θ, intercept u(1+θ²))")
print(f"self-confirming inflation x = {model.x(γ_sce):.2f}  (= Nash = θu)")
print(f"check g_bar = {model.g_bar(γ_sce)}")
```

```
self-confirming beliefs γ = [-1. 10.]  (slope -θ, intercept u(1+θ²))
self-confirming inflation x = 5.00  (= Nash = θu)
check g_bar = [0. 0.]
```

```
γ1_grid = np.linspace(-2, 1, 200)
fig, ax = plt.subplots(figsize=(7, 6))
ax.plot(γ1_grid, model.u * (1 + γ1_grid**2), label=r'$\gamma_{-1} = u(1+\gamma_1^2)$',
```

(continues on next page)

(continued from previous page)

```
    ↪ lw=2)
ax.axvline(-model.θ, color='C1', ls='--', label=r'$\gamma_1 = -\theta$')
ax.plot(γ_sce[0], γ_sce[1], 'ko', ms=8)
ax.annotate('SCE (Nash)', γ_sce, (γ_sce[0] + 0.1, γ_sce[1] + 1))
ax.set_xlabel(r'slope $\gamma_1$')
ax.set_ylabel(r'intercept $\gamma_{-1}$')
ax.set_ylim(0, 25)
ax.legend()
ax.set_title('The unique self-confirming equilibrium')
plt.show()
```

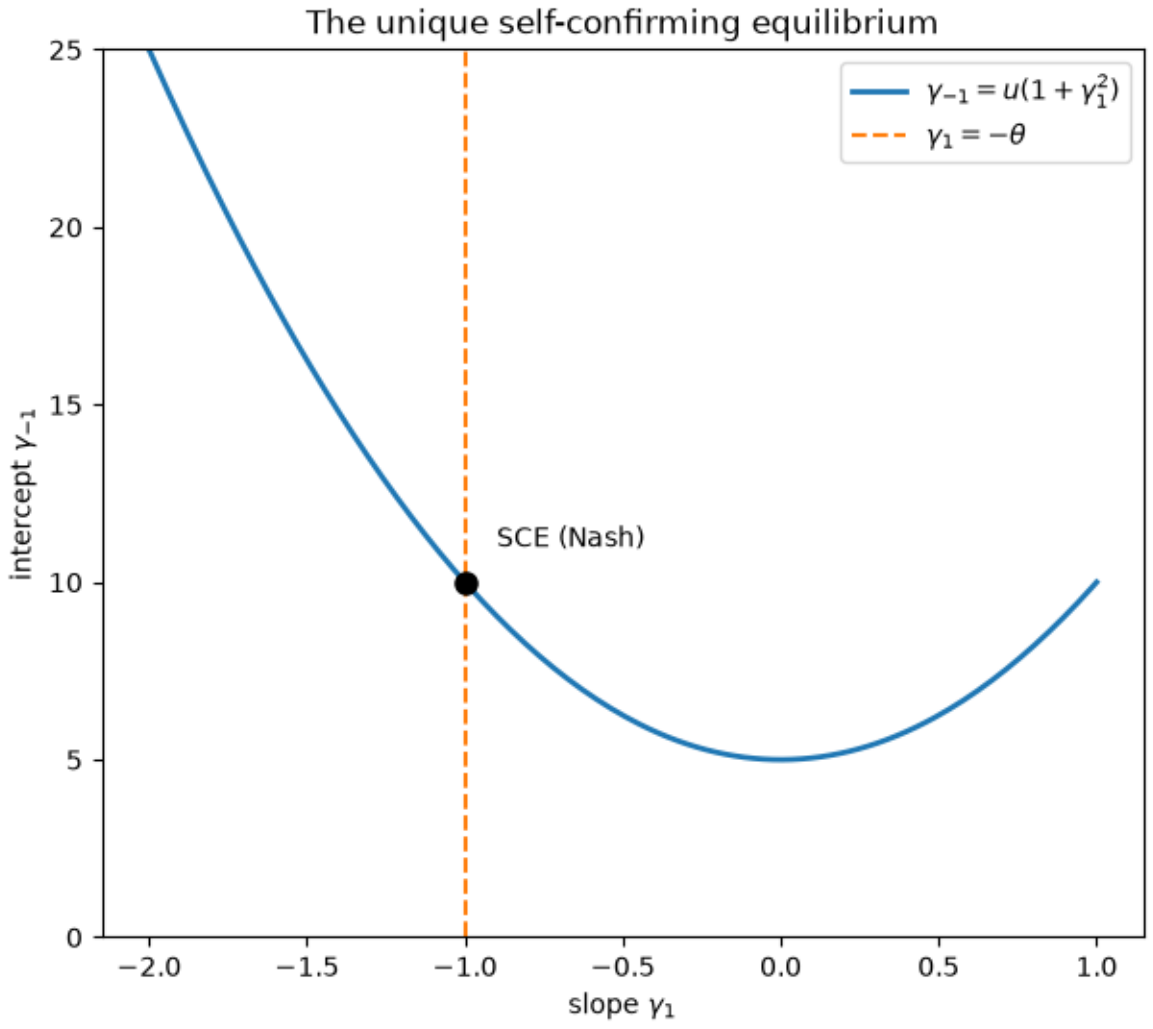


Fig. 98.1: The unique self-confirming equilibrium in belief space

98.4 Adaptation and the mean dynamics

We make the government adaptive: each period it updates γ by constant-gain recursive least squares and acts on its current estimate — an *anticipated-utility* model in the sense of [Kreps, 1998].

The literature on least squares learning ([Marcet and Sargent, 1989], [Woodford, 1990], [Evans and Honkapohja, 2001]) shows that, as the gain $\varepsilon \to 0$, the beliefs are approximated by the **mean-dynamics** ODE

$$\dot{\gamma} = R^{-1}\bar{g}(\gamma), \qquad \dot{R} = \bar{M}(\gamma) - R. \tag{98.4}$$

A rest point of (98.4) is a self-confirming equilibrium, and CWS show this ODE is *globally stable* about it.

So under the mean dynamics alone, the adaptive government is drawn to Nash inflation.

```
def mean_ode(t, z, model):
    y, R = z[:2], z[2:].reshape(2, 2)
    return np.concatenate([np.linalg.inv(R) @ model.g_bar(y),
                           (model.M(y) - R).ravel()])

z0 = np.concatenate([y_sce + np.array([0.4, -3.0]), model.M(y_sce).ravel()])
sol = solve_ivp(lambda t, z: mean_ode(t, z, model), [0, 60], z0,
                max_step=0.1, rtol=1e-9, atol=1e-11)

fig, ax = plt.subplots(figsize=(8, 4.5))
ax.plot(sol.t, -sol.y[1] * sol.y[0] / (1 + sol.y[0]**2))
ax.axhline(model.θ * model.u, color='k', ls='--', lw=1, label='Nash')
ax.set_xlabel('time')
ax.set_ylabel('inflation $x$')
ax.set_title('Mean dynamics: from a perturbed start, beliefs return to Nash')
ax.legend()
plt.show()
```

The mean dynamics cannot, on their own, explain the recurrent visits to low inflation in the simulations of *Adaptive Learning and Escape Dynamics*.

For that we need a second force.

98.5 Escape dynamics as a control problem

Although the impact of the noise vanishes as $\varepsilon \to 0$, for a fixed positive gain *rare* sequences of shocks can push beliefs a long way from the self-confirming equilibrium.

The theory of large deviations characterizes the *most likely* such rare event.

For a candidate belief path $\gamma(\cdot)$, one defines a log-moment-generating (H-) functional of the least-squares innovations, its Legendre transform L, and an **action functional** $S(T, \gamma) = \int_0^T L\, ds$ that measures how "costly" — how improbable — the path is.

The **dominant escape path** minimizes the action subject to leaving a neighbourhood G of $\bar{\gamma}$.

Drawing on [Williams, 2019], CWS reduce this to a clean control problem: the H-functional becomes a quadratic form with a normalizing matrix Q (a fourth-moment matrix obtained from Lyapunov equations), and the escape path solves

$$\bar{S} = \inf_{v(\cdot),\, T} \frac{1}{2} \int_0^T v(s)^\top Q(\gamma(s), R(s))^{-1} v(s)\, ds \tag{98.5}$$

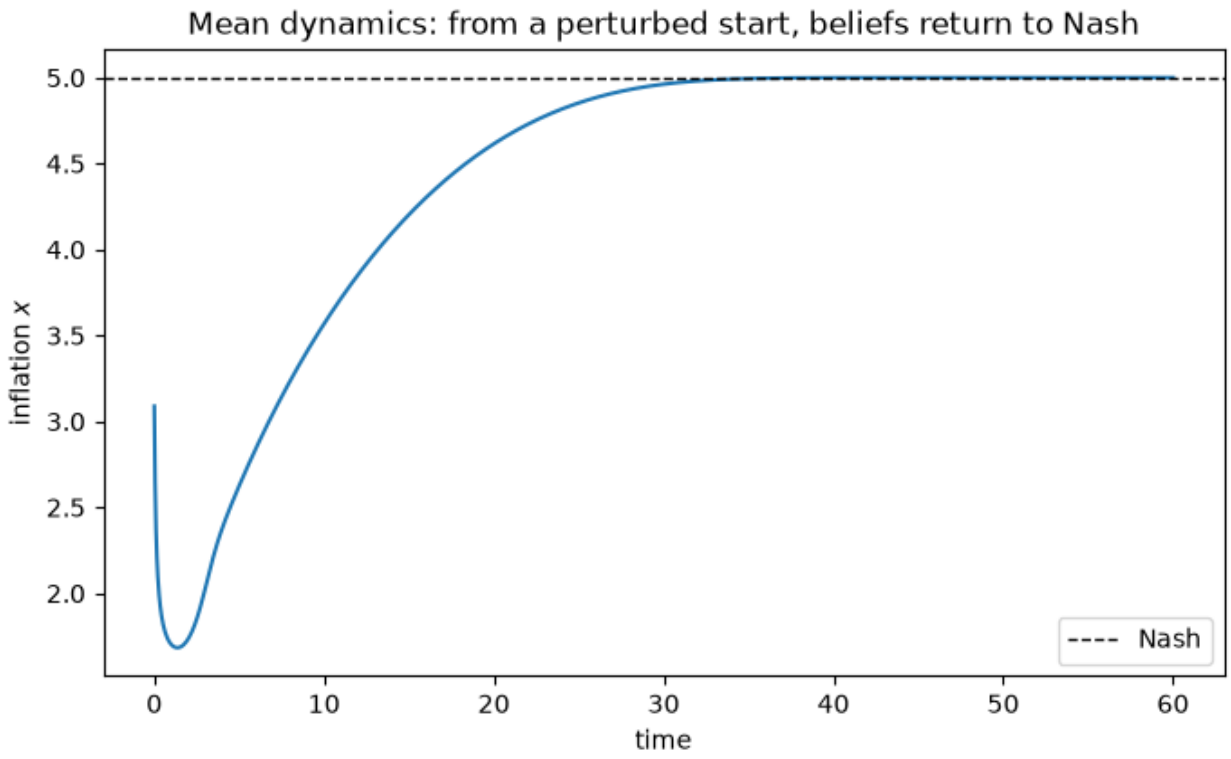


Fig. 98.2: Mean dynamics: from a perturbed start, beliefs return to Nash

subject to the *perturbed* mean dynamics

$$\dot{\gamma} = R^{-1}\bar{g}(\gamma) + v, \qquad \dot{R} = \bar{M}(\gamma) - R, \qquad \gamma(0) = \bar{\gamma},\ \gamma(T) \notin G. \tag{98.6}$$

Read (98.5) as a least squares problem: v is the extra "forcing" that the mean dynamics would need to escape, Q plays the role of a covariance matrix, and the least-cost forcing is the most likely unusual shock sequence.

Two consequences (their Theorem 5.3) tie the control problem to the stochastic model:

- the probability of an escape on a bounded interval is $\approx \exp(-\bar{S}/\varepsilon)$, so the **mean time between escapes** is $\approx \exp(\bar{S}/\varepsilon)$; and
- conditional on escaping, beliefs follow the dominant escape path with probability approaching one.

The escape dynamics, like the mean dynamics, are *deterministic*.

98.6 The dominant escape path

For the static model with binomial shocks, CWS solve the control problem in closed form (their Section 7).

The escape forcing is strikingly simple:

$$v = R^{-1}\begin{bmatrix}\sigma_1\sigma_2\\ 0\end{bmatrix}, \tag{98.7}$$

so the dominant escape path solves

$$\dot{\gamma} = R^{-1}\left(\bar{g}(\gamma) + \begin{bmatrix}\sigma_1\sigma_2\\ 0\end{bmatrix}\right), \qquad \dot{R} = \bar{M}(\gamma) - R. \tag{98.8}$$

Let's integrate (98.8) from the self-confirming equilibrium until beliefs leave a circle of radius 5.

```
def escape_ode(t, z, model):
    y, R = z[:2], z[2:].reshape(2, 2)
    force = np.array([model.σ1 * model.σ2, 0.0])
    return np.concatenate([np.linalg.inv(R) @ (model.g_bar(y) + force),
                           (model.M(y) - R).ravel()])

def left_circle(t, z):
    "Terminal event: beliefs leave the radius-5 circle around the SCE."
    return 5.0 - np.linalg.norm(z[:2] - y_sce)
left_circle.terminal = True
left_circle.direction = -1

z0 = np.concatenate([y_sce, model.M(y_sce).ravel()])
esc = solve_ivp(lambda t, z: escape_ode(t, z, model), [0, 200], z0,
                events=left_circle, max_step=0.02, rtol=1e-9, atol=1e-11)

slope, intercept = esc.y[0], esc.y[1]
infl = -intercept * slope / (1 + slope**2)
```

```
fig, axes = plt.subplots(1, 2, figsize=(12, 5))

axes[0].plot(esc.t, intercept, label='intercept $\\gamma_{-1}$')
axes[0].plot(esc.t, slope, label='slope $\\gamma_1$')
axes[0].axhline(0, color='C3', ls=':', lw=1, label='induction ($\\gamma_1 = 0$)')
axes[0].set_xlabel('time')
axes[0].set_ylabel('coefficient')
axes[0].set_title('Dominant escape path (cf. CWS Figure 4)')
axes[0].legend()

axes[1].plot(esc.t, infl)
axes[1].axhline(model.θ * model.u, color='k', ls='--', lw=1, label='Nash')
axes[1].axhline(0, color='C2', ls=':', lw=1, label='Ramsey')
axes[1].set_xlabel('time')
axes[1].set_ylabel('inflation $x$')
axes[1].set_title('Inflation along the escape')
axes[1].legend()

plt.tight_layout()
plt.show()
```

```
print(f"slope     : {slope[0]:.2f} → {slope[-1]:.3f}   (induction hypothesis at 0)")
print(f"intercept : {intercept[0]:.1f} → {intercept[-1]:.1f}")
print(f"inflation : {infl[0]:.2f} → {infl[-1]:.2f}   (Nash {model.θ*model.u:.0f} →␣
↪Ramsey 0)")
```

```
slope     : -1.00 → -0.023   (induction hypothesis at 0)
intercept : 10.0 → 5.1
inflation : 5.00 → 0.12   (Nash 5 → Ramsey 0)
```

Along the dominant escape path the slope rises from its self-confirming value of -1 toward zero — the **induction hypothesis** — and inflation falls from the Nash value toward Ramsey.

This is exactly the temporary stabilization we saw by simulation in *Adaptive Learning and Escape Dynamics*, now derived as a deterministic path: the government, by chance, generates enough inflation *experiments* to discover a good-enough

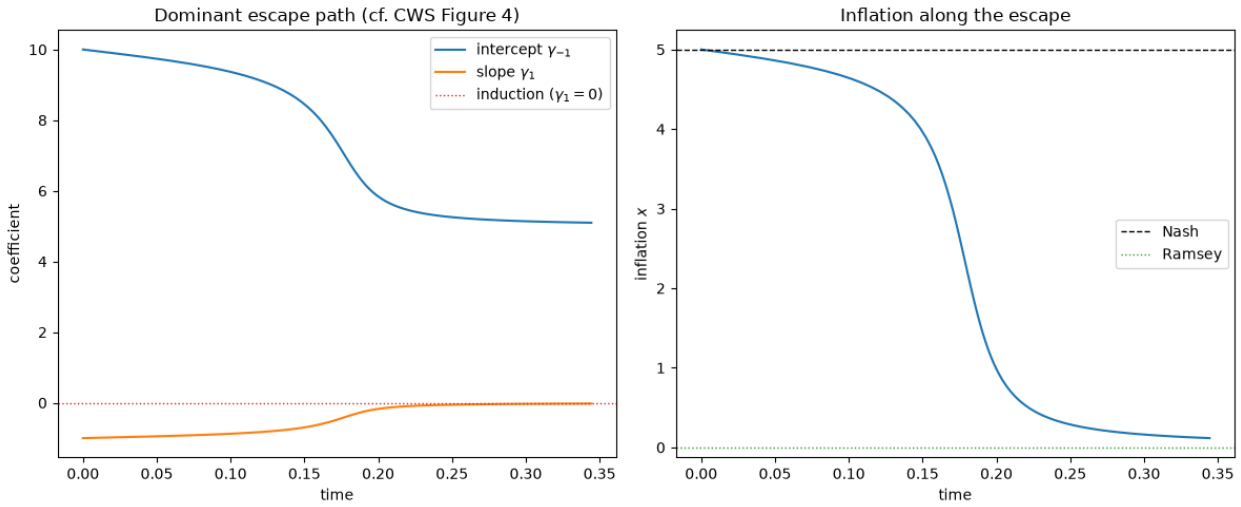


Fig. 98.3: The dominant escape path and the inflation rate along it

version of the natural-rate hypothesis, and acts on it.

98.7 The race of four ODEs

Where does the escape forcing (98.7) come from?

With binomial shocks $W_{in} \in \{-1, +1\}$, the shock pair realized in any period lies in one of four groups.

CWS show that the most likely escape uses the *same* unusual pair repeatedly, so there are four candidate escape ODEs — one per pair — and the dominant escape is the one that reaches the boundary fastest.

Let's compute the instantaneous velocity of each candidate at the self-confirming equilibrium.

```
R0 = model.M(y_sce)
R0_inv = np.linalg.inv(R0)
σ1σ2 = model.σ1 * model.σ2

x_sce = model.x(y_sce)

# each pair of shock realizations induces its own escape forcing
candidates = {
    "{(1,1),(-1,-1)}  → Ramsey":   np.array([σ1σ2, 0.0]),
    "{(1,-1),(-1,1)}  → higher π": np.array([-σ1σ2, 0.0]),
    "{(1,1),(1,-1)}":              np.array([x_sce * model.σ1, model.σ1]),
    "{(-1,1),(-1,-1)}":            np.array([-x_sce * model.σ1, -model.σ1]),
}

for name, force in candidates.items():
    v = R0_inv @ force                    # belief velocity along this candidate
    print(f"  {name:28s}  |velocity| = {np.linalg.norm(v):.3f}")
```

```
  {(1,1),(-1,-1)}  → Ramsey      |velocity| = 5.099
  {(1,-1),(-1,1)}  → higher π    |velocity| = 5.099
  {(1,1),(1,-1)}                 |velocity| = 0.300
  {(-1,1),(-1,-1)}               |velocity| = 0.300
```

The pair $\{(1,1),(-1,-1)\}$ produces a velocity far larger than the last two, and it points *toward* Ramsey (rising slope, falling intercept).

Its mirror image $\{(1,-1),(-1,1)\}$ has the same speed but points the wrong way — toward *higher* inflation — where the mean dynamics oppose it and quickly pull it back.

So the winner of the race is the Ramsey-ward path, and the escape forcing it induces is the $R^{-1}(\sigma_1\sigma_2,0)^\top$ of (98.7).

98.8 Mean dynamics reinforce the escape

Why does the Ramsey-ward escape succeed while its mirror image fails?

Near the self-confirming equilibrium the mean dynamics point *toward* it, opposing any escape.

But CWS (their Figures 8-9) show that once beliefs have moved a little way out along the Ramsey-ward direction, the mean dynamics themselves start pointing *toward* Ramsey, reinforcing the escape.

We can see this by plotting the mean-dynamics vector field in belief space.

```
gs = np.linspace(-1.2, 0.1, 16)       # slope
gi = np.linspace(4.5, 10.5, 16)       # intercept
GS, GI = np.meshgrid(gs, gi)
DS, DI = np.zeros_like(GS), np.zeros_like(GI)

for i in range(GS.shape[0]):
    for j in range(GS.shape[1]):
        y = np.array([GS[i, j], GI[i, j]])
        d = np.linalg.inv(model.M(y)) @ model.g_bar(y)
        DS[i, j], DI[i, j] = d

fig, ax = plt.subplots(figsize=(8, 6))
ax.quiver(GS, GI, DS, DI, angles='xy', alpha=0.7)
ax.plot(*y_sce, 'ko', ms=8, label='SCE (Nash)')
ax.plot(0, model.u, 'C2s', ms=8, label='Ramsey belief')
ax.plot(slope, intercept, 'C3', lw=2, label='escape path')
ax.set_xlabel(r'slope $\gamma_1$')
ax.set_ylabel(r'intercept $\gamma_{-1}$')
ax.set_title('Mean dynamics (arrows) and the escape path')
ax.legend()
plt.show()
```

Near the self-confirming equilibrium the arrows push back toward Nash, but away from it — along the escape route — they sweep toward the Ramsey belief.

The escape dynamics only need to *start* the departure; the mean dynamics finish the job.

This is the sense in which the mean dynamics trace a *circuitous* route: pushed off the equilibrium along the escape path, the system travels near Ramsey before the residual short-run Phillips curve is rediscovered and the mean dynamics eventually carry beliefs back to Nash.

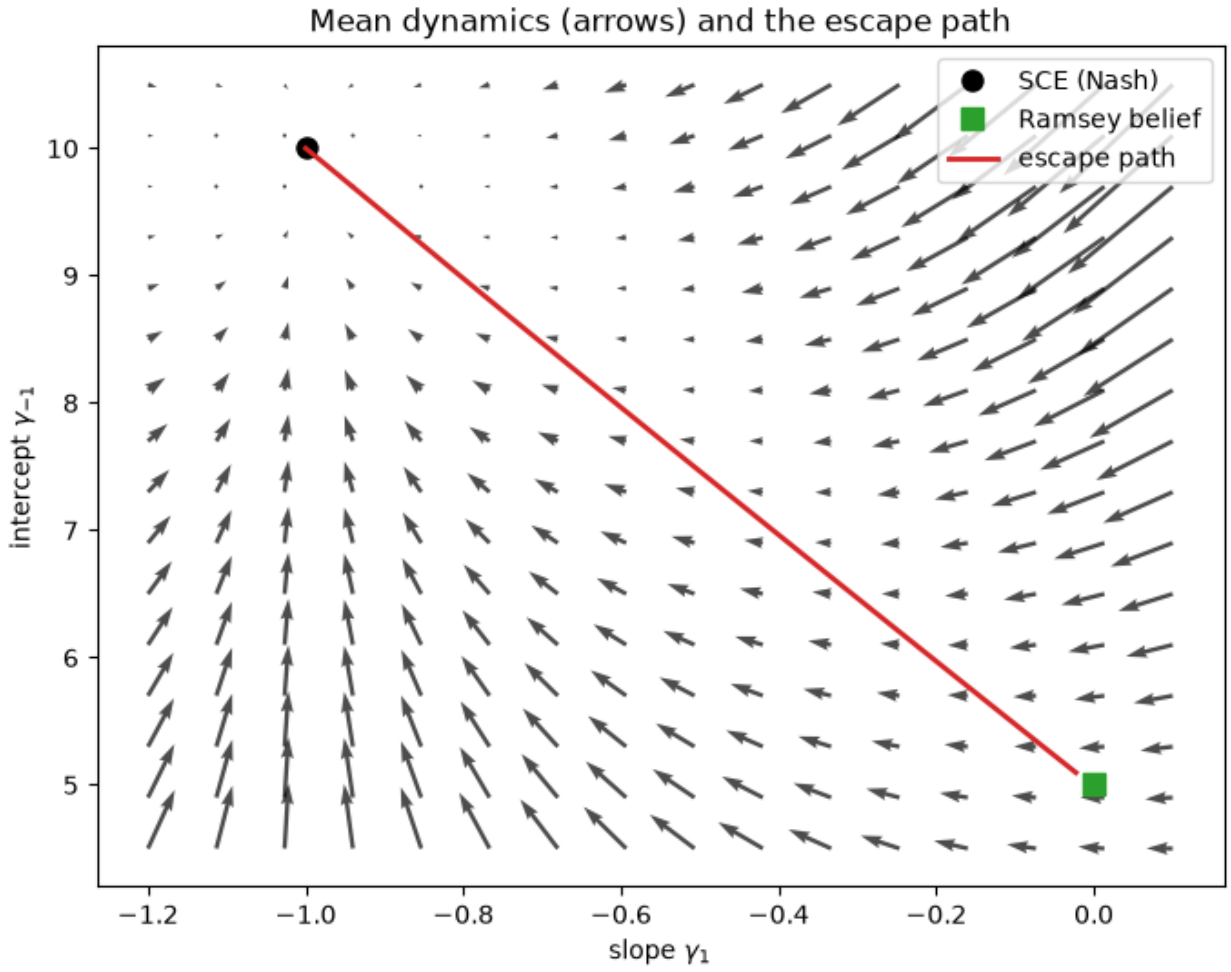


Fig. 98.4: Mean-dynamics vector field with the escape path superimposed

98.9 The experimentation trap

The escape has a compelling behavioural interpretation.

Within its approximating model, the government can only detect the natural-rate hypothesis if there is enough *dispersion* in inflation.

But inside a self-confirming equilibrium the government sets a constant systematic inflation rate, so it generates no such dispersion — it is caught in an **experimentation trap**.

Only an unusual run of shocks makes the government vary inflation enough to steepen its estimated Phillips curve; a steeper perceived curve leads it (through the best response) to cut inflation, which generates further influential observations that steepen the curve further.

This self-reinforcing process halts when the perceived Phillips curve is vertical — the induction hypothesis — and inflation is near Ramsey.

The system cannot stay there forever: in truth there *is* a short-run Phillips curve, which the government eventually rediscovers, rekindling the mean dynamics that carry it back to Nash.

98.10 Escape frequency and model richness

The minimized action $\bar{S}$ governs how often escapes occur: the mean escape time grows like $\exp(\bar{S}/\varepsilon)$.

A striking finding of CWS is that escapes are *more frequent* when the government's model is richer.

The full dynamic model of *Adaptive Learning and Escape Dynamics* — with lagged unemployment and inflation — has a much smaller $\bar{S}$ than the static model, even though the two share the same self-confirming equilibrium.

A richer model lets the government detect the subtler distributed-lag ("induction-hypothesis") version of the natural-rate hypothesis, so it escapes toward Ramsey more readily.

Note

The escape dynamics inherit the same "near determinism" that makes the mean dynamics useful: for small gains, the stochastic simulations of *Adaptive Learning and Escape Dynamics* hug the deterministic escape path derived here.

The next lecture, *Priors, Escapes, and Learning Cycles*, shows that the government's *prior* about how its coefficients drift reshapes both dynamics — and can even make the escape a deterministic *cycle*.

98.11 Escaping volatile inflation

The escape delivers a fall in the *level* of inflation.

[Ellison and Yates, 2007] extend the model to explain a second post-war fact: inflation *volatility* rose and fell together with the level.

Their device is to give the government a reason to stabilize.

Following [Phelps and Taylor, 1977], they add an unemployment shock W_3 that the government — but not the price-setting private sector — can react to.

Now a government that believes in an exploitable Phillips curve is tempted to *lean against* W_3 by varying inflation, so the perceived effectiveness of policy, $|\gamma_1|$, drives the volatility of inflation as well as its level.

In their model the expected inflation volatility a private agent faces is

$$\mathbb{E}(\sigma_\pi \mid \gamma) = \left[\sigma_2^2 + \left(\frac{\gamma_1}{1+\gamma_1^2}\right)^2 \sigma_3^2\right]^{1/2}. \tag{98.9}$$

At the self-confirming equilibrium $\gamma_1 = -\theta$, the government believes policy is effective and leans against W_3 aggressively, so inflation is *volatile*.

Along an escape, $\gamma_1 \to 0$: the government stops believing it can exploit the Phillips curve, abandons stabilization, and the volatility term collapses to the control error σ_2.

Applying (98.9) to the belief path we already computed shows the level and volatility of inflation escaping *in tandem*.

```
σ3 = 0.9                                    # size of the stabilizable shock
infl_vol = np.sqrt(model.σ2**2 + (slope / (1 + slope**2))**2 * σ3**2)

fig, axes = plt.subplots(1, 2, figsize=(12, 4.5))
axes[0].plot(esc.t, infl)
axes[0].axhline(model.θ * model.u, color='k', ls='--', lw=1, label='Nash')
axes[0].axhline(0, color='C2', ls=':', lw=1, label='Ramsey')
axes[0].set_xlabel('time'); axes[0].set_ylabel('inflation level')
axes[0].set_title('level escapes'); axes[0].legend()

axes[1].plot(esc.t, infl_vol, 'C1')
axes[1].axhline(model.σ2, color='k', ls='--', lw=1, label=r'control error $\sigma_2$')
axes[1].set_xlabel('time'); axes[1].set_ylabel('inflation volatility')
axes[1].set_title('volatility escapes too'); axes[1].legend()

plt.tight_layout()
plt.show()
```

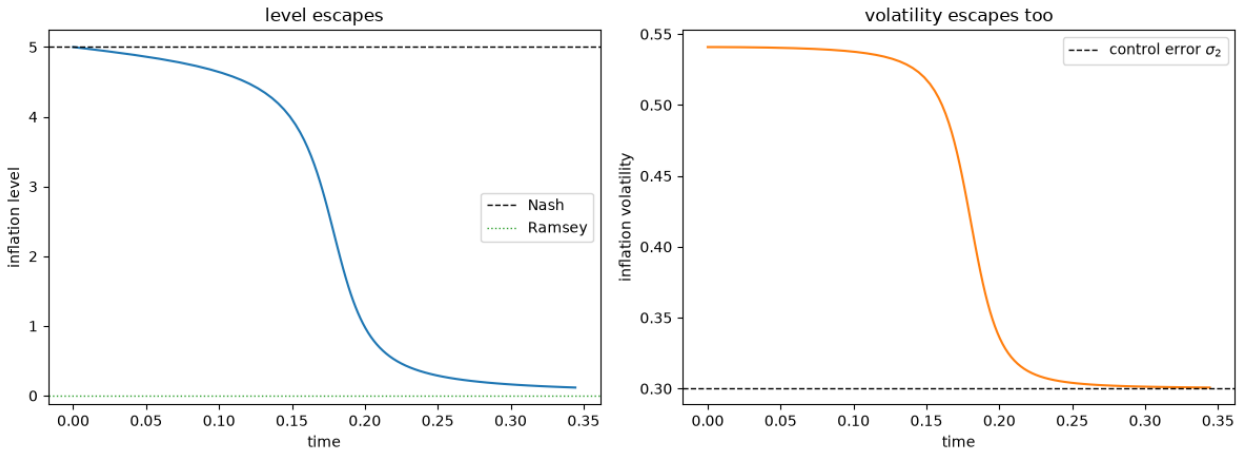


Fig. 98.5: The level and volatility of inflation escape in tandem

Both fall as beliefs escape, because both spring from the same source: the government's belief in an exploitable tradeoff.

[Ellison and Yates, 2007] draw a further, subtler lesson about the *timing* of escapes.

A larger stabilizable shock σ_3 makes an escape *harder to trigger*: to create the illusion that inflation moves while unemployment stays put, an unusual sequence of shocks must now offset not only the control errors but also the government's own stabilizing reaction to W_3.

The more shocks the government can offset, the more complex the escape-triggering sequence, and the longer the wait.

Taken literally, this says an economy is more likely to escape to low inflation precisely when there are *few* shocks to stabilize — a suggestive link between the arrival of the mid-1980s calm and the disinflation that accompanied it.

It is also a claim about data, and it points to the empirical lecture that closes this suite.

If the calm and the disinflation arrived together, then a statistical model must be able to separate *shrinking shocks* from *shifting dynamics* before it can say which caused which.

Drifts and Volatilities builds a model with room for both channels and lets the data apportion them.

98.12 Exercises

Exercise 98.12.1

The escape forcing (98.7) scales with $\sigma_1\sigma_2$ — the product of the two shock standard deviations.

Integrate the dominant escape ODE (98.8) for a grid of $\sigma_1 = \sigma_2 = \sigma \in \{0.2, 0.3, 0.4, 0.5\}$ and report, for each, the *exit time* from the radius-5 circle.

How does a noisier economy affect how quickly beliefs travel along the escape route?

Solution

```
for σ in [0.2, 0.3, 0.4, 0.5]:
    m = EscapeModel(θ=1.0, u=5.0, σ1=σ, σ2=σ)
    y0 = np.array([-m.θ, m.u * (1 + m.θ**2)])

    def leave(t, z, m=m, y0=y0):
        return 5.0 - np.linalg.norm(z[:2] - y0)
    leave.terminal = True
    leave.direction = -1

    z0 = np.concatenate([y0, m.M(y0).ravel()])
    s = solve_ivp(lambda t, z: escape_ode(t, z, m), [0, 500], z0,
                  events=leave, max_step=0.02, rtol=1e-9, atol=1e-11)
    print(f"σ = {σ}: exit time along the escape path = {s.t[-1]:.2f}")
```

```
σ = 0.2: exit time along the escape path = 0.20
σ = 0.3: exit time along the escape path = 0.34
σ = 0.4: exit time along the escape path = 0.50
σ = 0.5: exit time along the escape path = 0.67
```

A larger σ makes the escape forcing $R^{-1}(\sigma_1\sigma_2, 0)^\top$ stronger, so beliefs travel the escape route faster (a shorter exit *time* along the deterministic path).

Note that this is distinct from the *frequency* of escapes, which is governed by the action $\bar{S}$ and the gain ε; a noisier economy travels a given escape route more quickly once the escape is under way.

Exercise 98.12.2

Make the reinforcement in the vector-field plot quantitative.

At several points *along* the escape path, evaluate the mean-dynamics drift $R^{-1}\bar{g}(\gamma)$ and measure its cosine alignment with the direction from the current beliefs toward the Ramsey belief $(0, u)$.

A cosine near $+1$ means the mean dynamics are pushing beliefs *toward* Ramsey — reinforcing the escape.

Solution

```
γ_ramsey = np.array([0.0, model.u])            # Belief 2

def cosine_toward_ramsey(γ, R):
    drift = np.linalg.inv(R) @ model.g_bar(γ)
    to_ramsey = γ_ramsey - γ
    denom = np.linalg.norm(drift) * np.linalg.norm(to_ramsey)
    return np.nan if denom < 1e-9 else drift @ to_ramsey / denom

for frac in [0.0, 0.25, 0.5, 0.75, 0.95]:
    k = min(int(frac * len(esc.t)), len(esc.t) - 1)
    γ, R = esc.y[:2, k], esc.y[2:, k].reshape(2, 2)
    print(f"frac {frac:.2f}:  γ = ({γ[0]:+.2f}, {γ[1]:.1f}),  "
          f"cos(drift, →Ramsey) = {cosine_toward_ramsey(γ, R):+.2f}")
```

```
frac 0.00:  γ = (-1.00, 10.0),  cos(drift, →Ramsey) = +nan
frac 0.25:  γ = (-0.65, 8.2),  cos(drift, →Ramsey) = +1.00
frac 0.50:  γ = (-0.23, 6.1),  cos(drift, →Ramsey) = +1.00
frac 0.75:  γ = (-0.07, 5.3),  cos(drift, →Ramsey) = +1.00
frac 0.95:  γ = (-0.03, 5.1),  cos(drift, →Ramsey) = +1.00
```

Right at the self-confirming equilibrium the drift vanishes (the cosine is undefined), so the mean dynamics neither help nor hinder.

But once beliefs have moved even slightly along the escape route, the mean-dynamics drift points almost exactly toward the Ramsey belief (cosine $\approx +1$): the mean dynamics reinforce the escape all the way to Ramsey.

The opposition that CWS emphasize is confined to a *tiny* neighbourhood of the equilibrium — the escape dynamics only need to nudge beliefs out of it, after which the mean dynamics finish the job.

CHAPTER

NINETYNINE

PRIORS, ESCAPES, AND LEARNING CYCLES

Contents

- *Priors, Escapes, and Learning Cycles*
  - *Overview*
  - *The static model*
  - *The self-confirming equilibrium*
  - *Drifting beliefs and the Kalman filter*
  - *Mean dynamics and E-stability*
  - *Learning cycles*
  - *Escape dynamics and the direction of escape*
  - *Sims's nonconvergence*
  - *Conclusion*
  - *Exercises*

99.1 Overview

> The Bourbons remember everything and learn nothing.
>
> – Charles Maurice de Talleyrand

This lecture is a sequel to *Adaptive Learning and Escape Dynamics*.

It follows [Sargent and Williams, 2005], which generalizes the adaptive model of chapter 8 of [Sargent, 1999] and of [Cho *et al.*, 2002] (CWS), whose escape dynamics we derived in *Escaping Nash Inflation*.

In *Adaptive Learning and Escape Dynamics* the government estimated its Phillips curve by recursive least squares.

That is a very particular way to learn.

A least squares learner behaves as if it believes that the coefficients of its model follow a specific random walk — one with a specific innovation covariance matrix.

Here we free the government to hold *any* prior about how its coefficients drift, encoded in a covariance matrix V, and we ask how the shape of that prior affects the two forces that drive the model:

- the **mean dynamics**, which pull the government's beliefs *toward* a self-confirming (Nash) equilibrium; and
- the **escape dynamics**, which occasionally push beliefs *away* from it, toward the low-inflation Ramsey outcome.

We will find three things.

1. Some priors make the self-confirming equilibrium *unstable*, so that the mean dynamics themselves produce recurrent disinflations — a **learning cycle** born of a Hopf bifurcation, rather than a rare stochastic escape.
2. The prior shapes the *direction* and *speed* of escapes; but in every case the escape heads toward the Ramsey outcome.
3. The prior explains a long-standing puzzle: why the simulations of Sims and Chung escape Nash inflation and stay near Ramsey *forever*, while those of [Sargent, 1999] and CWS escape only to be pulled back.

Throughout we work with the analytically tractable **static** model of *Escaping Nash Inflation*, in which the government runs a simple regression of unemployment on inflation and a constant.

Let's start with our imports:

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.linalg import sqrtm
from scipy.integrate import solve_ivp
```

99.2 The static model

The true economy is a version of the natural-rate model of [Kydland and Prescott, 1977]:

$$\begin{aligned} U_n &= u - (\pi_n - \hat{x}_n) + \sigma_1 W_{1n}, \qquad u > 0, \\ \pi_n &= x_n + \sigma_2 W_{2n}, \\ \hat{x}_n &= x_n, \end{aligned} \tag{99.1}$$

where U_n is unemployment, π_n is inflation, x_n is the systematic part of inflation set by the government, $\hat{x}_n$ is the public's (rational) forecast, and $W_n = (W_{1n}, W_{2n})^\top$ is IID standard Gaussian noise.

Since $\pi_n - \hat{x}_n = \sigma_2 W_{2n}$, the true unemployment rate is $U_n = u - \sigma_2 W_{2n} + \sigma_1 W_{1n}$ — it fluctuates around the natural rate u regardless of systematic policy.

The government does not know this.

In the **static model** it fits a non-expectational Phillips curve by regressing unemployment on current inflation and a constant,

$$U_n = a + b\,\pi_n + \eta_n, \tag{99.2}$$

with belief vector $\gamma = (a, b)$ (intercept and slope), and it treats η_n as an exogenous shock.

> **Warning**
>
> *Escaping Nash Inflation* studies this same static model but orders the belief vector the other way round, as $\gamma = (\gamma_1, \gamma_{-1})$ with the *slope* first and regressors $\Phi = (\pi, 1)$, following [Cho *et al.*, 2002].
>
> Here we put the *intercept* first, with $\Phi = (1, \pi)$, following [Sargent and Williams, 2005].
>
> The two are the same model in different coordinates: $a = \gamma_{-1}$ and $b = \gamma_1$, so the self-confirming belief $(2u, -1)$ of this lecture is the $(-1, u(1+\theta^2))$ of the last one, with $\theta = 1$.
>
> Matrices such as M, V, and P have their rows and columns correspondingly transposed.

Believing (99.2), the government solves the Phelps problem — minimize $\widehat{\mathbb{E}} \sum_n \delta^n (U_n^2 + \pi_n^2)$ — whose static best response sets inflation to the constant

$$x(\gamma) = -\frac{a\,b}{1+b^2}. \tag{99.3}$$

```
class StaticPhillips:
    "The static Sargent-Williams model: government regresses U on (1, π)."

    def __init__(self, u=5.0, σ1=0.3, σ2=0.3):
        self.u, self.σ1, self.σ2 = u, σ1, σ2
        self.σ = σ1                         # govt regression error std = σ1

    def x(self, γ):
        "Government best response (systematic inflation) given beliefs γ."
        a, b = γ
        return -a * b / (1 + b**2)

    def M(self, γ):
        "Second moment matrix E[ΦΦ'] of regressors Φ = (1, π), given γ."
        x = self.x(γ)
        return np.array([[1.0, x], [x, x**2 + self.σ2**2]])

    def g_bar(self, γ):
        "Least squares moment E[Φ(U - Φ'γ)] under the data γ generates."
        u, σ2 = self.u, self.σ2
        x = self.x(γ)
        E_ΦU = np.array([u, x * u - σ2**2])
        return E_ΦU - self.M(γ) @ γ
```

Three belief vectors are worth naming.

- **Belief 1 (Nash):** $b = -1$ with an intercept that makes the government set $x = u$, the time-consistent outcome of [Kydland and Prescott, 1977].
- **Belief 2 (Ramsey):** $b = 0$, so the government perceives *no* tradeoff and sets $x = 0$.
- **Belief 3 (induction):** in a dynamic version, coefficients on current and lagged inflation summing to zero, which for a patient government also sends inflation toward 0.

99.3 The self-confirming equilibrium

A self-confirming equilibrium is a belief $\bar{\gamma}$ that reproduces itself: the data generated when the government acts on $\bar{\gamma}$ make the population regression coefficients equal $\bar{\gamma}$, i.e. $\bar{g}(\bar{\gamma}) = 0$.

For the static model this is easy to solve by hand.

The slope is $b = \mathrm{cov}(U, \pi)/\mathbb{V}[\pi] = -\sigma_2^2/\sigma_2^2 = -1$, and matching means gives the intercept $a = u + x(\bar{\gamma})$.

Substituting the best response (99.3) with $b = -1$ gives $x = a/2$, so $a = u + a/2$, i.e. $a = 2u$.

```
model = StaticPhillips(u=5.0, σ1=0.3, σ2=0.3)
γ_sce = np.array([2 * model.u, -1.0])

print(f"self-confirming beliefs  γ = {γ_sce}  (intercept 2u, slope -1)")
print(f"self-confirming inflation x = {model.x(γ_sce):.2f}  (= Nash = u)")
print(f"check g_bar(γ_sce) = {model.g_bar(γ_sce)}")
```

```
self-confirming beliefs  γ = [10. -1.]  (intercept 2u, slope -1)
self-confirming inflation x = 5.00  (= Nash = u)
check g_bar(γ_sce) = [0. 0.]
```

The self-confirming equilibrium inflation rate equals the Nash (time-consistent) outcome $u = 5$, even though the government's model is misspecified.

The government's non-expectational Phillips curve is observationally equivalent to the truth *along* the equilibrium path, but wrong *off* it — which is exactly what its adaptive behavior repeatedly probes.

99.4 Drifting beliefs and the Kalman filter

We now make the government adaptive.

Following [Sargent and Williams, 2005], the government believes that the coefficients of its Phillips curve *drift* as a random walk,

$$\alpha_n = \alpha_{n-1} + \Lambda_n, \qquad \operatorname{cov}(\Lambda_n) = V, \tag{99.4}$$

and it forms its estimate $\gamma_n = \hat{\alpha}_{n|n-1}$ by the Kalman filter.

The covariance matrix V is the government's **prior about parameter drift** — the object we set free.

With regressors $\Phi_n = (1, \pi_n)^\top$, a large-sample approximation to the Kalman filter (see [Benveniste *et al.*, 1990]) is

$$\begin{aligned} \gamma_{n+1} &= \gamma_n + P_n \Phi_n \left(U_n - \Phi_n^\top \gamma_n\right), \\ P_{n+1} &= P_n - P_n M(\gamma_n) P_n + \sigma^{-2} V, \end{aligned} \tag{99.5}$$

where σ^2 is the variance the government attributes to its regression error η_n.

For a fixed γ, the matrix P_n settles at the solution of the algebraic Riccati equation

$$-PM(\gamma)P + \sigma^{-2} V = 0. \tag{99.6}$$

The connection to *Adaptive Learning and Escape Dynamics* is exact.

Constant-gain recursive least squares is the special case in which the government's prior is $V = V^* \equiv \epsilon^2 \sigma^2 M(\bar{\gamma})^{-1}$ and $\sigma = \sigma_1$; then (99.6) gives $P = \epsilon M(\bar{\gamma})^{-1}$, and (99.5) reduces to the recursive least squares algorithm of the previous lecture, with gain ϵ.

```
def solve_riccati(V, M, σ):
    "Symmetric positive-definite P solving P M P = σ^{-2} V."
    W = V / σ**2
    Mh = sqrtm(M).real
    Mh_inv = np.linalg.inv(Mh)
    return Mh_inv @ sqrtm(Mh @ W @ Mh).real @ Mh_inv

M_sce = model.M(γ_sce)
V_star = model.σ**2 * np.linalg.inv(M_sce)      # the RLS prior (ε = 1)
P_star = solve_riccati(V_star, M_sce, model.σ)

print("RLS prior gives P = M^{-1}?", np.allclose(P_star, np.linalg.inv(M_sce)))
```

```
RLS prior gives P = M^{-1}? True
```

99.5 Mean dynamics and E-stability

As in *Adaptive Learning and Escape Dynamics*, the beliefs are organized by mean dynamics — now a *joint* ordinary differential equation in (γ, P),

$$\dot{\gamma} = P\,\bar{g}(\gamma), \qquad \dot{P} = \sigma^{-2}V - PM(\gamma)P. \tag{99.7}$$

A rest point of (99.7) has $\bar{g}(\gamma) = 0$ and $P =$ the Riccati solution — a self-confirming equilibrium.

Local stability turns on the Jacobian of $\bar{g}$ at the self-confirming equilibrium.

For the static model this can be computed in closed form,

$$\frac{\partial \bar{g}}{\partial \gamma}(\bar{\gamma}) = -\begin{bmatrix} \frac{1}{2} & u \\ \frac{1}{2}u & u^2 + \sigma_2^2 \end{bmatrix}. \tag{99.8}$$

```
def jacobian(model, γ, h=1e-6):
    "Numerical Jacobian of g_bar at γ."
    J = np.zeros((2, 2))
    for j in range(2):
        gp, gm = γ.copy(), γ.copy()
        gp[j] += h; gm[j] -= h
        J[:, j] = (model.g_bar(gp) - model.g_bar(gm)) / (2 * h)
    return J

J = jacobian(model, γ_sce)
print("∂g/∂γ at SCE =\n", J.round(3))
```

```
∂g/∂γ at SCE =
 [[ -0.5   -5.  ]
 [ -2.5  -25.09]]
```

Under recursive least squares the P block of (99.7) decouples, and stability is governed by the eigenvalues of $M^{-1}\,\partial\bar{g}/\partial\gamma$ — the classic **E-stability** condition of [Evans and Honkapohja, 2001].

```
eig_rls = np.linalg.eigvals(np.linalg.inv(M_sce) @ J)
print(f"E-stability eigenvalues (RLS): {eig_rls.round(3)}")
print("both negative ⇒ the SCE is E-stable, and least squares converges to Nash")
```

```
E-stability eigenvalues (RLS): [-0.5 -1. ]
both negative ⇒ the SCE is E-stable, and least squares converges to Nash
```

Both eigenvalues are negative — one of them exactly $-\frac{1}{2}$, the marginal value that appeared in *Adaptive Learning and Escape Dynamics*.

Under least squares, then, beliefs converge to the Nash self-confirming equilibrium.

But this reduction relied on the special RLS prior.

Under a *general* prior V, the P block does **not** decouple, and stability is instead governed by the eigenvalues of $\bar{P}\,\partial\bar{g}/\partial\gamma$ — where $\bar{P}$ is the Riccati solution for that prior.

Prior beliefs about parameter drift now matter for whether the self-confirming equilibrium is even stable.

99.6 Learning cycles

Here is the paper's most striking result: some priors make the self-confirming equilibrium *unstable*, and a **stable limit cycle** is born through a Hopf bifurcation.

We illustrate it by *tightening the government's prior on the slope coefficient*, starting from the RLS prior V^* and shrinking the slope-related entries by a factor $\lambda \in [0, 1]$:

$$V(\lambda) = \begin{bmatrix} V_{11}^* & \sqrt{\lambda}\, V_{12}^* \\ \sqrt{\lambda}\, V_{12}^* & \lambda\, V_{22}^* \end{bmatrix}. \tag{99.9}$$

For each λ we solve the Riccati equation and look at the largest real part among the eigenvalues of $\bar{P}(\lambda)\, \partial \bar{g}/\partial \gamma$: where it is positive, the self-confirming equilibrium is unstable.

```
def V_tighten_slope(λ, V_star):
    V = V_star.copy()
    V[0, 1] = V[1, 0] = np.sqrt(λ) * V_star[0, 1]
    V[1, 1] = λ * V_star[1, 1]
    return V

ε = 0.05                                    # gain (sets the timescale)
λ_grid = np.linspace(0.01, 0.999, 200)
max_re = []
for λ in λ_grid:
    V = ε**2 * V_tighten_slope(λ, V_star)
    P = solve_riccati(V, M_sce, model.σ)
    max_re.append(np.linalg.eigvals(P @ J).real.max())

fig, ax = plt.subplots(figsize=(8, 4.5))
ax.plot(λ_grid, max_re)
ax.axhline(0, color='k', lw=0.8)
ax.set_xlabel(r'prior-tightening parameter $\lambda$')
ax.set_ylabel('max real part of eigenvalue')
plt.show()
```

Over an intermediate range of λ the maximum real part becomes *positive*: the self-confirming equilibrium loses stability.

By the Hopf bifurcation theorem (see [Perko, 1996]), a unique stable limit cycle bifurcates as the real parts cross zero — what [Bullard, 1994] calls a *learning equilibrium*.

Let's integrate the mean dynamics for the regression coefficients at $\lambda = 0.7$ (well inside the unstable range), holding P at its Riccati value, and trace the cycle.

```
λ = 0.7
V = ε**2 * V_tighten_slope(λ, V_star)
P_bar = solve_riccati(V, M_sce, model.σ)

def coeff_ode(t, γ):
    return P_bar @ model.g_bar(γ)

sol = solve_ivp(coeff_ode, [0, 2500], γ_sce + np.array([0.3, 0.05]),
                max_step=1.0, rtol=1e-9, atol=1e-11, dense_output=True)
a_path, b_path = sol.y
x_path = -a_path * b_path / (1 + b_path**2)

# isolate one mature cycle for the phase plot
mask = sol.t > sol.t[-1] - 800
```

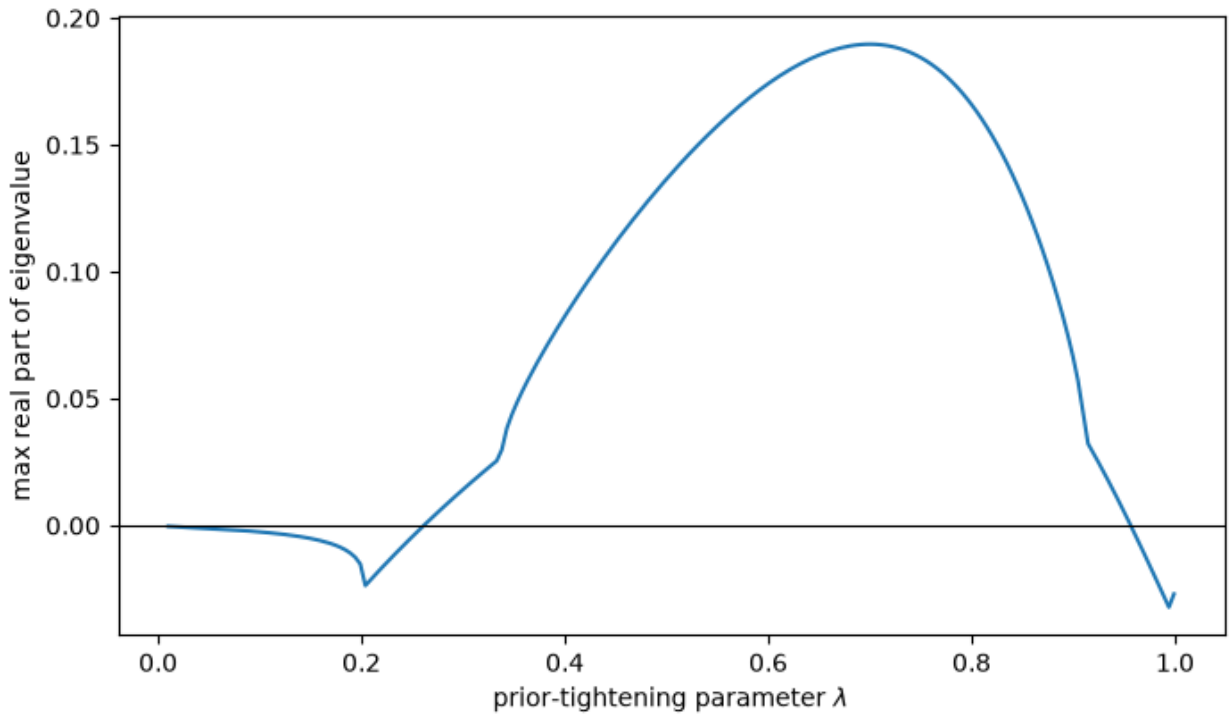


Fig. 99.1: Stability of the self-confirming equilibrium as the slope prior tightens

```
fig, axes = plt.subplots(1, 2, figsize=(12, 5))

axes[0].plot(sol.t[mask], a_path[mask], label='intercept', lw=2)
axes[0].plot(sol.t[mask], b_path[mask], label='slope', lw=2)
axes[0].set_xlabel('time')
axes[0].set_ylabel('coefficient')
axes[0].set_title('Figure 5a: coefficients cycle')
axes[0].legend()

axes[1].plot(a_path[mask], b_path[mask], lw=2)
axes[1].plot(*y_sce, 'kx', ms=10, label='SCE')
axes[1].set_xlabel('intercept')
axes[1].set_ylabel('slope')
axes[1].set_title('Figure 5b: the limit cycle')
axes[1].legend()

plt.tight_layout()
plt.show()
```

The beliefs settle into a closed orbit around the self-confirming equilibrium.

Because inflation is a function of the coefficients through the best response (99.3), the cycle in beliefs shows up as a cycle in inflation that oscillates between the Nash and Ramsey outcomes.

```
fig, ax = plt.subplots(figsize=(9, 4.5))
ax.plot(sol.t[mask], x_path[mask])
ax.axhline(model.u, color='k', ls='--', lw=1, label='Nash')
```

(continues on next page)

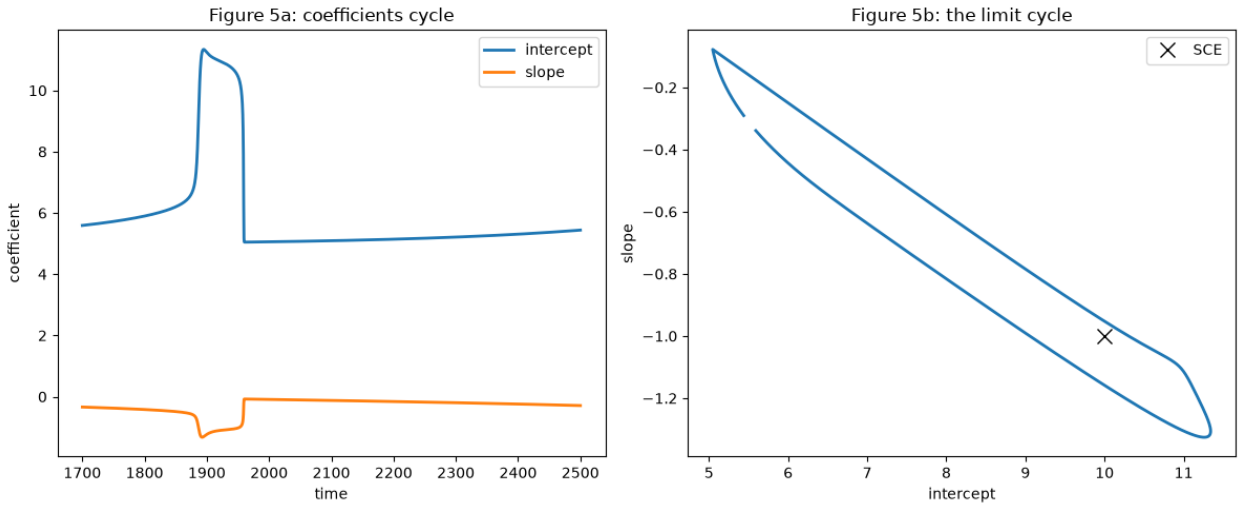


Fig. 99.2: Coefficients cycle, and the limit cycle in belief space

(continued from previous page)

```
ax.axhline(0, color='C2', ls=':', lw=1, label='Ramsey')
ax.set_xlabel('time')
ax.set_ylabel('inflation $x$')
ax.legend()
plt.show()
```

This is qualitatively different from the escapes of *Adaptive Learning and Escape Dynamics*.

There, disinflations were *rare* events, driven by unlikely sequences of shocks; between them the system sat at Nash.

Here the disinflations are a *typical* feature of the time series, produced by the mean dynamics themselves — a deterministic cycle that persists even as the gain shrinks to zero.

99.7 Escape dynamics and the direction of escape

For priors that keep the self-confirming equilibrium stable, disinflations return to being rare escapes, exactly as in *Adaptive Learning and Escape Dynamics*.

The theory of large deviations (developed in the primer of *Adaptive Learning and Escape Dynamics*, and applied here via [Williams, 2019]) characterizes the most likely escape as the solution of a control problem: apply the least-cost perturbation to beliefs that pushes them a fixed distance from $\bar{\gamma}$.

The *instantaneous* escape direction has a beautifully simple characterization: it is the eigenvector associated with the largest eigenvalue of the belief-innovation covariance $Q(\bar{\gamma}, \bar{P}) = \hat{V}$ — the prior itself.

```
w, vecs = np.linalg.eigh(V_star)            # baseline prior V*
v_escape = vecs[:, np.argmax(w)]
v_escape = v_escape / np.linalg.norm(v_escape)

# scale the escape so the slope moves from -1 up to 0
t_star = 1.0 / v_escape[1]
terminal = γ_sce + t_star * v_escape
```

(continues on next page)

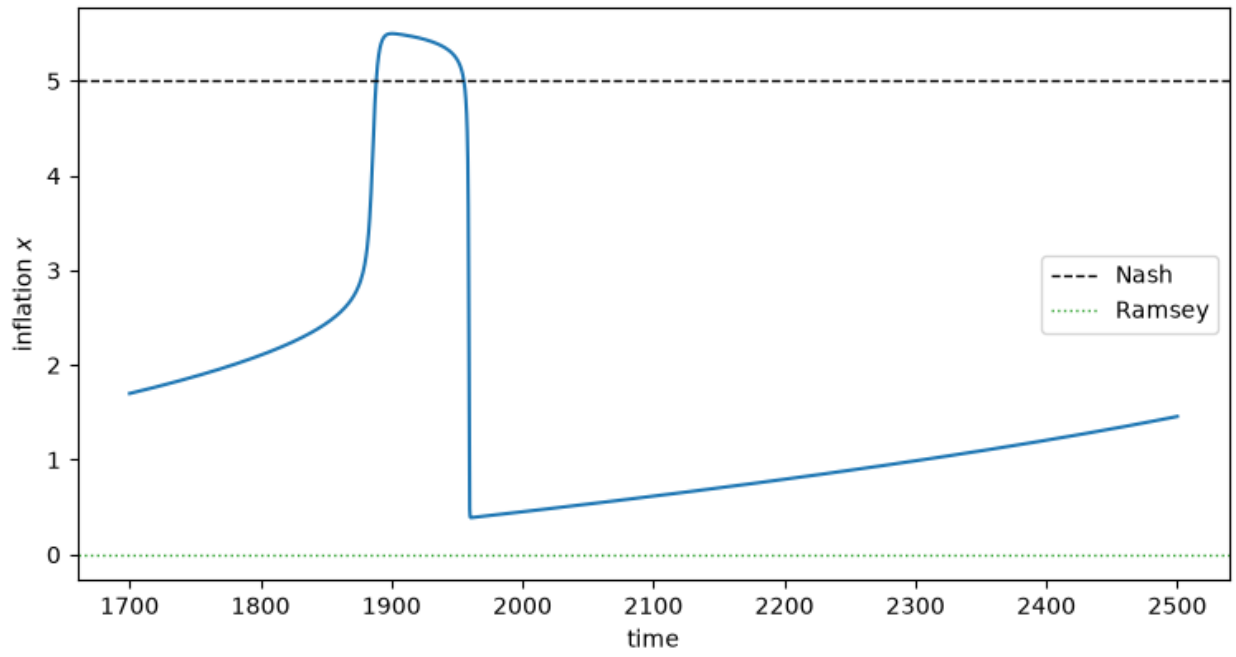


Fig. 99.3: Inflation oscillates between Nash and Ramsey along the learning cycle

(continued from previous page)

```
print(f"escape direction  ≈ {v_escape.round(3)}   (∝ [-u, 1])")
print(f"terminal beliefs  ≈ {terminal.round(2)}   (Belief 2 = [u, 0] = Ramsey)")
```

```
escape direction  ≈ [-0.981  0.195]   (∝ [-u, 1])
terminal beliefs  ≈ [4.98 0.  ]   (Belief 2 = [u, 0] = Ramsey)
```

For the baseline prior the escape direction is proportional to $(-u, 1)$, and it carries beliefs from Nash $(2u, -1)$ toward $(u, 0)$ — Belief 2, which supports the Ramsey outcome.

So an escape, when it happens, is a movement toward zero inflation.

Different priors bend the *path* and change the *rate* of escape — tightening the slope prior can even destabilize the equilibrium entirely (the cycles above), while tightening the intercept prior speeds escapes up — but the destination is always Ramsey.

99.8 Sims's nonconvergence

We can now resolve a puzzle noted in *The Rise and Fall of U.S. Inflation*.

The simulations of [Sims, 1988] and [Chung, 1990] start at the Nash self-confirming outcome, escape to low inflation, and then *stay there*, apparently indefinitely.

Those of [Sargent, 1999] and CWS instead escape and are then pulled back, again and again.

[Sargent and Williams, 2005] trace the difference to a single modeling choice: whether the government attributes the *right* amount of variance to its regression error.

When $\sigma = \sigma_1$ — as in a self-confirming equilibrium, where the regression (99.2) and the truth (99.1) coincide — the government correctly decomposes the variation it sees.

Sims instead used $\sigma \neq \sigma_1$ (and did not shrink the gain), which *misallocates* the observed variation and produces prolonged, perhaps permanent, departures from the self-confirming equilibrium.

The mechanism is visible in the Riccati equation (99.6) before we simulate anything.

A government that attributes *too little* variance to its regression error must attribute the variation it sees to something else, and the only thing else available is drift in its own coefficients.

So $\sigma < \sigma_1$ inflates the steady-state P, and P is what multiplies each forecast error in (99.5).

Understating the error variance therefore makes the government *learn faster* — which is precisely what weakens the pull of the mean dynamics.

```
def effective_gain(model, σ_govt, ε, λ=1.0):
    "Steady-state weight the filter puts on a single forecast error at the SCE."
    V = ε**2 * V_tighten_slope(λ, V_star)
    P = solve_riccati(V, M_sce, σ_govt)
    Φ = np.array([1.0, model.x(ɣ_sce)])
    return (P @ Φ)[0] / (σ_govt**2 + Φ @ P @ Φ)

ε_common = 0.0002
for σ_govt, tag in ((model.σ1, 'σ = σ1  (correct) '), (0.1, 'σ ≠ σ1  (Sims)    ')):
    print(f"{tag}: effective gain = {effective_gain(model, σ_govt, ε_common):.5f}")
```

```
σ = σ1  (correct) : effective gain = 0.00222
σ ≠ σ1  (Sims)    : effective gain = 0.05660
```

Understating the error variance multiplies the effective gain by more than twenty, even though the prior V and the scale ε are identical in the two runs.

Let's now simulate the static model under both specifications, holding ε fixed so that the *only* difference is the variance the government attributes to its own regression error.

```
def simulate(model, σ_govt, ε, λ=1.0, T=3000, seed=0):
    "Static Kalman-filter learning; σ_govt is the government's assumed error std."
    rng = np.random.default_rng(seed)
    u, σ1, σ2 = model.u, model.σ1, model.σ2

    V = ε**2 * V_tighten_slope(λ, V_star)
    ɣ = ɣ_sce.copy()
    P = ε * np.linalg.inv(M_sce)
    infl = np.empty(T)

    for n in range(T):
        x = model.x(ɣ)
        w1, w2 = rng.standard_normal(2)
        π = x + σ2 * w2
        U = u - σ2 * w2 + σ1 * w1                  # truth uses σ1
        Φ = np.array([1.0, π])
        denom = σ_govt**2 + Φ @ P @ Φ
        ɣ = ɣ + (P @ Φ) / denom * (U - Φ @ ɣ)
        P = P - np.outer(P @ Φ, Φ @ P) / denom + V / σ_govt**2
        infl[n] = π
    return infl

T_sim, seeds = 6000, range(10)
```

(continues on next page)

(continued from previous page)

```
def summarize(σ_govt):
    "Mean inflation and time near Ramsey, across seeds."
    paths = [simulate(model, σ_govt=σ_govt, ε=ε_common, T=T_sim, seed=s)
             for s in seeds]
    return (np.array([p.mean() for p in paths]),
            np.array([(p < 2).mean() for p in paths]))

mean_base, ramsey_base = summarize(model.σ1)
mean_sims, ramsey_sims = summarize(0.1)

print(f"{'seed':>4} | {'σ = σ1: mean π':>15} {'near Ramsey':>12}"
      f" | {'σ ≠ σ1: mean π':>15} {'near Ramsey':>12}")
for s in seeds:
    print(f"{s:>4} | {mean_base[s]:>15.2f} {ramsey_base[s]:>11.0%}"
          f" | {mean_sims[s]:>15.2f} {ramsey_sims[s]:>11.0%}")
print(f"\nnever escaped in {T_sim} periods:  "
      f"σ = σ1: {np.sum(ramsey_base < 0.01)}/{len(seeds)} paths,   "
      f"σ ≠ σ1: {np.sum(ramsey_sims < 0.01)}/{len(seeds)} paths")
```

```
seed |  σ = σ1: mean π  near Ramsey |  σ ≠ σ1: mean π  near Ramsey
   0 |            4.99          0% |            2.26         58%
   1 |            5.04          0% |            2.24         58%
   2 |            1.16         79% |            1.70         69%
   3 |            1.10         80% |            2.09         61%
   4 |            4.39         10% |            1.58         73%
   5 |            5.02          0% |            2.24         58%
   6 |            0.43         94% |            1.71         69%
   7 |            0.50         92% |            2.12         61%
   8 |            5.02          0% |            1.99         62%
   9 |            1.82         64% |            1.96         64%

never escaped in 6000 periods:  σ = σ1: 4/10 paths,   σ ≠ σ1: 0/10 paths
```

The two columns behave quite differently, and the difference is about *how often* the economy leaves Nash rather than about where it goes when it does.

With the correct error variance, an escape is a genuinely rare event: several of the sample paths sit at the Nash rate for six thousand periods without ever escaping at all, and the rest escape at some point and are held near Ramsey afterwards.

With Sims's misallocation, *every* path escapes, and every one of them spends most of its time near Ramsey.

Three seeds make the point without any cherry-picking: seed 0 never escapes, seed 4 escapes repeatedly and is pulled back each time, and seed 6 escapes and stays away for a long spell.

```
show = (0, 4, 6)

fig, axes = plt.subplots(2, 1, figsize=(9.5, 7), sharex=True, sharey=True)
for s in show:
    axes[0].plot(simulate(model, σ_govt=model.σ1, ε=ε_common, T=T_sim, seed=s),
                 lw=0.5, label=f'seed {s}')
    axes[1].plot(simulate(model, σ_govt=0.1, ε=ε_common, T=T_sim, seed=s),
                 lw=0.5, label=f'seed {s}')
for ax, title in zip(axes, [r'$\sigma = \sigma_1$ (correct): escapes are rare events',
                            r'$\sigma \neq \sigma_1$ (Sims): every path escapes and␣
↪lingers']):
    ax.axhline(model.u, color='k', ls='--', lw=1)
```

(continues on next page)

(continued from previous page)

```
    ax.axhline(0, color='C2', ls=':', lw=1)
    ax.set_ylabel('inflation')
    ax.set_title(title)
    ax.legend(frameon=False, fontsize=8, ncol=3, loc='lower right')
axes[1].set_xlabel('$n$')
plt.tight_layout()
plt.show()
```

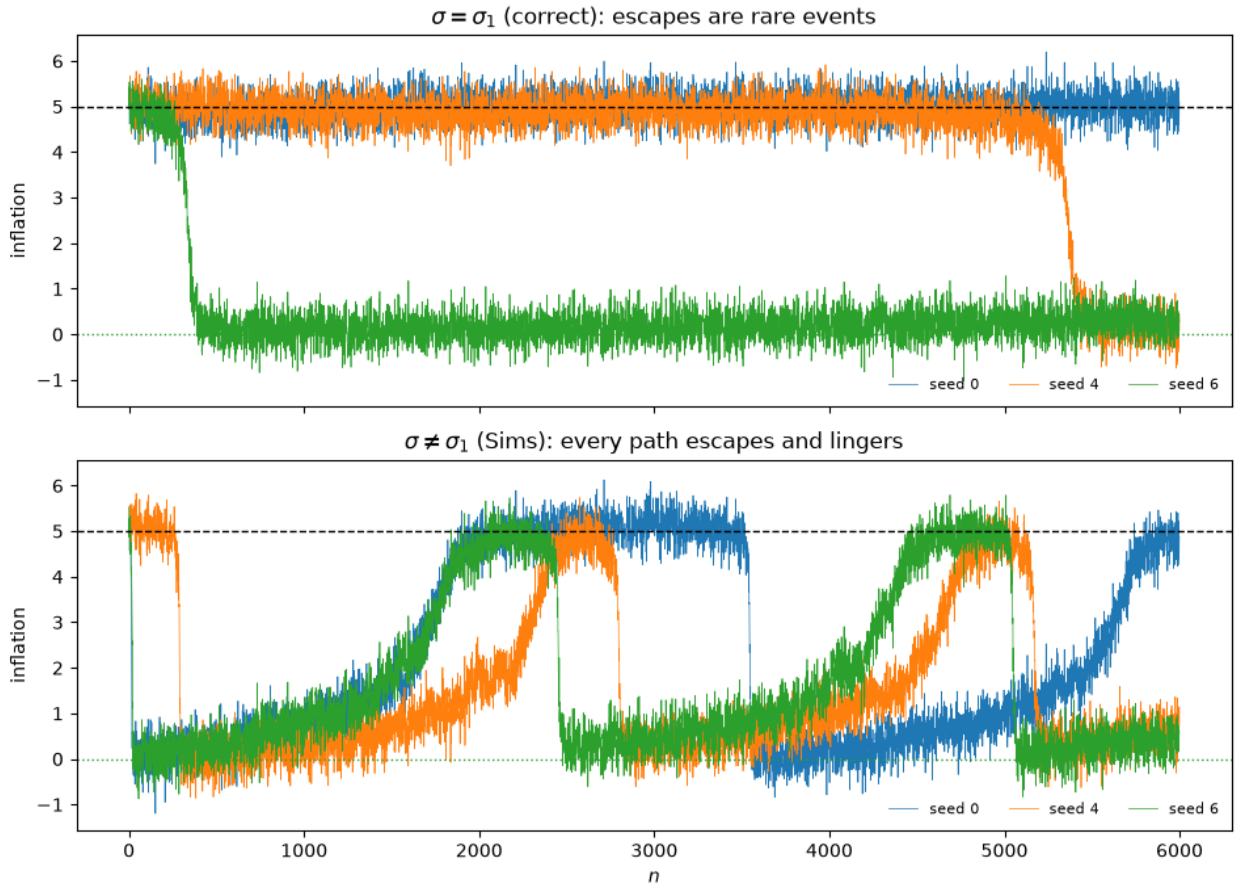


Fig. 99.4: Escapes are rare under the correct error variance and pervasive under Sims's misallocation

With the correct error variance the mean dynamics dominate: the economy sits at Nash, and only an unusual run of shocks dislodges it — the rare-escape picture of *Adaptive Learning and Escape Dynamics* and *Escaping Nash Inflation*.

With Sims's misallocation the government learns too fast for the mean dynamics to hold it, so it escapes almost immediately and behaves as if it had *permanently* learned a good-enough version of the natural-rate hypothesis.

As [Sargent and Williams, 2005] put it, one can read the difference in two equivalent ways: either Sims allowed too much parameter drift to permit convergence, or he did not let the government attribute enough variation to its regression error.

99.9 Conclusion

Free parameters are dangerous, as Arthur Goldberger and Robert Lucas warned.

This lecture's government carries free parameters in the covariance matrix V of the drift in its beliefs.

But those parameters buy something: the empirical sequel [Sargent *et al.*, 2006] shows that estimating V from post-war U.S. data lets the model reverse-engineer a sequence of policy makers' evolving subjective models of the Phillips curve that rationalize the actual rise and fall of American inflation — the vindication story of *The Rise and Fall of U.S. Inflation*, made quantitative.

Related evidence that the covariance of coefficient drift has itself moved over time appears in [Cogley and Sargent, 2005].

99.9.1 Are the estimated beliefs realistic?

That empirical success came with a challenge that is really a challenge about the *prior*.

To fit the data, [Sargent *et al.*, 2006] estimated a large drift covariance V — a government so open to new data that its beliefs about the monetary transmission mechanism lurch from month to month.

[Primiceri, 2006], Christopher Sims, and, candidly, [Sargent, 2008] in his presidential address, objected that such volatile beliefs are *unrealistic*: the imputed government forecasts unemployment poorly and holds views no real central bank would.

[Carboni and Ellison, 2009] answer the objection by disciplining the prior with data the Federal Reserve actually produced.

They re-estimate the model subject to the requirement that its beliefs reproduce the unemployment forecasts published in the Fed's *Greenbook* — imposing on the learning model a cross-equation restriction of the kind familiar from rational-expectations econometrics.

Imposing it shrinks the estimated V by orders of magnitude and removes the wild belief swings, yet leaves the low-frequency conquest story intact: a *stable* evolution of Federal Reserve beliefs still explains the rise and fall of inflation.

The volatile beliefs, it turns out, were doing nothing but overfitting the high-frequency wiggles that [Sargent, 1999] never set out to explain.

The moral for this lecture is direct: the prior V is not a nuisance parameter to be maximized over freely — it is an economic object, and pinning it down with independent evidence on what policy makers actually believed is what makes the vindication story credible.

The broader message is that *how* an adaptive government learns — the prior it brings to the drift in its own beliefs — is not a technical detail.

It determines whether the economy converges to Nash, cycles between Nash and Ramsey, or escapes to Ramsey and stays there.

The Lost Conquest: Fed Policy in the 2020s carries these same tools — constant-gain learning, an anticipated-utility Phelps problem, and a self-confirming equilibrium — into the present, to rationalize the Federal Reserve's response to the inflation of the 2020s.

Drifts and Volatilities then closes the suite by putting the whole account to an empirical test, fitting a drifting-coefficient, stochastic-volatility VAR to post-war data and asking how much of the Great Inflation was drifting beliefs and how much was simply bad luck.

99.10 Exercises

Exercise 99.10.1

The learning cycle above tightened the prior on the *slope* coefficient.

Tightening the prior on the *intercept* coefficient instead — using

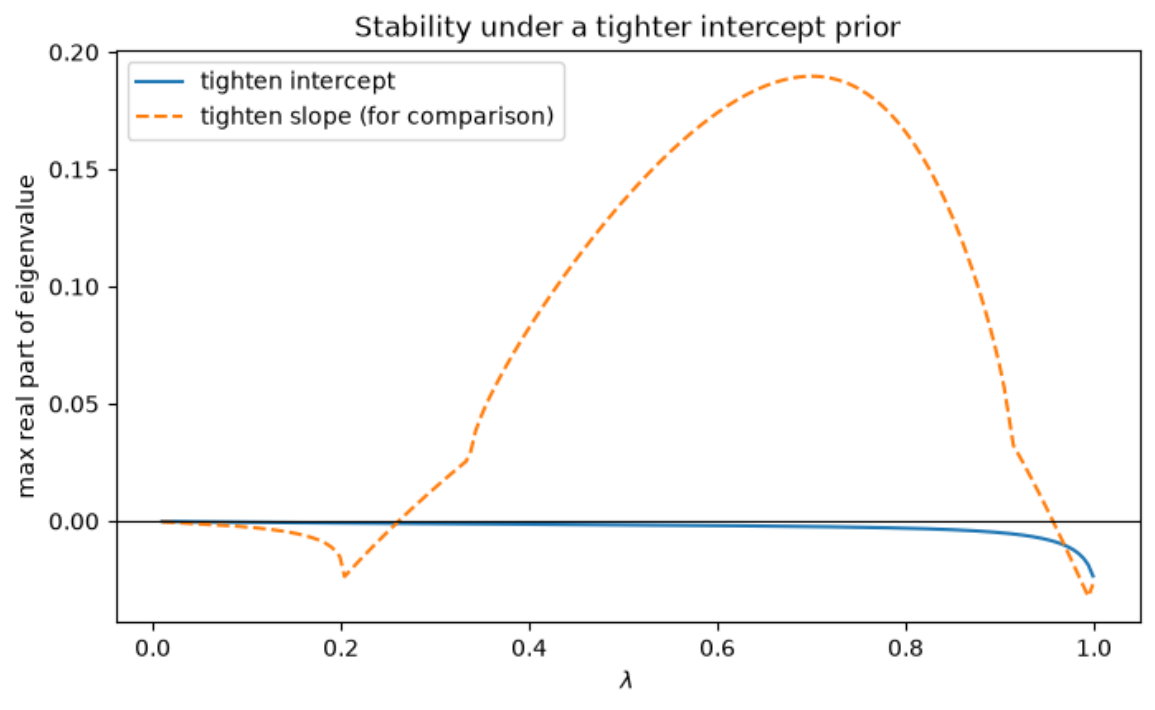
$$V(\lambda) = \begin{bmatrix} \lambda V_{11}^* & \sqrt{\lambda} V_{12}^* \\ \sqrt{\lambda} V_{12}^* & V_{22}^* \end{bmatrix}$$

— does *not* destabilize the self-confirming equilibrium.

Verify this by plotting the maximum real part of the eigenvalues of $\bar{P}(\lambda)\, \partial \bar{g}/\partial \gamma$ against λ for this intercept-tightening family, and confirm it stays negative.

Solution

```
def V_tighten_intercept(λ, V_star):
    V = V_star.copy()
    V[0, 0] = λ * V_star[0, 0]
    V[0, 1] = V[1, 0] = np.sqrt(λ) * V_star[0, 1]
    return V

max_re_int = []
for λ in λ_grid:
    V = ε**2 * V_tighten_intercept(λ, V_star)
    P = solve_riccati(V, M_sce, model.σ)
    max_re_int.append(np.linalg.eigvals(P @ J).real.max())

fig, ax = plt.subplots(figsize=(8, 4.5))
ax.plot(λ_grid, max_re_int, label='tighten intercept')
ax.plot(λ_grid, max_re, ls='--', label='tighten slope (for comparison)')
ax.axhline(0, color='k', lw=0.8)
ax.set_xlabel(r'$\lambda$')
ax.set_ylabel('max real part of eigenvalue')
ax.set_title('Stability under a tighter intercept prior')
ax.legend()
plt.show()
```



Tightening the intercept prior keeps the maximum real part negative for all λ: the self-confirming equilibrium stays stable, so there is no learning cycle.

As [Sargent and Williams, 2005] show, this prior does still change the *escape* dynamics — it speeds escapes up — but it does not overturn the mean dynamics.

Exercise 99.10.2

The instantaneous escape direction is the dominant eigenvector of the prior covariance $\hat{V}$.

Confirm that this direction is robust to the overall *scale* of the prior but sensitive to its *shape*.

Compute the escape direction (and the implied terminal beliefs) for the baseline prior V^* and for the slope-tightened prior $V(\lambda = 0.5)$, and compare where each sends the government's beliefs.

Solution

```
def escape_terminal(V):
    w, vecs = np.linalg.eigh(V)
    v = vecs[:, np.argmax(w)]
    v = v / np.linalg.norm(v)
    if v[1] < 0:                        # orient toward increasing slope
        v = -v
    return y_sce + (1.0 / v[1]) * v, v

for name, V in [("baseline V*", V_star),
                ("slope-tightened V(0.5)", V_tighten_slope(0.5, V_star))]:
    term, v = escape_terminal(V)
    print(f"{name:24s}: direction {v.round(3)}, terminal {term.round(2)}")
```

```
baseline V*             : direction [-0.981  0.195], terminal [4.98 0.  ]
slope-tightened V(0.5)  : direction [-0.99  0.14], terminal [2.9 0. ]
```

Both priors send beliefs toward a terminal point with slope 0 — the Ramsey belief — but along different directions, and the intercepts they arrive at differ considerably: the slope-tightened prior lands at an intercept a little over half the baseline's.

The destination is robust in the sense that matters for policy: slope 0 means the government perceives no exploitable tradeoff, and (99.3) then sets inflation to zero whatever the intercept.

The *route*, and where along the Ramsey line the escape terminates, depend on the shape of the prior.

CHAPTER

THE LOST CONQUEST: FED POLICY IN THE 2020S

Contents

- *The Lost Conquest: Fed Policy in the 2020s*
  - *Overview*
  - *The three elements*
  - *The Fed's drifting-coefficients beliefs*
  - *The Fed's Phelps problem*
  - *Why the Fed was slow: a counterfactual*
  - *A self-confirming equilibrium: nature's New Keynesian model*
  - *Exercises*

In addition to what's in Anaconda, this lecture will use the following library to download data from FRED:

```
!pip install pandas_datareader
```

100.1 Overview

This lecture is the contemporary sequel to the *Phillips curve tradeoffs* suite.

It follows [Sargent and Williams, 2025], which turns the tools of *Adaptive Learning and Escape Dynamics* and *Priors, Escapes, and Learning Cycles* — a *Phelps control problem*, an *anticipated-utility* government, a *drifting-coefficients* model estimated by *constant-gain recursive least squares*, and a *self-confirming equilibrium* — on the inflation of the 2020s.

The puzzle is a live one.

After the COVID-19 pandemic, U.S. inflation surged to its highest rate since the early 1980s — the very episode we plotted in the post-1999 data of *The Rise and Fall of U.S. Inflation*.

Yet for more than a year the Federal Reserve did not respond; only in 2022, with inflation already near its peak, did it begin to raise rates aggressively.

Why was the Fed so slow?

[Sargent and Williams, 2025] build an *artificial Fed* that, each period,

- re-estimates a drifting-coefficients Phillips curve by constant-gain recursive least squares, and
- solves a linear-quadratic Phelps problem — under an anticipated-utility assumption, in the sense of [Kreps, 1998] — to set its interest-rate instrument.

This is "model predictive control": the same *estimate-then-optimize* loop that drove the learning government of *Adaptive Learning and Escape Dynamics*, but now the instrument is the policy rate and the beliefs concern the *persistence* of inflation and the *slope* of the Phillips curve.

Three drifting beliefs turn out to rationalize the slow response:

1. **Declining inflation persistence** — the Fed had learned that inflation shocks fade quickly, so the surge looked *transitory*.
2. **A flatter Phillips curve** — the Fed had learned that inflation responds weakly to slack, so disinflation looked *costly*.
3. **Real-time output-gap mismeasurement** — the Fed perceived more economic slack than there really was.

We reproduce the first two from public data, show how they generate a Phelps rule that tracks the actual funds rate, and then — following the paper's New Keynesian appendix — ask a self-confirming-equilibrium question: *were the Fed's benign beliefs a consequence of its own past success?*

Let's import what we need:

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import datetime
from pandas_datareader import data as web
from scipy.linalg import solve_discrete_are
```

100.2 The three elements

The first two elements are among the most documented facts in modern macroeconomics.

Declining persistence. From the 1970s into the 1980s inflation was highly persistent; a shock raised inflation for years.

[Cogley and Sargent, 2005] and [Stock and Watson, 2007] document a marked decline in persistence after the mid-1980s — inflation began reverting to target much faster.

A flatter Phillips curve. Since the 1990s, estimates of the Phillips curve's slope have trended toward zero — the "missing disinflation" after the Great Recession being the leading example.

Both facts were on policy makers' minds.

Former Fed Chair Janet Yellen observed in 2019 that "the slope of the Phillips curve … has diminished very significantly since the 1960s … and … inflation has become much less persistent."

And, as [Bernanke, 2022] writes, "a flat Phillips curve means that inflation is a less reliable indicator of economic overheating [and] the costs, in terms of unemployment, of bringing inflation back down to target could be higher than in the past."

Real-time uncertainty. The third element, emphasized by [Orphanides, 2001], is that the output gap is badly mismeasured in *real time*, especially at business-cycle turning points.

Through 2020–2023 the real-time gap was persistently *below* the later-revised measure, so the Fed perceived more slack — reinforcing the belief that inflation would fade on its own.

We use current-vintage data below and return to the real-time distinction in the conclusion.

100.3 The Fed's drifting-coefficients beliefs

We give the artificial Fed a backward-looking Phillips curve with drifting coefficients,

$$\pi_t = \alpha_{0,t} + \rho_t\,\pi_{t-1} + \kappa_t\,x_t + \varepsilon^{\pi}_t, \tag{100.1}$$

where π_t is inflation, x_t the output gap, ρ_t the *perceived persistence*, and κ_t the *perceived slope*.

The Fed updates $\theta_t = (\alpha_{0,t}, \rho_t, \kappa_t)$ by constant-gain recursive least squares — exactly the algorithm of *Adaptive Learning and Escape Dynamics* and *Priors, Escapes, and Learning Cycles*, with gain γ discounting the past so the estimates can *track* drift:

$$\theta_{t+1} = \theta_t + \gamma R_t^{-1} X_t\,(\pi_t - X_t^{\top}\theta_t), \qquad R_{t+1} = R_t + \gamma\,(X_t X_t^{\top} - R_t),$$

with $X_t = (1, \pi_{t-1}, x_t)^{\top}$.

We download quarterly PCE inflation, the CBO output gap, and the federal funds rate from FRED.

```
start, end = datetime.datetime(1959, 1, 1), datetime.datetime(2025, 7, 1)

pcepi = web.DataReader('PCEPI', 'fred', start, end)['PCEPI'].resample('QS').mean()
gdp   = web.DataReader('GDPC1', 'fred', start, end)['GDPC1']          # real GDP
pot   = web.DataReader('GDPPOT', 'fred', start, end)['GDPPOT']        # CBO potential
ff    = web.DataReader('FEDFUNDS', 'fred', start, end)['FEDFUNDS'].resample('QS').
 ↪mean()

inflation = 100 * (pcepi / pcepi.shift(4) - 1)      # year-over-year PCE inflation
gap = 100 * (gdp / pot - 1)                         # output gap, percent

data = pd.concat([inflation.rename('pi'), gap.rename('x'),
                  ff.rename('i')], axis=1).dropna()
print(f"sample: {data.index[0].date()} to {data.index[-1].date()}, "
      f"{len(data)} quarters")
```

```
sample: 1960-01-01 to 2025-07-01, 263 quarters
```

Following the paper, we freeze beliefs during 2020–2021 (setting the gain to zero) because the pandemic observations are extreme outliers that would otherwise whipsaw the estimates; belief updating resumes in 2022.

```
def estimate_beliefs(data, gain=0.03, freeze=(2020, 2021)):
    "Constant-gain RLS of the drifting Phillips curve; returns α₀, ρ, κ paths."
    pi, x = data['pi'].values, data['x'].values
    θ = np.array([0.5, 0.9, 0.05])            # [intercept, persistence, slope]
    R = np.diag([1.0, 10.0, 5.0])
    rows = []
    for t in range(1, len(data)):
        g = 0.0 if data.index[t].year in freeze else gain
        X = np.array([1.0, pi[t - 1], x[t]])
        err = pi[t] - X @ θ
        R = R + g * (np.outer(X, X) - R)
        θ = θ + g * np.linalg.solve(R, X * err)
        rows.append((θ[0], θ[1], θ[2]))
    return pd.DataFrame(rows, index=data.index[1:],
                        columns=['alpha0', 'rho', 'kappa'])

beliefs = estimate_beliefs(data)
```

```
fig, axes = plt.subplots(2, 1, figsize=(10, 6), sharex=True)
axes[0].plot(beliefs['rho'])
axes[0].axhline(1, color='k', lw=0.5, ls=':')
axes[0].set_ylabel('persistence $\\rho_t$')
axes[0].set_title('Perceived inflation persistence')

axes[1].plot(beliefs['kappa'], color='C1')
axes[1].axhline(0, color='k', lw=0.5, ls=':')
axes[1].set_ylabel('slope $\\kappa_t$')
axes[1].set_xlabel('year')
axes[1].set_title('Perceived Phillips-curve slope')

plt.tight_layout()
plt.show()
```

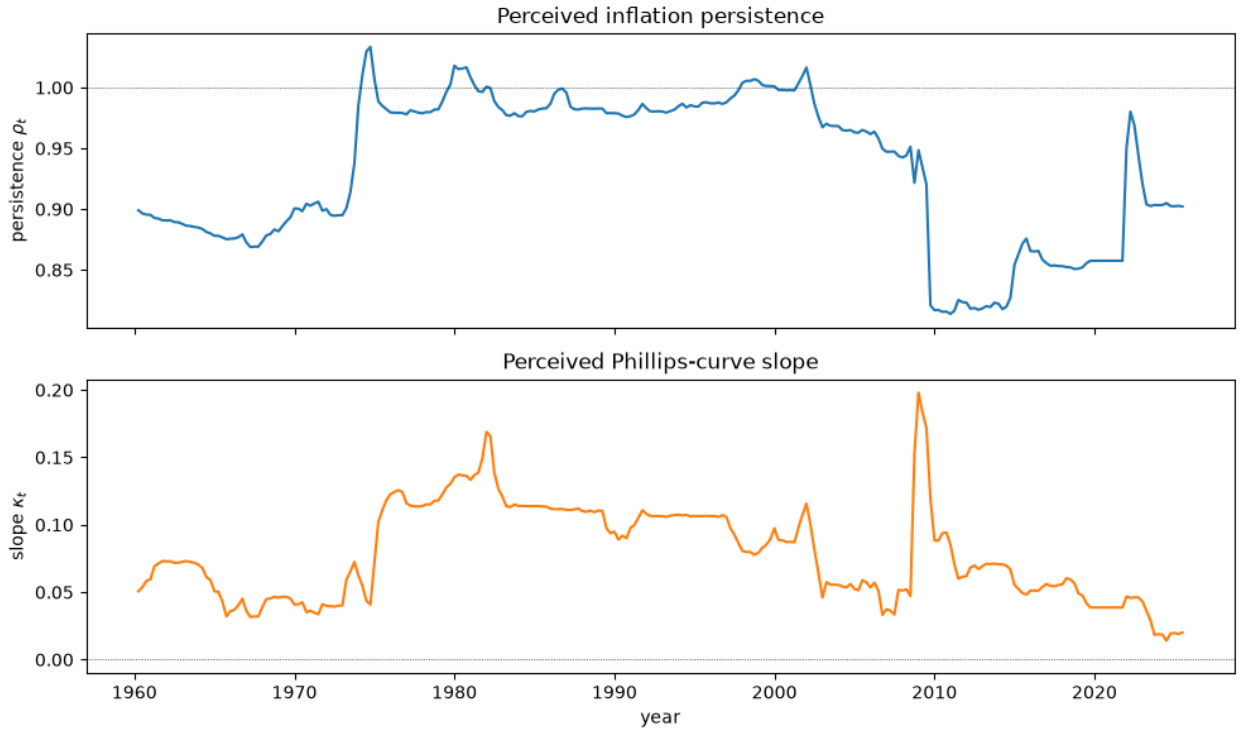


Fig. 100.1: The Fed's perceived inflation persistence and Phillips curve slope

Perceived **persistence** ρ_t is near one through the high-inflation 1970s and 1980s, then drifts down after the mid-1980s, reaching a post-2008 trough — and then *jumps back up* toward one when belief updating resumes in 2022, just as the Fed abandoned the "transitory" characterization and began to tighten.

Perceived **slope** κ_t trends toward zero over the 2010s: the Phillips curve flattens.

By 2019 the Fed's model said inflation was *not persistent* and only *weakly linked to slack* — the beliefs that, fed into a Phelps problem, will counsel patience.

100.4 The Fed's Phelps problem

Each period the Fed sets its policy rate by solving a linear-quadratic Phelps problem, taking its *current* estimates as if they will hold forever — the anticipated-utility assumption of [Kreps, 1998] that we met in *Adaptive Learning and Escape Dynamics*.

Pairing the belief Phillips curve (100.1) with a fixed "IS curve" $x_t = b_0 + b_1 x_{t-1} + g(i_{t-1} - \pi_{t-1}) + \varepsilon_t^x$ gives linear state dynamics for $X_t = (1, \pi_t, x_t, i_{t-1})^\top$,

$$X_{t+1} = A_t X_t + B_t\, i_t + C\varepsilon_{t+1},$$

whose matrices depend on the time-t beliefs.

The Fed minimizes

$$\mathbb{E}_t \sum_{s=t}^{\infty} \beta^{s-t}\Big[(\pi_s - \pi^*)^2 + \lambda_x\, x_s^2 + \eta\,(i_s - i_{s-1})^2\Big], \tag{100.2}$$

where π^* is the 2% target and the last term penalizes abrupt rate changes.

This is a discounted linear-quadratic regulator with a cross-term; its solution is a *smoothed Taylor rule* $i_t = -F_t X_t$ whose coefficients move as beliefs move.

```
# fixed IS curve, estimated once by OLS over the full sample
pi, x, i_ = data['pi'].values, data['x'].values, data['i'].values
n = len(data)
X_is = np.column_stack([np.ones(n - 1), x[:-1], i_[:-1] - pi[:-1]])
b0, b1, g = np.linalg.lstsq(X_is, x[1:], rcond=None)[0]

β, π_star, λ_x, η = 0.95, 2.0, 0.2, 0.5

def phelps_rate(θ, state, η=η):
    "Optimal (subjectively) funds rate given beliefs θ=(α₀,ρ,κ) and state."
    α0, ρ, κ = θ
    A = np.array([[1, 0, 0, 0],
                  [α0 + κ * b0, ρ - κ * g, κ * b1, 0],
                  [b0, -g, b1, 0],
                  [0, 0, 0, 0]], float)
    B = np.array([[0], [κ * g], [g], [1]], float)
    c_π = np.array([-π_star, 1, 0, 0.])       # π - π*
    c_x = np.array([0, 0, 1, 0.])             # x
    e_i = np.array([0, 0, 0, 1.])             # i_{-1}
    Q = np.outer(c_π, c_π) + λ_x * np.outer(c_x, c_x) + η * np.outer(e_i, e_i)
    N = (-η * e_i).reshape(4, 1)
    R = np.array([[η]])
    sb = np.sqrt(β)
    P = solve_discrete_are(sb * A, sb * B, Q, R, s=sb * N)
    F = np.linalg.solve(R + β * B.T @ P @ B, β * B.T @ P @ A + N.T)
    return max((-F @ state).item(), 0.0)      # impose the zero lower bound
```

```
θ = np.array([0.5, 0.9, 0.05])
R = np.diag([1.0, 10.0, 5.0])
gain = 0.03
opt, θ_2000 = [], None
for t in range(1, n):
    yr = data.index[t].year
    g_t = 0.0 if yr in (2020, 2021) else gain
```

(continues on next page)

(continued from previous page)

```
    X = np.array([1.0, pi[t - 1], x[t]])
    R = R + g_t * (np.outer(X, X) - R)
    θ = θ + g_t * np.linalg.solve(R, X * (pi[t] - X @ θ))
    if data.index[t].year == 2000 and θ_2000 is None:
        θ_2000 = θ.copy()                    # save beliefs for the counterfactual
    state = np.array([1.0, pi[t], x[t], i_[t - 1]])
    opt.append(phelps_rate(θ, state))

optimal = pd.Series(opt, index=data.index[1:])
```

```
fig, ax = plt.subplots(figsize=(10, 4.5))
window = slice('1991', None)
ax.plot(optimal[window], 'C0', label="Phelps problem's recommended rate")
ax.plot(data['i'][window], 'C3', lw=1, label='actual federal funds rate')
ax.set_xlabel('year')
ax.set_ylabel('percent')
ax.legend()
plt.show()

corr = np.corrcoef(optimal['1991':], data['i']['1991':optimal.index[-1]])[0, 1]
print(f"correlation of recommended and actual rate, 1991-2025: {corr:.2f}")
```

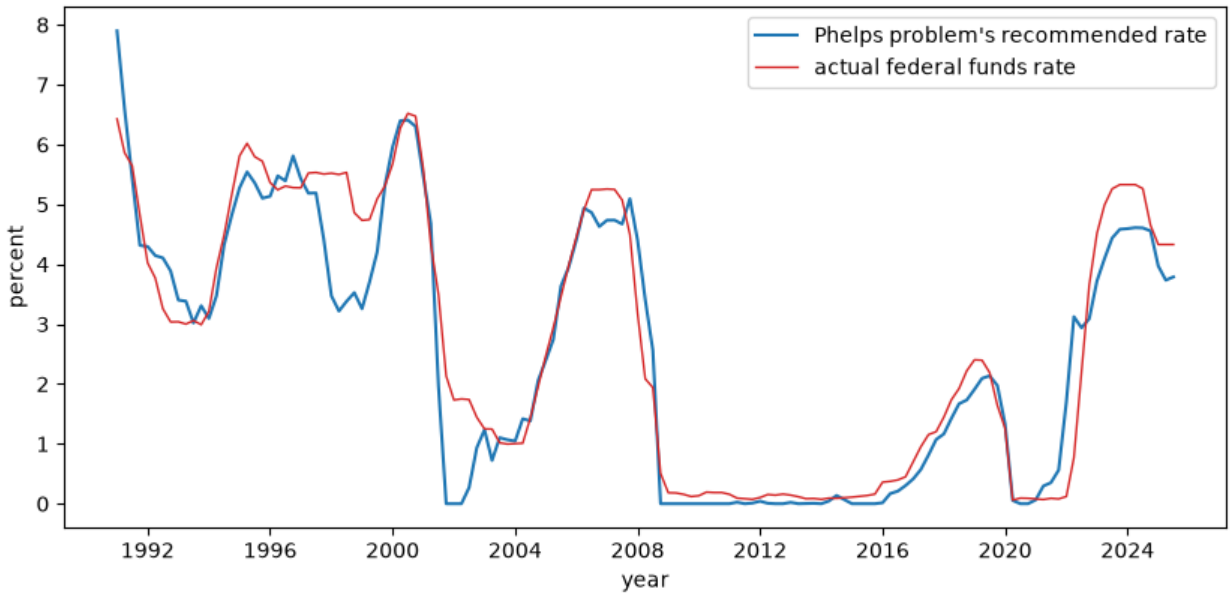


Fig. 100.2: The belief-driven Phelps rule against the actual federal funds rate

```
correlation of recommended and actual rate, 1991-2025: 0.95
```

The subjectively optimal rate tracks the level and turning points of actual policy over three decades (correlation about 0.95): the late-1990s tightening, the 2001 easing, the zero-bound era after 2008, and the 2015 normalization.

Crucially, around the 2021 surge the recommended rate barely moves — the belief-driven rule *also* counsels a slow response, and only tightens in 2022, mirroring the Fed.

100.5 Why the Fed was slow: a counterfactual

To isolate the role of the drifting beliefs, we recompute the Phelps recommendations holding beliefs *fixed* at their January 2000 values — when inflation was still perceived as persistent and the Phillips curve as steeper.

```
counterfactual = pd.Series(
    [phelps_rate(θ_2000, np.array([1.0, pi[t], x[t], i_[t - 1]]))
     for t in range(1, n)],
    index=data.index[1:])

fig, ax = plt.subplots(figsize=(10, 4.5))
w = slice('2015', None)
ax.plot(optimal[w], 'C0', label='baseline (drifting beliefs)')
ax.plot(counterfactual[w], 'C1--', label='counterfactual (beliefs frozen at 2000)')
ax.plot(data['i'][w], 'C3', lw=1, alpha=0.7, label='actual funds rate')
ax.set_xlabel('year')
ax.set_ylabel('percent')
ax.legend()
plt.show()
```

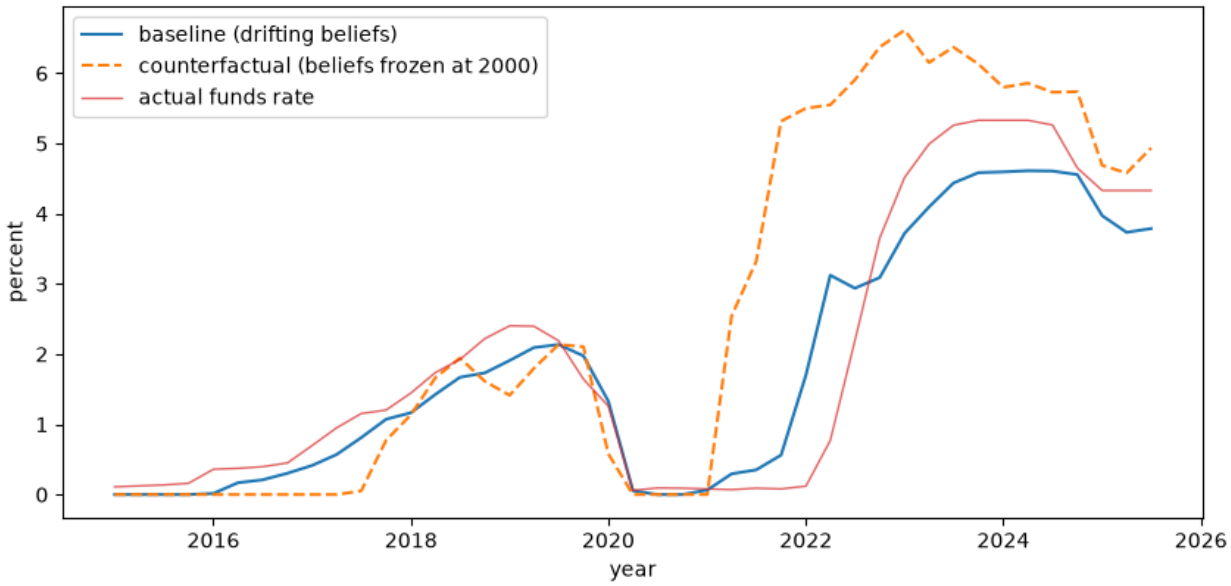


Fig. 100.3: Counterfactual: a Fed that had not updated its beliefs since 2000

A Fed with year-2000 beliefs — perceiving persistent inflation and a steeper Phillips curve — would have tightened *immediately and sharply* in 2021, driving the funds rate well above 4% before the actual Fed had moved at all.

The muted, delayed response was not a change in objectives; it was a change in *beliefs*.

Perceiving inflation as transitory and the Phillips curve as flat, the Fed's own Phelps problem told it to wait.

100.6 A self-confirming equilibrium: nature's New Keynesian model

Were those benign beliefs *correct*?

Here the paper adds a twist that connects directly to *Self-Confirming Equilibria* and *Escaping Nash Inflation*: a **self-confirming equilibrium** in which the Fed's flat-Phillips-curve, low-persistence beliefs are a *consequence of its own aggressive past policy*.

Suppose nature actually runs a small New Keynesian model,

$$\begin{aligned} \pi_t &= \beta\,\mathbb{E}_t\pi_{t+1} + \gamma_b\,\pi_{t-1} + \kappa\,x_t + u_t, \\ x_t &= \mathbb{E}_t x_{t+1} - \sigma\left(i_t - \mathbb{E}_t\pi_{t+1} - r_t^n\right), \\ i_t &= \phi_\pi\,\pi_t, \end{aligned} \tag{100.3}$$

with a structural slope κ and a Taylor rule of aggressiveness ϕ_π.

In its minimum-state-variable rational expectations equilibrium, $\mathbb{E}_t\pi_{t+1} = \lambda\,\pi_t$, where λ — the *measured* persistence an econometrician would recover — is the stable root of a cubic that depends on ϕ_π.

The paper proves two comparative statics.

Proposition 100.6.1 (Aggressive policy lowers measured persistence)

In the determinacy region, the stable root $\lambda(\phi_\pi)$ is strictly decreasing in the policy aggressiveness ϕ_π: a more aggressive Fed makes inflation *look* less persistent.

Proposition 100.6.2 (Aggressive policy flattens the measured slope)

Under sufficient conditions (a small lagged-inflation term and a sufficiently aggressive rule), the population OLS slope of a backward-looking Phillips curve is also decreasing in ϕ_π: a more aggressive Fed makes the Phillips curve *look* flatter.

Let's reproduce *Proposition 100.6.1* by solving the cubic for the stable root.

```
def measured_persistence(ψ_π, β=0.99, γ_b=0.5, κ=0.1, σ=1.0):
    "Stable MSV root λ(ψ_π): the persistence an econometrician would measure."
    coeffs = [β, -(1 + β + κ * σ), 1 + γ_b + κ * σ * ψ_π, -γ_b]
    roots = np.roots(coeffs)
    real = roots[np.abs(roots.imag) < 1e-9].real
    stable = real[np.abs(real) < 1.0]
    return stable[np.argmin(np.abs(stable))]

ψ_grid = np.linspace(1.05, 3.0, 40)
λ_path = [measured_persistence(ψ) for ψ in ψ_grid]

fig, ax = plt.subplots(figsize=(8, 4.5))
ax.plot(ψ_grid, λ_path, lw=2)
ax.set_xlabel(r'Taylor-rule aggressiveness $\phi_\pi$')
ax.set_ylabel(r'measured persistence $\lambda$')
plt.show()
```

As policy grows more aggressive, the measured persistence falls from near one toward one-half.

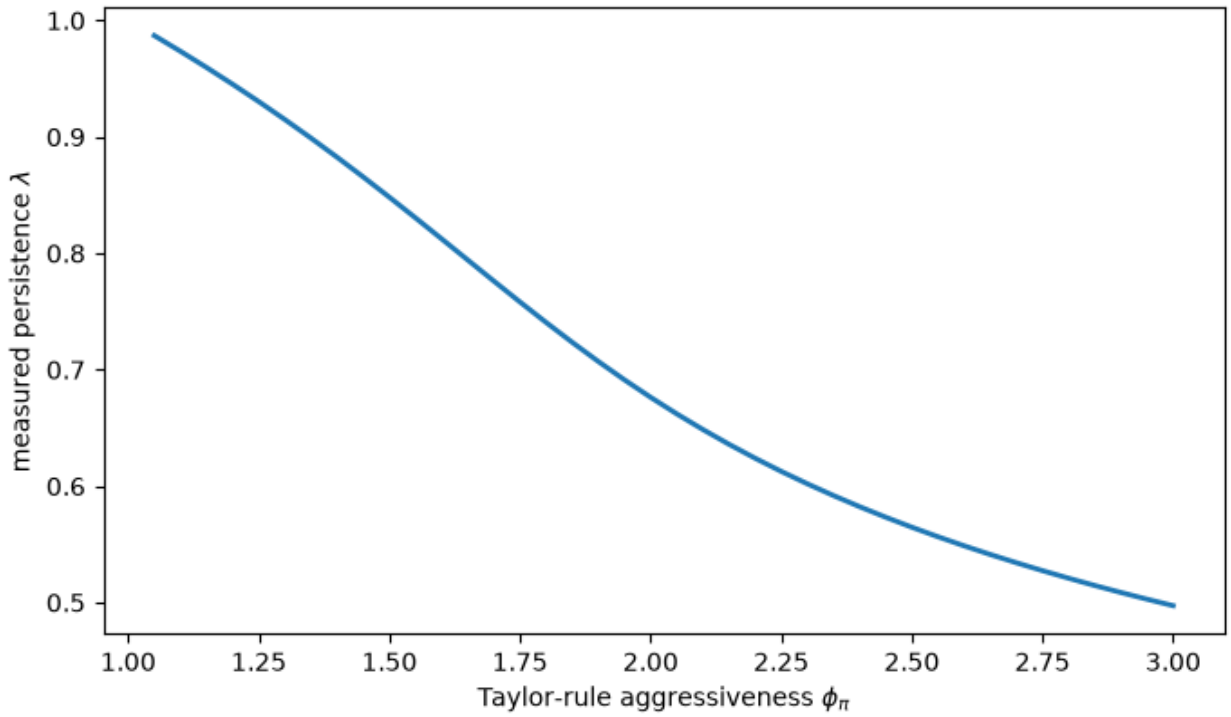


Fig. 100.4: Aggressive policy makes inflation look less persistent

The intuition behind both propositions is *policy endogeneity*, the point emphasized by [McLeay and Tenreyro, 2019]: when the Fed offsets inflationary pressure promptly, a regression that treats the output gap as an exogenous driver recovers a *smaller* slope and a *faster-mean-reverting* inflation process than the structural parameters imply.

100.6.1 The trap

Now read (100.3) as nature and the drifting-coefficients model of the previous sections as the Fed's *approximating* model.

Along a path of consistently aggressive policy — the Volcker–Greenspan conquest — the data the Fed generates make inflation *look* transitory and the Phillips curve *look* flat.

Estimating its backward-looking model, the Fed comes to believe exactly that.

Those beliefs are a **self-confirming equilibrium** in the sense of *Self-Confirming Equilibria*: under the Fed's *current* aggressive policy, its reduced-form model is observationally equivalent to nature's New Keynesian model.

But the beliefs are *wrong off the equilibrium path* — under a *less* aggressive policy, persistence and the slope would spring back up.

The Fed cannot see this, because it has no reason to experiment: it sits in the **lack-of-experimentation trap** of *Adaptive Learning and Escape Dynamics*, its complacency justified by a belief about a policy it never runs.

When large post-pandemic shocks finally hit, those self-confirming beliefs told the Fed the surge was transitory and tightening was costly — and the conquest was, for a while, *lost*.

This is a modern replay of the *Conquest*'s recurrent dynamics, one level up: the mechanism now works through the perceived slope and persistence of the Phillips curve and the policy-rate instrument, and the misspecification is not about expectations but about the *policy endogeneity* of the reduced-form Phillips curve that the Fed treats as structural — ignoring the Lucas Critique in just the way the "vindication" story of *The Rise and Fall of U.S. Inflation* describes.

Note

As [Sargent and Williams, 2025] note, the drifting-coefficients model is a purely descriptive "Kepler stage" model, not a structural "Newton stage" one.

The paper also acknowledges an alternative reading in which the 2020s accommodation was fiscal in origin — see the fiscal-theory accounts it cites — a very different rationalization of the same policy path.

This lecture imputed drifting beliefs to the Fed and then asked what policy they would recommend.

The closing lecture of the suite, *Drifts and Volatilities*, comes at the same period from the opposite direction: it asks the data whether the *reduced form* of the economy drifted at all, and how much of what looks like drifting beliefs is really drifting shock variances.

100.7 Exercises

Exercise 100.7.1

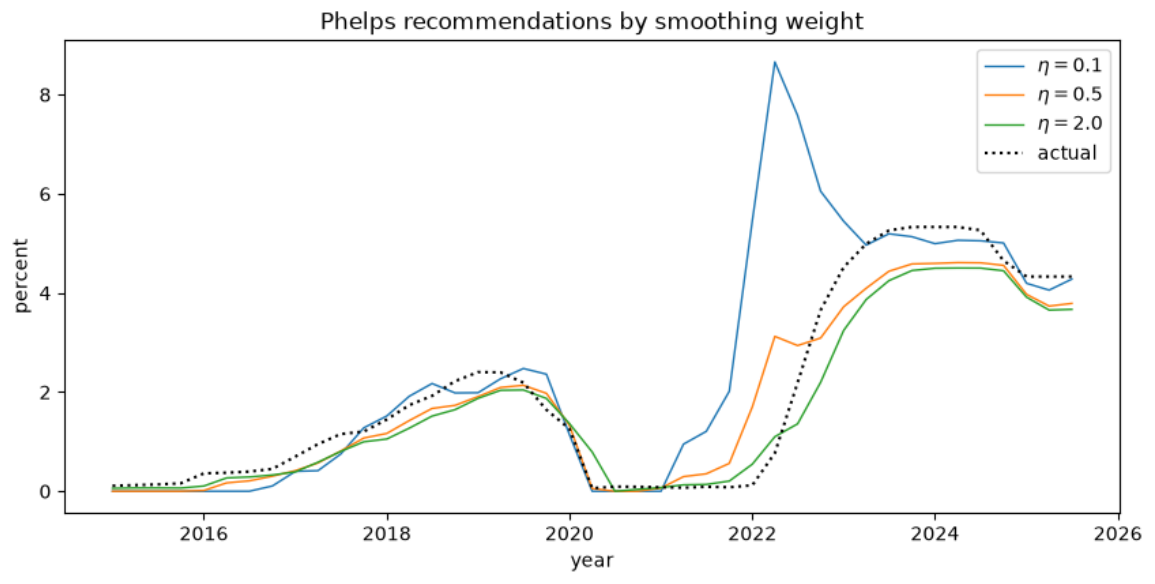
The interest-smoothing weight η in the loss (100.2) is, in the paper's words, "one of the most important parameters."

Recompute the baseline Phelps recommendations for $\eta \in \{0.1, 0.5, 2.0\}$ and plot them against the actual funds rate over 2015-2025.

How does a larger smoothing penalty change the character of the recommended policy?

Solution

```
def recommend(η_val):
    "Phelps recommendations along the whole sample, for a given smoothing weight."
    θ, R = np.array([0.5, 0.9, 0.05]), np.diag([1.0, 10.0, 5.0])
    out = []
    for t in range(1, n):
        g_t = 0.0 if data.index[t].year in (2020, 2021) else gain
        X = np.array([1.0, pi[t - 1], x[t]])
        R = R + g_t * (np.outer(X, X) - R)
        θ = θ + g_t * np.linalg.solve(R, X * (pi[t] - X @ θ))
        out.append(phelps_rate(θ, np.array([1.0, pi[t], x[t], i_[t - 1]]),
                               η=η_val))
    return pd.Series(out, index=data.index[1:])

fig, ax = plt.subplots(figsize=(10, 4.5))
w = slice('2015', None)
for η_val in [0.1, 0.5, 2.0]:
    ax.plot(recommend(η_val)[w], lw=1, label=rf'$\eta = {η_val}$')
ax.plot(data['i'][w], 'k:', lw=1.5, label='actual')
ax.set_xlabel('year')
ax.set_ylabel('percent')
ax.set_title('Phelps recommendations by smoothing weight')
ax.legend()
plt.show()
```



A small η produces a volatile rate that jumps sharply with inflation and the gap; a large η makes the recommendation hug the lagged rate, fitting the smooth observed path better but at the cost of interpretability.

The intermediate value strikes a balance — enough smoothing to resemble real Fed behavior, but still a recognizable response to the state of the economy.

Exercise 100.7.2

Proposition 100.6.1 says an aggressive Taylor rule lowers the *measured* persistence of inflation.

Trace out the implication for the Fed's own policy by combining the two halves of this lecture: for a grid of structural aggressiveness ϕ_π, compute the measured persistence $\lambda(\phi_\pi)$, then feed a belief with that persistence (holding the slope and intercept fixed) into `phelps_rate` at a representative state, and report the implied inflation response.

Does a Fed that has historically been *more* aggressive end up *less* willing to respond to a fresh inflation shock?

Solution

```
state = np.array([1.0, 4.0, 1.0, 2.0])          # π=4, x=1, i_{-1}=2: an inflation␣
↪shock

print(f"{'ψ_π (history)':>14} {'measured ρ':>12} {'recommended i':>15}")
for ψ in [1.2, 1.6, 2.0, 2.6]:
    ρ_meas = measured_persistence(ψ)
    i_rec = phelps_rate(np.array([0.3, ρ_meas, 0.05]), state)
    print(f"{ψ:>14} {ρ_meas:>12.2f} {i_rec:>15.2f}")
```

```
ψ_π (history)    measured ρ    recommended i
          1.2          0.95             3.38
          1.6          0.81             1.85
          2.0          0.68             1.70
          2.6          0.55             1.67
```

A history of more aggressive policy (higher ϕ_π) leaves the Fed believing inflation is less persistent (lower measured ρ), and — by the logic of *Proposition 100.6.1* together with the comparative statics of the Phelps problem — a Fed that thinks inflation will fade on its own responds *less* to a fresh shock.

Success breeds complacency: the very aggressiveness that conquered inflation teaches the Fed a lesson that, taken as structural, disarms it against the next surge.

CHAPTER

ONE

DRIFTS AND VOLATILITIES

Contents

- *Drifts and Volatilities*
  - *Overview*
  - *Bad policy or bad luck?*
  - *A VAR with drifting coefficients and stochastic volatility*
  - *The data*
  - *Priors*
  - *A Metropolis-within-Gibbs sampler*
  - *What the data say*
  - *Another quarter-century of evidence*
  - *Bad policy or bad luck? A verdict*
  - *Exercises*

101.1 Overview

The lectures in this section have told a story about how a government's *model* of the Phillips curve, and the *policy* it induces, can drift over time.

In *Adaptive Learning and Escape Dynamics* and *Escaping Nash Inflation* a government that fits and refits an approximating Phillips curve is repeatedly pushed away from a bad *self-confirming equilibrium* along an *escape route*, while *Priors, Escapes, and Learning Cycles* and *The Lost Conquest: Fed Policy in the 2020s* use drifting beliefs to interpret the rise and fall of American inflation.

Those lectures were mostly about *theory*.

This lecture turns to the *data*.

It studies Cogley and Sargent [2005], which asks a deceptively simple question:

> When we look at postwar U.S. time series on inflation, unemployment, and interest rates, do we see evidence that the dynamics have *drifted*?

Tim Cogley and Thomas Sargent began this work as an empirical companion to the *Conquest* book [Sargent, 1999] and the escape-route papers [Cho *et al.*, 2002].

It is also a response to searching comments by Sims [2001] and Stock [2001] on an earlier paper [Cogley and Sargent, 2001], and it grew into a friendly debate with Sims and Zha [2006] and Bernanke and Mihov [1998] about a question that organizes this whole section:

Was the Great Inflation of the 1970s and its conquest in the 1980s a story of bad policy, or of bad luck?

To let the data speak to that question we need a statistical model flexible enough to accommodate *both* answers.

That model is a *Bayesian vector autoregression whose coefficients drift as random walks and whose shock variances evolve as stochastic volatilities*.

Fitting it requires a Markov chain Monte Carlo algorithm that combines the *Kalman filter*, the forward-filter/backward-sample smoother of Carter and Kohn [1994], and the stochastic-volatility sampler of Jacquier *et al.* [1994].

Readers who want background will find companion-form vector autoregressions in *VARs and DMDs*, the Kalman smoother in *Another Look at the Kalman Filter*, and Bayesian inference for state-space models by MCMC in *Posterior Distributions for AR(1) Parameters* and *Forecasting an AR(1) Process*.

We work through the data transformation, prior, sampler, and main empirical results.

Let's start with some imports and the URL for the data.

```
import time

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from IPython.display import display, Math
from scipy import linalg
from scipy.special import expit
from scipy.stats import invwishart


data_url = (
    'https://raw.githubusercontent.com/QuantEcon/lecture-python.myst/'
    'main/lectures/_static/lecture_specific/phillips_drifts_volatilities/'
    'NEWQDATA.csv'
)
```

101.2 Bad policy or bad luck?

Two respectable views compete to explain the American Great Inflation — the same two stories, triumph versus vindication, that open *The Rise and Fall of U.S. Inflation*.

The **bad policy** view is the one dramatized throughout this section and in the *Conquest* book [Sargent, 1999].

Something about Arthur Burns's *model* of the economy, his *patience*, or his inability to *commit* to a better rule led the Federal Reserve to administer monetary policy in a way that produced the greatest peacetime inflation in U.S. history, while an improved model, more patience, or greater discipline led Paul Volcker to conquer it [DeLong, 1997, Taylor, 1997].

On this view, what changed between the 1970s and the 1980s was the *systematic part* of policy, namely the way the Fed's interest-rate setting responded to inflation and unemployment.

The **bad luck** view says something quite different.

What distinguished the Burns and Volcker eras was not their models or policies, but the *shocks* that hit the economy.

On this view the *coefficients* of a reduced-form description of the economy were essentially constant, and what changed was the *size* of the disturbances, namely the *volatility*.

Bernanke and Mihov [1998] and Sims and Zha [2006] marshaled evidence for this second view, in part by applying classical tests that *failed to reject* the hypothesis that VAR coefficients were time invariant.

How can we discriminate?

A constant-coefficient, constant-volatility VAR can generate unusually large realized shocks, but it cannot represent systematic changes in their variance over time.

A model with drifting coefficients but constant volatility can also mistake changing volatility for coefficient drift.

So Cogley and Sargent build a model that has room for *both* channels at once, and they let a Bayesian posterior sort out how much of each the data call for.

101.3 A VAR with drifting coefficients and stochastic volatility

Let the variables be ordered as nominal interest, transformed unemployment, and inflation,

$$y_t = \begin{bmatrix} i_t & u_t & \pi_t \end{bmatrix}^\top .$$

(Here u_t is not the raw unemployment rate but its logit, we define this transformation in the data section below.)

The measurement equation is a VAR with two lags and date-specific coefficients,

$$y_t = X_t^\top \theta_t + \varepsilon_t, \qquad X_t^\top = I_3 \otimes \begin{bmatrix} 1 & y_{t-1}^\top & y_{t-2}^\top \end{bmatrix}. \tag{101.1}$$

Each equation has an intercept and six lag coefficients, so θ_t contains $3(1 + 2 \times 3) = 21$ elements.

A two-lag VAR can be rewritten as a one-lag system by stacking y_t and y_{t-1} into a single vector; the matrix that multiplies this stacked vector is the **companion matrix**, and the rewritten system is the VAR in **companion form**.

The following function builds the companion matrix from the stacked coefficients.

```
n_variables = 3
n_lags = 2
n_regressors = 1 + n_variables * n_lags
n_coefficients = n_variables * n_regressors


def companion_matrix(θ):
    """Return the intercept and companion matrix for one coefficient vector."""
    equation_rows = np.asarray(θ, dtype=float).reshape(
        n_variables, n_regressors
    )
    intercept = np.r_[equation_rows[:, 0], np.zeros(n_variables)]
    companion = np.zeros((n_variables * n_lags, n_variables * n_lags))
    companion[:n_variables] = equation_rows[:, 1:]
    companion[n_variables:, :n_variables] = np.eye(n_variables)
    return intercept, companion


def design_matrix(regressors):
    """Return the observation matrix X_t prime for one date."""
    return np.kron(np.eye(n_variables), np.asarray(regressors, dtype=float))
```

The coefficient vector follows a driftless random walk,

$$\theta_t = \theta_{t-1} + v_t, \qquad v_t \sim N(0, Q). \tag{101.2}$$

A prior over how fast coefficients drift plays a mirror-image role in *Priors, Escapes, and Learning Cycles*: there it is the *government's* prior about a drifting Phillips curve that shapes the policy it chooses, whereas here it is the *econometrician's* prior in a posterior about drifting reduced-form dynamics.

The companion system is stable when every companion-matrix eigenvalue lies strictly inside the unit circle.

For an AR(1), this is $|\rho| < 1$; equivalently, the zero of $1 - \rho z$ lies outside the unit circle.

Cogley and Sargent rule out explosive paths by retaining a path only when the companion matrix is stable at every date and using the truncated prior

$$p(\theta^T, Q) \propto I(\theta^T) f(\theta^T \mid Q) f(Q), \tag{101.3}$$

where $I(\theta^T) = 1$ denotes a stable path.

This restriction encodes the belief that the economy did not in fact follow an explosive path.

This restriction also tilts the marginal prior for Q toward values that are less likely to generate explosive coefficient paths.

The code below applies the stability restriction to an entire trajectory

```
def companion_roots(θ_path):
    """Return all companion roots along a path with shape (21, T)."""
    θ_path = np.asarray(θ_path, dtype=float)
    if θ_path.ndim == 1:
        θ_path = θ_path[:, None]
    companions = np.stack(
        [
            companion_matrix(θ_path[:, t])[1]
            for t in range(θ_path.shape[1])
        ]
    )
    return np.linalg.eigvals(companions)


def is_stable(θ_path):
    """Test whether every companion root is strictly inside the unit circle."""
    return bool(np.max(np.abs(companion_roots(θ_path))) < 1)
```

The reduced-form innovation covariance changes over time according to

$$\varepsilon_t = R_t^{1/2} \xi_t, \qquad \xi_t \sim N(0, I_3), \qquad R_t = B^{-1} H_t B^{-1\prime}, \tag{101.4}$$

where

$$B = \begin{bmatrix} 1 & 0 & 0 \\ \beta_{21} & 1 & 0 \\ \beta_{31} & \beta_{32} & 1 \end{bmatrix}, \qquad H_t = \operatorname{diag}(h_{1t}, h_{2t}, h_{3t}).$$

The diagonal elements h_{it} let the size of each orthogonalized shock wax and wane over time.

The next two functions construct the triangular factor and the reduced-form innovation covariance.

```
def b_matrix(β):
    """Construct B from β_21, β_31, and β_32."""
    matrix = np.eye(n_variables)
```

(continues on next page)

(continued from previous page)

```
    matrix[1, 0], matrix[2, 0], matrix[2, 1] = np.asarray(β, dtype=float)
    return matrix


def innovation_covariance(h, β):
    """Construct R_t from one vector of orthogonalized variances."""
    inverse = np.linalg.inv(b_matrix(β))
    return inverse @ np.diag(h) @ inverse.T
```

Each diagonal volatility is a geometric random walk,

$$\log h_{it} = \log h_{i,t-1} + \sigma_i \eta_{it}, \qquad \eta_{it} \sim N(0,1). \tag{101.5}$$

The standardized measurement innovations, coefficient innovations, and volatility innovations are mutually independent.

Setting $Q = 0$ produces constant coefficients with drifting volatility, while holding H_t fixed produces drifting coefficients with constant volatility.

The posterior contains the full paths θ^T and H^T together with Q, β, and $(\sigma_1, \sigma_2, \sigma_3)$.

This posterior has thousands of dimensions, so later sections build a sampler that updates one group of parameters at a time, holding the rest fixed.

101.4 The data

We begin with Cogley and Sargent [2005]'s quarterly U.S. dataset ending in 2000Q4.

Inflation is the log difference of the seasonally adjusted CPI for all urban consumers, point sampled in the third month of each quarter.

Unemployment is the quarterly average of the seasonally adjusted civilian unemployment rate and enters the VAR as $0.01 \log[u/(1-u)]$, a logit transformation that maps a bounded rate into an unconstrained variable.

The nominal interest rate is the log of one plus the three-month Treasury-bill rate, averaged over daily observations in the first month of each quarter and expressed as a quarterly fraction.

The data starts in 1948Q2 because its first inflation observation is already differenced, so two VAR lags make 1948Q4 the first usable regression date.

The following cell performs every transformation and constructs the VAR(2) data directly from the series.

```
def prepare_data(source, ordering=('i', 'u', 'pi')):
    """Transform a quarterly table and construct the VAR data."""
    if isinstance(source, str):
        table = pd.read_csv(source)
    else:
        table = source.copy()
    variables = {
        'i': table['y3'].to_numpy(dtype=float),
        'u': 0.01 * np.log(
            table['ur'].to_numpy(dtype=float)
            / (1 - table['ur'].to_numpy(dtype=float))
        ),
        'pi': table['dp'].to_numpy(dtype=float),
    }
    if sorted(ordering) != ['i', 'pi', 'u']:
        raise ValueError("ordering must be a permutation of ('i', 'u', 'pi')")
```

(continues on next page)

(continued from previous page)

```
    raw_y = np.column_stack([variables[name] for name in ordering])
    raw_dates = table['date'].to_numpy(dtype=float)
    regressors = np.ones((len(table) - n_lags, n_regressors))
    for lag in range(1, n_lags + 1):
        left = 1 + n_variables * (lag - 1)
        regressors[:, left:left + n_variables] = raw_y[n_lags-lag:-lag]
    targets = raw_y[n_lags:]
    dates = raw_dates[n_lags:]
    n_training = 4 * 11 - n_lags - 1
    return {
        'raw_dates': raw_dates,
        'raw_y': raw_y,
        'prior_dates': dates[:n_training],
        'prior_y': targets[:n_training],
        'prior_x': regressors[:n_training],
        'dates': dates[n_training:],
        'y': targets[n_training:],
        'x': regressors[n_training:],
    }


data = prepare_data(data_url)

data_summary = pd.Series(
    {
        'ordering': 'interest, unemployment, inflation',
        'prior sample': '1948Q4--1958Q4',
        'prior observations': len(data['prior_dates']),
        'posterior sample': '1959Q1--2000Q4',
        'posterior observations': len(data['dates']),
        'VAR lags': n_lags,
        'coefficient dimension': n_coefficients,
    },
    name='value',
)

data_summary.to_frame()
```

	value
ordering	interest, unemployment, inflation
prior sample	1948Q4--1958Q4
prior observations	41
posterior sample	1959Q1--2000Q4
posterior observations	168
VAR lags	2
coefficient dimension	21

The early observations calibrate the prior, while the remaining observations form the posterior sample.

Let's view the data in familiar economic units.

```
dates_raw = data['raw_dates']
interest = 400 * np.expm1(data['raw_y'][:, 0])
unemployment = 100 * expit(100 * data['raw_y'][:, 1])
inflation = 400 * data['raw_y'][:, 2]

fig, axes = plt.subplots(3, 1, figsize=(9, 7), sharex=True)
```

(continues on next page)

(continued from previous page)

```
axes[0].plot(dates_raw, inflation, lw=2)
axes[0].set_ylabel('inflation (annual %)')
axes[1].plot(dates_raw, unemployment, lw=2)
axes[1].set_ylabel('unemployment (%)')
axes[2].plot(dates_raw, interest, lw=2)
axes[2].set_ylabel('interest (annual %)')
axes[2].set_xlabel('year')
plt.tight_layout()
plt.show()
```

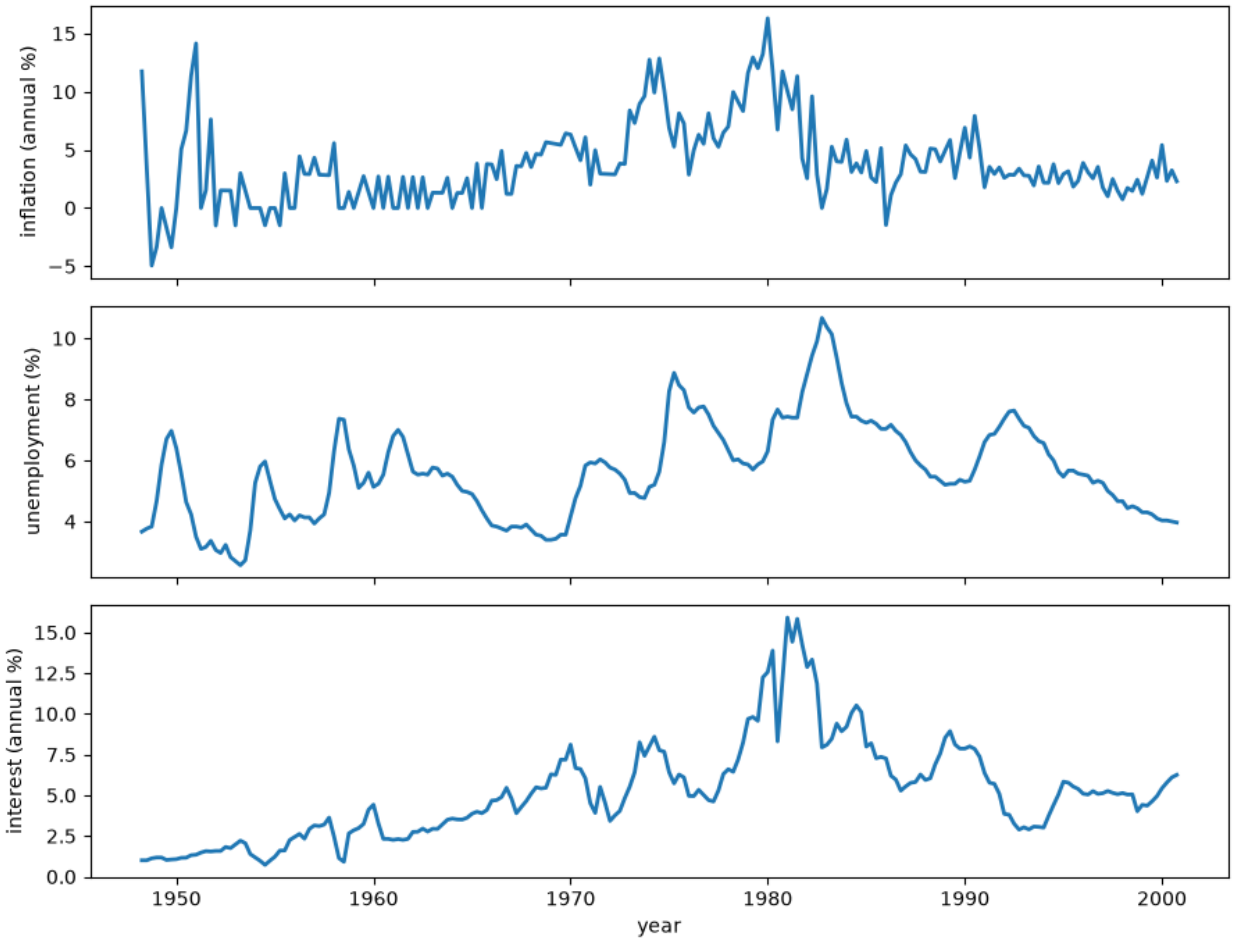


Fig. 101.1: Observed π_t, u_t, and i_t, 1948Q2–2000Q4

Inflation and nominal interest rates rise together through the 1970s and peak around 1980, unemployment peaks only after disinflation begins, and all three series are calmer in the 1990s.

These observations locate the episode but cannot tell whether changed systematic dynamics or unusually large shocks produced it, which is the distinction the model is built to examine.

101.5 Priors

The priors are independent across parameters and deliberately weak so that, in Cogley and Sargent's phrase, "the data are free to speak."

They are calibrated from a time-invariant VAR fitted to the short 1948–1958 training sample.

The initial coefficient prior is a stable, truncated Gaussian,

$$p(\theta_0) \propto I(\theta_0) N(\bar{\theta}, \bar{P}),$$

where $\bar{\theta}$ and $\bar{P}$ come from a constant-coefficient seemingly unrelated regression fitted to the 1948–1958 training sample.

Because all three equations have identical regressors, the SUR coefficient estimates equal equation-by-equation OLS estimates, which lets us implement the calibration compactly.

```
def sur_prior(y, x):
    """Calibrate the Gaussian coefficient prior from a constant VAR."""
    xx_inverse = np.linalg.inv(x.T @ x)
    coefficients = xx_inverse @ x.T @ y
    residuals = y - x @ coefficients
    residual_covariance = np.cov(residuals, rowvar=False, ddof=1)
    θ = coefficients.T.reshape(-1)
    covariance = np.kron(residual_covariance, xx_inverse)
    return θ, covariance, residual_covariance


θ_bar, p_bar, r_bar = sur_prior(data['prior_y'], data['prior_x'])

assert is_stable(θ_bar)
```

The coefficient-drift covariance has the inverse-Wishart prior

$$Q \sim IW_{21}\left(T_0, T_0 \bar{Q}\right), \qquad T_0 = 22, \qquad \bar{Q} = \gamma^2 \bar{P}, \qquad \gamma^2 = 3.5 \times 10^{-4}. \tag{101.6}$$

The convention in (101.6) lists degrees of freedom first and the inverse-Wishart scale matrix second.

Because T_0 is only one greater than the dimension of θ_t, the prior is proper but has no finite mean.

The matrix $\bar{Q}$ is therefore a conservative scale calibration rather than the expectation of Q.

This calibration favors slow coefficient drift before the posterior sees the main sample.

The remaining priors are

$$\begin{aligned} \log h_{i0} &\sim N(\log \bar{R}_{ii}, 10), \\ \beta &\sim N(0, 10000 I_3), \\ \sigma_i^2 &\sim IG\left(\frac{1}{2}, \frac{0.01^2}{2}\right), \end{aligned}$$

where $\bar{R}$ is the residual covariance from the training-sample regression.

The next function gathers these hyperparameters.

```
def calibrate_prior(model_data, y_squared=3.5e-4):
    """Return every calibrated prior for one variable ordering."""
    θ_mean, θ_covariance, residual_covariance = sur_prior(
        model_data['prior_y'], model_data['prior_x']
    )
```

(continues on next page)

(continued from previous page)

```
    degrees_freedom = n_coefficients + 1
    q_center = γ_squared * θ_covariance
    return {
        'θ_mean': θ_mean,
        'θ_covariance': θ_covariance,
        'q_center': q_center,
        'q_scale': degrees_freedom * q_center,
        'q_degrees_freedom': degrees_freedom,
        'log_h_mean': np.log(np.diag(residual_covariance)),
        'log_h_variance': 10.0,
        'β_mean': np.zeros(3),
        'β_variance': 10000.0,
        'σ_degrees_freedom': 1.0,
        'σ_scale': 0.01**2,
        'γ_squared': γ_squared,
    }


prior = calibrate_prior(data)
q_bar = prior['q_center']

prior_summary = pd.Series(
    {
        'dim(θ)': n_coefficients,
        'T0': prior['q_degrees_freedom'],
        'γ squared': prior['γ_squared'],
        'trace(Q bar)': np.trace(q_bar),
        'log-h prior variance': prior['log_h_variance'],
        'β prior variance': prior['β_variance'],
        'σ-squared IG shape': prior['σ_degrees_freedom'] / 2,
        'σ-squared IG scale': prior['σ_scale'] / 2,
    },
    name='value',
)

prior_summary.to_frame()
```

	value
dim(θ)	21.000000
T0	22.000000
γ squared	0.000350
trace(Q bar)	0.003004
log-h prior variance	10.000000
β prior variance	10000.000000
σ-squared IG shape	0.500000
σ-squared IG scale	0.000050

101.6 A Metropolis-within-Gibbs sampler

We simulate the posterior by cycling through five parameter blocks used by Cogley and Sargent [2005].

One pass through all five blocks is called a sweep, and the sampler runs many sweeps to build up the posterior sample.

Cogley and Sargent simulate the unrestricted posterior and then discard a complete MCMC realization whenever its coefficient path is explosive.

But in this case only stable sweeps contribute realizations to the retained restricted-posterior sample.

We instead impose stability inside the coefficient-path block with the elliptical slice sampler of Murray *et al.* [2010].

This changes the transition kernel, but not its target posterior.

1. An auxiliary Gaussian coefficient path is drawn by a Carter–Kohn forward-filter/backward-sample step and used to update the stable path by elliptical slice sampling.
2. The drift covariance Q is drawn from an inverse-Wishart distribution conditional on the coefficient innovations.
3. The volatility innovation variances σ_i^2 are drawn from inverse-gamma distributions conditional on the volatility increments.
4. The covariance parameters β are drawn from two transformed Gaussian regressions among the VAR residuals.
5. The volatility paths H^T are drawn one date at a time by a Jacquier–Polson–Rossi Metropolis step.

This ordering matters: every block in a sweep is paired with the values on which it was actually conditioned.

101.6.1 Coefficient path

Conditional on R^T and Q, a forward Kalman filter followed by the Carter–Kohn backward simulator draws the entire coefficient path [Carter and Kohn, 1994].

The forward pass is a Kalman filter, while the backward pass samples $\theta_T, \theta_{T-1}, \ldots, \theta_0$ in reverse with each state conditioned on the draw that follows it.

Sampling θ_0 is essential.

It supplies all T random-walk increments to the conjugate update for Q.

```
def covariance_root(matrix):
    """Return a numerically stable lower covariance factor."""
    matrix = 0.5 * (matrix + matrix.T)
    scale = max(1.0, np.max(np.abs(np.diag(matrix))))
    return np.linalg.cholesky(matrix + 1e-12 * scale * np.eye(len(matrix)))


def draw_coefficient_path(
    y, x, q, h, β, prior, rng, return_mean=False
):
    """Draw θ_0,...,θ_T and optionally return its smoothing mean."""
    periods = len(y)
    filtered_mean = np.empty((periods + 1, n_coefficients))
    filtered_covariance = np.empty(
        (periods + 1, n_coefficients, n_coefficients)
    )
    predicted_covariance = np.empty_like(filtered_covariance)
    filtered_mean[0] = prior['θ_mean']
    filtered_covariance[0] = prior['θ_covariance']
```

(continues on next page)

(continued from previous page)

```
    predicted_covariance[0] = prior['θ_covariance']
    for t in range(1, periods + 1):
        observation = design_matrix(x[t - 1])
        prediction_covariance = filtered_covariance[t - 1] + q
        r_t = innovation_covariance(h[t], β)
        forecast_covariance = (
            observation @ prediction_covariance @ observation.T + r_t
        )
        gain = linalg.solve(
            forecast_covariance,
            (prediction_covariance @ observation.T).T,
            assume_a='pos',
        ).T
        mean = filtered_mean[t - 1]
        mean = mean + gain @ (y[t - 1] - observation @ mean)
        covariance = (
            prediction_covariance
            - gain @ observation @ prediction_covariance
        )
        covariance = 0.5 * (covariance + covariance.T)
        filtered_mean[t] = mean
        filtered_covariance[t] = covariance
        predicted_covariance[t] = prediction_covariance
    path = np.empty((n_coefficients, periods + 1))
    if not return_mean:
        path[:, -1] = (
            filtered_mean[-1]
            + covariance_root(filtered_covariance[-1])
            @ rng.standard_normal(n_coefficients)
        )
        for t in range(periods - 1, -1, -1):
            smoother = linalg.solve(
                predicted_covariance[t + 1],
                filtered_covariance[t].T,
                assume_a='pos',
            ).T
            mean = filtered_mean[t] + smoother @ (
                path[:, t + 1] - filtered_mean[t]
            )
            covariance = (
                filtered_covariance[t]
                - smoother @ predicted_covariance[t + 1] @ smoother.T
            )
            path[:, t] = mean + covariance_root(covariance) @ (
                rng.standard_normal(n_coefficients)
            )
        return path

    smoothed_mean = np.empty_like(path)
    centered_draw = np.empty_like(path)
    smoothed_mean[:, -1] = filtered_mean[-1]
    centered_draw[:, -1] = covariance_root(filtered_covariance[-1]) @ (
        rng.standard_normal(n_coefficients)
    )
    for t in range(periods - 1, -1, -1):
        smoother = linalg.solve(
            predicted_covariance[t + 1],
```

(continues on next page)

(continued from previous page)

```
            filtered_covariance[t].T,
            assume_a='pos',
        ).T
        smoothed_mean[:, t] = filtered_mean[t] + smoother @ (
            smoothed_mean[:, t + 1] - filtered_mean[t]
        )
        covariance = (
            filtered_covariance[t]
            - smoother @ predicted_covariance[t + 1] @ smoother.T
        )
        centered_draw[:, t] = (
            smoother @ centered_draw[:, t + 1]
            + covariance_root(covariance) @ rng.standard_normal(n_coefficients)
        )
    return smoothed_mean + centered_draw, smoothed_mean


def draw_stable_coefficient_path(
    current, y, x, q, h, β, prior, rng, max_contractions=100
):
    """Elliptical-slice update of the stability-truncated Gaussian path."""
    if not is_stable(current):
        raise ValueError('the elliptical-slice update needs a stable path')
    gaussian_draw, mean = draw_coefficient_path(
        y, x, q, h, β, prior, rng, return_mean=True
    )
    current_centered = current - mean
    innovation = gaussian_draw - mean
    angle = rng.uniform(0, 2 * np.pi)
    lower = angle - 2 * np.pi
    upper = angle
    for contractions in range(max_contractions + 1):
        proposal = (
            mean
            + current_centered * np.cos(angle)
            + innovation * np.sin(angle)
        )
        if is_stable(proposal):
            return proposal, contractions
        if angle < 0:
            lower = angle
        else:
            upper = angle
        angle = rng.uniform(lower, upper)
    raise RuntimeError('elliptical-slice stability bracket did not contract')
```

101.6.2 Drift covariance

Conditional on the coefficient increments, Q has an inverse-Wishart full conditional.

The retained simulation draws Q from a scale matrix formed by its prior scale and all T squared increments from θ_0 through θ_T.

```
def draw_q(θ_path, prior, rng):
    """Draw Q conditional on the sampled coefficient path."""
    increments = np.diff(θ_path, axis=1)
    scale = prior['q_scale'] + increments @ increments.T
    degrees_freedom = prior['q_degrees_freedom'] + increments.shape[1]
    return invwishart.rvs(df=degrees_freedom, scale=scale, random_state=rng)
```

101.6.3 Volatility parameters and paths

Conditional on the volatility increments, each σ_i^2 has an inverse-gamma full conditional.

Conditional on the VAR residuals and H^T, the free elements of B are drawn from two Gaussian regressions.

Conditional on the orthogonalized residuals, each volatility state is updated with the single-site Metropolis step of Jacquier *et al.* [1994].

The random-walk neighbors determine the Gaussian proposal for a log volatility, while the corresponding orthogonalized residual determines whether that proposal is accepted.

The following code implements the conditional updates, including the different endpoint proposals.

```
def var_residuals(y, x, θ_path):
    """Return residuals with shape (T, 3)."""
    if θ_path.shape[1] == len(y) + 1:
        θ_path = θ_path[:, 1:]
    if θ_path.shape[1] != len(y):
        raise ValueError('θ_path must contain T or T + 1 states')
    coefficients = θ_path.T.reshape(len(y), n_variables, n_regressors)
    fitted = np.einsum('tk,tnk->tn', x, coefficients)
    return y - fitted


def draw_σ(h, prior, rng):
    """Draw the three log-volatility innovation standard deviations."""
    increments = np.diff(np.log(h), axis=0)
    shape = (prior['σ_degrees_freedom'] + increments.shape[0]) / 2
    scales = (prior['σ_scale'] + np.sum(increments**2, axis=0)) / 2
    σ_squared = scales / rng.gamma(shape, 1.0, size=n_variables)
    return np.sqrt(σ_squared)


def draw_β(residuals, h, prior, rng):
    """Draw the free elements of B from transformed Gaussian regressions."""
    β = np.empty(3)
    offset = 0
    for equation in range(1, n_variables):
        standardized = residuals / np.sqrt(h[1:, equation])[:, None]
        dependent = standardized[:, equation]
        regressors = -standardized[:, :equation]
        prior_precision = np.eye(equation) / prior['β_variance']
```

(continues on next page)

(continued from previous page)

```
        covariance = np.linalg.inv(prior_precision + regressors.T @ regressors)
        prior_slice = prior['β_mean'][offset:offset + equation]
        mean = covariance @ (
            prior_precision @ prior_slice + regressors.T @ dependent
        )
        β[offset:offset + equation] = (
            mean + covariance_root(covariance) @ rng.standard_normal(equation)
        )
        offset += equation
    return β


def accept_volatility(proposal, current, residual, rng):
    """Apply the Jacquier--Polson--Rossi likelihood acceptance step."""
    log_ratio = (
        -0.5 * np.log(proposal)
        - residual**2 / (2 * proposal)
        + 0.5 * np.log(current)
        + residual**2 / (2 * current)
    )
    return proposal if np.log(rng.random()) <= min(0.0, log_ratio) else current


def draw_volatility_path(h, residuals, β, σ, prior, rng):
    """Update all stochastic-volatility states one date at a time."""
    periods = len(residuals)
    orthogonalized = (b_matrix(β) @ residuals.T).T
    updated = np.empty_like(h)
    for equation in range(n_variables):
        variance = σ[equation]**2
        initial_variance = (
            prior['log_h_variance'] * variance
            / (variance + prior['log_h_variance'])
        )
        initial_mean = initial_variance * (
            prior['log_h_mean'][equation] / prior['log_h_variance']
            + np.log(h[1, equation]) / variance
        )
        updated[0, equation] = np.exp(
            initial_mean + np.sqrt(initial_variance) * rng.standard_normal()
        )
        for t in range(1, periods):
            mean = 0.5 * (
                np.log(updated[t - 1, equation]) + np.log(h[t + 1, equation])
            )
            proposal = np.exp(
                mean + np.sqrt(variance / 2) * rng.standard_normal()
            )
            updated[t, equation] = accept_volatility(
                proposal,
                h[t, equation],
                orthogonalized[t - 1, equation],
                rng,
            )
        proposal = np.exp(
            np.log(updated[-2, equation])
            + σ[equation] * rng.standard_normal()
```

(continues on next page)

(continued from previous page)

```
        )
        updated[-1, equation] = accept_volatility(
            proposal,
            h[-1, equation],
            orthogonalized[-1, equation],
            rng,
        )
    return updated
```

101.6.4 Complete sampler

The restricted posterior assigns zero density to coefficient paths that are explosive at any date, including θ_0.

Stack the whole coefficient path $\theta_0, \dots, \theta_T$ into z, and collect the other parameter blocks in $\lambda = (Q, H^T, \beta, \sigma)$.

Conditional on λ and the data Y^T, the unrestricted Carter–Kohn distribution is Gaussian with smoothing mean m and covariance C,

$$z \mid \lambda, Y^T \sim N(m, C).$$

Let $\mathcal{A}$ be the set of paths whose companion roots are strictly inside the unit circle at every date, so that the restricted full conditional is this Gaussian truncated to $\mathcal{A}$,

$$\pi_{\mathcal{A}}(z \mid \lambda, Y^T) = \frac{N(z; m, C)\, \mathbb{1}_{\mathcal{A}}(z)}{\mathbb{P}\{z \in \mathcal{A} \mid \lambda, Y^T\}}.$$

That normalizing probability is difficult to compute, but the elliptical transition never evaluates it.

Cogley and Sargent instead simulate the unrestricted joint posterior and keep a realization only when its whole path is stable, which is valid because conditioning the unrestricted posterior on $z \in \mathcal{A}$ reproduces the restricted posterior exactly.

If a denotes the unrestricted probability that a path is stable, this rejection scheme needs roughly $1/a$ complete sweeps, including the four other parameter updates, for every retained draw.

The elliptical slice sampler avoids that waste by moving within the stable region instead of restarting from an arbitrary draw.

It starts from the current stable path $z^{(c)}$ and a fresh Carter–Kohn draw $\tilde{z} \sim N(m, C)$, whose centered version $\nu = \tilde{z} - m$ traces an ellipse together with $z^{(c)}$,

$$z(\phi) = m + (z^{(c)} - m)\cos\phi + \nu \sin\phi, \qquad 0 \le \phi < 2\pi,$$

so that $z(0) = z^{(c)}$ and $z(\pi/2) = \tilde{z}$.

The algorithm draws an angle uniformly from the full circle and, whenever $z(\phi)$ is explosive, shrinks the bracket to the side containing the known-stable angle $\phi = 0$ before drawing again.

Because companion roots vary continuously with the coefficients, a nonzero interval around $\phi = 0$ is always stable, so this bracket search always terminates.

Each rejected angle costs only a linear combination and a companion-root check, far cheaper than another Kalman filter and backward simulation.

The transition is valid because rotating the pair $(z^{(c)} - m, \nu)$ by any angle leaves their joint Gaussian density unchanged, so the search moves along a fixed orbit on which every point is equally likely under the unrestricted density.

The stability indicator plays the role of the likelihood in a standard elliptical slice update, and since it equals one at the current point, every accepted angle is automatically a stable one and no separate slice-height draw is needed.

Marginalizing out the auxiliary path shows that this transition leaves the truncated Gaussian $N(z; m, C)\mathbb{1}_{\mathcal{A}}(z)$ invariant, exactly the property a valid transition kernel needs.

The other four blocks require no such adjustment.

Conditional on a stable z, the stability indicator is constant in Q, H^T, β, and σ, so it cancels from each of their full conditionals and leaves the same updates as the unrestricted sampler.

Q's conditional is unchanged in form, but its marginal posterior still tilts toward less explosive drift because every draw of Q is conditioned on a stable coefficient path.

Together, the elliptical transition for z and the unchanged updates for Q, H^T, β, and σ target the same stability-restricted posterior as Cogley and Sargent's original rejection sampler, at a fraction of the cost.

The next function composes these five blocks and includes a stochastic-volatility warm-up.

```
def initial_volatilities(y, prior):
    """Construct the sampler's initial volatility path."""
    changes = np.diff(y, axis=0)
    centered = changes - changes.mean(axis=0)
    log_h = np.empty((len(y) + 1, n_variables))
    log_h[:2] = prior['log_h_mean']
    log_h[2:] = np.log(np.maximum(centered**2, np.finfo(float).tiny))
    return np.exp(log_h)


def run_sampler(
    y,
    x,
    prior,
    n_sweeps=1_000,
    burn=500,
    thin=1,
    seed=42,
    warmup=200,
    max_contractions=100,
    stable=True,
    fixed_q=None,
    retain=('S0D', 'SD', 'QD', 'HD', 'CD', 'VD', 'stable_draw'),
    progress_every=0,
):
    """Run a Gibbs sampler for the unrestricted or stable posterior.

    For the stable posterior, an elliptical-slice transition updates the FFBS
    path inside its stability-truncated Gaussian full conditional.  Passing
    ``fixed_q`` holds Q at that value (for example a matrix of zeros) instead of
    drawing it, which nests the constant-coefficient model.
    """
    if not (0 <= burn < n_sweeps and thin >= 1):
        raise ValueError('require 0 <= burn < n_sweeps and thin >= 1')
    if (n_sweeps - burn) % thin:
        raise ValueError('(n_sweeps - burn) must be divisible by thin')
    valid_retain = {'S0D', 'SD', 'QD', 'HD', 'CD', 'VD', 'stable_draw'}
    unknown = set(retain) - valid_retain
    if unknown:
        raise ValueError(f'unknown retained arrays: {sorted(unknown)}')

    started = time.perf_counter()
    rng = np.random.default_rng(seed)
```

(continues on next page)

(continued from previous page)

```
    h = initial_volatilities(y, prior)
    β = prior['β_mean'].copy()
    warm_θ = np.repeat(prior['θ_mean'][:, None], len(y), axis=1)
    warm_residuals = var_residuals(y, x, warm_θ)
    for _ in range(warmup):
        σ = draw_σ(h, prior, rng)
        β = draw_β(warm_residuals, h, prior, rng)
        h = draw_volatility_path(
            h, warm_residuals, β, σ, prior, rng
        )

    q = (
        prior['q_center'].copy()
        if fixed_q is None
        else np.array(fixed_q, dtype=float)
    )
    θ = np.repeat(
        prior['θ_mean'][:, None], len(y) + 1, axis=1
    )
    if stable and not is_stable(θ):
        raise ValueError('the prior mean does not provide a stable start')
    slice_contractions = 0
    maximum_slice_contractions = 0

    retained = {name: [] for name in retain}
    saved_stability = []
    for sweep in range(1, n_sweeps + 1):
        if stable:
            θ, contractions = draw_stable_coefficient_path(
                θ,
                y,
                x,
                q,
                h,
                β,
                prior,
                rng,
                max_contractions=max_contractions,
            )
        else:
            θ = draw_coefficient_path(y, x, q, h, β, prior, rng)
            contractions = 0
        slice_contractions += contractions
        maximum_slice_contractions = max(
            maximum_slice_contractions, contractions
        )
        if fixed_q is None:
            q = draw_q(θ, prior, rng)
        residuals = var_residuals(y, x, θ)
        σ = draw_σ(h, prior, rng)
        β = draw_β(residuals, h, prior, rng)
        h = draw_volatility_path(
            h, residuals, β, σ, prior, rng
        )

        if sweep > burn and (sweep - burn) % thin == 0:
            path_is_stable = is_stable(θ)
```

(continues on next page)

(continued from previous page)

```
            saved_stability.append(path_is_stable)
            values = {
                'S0D': θ[:, 0],
                'SD': θ[:, 1:],
                'QD': q,
                'HD': h,
                'CD': β,
                'VD': σ,
                'stable_draw': path_is_stable,
            }
            for name in retained:
                retained[name].append(np.asarray(values[name]).copy())

        if progress_every and sweep % progress_every == 0:
            elapsed = time.perf_counter() - started
            print(
                f'{sweep:,}/{n_sweeps:,} sweeps; '
                f'{slice_contractions:,} slice contractions; '
                f'{elapsed / 60:.1f} minutes',
                flush=True,
            )

    stack_axis = {
        'S0D': 1,
        'SD': 2,
        'QD': 2,
        'HD': 2,
        'CD': 1,
        'VD': 1,
        'stable_draw': 0,
    }
    result = {
        name: np.stack(values, axis=stack_axis[name])
        for name, values in retained.items()
    }
    result['diagnostics'] = {
        'sampler_version': 3,
        'seed': int(seed),
        'stable_restriction': bool(stable),
        'n_sweeps': int(n_sweeps),
        'burn': int(burn),
        'thin': int(thin),
        'warmup': int(warmup),
        'retained_draws': int((n_sweeps - burn) // thin),
        'slice_contractions': int(slice_contractions),
        'mean_slice_contractions': float(slice_contractions / n_sweeps),
        'maximum_slice_contractions': int(maximum_slice_contractions),
        'retained_stability_rate': float(np.mean(saved_stability)),
        'elapsed_seconds': float(time.perf_counter() - started),
    }
    return result
```

The executable version below uses 1,000 sweeps, discards the first 500, and retains the remaining 500.

It uses the complete historical sample and the ordering (i, u, π).

Instead of running a large MCMC experiment, we intentionally keep the sampler run small so that it finishes in a reasonable time for a lecture.

This short run illustrates the method but not a numerical replication.

However the main qualitative features of the posterior are close to those reported by Cogley and Sargent.

```
posterior = run_sampler(
    data['y'],
    data['x'],
    prior,
    n_sweeps=1_000,
    burn=500,
    thin=1,
    seed=42,
    warmup=200,
    stable=True,
    progress_every=0,
)

def validate_posterior_arrays(result, periods):
    """Check posterior shapes, finiteness, positivity, and stability."""
    draws = result['diagnostics']['retained_draws']
    expected = {
        'S0D': (n_coefficients, draws),
        'SD': (n_coefficients, periods, draws),
        'QD': (n_coefficients, n_coefficients, draws),
        'HD': (periods + 1, n_variables, draws),
        'CD': (3, draws),
        'VD': (3, draws),
        'stable_draw': (draws,),
    }
    assert {name: result[name].shape for name in expected} == expected
    assert all(np.all(np.isfinite(result[name])) for name in expected)
    assert np.all(result['HD'] > 0)
    assert np.all(result['VD'] > 0)
    assert np.all(result['stable_draw'])
    return expected


expected_shapes = validate_posterior_arrays(posterior, len(data['dates']))
```

101.7 What the data say

We summarize the posterior by its mean coefficient path $\mathbb{E}(\theta_t \mid T)$ and mean covariance path $\mathbb{E}(R_t \mid T)$, and then interpret them in the context of the question we asked.

101.7.1 The rate and structure of drift

The trace of Q measures the total rate of coefficient drift, with $\mathrm{tr}(Q) = 0$ corresponding to constant coefficients.

The histogram shows the retained Q draws and the prior scale.

```
trace_q = np.trace(posterior['QD'], axis1=0, axis2=1)
fig, ax = plt.subplots()
ax.hist(trace_q, bins=30, histtype='step', lw=2)
ax.axvline(np.trace(q_bar), color='C1', lw=2,
           label=r'prior $\mathrm{tr}(\bar Q)$')
```

(continues on next page)

(continued from previous page)

```
ax.set_xlabel(r'$\mathrm{tr}(Q)$')
ax.set_ylabel('frequency')
ax.legend()
plt.show()
```

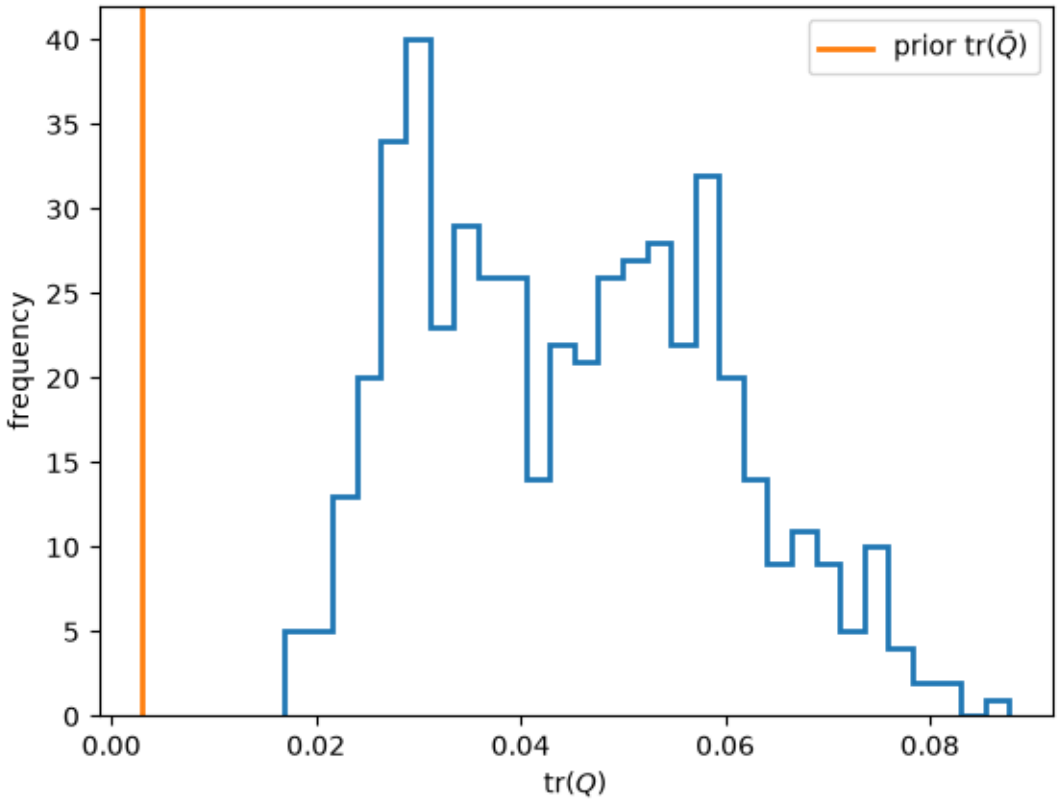


Fig. 101.2: Posterior tr(Q) and prior tr($\bar{Q}$)

The posterior drift rate lies well above the conservative prior calibration, indicating more coefficient variation than that calibration anticipated.

This is not a formal comparison with a fixed-coefficient model because the continuous prior assigns no point mass to $Q = 0$.

Within the fitted TVP-VAR, this variation is attributed to changing systematic relationships; it does not identify policy as the cause.

```
θ_mean = posterior['SD'].mean(axis=2)
mean_path_root_modulus = np.max(
    np.abs(companion_roots(θ_mean)), axis=1
)
assert np.all(mean_path_root_modulus < 1)
coefficient_labels = (
    'constant',
    r'$i_{t-1}$',
    r'$u_{t-1}$',
```

(continues on next page)

(continued from previous page)

```
    r'$\pi_{t-1}$',
    r'$i_{t-2}$',
    r'$u_{t-2}$',
    r'$\pi_{t-2}$',
)
equation_labels = (
    'interest equation',
    'unemployment equation',
    'inflation equation',
)


def plot_equation_coefficients(axes, dates, θ_path):
    """Plot seven labeled coefficients for each VAR equation."""
    first_lines = None
    for equation, (ax, label) in enumerate(zip(axes, equation_labels)):
        start = equation * len(coefficient_labels)
        stop = start + len(coefficient_labels)
        lines = ax.plot(dates, θ_path[start:stop].T, lw=2)
        for line, coefficient_label in zip(lines, coefficient_labels):
            line.set_label(coefficient_label)
        if first_lines is None:
            first_lines = lines
        ax.axhline(0, color='0.65', lw=1)
        ax.set_xlabel('year')
        ax.set_ylabel(f'{label} coefficient')
    return first_lines


fig, axes = plt.subplots(1, 3, figsize=(10, 4), sharex=True)
coefficient_lines = plot_equation_coefficients(
    axes,
    data['dates'],
    θ_mean,
)
fig.legend(
    coefficient_lines,
    coefficient_labels,
    loc='lower center',
    ncol=4,
)
plt.tight_layout(rect=(0, 0.17, 1, 1))
plt.show()
```

The unemployment equation is comparatively stable, whereas several inflation equation coefficients move strongly through the 1970s and turn around near 1980.

The drift is therefore concentrated in how inflation propagates rather than spread evenly across the VAR, and individual lag coefficients should not be given structural interpretations because paired lags can offset one another.

We summarize drift for the ordering (i, u, π), which has the smallest stable posterior mean $\operatorname{tr}(Q)$.

```
trace_q = np.trace(posterior['QD'], axis1=0, axis2=1)
q_mean = posterior['QD'].mean(axis=2)
drift_summary = {
    r'\text{Posterior mean } \operatorname{tr}(Q)': np.trace(q_mean),
```

(continues on next page)

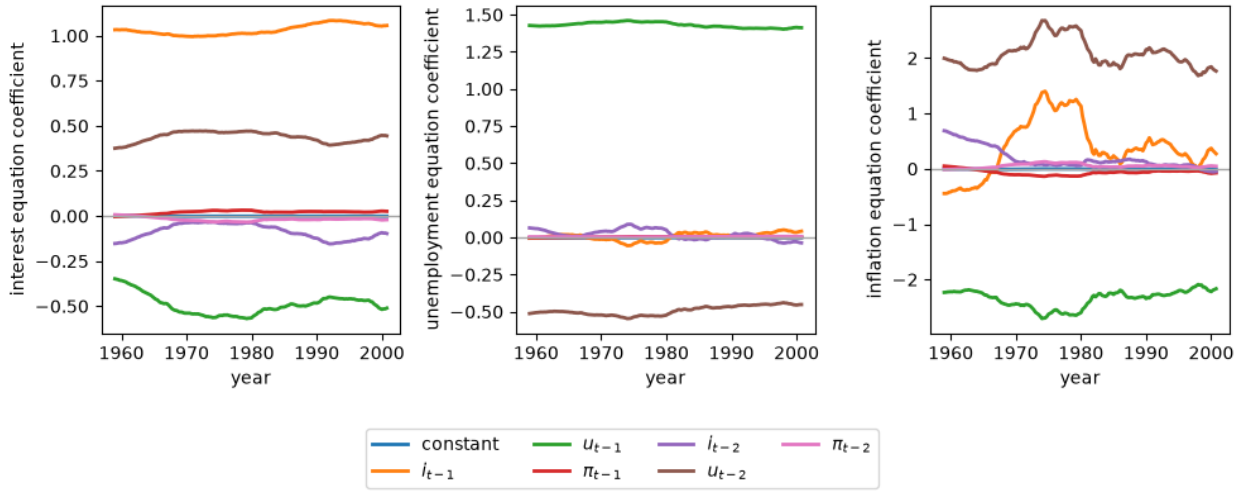


Fig. 101.3: Posterior mean VAR coefficients $E(\theta_t \mid T)$

(continued from previous page)

```
    r'\text{Posterior mean largest eigenvalue}': (
        np.linalg.eigvalsh(q_mean)[-1]
    ),
    r'\text{Prior } \operatorname{tr}(\bar Q)': np.trace(q_bar),
}
drift_rows = ' \\\\\n'.join(
    f'{label} & {value:.4f}' for label, value in drift_summary.items()
)
display(Math(rf'''
\begin{{array}}{{lr}}
\text{{Quantity}} & \text{{Estimate}} \\
\hline
{drift_rows}
\end{{array}}
'''))
```

Quantity	Estimate
Posterior mean $\operatorname{tr}(Q)$	0.0450
Posterior mean largest eigenvalue	0.0264
Prior $\operatorname{tr}(\bar{Q})$	0.0030

Cogley and Sargent estimated every ordering.

Their posterior means show that the ordering changes magnitudes but does not remove drift:

Ordering	Stable $\operatorname{tr}(Q)$	Stable $\max(\lambda)$	Unrestricted $\operatorname{tr}(Q)$	Unrestricted $\max(\lambda)$
(i, π, u)	0.055	0.025	0.056	0.027
(i, u, π)	0.047	0.023	0.059	0.031
(π, i, u)	0.064	0.031	0.082	0.044
(π, u, i)	0.062	0.031	0.088	0.051
(u, i, π)	0.057	0.026	0.051	0.028
(u, π, i)	0.055	0.024	0.072	0.035

For the minimum-Q ordering, removing the stability restriction raises the posterior mean drift rate, as the table above shows.

The analysis that follows adopts the (i, u, π) ordering, which places the nominal interest rate first and inflation last.

Diagonalizing the posterior mean of Q reveals that the drift is low dimensional.

The following eigendecomposition summarizes the posterior mean of Q.

```
q_mean = posterior['QD'].mean(axis=2)
q_eigenvalues = np.linalg.eigvalsh(q_mean)[::-1]
q_cumulative = np.cumsum(q_eigenvalues) / q_eigenvalues.sum()

drift_structure = pd.DataFrame(
    {
        'eigenvalue': q_eigenvalues[:3],
        'cumulative share': q_cumulative[:3],
    },
    index=pd.Index(range(1, 4), name='principal component'),
)
drift_structure.round(4)
```

principal component	eigenvalue	cumulative share
1	0.0264	0.5875
2	0.0156	0.9339
3	0.0013	0.9625

The first three principal components account for the large majority of total coefficient drift even though the VAR contains 21 coefficients.

101.7.2 The evolution of volatility

We first ask how the *size* of the shocks changed.

Equation (101.4) can be averaged over draws without constructing a four-dimensional covariance array.

```
def mean_innovation_covariance(h_draws, β_draws):
    """Compute E(R_t | T) with working memory proportional to T times D."""
    n_draws = h_draws.shape[2]
    matrices = np.broadcast_to(np.eye(3), (n_draws, 3, 3)).copy()
    matrices[:, 1, 0] = β_draws[0]
    matrices[:, 2, 0] = β_draws[1]
    matrices[:, 2, 1] = β_draws[2]
    inverses = np.linalg.solve(
        matrices,
        np.broadcast_to(np.eye(3), matrices.shape),
    )
    h = h_draws[1:]
    mean = np.empty((h.shape[0], 3, 3))
    for row in range(3):
        for column in range(row + 1):
            value = np.zeros(h.shape[0])
            for shock in range(3):
                weights = inverses[:, row, shock] * inverses[:, column, shock]
                value += h[:, shock, :] @ weights
            mean[:, row, column] = value / n_draws
```

(continues on next page)

(continued from previous page)

```
            mean[:, column, row] = mean[:, row, column]
    return mean


r_mean = mean_innovation_covariance(posterior['HD'], posterior['CD'])
```

The next plot shows the innovation standard deviations and correlations.

```
variances = ((0, 'Nominal interest'), (2, 'Inflation'), (1, 'Unemployment'))
correlations = (
    (0, 1, 'Interest--unemployment'),
    (0, 2, 'Interest--inflation'),
    (2, 1, 'Inflation--unemployment'),
)
fig, axes = plt.subplots(3, 2, figsize=(9, 8), sharex=True)
for row, (index, label) in enumerate(variances):
    axes[row, 0].plot(
        data['dates'], 10000 * np.sqrt(r_mean[:, index, index]), lw=2
    )
    axes[row, 0].set_title(label)
for row, (left, right, label) in enumerate(correlations):
    scale = np.sqrt(r_mean[:, left, left] * r_mean[:, right, right])
    axes[row, 1].plot(data['dates'], r_mean[:, left, right] / scale, lw=2)
    axes[row, 1].set_title(label)
axes[1, 0].set_ylabel(
    r'innovation standard deviation $\times 10^4$'
)
axes[1, 1].set_ylabel('correlation')
axes[-1, 0].set_xlabel('year')
axes[-1, 1].set_xlabel('year')
plt.tight_layout()
plt.show()
```

The interest-rate and inflation innovation standard deviations peak sharply around 1980, whereas unemployment innovation volatility declines more gradually toward the end of the sample.

All three innovation correlations also move most abruptly around 1980, so both the size and the joint composition of reduced-form shocks changed.

These movements give the changing-shocks, or bad-luck, explanation an important role, although the interest-rate innovation is not itself a structural monetary-policy shock.

The log determinant of the posterior mean covariance matrix summarizes the generalized one-step innovation variance [Whittle, 1953].

The following transformation summarizes generalized innovation variance.

```
sign, logdet_r = np.linalg.slogdet(r_mean)
assert np.all(sign > 0)
fig, ax = plt.subplots()
ax.plot(data['dates'], logdet_r, lw=2)
ax.set_xlabel('year')
ax.set_ylabel(r'$\log |E(R_t\mid T)|$')
plt.show()
```

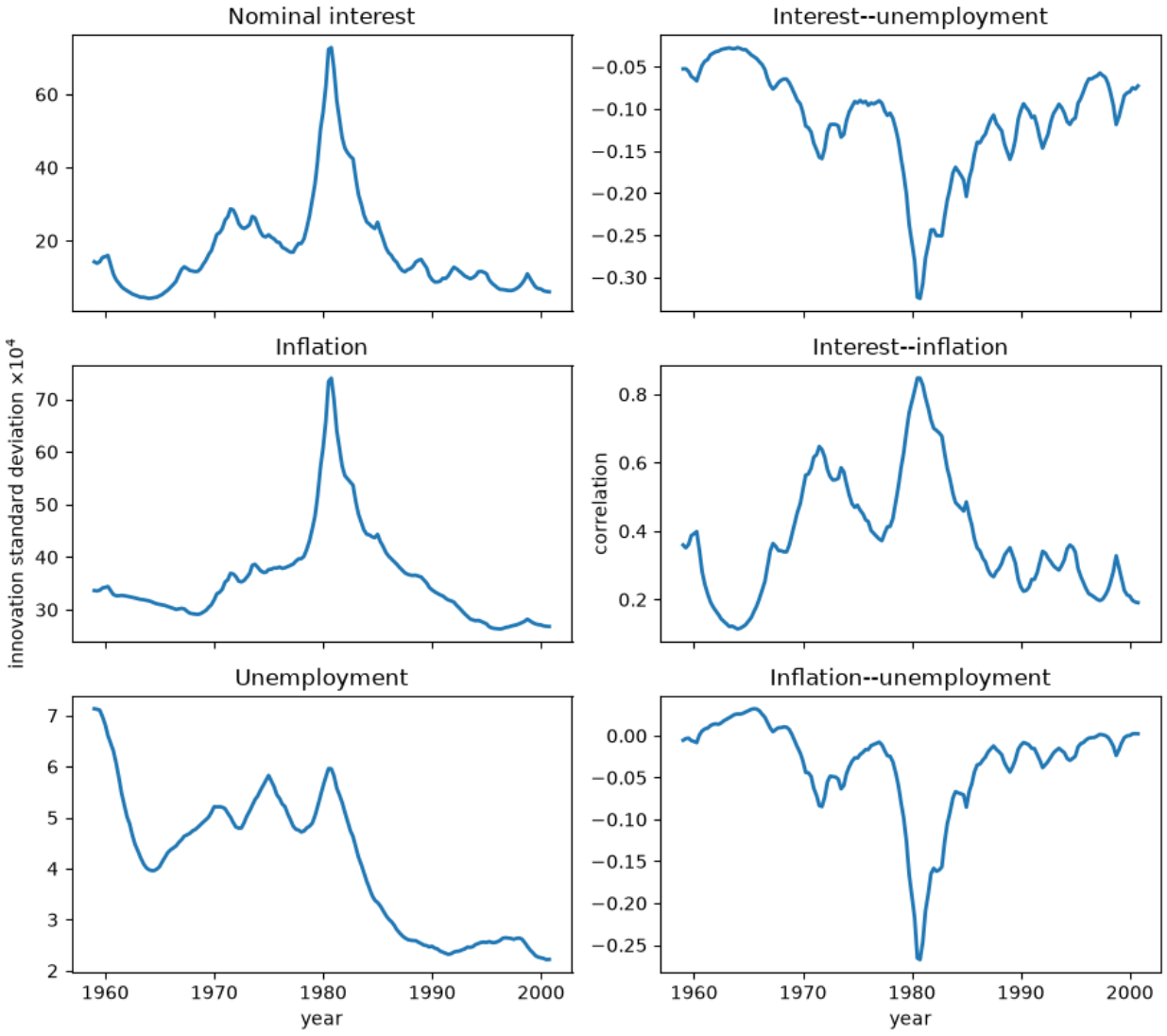


Fig. 101.4: Standard deviations and correlations implied by $E(R_t \mid T)$

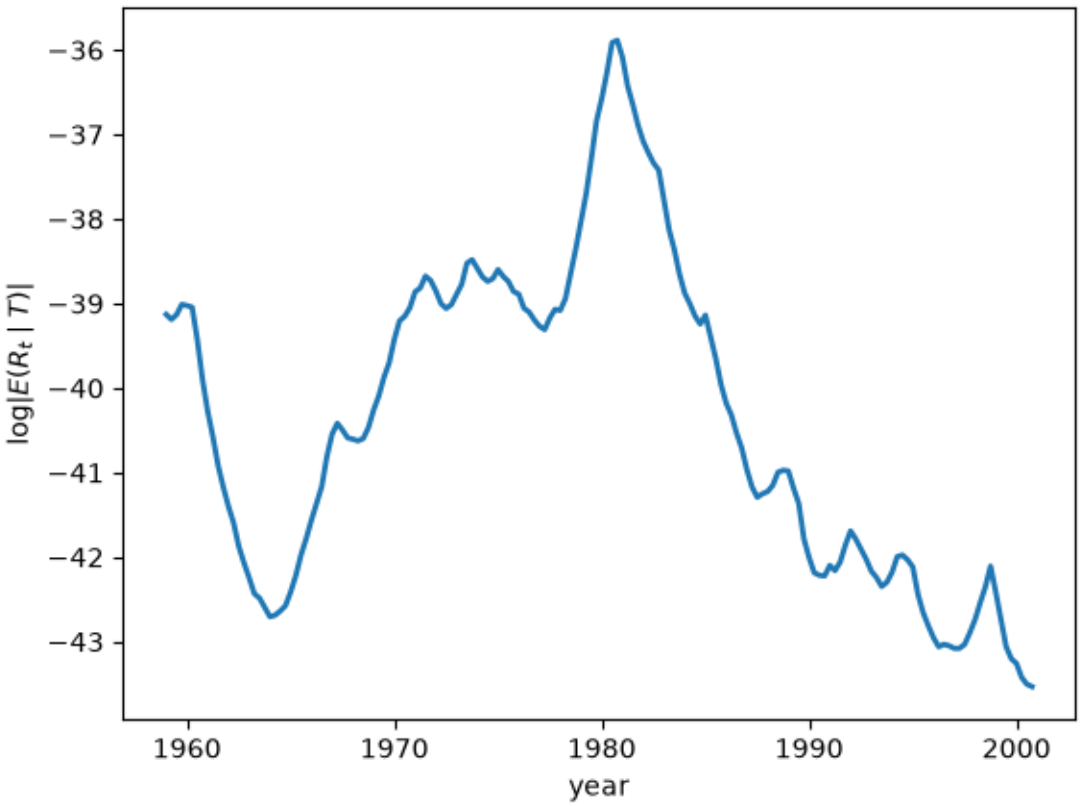


Fig. 101.5: Generalized innovation variance $\log |E(R_t \mid T)|$

Because a less negative log determinant means greater joint innovation variance, the two-step rise to an exceptional 1981 peak and the long subsequent decline mark a large shock episode followed by the Great Moderation documented by Kim and Nelson [1999] and McConnell and Perez-Quiros [2000].

This is the clearest aggregate evidence for changing luck, but it cannot explain the coefficient-based changes in inflation dynamics examined next.

101.7.3 Core inflation and the natural rate

To study the systematic dynamics, write the VAR at date t in companion form as

$$z_t = \mu_{t|T} + A_{t|T} z_{t-1} + e_t.$$

At date t, the local mean m_t is the fixed point to which the companion-form VAR would converge if its coefficients remained fixed at their posterior means and future innovations were zero.

$$m_t = (I - A_{t|T})^{-1} \mu_{t|T}, \qquad \bar{\pi}_t = 4 s_\pi m_t, \qquad \bar{u}_t = \frac{\exp(100 s_u m_t)}{1 + \exp(100 s_u m_t)}. \tag{101.7}$$

Here s_π and s_u select inflation and unemployment from m_t, the factor four annualizes inflation, and the inverse-logit transformation returns unemployment to its observed rate.

These are date-specific steady states of frozen local systems rather than unconditional means of the globally drifting process.

Core inflation is the long-horizon inflation forecast implied by freezing the date-t coefficients, while the natural rate is the corresponding long-run unemployment anchor rather than a natural interest rate or a forecast of next quarter's unemployment.

Freezing the current coefficients and projecting forward is exactly the *anticipated-utility* device used by the learning governments of *Adaptive Learning and Escape Dynamics* and *Escaping Nash Inflation*, who act as if their current beliefs will never be revised.

The following implementation annualizes core inflation and reverses the archive's unemployment transformation.

```
def local_means(θ_path):
    """Compute local core inflation and the natural unemployment rate."""
    θ_path = np.asarray(θ_path, dtype=float)
    if θ_path.ndim == 1:
        θ_path = θ_path[:, None]
    core = np.empty(θ_path.shape[1])
    natural = np.empty(θ_path.shape[1])
    for t in range(θ_path.shape[1]):
        intercept, companion = companion_matrix(θ_path[:, t])
        mean = np.linalg.solve(np.eye(6) - companion, intercept)
        core[t] = 4 * mean[2]
        natural[t] = expit(100 * mean[1])
    return core, natural


core_inflation, natural_rate = local_means(θ_mean)
```

We plot the fourth-quarter observation from each year.

```
def annual_indices(dates, start=1960):
    """Return the final observation in each year from a start date."""
    year_values = np.floor(dates + 1e-8).astype(int)
    return np.array(
```

(continues on next page)

(continued from previous page)

```
        [
            np.flatnonzero(year_values == year)[-1]
            for year in np.unique(year_values)
            if year >= start
        ]
    )


years = np.floor(data['dates'] + 1e-8).astype(int)
annual = annual_indices(data['dates'])

fig, ax = plt.subplots()
ax.plot(data['dates'][annual], 100 * core_inflation[annual], 'o-', lw=2,
        markersize=3, label='core inflation')
ax.plot(data['dates'][annual], 100 * natural_rate[annual], '+-', lw=2,
        markersize=5, label='natural rate')
ax.set_xlabel('year')
ax.set_ylabel('percent')
ax.legend()
plt.show()
```

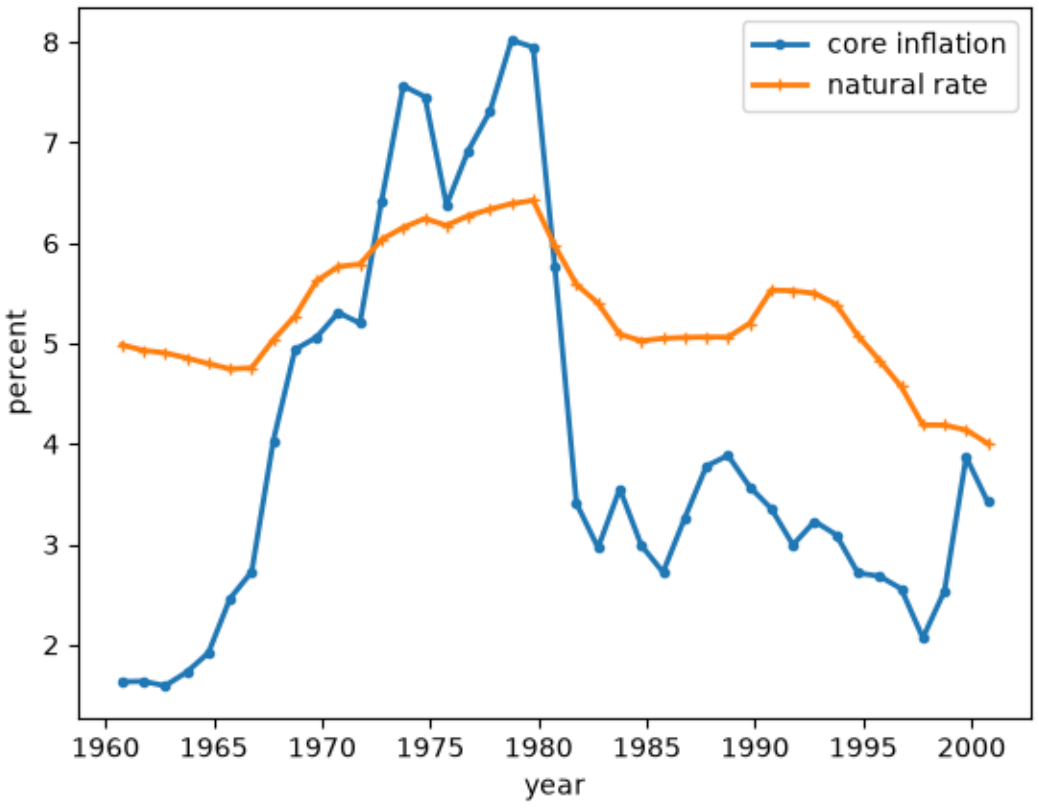


Fig. 101.6: Local means $\bar{\pi}_t$ and $\bar{u}_t$

```
core_summary = pd.Series(
    {
```

(continues on next page)

(continued from previous page)

```
        'early-1960s mean core inflation (%)': (
            100 * core_inflation[(years >= 1960) & (years <= 1964)].mean()
        ),
        'peak core inflation (%)': 100 * core_inflation[annual].max(),
        '1985--2000 mean core inflation (%)': (
            100 * core_inflation[(years >= 1985) & (years <= 2000)].mean()
        ),
    },
    name='estimate',
)
core_summary.to_frame().round(2)
```

```
                                    estimate
early-1960s mean core inflation (%)      1.67
peak core inflation (%)                  8.02
1985--2000 mean core inflation (%)       3.09
```

Core inflation climbs from a low level in the early 1960s to a high peak near 1980 and then falls back, while the natural unemployment rate rises more smoothly from about 5 to 6.5 percent before returning toward 4 percent.

Because both lines are determined by the fitted coefficients rather than by realized shocks, their persistent shifts point to a changing systematic component.

The following calculation summarizes their comovement over the full posterior sample.

```
core_natural_correlation = np.corrcoef(core_inflation, natural_rate)[0, 1]

core_natural_correlation
```

```
np.float64(0.8375125177866256)
```

The strong positive quarterly correlation between $\bar{\pi}_t$ and $\bar{u}_t$ says that the model-implied long-run inflation and unemployment anchors share a broad cycle, not that current inflation and current unemployment must move together.

Because $(I - A_t)^{-1}$ amplifies small coefficient changes when the largest root is close to one, long-run means are intrinsically more sensitive than short-horizon forecasts.

101.7.4 Inflation persistence

We now ask whether the *systematic* dynamics drifted on top of the moving volatilities.

The main summary is inflation persistence, measured by the normalized spectrum of inflation at frequency zero.

The spectral density of inflation at date t is

$$f_{\pi\pi}(\omega, t) = \frac{1}{2\pi} s_\pi (I - A_{t|T} e^{-i\omega})^{-1} \mathcal{R}_t (I - A_{t|T}^\top e^{i\omega})^{-1} s_\pi^\top, \tag{101.8}$$

where $\mathcal{R}_t$ embeds $\mathbb{E}(R_t \mid T)$ in the companion system.

Low-frequency power depends on both the autoregressive coefficients and the innovation covariance.

The next function evaluates inflation power at any frequency measured in cycles per quarter and also returns inflation variance at date t.

```
def inflation_spectrum(θ, covariance, frequencies):
    """Compute inflation power and its variance-normalized counterpart."""
    _, companion = companion_matrix(θ)
    innovation = np.zeros((6, 6))
    innovation[:3, :3] = covariance
    selector = np.zeros(6)
    selector[2] = 1
    stationary = linalg.solve_discrete_lyapunov(companion, innovation)
    variance = float(selector @ stationary @ selector)
    power = np.empty(len(frequencies))
    for index, frequency in enumerate(frequencies):
        phase = np.exp(-2j * np.pi * frequency)
        transfer = np.linalg.solve(np.eye(6) - companion * phase, np.eye(6))
        power[index] = np.real(
            selector @ transfer @ innovation @ transfer.conj().T @ selector
        ) / (2 * np.pi)
    return power, power / variance
```

The normalized spectrum divides by inflation variance at date t,

$$g_{\pi\pi}(\omega, t) = \frac{f_{\pi\pi}(\omega, t)}{\int_{-\pi}^{\pi} f_{\pi\pi}(\omega, t) d\omega}, \tag{101.9}$$

so $g_{\pi\pi}(0, t)$ is an autocorrelation-based persistence measure.

The normalization removes a common scale factor from R_t, but it can still depend on the relative variances and covariances in R_t.

The first figure isolates frequency zero as a one-dimensional persistence summary.

```
zero_frequency = np.array([0.0])
inflation_persistence = np.array([
    inflation_spectrum(θ_mean[:, t], r_mean[t], zero_frequency)[1][0]
    for t in range(len(data['dates']))
])

fig, ax = plt.subplots()
ax.plot(
    data['dates'][annual],
    inflation_persistence[annual],
    'o-',
    lw=2,
    markersize=3,
)
ax.set_xlabel('year')
ax.set_ylabel(r'$g_{\pi\pi}(0,t)$')
plt.show()
```

```
persistence_summary = pd.Series(
    {
        '1960--64 mean': inflation_persistence[
            (years >= 1960) & (years <= 1964)
        ].mean(),
        '1970--79 mean': inflation_persistence[
            (years >= 1970) & (years <= 1979)
        ].mean(),
```

(continues on next page)

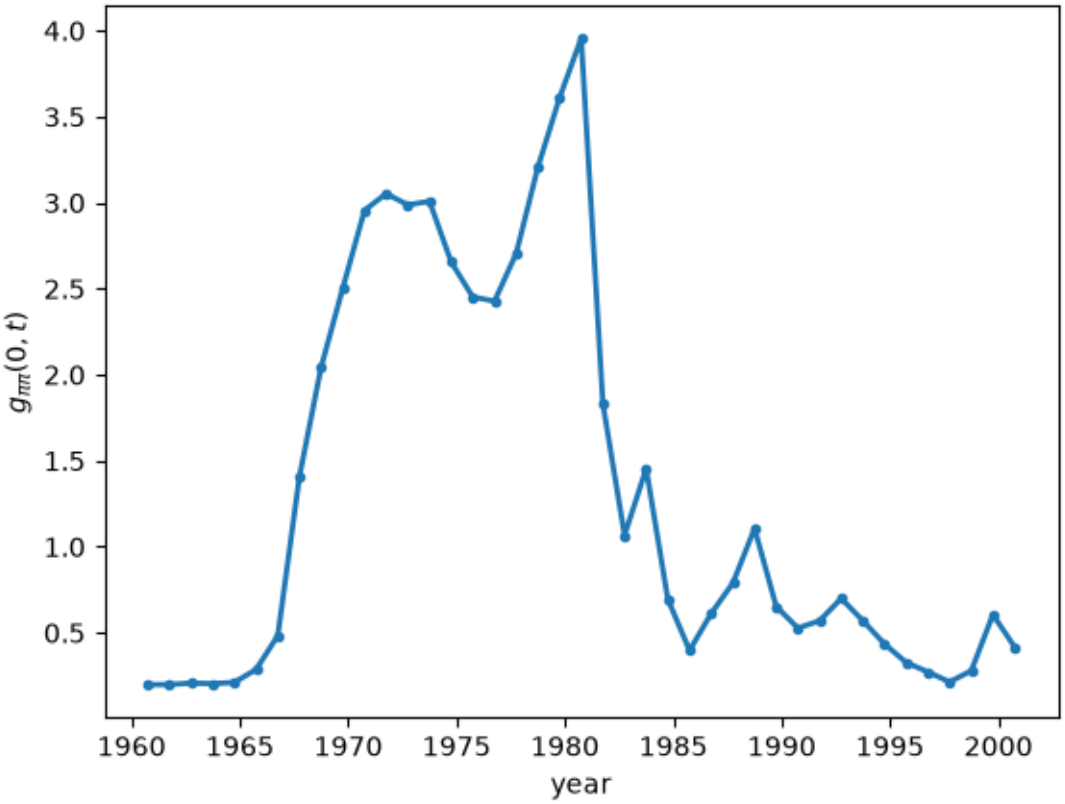


Fig. 101.7: Normalized zero-frequency spectrum $g_{\pi\pi}(0, t)$

(continued from previous page)

```
        '1985--2000 mean': inflation_persistence[
            (years >= 1985) & (years <= 2000)
        ].mean(),
        'peak': inflation_persistence[annual].max(),
        'peak year': years[annual][np.argmax(inflation_persistence[annual])],
    },
    name='estimate',
)
persistence_summary.to_frame().round(3)
```

```
                 estimate
1960--64 mean       0.204
1970--79 mean       2.865
1985--2000 mean     0.537
peak                3.956
peak year        1980.000
```

Normalized zero-frequency power rises sharply from a low level in the early 1960s to a high peak around 1980 and then falls back below one for most of the remaining sample.

The post-1980 collapse cannot be explained by a proportional rescaling of all innovations, although the normalized statistic can still depend on the composition of R_t.

For comparison, an $AR(1)$ with coefficient ρ has normalized zero-frequency power $(1+\rho)/[2\pi(1-\rho)]$.

Values between 2 and 10 correspond to ρ between approximately 0.85 and 0.97.

The zero-frequency path omits the rest of the frequency distribution.

The following heatmaps show how raw and variance-normalized inflation power move over both time and frequency.

```
def inflation_spectrum_surface(θ_path, covariance_path, frequencies):
    """Evaluate the inflation spectrum at each date on a frequency grid."""
    raw = np.empty((len(frequencies), θ_path.shape[1]))
    normalized = np.empty_like(raw)
    for date in range(θ_path.shape[1]):
        raw[:, date], normalized[:, date] = inflation_spectrum(
            θ_path[:, date],
            covariance_path[date],
            frequencies,
        )
    return raw, normalized


spectrum_frequencies = np.linspace(0, 0.5, 41)
raw_spectrum, normalized_spectrum = inflation_spectrum_surface(
    θ_mean,
    r_mean,
    spectrum_frequencies,
)
```

```
spectrum_start = data['dates'] >= 1960
fig, axes = plt.subplots(1, 2, figsize=(9, 4), sharey=True)
surfaces = (
    (raw_spectrum, 'raw spectrum', 'log10 power'),
    (normalized_spectrum, 'normalized spectrum', 'log10 normalized power'),
)
```

(continues on next page)

(continued from previous page)

```
for ax, (surface, title, color_label) in zip(axes, surfaces):
    image = ax.pcolormesh(
        data['dates'][spectrum_start],
        spectrum_frequencies,
        np.log10(surface[:, spectrum_start]),
        shading='auto',
    )
    ax.set_xlabel('year')
    ax.set_ylabel(f'{title}\ncycles per quarter')
    fig.colorbar(image, ax=ax, label=color_label)
plt.tight_layout()
plt.show()
```

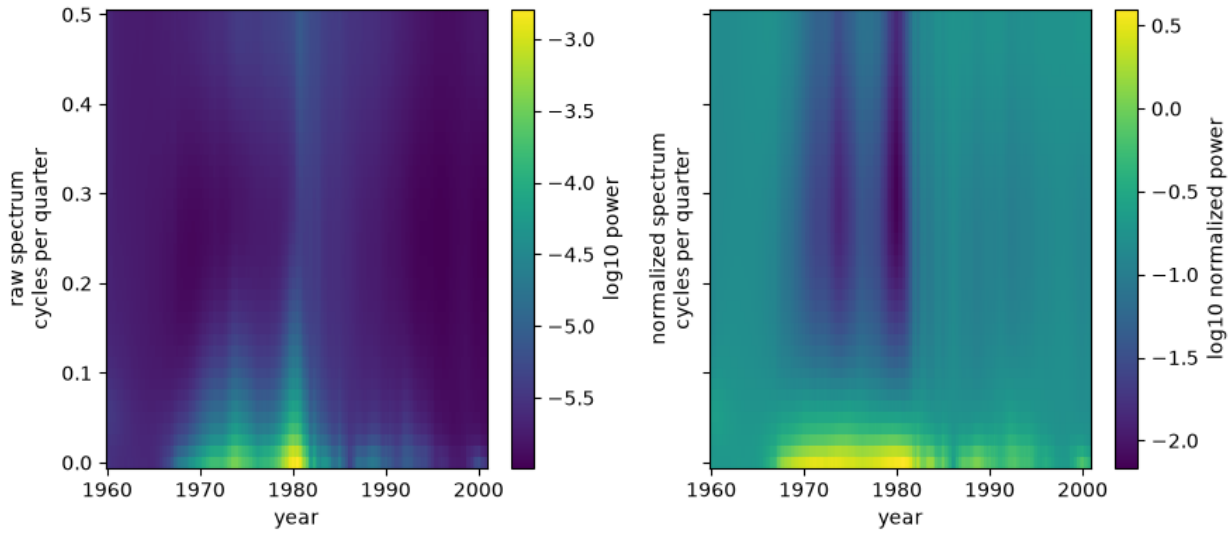


Fig. 101.8: Inflation spectra $f_{\pi\pi}(\omega, t)$ and $g_{\pi\pi}(\omega, t)$

The raw spectrum is brightest near frequency zero around 1980 because shocks and persistence are both elevated, while the normalized spectrum retains a broad low-frequency ridge through the 1970s that recedes after 1980.

The ridge that survives normalization shows that the Great Inflation was not only a high-volatility episode because inflation shocks were also propagated more persistently.

Point estimates do not reveal how strongly the data locate these paths.

We therefore compute the same annual features from every retained draw.

```
def posterior_feature_draws(result, indices):
    """Compute selected local means and persistence for retained draws."""
    n_draws = result['SD'].shape[2]
    shape = (len(indices), n_draws)
    core = np.empty(shape)
    natural = np.empty(shape)
    persistence = np.empty(shape)
    for draw in range(n_draws):
        core[:, draw], natural[:, draw] = local_means(
            result['SD'][:, indices, draw]
```

(continues on next page)

(continued from previous page)

```
        )
        for row, date in enumerate(indices):
            covariance = innovation_covariance(
                result['HD'][date + 1, :, draw],
                result['CD'][:, draw],
            )
            persistence[row, draw] = inflation_spectrum(
                result['SD'][:, date, draw],
                covariance,
                zero_frequency,
            )[1][0]
    return {
        'core': 100 * core,
        'natural': 100 * natural,
        'persistence': persistence,
    }


historical_feature_draws = posterior_feature_draws(posterior, annual)
```

```
fig, axes = plt.subplots(3, 1, figsize=(9, 8), sharex=True)
feature_specs = (
    ('core', 'core inflation (%)'),
    ('natural', 'natural rate (%)'),
    ('persistence', r'$g_{\pi\pi}(0,t)$'),
)
for ax, (key, ylabel) in zip(axes, feature_specs):
    lower, median, upper = np.quantile(
        historical_feature_draws[key],
        (0.05, 0.5, 0.95),
        axis=1,
    )
    line, = ax.plot(data['dates'][annual], median, lw=2)
    ax.fill_between(
        data['dates'][annual],
        lower,
        upper,
        color=line.get_color(),
        alpha=0.2,
    )
    ax.set_ylabel(ylabel)
axes[-1].set_xlabel('year')
plt.tight_layout()
plt.show()
```

The solid curves are posterior medians, and the shaded regions are pointwise 90 percent intervals rather than simultaneous bands for entire paths.

The median core-inflation and persistence paths preserve the rise and post-1980 fall, but their right-skewed bands widen markedly through the 1970s and around 1980.

The natural-rate median moves more smoothly, with broad uncertainty around 1980 and again at the sample endpoint, so the direction of the historical movement is clearer than its exact magnitude.

The following plot compares the timing of core inflation and persistence.

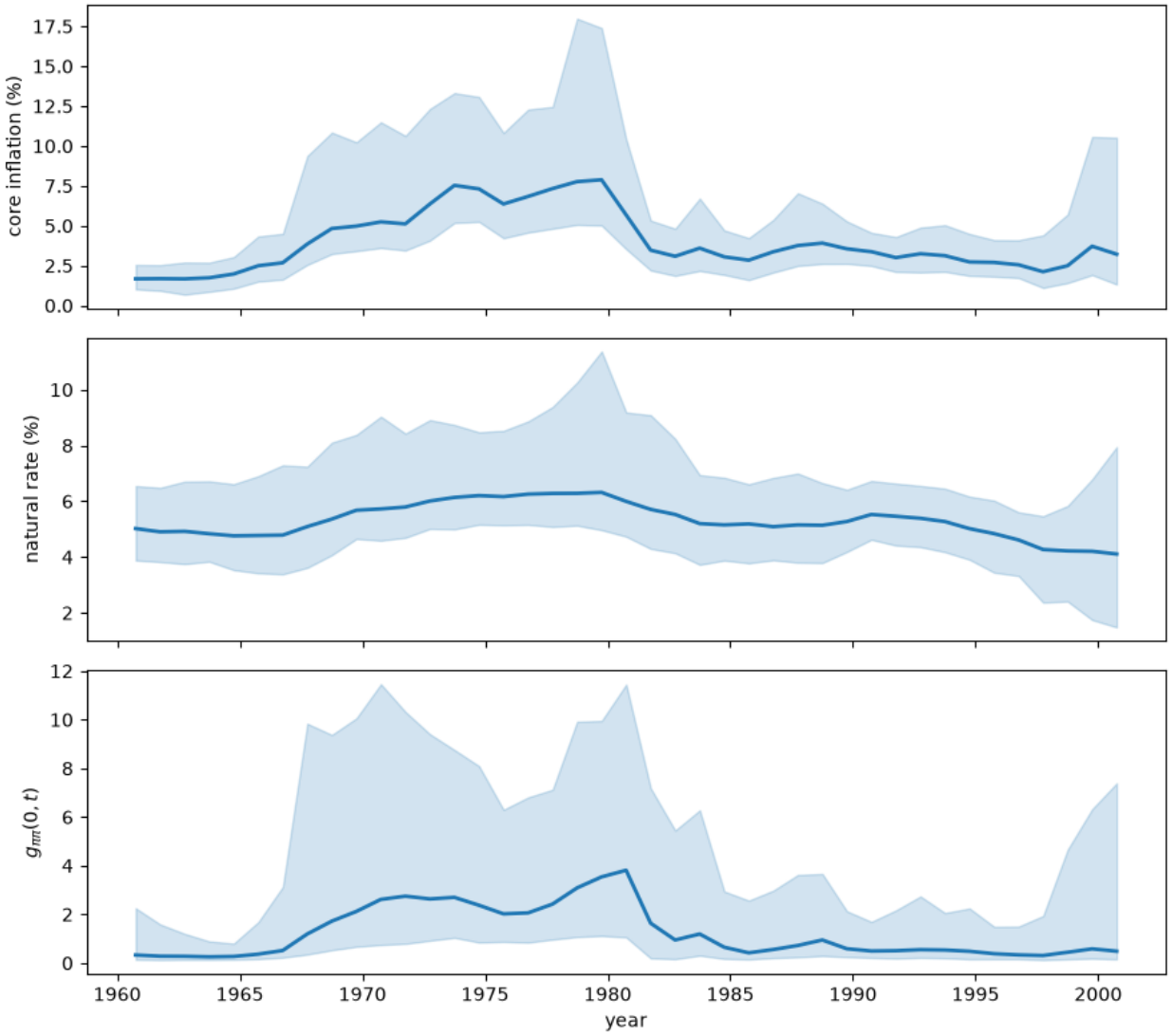


Fig. 101.9: Posterior medians and pointwise 90 percent intervals for $\bar{\pi}_t$, $\bar{u}_t$, and $g_{\pi\pi}(0,t)$

```
core_persistence_correlation = np.corrcoef(
    core_inflation, inflation_persistence
)[0, 1]

fig, ax = plt.subplots()
ax.plot(data['dates'][annual], 100 * core_inflation[annual], 'o-',
        lw=2, markersize=3, label='core inflation (%)')
ax.plot(data['dates'][annual], inflation_persistence[annual], 'x-',
        lw=2, markersize=4, label='normalized spectrum at zero')
ax.set_xlabel('year')
ax.legend()
plt.show()
```

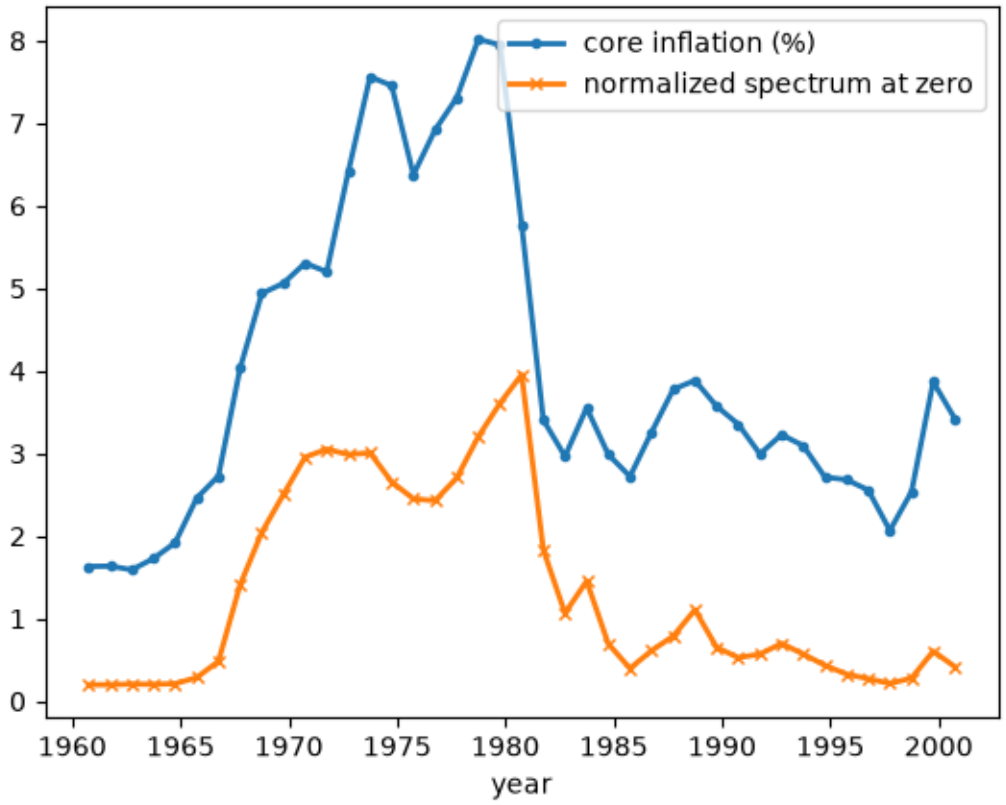


Fig. 101.10: $\bar{\pi}_t$ and $g_{\pi\pi}(0, t)$

Core inflation and normalized persistence rise together through the late 1960s and 1970s and collapse almost simultaneously after 1980, although core remains positive after persistence falls below one.

Their quarterly correlation of 0.909 summarizes common timing rather than a causal relationship because the two measures have different units and are nonlinear summaries of the same fitted coefficients.

Within the fitted TVP-VAR, the joint movement supports an important role for changing propagation as well as changing shock volatility.

A direct comparison with fixed coefficients requires fitting the restricted $Q = 0$ model.

The fall in persistence during the Volcker disinflation conflicts with escape-route models in which persistence grows along a transition from high to low inflation [Cho *et al.*, 2002, Sargent, 1999].

That tension helped motivate later learning models in which policymakers became reluctant to disinflate during the 1970s and then changed course.

101.7.5 Monetary policy activism

Cogley and Sargent summarize systematic policy with a forward-looking Taylor rule,

$$i_t = \beta_0 + \beta_1 \mathbb{E}_t \bar{\pi}_{t,t+h_\pi} + \beta_2 \mathbb{E}_t \bar{u}_{t,t+h_u} + \beta_3 i_{t-1} + \nu_t. \tag{101.10}$$

They define the activism coefficient as $\mathcal{A}_t = \beta_1/(1-\beta_3)$ and call policy active when $\mathcal{A}_t \geq 1$.

At each date, population two-stage least squares projections implied by the local VAR produce the policy-rule coefficients.

The benchmark horizons are $h_\pi = 4$ quarters and $h_u = 2$ quarters, reflecting conventional views about monetary-policy lags.

The following function projects the short rate on the model-implied inflation and unemployment forecasts using the stationary second moments of each local VAR.

```
def policy_rule_coefficients(θ, covariance, h_pi=4, h_u=2):
    """Return the local policy-rule coefficients."""
    _, companion = companion_matrix(θ)
    innovation = np.zeros((6, 6))
    innovation[:3, :3] = covariance
    stationary_covariance = linalg.solve_discrete_lyapunov(
        companion, innovation
    )
    selectors = np.eye(6)
    companion_power = np.eye(6)
    inflation_loading = np.zeros(6)
    unemployment_loading = np.zeros(6)
    for horizon in range(1, max(h_pi, h_u) + 1):
        companion_power = companion_power @ companion
        if horizon <= h_pi:
            inflation_loading += selectors[2] @ companion_power
        if horizon <= h_u:
            unemployment_loading += selectors[1] @ companion_power
    inflation_loading /= h_pi
    unemployment_loading /= h_u
    loadings = np.vstack(
        (inflation_loading, unemployment_loading, selectors[0])
    )
    regressor_covariance = loadings @ stationary_covariance @ loadings.T
    cross_covariance = (
        loadings @ stationary_covariance @ companion.T @ selectors[0]
    )
    return np.linalg.solve(regressor_covariance, cross_covariance)


policy_coefficients = np.array(
    [
        policy_rule_coefficients(θ_mean[:, t], r_mean[t])
        for t in range(len(data['dates']))
    ]
)
inflation_response = policy_coefficients[:, 0]
interest_persistence = policy_coefficients[:, 2]
```

(continues on next page)

(continued from previous page)

```
policy_margin = np.where(
    np.abs(interest_persistence) < 1,
    inflation_response + interest_persistence - 1,
    np.nan,
)
```

For $|\beta_3| < 1$, the policy margin $\mathcal{M}_t = \beta_{1t} + \beta_{3t} - 1$ is nonnegative exactly when $\mathcal{A}_t \geq 1$ and avoids division by $1 - \beta_3$.

Because β_3 multiplies the lagged interest rate, the long-run response sums $\beta_1(1 + \beta_3 + \beta_3^2 + \cdots)$ and exists only when $|\beta_3| < 1$.

The figures leave other dates blank because $\mathcal{A}_t$ has no finite long-run interpretation there.

```
fig, ax = plt.subplots()
ax.plot(data['dates'], policy_margin, lw=2)
ax.axhline(0, color='0.45', lw=1)
ax.set_xlabel('year')
ax.set_ylabel(r'policy margin $\mathcal{M}_t$')
plt.show()
```

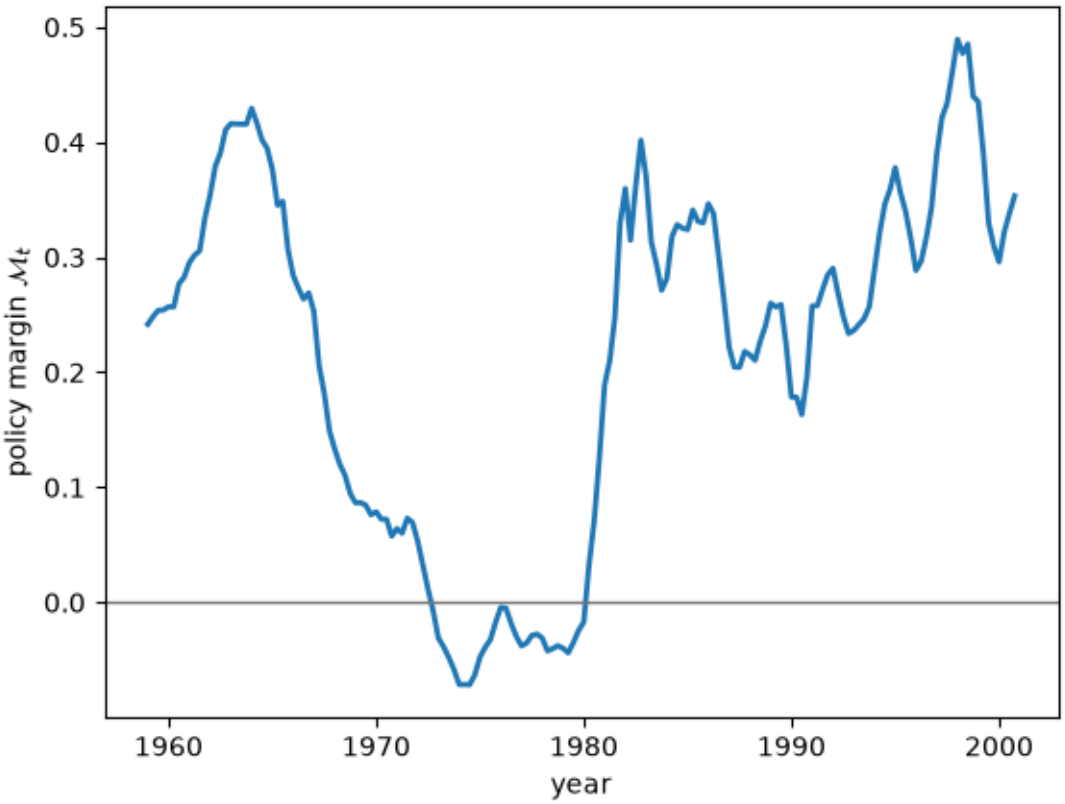


Fig. 101.11: Policy margin $\mathcal{M}_t = \beta_{1t} + \beta_{3t} - 1$

The policy margin falls below zero through much of the 1970s and then moves decisively above zero after the early 1980s.

This timing is consistent with a policy-regime contribution to the Great Inflation.

Blank intervals mark dates at which the policy margin is omitted by the rule above.

The following scatter plots use fourth-quarter observations.

```
displayed_annual = annual[np.isfinite(policy_margin[annual])]

fig, axes = plt.subplots(1, 2, figsize=(9, 4))
pairs = (
    (100 * core_inflation, 'core inflation (%)'),
    (inflation_persistence, 'normalized spectrum at zero'),
)
for ax, (feature, label) in zip(axes, pairs):
    ax.scatter(
        policy_margin[displayed_annual],
        feature[displayed_annual],
        s=18,
    )
    ax.axvline(0, color='0.45', lw=1)
    ax.set_xlabel(r'policy margin $\mathcal{M}_t$')
    ax.set_ylabel(label)
plt.tight_layout()
plt.show()
```

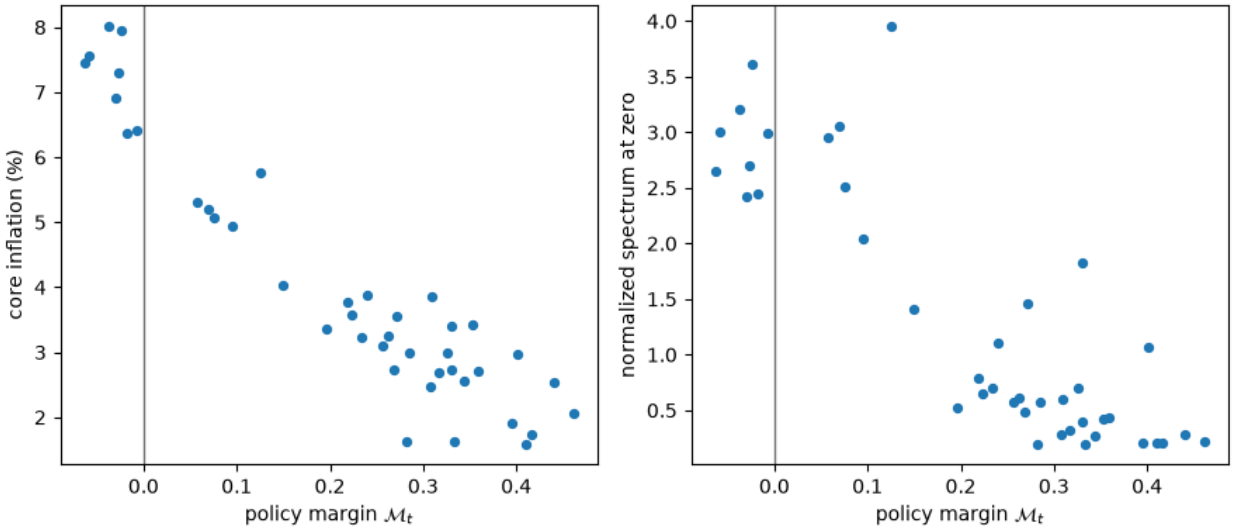


Fig. 101.12: $\mathcal{M}_t$ versus $\bar{\pi}_t$ and $g_{\pi\pi}(0, t)$

High core-inflation and persistence observations cluster near or below the zero margin, whereas positive policy margins cluster at low values of both measures.

The displayed fourth-quarter observations establish a historical association rather than a causal policy effect.

The policy-rule coefficients are weakly identified at some dates, so a path based on posterior mean inputs understates uncertainty.

We therefore calculate activism from every retained draw in 1975, 1985, and 1995.

```
selected_years = (1975, 1985, 1995)
selected_dates = [np.flatnonzero(years == year)[-1] for year in selected_years]

activism_draws = {
```

(continues on next page)

(continued from previous page)

```
    year: np.empty(posterior['SD'].shape[2]) for year in selected_years
}
stable_response_by_year = {
    year: np.empty(posterior['SD'].shape[2], dtype=bool)
    for year in selected_years
}
for draw in range(posterior['SD'].shape[2]):
    for year, date in zip(selected_years, selected_dates):
        covariance = innovation_covariance(
            posterior['HD'][date + 1, :, draw],
            posterior['CD'][:, draw],
        )
        rule = policy_rule_coefficients(
            posterior['SD'][:, date, draw], covariance
        )
        activism_draws[year][draw] = rule[0] / (1 - rule[2])
        stable_response_by_year[year][draw] = np.abs(rule[2]) < 1
```

These draws give posterior probabilities of active policy at each date and of a rise in activism after 1975, conditional on $|\beta_3| < 1$.

```
activism_events = (
    activism_draws[1975] > 1,
    activism_draws[1985] > 1,
    activism_draws[1995] > 1,
    activism_draws[1985] > activism_draws[1975],
    activism_draws[1995] > activism_draws[1975],
)
activism_conditions = (
    stable_response_by_year[1975],
    stable_response_by_year[1985],
    stable_response_by_year[1995],
    stable_response_by_year[1985] & stable_response_by_year[1975],
    stable_response_by_year[1995] & stable_response_by_year[1975],
)
assert all(condition.any() for condition in activism_conditions)
activism_probability_values = np.array(
    [
        event[condition].mean()
        for event, condition in zip(activism_events, activism_conditions)
    ]
)

activism_probability_index = (
    'P(A_1975 > 1)',
    'P(A_1985 > 1)',
    'P(A_1995 > 1)',
    'P(A_1985 > A_1975)',
    'P(A_1995 > A_1975)',
)
activism_probabilities = pd.DataFrame(
    {'conditional estimate': activism_probability_values},
    index=activism_probability_index,
)
stable_response_shares = pd.Series(
    {year: draws.mean() for year, draws in stable_response_by_year.items()},
    name='share with |beta_3| < 1',
```

(continues on next page)

(continued from previous page)

```
)

display(activism_probabilities.round(3))
stable_response_shares.to_frame().round(3)
```

```
                    conditional estimate
P(A_1975 > 1)                      0.310
P(A_1985 > 1)                      0.910
P(A_1995 > 1)                      0.941
P(A_1985 > A_1975)                 0.912
P(A_1995 > A_1975)                 0.943
```

```
      share with |beta_3| < 1
1975                    1.000
1985                    0.932
1995                    0.980
```

The first three probabilities condition on $|\beta_3| < 1$ at that date, while the comparisons require this condition at both dates.

The central draw distributions expose the overlap and skewness behind the probability estimates.

```
stable_activism_draws = {
    year: activism_draws[year][stable_response_by_year[year]]
    for year in selected_years
}
pooled_activism = np.concatenate(tuple(stable_activism_draws.values()))
activism_limits = np.quantile(pooled_activism, (0.05, 0.95))
activism_bins = np.linspace(*activism_limits, 31)

fig, ax = plt.subplots()
for year in selected_years:
    central_draws = stable_activism_draws[year]
    central_draws = central_draws[
        (central_draws >= activism_limits[0])
        & (central_draws <= activism_limits[1])
    ]
    ax.hist(
        central_draws,
        bins=activism_bins,
        histtype='step',
        lw=2,
        label=str(year),
    )
ax.axvline(1, color='0.45', lw=1)
ax.set_xlabel('activism coefficient')
ax.set_ylabel('retained draws')
ax.legend()
plt.show()
```

The 1975 distribution is concentrated around or below one, while the 1985 and 1995 distributions shift substantially to the right but remain broad, skewed, and overlapping.

The posterior therefore favors a passive-to-active shift after 1975 but does not sharply distinguish 1985 from 1995.

The figure plots draws with $|\beta_3| < 1$ inside the pooled 5th and 95th percentiles, while the probability calculations use every such draw.

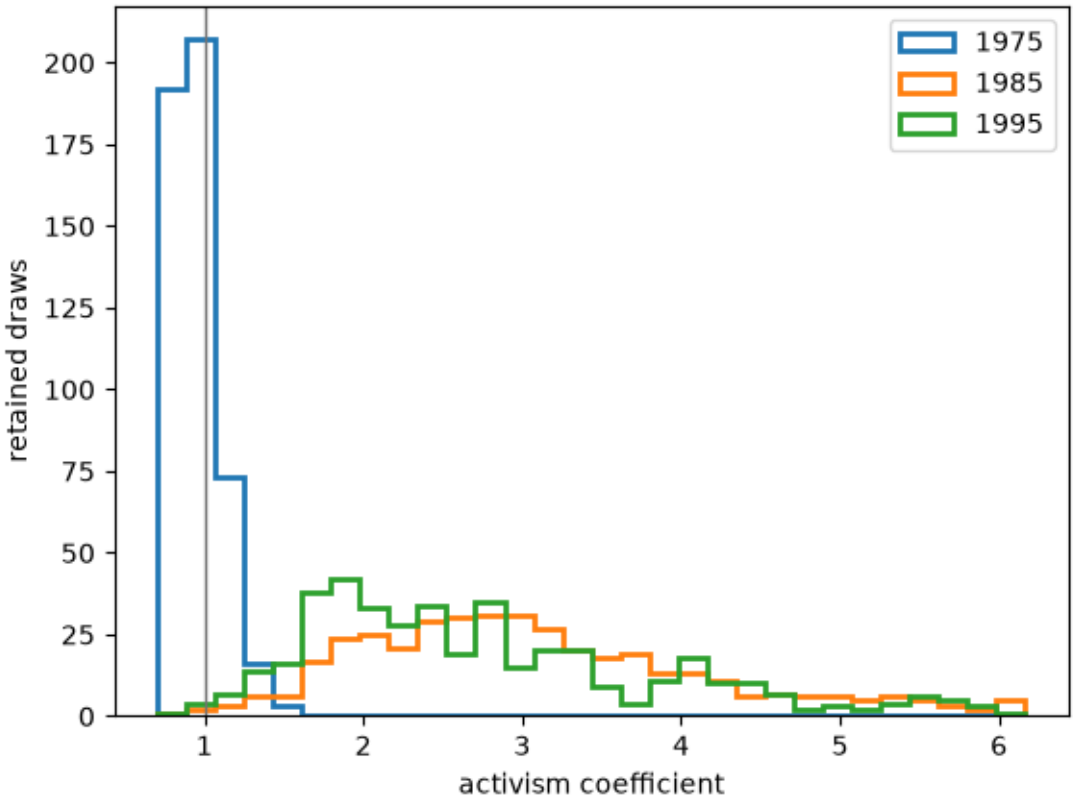


Fig. 101.13: Central posterior draws of $\mathcal{A}_t$ conditional on $|\beta_3| < 1$ in 1975, 1985, and 1995

101.8 Another quarter-century of evidence

The sample ends in 2000Q4, so it misses the financial crisis, the zero-interest-rate period, the pandemic, and the 2021–2022 inflation surge.

The question is whether these episodes alter the earlier evidence about drift, volatility, inflation persistence, and systematic policy.

101.8.1 The new observations

To examine those observations, we append current data after 2000Q4 while leaving pre-2000Q4 sample unchanged.

This splice prevents revisions to pre-2001 CPI and unemployment data from being mistaken for information in the additional quarter-century.

We download seasonally adjusted CPI, seasonally adjusted unemployment, and the three-month Treasury-bill rate from FRED.

The transformations and within-quarter timing remain unchanged: CPI comes from the third month, unemployment is a three-month average, and the interest rate comes from the first month.

```
fred_url = (
    'https://fred.stlouisfed.org/graph/fredgraph.csv?'
    'id=CPIAUCSL%2CUNRATE%2CTB3MS'
)
fred_monthly = pd.read_csv(
    fred_url,
    parse_dates=['observation_date'],
).set_index('observation_date')
```

The BLS notes that the October 2025 unemployment observation was not collected during the federal government shutdown, leaving 2025Q4 without a complete three-month average.

The code below therefore ends the updated sample at the last quarter whose three monthly unemployment readings are all present, detected automatically rather than hard-coded; at the time of writing that quarter is 2025Q3.

```
def fred_quarterly_table(unemployment_monthly):
    """Construct transformed quarterly observations from current FRED data."""
    interest = fred_monthly.loc[
        fred_monthly.index.month.isin((1, 4, 7, 10)), 'TB3MS'
    ].copy()
    interest.index = interest.index.to_period('Q').start_time

    cpi = fred_monthly.loc[
        fred_monthly.index.month.isin((3, 6, 9, 12)), 'CPIAUCSL'
    ].copy()
    cpi.index = cpi.index.to_period('Q').start_time

    unemployment = unemployment_monthly.resample('QS').mean()
    quarterly = pd.concat(
        {
            'interest': interest,
            'unemployment': unemployment,
            'cpi': cpi,
        },
        axis=1,
    )
```

(continues on next page)

(continued from previous page)

```
    quarterly['y3'] = np.log1p(quarterly['interest'] / 400)
    quarterly['ur'] = quarterly['unemployment'] / 100
    quarterly['dp'] = np.log(quarterly['cpi']).diff()
    quarterly['date'] = (
        quarterly.index.year + (quarterly.index.quarter - 1) / 4
    )
    columns = ['date', 'y3', 'ur', 'dp']
    return quarterly.loc[:, columns].dropna()


unemployment_monthly = fred_monthly['UNRATE']
unemployment_counts = unemployment_monthly.resample('QS').count()
latest_quarterly = fred_quarterly_table(unemployment_monthly)

# Extend the archived sample dynamically: append post-2000Q4 quarters only up
# to the first one missing any of its three monthly unemployment readings, so
# no endpoint is hard-coded.  An isolated missing month -- for example October
# 2025, which the federal shutdown left uncollected -- caps the sample at the
# preceding complete quarter, and a normally incomplete current quarter caps it
# at the last finished one.
quarterly_unfilled = latest_quarterly.loc['2001-01-01':]
month_counts = unemployment_counts.reindex(quarterly_unfilled.index)
incomplete_quarters = month_counts.index[month_counts < 3]
if len(incomplete_quarters):
    first_incomplete_quarter = incomplete_quarters[0]
    complete_extension = quarterly_unfilled.loc[
        quarterly_unfilled.index < first_incomplete_quarter
    ]
else:
    complete_extension = quarterly_unfilled

cs_sample = pd.read_csv(data_url)
overlap_date = pd.Timestamp('2000-10-01')
cs_sample_overlap = cs_sample.iloc[-1]
latest_overlap = latest_quarterly.loc[overlap_date]


def scaled_observation(row):
    """Return observable units for a transformed quarterly row."""
    return pd.Series(
        {
            'interest rate (annual %)': 400 * np.expm1(row['y3']),
            'unemployment (%)': 100 * row['ur'],
            'inflation (annual %)': 400 * row['dp'],
        }
    )


cs_sample_scaled = scaled_observation(cs_sample_overlap)
latest_scaled = scaled_observation(latest_overlap)
splice_audit = pd.DataFrame(
    {
        'Cogley-Sargent (2005) 2000Q4': cs_sample_scaled,
        'latest revised 2000Q4 data': latest_scaled,
        'current minus Cogley-Sargent (2005)': (
            latest_scaled - cs_sample_scaled
        ),
```

(continues on next page)

(continued from previous page)

```
        'first appended 2001Q1': scaled_observation(
            complete_extension.iloc[0]
        ),
    }
)
extended_observations = pd.concat(
    (cs_sample, complete_extension.reset_index(drop=True)),
    ignore_index=True,
)


def quarter_label(timestamp):
    """Format a timestamp as year and quarter."""
    return str(timestamp.to_period('Q'))


display(splice_audit.round(3))
```

```
                          Cogley-Sargent (2005) 2000Q4  \
interest rate (annual %)                          6.253
unemployment (%)                                  3.967
inflation (annual %)                              2.298

                          latest revised 2000Q4 data  \
interest rate (annual %)                       6.110
unemployment (%)                               3.900
inflation (annual %)                           2.298

                          current minus Cogley-Sargent (2005)  \
interest rate (annual %)                                -0.143
unemployment (%)                                        -0.067
inflation (annual %)                                    -0.000

                          first appended 2001Q1
interest rate (annual %)                  5.150
unemployment (%)                          4.233
inflation (annual %)                      3.422
```

No missing value is filled or otherwise treated as observed.

The overlap table shows any break created by joining the archived data to the newly downloaded data.

The appended 2001Q1 inflation rate uses the latest revised CPI level for both 2000Q4 and 2001Q1, so one log difference never combines observations from two data releases.

```
fig, axes = plt.subplots(3, 1, figsize=(9, 7), sharex=True)
extended_plot = extended_observations[
    extended_observations['date'] >= 1959
]
updated_sample_end = complete_extension['date'].iloc[-1]
axes[0].plot(
    extended_plot['date'],
    400 * extended_plot['dp'],
    lw=2,
)
axes[0].set_ylabel('inflation (annual %)')
axes[1].plot(
```

(continues on next page)

(continued from previous page)

```
    extended_plot['date'],
    100 * extended_plot['ur'],
    lw=2,
)
axes[1].set_ylabel('unemployment (%)')
axes[2].plot(
    extended_plot['date'],
    400 * np.expm1(extended_plot['y3']),
    lw=2,
)
axes[2].set_ylabel('interest (annual %)')
axes[2].set_xlabel('year')
for index, ax in enumerate(axes):
    cs_sample_label = 'Cogley-Sargent (2005)' if index == 0 else None
    sample_end_label = 'updated sample end' if index == 0 else None
    ax.axvline(
        2000.75,
        color='0.65',
        ls='--',
        lw=1,
        label=cs_sample_label,
    )
    ax.axvline(
        updated_sample_end,
        color='0.45',
        ls=':',
        lw=1,
        label=sample_end_label,
    )
axes[0].legend()
plt.tight_layout()
plt.show()
```

```
endpoint_observation = complete_extension.iloc[-1]
pd.Series(
    {
        'annualized quarterly inflation (%)': (
            400 * endpoint_observation['dp']
        ),
        'unemployment (%)': 100 * endpoint_observation['ur'],
        'three-month interest rate (%)': (
            400 * np.expm1(endpoint_observation['y3'])
        ),
    },
    name=quarter_label(complete_extension.index[-1]),
).to_frame().round(3)
```

```
                                  2025Q3
annualized quarterly inflation (%)  3.482
unemployment (%)                    4.333
three-month interest rate (%)       4.250
```

The extension adds the financial-crisis contraction, a pandemic quarterly unemployment spike to 13.0 percent, and a short 2021–2022 inflation surge alongside two long stretches of near-zero interest rates.

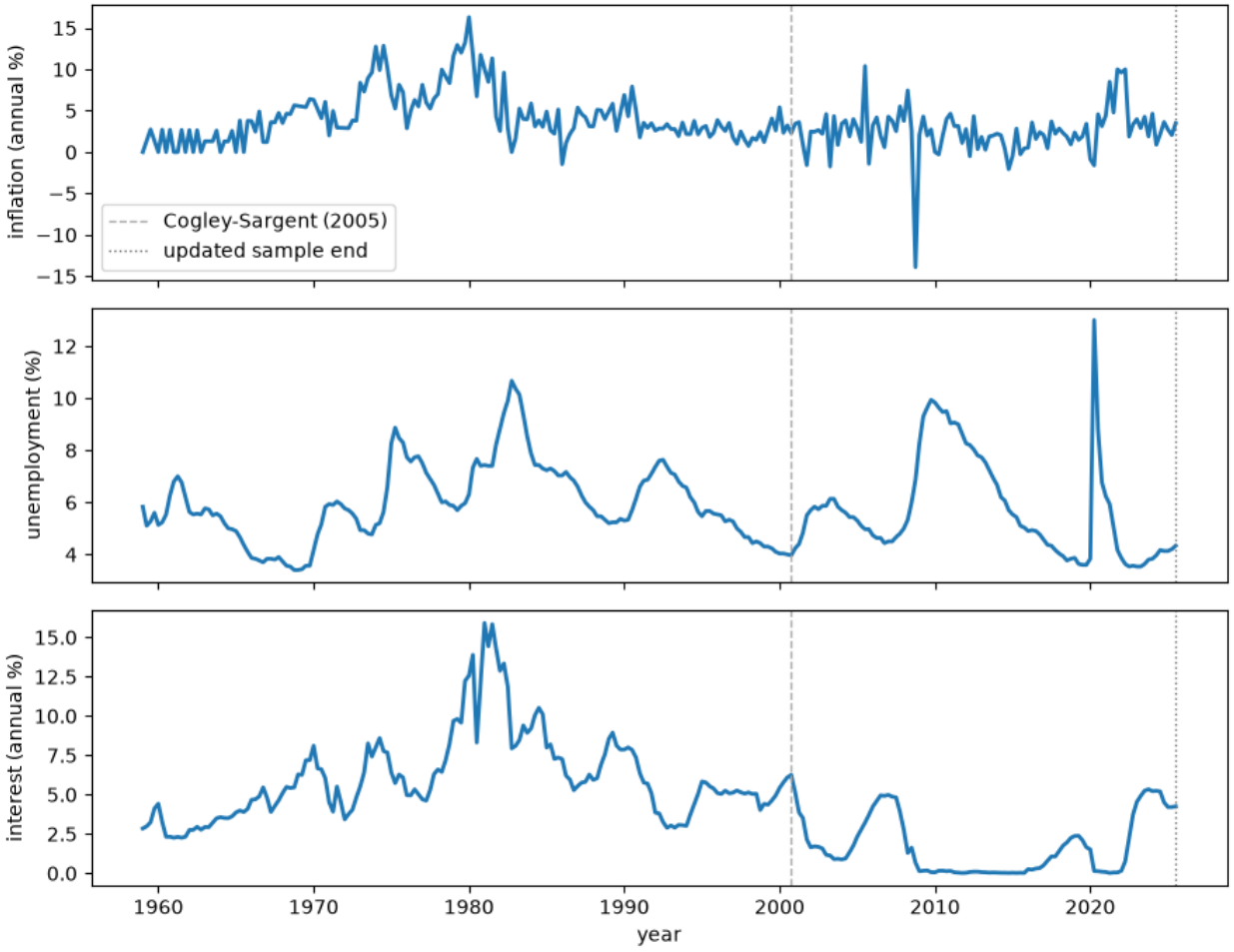


Fig. 101.14: Observed π_t, u_t, and i_t, 1959Q1–2025Q3

Because inflation is annualized from quarterly changes, isolated movements look especially large in this panel, but the recent surge is still visibly much shorter than the sustained 1970s rise.

These observations provide a demanding test of whether the model assigns recent extremes to shock volatility or persistent dynamics, while the near-zero rate also weakens short-rate measures of policy after 2008.

We fit the stable TVP-VAR through 2025Q3, the last complete quarter, using the Treasury-bill measure above.

```
def append_extension(extension):
    """Append a transformed FRED extension to the Cogley-Sargent sample."""
    extension = extension.reset_index(drop=True)
    table = pd.concat((cs_sample, extension), ignore_index=True)
    assert table['date'].is_unique
    assert np.allclose(np.diff(table['date']), 0.25)
    return table


def fit_updated_model(table):
    """Calibrate and fit the stable drifting VAR to one table."""
    model_data = prepare_data(table)
    model_prior = calibrate_prior(model_data)
    result = run_sampler(
        model_data['y'],
        model_data['x'],
        model_prior,
        n_sweeps=5_000,
        burn=2_500,
        thin=5,
        seed=42,
        warmup=500,
        stable=True,
        progress_every=500,
    )
    validate_posterior_arrays(result, len(model_data['dates']))
    trace = np.trace(result['QD'], axis1=0, axis2=1)
    return {
        'data': model_data,
        'prior': model_prior,
        'posterior': result,
        'trace': trace,
    }


updated_fit = fit_updated_model(append_extension(complete_extension))

latest_data_table = latest_quarterly.loc[
    (latest_quarterly.index >= pd.Timestamp('1948-04-01'))
    & (latest_quarterly.index <= complete_extension.index[-1])
].reset_index(drop=True)
assert np.allclose(np.diff(latest_data_table['date']), 0.25)
latest_data_fit = fit_updated_model(latest_data_table)
```

```
500/5,000 sweeps; 907 slice contractions; 0.9 minutes
1,000/5,000 sweeps; 2,157 slice contractions; 1.8 minutes
1,500/5,000 sweeps; 3,239 slice contractions; 2.7 minutes
2,000/5,000 sweeps; 5,002 slice contractions; 3.6 minutes
2,500/5,000 sweeps; 6,186 slice contractions; 4.5 minutes
3,000/5,000 sweeps; 7,163 slice contractions; 5.4 minutes
```

(continues on next page)

(continued from previous page)

```
3,500/5,000 sweeps; 7,943 slice contractions; 6.3 minutes
4,000/5,000 sweeps; 8,577 slice contractions; 7.2 minutes
4,500/5,000 sweeps; 9,404 slice contractions; 8.2 minutes
5,000/5,000 sweeps; 10,158 slice contractions; 9.1 minutes
500/5,000 sweeps; 862 slice contractions; 1.0 minutes
1,000/5,000 sweeps; 2,000 slice contractions; 2.0 minutes
1,500/5,000 sweeps; 3,099 slice contractions; 2.9 minutes
2,000/5,000 sweeps; 4,751 slice contractions; 3.9 minutes
2,500/5,000 sweeps; 6,298 slice contractions; 4.9 minutes
3,000/5,000 sweeps; 7,542 slice contractions; 5.9 minutes
3,500/5,000 sweeps; 8,617 slice contractions; 6.8 minutes
4,000/5,000 sweeps; 9,134 slice contractions; 7.7 minutes
4,500/5,000 sweeps; 9,568 slice contractions; 8.6 minutes
5,000/5,000 sweeps; 10,764 slice contractions; 9.5 minutes
```

101.8.2 Did coefficient drift continue?

```
def updated_drift_summary(fit):
    """Summarize the updated coefficient-drift distribution."""
    trace = fit['trace']
    q_mean = fit['posterior']['QD'].mean(axis=2)
    eigenvalues = np.linalg.eigvalsh(q_mean)[::-1]
    return {
        'posterior mean tr(Q)': trace.mean(),
        'share in first three eigen-directions': (
            eigenvalues[:3].sum() / eigenvalues.sum()
        ),
    }


updated_label = quarter_label(complete_extension.index[-1])
updated_summary = pd.DataFrame(
    {
        'Cogley-Sargent (2005) + extension': (
            updated_drift_summary(updated_fit)
        ),
        'latest revised data for full sample': updated_drift_summary(
            latest_data_fit
        ),
    }
)
updated_summary.round(3)
```

```
                                    Cogley-Sargent (2005) + extension  \
posterior mean tr(Q)                                                0.073
share in first three eigen-directions                               0.976

                                    latest revised data for full sample
posterior mean tr(Q)                                                 0.034
share in first three eigen-directions                                0.955
```

The drift-rate distributions and coefficient paths provide the same views used for the Cogley and Sargent [2005] sample.

The dashed vertical line in updated time-series figures marks the Cogley and Sargent [2005] sample's 2000Q4 endpoint, but each updated path comes from a full re-estimation rather than from attaching new points to an unchanged historical

estimate.

```
fig, ax = plt.subplots()
ax.hist(updated_fit['trace'], bins=30, histtype='step', lw=2)
ax.axvline(
    np.trace(updated_fit['prior']['q_center']),
    color='C1',
    lw=2,
    label=r'prior $\mathrm{tr}(\bar Q)$',
)
ax.set_xlabel(r'$\mathrm{tr}(Q)$')
ax.set_ylabel('frequency')
ax.legend()
plt.show()
```

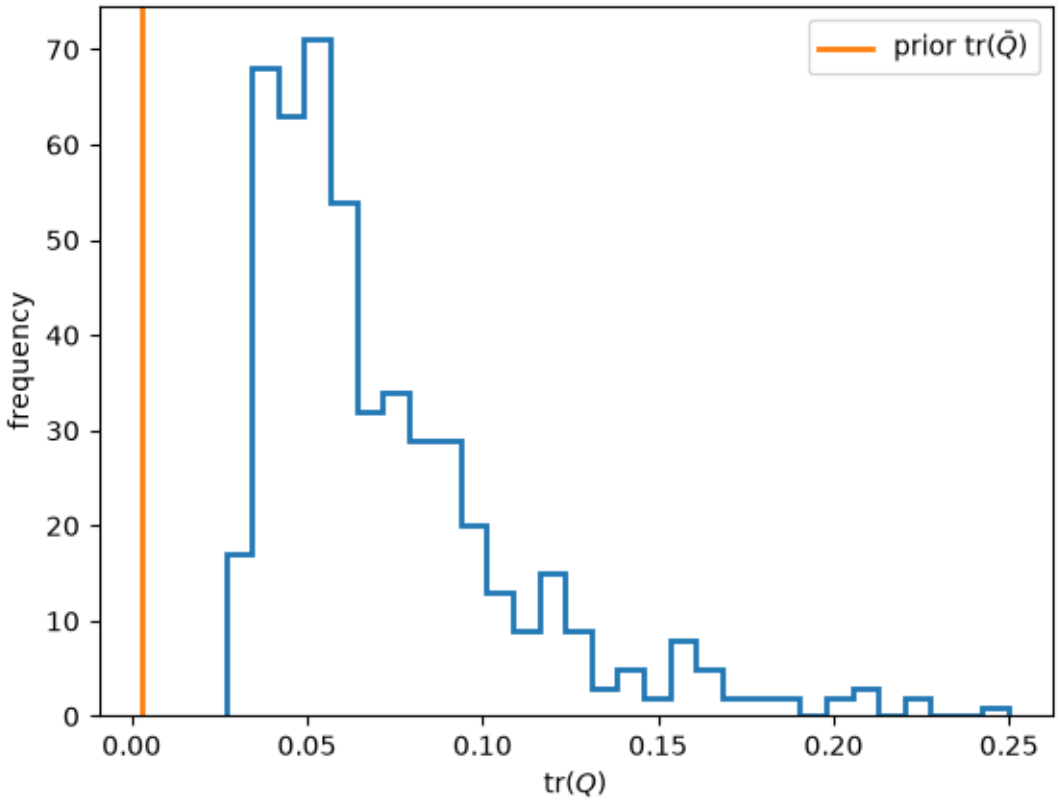


Fig. 101.15: Posterior tr(Q) and prior tr$(\bar{Q})$ through 2025Q3

In both data constructions, the full-sample posterior drift rate remains above its conservative prior calibration.

Within the TVP-VAR this indicates non-negligible full-sample drift, but it is not a formal comparison of $Q = 0$ with $Q > 0$.

Its magnitude depends on whether the historical observations come from the archived dataset or the latest revisions.

Because Q is a single variance parameter for the full 1959–2025 path, this histogram does not by itself show that drift accelerated after 2000.

```
fig, axes = plt.subplots(1, 3, figsize=(10, 4), sharex=True)
updated_dates = updated_fit['data']['dates']
updated_coefficients = updated_fit['posterior']['SD'].mean(axis=2)
coefficient_lines = plot_equation_coefficients(
    axes,
    updated_dates,
    updated_coefficients,
)
for ax in axes:
    ax.axvline(2000.75, color='0.65', ls='--', lw=1)
fig.legend(
    coefficient_lines,
    coefficient_labels,
    loc='lower center',
    ncol=4,
)
plt.tight_layout(rect=(0, 0.17, 1, 1))
plt.show()
```

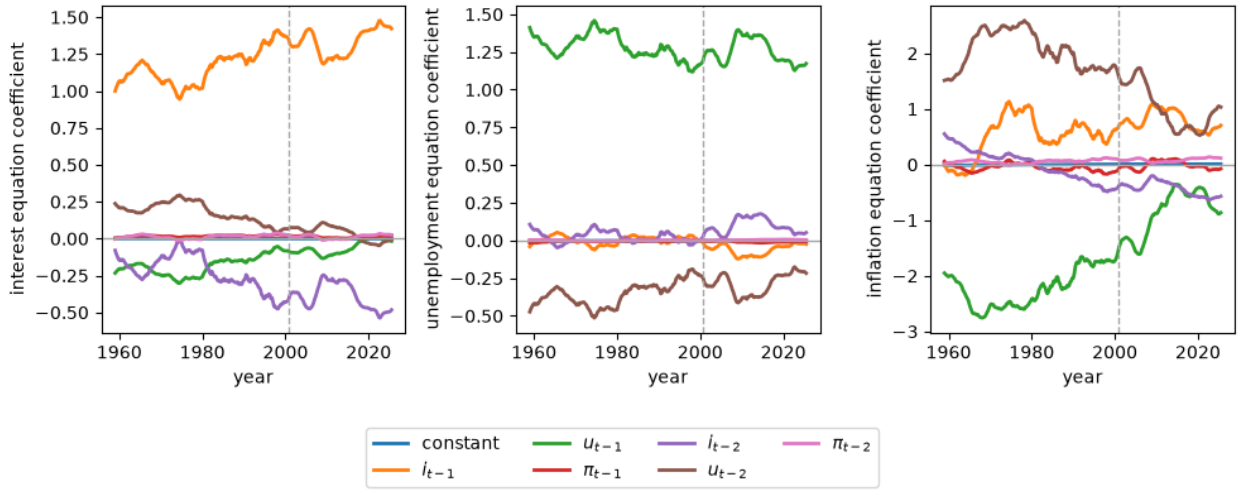


Fig. 101.16: Posterior mean VAR coefficients $E(\theta_t \mid T)$ through 2025Q3

Several coefficient paths continue moving gradually after 2000, with the largest changes again concentrated in the inflation equation rather than appearing as abrupt financial-crisis or pandemic breaks.

The opposing movements of some paired lag coefficients also show why their combined dynamic implications are more informative than any single line.

The first three eigen-directions account for the large majority of the posterior mean drift variance, so the estimated movement remains low dimensional.

The stability restriction now applies to a longer path, which makes the posterior drift rate a property of the full 1959–2025 sample.

101.8.3 Volatility after the Great Moderation

We next separate changes in the sizes and correlations of innovations from changes in the VAR dynamics.

```
def model_features(fit):
    """Compute features at posterior mean parameters."""
    result = fit['posterior']
    θ = result['SD'].mean(axis=2)
    mean_root_modulus = np.max(
        np.abs(companion_roots(θ)), axis=1
    )
    if not np.all(mean_root_modulus < 1):
        raise ValueError('mean coefficient path is unstable')
    covariance = mean_innovation_covariance(result['HD'], result['CD'])
    core, natural = local_means(θ)
    persistence = np.array([
        inflation_spectrum(
            θ[:, t], covariance[t], zero_frequency
        )[1][0]
        for t in range(len(fit['data']['dates']))
    ])
    sign, logdet = np.linalg.slogdet(covariance)
    assert np.all(sign > 0)
    policy_coefficients = np.array(
        [
            policy_rule_coefficients(θ[:, t], covariance[t])
            for t in range(len(fit['data']['dates']))
        ]
    )
    inflation_response = policy_coefficients[:, 0]
    interest_persistence = policy_coefficients[:, 2]
    policy_margin = np.where(
        np.abs(interest_persistence) < 1,
        inflation_response + interest_persistence - 1,
        np.nan,
    )
    return {
        'θ': θ,
        'maximum_companion_root': mean_root_modulus.max(),
        'covariance': covariance,
        'core': core,
        'natural': natural,
        'persistence': persistence,
        'logdet': logdet,
        'policy_margin': policy_margin,
    }


updated_features = model_features(updated_fit)
latest_data_features = model_features(latest_data_fit)

pd.Series(
    {
        'Cogley-Sargent (2005) + extension': (
            updated_features['maximum_companion_root']
        ),
        'latest revised data for full sample': (
            latest_data_features['maximum_companion_root']
```

(continues on next page)

(continued from previous page)

```
        ),
    },
    name='maximum companion-root modulus of mean path',
).to_frame().round(4)
```

	maximum companion-root modulus of mean path
Cogley-Sargent (2005) + extension	0.9569
latest revised data for full sample	0.9613

```
fig, axes = plt.subplots(3, 2, figsize=(9, 8), sharex=True)
dates = updated_fit['data']['dates']
covariance = updated_features['covariance']
for row, (index, _) in enumerate(variances):
    axes[row, 0].plot(
        dates,
        10000 * np.sqrt(covariance[:, index, index]),
        lw=2,
    )
for row, (left, right, _) in enumerate(correlations):
    scale = np.sqrt(
        covariance[:, left, left] * covariance[:, right, right]
    )
    axes[row, 1].plot(
        dates,
        covariance[:, left, right] / scale,
        lw=2,
    )
for row, (_, label) in enumerate(variances):
    axes[row, 0].set_title(label)
for row, (_, _, label) in enumerate(correlations):
    axes[row, 1].set_title(label)
for ax in axes.flat:
    ax.axvline(2000.75, color='0.65', ls='--', lw=1)
axes[1, 0].set_ylabel(
    r'innovation standard deviation $\times 10^4$'
)
axes[1, 1].set_ylabel('correlation')
axes[-1, 0].set_xlabel('year')
axes[-1, 1].set_xlabel('year')
plt.tight_layout()
plt.show()
```

The Volcker transition still dominates interest-rate innovation volatility, the financial crisis produces the largest inflation-volatility spike, and the pandemic uniquely dominates unemployment volatility.

The pandemic also drives a sharp fall in the inflation–unemployment correlation, so recent episodes changed the mix of reduced-form shocks as well as their size.

This is direct evidence for a bad-luck component, although reduced-form innovations do not establish that the underlying disturbances were structurally exogenous.

```
fig, ax = plt.subplots()
ax.plot(
    updated_fit['data']['dates'],
    updated_features['logdet'],
```

(continues on next page)

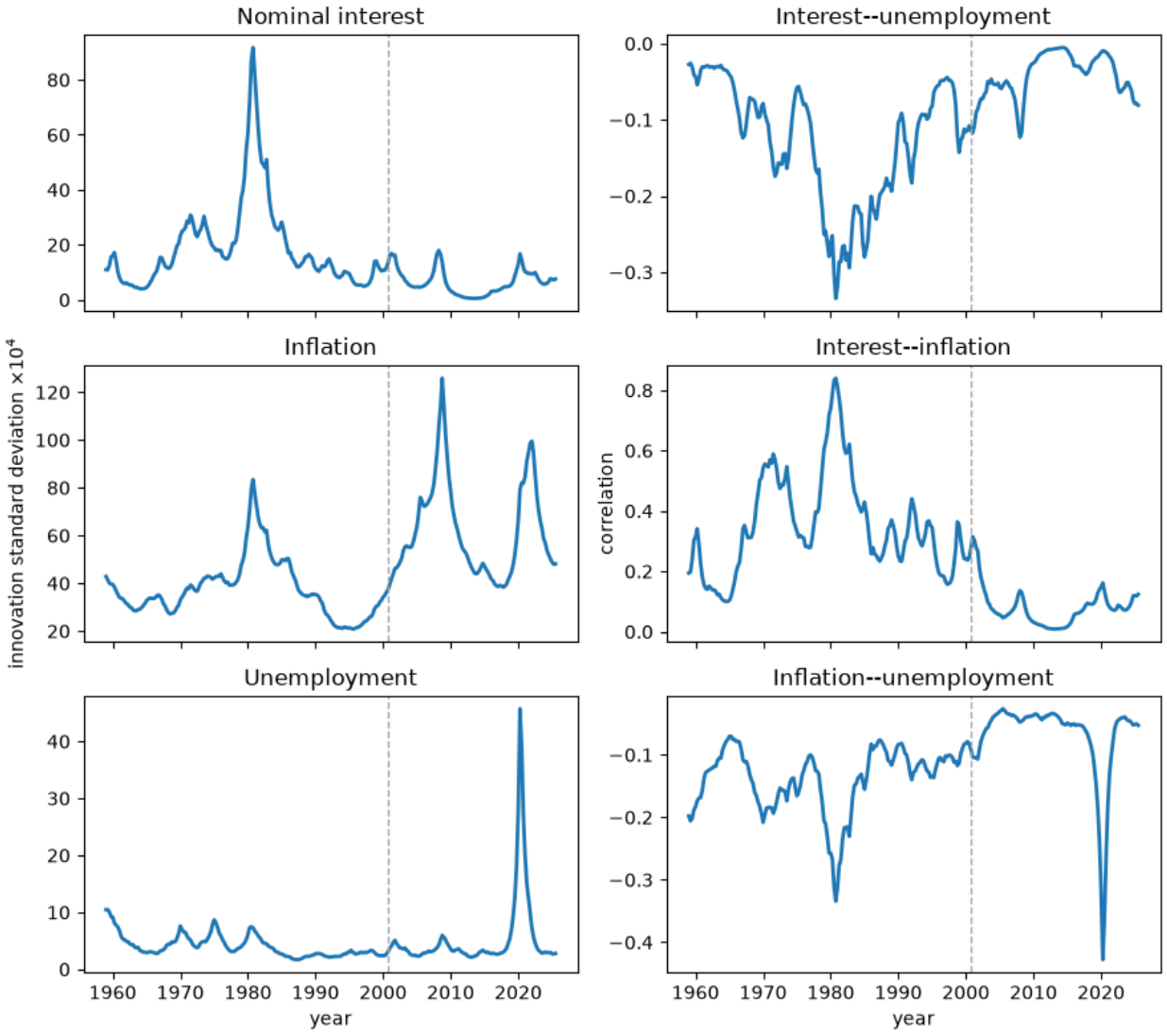


Fig. 101.17: Standard deviations and correlations implied by $E(R_t \mid T)$ through 2025Q3

(continued from previous page)

```
    lw=2,
)
ax.axvline(2000.75, color='0.65', ls='--', lw=1)
ax.set_xlabel('year')
ax.set_ylabel(r'$\log |E(R_t\mid T)|$')
plt.show()
```

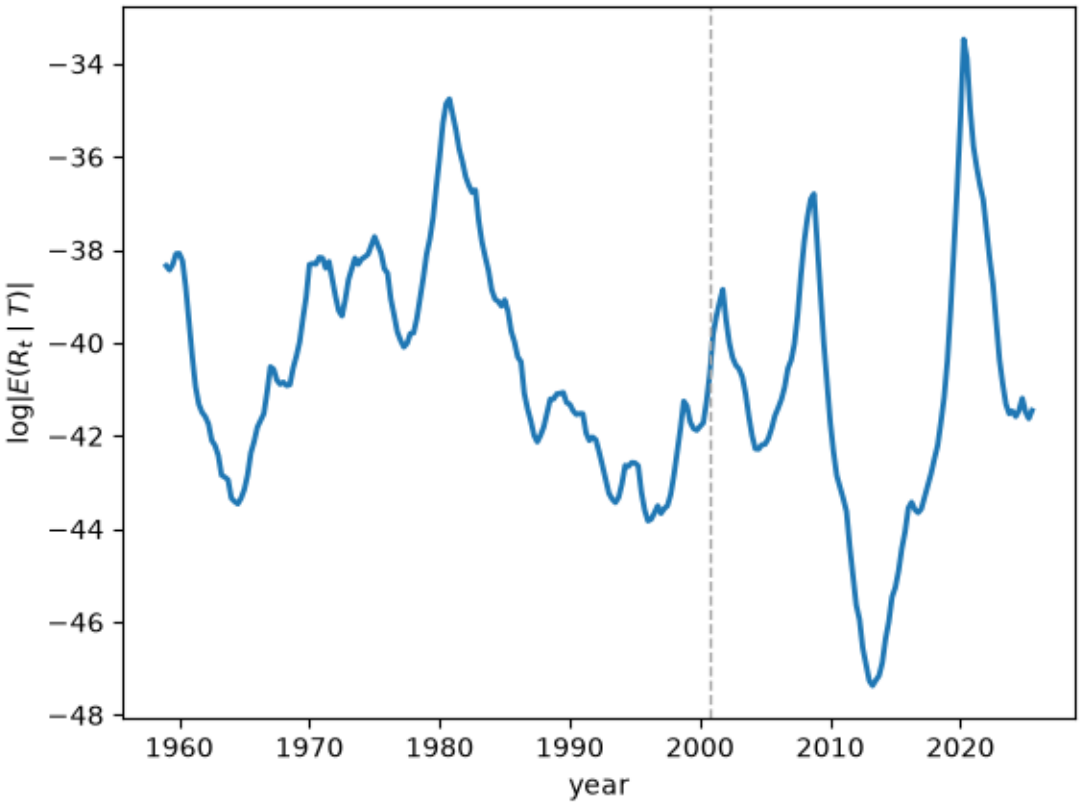


Fig. 101.18: $\log |E(R_t \mid T)|$ through 2025Q3

Joint innovation variance rises during the financial crisis, reaches a deep Great Moderation trough in the 2010s, and then jumps in 2020 above even its 1981 peak before falling rapidly.

```
def decimal_quarter_label(value):
    """Format a decimal quarterly date."""
    year = int(np.floor(value + 1e-8))
    quarter = int(round(4 * (value - year))) + 1
    return f'{year}Q{quarter}'


def volatility_summary(fit, features):
    """Summarize innovation-volatility peaks and endpoints."""
    dates = fit['data']['dates']
    standard_deviation = 10000 * np.sqrt(
        np.diagonal(features['covariance'], axis1=1, axis2=2)
    )
```

(continues on next page)

(continued from previous page)

```
    names = ('interest', 'unemployment', 'inflation')
    return pd.DataFrame(
        {
            'peak quarter': [
                decimal_quarter_label(
                    dates[np.argmax(standard_deviation[:, index])]
                )
                for index in range(n_variables)
            ],
        },
        index=names,
    )


updated_volatility = volatility_summary(updated_fit, updated_features)
updated_volatility
```

```
             peak quarter
interest           1980Q4
unemployment       2020Q2
inflation          2008Q4
```

All three innovation standard deviations are well below their peaks at the 2025Q3 endpoint, which supports a large but transient pandemic-shock interpretation inside this model.

101.8.4 Core inflation and the natural rate after 2000

The same local-mean calculation distinguishes temporary inflation from a shift in the model's long-horizon forecast.

```
updated_dates = updated_fit['data']['dates']
updated_annual = annual_indices(updated_dates)
fig, ax = plt.subplots()
ax.plot(
    updated_dates[updated_annual],
    100 * updated_features['core'][updated_annual],
    'o-',
    lw=2,
    markersize=3,
    label='core inflation',
)
ax.plot(
    updated_dates[updated_annual],
    100 * updated_features['natural'][updated_annual],
    '+-',
    lw=2,
    markersize=5,
    label='natural rate',
)
ax.axvline(2000.75, color='0.65', ls='--', lw=1)
ax.set_xlabel('year')
ax.set_ylabel('percent')
ax.legend()
plt.show()
```

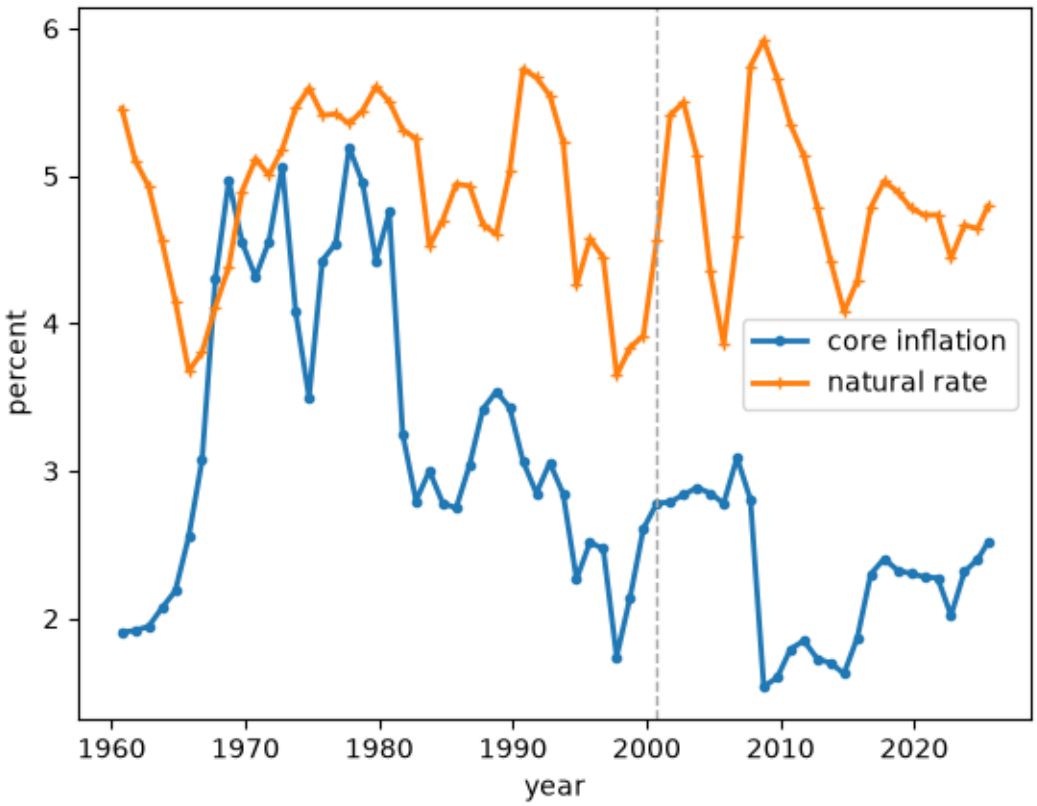


Fig. 101.19: Local means $\bar{\pi}_t$ and $\bar{u}_t$ through 2025Q3

The 1970s core-inflation peak is lower here than in the Cogley and Sargent [2005] figure because later observations revise the smoothed history, so values on both sides of the dashed line belong to one updated fit.

After 2000 the core rate stays mostly between about 2 and 3 percent and rises only modestly after 2020, even though observed inflation moves much more sharply.

The natural-rate line is the model's locally implied long-run unemployment anchor, which is why quarterly unemployment can jump to 13.0 percent in 2020 while this line remains near 5 percent.

The monthly peak was 14.8 percent.

The final annual point represents 2025Q3 because the sample ends before the fourth quarter.

```
def endpoint_features(features):
    """Return economically scaled endpoint features."""
    return pd.Series(
        {
            'core inflation (%)': 100 * features['core'][-1],
            'natural rate (%)': 100 * features['natural'][-1],
            'normalized persistence': features['persistence'][-1],
            'log generalized innovation variance': features['logdet'][-1],
        }
    )


updated_endpoints = endpoint_features(updated_features).to_frame(
    name=updated_label
).T
latest_data_endpoints = endpoint_features(
    latest_data_features
).to_frame(name=updated_label).T
data_revision_sensitivity = pd.concat(
    {
        'Cogley-Sargent (2005) + extension': updated_endpoints,
        'latest revised data for full sample': latest_data_endpoints,
    }
)
data_revision_sensitivity.round(3)
```

```
                                          core inflation (%)  \
Cogley-Sargent (2005) + extension   2025Q3               2.514
latest revised data for full sample 2025Q3               2.387

                                          natural rate (%)  \
Cogley-Sargent (2005) + extension   2025Q3             4.794
latest revised data for full sample 2025Q3             4.539

                                          normalized persistence  \
Cogley-Sargent (2005) + extension   2025Q3                   0.304
latest revised data for full sample 2025Q3                   0.250

                                          log generalized innovation variance
Cogley-Sargent (2005) + extension   2025Q3                             -41.454
latest revised data for full sample 2025Q3                             -41.075
```

The endpoint core-inflation, natural-rate, and persistence summaries are similar across the two data constructions.

The draw-wise annual paths show how uncertainty evolves inside the updated fit.

```
updated_feature_draws = posterior_feature_draws(
    updated_fit['posterior'],
    updated_annual,
)
```

```
fig, axes = plt.subplots(3, 1, figsize=(9, 8), sharex=True)
for ax, (key, ylabel) in zip(axes, feature_specs):
    lower, median, upper = np.quantile(
        updated_feature_draws[key],
        (0.05, 0.5, 0.95),
        axis=1,
    )
    line, = ax.plot(updated_dates[updated_annual], median, lw=2)
    ax.fill_between(
        updated_dates[updated_annual],
        lower,
        upper,
        color=line.get_color(),
        alpha=0.2,
    )
    ax.axvline(2000.75, color='0.65', ls='--', lw=1)
    ax.set_ylabel(ylabel)
axes[-1].set_xlabel('year')
plt.tight_layout()
plt.show()
```

The solid lines are posterior medians, and the shaded regions are pointwise 90 percent intervals rather than simultaneous whole-path bands.

After 2000 the core-inflation median is comparatively flat and the persistence median remains low, whereas the natural-rate band is broad and widens again at the endpoint.

The endpoint rows summarize uncertainty in the three nonlinear features.

```
def endpoint_feature_intervals(draws):
    """Summarize draw-wise uncertainty in endpoint model features."""
    labels = {
        'core': 'core inflation (%)',
        'natural': 'natural rate (%)',
        'persistence': 'normalized persistence',
    }
    rows = {}
    for key, label in labels.items():
        values = draws[key][-1]
        rows[label] = {
            'median': np.median(values),
            '5th percentile': np.quantile(values, 0.05),
            '95th percentile': np.quantile(values, 0.95),
        }
    return pd.DataFrame.from_dict(rows, orient='index')


updated_endpoint_intervals = endpoint_feature_intervals(
    updated_feature_draws
)
updated_endpoint_intervals.round(3)
```

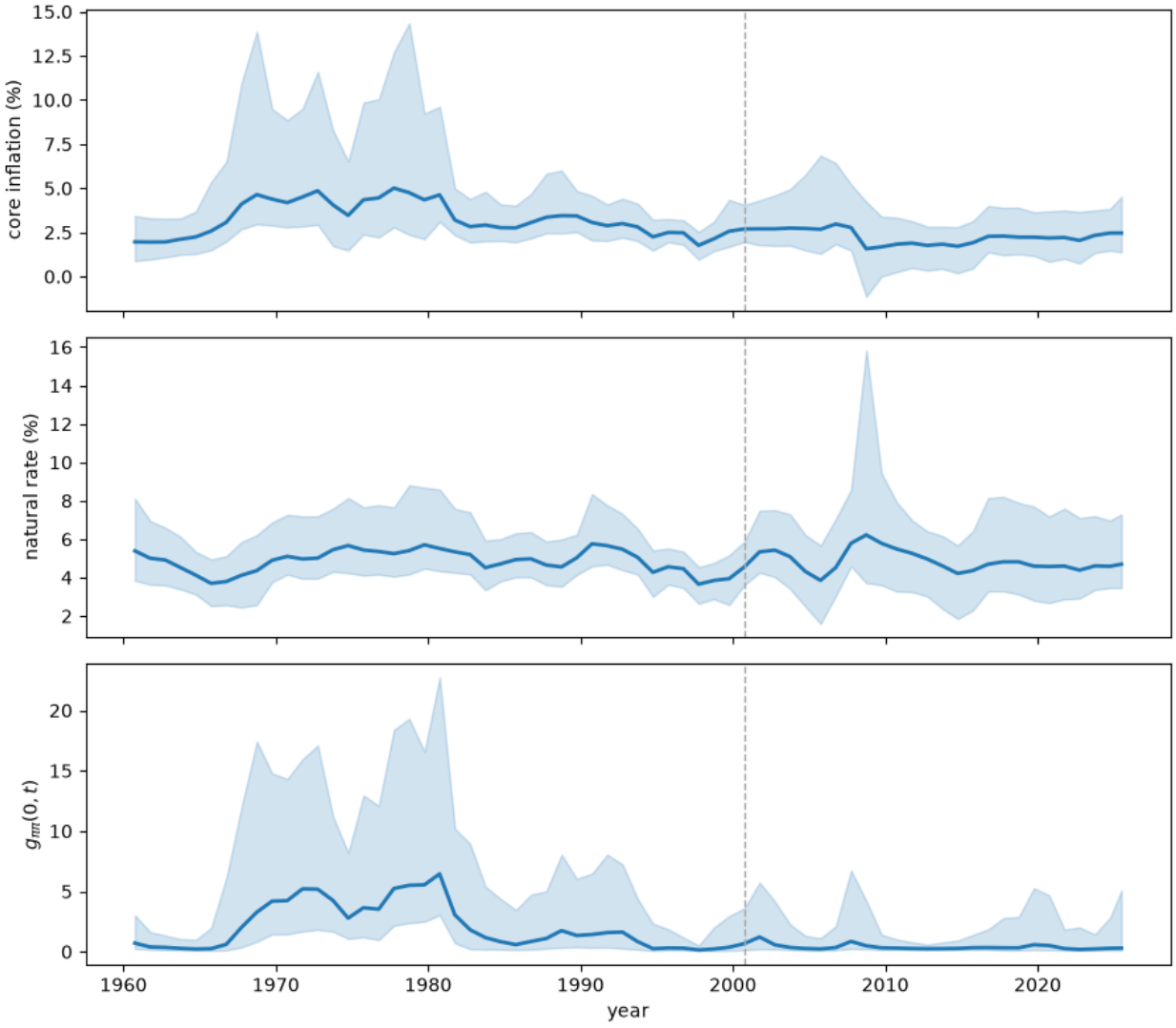


Fig. 101.20: Posterior medians and pointwise 90 percent intervals for $\bar{\pi}_t$, $\bar{u}_t$, and $g_{\pi\pi}(0,t)$ through 2025Q3

```
                         median  5th percentile  95th percentile
core inflation (%)        2.463           1.383            4.529
natural rate (%)          4.698           3.470            7.302
normalized persistence    0.301           0.107            5.102
```

At 2025Q3 both the core-inflation and natural-rate medians remain historically moderate, as the table above shows.

Their intervals remain broad, and normalized persistence retains a substantial upper tail, so the classification of recent inflation as temporary is not certain.

101.8.5 Did inflation become persistent again?

Here persistence means propagation of an inflation innovation into future inflation, rather than the number of quarters in which observed inflation remains high.

The normalized spectrum reduces sensitivity to a common change in shock scale, although it still depends on the relative variances and covariances in R_t.

```
fig, ax = plt.subplots()
ax.plot(
    updated_dates[updated_annual],
    updated_features['persistence'][updated_annual],
    'o-',
    lw=2,
    markersize=3,
)
ax.axvline(2000.75, color='0.65', ls='--', lw=1)
ax.set_xlabel('year')
ax.set_ylabel(r'$g_{\pi\pi}(0,t)$')
plt.show()
```

The estimated $g_{\pi\pi}(0,t)$ path recreates the rise to a 1980 peak and the subsequent collapse, but it stays near 0.2–0.5 after 2000 and rises only slightly after 2020.

A sequence of large reduced-form innovations can keep observed inflation high for several quarters without generating the strong propagation estimated for the 1970s.

```
fig, axes = plt.subplots(
    2,
    1,
    figsize=(9, 6),
    sharex=True,
)
axes[0].plot(
    updated_dates[updated_annual],
    100 * updated_features['core'][updated_annual],
    'o-',
    lw=2,
    markersize=3,
)
axes[1].plot(
    updated_dates[updated_annual],
    updated_features['persistence'][updated_annual],
    'x-',
    lw=2,
```

(continues on next page)

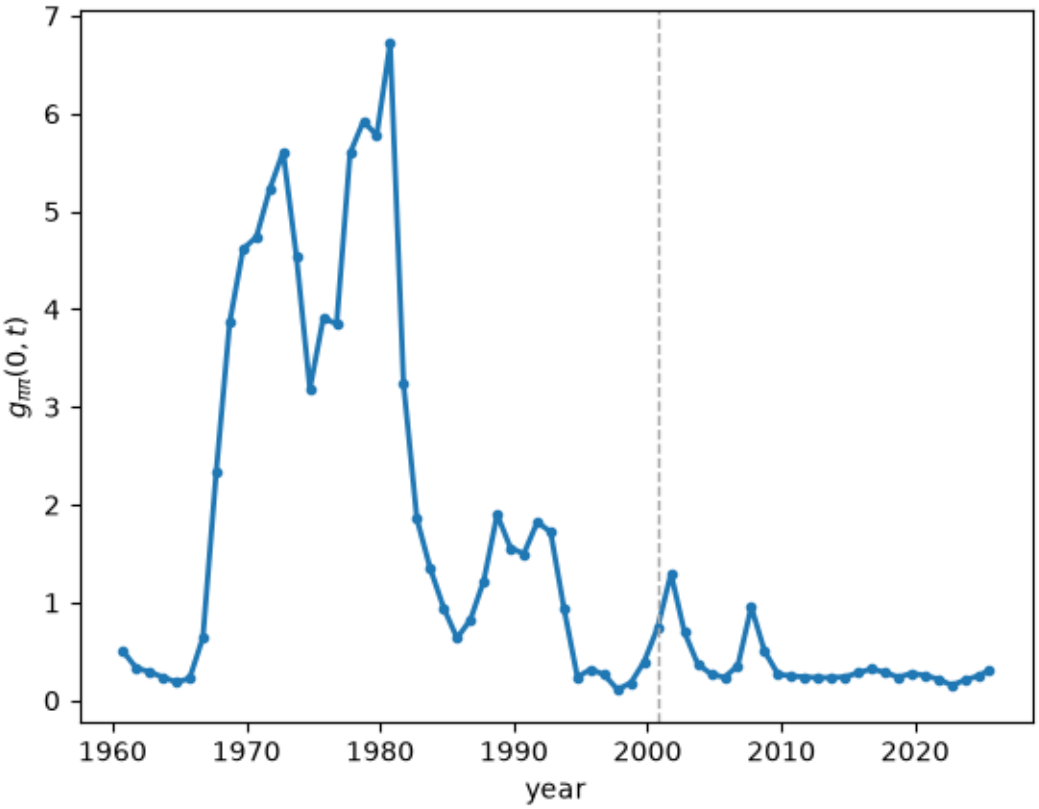


Fig. 101.21: $g_{\pi\pi}(0,t)$ through 2025Q3

(continued from previous page)

```
    markersize=4,
)
for ax in axes:
    ax.axvline(2000.75, color='0.65', ls='--', lw=1)
axes[0].set_ylabel('core inflation (%)')
axes[1].set_ylabel('normalized spectrum at zero')
axes[1].set_xlabel('year')
plt.tight_layout()
plt.show()
```

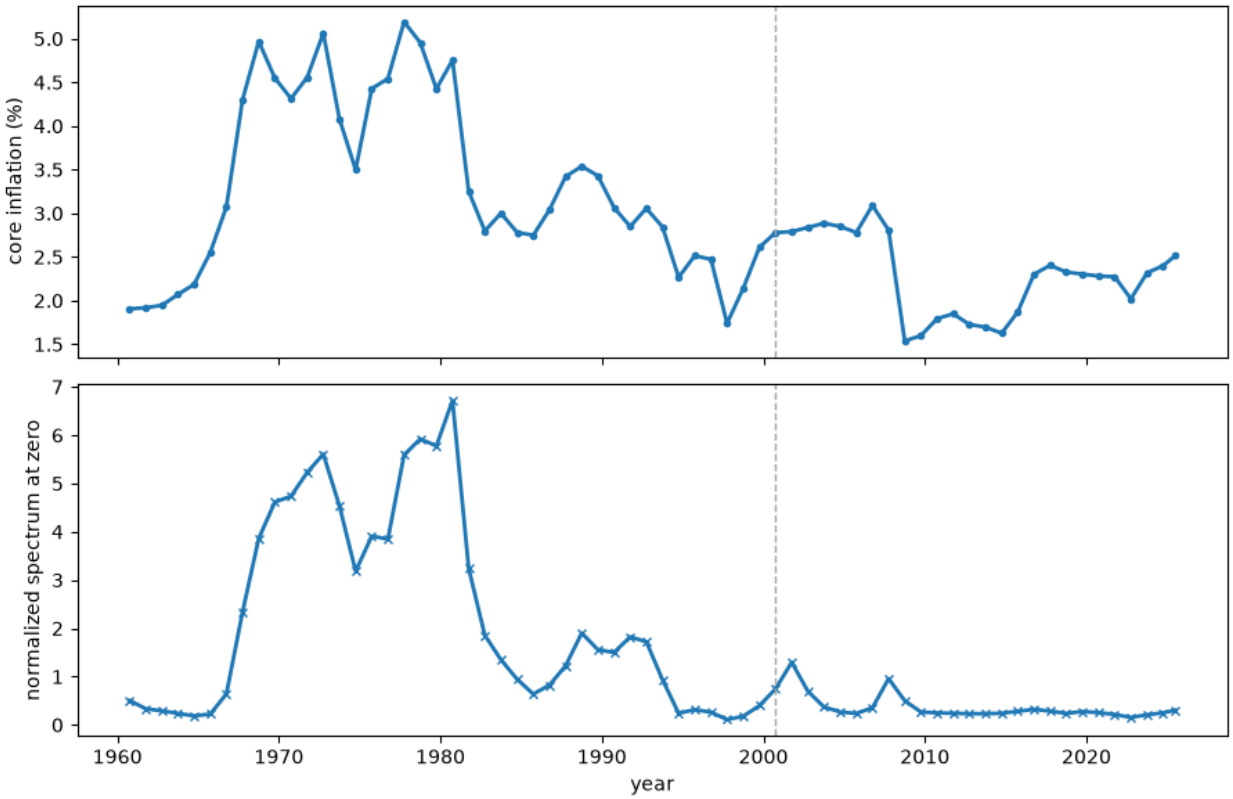


Fig. 101.22: $\bar{\pi}_t$ and $g_{\pi\pi}(0, t)$ through 2025Q3

Core inflation recovers from its mid-2010s low toward its earlier level, while persistence remains in its low post-1980 range instead of rising with it.

This divergence separates a modest rise in the model's long-run inflation rate from a return to 1970s-style propagation.

The full spectrum shows where the difference comes from.

```
updated_raw_spectrum, updated_normalized_spectrum = (
    inflation_spectrum_surface(
        updated_features['θ'],
        updated_features['covariance'],
        spectrum_frequencies,
    )
)
```

```
fig, axes = plt.subplots(
    1,
    2,
    figsize=(9, 4),
    sharey=True,
    constrained_layout=True,
)
updated_surfaces = (
    (updated_raw_spectrum, 'raw spectrum', 'log10 power'),
    (
        updated_normalized_spectrum,
        'normalized spectrum',
        'log10 normalized power',
    ),
)
for ax, (surface, title, color_label) in zip(axes, updated_surfaces):
    image = ax.pcolormesh(
        updated_dates,
        spectrum_frequencies,
        np.log10(surface),
        shading='auto',
    )
    ax.axvline(2000.75, color='0.85', ls='--', lw=1)
    ax.set_xlabel('year')
    ax.set_ylabel(f'{title}\ncycles per quarter')
    fig.colorbar(image, ax=ax, label=color_label)
plt.show()
```

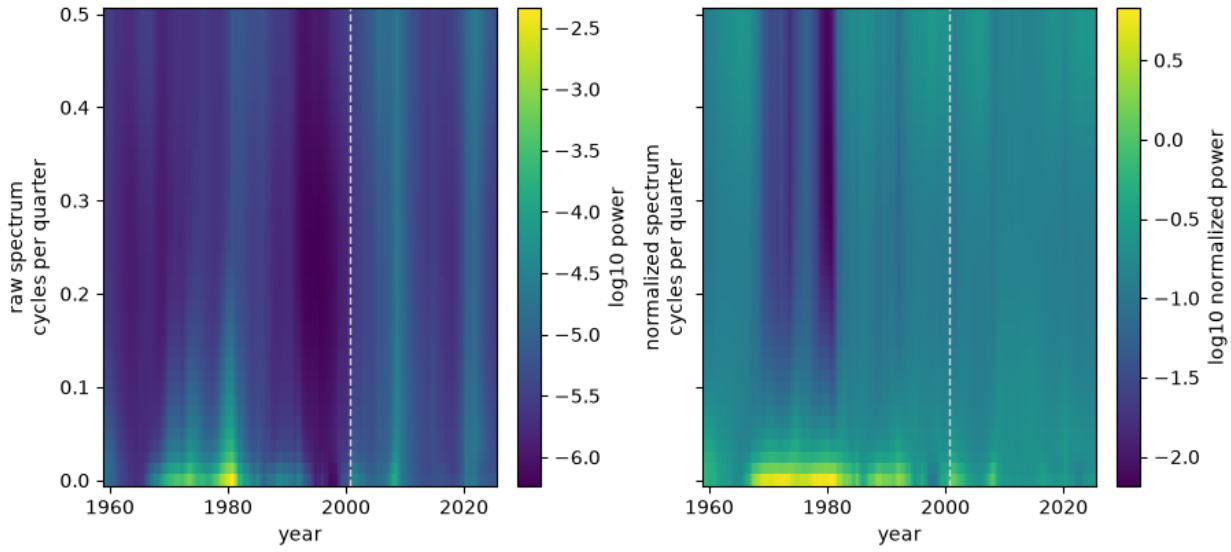


Fig. 101.23: $f_{\pi\pi}(\omega, t)$ and $g_{\pi\pi}(\omega, t)$ through 2025Q3

The financial crisis and pandemic appear as bright, broad bands in $f_{\pi\pi}(\omega, t)$ because reduced-form innovation variance increased.

The normalized spectrum $g_{\pi\pi}(\omega, t)$ lacks a post-2000 low-frequency ridge comparable to the 1970s.

Together, the panels weigh against a return to 1970s-style persistence without making the normalized statistic independent

of R_t.

```
def episode_summary(fit, features):
    """Average selected features over economically distinct episodes."""
    years = np.floor(fit['data']['dates'] + 1e-8).astype(int)
    periods = {
        '1970--1979': (years >= 1970) & (years <= 1979),
        '1985--2000': (years >= 1985) & (years <= 2000),
        '2001--2019': (years >= 2001) & (years <= 2019),
        '2020--2022': (years >= 2020) & (years <= 2022),
        '2023--2025Q3': years >= 2023,
    }
    rows = {}
    for label, mask in periods.items():
        rows[label] = {
            'core inflation (%)': 100 * features['core'][mask].mean(),
            'normalized persistence': features['persistence'][mask].mean(),
            'log generalized innovation variance': (
                features['logdet'][mask].mean()
            ),
        }
    return pd.DataFrame.from_dict(rows, orient='index')


updated_episodes = episode_summary(updated_fit, updated_features)
updated_episodes.round(3)
```

```
              core inflation (%)  normalized persistence  \
1970--1979                 4.518                   4.756
1985--2000                 2.786                   0.914
2001--2019                 2.293                   0.414
2020--2022                 2.226                   0.229
2023--2025Q3               2.343                   0.228

              log generalized innovation variance
1970--1979                                -38.614
1985--2000                                -42.006
2001--2019                                -42.099
2020--2022                                -36.432
2023--2025Q3                              -41.306
```

The episode averages contrast the high volatility of 2020–2022 with the lower normalized persistence of the post-2000 decades.

101.8.6 Can recent policy activism be measured?

When $|\beta_3| < 1$, the policy margin gives the same active-policy classification after 2000 without division by $1 - \beta_3$.

In the following figures, the gray line marks the active-policy threshold $\mathcal{M}_t = 0$.

```
fig, ax = plt.subplots()
updated_policy_margin = updated_features['policy_margin']
ax.plot(updated_dates, updated_policy_margin, lw=2)
ax.axhline(0, color='0.45', lw=1)
ax.axvline(2000.75, color='0.65', ls='--', lw=1)
ax.set_xlabel('year')
```

(continues on next page)

(continued from previous page)

```
ax.set_ylabel(r'policy margin $\mathcal{M}_t$')
plt.show()
```

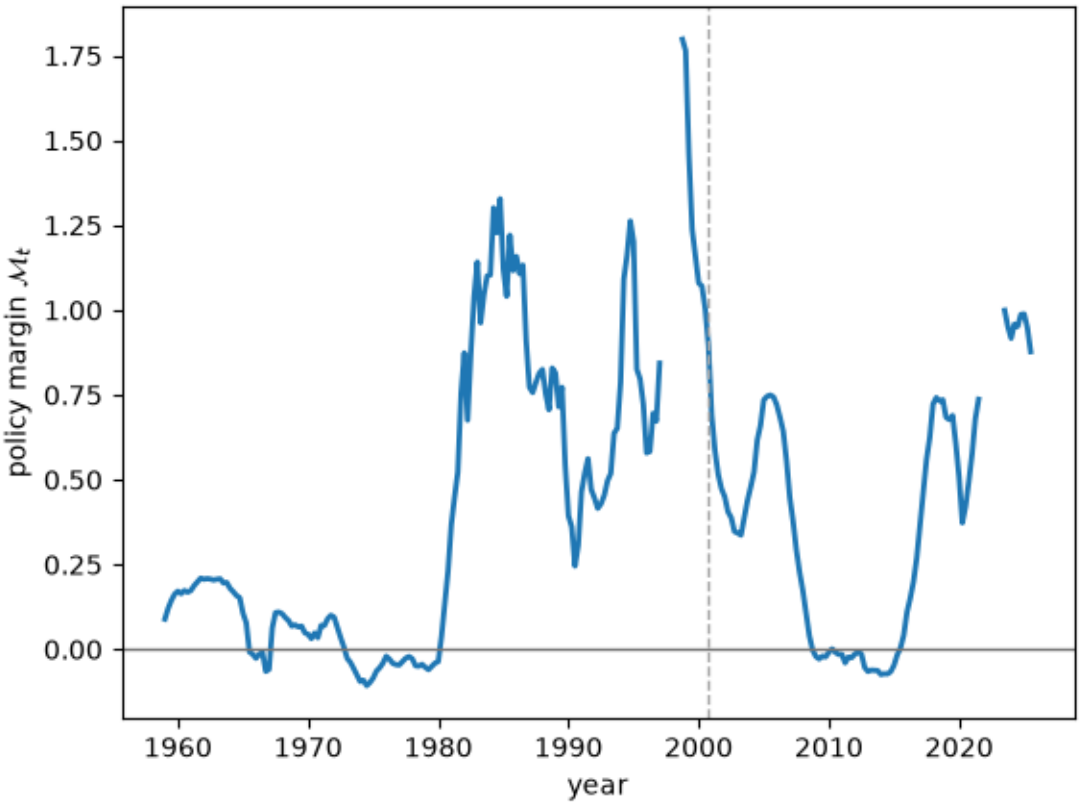


Fig. 101.24: Policy margin $\mathcal{M}_t$ through 2025Q3

Among the displayed post-2000 dates, the margin is mostly positive and moves toward zero in the mid-2010s and around 2020.

Blank intervals omit dates for which the fitted interest-rate response does not settle down.

```
fig, axes = plt.subplots(1, 2, figsize=(9, 4))
recent_annual = annual_indices(updated_dates, start=2001)
displayed_recent = recent_annual[
    np.isfinite(updated_policy_margin[recent_annual])
]
axes[0].scatter(
    updated_policy_margin[displayed_recent],
    100 * updated_features['core'][displayed_recent],
    s=18,
)
axes[1].scatter(
    updated_policy_margin[displayed_recent],
    updated_features['persistence'][displayed_recent],
    s=18,
)
```

(continues on next page)

(continued from previous page)

```
for ax in axes:
    ax.axvline(0, color='0.45', lw=1)
    ax.set_xlabel(r'policy margin $\mathcal{M}_t$')
axes[0].set_ylabel('core inflation (%)')
axes[1].set_ylabel('normalized spectrum at zero')
plt.tight_layout()
plt.show()
```

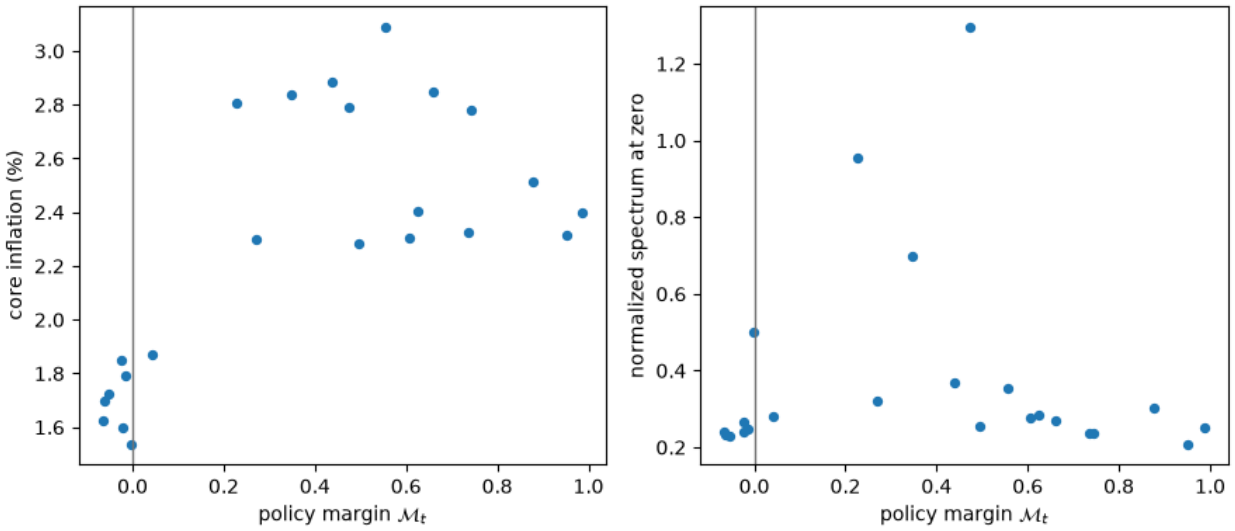


Fig. 101.25: Post-2000 $\mathcal{M}_t$ versus $\bar{\pi}_t$ and $g_{\pi\pi}(0, t)$

The remaining post-2000 observations do not establish a stable causal relation between the policy margin, core inflation, and persistence.

We also examine the central draw distribution at the 2025Q3 endpoint.

```
def endpoint_policy_margin_draws(fit):
    """Return endpoint margins and stability indicators."""
    result = fit['posterior']
    margins = np.empty(result['SD'].shape[2])
    stable_response = np.empty(result['SD'].shape[2], dtype=bool)
    for draw in range(len(margins)):
        covariance = innovation_covariance(
            result['HD'][-1, :, draw],
            result['CD'][:, draw],
        )
        rule = policy_rule_coefficients(
            result['SD'][:, -1, draw], covariance
        )
        margins[draw] = rule[0] + rule[2] - 1
        stable_response[draw] = np.abs(rule[2]) < 1
    return margins, stable_response


updated_margin_draws, stable_response_draws = (
    endpoint_policy_margin_draws(updated_fit)
```

(continues on next page)

(continued from previous page)

```
)
assert np.any(stable_response_draws)
stable_margin_draws = updated_margin_draws[stable_response_draws]
updated_margin_summary = pd.Series(
    {
        'median M': np.median(stable_margin_draws),
        '5th percentile of M': np.quantile(stable_margin_draws, 0.05),
        '95th percentile of M': np.quantile(stable_margin_draws, 0.95),
        'P(M >= 0 | |beta_3| < 1)': np.mean(stable_margin_draws >= 0),
        'share with |beta_3| < 1': stable_response_draws.mean(),
    },
    name=updated_label,
).to_frame().T
updated_margin_summary.round(3)
```

```
        median M  5th percentile of M  95th percentile of M  \
2025Q3     0.383               -0.643                 1.121

        P(M >= 0 | |beta_3| < 1)  share with |beta_3| < 1
2025Q3                     0.904                    0.732
```

```
updated_margin_limits = np.quantile(
    stable_margin_draws,
    (0.05, 0.95),
)
updated_margin_bins = np.linspace(*updated_margin_limits, 31)

fig, ax = plt.subplots()
central_draws = stable_margin_draws[
    (stable_margin_draws >= updated_margin_limits[0])
    & (stable_margin_draws <= updated_margin_limits[1])
]
ax.hist(
    central_draws,
    bins=updated_margin_bins,
    histtype='step',
    lw=2,
)
ax.axvline(0, color='0.45', lw=1)
ax.set_xlabel(r'policy margin $\mathcal{M}_t$')
ax.set_ylabel('draw count')
plt.show()
```

The table reports the share of draws with a stable interest-rate response and the active-policy probability among those draws.

An interval spanning zero indicates that the endpoint classification remains uncertain.

The plot displays only draws between the 5th and 95th percentiles, and the zero lower bound and unconventional policy further weaken the interpretation of this short-rate projection after 2008.

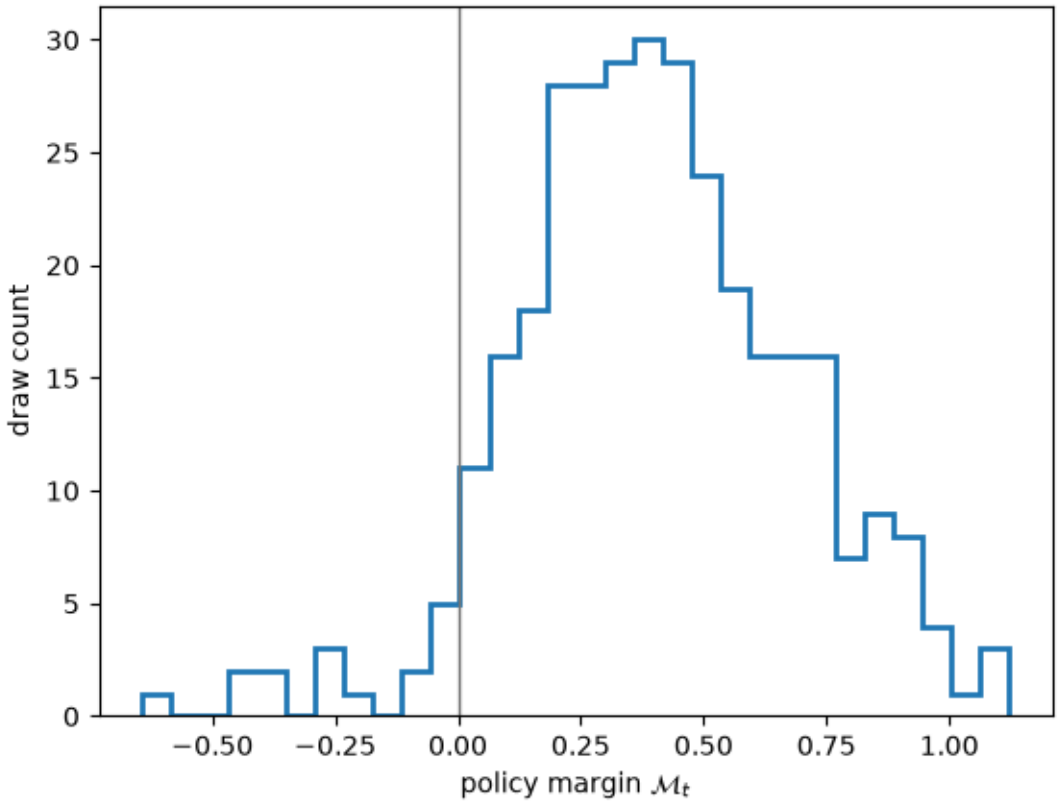


Fig. 101.26: Central 90 percent of posterior draws for $\mathcal{M}_t$ conditional on $|\beta_3| < 1$ at 2025Q3

101.8.7 What the additional observations change

The extra quarter-century adds a dramatic volatility episode without a post-2000 low-frequency ridge comparable to the 1970s.

The pandemic is the dominant aggregate uncertainty episode, but the recent inflation surge does not reproduce the 1970s low-frequency persistence ridge.

Within the updated TVP-VAR, the full-sample drift-rate posterior remains above its conservative prior calibration; a fixed-coefficient comparison would require a separate $Q = 0$ model.

The 2025Q3 natural-rate and policy-margin estimates remain imprecise, especially because not every posterior draw satisfies $|\beta_3| < 1$.

101.9 Bad policy or bad luck? A verdict

The Bayesian VAR delivers a nuanced answer to the question that opened this lecture.

- *Volatilities drifted:* the size of the shocks changed enormously, with a Volcker-era spike and a subsequent Great Moderation, so the bad-luck story captures something real.
- *The fitted TVP-VAR attributes variation to coefficients too:* inflation persistence and core inflation rose through the 1970s and fell in the 1980s, although a formal fixed-versus-drifting comparison requires a separate $Q = 0$ model.
- *The new observations do not overturn that distinction:* the pandemic produces an extreme volatility episode, while the recent inflation surge does not recreate the persistence of the 1970s.

The reduced-form VAR cannot by itself prove that changes in Federal Reserve beliefs caused the coefficient drift, because private behavior and other omitted mechanisms can also change reduced-form dynamics.

There is one more twist, and it loops us back to the theory of this section.

The escape-route models of *Adaptive Learning and Escape Dynamics* and *Escaping Nash Inflation* predict that inflation persistence should *grow* along a disinflation as a learning government becomes reluctant to abandon a high-inflation self-confirming equilibrium.

The data show the opposite because persistence fell as inflation came down after 1980.

It helped motivate later learning models, including Cogley and Sargent [2005] and Primiceri [2006], in which policymakers' reluctance to disinflate in the 1970s and their eventual conversion generate persistence that first rises and then falls.

The friendly debate with Sims, Zha, Bernanke, and Mihov thus did more than adjudicate a historical question.

It sharpened the theoretical models of learning and drift that run through this section, from the *self-confirming equilibria* of the *Conquest* book to the *drifting Fed beliefs* used to interpret later inflation.

101.10 Exercises

Exercise 101.10.1

For an $AR(1)$ process, normalized zero-frequency power is

$$g(0) = \frac{1+\rho}{2\pi(1-\rho)}.$$

Compute $g(0)$ for $\rho = 0$, 0.85, and 0.97, and use the persistence path above to interpret how inflation dynamics changed around 1980.

Solution

```
ρ = np.array([0.0, 0.85, 0.97])
g0 = (1 + ρ) / (2 * np.pi * (1 - ρ))
pd.Series(g0, index=ρ, name='normalized power at zero').to_frame()
```

```
      normalized power at zero
0.00                  0.159155
0.85                  1.962911
0.97                 10.451175
```

White noise has $g(0) = 1/(2\pi)$, while values between roughly 2 and 10 correspond to highly persistent autoregressions with coefficients between about 0.85 and 0.97.

Thus, the rise in zero-frequency power during the Great Inflation and its decline after 1980 represent a large change in persistence.

Exercise 101.10.2

Throughout this lecture we noted that a formal contrast between drifting and constant coefficients requires refitting the model with $Q = 0$, the pure "bad luck" special case in which the VAR coefficients are frozen and only the stochastic volatilities H_t move.

The sampler already supports this: pass `fixed_q=np.zeros((n_coefficients, n_coefficients))` to `run_sampler` to hold Q at zero instead of drawing it.

Fit this constant-coefficient model to the Cogley–Sargent sample and plot its normalized zero-frequency spectrum $g_{\pi\pi}(0, t)$ against the drifting-coefficient path from Fig. 101.7.

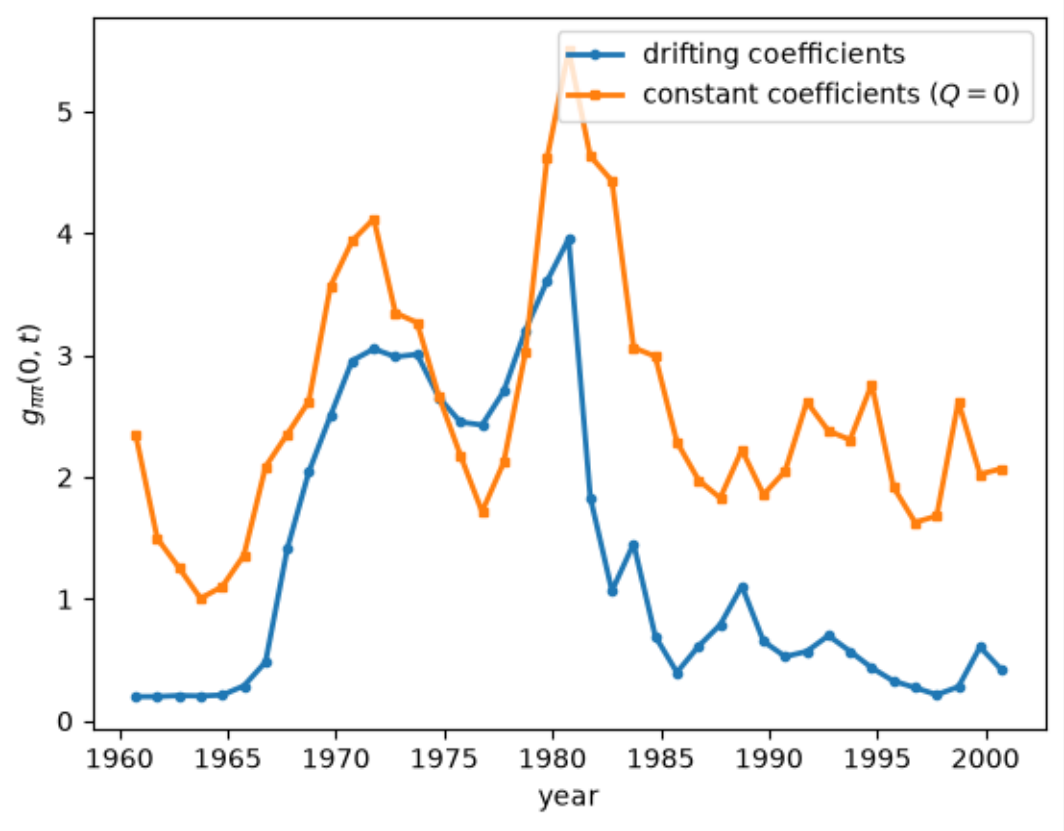
What happens to the 1970s rise and post-1980 fall in measured persistence, and what does that tell you about whether drifting *volatility* alone can account for the persistence dynamics?

Solution

With $Q = 0$ the elliptical-slice update returns a coefficient path that is constant across time, so $A_{t|T}$ no longer moves and the only remaining source of time variation in $g_{\pi\pi}(0, t)$ is the drifting covariance R_t.

```
constant_posterior = run_sampler(
    data['y'],
    data['x'],
    prior,
    n_sweeps=1_000,
    burn=500,
    thin=1,
    seed=42,
    warmup=200,
    stable=True,
    fixed_q=np.zeros((n_coefficients, n_coefficients)),
)
```

```
constant_θ = constant_posterior['SD'].mean(axis=2)
constant_R = mean_innovation_covariance(
    constant_posterior['HD'], constant_posterior['CD']
)
constant_persistence = np.array([
    inflation_spectrum(constant_θ[:, t], constant_R[t], zero_frequency)[1][0]
    for t in range(len(data['dates']))
])

fig, ax = plt.subplots()
ax.plot(data['dates'][annual], inflation_persistence[annual], 'o-',
        lw=2, markersize=3, label='drifting coefficients')
ax.plot(data['dates'][annual], constant_persistence[annual], 's-',
        lw=2, markersize=3, label=r'constant coefficients ($Q=0$)')
ax.set_xlabel('year')
ax.set_ylabel(r'$g_{\pi\pi}(0,t)$')
ax.legend()
plt.show()
```



With a fixed A the persistence measure still moves, because the *composition* of R_t — the relative sizes of the three orthogonal shocks — changes even after its overall scale is normalized out.

In fact the constant-coefficient path also climbs to a peak around 1980, as inflation innovations grow large relative to the others, so the bad-luck channel alone can manufacture much of the *rise*.

What it cannot reproduce is the *fall*: after 1980 the constant-coefficient persistence stays elevated, around 2 to 3, for the rest of the sample, whereas the drifting-coefficient path collapses back below one.

Freezing A at its full-sample average leaves inflation propagating almost as strongly in the 1990s as in the 1970s.

So drifting volatility alone accounts for part of the run-up but none of the Volcker-era disinflation of persistence — the post-1980 collapse is evidence about the *systematic* dynamics, which is exactly why both channels are needed to answer the bad-policy-or-bad-luck question.

Part XIV

Optimal Growth

CHAPTER

TWO

CASS-KOOPMANS MODEL

102.1 Overview

This lecture and *Cass-Koopmans Competitive Equilibrium* describe a model that Tjalling Koopmans [Koopmans, 1965] and David Cass [Cass, 1965] used to analyze optimal growth.

The model extends the model of Robert Solow described in an earlier lecture.

It does so by making saving rate be a decision, instead of a hard-wired constant.

(Solow assumed a constant saving rate determined outside the model.)

We describe two versions of the model, a planning problem in this lecture, and a competitive equilibrium in this lecture *Cass-Koopmans Competitive Equilibrium*.

Together, the two lectures illustrate what is, in fact, a more general connection between a **planned economy** and a decentralized economy organized as a **competitive equilibrium**.

This lecture is devoted to the planned economy version.

In the planned economy, there are

- no prices
- no budget constraints

Instead there is a dictator that tells people

- what to produce
- what to invest in physical capital
- who is to consume what and when

The lecture uses important ideas including

- A min-max problem for solving a planning problem.
- A **shooting algorithm** for solving difference equations subject to initial and terminal conditions.
- A **turnpike** property of optimal paths for long but finite-horizon economies.
- A **stable manifold** and a **phase plane**

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install quantecon
```

Let's start with some standard imports:

```
import matplotlib.pyplot as plt
from numba import jit, float64
from numba.experimental import jitclass
import numpy as np
from quantecon.optimize import brentq
```

102.2 The Model

Time is discrete and takes values $t = 0, 1, \ldots, T$ where T is finite.

(We'll eventually study a limiting case in which $T = +\infty$)

A single good can either be consumed or invested in physical capital.

The consumption good is not durable and depreciates completely if not consumed immediately.

The capital good is durable but depreciates.

We let C_t be the total consumption of a nondurable consumption good at time t.

Let K_t be the stock of physical capital at time t.

Let $\vec{C} = \{C_0, \ldots, C_T\}$ and $\vec{K} = \{K_0, \ldots, K_{T+1}\}$.

102.2.1 Digression: Aggregation Theory

We use a concept of a representative consumer to be thought of as follows.

There is a unit mass of identical consumers indexed by $\omega \in [0, 1]$.

Consumption of consumer ω is $c(\omega)$.

Aggregate consumption is

$$C = \int_0^1 c(\omega) d\omega$$

Consider a welfare problem that chooses an allocation $\{c(\omega)\}$ across consumers to maximize

$$\int_0^1 u(c(\omega)) d\omega$$

where $u(\cdot)$ is a concave utility function with $u' > 0, u'' < 0$ and maximization is subject to

$$C = \int_0^1 c(\omega) d\omega. \tag{102.1}$$

Form a Lagrangian $L = \int_0^1 u(c(\omega)) d\omega + \lambda[C - \int_0^1 c(\omega) d\omega]$.

Differentiate under the integral signs with respect to each ω to obtain the first-order necessary conditions

$$u'(c(\omega)) = \lambda.$$

These conditions imply that $c(\omega)$ equals a constant c that is independent of ω.

To find c, use feasibility constraint (102.1) to conclude that

$$c(\omega) = c = C.$$

This line of argument indicates the special *aggregation theory* that lies beneath outcomes in which a representative consumer consumes amount C.

It appears often in aggregate economics.

We shall use this aggregation theory here and also in this lecture *Cass-Koopmans Competitive Equilibrium*.

An Economy

A representative household is endowed with one unit of labor at each t and likes the consumption good at each t.

The representative household inelastically supplies a single unit of labor N_t at each t, so that $N_t = 1$ for all $t \in \{0, 1, \dots, T\}$.

The representative household has preferences over consumption bundles ordered by the utility functional:

$$U(\vec{C}) = \sum_{t=0}^{T} \beta^t \frac{C_t^{1-\gamma}}{1-\gamma} \tag{102.2}$$

where $\beta \in (0, 1)$ is a discount factor and $\gamma > 0$ governs the curvature of the one-period utility function.

Larger γ's imply more curvature.

Note that

$$u(C_t) = \frac{C_t^{1-\gamma}}{1-\gamma} \tag{102.3}$$

satisfies $u' > 0, u'' < 0$.

$u' > 0$ asserts that the consumer prefers more to less.

$u'' < 0$ asserts that marginal utility declines with increases in C_t.

We assume that $K_0 > 0$ is an exogenous initial capital stock.

There is an economy-wide production function

$$F(K_t, N_t) = AK_t^{\alpha} N_t^{1-\alpha} \tag{102.4}$$

with $0 < \alpha < 1$, $A > 0$.

A feasible allocation $\vec{C}, \vec{K}$ satisfies

$$C_t + K_{t+1} \leq F(K_t, N_t) + (1-\delta)K_t \quad \text{for all } t \in \{0, 1, \dots, T\} \tag{102.5}$$

where $\delta \in (0, 1)$ is a depreciation rate of capital.

102.3 Planning Problem

A planner chooses an allocation $\{\vec{C}, \vec{K}\}$ to maximize (102.2) subject to (102.5).

Let $\vec{\mu} = \{\mu_0, \dots, \mu_T\}$ be a sequence of nonnegative **Lagrange multipliers**.

To find an optimal allocation, form a Lagrangian

$$\mathcal{L}(\vec{C}, \vec{K}, \vec{\mu}) = \sum_{t=0}^{T} \beta^t \left\{ u(C_t) + \mu_t \left(F(K_t, 1) + (1-\delta)K_t - C_t - K_{t+1} \right) \right\} \tag{102.6}$$

and pose the following min-max problem:

$$\min_{\vec{\mu}} \max_{\vec{C},\vec{K}} \mathcal{L}(\vec{C},\vec{K},\vec{\mu}) \tag{102.7}$$

- **Extremization** means maximization with respect to $\vec{C},\vec{K}$ and minimization with respect to $\vec{\mu}$.
- Our problem satisfies conditions that assure that second-order conditions are satisfied at an allocation that satisfies the first-order necessary conditions that we are about to compute.

Before computing first-order conditions, we present some handy formulas.

102.3.1 Useful Properties of Linearly Homogeneous Production Function

The following technicalities will help us.

Notice that

$$F(K_t,N_t) = AK_t^{\alpha}N_t^{1-\alpha} = N_t A\left(\frac{K_t}{N_t}\right)^{\alpha}$$

Define the **output per-capita production function**

$$\frac{F(K_t,N_t)}{N_t} \equiv f\left(\frac{K_t}{N_t}\right) = A\left(\frac{K_t}{N_t}\right)^{\alpha}$$

whose argument is **capital per-capita**.

It is useful to recall the following calculations for the marginal product of capital

$$\begin{aligned}
\frac{\partial F(K_t,N_t)}{\partial K_t} &= \frac{\partial N_t f\left(\frac{K_t}{N_t}\right)}{\partial K_t} \\
&= N_t f'\left(\frac{K_t}{N_t}\right)\frac{1}{N_t} \quad \text{(Chain rule)} \\
&= f'\left(\frac{K_t}{N_t}\right)\Big|_{N_t=1} \\
&= f'(K_t)
\end{aligned} \tag{102.8}$$

and the marginal product of labor

$$\begin{aligned}
\frac{\partial F(K_t,N_t)}{\partial N_t} &= \frac{\partial N_t f\left(\frac{K_t}{N_t}\right)}{\partial N_t} \quad \text{(Product rule)} \\
&= f\left(\frac{K_t}{N_t}\right) + N_t f'\left(\frac{K_t}{N_t}\right)\frac{-K_t}{N_t^2} \quad \text{(Chain rule)} \\
&= f\left(\frac{K_t}{N_t}\right) - \frac{K_t}{N_t}f'\left(\frac{K_t}{N_t}\right)\Big|_{N_t=1} \\
&= f(K_t) - f'(K_t)K_t
\end{aligned}$$

(Here we are using that $N_t = 1$ for all t, so that $K_t = \frac{K_t}{N_t}$.)

102.3.2 First-order necessary conditions

We now compute **first-order necessary conditions** for extremization of Lagrangian (102.6):

$$C_t: \qquad u'(C_t) - \mu_t = 0 \qquad \text{for all} \quad t = 0, 1, \ldots, T \tag{102.9}$$

$$K_t: \qquad \beta\mu_t \left[(1-\delta) + f'(K_t)\right] - \mu_{t-1} = 0 \qquad \text{for all} \quad t = 1, 2, \ldots, T \tag{102.10}$$

$$\mu_t: \qquad F(K_t, 1) + (1-\delta)K_t - C_t - K_{t+1} = 0 \qquad \text{for all} \quad t = 0, 1, \ldots, T \tag{102.11}$$

$$K_{T+1}: \qquad -\mu_T \leq 0, \ \leq 0 \text{ if } K_{T+1} = 0; \ = 0 \text{ if } K_{T+1} > 0 \tag{102.12}$$

In computing (102.10) we recognize that K_t appears in both the time t and time $t-1$ feasibility constraints (102.5).

Restrictions (102.12) come from differentiating with respect to K_{T+1} and applying the following **Karush-Kuhn-Tucker condition** (KKT) (see Karush-Kuhn-Tucker conditions):

$$\mu_T K_{T+1} = 0 \tag{102.13}$$

Combining (102.9) and (102.10) gives

$$\beta u'\left(C_t\right)\left[(1-\delta) + f'\left(K_t\right)\right] - u'\left(C_{t-1}\right) = 0 \quad \text{for all } t = 1, 2, \ldots, T+1$$

which can be rearranged to become

$$\beta u'\left(C_{t+1}\right)\left[(1-\delta) + f'\left(K_{t+1}\right)\right] = u'\left(C_t\right) \quad \text{for all } t = 0, 1, \ldots, T \tag{102.14}$$

Applying the inverse marginal utility of consumption function on both sides of the above equation gives

$$C_{t+1} = u'^{-1}\left(\left(\frac{\beta}{u'(C_t)}[f'(K_{t+1}) + (1-\delta)]\right)^{-1}\right)$$

which for our utility function (102.3) becomes the consumption **Euler equation**

$$C_{t+1} = \left(\beta C_t^{\gamma}[f'(K_{t+1}) + (1-\delta)]\right)^{1/\gamma} \tag{102.15}$$

which we can combine with the feasibility constraint (102.5) to get

$$\begin{aligned} C_{t+1} &= C_t\left(\beta[f'(F(K_t, 1) + (1-\delta)K_t - C_t) + (1-\delta)]\right)^{1/\gamma} \\ K_{t+1} &= F(K_t, 1) + (1-\delta)K_t - C_t. \end{aligned} \tag{102.16}$$

This is a pair of non-linear first-order difference equations that map C_t, K_t into C_{t+1}, K_{t+1} and that an optimal sequence $\vec{C}, \vec{K}$ must satisfy.

It must also satisfy the initial condition that K_0 is given and $K_{T+1} = 0$.

Below we define a `jitclass` that stores parameters and functions that define our economy.

```
planning_data = [
    ('γ', float64),     # Coefficient of relative risk aversion
    ('β', float64),     # Discount factor
    ('δ', float64),     # Depreciation rate on capital
    ('α', float64),     # Return to capital per capita
    ('A', float64)      # Technology
]
```

```
@jitclass(planning_data)
class PlanningProblem():

    def __init__(self, γ=2, β=0.95, δ=0.02, α=0.33, A=1):

        self.γ, self.β = γ, β
        self.δ, self.α, self.A = δ, α, A

    def u(self, c):
        '''
        Utility function
        ASIDE: If you have a utility function that is hard to solve by hand
        you can use automatic or symbolic differentiation
        See https://github.com/HIPS/autograd
        '''
        γ = self.γ

        return c ** (1 - γ) / (1 - γ) if γ!= 1 else np.log(c)

    def u_prime(self, c):
        'Derivative of utility'
        γ = self.γ

        return c ** (-γ)

    def u_prime_inv(self, c):
        'Inverse of derivative of utility'
        γ = self.γ

        return c ** (-1 / γ)

    def f(self, k):
        'Production function'
        α, A = self.α, self.A

        return A * k ** α

    def f_prime(self, k):
        'Derivative of production function'
        α, A = self.α, self.A

        return α * A * k ** (α - 1)

    def f_prime_inv(self, k):
        'Inverse of derivative of production function'
        α, A = self.α, self.A

        return (k / (A * α)) ** (1 / (α - 1))

    def next_k_c(self, k, c):
        ''''
        Given the current capital Kt and an arbitrary feasible
        consumption choice Ct, computes Kt+1 by state transition law
        and optimal Ct+1 by Euler equation.
        '''
        β, δ = self.β, self.δ
        u_prime, u_prime_inv = self.u_prime, self.u_prime_inv
```

(continues on next page)

(continued from previous page)

```
        f, f_prime = self.f, self.f_prime

        k_next = f(k) + (1 - δ) * k - c
        c_next = u_prime_inv(u_prime(c) / (β * (f_prime(k_next) + (1 - δ))))

        return k_next, c_next
```

We can construct an economy with the Python code:

```
pp = PlanningProblem()
```

102.4 Shooting Algorithm

We use **shooting** to compute an optimal allocation $\vec{C}, \vec{K}$ and an associated Lagrange multiplier sequence $\vec{\mu}$.

First-order necessary conditions (102.9), (102.10), and (102.11) for the planning problem form a system of **difference equations** with two boundary conditions:

- K_0 is a given **initial condition** for capital
- $K_{T+1} = 0$ is a **terminal condition** for capital that we deduced from the first-order necessary condition for K_{T+1} the KKT condition (102.13)

We have no initial condition for the Lagrange multiplier μ_0.

If we did, our job would be easy:

- Given μ_0 and k_0, we could compute c_0 from equation (102.9) and then k_1 from equation (102.11) and μ_1 from equation (102.10).
- We could continue in this way to compute the remaining elements of $\vec{C}, \vec{K}, \vec{\mu}$.

However, we woujld not be assured that the Kuhn-Tucker condition (102.13) would be satisfied.

Furthermore, we don't have an initial condition for μ_0.

So this won't work.

Indeed, part of our task is to compute the **optimal** value of μ_0.

To compute μ_0 and the other objects we want, a simple modification of the above procedure will work.

It is called the **shooting algorithm**.

It is an instance of a **guess and verify** algorithm that consists of the following steps:

- Guess an initial Lagrange multiplier μ_0.
- Apply the **simple algorithm** described above.
- Compute K_{T+1} and check whether it equals zero.
- If $K_{T+1} = 0$, we have solved the problem.
- If $K_{T+1} > 0$, lower μ_0 and try again.
- If $K_{T+1} < 0$, raise μ_0 and try again.

The following Python code implements the shooting algorithm for the planning problem.

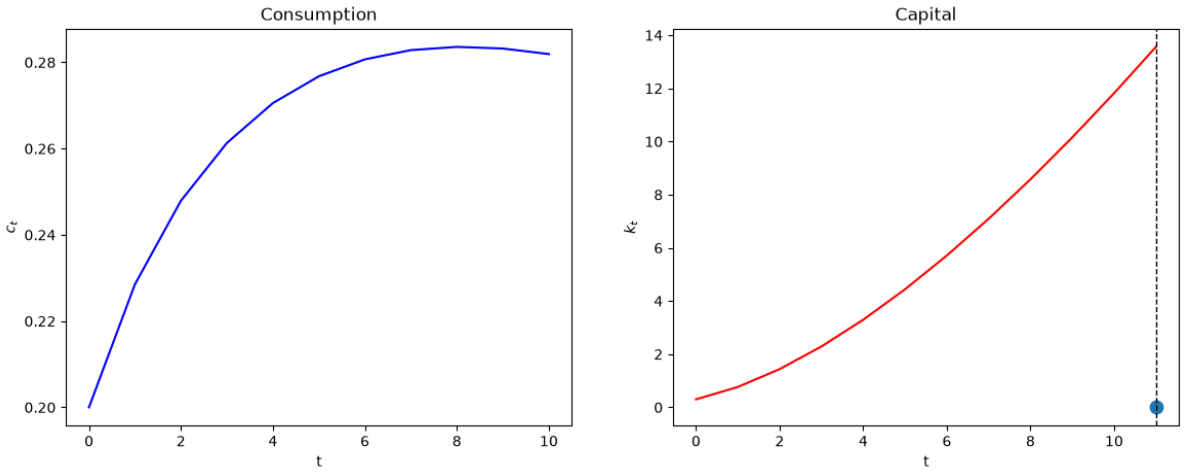
(Actually, we modified the preceding algorithm slightly by starting with a guess for c_0 instead of μ_0 in the following code.)

```
@jit
def shooting(pp, c0, k0, T=10):
    '''
    Given the initial condition of capital k0 and an initial guess
    of consumption c0, computes the whole paths of c and k
    using the state transition law and Euler equation for T periods.
    '''
    if c0 > pp.f(k0) + (1 - pp.δ) * k0:
        print("initial consumption is not feasible")

        return None

    # initialize vectors of c and k
    c_vec = np.empty(T+1)
    k_vec = np.empty(T+2)

    c_vec[0] = c0
    k_vec[0] = k0

    for t in range(T):
        k_vec[t+1], c_vec[t+1] = pp.next_k_c(k_vec[t], c_vec[t])

    k_vec[T+1] = pp.f(k_vec[T]) + (1 - pp.δ) * k_vec[T] - c_vec[T]

    return c_vec, k_vec
```

We'll start with an incorrect guess.

```
paths = shooting(pp, 0.2, 0.3, T=10)
```

```
fig, axs = plt.subplots(1, 2, figsize=(14, 5))

colors = ['blue', 'red']
titles = ['Consumption', 'Capital']
ylabels = ['$c_t$', '$k_t$']

T = paths[0].size - 1
for i in range(2):
    axs[i].plot(paths[i], c=colors[i])
    axs[i].set(xlabel='t', ylabel=ylabels[i], title=titles[i])

axs[1].scatter(T+1, 0, s=80)
axs[1].axvline(T+1, color='k', ls='--', lw=1)

plt.show()
```



Evidently, our initial guess for μ_0 is too high, so initial consumption too low.

We know this because we miss our $K_{T+1} = 0$ target on the high side.

Now we automate things with a search-for-a-good μ_0 algorithm that stops when we hit the target $K_{t+1} = 0$.

We use a **bisection method**.

We make an initial guess for C_0 (we can eliminate μ_0 because C_0 is an exact function of μ_0).

We know that the lowest C_0 can ever be is 0 and that the largest it can be is initial output $f(K_0)$.

Guess C_0 and shoot forward to $T + 1$.

If $K_{T+1} > 0$, we take it to be our new **lower** bound on C_0.

If $K_{T+1} < 0$, we take it to be our new **upper** bound.

Make a new guess for C_0 that is halfway between our new upper and lower bounds.

Shoot forward again, iterating on these steps until we converge.

When K_{T+1} gets close enough to 0 (i.e., within an error tolerance bounds), we stop.

```
@jit
def bisection(pp, c0, k0, T=10, tol=1e-4, max_iter=500, k_ter=0, verbose=True):

    # initial boundaries for guess c0
    c0_upper = pp.f(k0) + (1 - pp.δ) * k0
    c0_lower = 0

    i = 0
    while True:
        c_vec, k_vec = shooting(pp, c0, k0, T)
        error = k_vec[-1] - k_ter

        # check if the terminal condition is satisfied
        if np.abs(error) < tol:
            if verbose:
                print('Converged successfully on iteration ', i+1)
            return c_vec, k_vec

        i += 1
        if i == max_iter:
            if verbose:
```

(continues on next page)

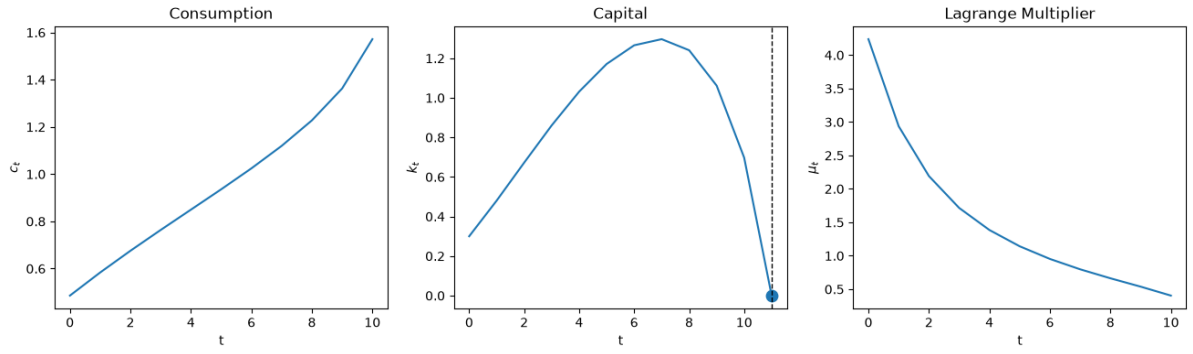
(continued from previous page)

```
                print('Convergence failed.')
            return c_vec, k_vec

        # if iteration continues, updates boundaries and guess of c0
        if error > 0:
            c0_lower = c0
        else:
            c0_upper = c0

        c0 = (c0_lower + c0_upper) / 2
```

```
def plot_paths(pp, c0, k0, T_arr, k_ter=0, k_ss=None, axs=None):

    if axs is None:
        fix, axs = plt.subplots(1, 3, figsize=(16, 4))
    ylabels = ['$c_t$', '$k_t$', r'$\mu_t$']
    titles = ['Consumption', 'Capital', 'Lagrange Multiplier']

    c_paths = []
    k_paths = []
    for T in T_arr:
        c_vec, k_vec = bisection(pp, c0, k0, T, k_ter=k_ter, verbose=False)
        c_paths.append(c_vec)
        k_paths.append(k_vec)

        μ_vec = pp.u_prime(c_vec)
        paths = [c_vec, k_vec, μ_vec]

        for i in range(3):
            axs[i].plot(paths[i])
            axs[i].set(xlabel='t', ylabel=ylabels[i], title=titles[i])

        # Plot steady state value of capital
        if k_ss is not None:
            axs[1].axhline(k_ss, c='k', ls='--', lw=1)

        axs[1].axvline(T+1, c='k', ls='--', lw=1)
        axs[1].scatter(T+1, paths[1][-1], s=80)

    return c_paths, k_paths
```

Now we can solve the model and plot the paths of consumption, capital, and Lagrange multiplier.

```
plot_paths(pp, 0.3, 0.3, [10]);
```



102.5 Setting Initial Capital to Steady State Capital

When $T \to +\infty$, the optimal allocation converges to steady state values of C_t and K_t.

It is instructive to set K_0 equal to the $\lim_{T \to +\infty} K_t$, which we'll call steady state capital.

In a steady state $K_{t+1} = K_t = \bar{K}$ for all very large t.

Evalauating feasibility constraint (102.5) at $\bar{K}$ gives

$$f(\bar{K}) - \delta \bar{K} = \bar{C} \tag{102.17}$$

Substituting $K_t = \bar{K}$ and $C_t = \bar{C}$ for all t into (102.14) gives

$$1 = \beta \frac{u'(\bar{C})}{u'(\bar{C})} [f'(\bar{K}) + (1 - \delta)]$$

Defining $\beta = \frac{1}{1+\rho}$, and cancelling gives

$$1 + \rho = 1[f'(\bar{K}) + (1 - \delta)]$$

Simplifying gives

$$f'(\bar{K}) = \rho + \delta$$

and

$$\bar{K} = f'^{-1}(\rho + \delta)$$

For production function (102.4), this becomes

$$\alpha \bar{K}^{\alpha - 1} = \rho + \delta$$

As an example, after setting $\alpha = .33$, $\rho = 1/\beta - 1 = 1/(19/20) - 1 = 20/19 - 19/19 = 1/19$, $\delta = 1/50$, we get

$$\bar{K} = \left(\frac{\frac{33}{100}}{\frac{1}{50} + \frac{1}{19}} \right)^{\frac{67}{100}} \approx 9.57583$$

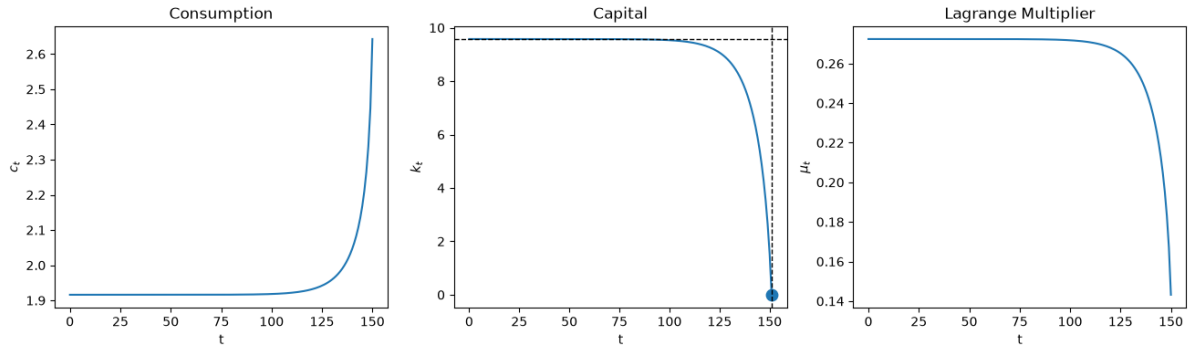
Let's verify this with Python and then use this steady state $\bar{K}$ as our initial capital stock K_0.

```
ρ = 1 / pp.β - 1
k_ss = pp.f_prime_inv(ρ+pp.δ)

print(f'steady state for capital is: {k_ss}')
```

```
steady state for capital is: 9.57583816331462
```

Now we plot

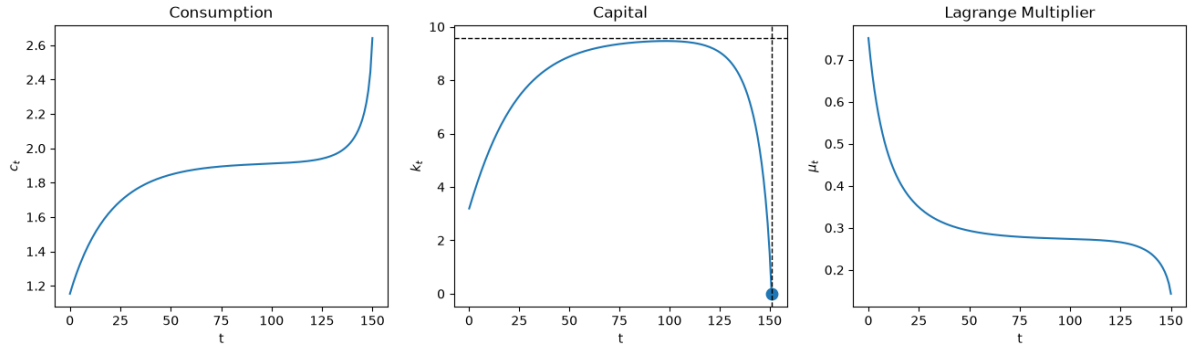
```
plot_paths(pp, 0.3, k_ss, [150], k_ss=k_ss);
```



Evidently, with a large value of T, K_t stays near K_0 until t approaches T closely.

Let's see what the planner does when we set K_0 below $\bar{K}$.

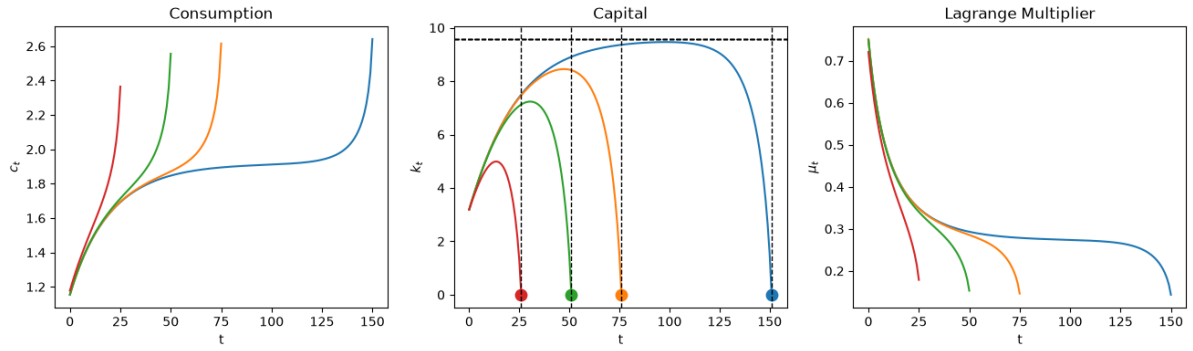
```
plot_paths(pp, 0.3, k_ss/3, [150], k_ss=k_ss);
```



Notice how the planner pushes capital toward the steady state, stays near there for a while, then pushes K_t toward the terminal value $K_{T+1} = 0$ when t closely approaches T.

The following graphs compare optimal outcomes as we vary T.

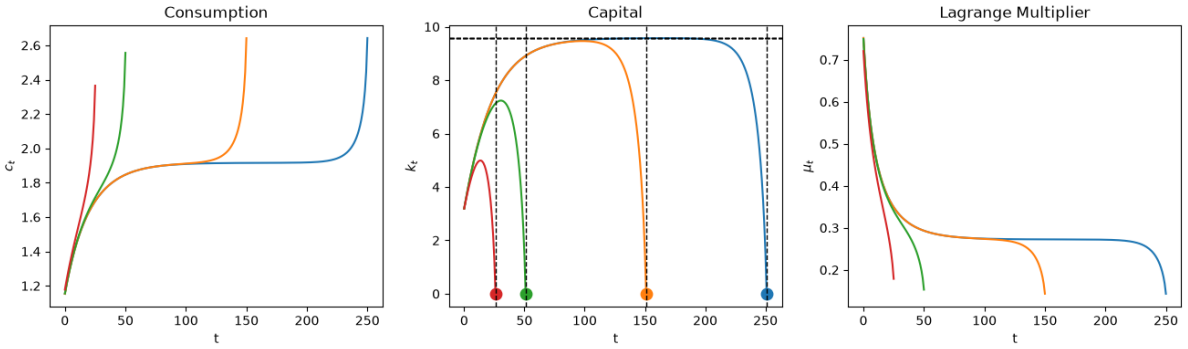
```
plot_paths(pp, 0.3, k_ss/3, [150, 75, 50, 25], k_ss=k_ss);
```



102.6 A Turnpike Property

The following calculation indicates that when T is very large, the optimal capital stock stays close to its steady state value most of the time.

```
plot_paths(pp, 0.3, k_ss/3, [250, 150, 50, 25], k_ss=k_ss);
```



In the above graphs, different colors are associated with different horizons T.

Notice that as the horizon increases, the planner keeps K_t closer to the steady state value $\bar{K}$ for longer.

This pattern reflects a **turnpike** property of the steady state.

A rule of thumb for the planner is

- from K_0, push K_t toward the steady state and stay close to the steady state until time approaches T.

The planner accomplishes this by adjusting the saving rate $\frac{f(K_t)-C_t}{f(K_t)}$ over time.

Exercise 102.6.1

The turnpike property is independent of the initial condition K_0 provided that T is sufficiently large.

Expand the `plot_paths` function so that it plots trajectories for multiple initial points using `k0s = [k_ss*2, k_ss*3, k_ss/3]`.

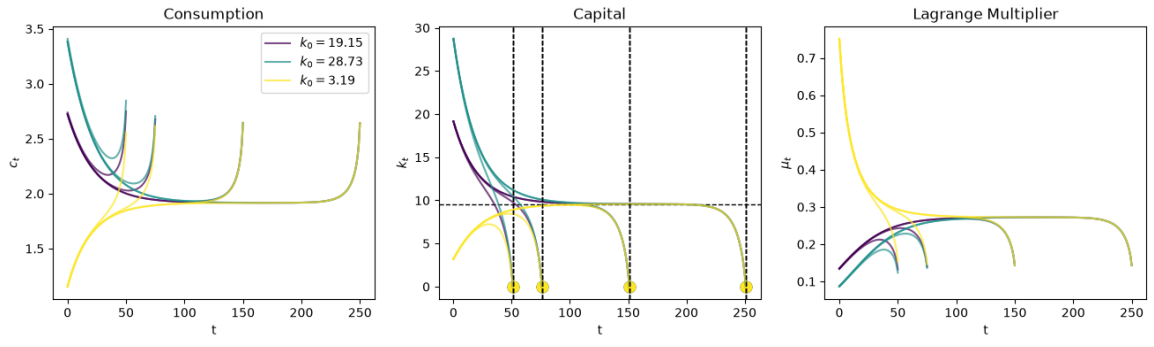
Solution

Here is one solution

```
def plot_multiple_paths(pp, c0, k0s, T_arr, k_ter=0, k_ss=None, axs=None):
    if axs is None:
        fig, axs = plt.subplots(1, 3, figsize=(16, 4))

    ylabels = ['$c_t$', '$k_t$', r'$\mu_t$']
    titles = ['Consumption', 'Capital', 'Lagrange Multiplier']

    colors = plt.cm.viridis(np.linspace(0, 1, len(k0s)))

    all_c_paths = []
    all_k_paths = []

    for i, k0 in enumerate(k0s):
        k0_c_paths = []
        k0_k_paths = []
```

```
        for T in T_arr:
            c_vec, k_vec = bisection(pp, c0, k0, T, k_ter=k_ter, verbose=False)
            k0_c_paths.append(c_vec)
            k0_k_paths.append(k_vec)

            μ_vec = pp.u_prime(c_vec)
            paths = [c_vec, k_vec, μ_vec]

            for j in range(3):
                axs[j].plot(paths[j], color=colors[i],
                            label=f'$k_0 = {k0:.2f}$' if j == 0 and T == T_arr[0]␣
↪else "", alpha=0.7)
                axs[j].set(xlabel='t', ylabel=ylabels[j], title=titles[j])

            if k_ss is not None and i == 0 and T == T_arr[0]:
                axs[1].axhline(k_ss, c='k', ls='--', lw=1)

            axs[1].axvline(T+1, c='k', ls='--', lw=1)
            axs[1].scatter(T+1, paths[1][-1], s=80, color=colors[i])

        all_c_paths.append(k0_c_paths)
        all_k_paths.append(k0_k_paths)

    # Add legend if multiple initial points
    if len(k0s) > 1:
        axs[0].legend()

    return all_c_paths, all_k_paths
```

```
_ = plot_multiple_paths(pp, 0.3, [k_ss*2, k_ss*3, k_ss/3], [250, 150, 75, 50], k_
↪ss=k_ss)
```



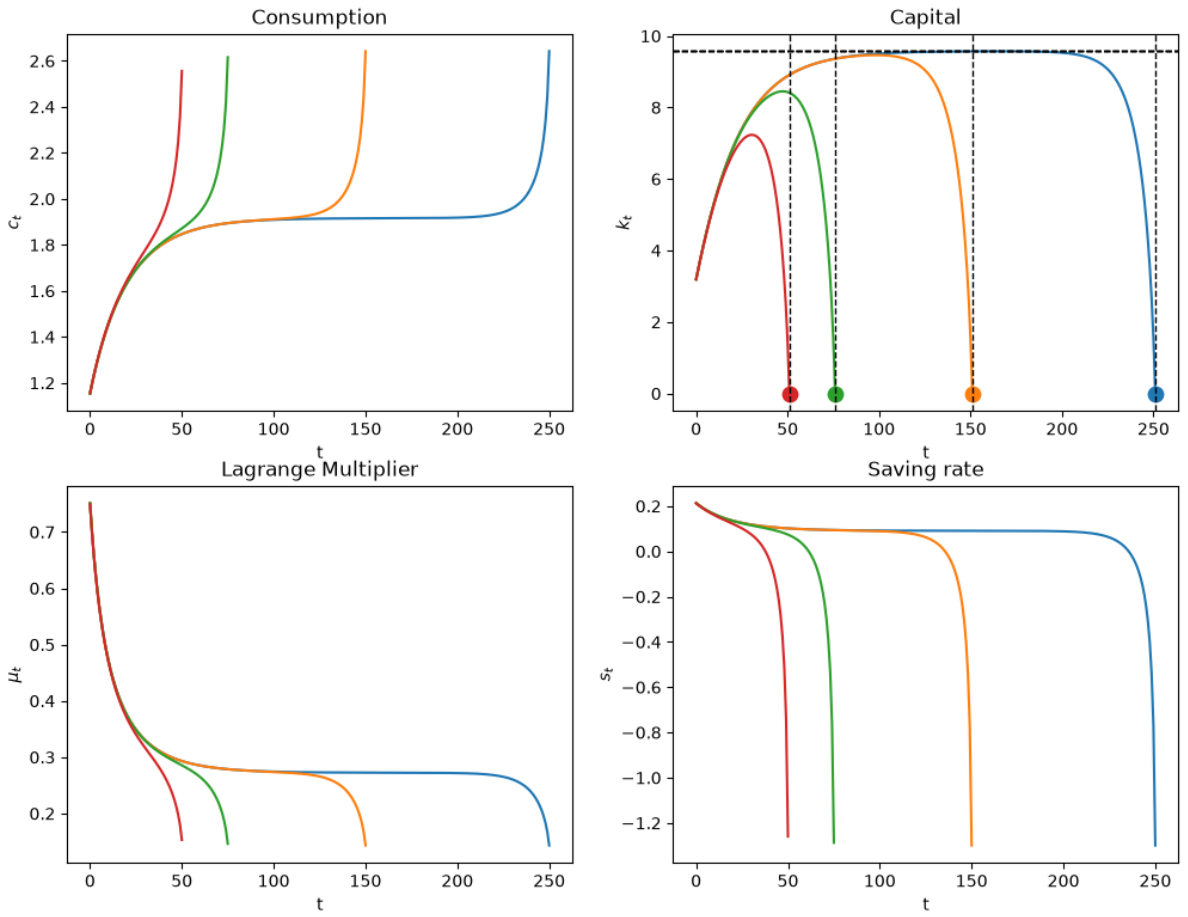
We see that the turnpike property holds for various initial values of K_0.

Let's calculate and plot the saving rate.

```
@jit
def saving_rate(pp, c_path, k_path):
    'Given paths of c and k, computes the path of saving rate.'
    production = pp.f(k_path[:-1])

    return (production - c_path) / production
```

```
def plot_saving_rate(pp, c0, k0, T_arr, k_ter=0, k_ss=None, s_ss=None):

    fix, axs = plt.subplots(2, 2, figsize=(12, 9))

    c_paths, k_paths = plot_paths(pp, c0, k0, T_arr, k_ter=k_ter, k_ss=k_ss, axs=axs.
↪flatten())

    for i, T in enumerate(T_arr):
        s_path = saving_rate(pp, c_paths[i], k_paths[i])
        axs[1, 1].plot(s_path)

    axs[1, 1].set(xlabel='t', ylabel='$s_t$', title='Saving rate')

    if s_ss is not None:
        axs[1, 1].hlines(s_ss, 0, np.max(T_arr), linestyle='--')
```

```
plot_saving_rate(pp, 0.3, k_ss/3, [250, 150, 75, 50], k_ss=k_ss)
```



102.7 A Limiting Infinite Horizon Economy

We want to set $T = +\infty$.

The appropriate thing to do is to replace terminal condition (102.12) with

$$\lim_{T \to +\infty} \beta^T u'(C_T) K_{T+1} = 0,$$

a condition that will be satisfied by a path that converges to an optimal steady state.

We can approximate the optimal path by starting from an arbitrary initial K_0 and shooting towards the optimal steady state K at a large but finite $T + 1$.

In the following code, we do this for a large T and plot consumption, capital, and the saving rate.

We know that in the steady state that the saving rate is constant and that $\bar{s} = \frac{f(\bar{K}) - \bar{C}}{f(\bar{K})}$.

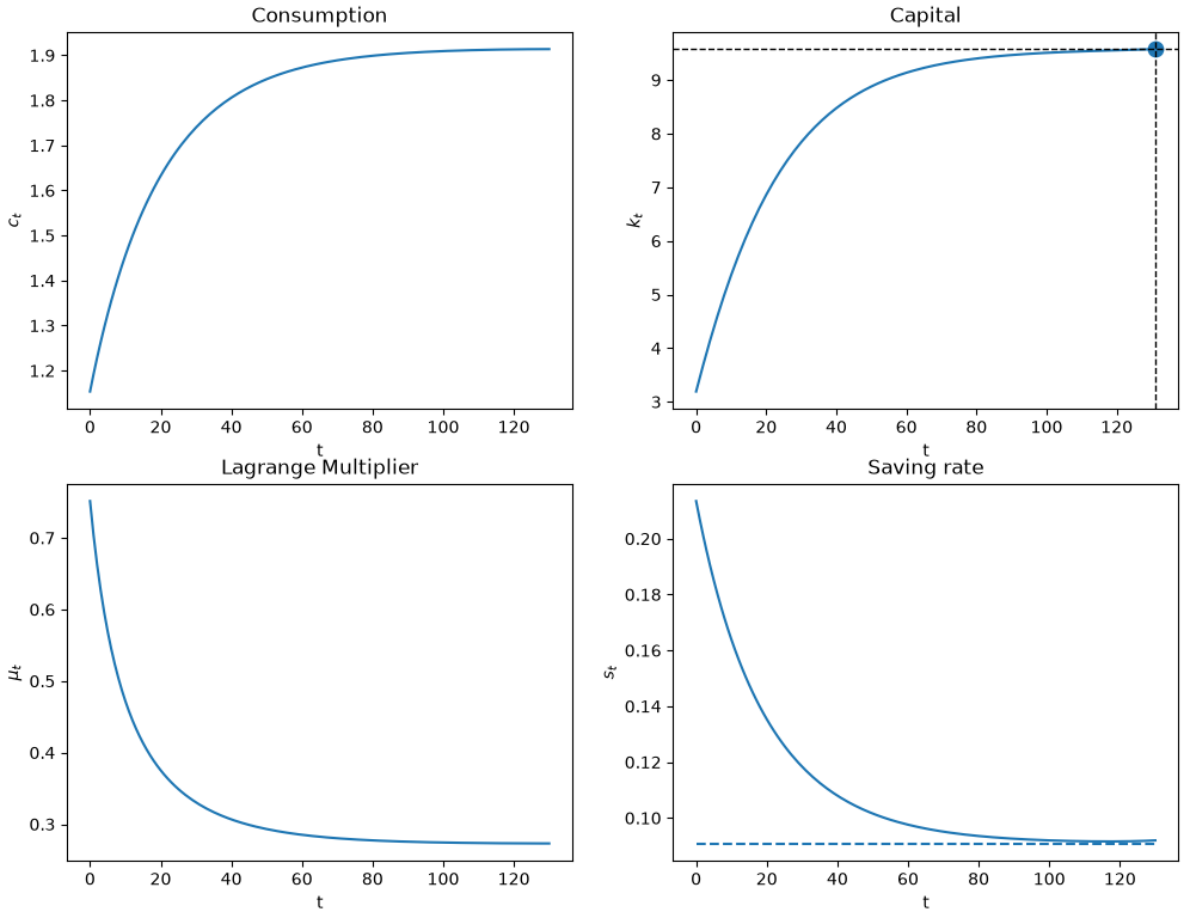
From (102.17) the steady state saving rate equals

$$\bar{s} = \frac{\delta \bar{K}}{f(\bar{K})}$$

The steady state saving rate $\bar{S} = \bar{s} f(\bar{K})$ is the amount required to offset capital depreciation each period.

We first study optimal capital paths that start below the steady state.

```
# steady state of saving rate
s_ss = pp.δ * k_ss / pp.f(k_ss)

plot_saving_rate(pp, 0.3, k_ss/3, [130], k_ter=k_ss, k_ss=k_ss, s_ss=s_ss)
```



Since $K_0 < \bar{K}$, $f'(K_0) > \rho + \delta$.

The planner chooses a positive saving rate that is higher than the steady state saving rate.

Note that $f''(K) < 0$, so as K rises, $f'(K)$ declines.

The planner slowly lowers the saving rate until reaching a steady state in which $f'(K) = \rho + \delta$.

102.8 Stable Manifold and Phase Diagram

We now describe a classic diagram that describes an optimal (K_{t+1}, C_t) path.

The diagram has K on the ordinate axis and C on the coordinate axis.

Given an arbitrary and fixed K, a fixed point C of the consumption Euler equation (102.15) satisfies

$$C = C\left(\beta\left[f'\left(f\left(K\right) + \left(1-\delta\right)K - C\right) + \left(1-\delta\right)\right]\right)^{1/\gamma}$$

which implies

$$\begin{aligned} C &= f\left(K\right) + \left(1-\delta\right)K - f'^{-1}\left(\frac{1}{\beta} - \left(1-\delta\right)\right) \\ &\equiv \tilde{C}\left(K\right) \end{aligned} \tag{102.18}$$

A positive fixed point $C = \tilde{C}(K)$ exists only if $f\left(K\right) + \left(1-\delta\right)K - f'^{-1}\left(\frac{1}{\beta} - \left(1-\delta\right)\right) > 0$

```
@jit
def C_tilde(K, pp):

    return pp.f(K) + (1 - pp.δ) * K - pp.f_prime_inv(1 / pp.β - 1 + pp.δ)
```

Next note that given a time-invariant arbitrary C, a fixed point K of the feasibility condition (102.5) solves the following equation

$$K = f(K) + (1 - \delta K) - C.$$

A fixed point of the above equation is described by a function

$$K = \tilde{K}(C) \tag{102.19}$$

```
@jit
def K_diff(K, C, pp):
    return pp.f(K) - pp.δ * K - C

@jit
def K_tilde(C, pp):

    res = brentq(K_diff, 1e-6, 100, args=(C, pp))

    return res.root
```

A steady state (K_s, C_s) is a pair (K, C) that satisfies both equations (102.18) and (102.19).

It is thus the intersection of the two curves $\tilde{C}$ and $\tilde{K}$ that we'll plot in Figure Fig. 102.1 below.

We can compute K_s by solving the equation $K_s = \tilde{K}\left(\tilde{C}\left(K_s\right)\right)$

```
@jit
def K_tilde_diff(K, pp):

    K_out = K_tilde(C_tilde(K, pp), pp)

    return K - K_out
```

```
res = brentq(K_tilde_diff, 8, 10, args=(pp,))

Ks = res.root
Cs = C_tilde(Ks, pp)

Ks, Cs
```

```
(9.575838163314447, 1.9160839808123402)
```

We can use the shooting algorithm to compute trajectories that approach (K_s, C_s).

For a given K, let's compute $\vec{C}$ and $\vec{K}$ for a large T , e.g., $= 200$.

We compute C_0 by the bisection algorithm that assures that $K_T = K_s$.

Let's compute two trajectories towards (K_s, C_s) that start from different sides of K_s: $\bar{K}_0 = 1e-3 < K_s < \bar{K}_1 = 15$.

```
c_vec1, k_vec1 = bisection(pp, 5, 15, T=200, k_ter=Ks)
c_vec2, k_vec2 = bisection(pp, 1e-3, 1e-3, T=200, k_ter=Ks)
```

```
Converged successfully on iteration  46
Converged successfully on iteration  51
```

The following code generates Figure Fig. 102.1, which is patterned on a graph that appears on page 411 of [Intriligator, 2002].

Figure Fig. 102.1 is a classic "phase plane" with "state" variable K on the ordinate axis and "co-state" variable C on the coordinate axis.

Figure Fig. 102.1 plots three curves:

- the blue line graphs $C = \tilde{C}(K)$ of fixed points described by equation (102.18).
- the red line graphs $K = \tilde{K}(C)$ of fixed points described by equation (102.19)
- the green line graphs the stable traced out by paths that converge to the steady state starting from an arbitrary K_0 at time 0.
  - for a given K_0, the shooting algorithm sets C_0 to the coordinate on the green line in order to initiate a path that converges to the optimal steady state
  - the arrows on the green line show the direction in which dynamics (102.16) push successive (K_{t+1}, C_t) pairs.

In addition to the three curves, Figure Fig. 102.1 plots arrows that point where the dynamics (102.16) drive the system when, for a given K_0, C_0 is not on the stable manifold depicted in the green line.

- If C_0 is set below the green line for a given K_0, too much capital is accumulated
- If C_0 is set above the green line for a given K_0, too little capital is accumulated

102.9 Concluding Remarks

In *Cass-Koopmans Competitive Equilibrium*, we study a decentralized version of an economy with exactly the same technology and preference structure as deployed here.

In that lecture, we replace the planner of this lecture with Adam Smith's **invisible hand**.

In place of quantity choices made by the planner, there are market prices that are set by a *deus ex machina* from outside the model, a so-called invisible hand.

Equilibrium market prices must reconcile distinct decisions that are made independently by a representative household and a representative firm.

The relationship between a command economy like the one studied in this lecture and a market economy like that studied in *Cass-Koopmans Competitive Equilibrium* is a foundational topic in general equilibrium theory and welfare economics.

102.9.1 Exercise

Exercise 102.9.1

- Plot the optimal consumption, capital, and saving paths when the initial capital level begins at 1.5 times the steady state level as we shoot towards the steady state at $T = 130$.
- Why does the saving rate respond as it does?

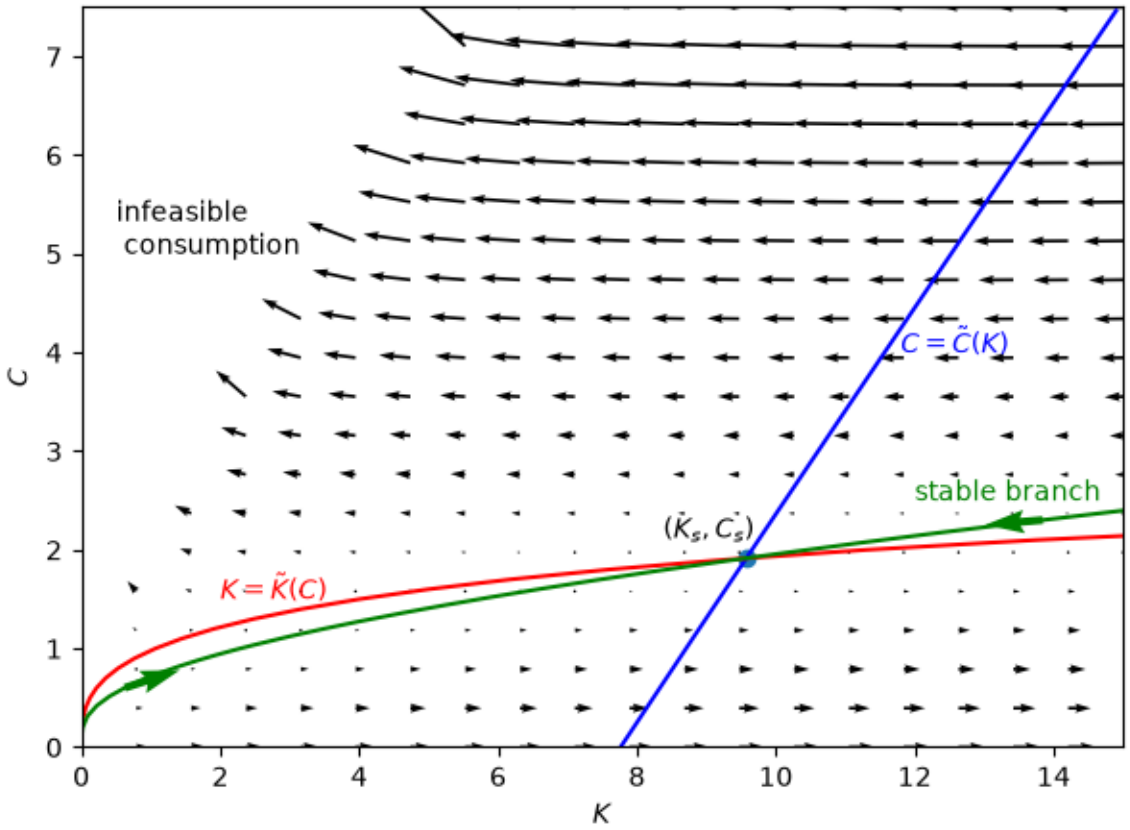
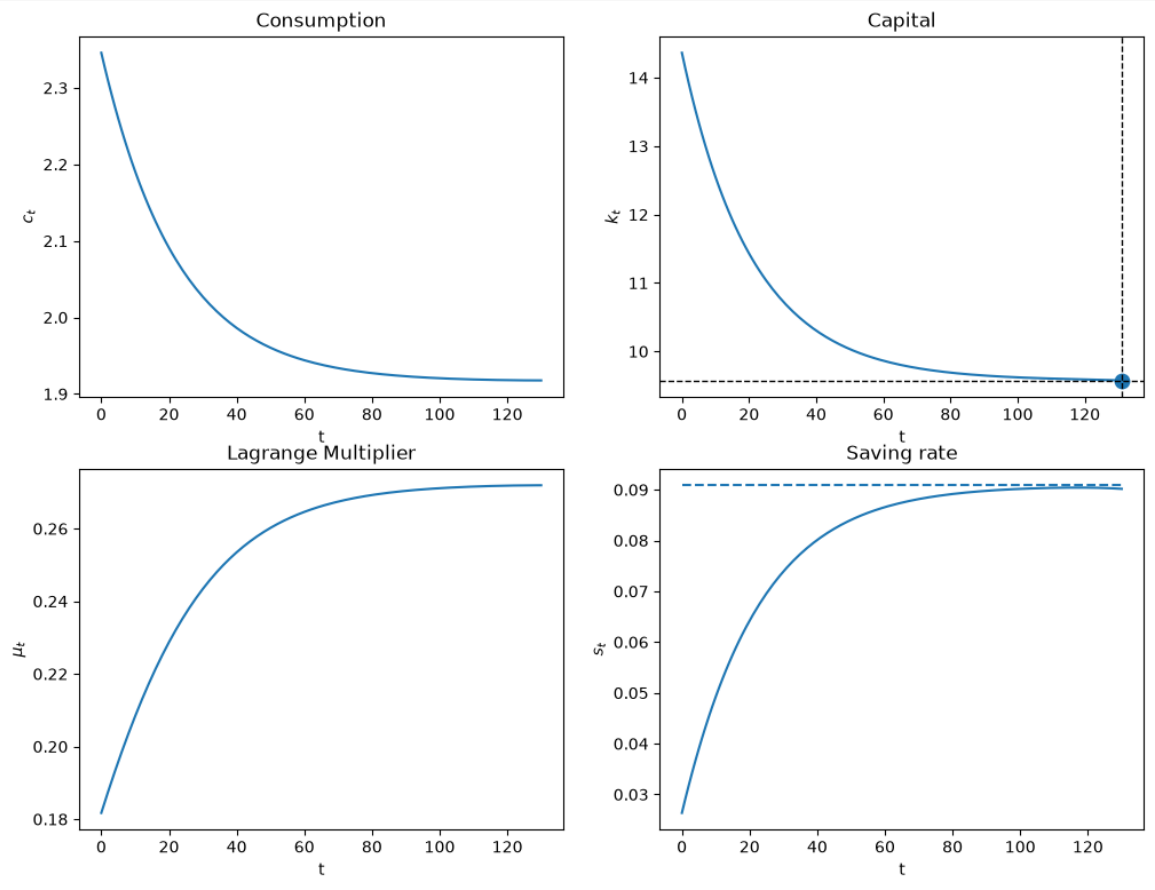


Fig. 102.1: Stable Manifold and Phase Plane

Solution

```
plot_saving_rate(pp, 0.3, k_ss*1.5, [130], k_ter=k_ss, k_ss=k_ss, s_ss=s_ss)
```



CHAPTER

THREE

CASS-KOOPMANS COMPETITIVE EQUILIBRIUM

Contents

- *Cass-Koopmans Competitive Equilibrium*
  - *Overview*
  - *Review of Cass-Koopmans Model*
  - *Competitive Equilibrium*
  - *Market Structure*
  - *Firm Problem*
  - *Household Problem*
  - *Computing a Competitive Equilibrium*
  - *Yield Curves and Hicks-Arrow Prices*

103.1 Overview

This lecture continues our analysis in this lecture *Cass-Koopmans Planning Model* about the model that Tjalling Koopmans [Koopmans, 1965] and David Cass [Cass, 1965] used to study optimal capital accumulation.

This lecture illustrates what is, in fact, a more general connection between a **planned economy** and an economy organized as a competitive equilibrium or a **market economy**.

The earlier lecture *Cass-Koopmans Planning Model* studied a planning problem and used ideas including

- A Lagrangian formulation of the planning problem that leads to a system of difference equations.
- A **shooting algorithm** for solving difference equations subject to initial and terminal conditions.
- A **turnpike** property that describes optimal paths for long-but-finite horizon economies.

The present lecture uses additional ideas including

- Hicks-Arrow prices, named after John R. Hicks and Kenneth Arrow.
- A connection between some Lagrange multipliers from the planning problem and the Hicks-Arrow prices.
- A **Big** K **, little** k trick widely used in macroeconomic dynamics.
  - We shall encounter this trick in this lecture and also in this lecture.

- A non-stochastic version of a theory of the **term structure of interest rates**.
- An intimate connection between two ways to organize an economy, namely:
  - **socialism** in which a central planner commands the allocation of resources, and
  - **competitive markets** in which competitive equilibrium **prices** induce individual consumers and producers to choose a socially optimal allocation as unintended consequences of their selfish decisions

Let's start with some standard imports:

```
import matplotlib.pyplot as plt
from numba import jit, float64
from numba.experimental import jitclass
import numpy as np
```

103.2 Review of Cass-Koopmans Model

The physical setting is identical with that in *Cass-Koopmans Planning Model*.

Time is discrete and takes values $t = 0, 1, \ldots, T$.

Output of a single good can either be consumed or invested in physical capital.

The capital good is durable but partially depreciates each period at a constant rate.

We let C_t be a nondurable consumption good at time t.

Let K_t be the stock of physical capital at time t.

Let $\vec{C} = \{C_0, \ldots, C_T\}$ and $\vec{K} = \{K_0, \ldots, K_{T+1}\}$.

A representative household is endowed with one unit of labor at each t and likes the consumption good at each t.

The representative household inelastically supplies a single unit of labor N_t at each t, so that $N_t = 1$ for all $t \in \{0, 1, \ldots, T\}$.

The representative household has preferences over consumption bundles ordered by the utility functional:

$$U(\vec{C}) = \sum_{t=0}^{T} \beta^t \frac{C_t^{1-\gamma}}{1-\gamma}$$

where $\beta \in (0, 1)$ is a discount factor and $\gamma > 0$ governs the curvature of the one-period utility function.

We assume that $K_0 > 0$.

There is an economy-wide production function

$$F(K_t, N_t) = AK_t^{\alpha} N_t^{1-\alpha}$$

with $0 < \alpha < 1$, $A > 0$.

A feasible allocation $\vec{C}, \vec{K}$ satisfies

$$C_t + K_{t+1} \leq F(K_t, N_t) + (1-\delta)K_t \quad \text{for all } t \in \{0, 1, \ldots, T\}$$

where $\delta \in (0, 1)$ is a depreciation rate of capital.

103.2.1 Planning Problem

In this lecture *Cass-Koopmans Planning Model*, we studied a problem in which a planner chooses an allocation $\{\vec{C}, \vec{K}\}$ to maximize (102.2) subject to (102.5).

The allocation that solves the planning problem reappears in a competitive equilibrium, as we shall see below.

103.3 Competitive Equilibrium

We now study a decentralized version of the economy.

It shares the same technology and preference structure as the planned economy studied in this lecture *Cass-Koopmans Planning Model*.

But now there is no planner.

There are (unit masses of) price-taking consumers and firms.

Market prices are set to reconcile distinct decisions that are made separately by a representative consumer and a representative firm.

There is a representative consumer who has the same preferences over consumption plans as did a consumer in the planned economy.

Instead of being told what to consume and save by a planner, a consumer (also known as a *household*) chooses for itself subject to a budget constraint.

- At each time t, the consumer receives wages and rentals of capital from a firm – these comprise its **income** at time t.
- The consumer decides how much income to allocate to consumption or to savings.
- The household can save either by acquiring additional physical capital (it trades one for one with time t consumption) or by acquiring claims on consumption at dates other than t.
- The household owns physical capital and labor and rents them to the firm.
- The household consumes, supplies labor, and invests in physical capital.
- A profit-maximizing representative firm operates the production technology.
- The firm rents labor and capital each period from the representative household and sells its output each period to the household.
- The representative household and the representative firm are both **price takers** who believe that prices are not affected by their choices

> **Note**
>
> Again, we can think of there being unit measures of identical representative consumers and identical representative firms.

103.4 Market Structure

The representative household and the representative firm are both price takers.

The household owns both factors of production, namely, labor and physical capital.

Each period, the firm rents both factors from the household.

There is a **single** grand competitive market in which a household trades date 0 goods for goods at all other dates $t = 1, 2, \ldots, T$.

103.4.1 Prices

There are sequences of prices $\{w_t, \eta_t\}_{t=0}^T = \{\vec{w}, \vec{\eta}\}$ where

- w_t is a wage, i.e., a rental rate, for labor at time t
- η_t is a rental rate for capital at time t

In addition there is a vector $\{q_t^0\}$ of intertemporal prices where

- q_t^0 is the price at time 0 of one unit of the good at date t.

We call $\{q_t^0\}_{t=0}^T$ a vector of **Hicks-Arrow prices**, named after the 1972 economics Nobel prize winners.

Because is a **relative price**. the unit of account in terms of which the prices q_t^0 are stated is; we are free to re-normalize them by multiplying all of them by a positive scalar, say $\lambda > 0$.

Units of q_t^0 could be set so that they are

$$\frac{\text{number of time 0 goods}}{\text{number of time t goods}}$$

In this case, we would be taking the time 0 consumption good to be the **numeraire**.

103.5 Firm Problem

At time t a representative firm hires labor $\tilde{n}_t$ and capital $\tilde{k}_t$.

The firm's profits at time t are

$$F(\tilde{k}_t, \tilde{n}_t) - w_t \tilde{n}_t - \eta_t \tilde{k}_t$$

where w_t is a wage rate at t and η_t is the rental rate on capital at t.

As in the planned economy model

$$F(\tilde{k}_t, \tilde{n}_t) = A \tilde{k}_t^\alpha \tilde{n}_t^{1-\alpha}$$

103.5.1 Zero Profit Conditions

Zero-profits conditions for capital and labor are

$$F_k(\tilde{k}_t, \tilde{n}_t) = \eta_t$$

and

$$F_n(\tilde{k}_t, \tilde{n}_t) = w_t \tag{103.1}$$

These conditions emerge from a no-arbitrage requirement.

To describe this no-arbitrage profits reasoning, we begin by applying a theorem of Euler about linearly homogenous functions.

The theorem applies to the Cobb-Douglas production function because we it displays constant returns to scale:

$$\alpha F(\tilde{k}_t, \tilde{n}_t) = F(\alpha\tilde{k}_t, \alpha\tilde{n}_t)$$

for $\alpha \in (0, 1)$.

Taking partial derivatives $\frac{\partial}{\partial\alpha}$ on both sides of the above equation gives

$$F(\tilde{k}_t, \tilde{n}_t) = \frac{\partial F}{\partial \tilde{k}_t}\tilde{k}_t + \frac{\partial F}{\partial \tilde{n}_t}\tilde{n}_t$$

Rewrite the firm's profits as

$$\frac{\partial F}{\partial \tilde{k}_t}\tilde{k}_t + \frac{\partial F}{\partial \tilde{n}_t}\tilde{n}_t - w_t\tilde{n}_t - \eta_t k_t$$

or

$$\left(\frac{\partial F}{\partial \tilde{k}_t} - \eta_t\right)\tilde{k}_t + \left(\frac{\partial F}{\partial \tilde{n}_t} - w_t\right)\tilde{n}_t$$

Because F is homogeneous of degree 1, it follows that $\frac{\partial F}{\partial \tilde{k}_t}$ and $\frac{\partial F}{\partial \tilde{n}_t}$ are homogeneous of degree 0 and therefore fixed with respect to $\tilde{k}_t$ and $\tilde{n}_t$.

If $\frac{\partial F}{\partial \tilde{k}_t} > \eta_t$, then the firm makes positive profits on each additional unit of $\tilde{k}_t$, so it would want to make $\tilde{k}_t$ arbitrarily large.

But setting $\tilde{k}_t = +\infty$ is not physically feasible, so **equilibrium** prices must take values that present the firm with no such arbitrage opportunity.

A similar argument applies if $\frac{\partial F}{\partial \tilde{n}_t} > w_t$.

If $\frac{\partial \tilde{k}_t}{\partial \tilde{k}_t} < \eta_t$, the firm would want to set $\tilde{k}_t$ to zero, which is not feasible.

It is convenient to define $\vec{w} = \{w_0, \ldots, w_T\}$ and $\vec{\eta} = \{\eta_0, \ldots, \eta_T\}$.

103.6 Household Problem

A representative household lives at $t = 0, 1, \ldots, T$.

At t, the household rents 1 unit of labor and k_t units of capital to a firm and receives income

$$w_t 1 + \eta_t k_t$$

At t the household allocates its income to the following purchases between the following two categories:

- consumption c_t
- net investment $k_{t+1} - (1-\delta)k_t$

Here $(k_{t+1} - (1-\delta)k_t)$ is the household's net investment in physical capital and $\delta \in (0,1)$ is again a depreciation rate of capital.

In period t, the consumer is free to purchase more goods to be consumed and invested in physical capital than its income from supplying capital and labor to the firm, provided that in some other periods its income exceeds its purchases.

A consumer's net excess demand for time t consumption goods is the gap

$$e_t \equiv (c_t + (k_{t+1} - (1-\delta)k_t)) - (w_t 1 + \eta_t k_t)$$

Let $\vec{c} = \{c_0, \ldots, c_T\}$ and let $\vec{k} = \{k_1, \ldots, k_{T+1}\}$.

k_0 is given to the household.

The household faces a **single** budget constraint that requires that the present value of the household's net excess demands must be zero:

$$\sum_{t=0}^{T} q_t^0 e_t \leq 0$$

or

$$\sum_{t=0}^{T} q_t^0 \left(c_t + (k_{t+1} - (1-\delta)k_t)\right) \leq \sum_{t=0}^{T} q_t^0 (w_t 1 + \eta_t k_t)$$

The household faces price system $\{q_t^0, w_t, \eta_t\}$ as a price-taker and chooses an allocation to solve the constrained optimization problem:

$$\max_{\vec{c},\vec{k}} \sum_{t=0}^{T} \beta^t u(c_t)$$

$$\text{subject to } \sum_{t=0}^{T} q_t^0 \left(c_t + (k_{t+1} - (1-\delta)k_t) - (w_t - \eta_t k_t)\right) \leq 0$$

Components of a **price system** have the following units:

- w_t is measured in units of the time t good per unit of time t labor hired
- η_t is measured in units of the time t good per unit of time t capital hired
- q_t^0 is measured in units of a numeraire per unit of the time t good

103.6.1 Definitions

- A **price system** is a sequence $\{q_t^0, \eta_t, w_t\}_{t=0}^T = \{\vec{q}, \vec{\eta}, \vec{w}\}$.
- An **allocation** is a sequence $\{c_t, k_{t+1}, n_t = 1\}_{t=0}^T = \{\vec{c}, \vec{k}, \vec{n}\}$.
- A **competitive equilibrium** is a price system and an allocation with the following properties:
  - Given the price system, the allocation solves the household's problem.
  - Given the price system, the allocation solves the firm's problem.

The vision here is that an equilibrium price system and allocation are determined once and for all.

In effect, we imagine that all trades occur just before time 0.

103.7 Computing a Competitive Equilibrium

We compute a competitive equilibrium by using a **guess and verify** approach.

- We **guess** equilibrium price sequences $\{\vec{q}, \vec{\eta}, \vec{w}\}$.
- We then **verify** that at those prices, the household and the firm choose the same allocation.

103.7.1 Guess for Price System

In this lecture *Cass-Koopmans Planning Model*, we computed an allocation $\{\vec{C}, \vec{K}, \vec{N}\}$ that solves a planning problem.

We use that allocation to construct a guess for the equilibrium price system.

> **Note**
>
> This allocation will constitute the **Big** K to be in the present instance of the **Big** K **, little** k trick that we'll apply to a competitive equilibrium in the spirit of this lecture and this lecture.

In particular, we shall use the following procedure:

- obtain first-order conditions for the representative firm and the representative consumer.
- from these equations, obtain a new set of equations by replacing the firm's choice variables $\tilde{k}, \tilde{n}$ and the consumer's choice variables with the quantities $\vec{C}, \vec{K}$ that solve the planning problem.
- solve the resulting equations for $\{\vec{q}, \vec{\eta}, \vec{w}\}$ as functions of $\vec{C}, \vec{K}$.
- verify that at these prices, $c_t = C_t, k_t = \tilde{k}_t = K_t, \tilde{n}_t = 1$ for $t = 0, 1, \ldots, T$.

Thus, we guess that for $t = 0, \ldots, T$:

$$q_t^0 = \beta^t u'(C_t) \tag{103.2}$$

$$w_t = f(K_t) - K_t f'(K_t) \tag{103.3}$$

$$\eta_t = f'(K_t) \tag{103.4}$$

At these prices, let capital chosen by the household be

$$k_t^*(\vec{q}, \vec{w}, \vec{\eta}), \quad t \geq 0 \tag{103.5}$$

and let the allocation chosen by the firm be

$$\tilde{k}_t^*(\vec{q}, \vec{w}, \vec{\eta}), \quad t \geq 0$$

and so on.

If our guess for the equilibrium price system is correct, then it must occur that

$$k_t^* = \tilde{k}_t^* \tag{103.6}$$

$$1 = \tilde{n}_t^* \tag{103.7}$$

$$c_t^* + k_{t+1}^* - (1-\delta)k_t^* = F(\tilde{k}_t^*, \tilde{n}_t^*)$$

We shall verify that for $t = 0, \ldots, T$ allocations chosen by the household and the firm both equal the allocation that solves the planning problem:

$$k_t^* = \tilde{k}_t^* = K_t, \tilde{n}_t = 1, c_t^* = C_t \tag{103.8}$$

103.7.2 Verification Procedure

Our approach is firsts to stare at first-order necessary conditions for optimization problems of the household and the firm.

At the price system we have guessed, we'll then verify that both sets of first-order conditions are satisfied at the allocation that solves the planning problem.

103.7.3 Household's Lagrangian

To solve the household's problem, we formulate the Lagrangian

$$\mathcal{L}(\vec{c}, \vec{k}, \lambda) = \sum_{t=0}^{T} \beta^t u(c_t) + \lambda \left(\sum_{t=0}^{T} q_t^0 \left(((1-\delta)k_t - w_t) + \eta_t k_t - c_t - k_{t+1} \right) \right)$$

and attack the min-max problem:

$$\min_{\lambda} \max_{\vec{c}, \vec{k}} \mathcal{L}(\vec{c}, \vec{k}, \lambda)$$

First-order conditions are

$$c_t : \quad \beta^t u'(c_t) - \lambda q_t^0 = 0 \quad t = 0, 1, \ldots, T \tag{103.9}$$

$$k_t : \quad -\lambda q_t^0 \left[(1-\delta) + \eta_t \right] + \lambda q_{t-1}^0 = 0 \quad t = 1, 2, \ldots, T+1 \tag{103.10}$$

$$\lambda : \quad \left(\sum_{t=0}^{T} q_t^0 \left(c_t + (k_{t+1} - (1-\delta)k_t) - w_t - \eta_t k_t \right) \right) \leq 0 \tag{103.11}$$

$$k_{T+1} : \quad -\lambda q_0^{T+1} \leq 0, \ \leq 0 \text{ if } k_{T+1} = 0; \ = 0 \text{ if } k_{T+1} > 0 \tag{103.12}$$

Now we plug in our guesses of prices and do some algebra in the hope of recovering all first-order necessary conditions (102.9)-(102.12) for the planning problem from this lecture *Cass-Koopmans Planning Model*.

Combining (103.9) and (103.2), we get:

$$u'(C_t) = \mu_t$$

which is (102.9).

Combining (103.10), (103.2), and (103.4), we get:

$$-\lambda \beta^t \mu_t \left[(1-\delta) + f'(K_t) \right] + \lambda \beta^{t-1} \mu_{t-1} = 0 \tag{103.13}$$

Rewriting (103.13) by dividing by λ on both sides (which is nonzero since u'>0) we get:

$$\beta^t \mu_t [(1 - \delta + f'(K_t)] = \beta^{t-1} \mu_{t-1}$$

or

$$\beta \mu_t [(1 - \delta + f'(K_t)] = \mu_{t-1}$$

which is (102.10).

Combining (103.11), (103.2), (103.3) and (103.4) after multiplying both sides of (103.11) by λ, we get

$$\sum_{t=0}^{T} \beta^t \mu_t \left(C_t + (K_{t+1} - (1-\delta)K_t) - f(K_t) + K_t f'(K_t) - f'(K_t) K_t \right) \leq 0$$

which simplifies to

$$\sum_{t=0}^{T} \beta^t \mu_t \left(C_t + K_{t+1} - (1-\delta) K_t - F(K_t, 1) \right) \leq 0$$

Since $\beta^t \mu_t > 0$ for $t = 0, \ldots, T$, it follows that

$$C_t + K_{t+1} - (1-\delta) K_t - F(K_t, 1) = 0 \quad \text{for all } t \text{ in } \{0, 1, \ldots, T\}$$

which is (102.11).

Combining (103.12) and (103.2), we get:

$$-\beta^{T+1} \mu_{T+1} \leq 0$$

Dividing both sides by β^{T+1} gives

$$-\mu_{T+1} \leq 0$$

which is (102.12) for the planning problem.

Thus, at our guess of the equilibrium price system, the allocation that solves the planning problem also solves the problem faced by a representative household living in a competitive equilibrium.

103.7.4 Representative Firm's Problem

We now turn to the problem faced by a firm in a competitive equilibrium:

If we plug (103.8) into (103.1) for all t, we get

$$\frac{\partial F(K_t, 1)}{\partial K_t} = f'(K_t) = \eta_t$$

which is (103.4).

If we now plug (103.8) into (103.1) for all t, we get:

$$\frac{\partial F(\tilde{K}_t, 1)}{\partial \tilde{L}_t} = f(K_t) - f'(K_t) K_t = w_t$$

which is exactly (103.5).

Thus, at our guess for the equilibrium price system, the allocation that solves the planning problem also solves the problem faced by a firm within a competitive equilibrium.

By (103.6) and (103.7) this allocation is identical to the one that solves the consumer's problem.

Note

Because budget sets are affected only by relative prices, $\{q_t^0\}$ is determined only up to multiplication by a positive constant.

Normalization: We are free to choose a $\{q_t^0\}$ that makes $\lambda = 1$ so that we are measuring q_t^0 in units of the marginal utility of time 0 goods.

We will plot q, w, η below to show these equilibrium prices induce the same aggregate movements that we saw earlier in the planning problem.

To proceed, we bring in Python code that *Cass-Koopmans Planning Model* used to solve the planning problem

First let's define a `jitclass` that stores parameters and functions the characterize an economy.

```
planning_data = [
    ('γ', float64),     # Coefficient of relative risk aversion
    ('β', float64),     # Discount factor
    ('δ', float64),     # Depreciation rate on capital
    ('α', float64),     # Return to capital per capita
    ('A', float64)      # Technology
]
```

```
@jitclass(planning_data)
class PlanningProblem():

    def __init__(self, γ=2, β=0.95, δ=0.02, α=0.33, A=1):

        self.γ, self.β = γ, β
        self.δ, self.α, self.A = δ, α, A

    def u(self, c):
        '''
        Utility function
        ASIDE: If you have a utility function that is hard to solve by hand
        you can use automatic or symbolic differentiation
        See https://github.com/HIPS/autograd
        '''
        γ = self.γ

        return c ** (1 - γ) / (1 - γ) if γ!= 1 else np.log(c)

    def u_prime(self, c):
        'Derivative of utility'
        γ = self.γ

        return c ** (-γ)

    def u_prime_inv(self, c):
        'Inverse of derivative of utility'
        γ = self.γ

        return c ** (-1 / γ)

    def f(self, k):
        'Production function'
        α, A = self.α, self.A

        return A * k ** α

    def f_prime(self, k):
        'Derivative of production function'
        α, A = self.α, self.A

        return α * A * k ** (α - 1)

    def f_prime_inv(self, k):
        'Inverse of derivative of production function'
        α, A = self.α, self.A

        return (k / (A * α)) ** (1 / (α - 1))
```

(continues on next page)

(continued from previous page)

```
    def next_k_c(self, k, c):
        '''
        Given the current capital Kt and an arbitrary feasible
        consumption choice Ct, computes Kt+1 by state transition law
        and optimal Ct+1 by Euler equation.
        '''
        β, δ = self.β, self.δ
        u_prime, u_prime_inv = self.u_prime, self.u_prime_inv
        f, f_prime = self.f, self.f_prime

        k_next = f(k) + (1 - δ) * k - c
        c_next = u_prime_inv(u_prime(c) / (β * (f_prime(k_next) + (1 - δ))))

        return k_next, c_next
```

```
@jit
def shooting(pp, c0, k0, T=10):
    '''
    Given the initial condition of capital k0 and an initial guess
    of consumption c0, computes the whole paths of c and k
    using the state transition law and Euler equation for T periods.
    '''
    if c0 > pp.f(k0):
        print("initial consumption is not feasible")

        return None

    # initialize vectors of c and k
    c_vec = np.empty(T+1)
    k_vec = np.empty(T+2)

    c_vec[0] = c0
    k_vec[0] = k0

    for t in range(T):
        k_vec[t+1], c_vec[t+1] = pp.next_k_c(k_vec[t], c_vec[t])

    k_vec[T+1] = pp.f(k_vec[T]) + (1 - pp.δ) * k_vec[T] - c_vec[T]

    return c_vec, k_vec
```

```
@jit
def bisection(pp, c0, k0, T=10, tol=1e-4, max_iter=500, k_ter=0, verbose=True):

    # initial boundaries for guess c0
    c0_upper = pp.f(k0)
    c0_lower = 0

    i = 0
    while True:
        c_vec, k_vec = shooting(pp, c0, k0, T)
        error = k_vec[-1] - k_ter

        # check if the terminal condition is satisfied
        if np.abs(error) < tol:
            if verbose:
```

(continues on next page)

(continued from previous page)

```
                print('Converged successfully on iteration ', i+1)
            return c_vec, k_vec

        i += 1
        if i == max_iter:
            if verbose:
                print('Convergence failed.')
            return c_vec, k_vec

        # if iteration continues, updates boundaries and guess of c0
        if error > 0:
            c0_lower = c0
        else:
            c0_upper = c0

        c0 = (c0_lower + c0_upper) / 2
```

```
pp = PlanningProblem()

# Steady states
ρ = 1 / pp.β - 1
k_ss = pp.f_prime_inv(ρ+pp.δ)
c_ss = pp.f(k_ss) - pp.δ * k_ss
```

The above code from this lecture *Cass-Koopmans Planning Model* lets us compute an optimal allocation for the planning problem.

- from the preceding analysis, we know that it will also be an allocation associated with a competitive equilibium.

Now we're ready to bring in Python code that we require to compute additional objects that appear in a competitive equilibrium.

```
@jit
def q(pp, c_path):
    # Here we choose numeraire to be u'(c_0) -- this is q^(t_0)_t
    T = len(c_path) - 1
    q_path = np.ones(T+1)
    q_path[0] = 1
    for t in range(1, T+1):
        q_path[t] = pp.β ** t * pp.u_prime(c_path[t])
    return q_path

@jit
def w(pp, k_path):
    w_path = pp.f(k_path) - k_path * pp.f_prime(k_path)
    return w_path

@jit
def η(pp, k_path):
    η_path = pp.f_prime(k_path)
    return η_path
```

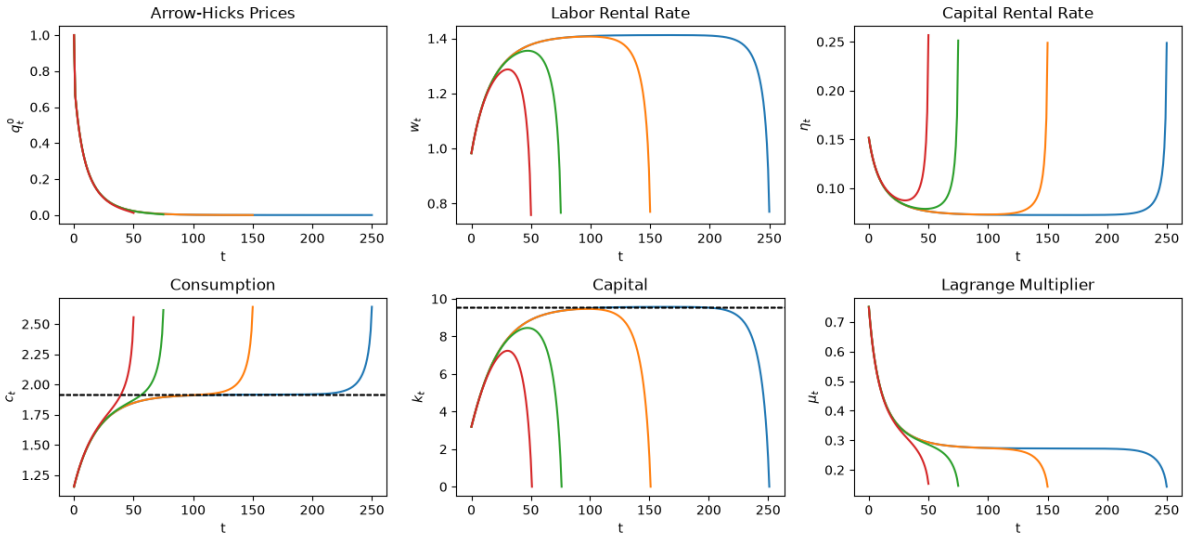
Now we calculate and plot for each T

```
T_arr = [250, 150, 75, 50]

fix, axs = plt.subplots(2, 3, figsize=(13, 6))
```

(continues on next page)

(continued from previous page)

```
titles = ['Arrow-Hicks Prices', 'Labor Rental Rate', 'Capital Rental Rate',
          'Consumption', 'Capital', 'Lagrange Multiplier']
ylabels = ['$q_t^0$', '$w_t$', r'$\eta_t$', '$c_t$', '$k_t$', r'$\mu_t$']

for T in T_arr:
    c_path, k_path = bisection(pp, 0.3, k_ss/3, T, verbose=False)
    μ_path = pp.u_prime(c_path)

    q_path = q(pp, c_path)
    w_path = w(pp, k_path)[:-1]
    η_path = η(pp, k_path)[:-1]
    paths = [q_path, w_path, η_path, c_path, k_path, μ_path]

    for i, ax in enumerate(axs.flatten()):
        ax.plot(paths[i])
        ax.set(title=titles[i], ylabel=ylabels[i], xlabel='t')
        if titles[i] == 'Capital':
            ax.axhline(k_ss, lw=1, ls='--', c='k')
        if titles[i] == 'Consumption':
            ax.axhline(c_ss, lw=1, ls='--', c='k')

plt.tight_layout()
plt.show()
```



Varying Curvature

Now we see how our results change if we keep T constant, but allow the curvature parameter, γ to vary, starting with K_0 below the steady state.

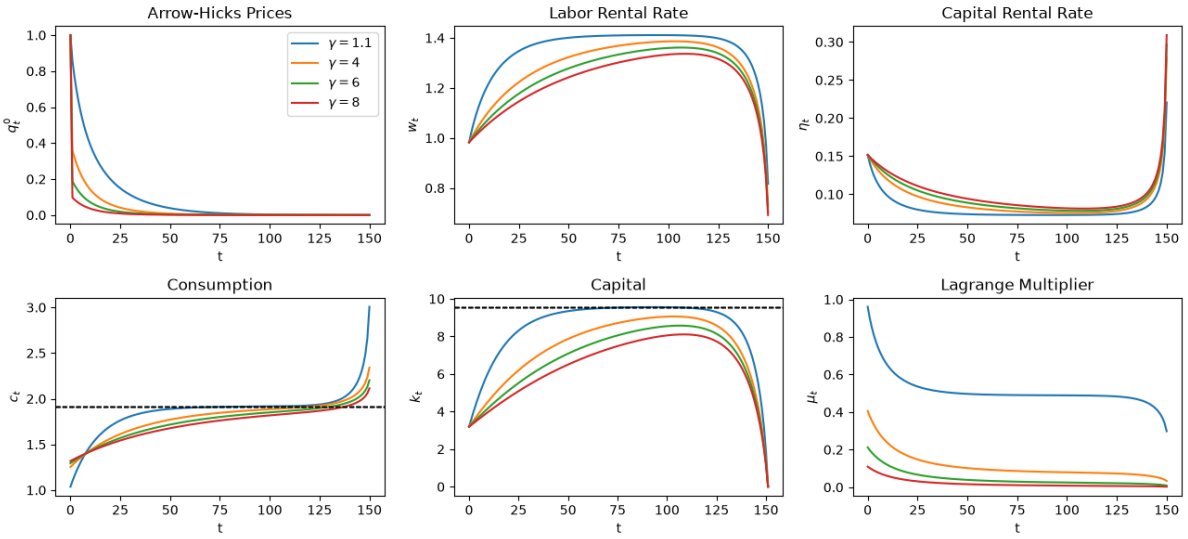
We plot the results for $T = 150$

```
T = 150
γ_arr = [1.1, 4, 6, 8]

fix, axs = plt.subplots(2, 3, figsize=(13, 6))
```

(continues on next page)

(continued from previous page)

```
for γ in γ_arr:
    pp_γ = PlanningProblem(γ=γ)
    c_path, k_path = bisection(pp_γ, 0.3, k_ss/3, T, verbose=False)
    μ_path = pp_γ.u_prime(c_path)

    q_path = q(pp_γ, c_path)
    w_path = w(pp_γ, k_path)[:-1]
    η_path = η(pp_γ, k_path)[:-1]
    paths = [q_path, w_path, η_path, c_path, k_path, μ_path]

    for i, ax in enumerate(axs.flatten()):
        ax.plot(paths[i], label=fr'$\gamma = {γ}$')
        ax.set(title=titles[i], ylabel=ylabels[i], xlabel='t')
        if titles[i] == 'Capital':
            ax.axhline(k_ss, lw=1, ls='--', c='k')
        if titles[i] == 'Consumption':
            ax.axhline(c_ss, lw=1, ls='--', c='k')

axs[0, 0].legend()
plt.tight_layout()
plt.show()
```



Adjusting γ means adjusting how much individuals prefer to smooth consumption.

Higher γ means individuals prefer to smooth more resulting in slower convergence to a steady state allocation.

Lower γ means individuals prefer to smooth less, resulting in faster convergence to a steady state allocation.

103.8 Yield Curves and Hicks-Arrow Prices

We return to Hicks-Arrow prices and calculate how they are related to **yields** on loans of alternative maturities.

This will let us plot a **yield curve** that graphs yields on bonds of maturities $j = 1, 2, ...$ against $j = 1, 2, ...$

We use the following formulas.

A **yield to maturity** on a loan made at time t_0 that matures at time $t > t_0$

$$r_{t_0,t} = -\frac{\log q_t^{t_0}}{t - t_0}$$

A Hicks-Arrow price system for a base-year $t_0 \leq t$ satisfies

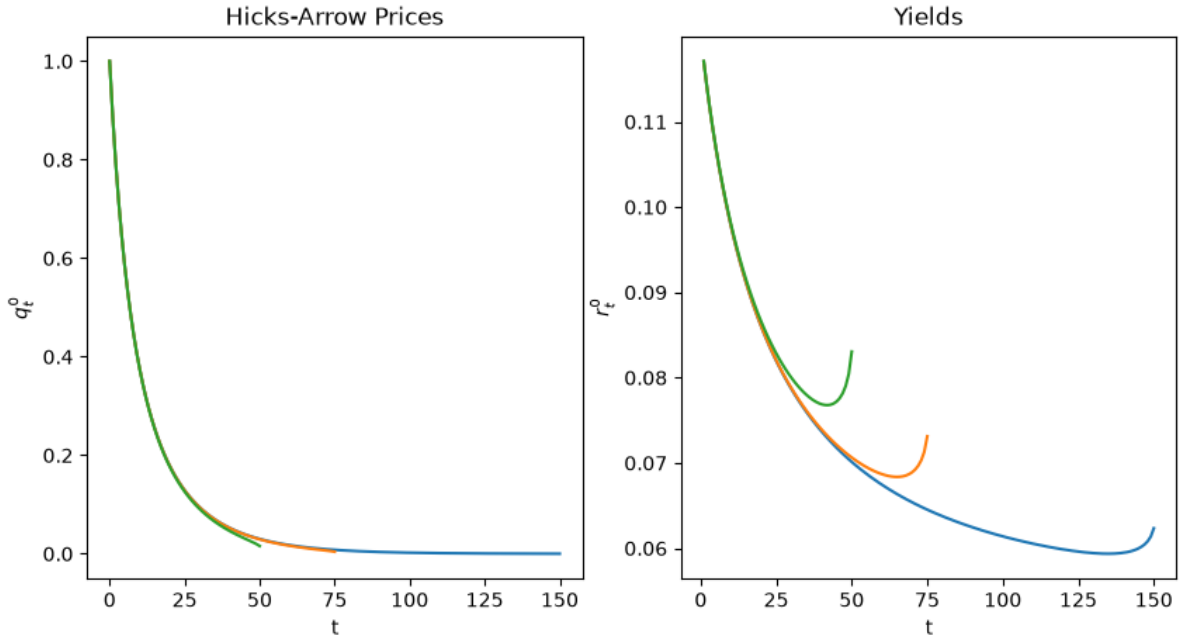
$$q_t^{t_0} = \beta^{t-t_0} \frac{u'(c_t)}{u'(c_{t_0})} = \beta^{t-t_0} \frac{c_t^{-\gamma}}{c_{t_0}^{-\gamma}}$$

We redefine our function for q to allow arbitrary base years, and define a new function for r, then plot both.

We begin by continuing to assume that $t_0 = 0$ and plot things for different maturities $t = T$, with K_0 below the steady state

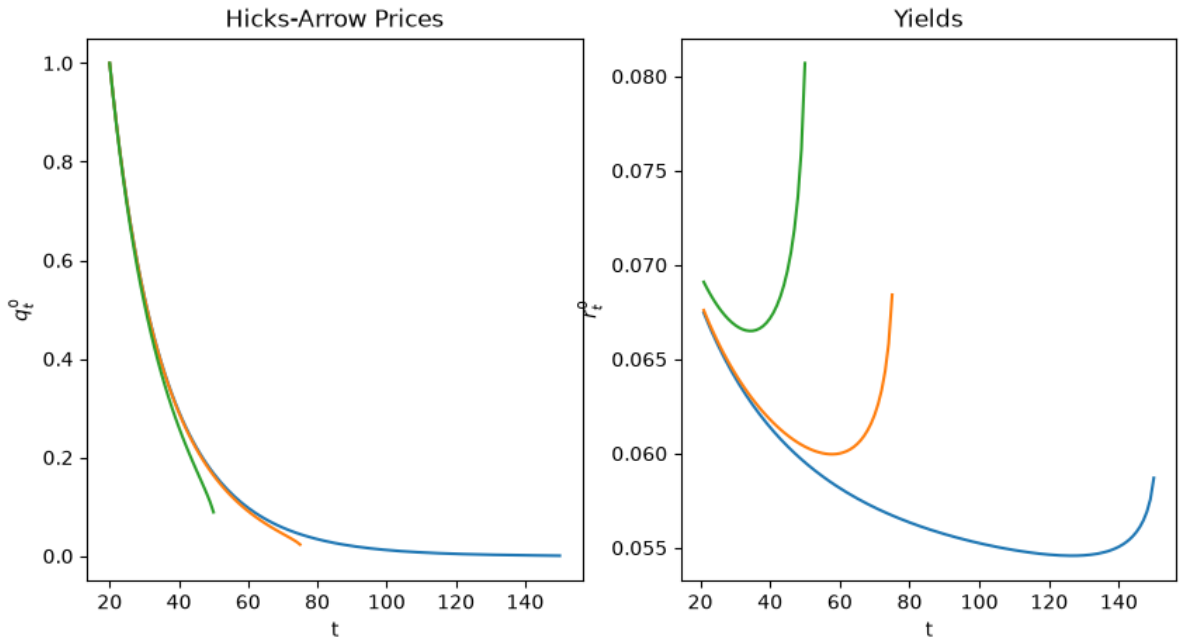
```
@jit
def q_generic(pp, t0, c_path):
    # simplify notations
    β = pp.β
    u_prime = pp.u_prime

    T = len(c_path) - 1
    q_path = np.zeros(T+1-t0)
    q_path[0] = 1
    for t in range(t0+1, T+1):
        q_path[t-t0] = β ** (t-t0) * u_prime(c_path[t]) / u_prime(c_path[t0])
    return q_path

@jit
def r(pp, t0, q_path):
    '''Yield to maturity'''
    r_path = - np.log(q_path[1:]) / np.arange(1, len(q_path))
    return r_path

def plot_yield_curves(pp, t0, c0, k0, T_arr):

    fig, axs = plt.subplots(1, 2, figsize=(10, 5))

    for T in T_arr:
        c_path, k_path = bisection(pp, c0, k0, T, verbose=False)
        q_path = q_generic(pp, t0, c_path)
        r_path = r(pp, t0, q_path)

        axs[0].plot(range(t0, T+1), q_path)
        axs[0].set(xlabel='t', ylabel='$q_t^0$', title='Hicks-Arrow Prices')

        axs[1].plot(range(t0+1, T+1), r_path)
        axs[1].set(xlabel='t', ylabel='$r_t^0$', title='Yields')
```

```
T_arr = [150, 75, 50]
plot_yield_curves(pp, 0, 0.3, k_ss/3, T_arr)
```



Now we plot when $t_0 = 20$

```
plot_yield_curves(pp, 20, 0.3, k_ss/3, T_arr)
```



CHAPTER

FOUR

CASS-KOOPMANS MODEL WITH DISTORTING TAXES

104.1 Overview

This lecture studies effects of foreseen fiscal and technology shocks on competitive equilibrium prices and quantities in a nonstochastic version of a Cass-Koopmans growth model with features described in this QuantEcon lecture *Cass-Koopmans Competitive Equilibrium*.

This model is discussed in more detail in chapter 11 of [Ljungqvist and Sargent, 2018].

We use the model as a laboratory to experiment with numerical techniques for approximating equilibria and to display the structure of dynamic models in which decision makers have perfect foresight about future government decisions.

Following a classic paper by Robert E. Hall [Hall, 1971], we augment a nonstochastic version of the Cass-Koopmans optimal growth model with a government that purchases a stream of goods and that finances its purchases with an sequences of several distorting flat-rate taxes.

Distorting taxes prevent a competitive equilibrium allocation from solving a planning problem.

Therefore, to compute an equilibrium allocation and price system, we solve a system of nonlinear difference equations consisting of the first-order conditions for decision makers and the other equilibrium conditions.

We present two ways to approximate an equilibrium:

- The first is a shooting algorithm like the one that we deployed in *Cass-Koopmans Competitive Equilibrium*.
- The second method is a root-finding algorithm that minimizes residuals from the first-order conditions of the consumer and representative firm.

104.2 The Economy

104.2.1 Technology

Feasible allocations satisfy

$$g_t + c_t + x_t \leq F(k_t, n_t), \tag{104.1}$$

where

- g_t is government purchases of the time t good
- x_t is gross investment, and
- $F(k_t, n_t)$ is a linearly homogeneous production function with positive and decreasing marginal products of capital k_t and labor n_t.

Physical capital evolves according to

$$k_{t+1} = (1-\delta)k_t + x_t,$$

where $\delta \in (0,1)$ is a depreciation rate.

It is sometimes convenient to eliminate x_t from (104.1) and to represent it as as

$$g_t + c_t + k_{t+1} \leq F(k_t, n_t) + (1-\delta)k_t.$$

104.2.2 Components of a competitive equilibrium

All trades occurring at time 0.

The representative household owns capital, makes investment decisions, and rents capital and labor to a representative production firm.

The representative firm uses capital and labor to produce goods with the production function $F(k_t, n_t)$.

A **price system** is a triple of sequences $\{q_t, \eta_t, w_t\}_{t=0}^{\infty}$, where

- q_t is the time 0 pretax price of one unit of investment or consumption at time t (x_t or c_t),
- η_t is the pretax price at time t that the household receives from the firm for renting capital at time t, and
- w_t is the pretax price at time t that the household receives for renting labor to the firm at time t.

The prices w_t and η_t are expressed in terms of time t goods, while q_t is expressed in terms of a numeraire at time 0, as in *Cass-Koopmans Competitive Equilibrium*.

The presence of a government distinguishes this lecture from *Cass-Koopmans Competitive Equilibrium*.

Government purchases of goods at time t are $g_t \geq 0$.

A government expenditure plan is a sequence $g = \{g_t\}_{t=0}^{\infty}$.

A government tax plan is a 4-tuple of sequences $\{\tau_{ct}, \tau_{kt}, \tau_{nt}, \tau_{ht}\}_{t=0}^{\infty}$, where

- τ_{ct} is a tax rate on consumption at time t,
- τ_{kt} is a tax rate on rentals of capital at time t,
- τ_{nt} is a tax rate on wage earnings at time t, and
- τ_{ht} is a lump sum tax on a consumer at time t.

Because lump-sum taxes τ_{ht} are available, the government actually should not use any distorting taxes.

Nevertheless, we include all of these taxes because, like [Hall, 1971], they allow us to analyze how various taxes distort production and consumption decisions.

In the *experiment section*, we shall see how variations in government tax plan affect the transition path and equilibrium.

104.2.3 Representative Household

A representative household has preferences over nonnegative streams of a single consumption good c_t and leisure $1 - n_t$ that are ordered by:

$$\sum_{t=0}^{\infty} \beta^t U(c_t, 1-n_t), \quad \beta \in (0,1), \tag{104.2}$$

where

- U is strictly increasing in c_t, twice continuously differentiable, and strictly concave with $c_t \geq 0$ and $n_t \in [0, 1]$.

The representative hßousehold maximizes (104.2) subject to the single budget constraint:

$$\begin{aligned}\sum_{t=0}^{\infty} q_t \left\{ (1+\tau_{ct})c_t + \underbrace{[k_{t+1} - (1-\delta)k_t]}_{\text{no tax when investing}} \right\} \\ \leq \sum_{t=0}^{\infty} q_t \left\{ \eta_t k_t - \underbrace{\tau_{kt}(\eta_t - \delta)k_t}_{\text{tax on rental return}} + (1-\tau_{nt})w_t n_t - \tau_{ht} \right\}.\end{aligned} \tag{104.3}$$

Here we have assumed that the government gives a depreciation allowance δk_t from the gross rentals on capital $\eta_t k_t$ and so collects taxes $\tau_{kt}(\eta_t - \delta)k_t$ on rentals from capital.

104.2.4 Government

Government plans $\{g_t\}_{t=0}^{\infty}$ for government purchases and taxes $\{\tau_{ct}, \tau_{kt}, \tau_{nt}, \tau_{ht}\}_{t=0}^{\infty}$ must respect the budget constraint

$$\sum_{t=0}^{\infty} q_t g_t \leq \sum_{t=0}^{\infty} q_t \left\{ \tau_{ct} c_t + \tau_{kt}(\eta_t - \delta)k_t + \tau_{nt} w_t n_t + \tau_{ht} \right\}. \tag{104.4}$$

Given a budget-feasible government policy $\{g_t\}_{t=0}^{\infty}$ and $\{\tau_{ct}, \tau_{kt}, \tau_{nt}, \tau_{ht}\}_{t=0}^{\infty}$ subject to (104.4),

- **Household** chooses $\{c_t\}_{t=0}^{\infty}$, $\{n_t\}_{t=0}^{\infty}$, and $\{k_{t+1}\}_{t=0}^{\infty}$ to maximize utility(104.2) subject to budget constraint(104.3), and
- **Firm** chooses sequences of capital $\{k_t\}_{t=0}^{\infty}$ and $\{n_t\}_{t=0}^{\infty}$ to maximize profits

$$\sum_{t=0}^{\infty} q_t [F(k_t, n_t) - \eta_t k_t - w_t n_t] \tag{104.5}$$

- A **feasible allocation** is a sequence $\{c_t, x_t, n_t, k_t\}_{t=0}^{\infty}$ that satisfies feasibility condition (104.1).

104.3 Equilibrium

Definition 104.3.1

A **competitive equilibrium with distorting taxes** is a **budget-feasible government policy**, **a feasible allocation**, and **a price system** for which, given the price system and the government policy, the allocation solves the household's problem and the firm's problem.

104.4 No-arbitrage Condition

A no-arbitrage argument implies a restriction on prices and tax rates across time.

By rearranging (104.3) and group k_t at the same t, we can get

$$\begin{aligned}\sum_{t=0}^{\infty} q_t \left[(1+\tau_{ct})c_t\right] \leq \sum_{t=0}^{\infty} q_t (1-\tau_{nt}) w_t n_t - \sum_{t=0}^{\infty} q_t \tau_{ht} \\ + \sum_{t=1}^{\infty} \left\{ \left[(1-\tau_{kt})(\eta_t - \delta) + 1\right] q_t - q_{t-1} \right\} k_t \\ + \left[(1-\tau_{k0})(\eta_0 - \delta) + 1\right] q_0 k_0 - \lim_{T \to \infty} q_T k_{T+1}\end{aligned} \tag{104.6}$$

The household inherits a given k_0 that it takes as initial condition and is free to choose $\{c_t, n_t, k_{t+1}\}_{t=0}^{\infty}$.

The household's budget constraint (104.3) must be bounded in equilibrium due to finite resources.

This imposes a restriction on price and tax sequences.

Specifically, for $t \geq 1$, the terms multiplying k_t must equal zero.

If they were strictly positive (negative), the household could arbitrarily increase (decrease) the right-hand side of (104.3) by selecting an arbitrarily large positive (negative) k_t, leading to unbounded profit or arbitrage opportunities:

- For strictly positive terms, the household could purchase large capital stocks k_t and profit from their rental services and undepreciated value.
- For strictly negative terms, the household could engage in "short selling" synthetic units of capital. Both cases would make (104.3) unbounded.

Hence, by setting the terms multiplying k_t to 0 we have the non-arbitrage condition:

$$\frac{q_t}{q_{t+1}} = \left[(1 - \tau_{kt+1})(\eta_{t+1} - \delta) + 1\right]. \tag{104.7}$$

Moreover, we have terminal condition:

$$-\lim_{T \to \infty} q_T k_{T+1} = 0. \tag{104.8}$$

Zero-profit conditions for the representative firm impose additional restrictions on equilibrium prices and quantities.

The present value of the firm's profits is

$$\sum_{t=0}^{\infty} q_t \left[F(k_t, n_t) - w_t n_t - \eta_t k_t\right].$$

Applying Euler's theorem on linearly homogeneous functions to $F(k, n)$, the firm's present value is:

$$\sum_{t=0}^{\infty} q_t \left[(F_{kt} - \eta_t)k_t + (F_{nt} - w_t)n_t\right].$$

No-arbitrage (or zero-profit) conditions are:

$$\eta_t = F_{kt}, \quad w_t = F_{nt}. \tag{104.9}$$

104.5 Household's First Order Condition

Household maximize (104.2) under (104.3).

Let $U_1 = \frac{\partial U}{\partial c}, U_2 = \frac{\partial U}{\partial (1-n)} = -\frac{\partial U}{\partial n}$., we can derive FOC from the Lagrangian

$$\mathcal{L} = \sum_{t=0}^{\infty} \beta^t U(c_t, 1 - n_t) + \mu \left(\sum_{t=0}^{\infty} q_t \left[(1 + \tau_{ct})c_t - (1 - \tau_{nt})w_t n_t + ...\right] \right),$$

First-order necessary conditions for the representative household's problem are

$$\frac{\partial \mathcal{L}}{\partial c_t} = \beta^t U_1(c_t, 1 - n_t) - \mu q_t (1 + \tau_{ct}) = 0 \tag{104.10}$$

and

$$\frac{\partial \mathcal{L}}{\partial n_t} = \beta^t \left(-U_{2t}(c_t, 1 - n_t)\right) - \mu q_t (1 - \tau_{nt}) w_t = 0 \tag{104.11}$$

Rearranging (104.10) and (104.11), we have

$$\beta^t U_1(c_t, 1-n_t) = \beta^t U_{1t} = \mu q_t (1+\tau_{ct}), \tag{104.12}$$

$$\beta^t U_2(c_t, 1-n_t) = \beta^t U_{2t} = \mu q_t (1-\tau_{nt}) w_t. \tag{104.13}$$

Plugging (104.12) into (104.8) and replacing q_t, we get terminal condition

$$-\lim_{T\to\infty} \beta^T \frac{U_{1T}}{(1+\tau_{cT})} k_{T+1} = 0. \tag{104.14}$$

104.6 Computing Equilibria

To compute an equilibrium, we seek a price system $\{q_t, \eta_t, w_t\}$, a budget feasible government policy $\{g_t, \tau_t\} \equiv \{g_t, \tau_{ct}, \tau_{nt}, \tau_{kt}, \tau_{ht}\}$, and an allocation $\{c_t, n_t, k_{t+1}\}$ that solve a system of nonlinear difference equations consisting of

- feasibility condition (104.1), no-arbitrage condition for household (104.7) and firms (104.9), household's first order conditions (104.12) and (104.13).
- an initial condition k_0 and a terminal condition (104.14).

104.7 Python Code

We require the following imports

```
import numpy as np
from scipy.optimize import root
import matplotlib.pyplot as plt
from collections import namedtuple
from mpmath import mp, mpf
from warnings import warn

# Set the precision
mp.dps = 40
mp.pretty = True
```

We use the `mpmath` library to perform high-precision arithmetic in the shooting algorithm in cases where the solution diverges due to numerical instability.

Note

In the functions below, we include routines to handle the growth component, which will be discussed further in the section *Exogenous growth*.

We include them here to avoid code duplication.

We set the following parameters

```
# Create a namedtuple to store the model parameters
Model = namedtuple("Model",
                   ["β", "γ", "δ", "α", "A"])
```

(continues on next page)

(continued from previous page)

```
def create_model(β=0.95,  # discount factor
                 γ=2.0,   # relative risk aversion coefficient
                 δ=0.2,   # depreciation rate
                 α=0.33,  # capital share
                 A=1.0    # TFP
                 ):
    """Create a model instance."""
    return Model(β=β, γ=γ, δ=δ, α=α, A=A)

model = create_model()

# Total number of periods
S = 100
```

104.7.1 Inelastic Labor Supply

In this lecture, we consider the special case where $U(c, 1-n) = u(c)$ and $f(k) := F(k, 1)$.

We rewrite (104.1) with $f(k) := F(k, 1)$,

$$k_{t+1} = f(k_t) + (1-\delta)k_t - g_t - c_t. \tag{104.15}$$

```
def next_k(k_t, g_t, c_t, model, μ_t=1):
    """
    Capital next period: k_{t+1} = f(k_t) + (1 - δ) * k_t - c_t - g_t
    with optional growth adjustment: k_{t+1} = (f(k_t) + (1 - δ) * k_t - c_t - g_t) /␣
↪μ_{t+1}
    """
    return (f(k_t, model) + (1 - model.δ) * k_t - g_t - c_t) / μ_t
```

By the properties of a linearly homogeneous production function, we have $F_k(k, n) = f'(k)$ and $F_n(k, 1) = f(k, 1) - f'(k)k$.

Substituting (104.12), (104.9), and (104.15) into (104.7), we obtain the Euler equation

$$\begin{aligned}
&\frac{u'(f(k_t) + (1-\delta)k_t - g_t - k_{t+1})}{(1+\tau_{ct})} \\
&- \beta \frac{u'(f(k_{t+1}) + (1-\delta)k_{t+1} - g_{t+1} - k_{t+2})}{(1+\tau_{ct+1})} \\
&\times [(1-\tau_{kt+1})(f'(k_{t+1}) - \delta) + 1] = 0.
\end{aligned} \tag{104.16}$$

This can be simplified to:

$$u'(c_t) = \beta u'(c_{t+1}) \frac{(1+\tau_{ct})}{(1+\tau_{ct+1})} [(1-\tau_{kt+1})(f'(k_{t+1}) - \delta) + 1]. \tag{104.17}$$

Equation (104.17) will appear prominently in our equilibrium computation algorithms.

104.7.2 Steady state

Tax rates and government expenditures act as **forcing functions** for difference equations (104.15) and (104.17).

Define $z_t = [g_t, \tau_{kt}, \tau_{ct}]'$.

Represent the second-order difference equation as:

$$H(k_t, k_{t+1}, k_{t+2}; z_t, z_{t+1}) = 0. \tag{104.18}$$

We assume that a government policy reaches a steady state such that $\lim_{t\to\infty} z_t = \bar{z}$ and that the steady state prevails for $t > T$.

The terminal steady-state capital stock $\bar{k}$ satisfies:

$$H(\bar{k}, \bar{k}, \bar{k}, \bar{z}, \bar{z}) = 0.$$

From difference equation (104.17), we can infer a restriction on the steady-state:

$$\begin{aligned} u'(\bar{c}) &= \beta u'(\bar{c}) \frac{(1+\bar{\tau}_c)}{(1+\bar{\tau}_c)}[(1-\bar{\tau}_k)(f'(\bar{k}) - \delta) + 1]. \\ \implies 1 &= \beta[(1-\bar{\tau}_k)(f'(\bar{k}) - \delta) + 1]. \end{aligned} \tag{104.19}$$

104.7.3 Other equilibrium quantities and prices

Price:

$$q_t = \frac{\beta^t u'(c_t)}{u'(c_0)} \tag{104.20}$$

```
def compute_q_path(c_path, model, S=100, A_path=None):
    """
    Compute q path: q_t = (β^t * u'(c_t)) / u'(c_0)
    with optional A_path for growth models.
    """
    A = np.ones_like(c_path) if A_path is None else np.asarray(A_path)
    q_path = np.zeros_like(c_path)
    for t in range(S):
        q_path[t] = (model.β ** t *
                     u_prime(c_path[t], model, A[t])) / u_prime(c_path[0], model,␣
↪A[0])
    return q_path
```

Capital rental rate

$$\eta_t = f'(k_t)$$

```
def compute_η_path(k_path, model, S=100, A_path=None):
    """
    Compute η path: η_t = f'(k_t)
    with optional A_path for growth models.
    """
    A = np.ones_like(k_path) if A_path is None else np.asarray(A_path)
    η_path = np.zeros_like(k_path)
    for t in range(S):
        η_path[t] = f_prime(k_path[t], model, A[t])
    return η_path
```

Labor rental rate:

$$w_t = f(k_t) - k_t f'(k_t)$$

```
def compute_w_path(k_path, η_path, model, S=100, A_path=None):
    """
    Compute w path: w_t = f(k_t) - k_t * f'(k_t)
    with optional A_path for growth models.
    """
    A = np.ones_like(k_path) if A_path is None else np.asarray(A_path)
    w_path = np.zeros_like(k_path)
    for t in range(S):
        w_path[t] = f(k_path[t], model, A[t]) - k_path[t] * η_path[t]
    return w_path
```

Gross one-period return on capital:

$$\bar{R}_{t+1} = \frac{(1+\tau_{ct})}{(1+\tau_{ct+1})}\left[(1-\tau_{kt+1})(f'(k_{t+1})-\delta)+1\right] = \frac{(1+\tau_{ct})}{(1+\tau_{ct+1})}R_{t,t+1} \tag{104.21}$$

```
def compute_R_bar(τ_ct, τ_ctp1, τ_ktp1, k_tp1, model):
    """
    Gross one-period return on capital:
    R_bar = [(1 + τ_c_t) / (1 + τ_c_{t+1})]
        * { [1 - τ_k_{t+1}] * [f'(k_{t+1}) - δ] + 1 }
    """
    return ((1 + τ_ct) / (1 + τ_ctp1)) * (
        (1 - τ_ktp1) * (f_prime(k_tp1, model) - model.δ) + 1)
```

```
def compute_R_bar_path(shocks, k_path, model, S=100):
    """
    Compute R_bar path over time.
    """
    R_bar_path = np.zeros(S + 1)
    for t in range(S):
        R_bar_path[t] = compute_R_bar(
            shocks['τ_c'][t], shocks['τ_c'][t + 1], shocks['τ_k'][t + 1],
            k_path[t + 1], model)
    R_bar_path[S] = R_bar_path[S - 1]
    return R_bar_path
```

One-period discount factor:

$$R_{t,t+1}^{-1} = \frac{q_{t+1}}{q_t} = m_{t,t+1} = \beta\frac{u'(c_{t+1})}{u'(c_t)}\frac{(1+\tau_{ct})}{(1+\tau_{ct+1})} \tag{104.22}$$

Net one-period rate of interest:

$$r_{t,t+1} \equiv R_{t,t+1} - 1 = (1-\tau_{k,t+1})(f'(k_{t+1})-\delta) \tag{104.23}$$

By (104.22) and $r_{t,t+1} = -\ln(\frac{q_{t+1}}{q_t})$, we have

$$R_{t,t+s} = e^{s\cdot r_{t,t+s}}.$$

Then by (104.23), we have

$$\frac{q_{t+s}}{q_t} = e^{-s\cdot r_{t,t+s}}.$$

Rearranging the above equation, we have

$$r_{t,t+s} = -\frac{1}{s} \ln \left(\frac{q_{t+s}}{q_t} \right).$$

```
def compute_rts_path(q_path, S, t):
    """
    Compute r path:
    r_t,t+s = - (1/s) * ln(q_{t+s} / q_t)
    """
    s = np.arange(1, S + 1)
    q_path = np.array([float(q) for q in q_path])

    with np.errstate(divide='ignore', invalid='ignore'):
        rts_path = - np.log(q_path[t + s] / q_path[t]) / s
    return rts_path
```

104.8 Some functional forms

We assume that the representative household' period utility has the following CRRA (constant relative risk aversion) form

$$u(c) = \frac{c^{1-\gamma}}{1-\gamma}$$

```
def u_prime(c, model, A_t=1):
    """
    Marginal utility: u'(c) = c^{-γ}
    with optional technology adjustment: u'(cA) = (cA)^{-γ}
    """
    return (c * A_t) ** (-model.γ)
```

By substituting (104.21) into (104.17), we obtain

$$c_{t+1} = c_t \left[\beta \frac{(1+\tau_{ct})}{(1+\tau_{ct+1})} \left[(1-\tau_{k,t+1})(f'(k_{t+1}) - \delta) + 1 \right] \right]^{\frac{1}{\gamma}} = c_t \left[\beta \overline{R}_{t+1} \right]^{\frac{1}{\gamma}} \tag{104.24}$$

```
def next_c(c_t, R_bar, model, μ_t=1):
    """
    Consumption next period: c_{t+1} = c_t * (β * R̄)^{1/γ}
    with optional growth adjustment: c_{t+1} = c_t * (β * R_bar)^{1/γ} * μ_{t+1}^{-1}
    """
    return c_t * (model.β * R_bar) ** (1 / model.γ) / μ_t
```

For the production function we assume a Cobb-Douglas form:

$$F(k, 1) = Ak^{\alpha}$$

```
def f(k, model, A=1):
    """
    Production function: f(k) = A * k^{α}
    """
    return A * k ** model.α

def f_prime(k, model, A=1):
```

(continues on next page)

(continued from previous page)

```
    """
    Marginal product of capital: f'(k) = α * A * k^{α - 1}
    """
    return model.α * A * k ** (model.α - 1)
```

104.9 Computation

We describe two ways to compute an equilibrium:

- a shooting algorithm
- a residual-minimization method that focuses on imposing Euler equation (104.17) and the feasibility condition (104.15).

104.9.1 Shooting Algorithm

This algorithm deploys the following steps.

1. Solve the equation (104.19) for the terminal steady-state capital $\bar{k}$ that corresponds to the permanent policy vector $\bar{z}$.
2. Select a large time index $S \gg T$, guess an initial consumption rate c_0, and use the equation (104.15) to solve for k_1.
3. Use the equation (104.24) to determine c_{t+1}. Then, apply the equation (104.15) to compute k_{t+2}.
4. Iterate step 3 to compute candidate values $\hat{k}_t$ for $t = 1, \ldots, S$.
5. Compute the difference $\hat{k}_S - \bar{k}$. If $\left|\hat{k}_S - \bar{k}\right| > \epsilon$ for some small ϵ, adjust c_0 and repeat steps 2-5.
6. Adjust c_0 iteratively using the bisection method to find a value that ensures $\left|\hat{k}_S - \bar{k}\right| < \epsilon$.

The following code implements these steps.

```
# Steady-state calculation
def steady_states(model, g_ss, τ_k_ss=0.0, μ_ss=None):
    """
    Calculate steady state values for capital and
    consumption with optional A_path for growth models.
    """

    β, δ, α, γ = model.β, model.δ, model.α, model.γ

    A = model.A or 1.0

    # growth-adjustment in the numerator: μ^γ or 1
    μ_eff = μ_ss**γ if μ_ss is not None else 1.0

    num = δ + (μ_eff/β - 1) / (1 - τ_k_ss)
    k_ss = (num / (α * A)) ** (1 / (α - 1))

    c_ss = (
        A * k_ss**α - δ * k_ss - g_ss
        if μ_ss is None
        else k_ss**α + (1 - δ - μ_ss) * k_ss - g_ss
```

(continues on next page)

(continued from previous page)

```
    )

    return k_ss, c_ss

def shooting_algorithm(
    c0, k0, shocks, S, model, A_path=None):
    """
    Shooting algorithm for given initial c0 and k0
    with optional A_path for growth models.
    """
    # unpack & mpf-ify shocks, fill μ with ones if missing
    g = np.array(list(map(mpf, shocks['g'])), dtype=object)
    τ_c = np.array(list(map(mpf, shocks['τ_c'])), dtype=object)
    τ_k = np.array(list(map(mpf, shocks['τ_k'])), dtype=object)
    μ = (np.array(list(map(mpf, shocks['μ'])), dtype=object)
             if 'μ' in shocks else np.ones_like(g))
    A = np.ones_like(g) if A_path is None else A_path

    k_path = np.empty(S+1, dtype=object)
    c_path = np.empty(S+1, dtype=object)
    k_path[0], c_path[0] = mpf(k0), mpf(c0)

    for t in range(S):
        k_t, c_t = k_path[t], c_path[t]
        k_tp1 = next_k(k_t, g[t], c_t, model, μ[t+1])
        if k_tp1 < 0:
            return None, None
        k_path[t+1] = k_tp1

        R_bar = compute_R_bar(
            τ_c[t], τ_c[t+1], τ_k[t+1], k_tp1, model
        )
        c_tp1 = next_c(c_t, R_bar, model, μ[t+1])
        if c_tp1 < 0:
            return None, None
        c_path[t+1] = c_tp1

    return k_path, c_path


def bisection_c0(
    c0_guess, k0, shocks, S, model, tol=mpf('1e-6'),
    max_iter=1000, verbose=False, A_path=None):
    """
    Bisection method to find initial c0
    """
    # steady-state uses last shocks (μ=1 if missing)
    g_last    = mpf(shocks['g'][-1])
    τ_k_last  = mpf(shocks['τ_k'][-1])
    μ_last    = mpf(shocks['μ'][-1]) if 'μ' in shocks else mpf('1')
    k_ss_fin, _ = steady_states(model, g_last, τ_k_last, μ_last)

    c0_lo, c0_hi = mpf('0'), f(k_ss_fin, model)
    c0 = mpf(c0_guess)

    for i in range(1, max_iter+1):
        k_path, _ = shooting_algorithm(c0, k0, shocks, S, model, A_path)
```

(continues on next page)

(continued from previous page)

```
        if k_path is None:
            if verbose:
                print(f"[{i}] shoot failed at c0={c0}")
            c0_hi = c0
        else:
            err = k_path[-1] - k_ss_fin
            if verbose and i % 100 == 0:
                print(f"[{i}] c0={c0}, err={err}")
            if abs(err) < tol:
                if verbose:
                    print(f"Converged after {i} iter")
                return c0
            # update bounds in one line
            c0_lo, c0_hi = (c0, c0_hi) if err > 0 else (c0_lo, c0)
        c0 = (c0_lo + c0_hi) / mpf('2')

    warn(f"bisection did not converge after {max_iter} iters; returning c0={c0}")
    return c0


def run_shooting(
    shocks, S, model, A_path=None,
    c0_finder=bisection_c0, shooter=shooting_algorithm):
    """
    Compute initial SS, find c0, and return [k,c] paths
    with optional A_path for growth models.
    """
    # initial SS at t=0 (μ=1 if missing)
    g0    = mpf(shocks['g'][0])
    τ_k0  = mpf(shocks['τ_k'][0])
    μ0    = mpf(shocks['μ'][0]) if 'μ' in shocks else mpf('1')
    k0, c0 = steady_states(model, g0, τ_k0, μ0)

    optimal_c0 = c0_finder(c0, k0, shocks, S, model, A_path=A_path)
    print(f"Model: {model}\nOptimal initial consumption c0 = {mpf(optimal_c0)}")

    k_path, c_path = shooter(optimal_c0, k0, shocks, S, model, A_path)
    return np.column_stack([k_path, c_path])
```

104.9.2 Experiments

Let's run some experiments.

1. A foreseen once-and-for-all increase in g from 0.2 to 0.4 occurring in period 10,
2. A foreseen once-and-for-all increase in τ_c from 0.0 to 0.2 occurring in period 10,
3. A foreseen once-and-for-all increase in τ_k from 0.0 to 0.2 occurring in period 10, and
4. A foreseen one-time increase in g from 0.2 to 0.4 in period 10, after which g reverts to 0.2 permanently.

To start, we prepare sequences that we'll used to initialize our iterative algorithm.

We will start from an initial steady state and apply shocks at an the indicated time.

```
def plot_results(
    solution, k_ss, c_ss, shocks, shock_param, axes, model,
```

(continues on next page)

(continued from previous page)

```
    A_path=None, label='', linestyle='-', T=40):
    """
    Plot simulation results (k, c, R, η, and a policy shock)
    with optional A_path for growth models.
    """
    k_path = solution[:, 0]
    c_path = solution[:, 1]
    T = min(T, k_path.size)

    # handle growth parameters
    μ0 = shocks['μ'][0] if 'μ' in shocks else 1.0
    A0 = A_path[0] if A_path is not None else (model.A or 1.0)

    # steady-state lines
    R_bar_ss = (1 / model.β) * (μ0**model.γ)
    η_ss     = model.α * A0 * k_ss**(model.α - 1)

    # plot k
    axes[0].plot(k_path[:T], linestyle=linestyle, label=label)
    axes[0].axhline(k_ss, linestyle='--', color='black')
    axes[0].set_title('k')

    # plot c
    axes[1].plot(c_path[:T], linestyle=linestyle, label=label)
    axes[1].axhline(c_ss, linestyle='--', color='black')
    axes[1].set_title('c')

    # plot R bar
    S_full    = k_path.size - 1
    R_bar_path = compute_R_bar_path(shocks, k_path, model, S_full)
    axes[2].plot(R_bar_path[:T], linestyle=linestyle, label=label)
    axes[2].axhline(R_bar_ss, linestyle='--', color='black')
    axes[2].set_title(r'$\bar{R}$')

    # plot η
    η_path = compute_η_path(k_path, model, S_full)
    axes[3].plot(η_path[:T], linestyle=linestyle, label=label)
    axes[3].axhline(η_ss, linestyle='--', color='black')
    axes[3].set_title(r'$\eta$')

    # plot shock
    shock_series = np.array(shocks[shock_param], dtype=object)
    axes[4].plot(shock_series[:T], linestyle=linestyle, label=label)
    axes[4].axhline(shock_series[0], linestyle='--', color='black')
    axes[4].set_title(rf'${shock_param}$')

    if label:
        for ax in axes[:5]:
            ax.legend()
```

Experiment 1: Foreseen once-and-for-all increase in g from 0.2 to 0.4 in period 10

The figure below shows consequences of a foreseen permanent increase in g at $t = T = 10$ that is financed by an increase in lump-sum taxes

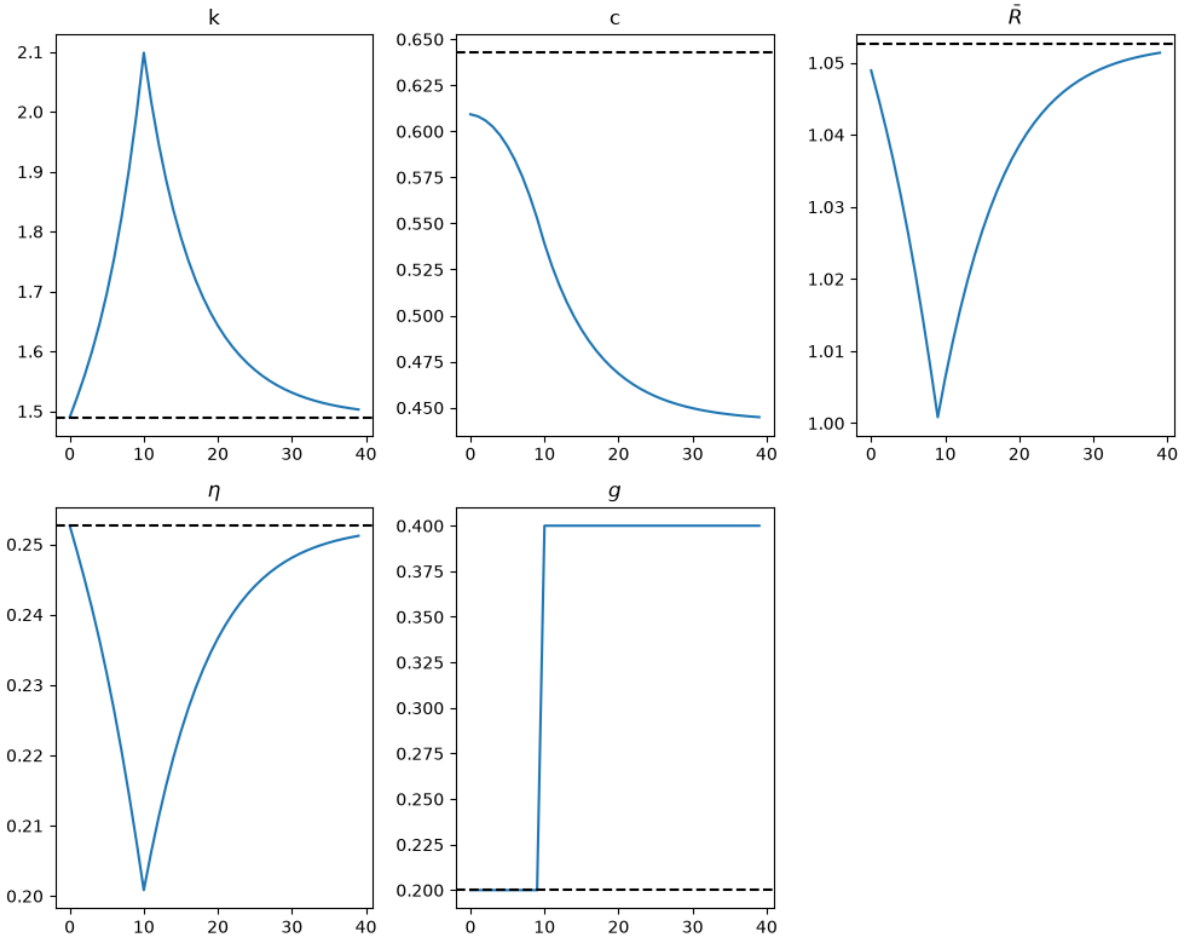
```
# Define shocks as a dictionary
shocks = {
```

(continues on next page)

(continued from previous page)

```
    'g': np.concatenate(
        (np.repeat(0.2, 10), np.repeat(0.4, S - 9))
    ),
    'τ_c': np.repeat(0.0, S + 1),
    'τ_k': np.repeat(0.0, S + 1)
}

k_ss_initial, c_ss_initial = steady_states(model,
                                           shocks['g'][0],
                                           shocks['τ_k'][0])

print(f"Steady-state capital: {k_ss_initial:.4f}")
print(f"Steady-state consumption: {c_ss_initial:.4f}")

solution = run_shooting(shocks, S, model)

fig, axes = plt.subplots(2, 3, figsize=(10, 8))
axes = axes.flatten()

plot_results(solution, k_ss_initial,
             c_ss_initial, shocks, 'g', axes, model, T=40)

for ax in axes[5:]:
    fig.delaxes(ax)

plt.tight_layout()
plt.show()
```

```
Steady-state capital: 1.4900
Steady-state consumption: 0.6426
Model: Model(β=0.95, γ=2.0, δ=0.2, α=0.33, A=1.0)
Optimal initial consumption c0 = 0.6092419528879239645312185699727132533517
```



The above figures indicate that an equilibrium **consumption smoothing** mechanism is at work, driven by the representative consumer's preference for smooth consumption paths coming from the curvature of its one-period utility function.

- The steady-state value of the capital stock remains unaffected:
  - This follows from the fact that g disappears from the steady state version of the Euler equation ((104.19)).
- Consumption begins to decline gradually before time T due to increased government consumption:
  - Households reduce consumption to offset government spending, which is financed through increased lump-sum taxes.
  - The competitive economy signals households to consume less through an increase in the stream of lump-sum taxes.
  - Households, caring about the present value rather than the timing of taxes, experience an adverse wealth effect on consumption, leading to an immediate response.
- Capital gradually accumulates between time 0 and T due to increased savings and reduces gradually after time T:
  - This temporal variation in capital stock smooths consumption over time, driven by the representative consumer's consumption-smoothing motive.

Let's collect the procedures used above into a function that runs the solver and draws plots for a given experiment

The following figure compares responses to a foreseen increase in g at $t = 10$ for two economies:

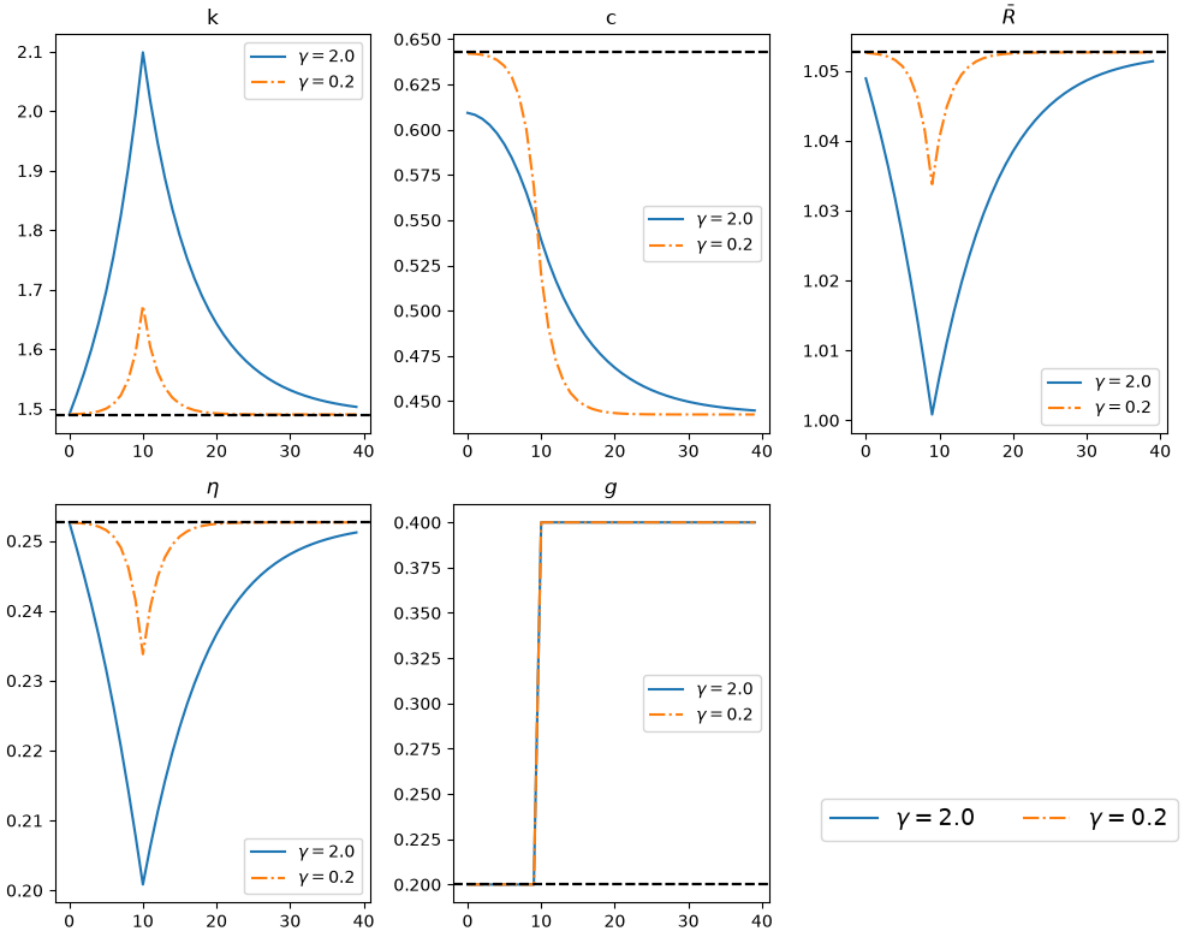
- our original economy with $\gamma = 2$, shown in the solid line, and

- an otherwise identical economy with $\gamma = 0.2$.

This comparison interest us because the utility curvature parameter γ governs the household's willingness to substitute consumption over time, and thus it preferences about smoothness of consumption paths over time.

```
# Solve the model using shooting
solution = run_shooting(shocks, S, model)

# Compute the initial steady states
k_ss_initial, c_ss_initial = steady_states(model,
                                           shocks['g'][0],
                                           shocks['τ_k'][0])

# Plot the solution for γ=2
fig, axes = plt.subplots(2, 3, figsize=(10, 8))
axes = axes.flatten()

label = fr"$\gamma = {model.γ}$"
plot_results(solution, k_ss_initial, c_ss_initial,
             shocks, 'g', axes, model, label=label,
             T=40)

# Solve and plot the result for γ=0.2
model_γ2 = create_model(γ=0.2)
solution = run_shooting(shocks, S, model_γ2)

plot_results(solution, k_ss_initial, c_ss_initial,
             shocks, 'g', axes, model_γ2,
             label=fr"$\gamma = {model_γ2.γ}$",
             linestyle='-.', T=40)

handles, labels = axes[0].get_legend_handles_labels()
fig.legend(handles, labels, loc='lower right',
           ncol=3, fontsize=14, bbox_to_anchor=(1, 0.1))

for ax in axes[5:]:
    fig.delaxes(ax)

plt.tight_layout()
plt.show()
```

```
Model: Model(β=0.95, γ=2.0, δ=0.2, α=0.33, A=1.0)
Optimal initial consumption c0 = 0.60924195288792396453121856997271325335517
Model: Model(β=0.95, γ=0.2, δ=0.2, α=0.33, A=1.0)
Optimal initial consumption c0 = 0.64203304129879029264147687246076236817745
```



The outcomes indicate that lowering γ affects both the consumption and capital stock paths because - it increases the representative consumer's willingness to substitute consumption across time:

- Consumption path:
  - When $\gamma = 0.2$, consumption becomes less smooth compared to $\gamma = 2$.
  - For $\gamma = 0.2$, consumption mirrors the government expenditure path more closely, staying higher until $t = 10$.
- Capital stock path:
  - With $\gamma = 0.2$, there are smaller build-ups and drawdowns of capital stock.
  - There are also smaller fluctuations in $\bar{R}$ and η.

Let's write another function that runs the solver and draws plots for these two experiments

Now we plot other equilibrium quantities:

```
def plot_prices(solution, c_ss, shock_param, axes,
                model, label='', linestyle='-', T=40):
    """
    Compares and plots prices
    """
    α, β, δ, γ, A = model.α, model.β, model.δ, model.γ, model.A

    k_path = solution[:, 0]
    c_path = solution[:, 1]
```

(continues on next page)

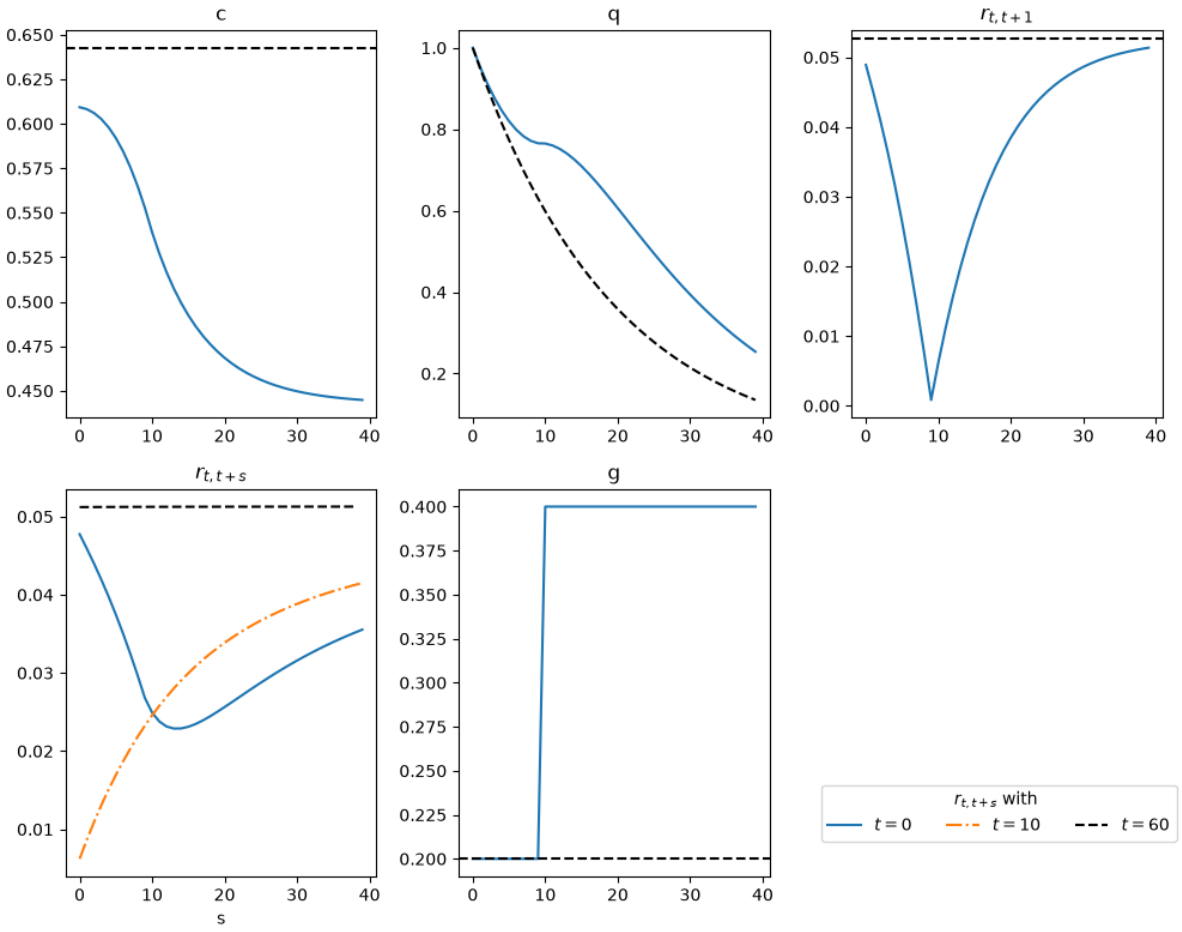
(continued from previous page)

```
    # Plot for c
    axes[0].plot(c_path[:T], linestyle=linestyle, label=label)
    axes[0].axhline(c_ss, linestyle='--', color='black')
    axes[0].set_title('c')

    # Plot for q
    q_path = compute_q_path(c_path, model, S=S)
    axes[1].plot(q_path[:T], linestyle=linestyle, label=label)
    axes[1].plot(β**np.arange(T), linestyle='--', color='black')
    axes[1].set_title('q')

    # Plot for r_{t,t+1}
    R_bar_path = compute_R_bar_path(shocks, k_path, model, S)
    axes[2].plot(R_bar_path[:T] - 1, linestyle=linestyle, label=label)
    axes[2].axhline(1 / β - 1, linestyle='--', color='black')
    axes[2].set_title('$r_{t,t+1}$')

    # Plot for r_{t,t+s}
    for style, s in zip(['-', '-.', '--'], [0, 10, 60]):
        rts_path = compute_rts_path(q_path, T, s)
        axes[3].plot(rts_path, linestyle=style,
                     color='black' if style == '--' else None,
                     label=f'$t={s}$')
        axes[3].set_xlabel('s')
        axes[3].set_title('$r_{t,t+s}$')

    # Plot for g
    axes[4].plot(shocks[shock_param][:T], linestyle=linestyle, label=label)
    axes[4].axhline(shocks[shock_param][0], linestyle='--', color='black')
    axes[4].set_title(shock_param)
```

For $\gamma = 2$ again, the next figure describes the response of q_t and the term structure of interest rates to a foreseen increase in g_t at $t = 10$

```
solution = run_shooting(shocks, S, model)

fig, axes = plt.subplots(2, 3, figsize=(10, 8))
axes = axes.flatten()

plot_prices(solution, c_ss_initial, 'g', axes, model, T=40)

for ax in axes[5:]:
    fig.delaxes(ax)

handles, labels = axes[3].get_legend_handles_labels()
fig.legend(handles, labels, title=r"$r_{t,t+s}$ with ", loc='lower right',
           ncol=3, fontsize=10, bbox_to_anchor=(1, 0.1))
plt.tight_layout()
plt.show()
```

```
Model: Model(β=0.95, γ=2.0, δ=0.2, α=0.33, A=1.0)
Optimal initial consumption c0 = 0.609241952887923964531218569972713253351 7
```



The second panel on the top compares q_t for the initial steady state with q_t after the increase in g is foreseen at $t = 0$, while the third panel compares the implied short rate r_t.

The fourth panel shows the term structure of interest rates at $t = 0$, $t = 10$, and $t = 60$.

Notice that, by $t = 60$, the system has converged to the new steady state, and the term structure of interest rates becomes flat.

At $t = 10$, the term structure of interest rates is upward sloping.

This upward slope reflects the expected increase in the rate of growth of consumption over time, as shown in the consumption panel.

At $t = 0$, the term structure of interest rates exhibits a "U-shaped" pattern:

- It declines until maturity at $s = 10$.
- After $s = 10$, it increases for longer maturities.

This pattern aligns with the pattern of consumption growth in the first two figures, which declines at an increasing rate until $t = 10$ and then declines at a decreasing rate afterward.

Experiment 2: Foreseen once-and-for-all increase in τ_c from 0.0 to 0.2 in period 10

With an inelastic labor supply, the Euler equation (104.16) and the other equilibrium conditions show that

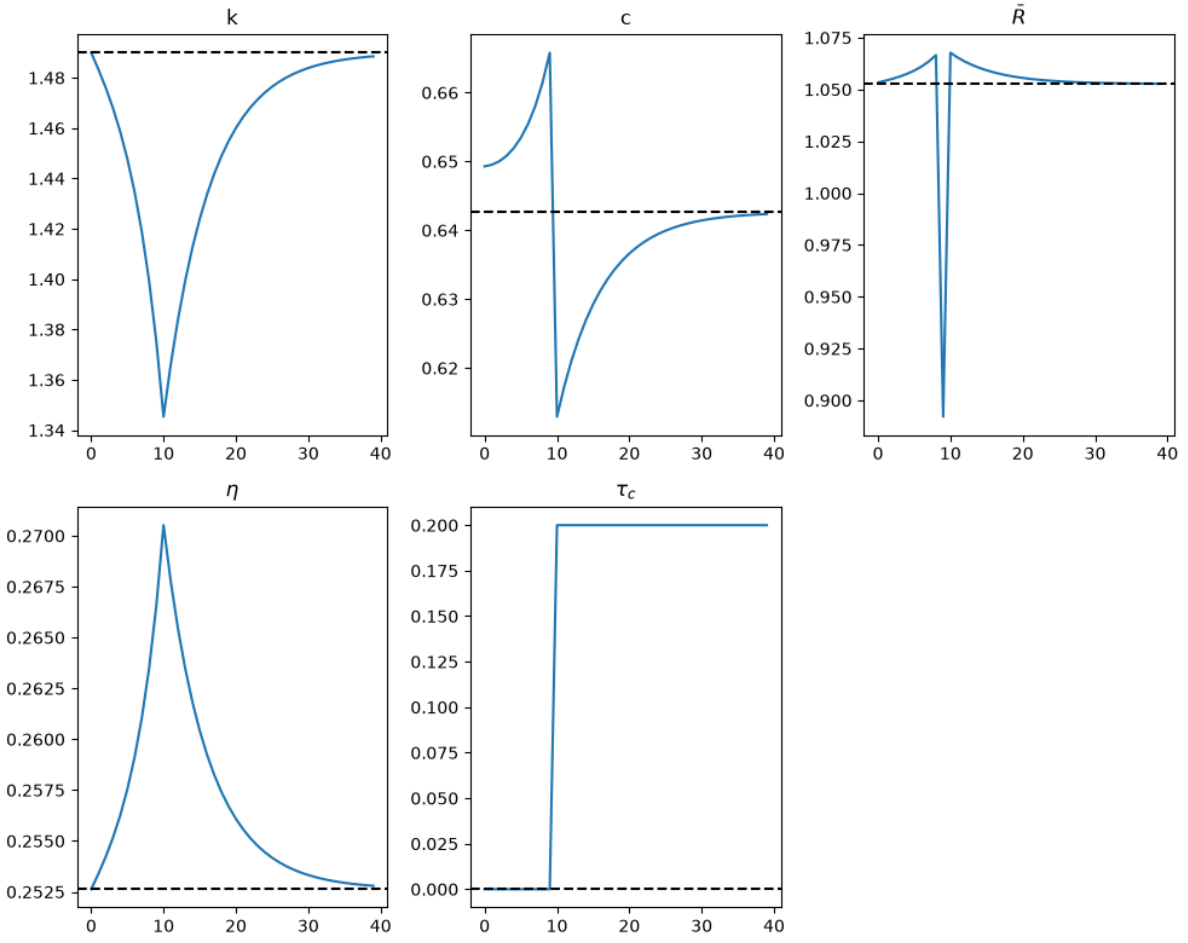
- constant consumption taxes do not distort decisions, but
- anticipated changes in them do.

Indeed, (104.16) or (104.17) indicates that a foreseen in- crease in τ_{ct} (i.e., a decrease in $(1+\tau_{ct})\ (1+\tau_{ct+1})$) operates like an increase in τ_{kt}.

The following figure portrays the response to a foreseen increase in the consumption tax τ_c.

```
shocks = {
    'g': np.repeat(0.2, S + 1),
    'τ_c': np.concatenate((np.repeat(0.0, 10), np.repeat(0.2, S - 9))),
    'τ_k': np.repeat(0.0, S + 1)
}

experiment_model(shocks, S, model,
                 solver=run_shooting,
                 plot_func=plot_results,
                 policy_shock='τ_c')
```

```
Steady-state capital: 1.4900
Steady-state consumption: 0.6426
----------------------------------------------------------------
Model: Model(β=0.95, γ=2.0, δ=0.2, α=0.33, A=1.0)
Optimal initial consumption c0 = 0.64927956146815433723018647051957883983 96
```



Evidently all variables in the figures above eventually return to their initial steady-state values.

The anticipated increase in τ_{ct} leads to variations in consumption and capital stock across time:

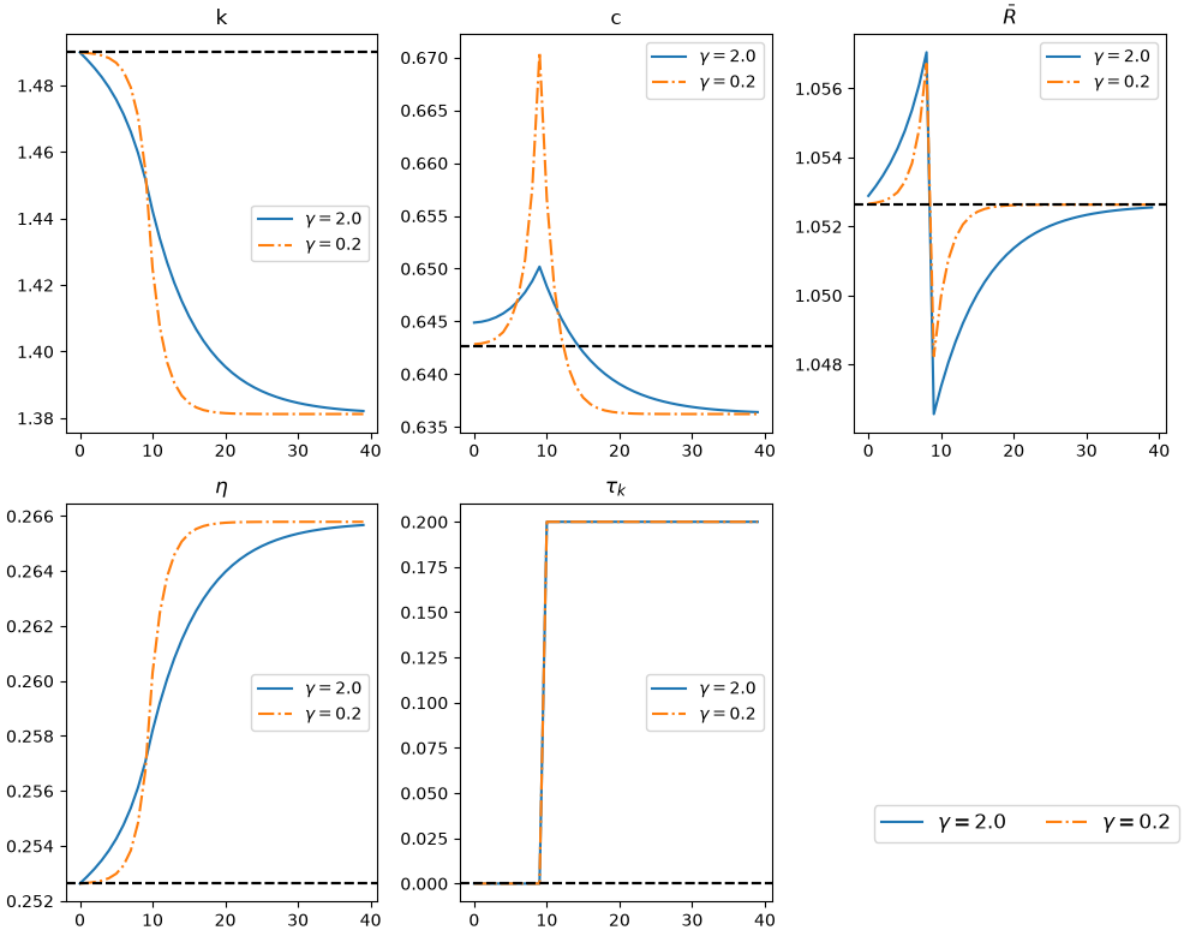
- At $t = 0$:
  - Anticipation of the increase in τ_c causes an *immediate jump in consumption*.
  - This is followed by a *consumption binge* that sends the capital stock downward until $t = T = 10$.
- Between $t = 0$ and $t = T = 10$:
  - The decline in the capital stock raises $\bar{R}$ over time.
  - The equilibrium conditions require the growth rate of consumption to rise until $t = T$.
- At $t = T = 10$:
  - The jump in τ_c depresses $\bar{R}$ below 1, causing a *sharp drop in consumption*.
- After $T = 10$:
  - The effects of anticipated distortion are over, and the economy gradually adjusts to the lower capital stock.
  - Capital must now rise, requiring *austerity* –consumption plummets after $t = T$, indicated by lower levels of consumption.
  - The interest rate gradually declines, and consumption grows at a diminishing rate along the path to the terminal steady-state.

Experiment 3: Foreseen once-and-for-all increase in τ_k from 0.0 to 0.2 in period 10

For the two γ values 2 and 0.2, the next figure shows the response to a foreseen permanent jump in τ_{kt} at $t = T = 10$.

```
shocks = {
    'g': np.repeat(0.2, S + 1),
    'τ_c': np.repeat(0.0, S + 1),
    'τ_k': np.concatenate((np.repeat(0.0, 10), np.repeat(0.2, S - 9)))
}

experiment_two_models(shocks, S, model, model_γ2,
                solver=run_shooting,
                 plot_func=plot_results,
                 policy_shock='τ_k')
```

```
Model 1 (γ=2.0): steady state k=1.4900, c=0.6426
Model 2 (γ=0.2): steady state k=1.4900, c=0.6426
----------------------------------------------------------------
Model: Model(β=0.95, γ=2.0, δ=0.2, α=0.33, A=1.0)
Optimal initial consumption c0 = 0.64488564003186084609963008227070148901 99
Model: Model(β=0.95, γ=0.2, δ=0.2, α=0.33, A=1.0)
Optimal initial consumption c0 = 0.64284077722405067274641526959215137674 15
```



The path of government expenditures remains fixed

- the increase in τ_{kt} is offset by a reduction in the present value of lump-sum taxes to keep the budget balanced.

The figure shows that:

- Anticipation of the increase in τ_{kt} leads to immediate decline in capital stock due to increased current consumption and a growing consumption flow.
- $\bar{R}$ starts rising at $t = 0$ and peaks at $t = 9$, and at $t = 10$, $\bar{R}$ drops sharply due to the tax change.
  - Variations in $\bar{R}$ align with the impact of the tax increase at $t = 10$ on consumption across time.
- Transition dynamics push k_t (capital stock) toward a new, lower steady-state level. In the new steady state:
  - Consumption is lower due to reduced output from the lower capital stock.
  - Smoother consumption paths occur when $\gamma = 2$ than when $\gamma = 0.2$.

So far we have explored consequences of foreseen once-and-for-all changes in government policy. Next we describe some experiments in which there is a foreseen one-time change in a policy variable (a "pulse").

Experiment 4: Foreseen one-time increase in g from 0.2 to 0.4 in period 10, after which g returns to 0.2 forever

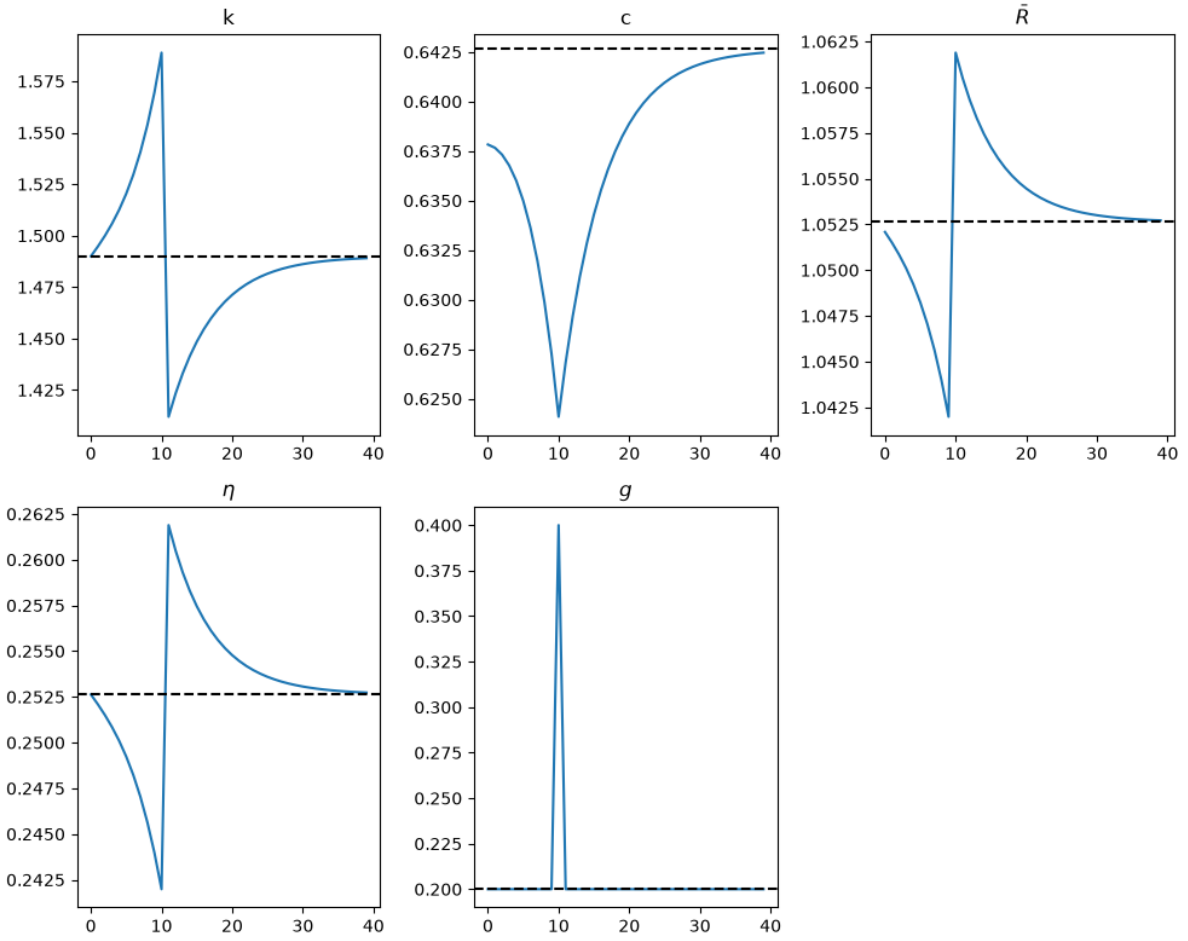
```
g_path = np.repeat(0.2, S + 1)
g_path[10] = 0.4

shocks = {
```

(continues on next page)

(continued from previous page)

```
    'g': g_path,
    'τ_c': np.repeat(0.0, S + 1),
    'τ_k': np.repeat(0.0, S + 1)
}

experiment_model(shocks, S, model,
                 solver=run_shooting,
                 plot_func=plot_results,
                 policy_shock='g')
```

```
Steady-state capital: 1.4900
Steady-state consumption: 0.6426
------------------------------------------------------------
Model: Model(β=0.95, γ=2.0, δ=0.2, α=0.33, A=1.0)
Optimal initial consumption c0 = 0.6378298012463969247674771825320030214755
```



The figure indicates how:

- Consumption:
  - Drops immediately upon announcement of the policy and continues to decline over time in anticipation of the one-time surge in g.
  - After the shock at $t = 10$, consumption begins to recover, rising at a diminishing rate toward its steady-state value.

- Capital and $\bar{R}$:
  - Before $t = 10$, capital accumulates as interest rate changes induce households to prepare for the anticipated increase in government spending.
  - At $t = 10$, the capital stock sharply decreases as the government consumes part of it.
  - $\bar{R}$ jumps above its steady-state value due to the capital reduction and then gradually declines toward its steady-state level.

104.9.3 Method 2: Residual Minimization

The second method involves minimizing residuals (i.e., deviations from equalities) of the following equations:

- The Euler equation (104.17):

$$1 = \beta \left(\frac{c_{t+1}}{c_t} \right)^{-\gamma} \frac{(1 + \tau_{ct})}{(1 + \tau_{ct+1})} \left[(1 - \tau_{kt+1})(\alpha A k_{t+1}^{\alpha - 1} - \delta) + 1 \right]$$

- The feasibility condition (104.15):

$$k_{t+1} = A k_t^{\alpha} + (1 - \delta) k_t - g_t - c_t.$$

```
# Euler's equation and feasibility condition
def euler_residual(c_t, c_tp1, τ_c_t, τ_c_tp1, τ_k_tp1, k_tp1, model, μ_tp1=1):
    """
    Computes the residuals for Euler's equation
    with optional growth model parameters μ_tp1.
    """
    R_bar = compute_R_bar(τ_c_t, τ_c_tp1, τ_k_tp1, k_tp1, model)

    c_expected = next_c(c_t, R_bar, model, μ_tp1)

    return c_expected / c_tp1 - 1.0

def feasi_residual(k_t, k_tm1, c_tm1, g_t, model, μ_t=1):
    """
    Computes the residuals for feasibility condition
    with optional growth model parameter μ_t.
    """
    k_t_expected = next_k(k_tm1, g_t, c_tm1, model, μ_t)
    return k_t_expected - k_t
```

The algorithm proceeds follows:

1. Find initial steady state k_0 based on the government plan at $t = 0$.
2. Initialize a sequence of initial guesses $\{\hat{c}_t, \hat{k}_t\}_{t=0}^{S}$.
3. Compute residuals l_a and l_k for $t = 0, \ldots, S$, as well as l_{k_0} for $t = 0$ and l_{k_S} for $t = S$:
   - Compute the Euler equation residual for $t = 0, \ldots, S$ using (104.17):

$$l_{ta} = \beta u'(c_{t+1}) \frac{(1 + \tau_{ct})}{(1 + \tau_{ct+1})} \left[(1 - \tau_{kt+1})(f'(k_{t+1}) - \delta) + 1 \right] - 1$$

   - Compute the feasibility condition residual for $t = 1, \ldots, S - 1$ using (104.15):

$$l_{tk} = k_{t+1} - f(k_t) - (1 - \delta) k_t + g_t + c_t$$

- Compute the residual for the initial condition for k_0 using (104.19) and the initial capital k_0:

$$l_{k_0} = 1 - \beta\left[(1-\tau_{k0})\left(f'(k_0) - \delta\right) + 1\right]$$

- Compute the residual for the terminal condition for $t = S$ using (104.17) under the assumptions $c_t = c_{t+1} = c_S$, $k_t = k_{t+1} = k_S$, $\tau_{ct} = \tau_{ct+1} = \tau_{c_s}$, and $\tau_{kt} = \tau_{kt+1} = \tau_{k_s}$:

$$l_{k_S} = \beta u'(c_S)\frac{(1+\tau_{c_s})}{(1+\tau_{c_s})}\left[(1-\tau_{k_s})(f'(k_S) - \delta) + 1\right] - 1$$

4. Iteratively adjust guesses for $\{\hat{c}_t, \hat{k}_t\}_{t=0}^{S}$ to minimize residuals l_{k_0}, l_{ta}, l_{tk}, and l_{k_S} for $t = 0, \ldots, S$.

```
def compute_residuals(vars_flat, k_init, S, shock_paths, model):
    """
    Compute the residuals for the Euler equation and feasibility condition.
    """
    g, τ_c, τ_k, μ = (shock_paths[key] for key in ('g','τ_c','τ_k','μ'))
    k, c = vars_flat.reshape((S+1, 2)).T
    res = np.empty(2*S+2, dtype=float)

    # boundary condition on initial capital
    res[0] = k[0] - k_init

    # interior Euler and feasibility
    for t in range(S):
        res[2*t + 1] = euler_residual(
            c[t],    c[t+1],
            τ_c[t],  τ_c[t+1],
            τ_k[t+1],k[t+1],
            model, μ[t+1])
        res[2*t + 2] = feasi_residual(
            k[t+1], k[t], c[t],
            g[t],  model,
            μ[t+1])

    # terminal Euler condition at t=S
    res[-1] = euler_residual(
        c[S],   c[S],
        τ_c[S], τ_c[S],
        τ_k[S], k[S],
        model,
        μ[S])

    return res


def run_min(shocks, S, model, A_path=None):
    """
    Solve for the full (k,c) path by root-finding the residuals.
    """
    shocks['μ'] = shocks['μ'] if 'μ' in shocks else np.ones_like(shocks['g'])

    # compute the steady-state to serve as both initial capital and guess
    k_ss, c_ss = steady_states(
        model,
        shocks['g'][0],
```

(continues on next page)

(continued from previous page)

```
        shocks['τ_k'][0],
        shocks['μ'][0]  # =1 if no growth
    )

    # initial guess: flat at the steady-state
    guess = np.column_stack([
        np.full(S+1, k_ss),
        np.full(S+1, c_ss)
    ]).flatten()

    sol = root(
        compute_residuals,
        guess,
        args=(k_ss, S, shocks, model),
        tol=1e-8
    )

    return sol.x.reshape((S+1, 2))
```

We found that method 2 did not encounter numerical stability issues, so using `mp.mpf` is not necessary.

We leave as exercises replicating some of our experiments by using the second method.

Exercise 104.9.1

Replicate the plots of our four experiments using the second method of residual minimization:

1. A foreseen once-and-for-all increase in g from 0.2 to 0.4 occurring in period 10,
2. A foreseen once-and-for-all increase in τ_c from 0.0 to 0.2 occurring in period 10,
3. A foreseen once-and-for-all increase in τ_k from 0.0 to 0.2 occurring in period 10, and
4. A foreseen one-time increase in g from 0.2 to 0.4 in period 10, after which g reverts to 0.2 permanently,

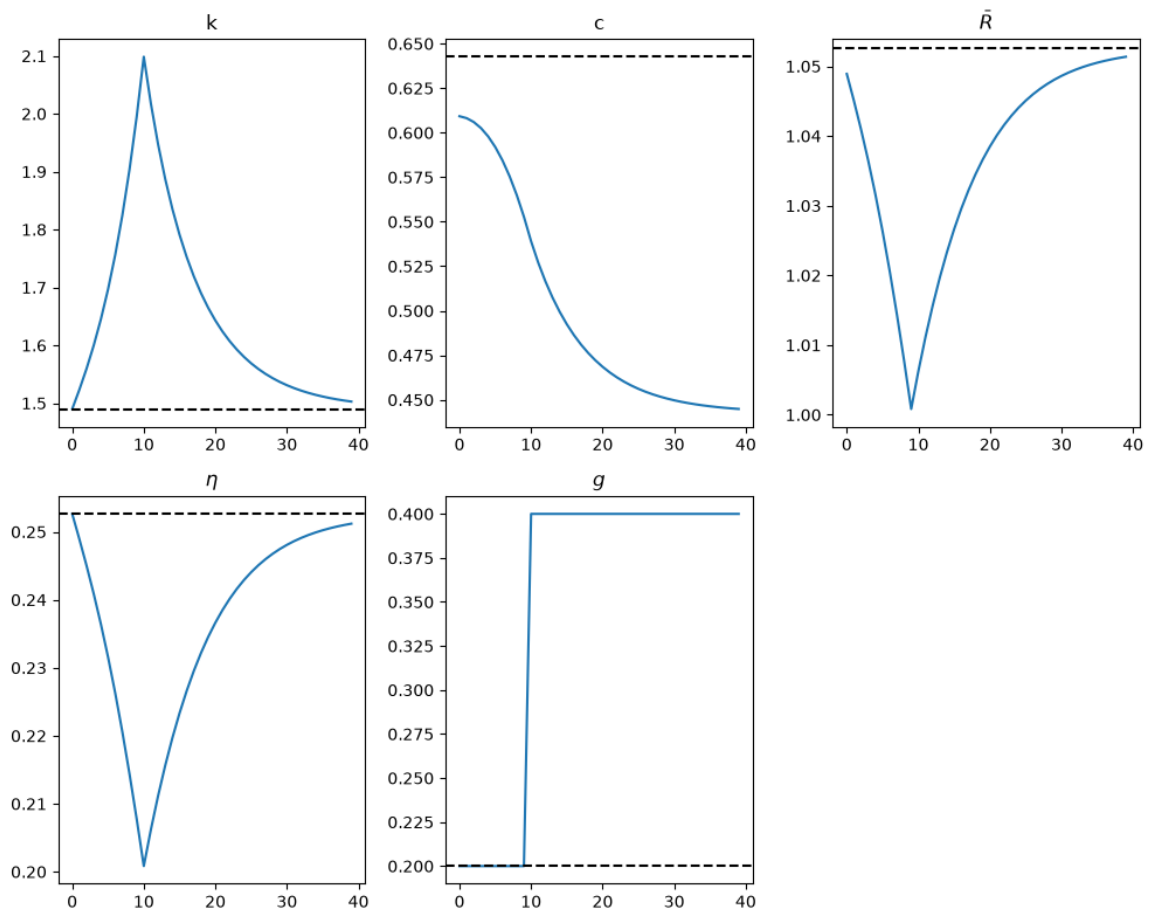
Solution

Here is one solution:

Experiment 1: Foreseen once-and-for-all increase in g from 0.2 to 0.4 in period 10

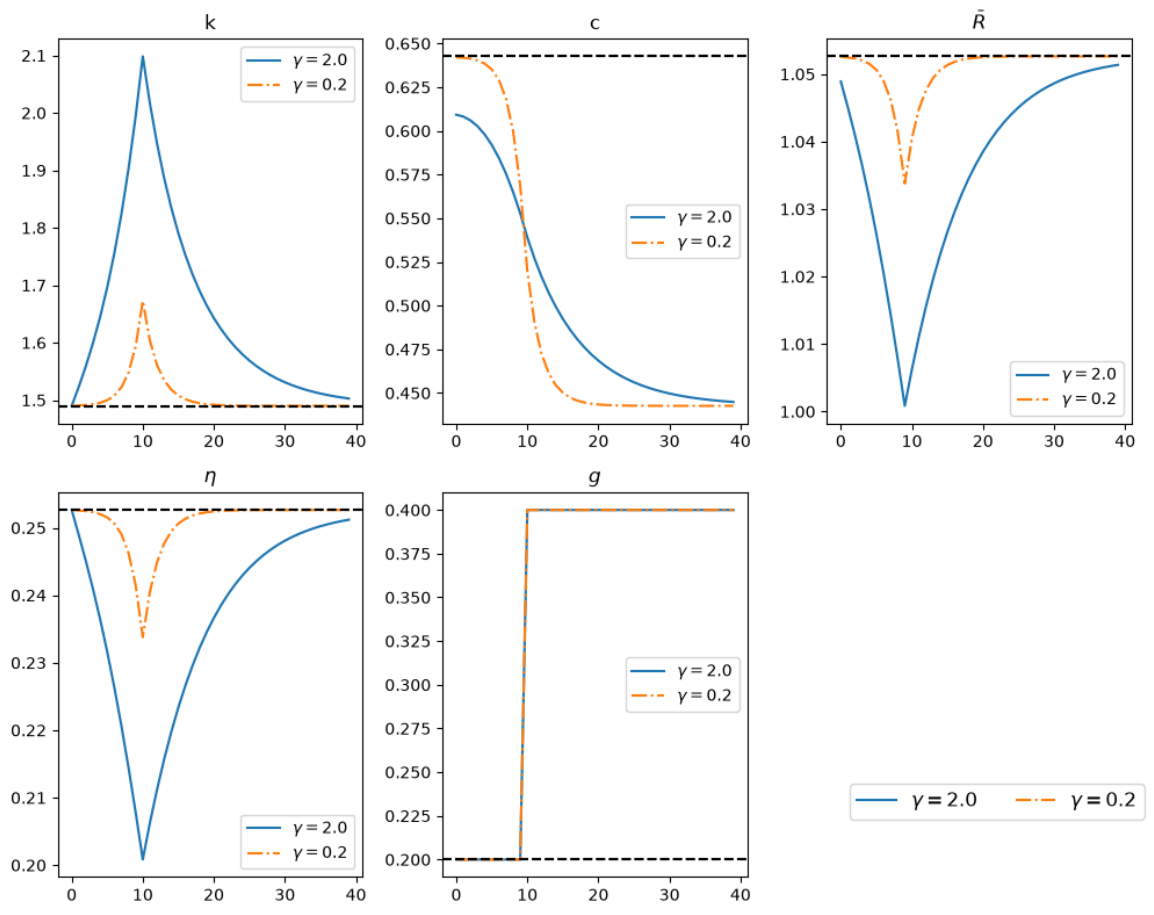
```
shocks = {
    'g': np.concatenate((np.repeat(0.2, 10), np.repeat(0.4, S - 9))),
    'τ_c': np.repeat(0.0, S + 1),
    'τ_k': np.repeat(0.0, S + 1)
}

experiment_model(shocks, S, model, solver=run_min,
                 plot_func=plot_results,
                 policy_shock='g')
```

```
Steady-state capital: 1.4900
Steady-state consumption: 0.6426
----------------------------------------------------------------
```

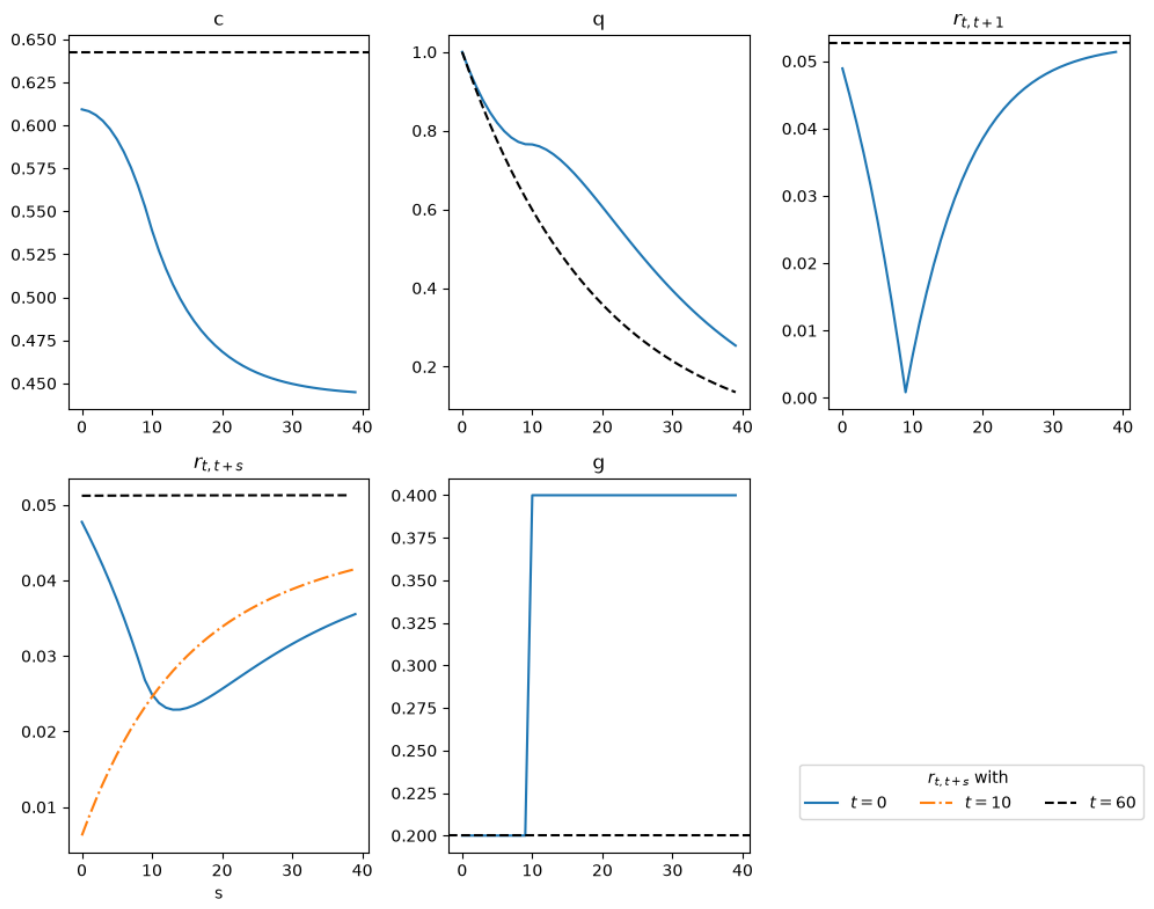



```
experiment_two_models(shocks, S, model, model_y2,
                run_min, plot_results, 'g')
```

```
Model 1 (y=2.0): steady state k=1.4900, c=0.6426
Model 2 (y=0.2): steady state k=1.4900, c=0.6426
----------------------------------------------------------------
```



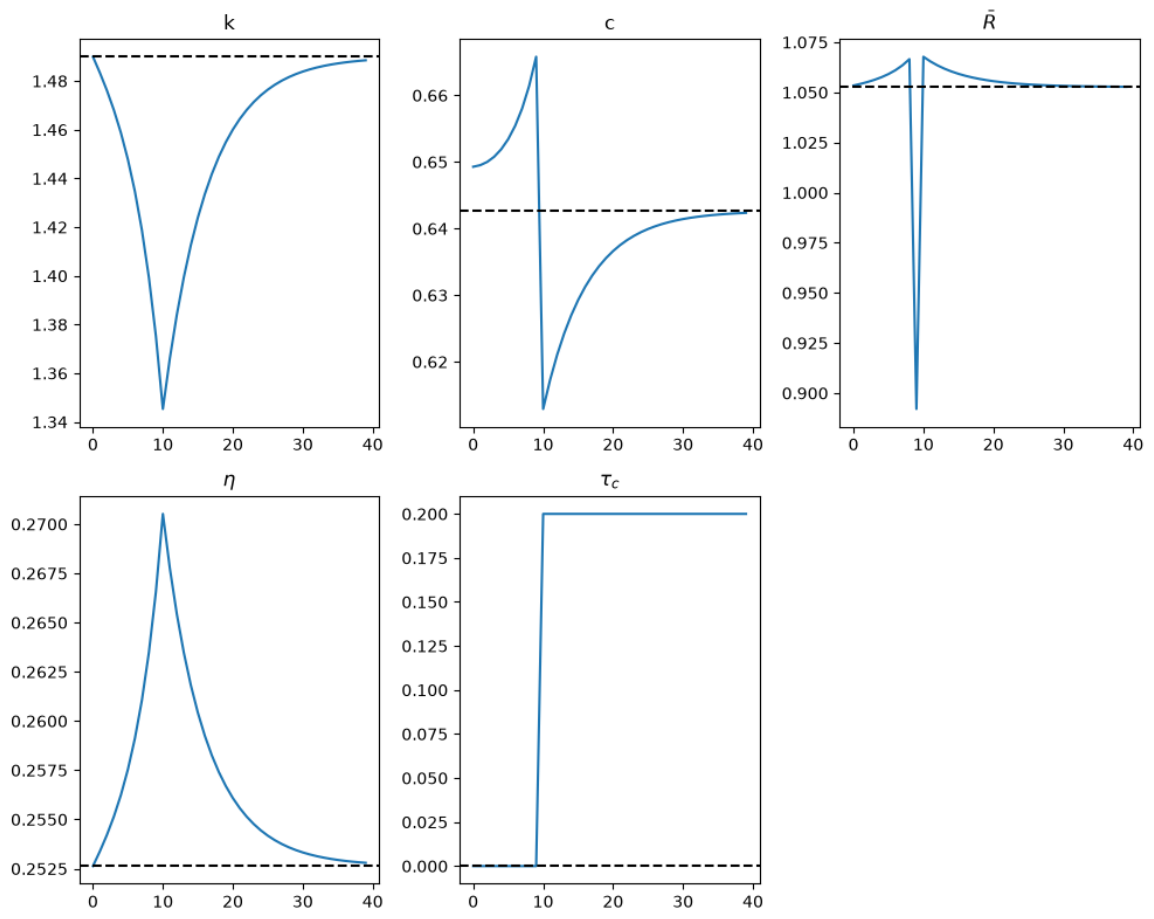

```
solution = run_min(shocks, S, model)

fig, axes = plt.subplots(2, 3, figsize=(10, 8))
axes = axes.flatten()

plot_prices(solution, c_ss_initial, 'g', axes, model, T=40)

for ax in axes[5:]:
    fig.delaxes(ax)

handles, labels = axes[3].get_legend_handles_labels()
fig.legend(handles, labels, title=r"$r_{t,t+s}$ with ", loc='lower right', ncol=3,␣
↪fontsize=10, bbox_to_anchor=(1, 0.1))
plt.tight_layout()
plt.show()
```



Experiment 2: Foreseen once-and-for-all increase in τ_c from 0.0 to 0.2 in period 10.

```
shocks = {
    'g': np.repeat(0.2, S + 1),
    'τ_c': np.concatenate((np.repeat(0.0, 10), np.repeat(0.2, S - 9))),
    'τ_k': np.repeat(0.0, S + 1)
}

experiment_model(shocks, S, model, solver=run_min,
                 plot_func=plot_results,
                 policy_shock='τ_c')
```

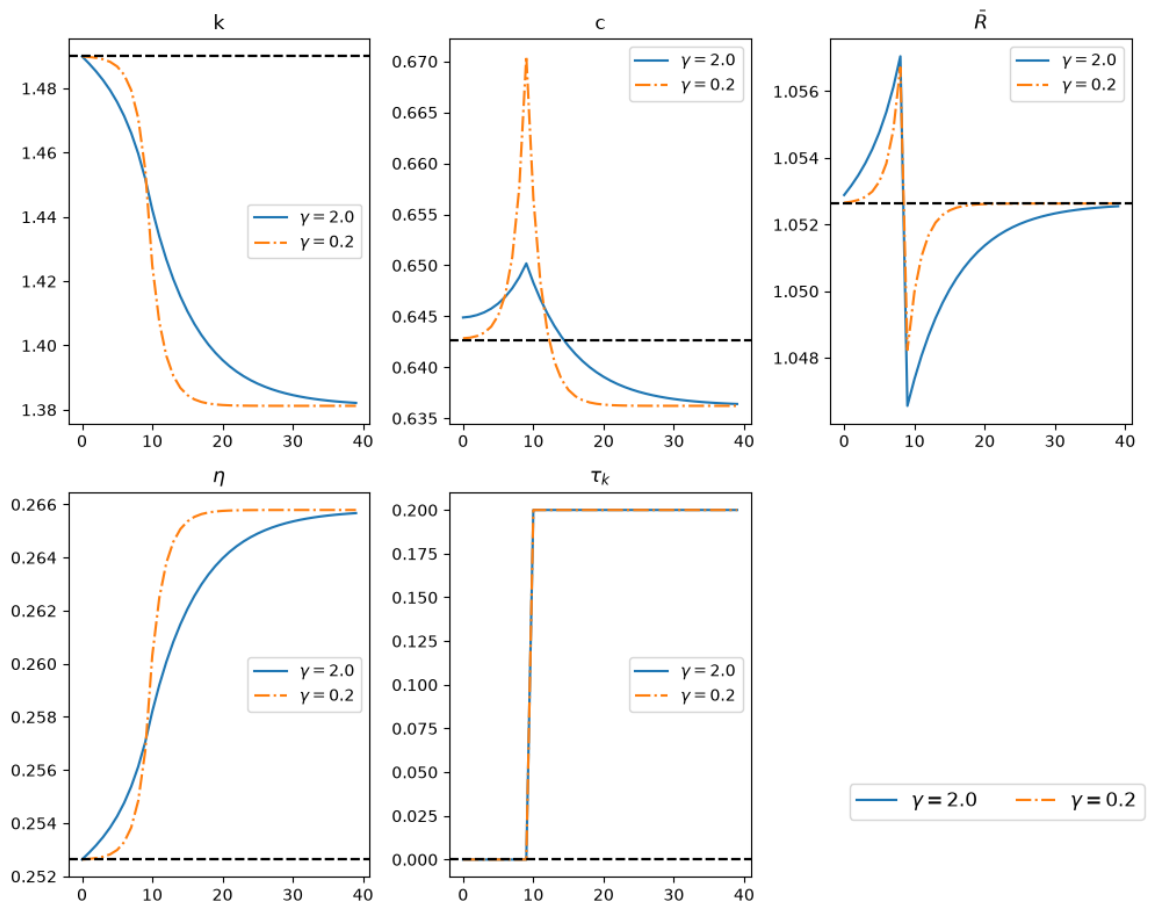
```
Steady-state capital: 1.4900
Steady-state consumption: 0.6426
----------------------------------------------------------------------
```



Experiment 3: Foreseen once-and-for-all increase in τ_k from 0.0 to 0.2 in period 10.

```
shocks = {
    'g': np.repeat(0.2, S + 1),
    'τ_c': np.repeat(0.0, S + 1),
    'τ_k': np.concatenate((np.repeat(0.0, 10), np.repeat(0.2, S - 9)))
}

experiment_two_models(shocks, S, model, model_γ2,
                solver=run_min,
                 plot_func=plot_results,
                 policy_shock='τ_k')
```

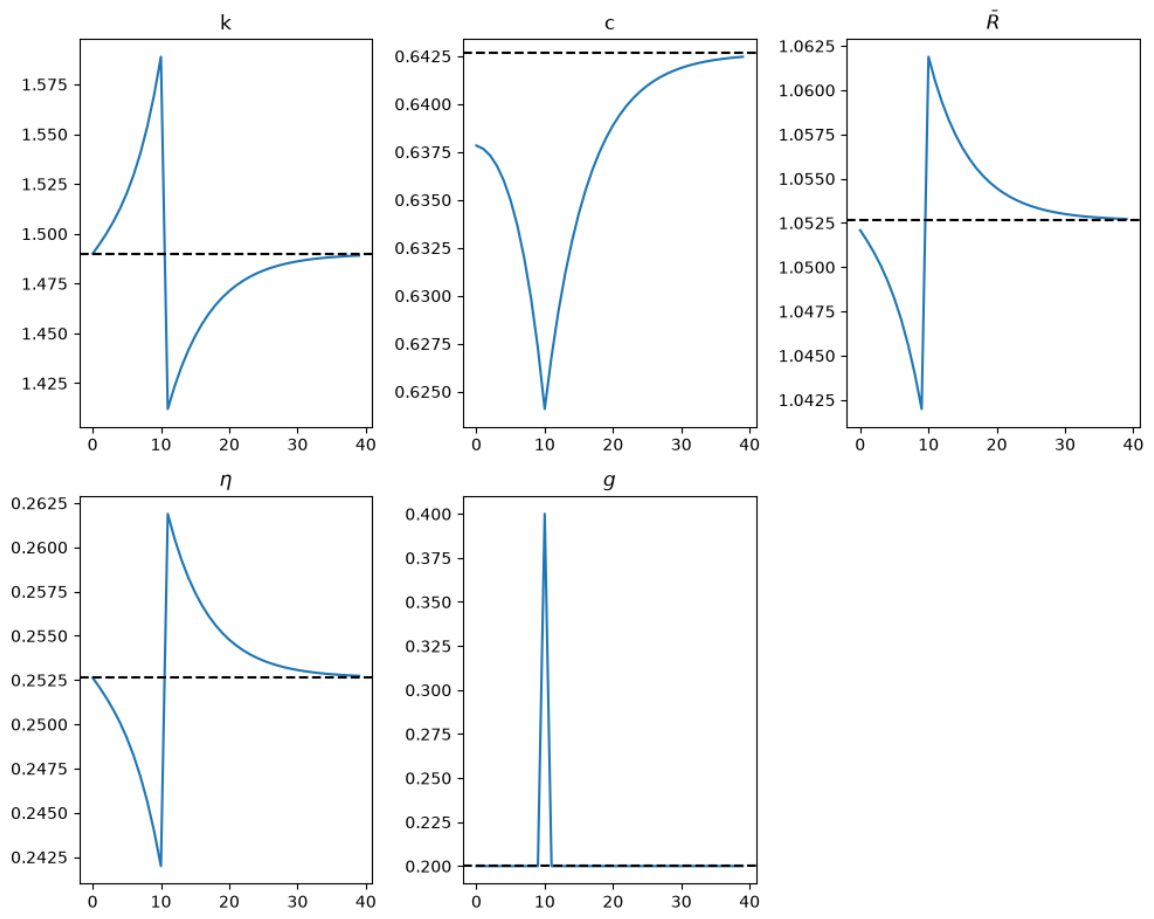
```
Model 1 (γ=2.0): steady state k=1.4900, c=0.6426
Model 2 (γ=0.2): steady state k=1.4900, c=0.6426
----------------------------------------------------------------
```



Experiment 4: Foreseen one-time increase in g from 0.2 to 0.4 in period 10, after which g returns to 0.2 forever

```
g_path = np.repeat(0.2, S + 1)
g_path[10] = 0.4

shocks = {
    'g': g_path,
    'τ_c': np.repeat(0.0, S + 1),
    'τ_k': np.repeat(0.0, S + 1)
}

experiment_model(shocks, S, model, solver=run_min,
                 plot_func=plot_results,
                 policy_shock='g')
```

```
Steady-state capital: 1.4900
Steady-state consumption: 0.6426
------------------------------------------------------------------
```



Exercise 104.9.2

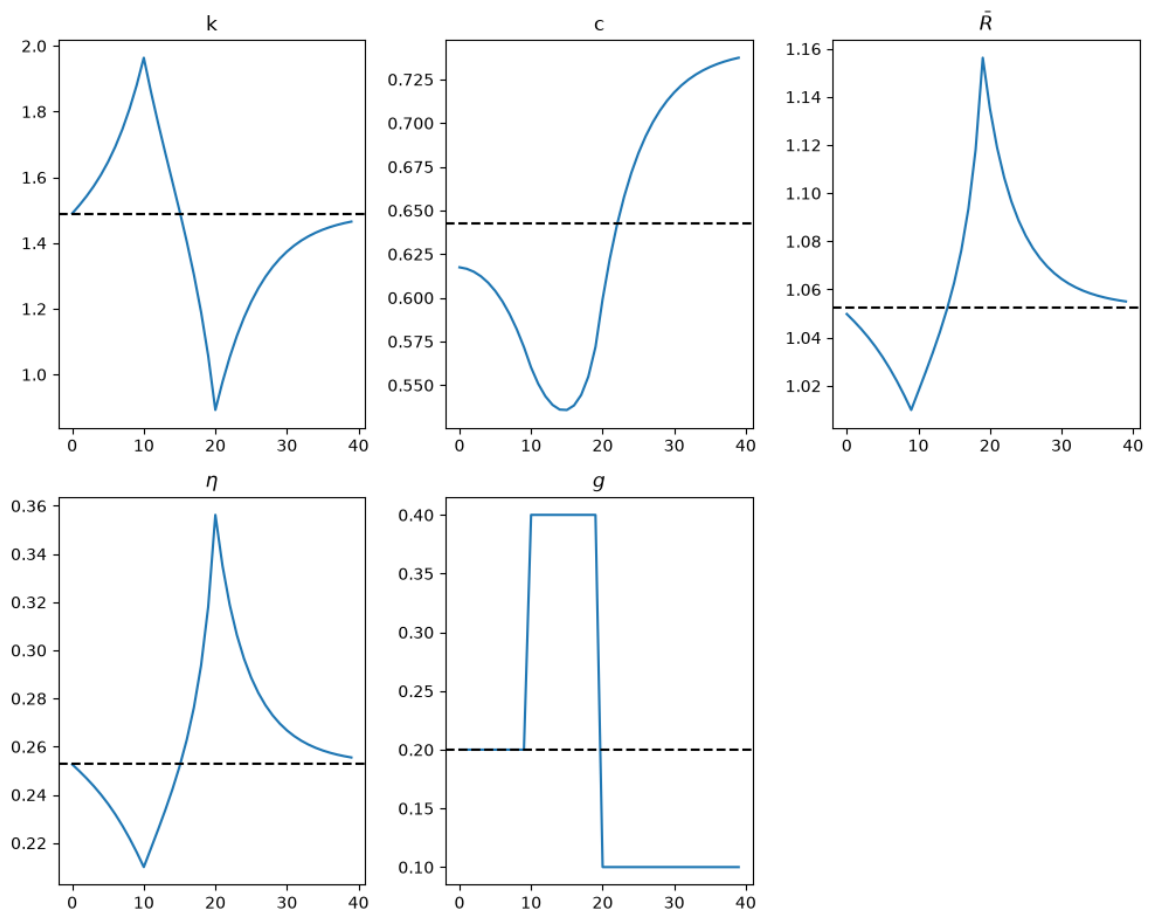
Design a new experiment where the government expenditure g increases from 0.2 to 0.4 in period 10, and then decreases to 0.1 in period 20 permanently.

Solution

Here is one solution:

```
g_path = np.repeat(0.2, S + 1)
g_path[10:20] = 0.4
g_path[20:] = 0.1

shocks = {
    'g': g_path,
    'τ_c': np.repeat(0.0, S + 1),
    'τ_k': np.repeat(0.0, S + 1)
}

experiment_model(shocks, S, model, solver=run_min,
                 plot_func=plot_results,
                 policy_shock='g')
```

```
Steady-state capital: 1.4900
Steady-state consumption: 0.6426
------------------------------------------------------------------
```



104.10 Exogenous growth

In the previous section, we considered a model without exogenous growth.

We set the term A_t in the production function to a constant by setting $A_t = 1$ for all t.

Now we are ready to consider growth.

To incorporate growth, we modify the production function to be

$$Y_t = F(K_t, A_t n_t)$$

where Y_t is aggregate output, N_t is total employment, A_t is labor-augmenting technical change, and $F(K, AN)$ is the same linearly homogeneous production function as before.

We assume that A_t follows the process

$$A_{t+1} = \mu_{t+1} A_t \tag{104.25}$$

and that $\mu_{t+1} = \bar{\mu} > 1$.

```
# Set the constant A parameter to None
model = create_model(A=None)
```

```
def compute_A_path(A0, shocks, S=100):
    """
    Compute A path over time.
    """
    A_path = np.full(S + 1, A0)
    for t in range(1, S + 1):
        A_path[t] = A_path[t-1] * shocks['μ'][t-1]
    return A_path
```

104.10.1 Inelastic Labor Supply

By linear homogeneity, the production function can be expressed as

$$y_t = f(k_t)$$

where $f(k) = F(k, 1) = k^\alpha$ and $k_t = \frac{K_t}{n_t A_t}$, $y_t = \frac{Y_t}{n_t A_t}$.

k_t and y_t are measured per unit of "effective labor" $A_t n_t$.

We also let $c_t = \frac{C_t}{A_t n_t}$ and $g_t = \frac{G_t}{A_t n_t}$, where C_t and G_t are total consumption and total government expenditures.

We continue to consider the case of inelastic labor supply.

Based on this, feasibility can be summarized by the following modified version of equation (104.15):

$$k_{t+1} = \mu_{t+1}^{-1}[f(k_t) + (1 - \delta)k_t - g_t - c_t] \tag{104.26}$$

Again, by the properties of a linearly homogeneous production function, we have

$$\eta_t = F_k(k_t, 1) = f'(k_t), w_t = F_n(k_t, 1) = f(k_t) - f'(k_t)k_t$$

Since per capita consumption is now $c_t A_t$, the counterpart to the Euler equation (104.17) is

$$u'(c_t A_t) = \beta u'(c_{t+1} A_{t+1}) \frac{(1 + \tau_{ct})}{(1 + \tau_{ct+1})}[(1 - \tau_{kt+1})(f'(k_{t+1}) - \delta) + 1]. \tag{104.27}$$

$\bar{R}_{t+1}$ continues to be defined by (104.21), except that now k_t is capital per effective unit of labor.

Thus, substituting (104.21), (104.27) becomes

$$u'(c_t A_t) = \beta u'(c_{t+1} A_{t+1}) \bar{R}_{t+1}$$

Assuming that the household's utility function is the same as before, we have

$$(c_t A_t)^{-\gamma} = \beta (c_{t+1} A_{t+1})^{-\gamma} \bar{R}_{t+1}$$

Thus, the counterpart to (104.24) is

$$c_{t+1} = c_t \left[\beta \bar{R}_{t+1}\right]^{\frac{1}{\gamma}} \mu_{t+1}^{-1} \tag{104.28}$$

104.10.2 Steady State

In a steady state, $c_{t+1} = c_t$. Then (104.27) becomes

$$1 = \mu^{-\gamma}\beta[(1-\tau_k)(f'(k)-\delta)+1] \tag{104.29}$$

from which we can compute that the steady-state level of capital per unit of effective labor satisfies

$$f'(k) = \delta + (\frac{\frac{1}{\beta}\mu^{\gamma}-1}{1-\tau_k}) \tag{104.30}$$

and that

$$\bar{R} = \frac{\mu^{\gamma}}{\beta} \tag{104.31}$$

The steady-state level of consumption per unit of effective labor can be found using (104.26):

$$c = f(k) + (1-\delta-\mu)k - g$$

Since the algorithm and plotting routines are the same as before, we include the steady-state calculations and shooting routine in the section *Python Code*.

104.10.3 Shooting Algorithm

Now we can apply the shooting algorithm to compute equilibrium. We augment the vector of shock variables by including μ_t, then proceed as before.

104.10.4 Experiments

Let's run some experiments:

1. A foreseen once-and-for-all increase in μ from 1.02 to 1.025 in period 10
2. An unforeseen once-and-for-all increase in μ to 1.025 in period 0

Experiment 1: A foreseen increase in μ from 1.02 to 1.025 at t=10

The figures below show the effects of a permanent increase in productivity growth μ from 1.02 to 1.025 at t=10.

They now measure c and k in effective units of labor.

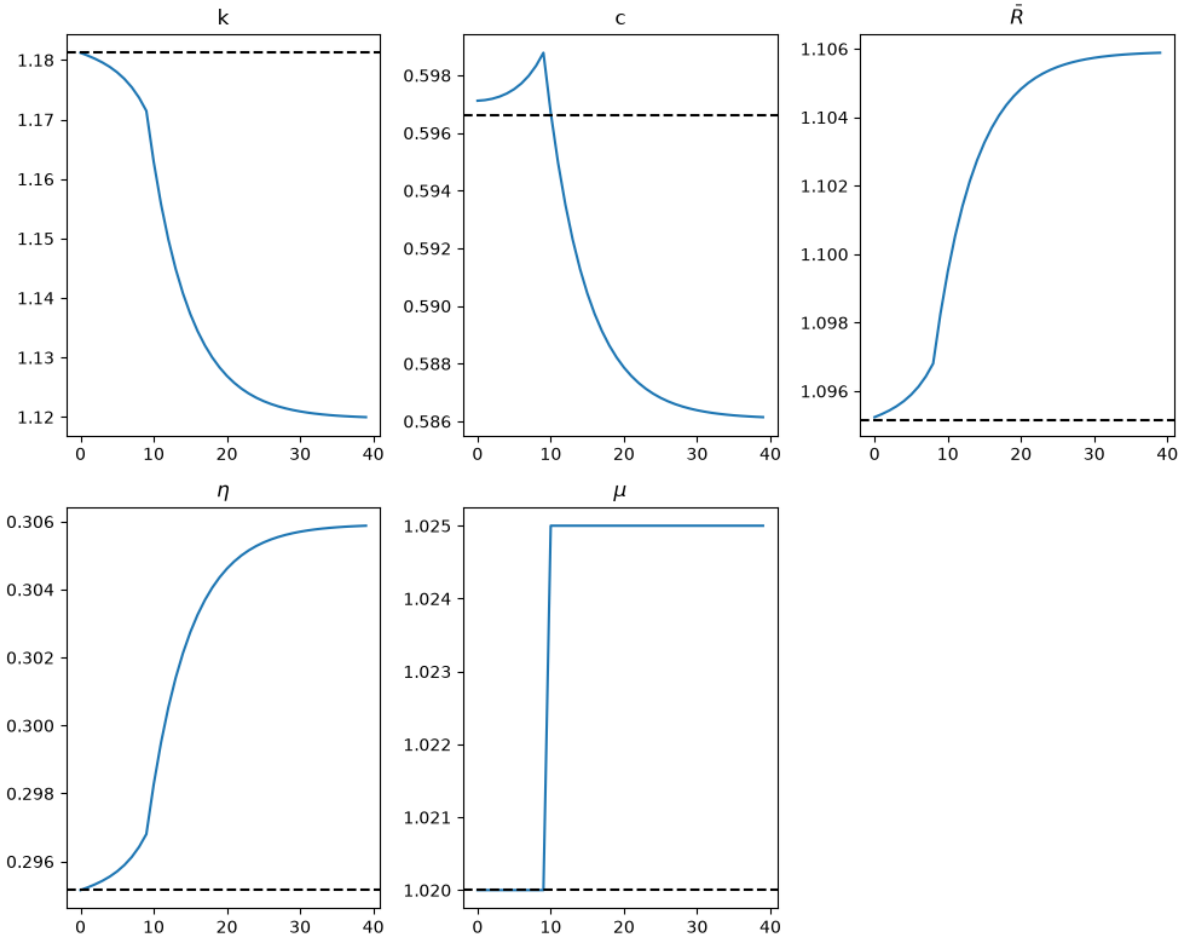
```
shocks = {
    'g': np.repeat(0.2, S + 1),
    'τ_c': np.repeat(0.0, S + 1),
    'τ_k': np.repeat(0.0, S + 1),
    'μ': np.concatenate((np.repeat(1.02, 10), np.repeat(1.025, S - 9)))
}

A_path = compute_A_path(1.0, shocks, S)

k_ss_initial, c_ss_initial = steady_states(model,
                                           shocks['g'][0],
                                           shocks['τ_k'][0],
                                           shocks['μ'][0]
```

(continues on next page)

(continued from previous page)

```
                                        )

print(f"Steady-state capital: {k_ss_initial:.4f}")
print(f"Steady-state consumption: {c_ss_initial:.4f}")

# Run the shooting algorithm with the A_path parameter
solution = run_shooting(shocks, S, model, A_path)

fig, axes = plt.subplots(2, 3, figsize=(10, 8))
axes = axes.flatten()

plot_results(solution, k_ss_initial,
             c_ss_initial, shocks, 'μ', axes, model,
             A_path, T=40)

for ax in axes[5:]:
    fig.delaxes(ax)

plt.tight_layout()
plt.show()
```

```
Steady-state capital: 1.1812
Steady-state consumption: 0.5966
Model: Model(β=0.95, γ=2.0, δ=0.2, α=0.33, A=None)
Optimal initial consumption c0 = 0.597118474934446239627091833718360733991 9
```



The results in the figures are mainly driven by (104.29) and imply that a permanent increase in μ will lead to a decrease in the steady-state value of capital per unit of effective labor.

The figures indicate the following:

- As capital becomes more efficient, even with less of it, consumption per capita can be raised.
- Consumption smoothing drives an *immediate jump in consumption* in anticipation of the increase in μ.
- The increased productivity of capital leads to an increase in the gross return $\bar{R}$.
- Perfect foresight makes the effects of the increase in the growth of capital precede it, with the effect visible at $t = 0$.

Experiment 2: An unforeseen increase in μ from 1.02 to 1.025 at t=0

The figures below show the effects of an immediate jump in μ to 1.025 at t=0.

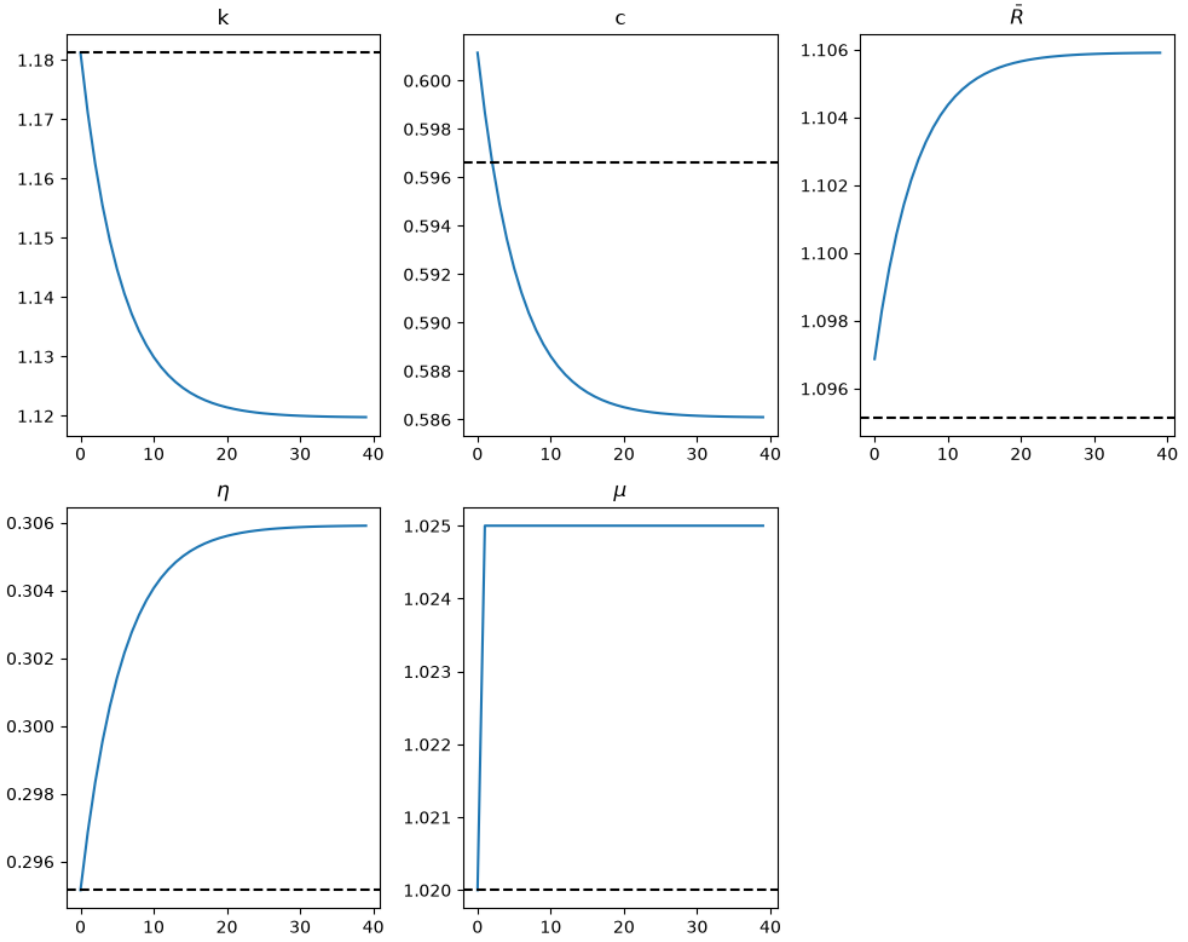
```
shocks = {
    'g': np.repeat(0.2, S + 1),
    'τ_c': np.repeat(0.0, S + 1),
    'τ_k': np.repeat(0.0, S + 1),
    'μ': np.concatenate((np.repeat(1.02, 1), np.repeat(1.025, S)))
}

A_path = compute_A_path(1.0, shocks, S)
```

(continues on next page)

(continued from previous page)

```
k_ss_initial, c_ss_initial = steady_states(model,
                                           shocks['g'][0],
                                           shocks['τ_k'][0],
                                           shocks['μ'][0]
                                          )

print(f"Steady-state capital: {k_ss_initial:.4f}")
print(f"Steady-state consumption: {c_ss_initial:.4f}")

# Run the shooting algorithm with the A_path parameter
solution = run_shooting(shocks, S, model, A_path)

fig, axes = plt.subplots(2, 3, figsize=(10, 8))
axes = axes.flatten()

plot_results(solution, k_ss_initial,
             c_ss_initial, shocks, 'μ', axes, model, A_path, T=40)

for ax in axes[5:]:
    fig.delaxes(ax)

plt.tight_layout()
plt.show()
```

```
Steady-state capital: 1.1812
Steady-state consumption: 0.5966
Model: Model(β=0.95, γ=2.0, δ=0.2, α=0.33, A=None)
Optimal initial consumption c0 = 0.601149493043064115039588375310958831663 8
```

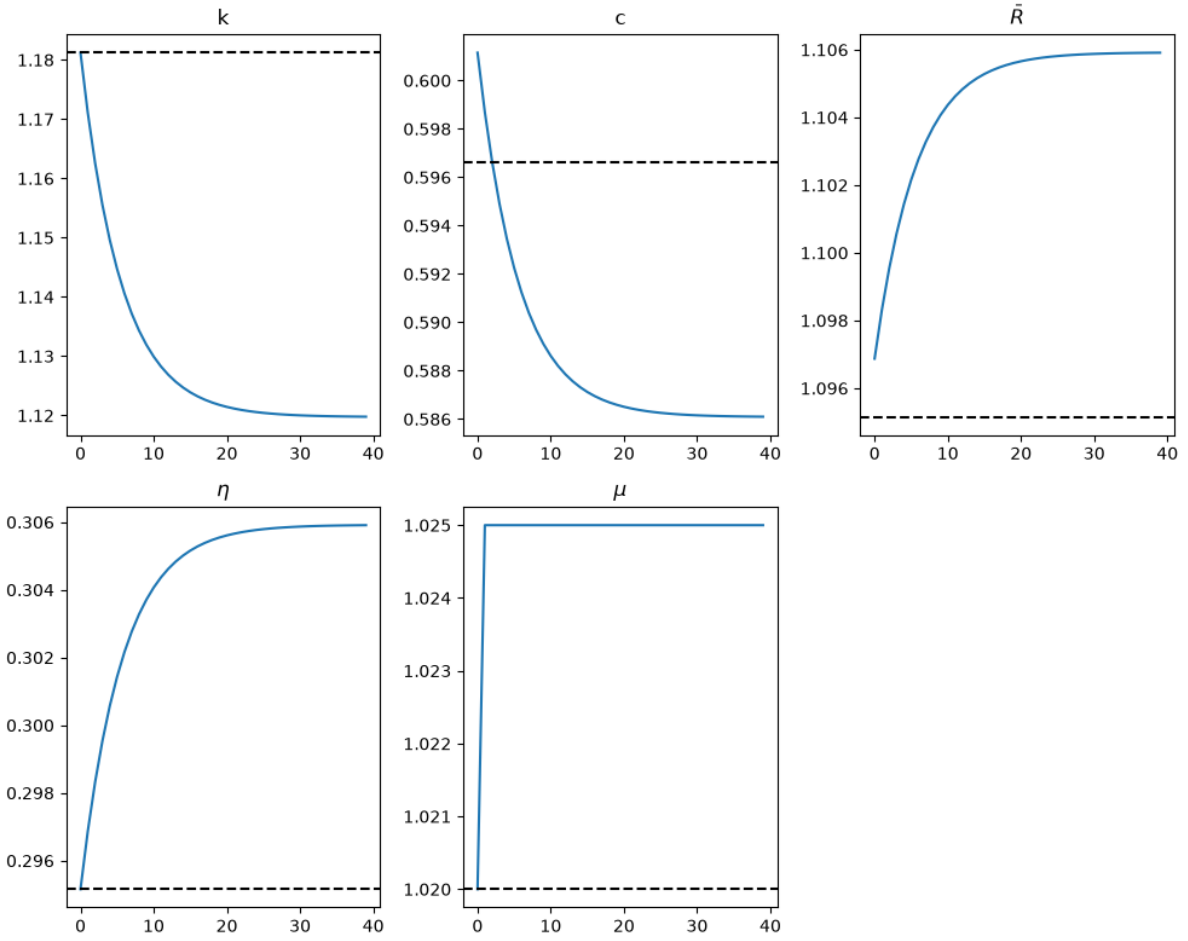


Again, we can collect the procedures used above into a function that runs the solver and draws plots for a given experiment.

```
def experiment_model(shocks, S, model, A_path, solver, plot_func, policy_shock, T=40):
    """
    Run the shooting algorithm given a model and plot the results.
    """
    k0, c0 = steady_states(model, shocks['g'][0], shocks['τ_k'][0], shocks['μ'][0])

    print(f"Steady-state capital: {k0:.4f}")
    print(f"Steady-state consumption: {c0:.4f}")
    print('-'*64)

    fig, axes = plt.subplots(2, 3, figsize=(10, 8))
    axes = axes.flatten()

    solution = solver(shocks, S, model, A_path)
    plot_func(solution, k0, c0,
              shocks, policy_shock, axes, model, A_path, T=T)

    for ax in axes[5:]:
        fig.delaxes(ax)

    plt.tight_layout()
    plt.show()
```

```
shocks = {
    'g': np.repeat(0.2, S + 1),
    'τ_c': np.repeat(0.0, S + 1),
    'τ_k': np.repeat(0.0, S + 1),
    'μ': np.concatenate((np.repeat(1.02, 1), np.repeat(1.025, S)))
}

experiment_model(shocks, S, model, A_path, run_shooting, plot_results, 'μ')
```

```
Steady-state capital: 1.1812
Steady-state consumption: 0.5966
----------------------------------------------------------------------
Model: Model(β=0.95, γ=2.0, δ=0.2, α=0.33, A=None)
Optimal initial consumption c0 = 0.601149493043064115039588375310958831663 8
```



The figures show that:

- The paths of all variables are now smooth due to the absence of feedforward effects.
- Capital per effective unit of labor gradually declines to a lower steady-state level.
- Consumption per effective unit of labor jumps immediately and then declines smoothly toward its lower steady-state value.
- The after-tax gross return $\bar{R}$ once again co-moves with the consumption growth rate, verifying the Euler equation (104.29).

Exercise 104.10.1

Replicate the plots of our two experiments using the second method of residual minimization:

1. A foreseen increase in μ from 1.02 to 1.025$ at t=10
2. An unforeseen increase in μ from 1.02 to 1.025$ at t=0

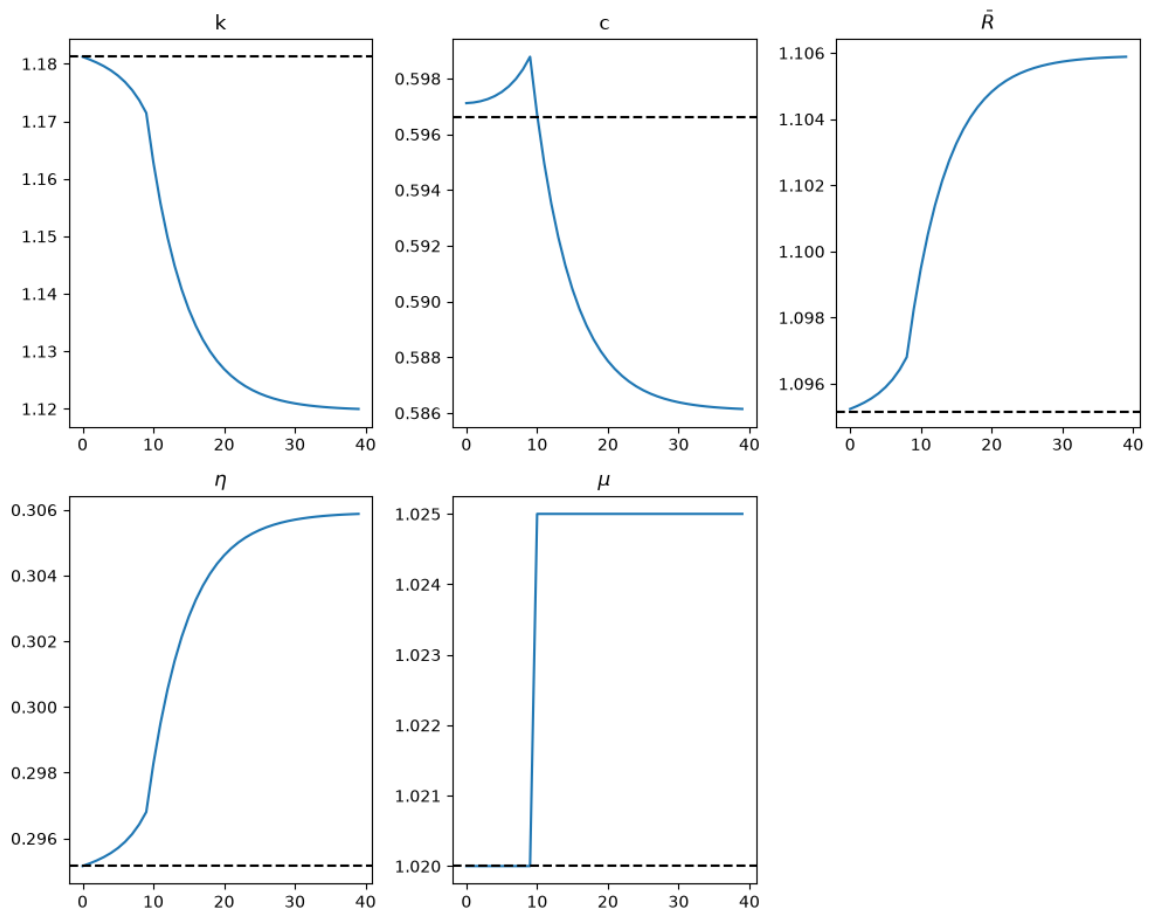
Solution

Here is one solution:

Experiment 1: A foreseen increase in μ from 1.02 to 1.025 at $t = 10$

```
shocks = {
    'g': np.repeat(0.2, S + 1),
    'τ_c': np.repeat(0.0, S + 1),
    'τ_k': np.repeat(0.0, S + 1),
    'μ': np.concatenate((np.repeat(1.02, 10), np.repeat(1.025, S - 9)))
}

A_path = compute_A_path(1.0, shocks, S)

experiment_model(shocks, S, model, A_path, run_min, plot_results, 'μ')
```

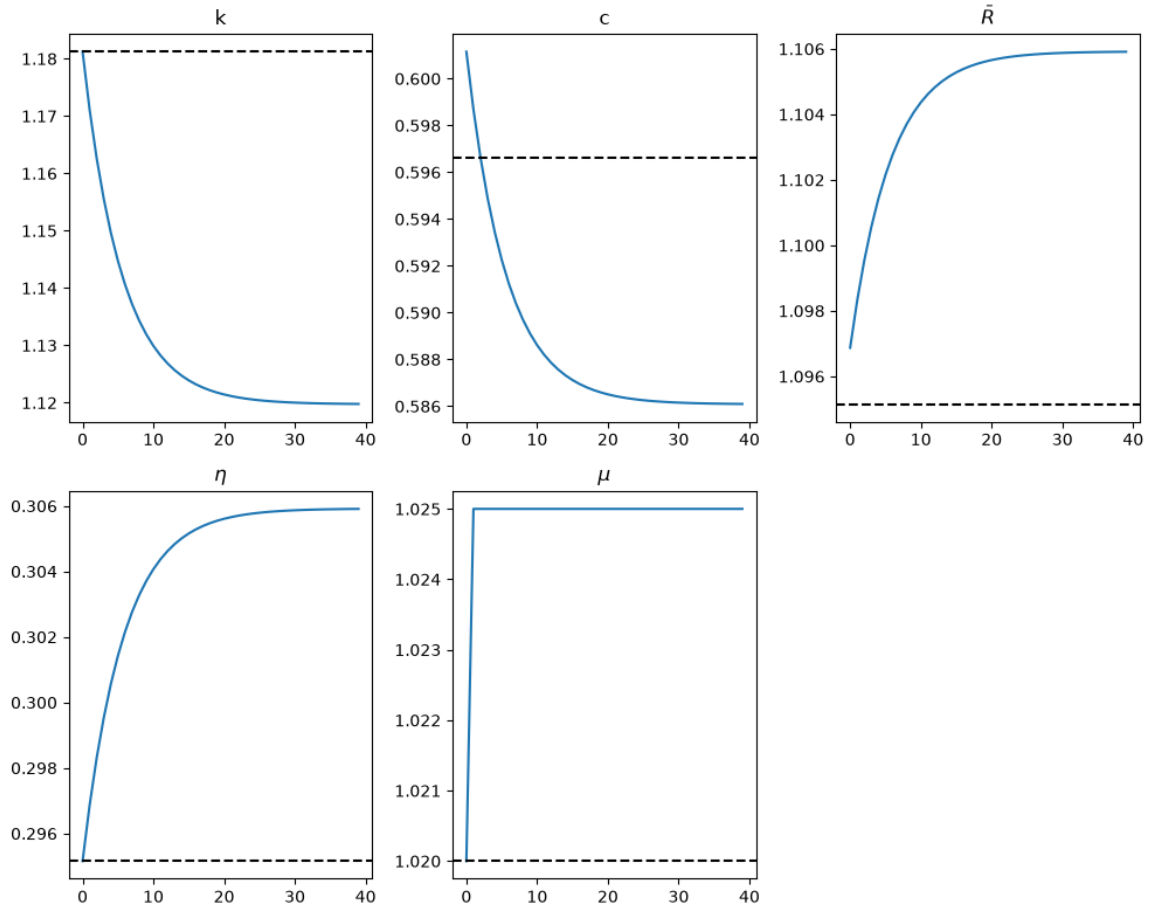
```
Steady-state capital: 1.1812
Steady-state consumption: 0.5966
----------------------------------------------------------------------
```



Experiment 2: An unforeseen increase in μ from 1.02 to 1.025 at $t = 0$

```
shocks = {
    'g': np.repeat(0.2, S + 1),
    'τ_c': np.repeat(0.0, S + 1),
    'τ_k': np.repeat(0.0, S + 1),
    'μ': np.concatenate((np.repeat(1.02, 1), np.repeat(1.025, S)))
}

experiment_model(shocks, S, model, A_path, run_min, plot_results, 'μ')
```

```
Steady-state capital: 1.1812
Steady-state consumption: 0.5966
----------------------------------------------------------------
```



In this this sequel *Two-Country Model with Distorting Taxes*, we study a two-country version of our one-country model that is closely related to Mendoza and Tesar [1998].

CHAPTER

FIVE

TWO-COUNTRY MODEL WITH DISTORTING TAXES

105.1 Overview

This lecture is a sequel to this QuantEcon lecture *Cass-Koopmans Model with Distorting Taxes* in which we studied consequences of foreseen fiscal and technology shocks on competitive equilibrium prices and quantities in a nonstochastic version of a Cass-Koopmans growth model like the one described in this QuantEcon lecture *Cass-Koopmans Competitive Equilibrium*.

Here we study a two-country version of that model.

We construct it by putting instances of two *Cass-Koopmans Competitive Equilibrium* economies together back to back, and then opening international trade in some commodities, but not in others.

This lets us focus on some of the issues studied by Mendoza and Tesar [1998].

Let's start with some imports:

```
import numpy as np
from scipy.optimize import root
import matplotlib.pyplot as plt
from collections import namedtuple
from mpmath import mp, mpf
from warnings import warn

# Set the precision
mp.dps = 40
mp.pretty = True
```

105.2 A Two-Country Cass-Koopmans Model

This section describes a two-country version of the basic model of *The Economy*.

The model has a structure similar to ones used in the international real business cycle literature and is in the spirit of an analysis of distorting taxes by Mendoza and Tesar [1998].

We allow two countries to trade goods and claims on future goods, but not labor.

Both countries have production technologies, and consumers in each country can hold capital in either country, subject to different tax treatments.

We denote variables in the second country with asterisks (*).

Households in both countries maximize lifetime utility:

$$\sum_{t=0}^{\infty} \beta^t u(c_t) \quad \text{and} \quad \sum_{t=0}^{\infty} \beta^t u(c_t^*),$$

where $u(c) = \frac{c^{1-\gamma}}{1-\gamma}$ with $\gamma > 0$.

There are Cobb-Douglas functions with identical technology parameters in the two countries.

The world resource constraint in this two-country economy is:

$$(c_t + c_t^*) + (g_t + g_t^*) + (k_{t+1} - (1-\delta)k_t) + (k_{t+1}^* - (1-\delta)k_t^*) = f(k_t) + f(k_t^*)$$

which combines the feasibility constraints for the two countries.

Later, we will use this constraint as a global feasibility constraint in our computation.

To connect the two countries, we need to specify how capital flows across borders and how taxes are levied in different jurisdictions.

105.2.1 Capital Mobility and Taxation

A consumer in country one can hold capital in either country but pays taxes on rentals from foreign holdings of capital at the rate set by the foreign country.

Residents in both countries can purchase consumption at date t at a common Arrow-Debreu price q_t. We assume capital markets are complete.

Let B_t^f be the amount of time t goods that the representative domestic consumer raises by issuing a one-period IOU to the representative foreign consumer.

So $B_t^f > 0$ indicates the domestic consumer is borrowing from abroad at t, and $B_t^f < 0$ indicates the domestic consumer is lending abroad at t.

Hence, the budget constraint of a representative consumer in country one is:

$$\sum_{t=0}^{\infty} q_t \left(c_t + (k_{t+1} - (1-\delta)k_t) + (\tilde{k}_{t+1} - (1-\delta)\tilde{k}_t) + R_{t-1,t} B_{t-1}^f \right) \leq$$

$$\sum_{t=0}^{\infty} q_t \left((\eta_t - \tau_{kt}(\eta_t - \delta))k_t + (\eta_t^* - \tau_{kt}^*(\eta_t^* - \delta))\tilde{k}_t + (1-\tau_{nt}) w_t n_t - \tau_{ht} + B_t^f \right).$$

No-arbitrage conditions for k_t and $\tilde{k}_t$ for $t \geq 1$ imply

$$q_{t-1} = [(1-\tau_{kt})(\eta_t - \delta) + 1] q_t,$$
$$q_{t-1} = [(1-\tau_{kt}^*)(\eta_t^* - \delta) + 1] q_t,$$

which together imply that after-tax rental rates on capital are equalized across the two countries:

$$(1-\tau_{kt}^*)(\eta_t^* - \delta) = (1-\tau_{kt})(\eta_t - \delta).$$

The no-arbitrage conditions for B_t^f for $t \geq 0$ are $q_t = q_{t+1} R_{t+1,t}$, which implies that

$$q_{t-1} = q_t R_{t-1,t}$$

for $t \geq 1$.

Since domestic capital, foreign capital, and consumption loans bear the same rates of return, portfolios are indeterminate.

We can set holdings of foreign capital equal to zero in each country if we allow B_t^f to be nonzero.

This way of resolving portfolio indeterminacy is convenient because it reduces the number of initial conditions we need to specify.

Therefore, we set holdings of foreign capital equal to zero in both countries while allowing international lending.

Given an initial level B_{-1}^f of debt from the domestic country to the foreign country, and where $R_{t-1,t} = \frac{q_{t-1}}{q_t}$, international debt dynamics satisfy

$$B_t^f = R_{t-1,t} B_{t-1}^f + c_t + (k_{t+1} - (1-\delta)k_t) + g_t - f(k_t)$$

```
def Bf_path(k, c, g, model):
    """
    Compute B^{f}_t:
      Bf_t = R_{t-1} Bf_{t-1} + c_t + (k_{t+1}-(1-δ)k_t) + g_t - f(k_t)
    with Bf_0 = 0.
    """
    S = len(c) - 1
    R = c[:-1]**(-model.γ) / (model.β * c[1:]**(-model.γ))

    Bf = np.zeros(S + 1)
    for t in range(1, S + 1):
        inv = k[t] - (1 - model.δ) * k[t-1]
        Bf[t] = (
            R[t-1] * Bf[t-1] + c[t] + inv + g[t-1]
            - f(k[t-1], model))
    return Bf

def Bf_ss(c_ss, k_ss, g_ss, model):
    """
    Compute the steady-state B^f
    """
    R_ss   = 1.0 / model.β
    inv_ss = model.δ * k_ss
    num    = c_ss + inv_ss + g_ss - f(k_ss, model)
    den    = 1.0 - R_ss
    return num / den
```

and

$$c_t^* + (k_{t+1}^* - (1-\delta)k_t^*) + g_t^* - R_{t-1,t} B_{t-1}^f = f(k_t^*) - B_t^f.$$

The firms' first-order conditions in the two countries are:

$$\begin{aligned} \eta_t &= f'(k_t), \quad w_t = f(k_t) - k_t f'(k_t) \\ \eta_t^* &= f'(k_t^*), \quad w_t^* = f(k_t^*) - k_t^* f'(k_t^*). \end{aligned}$$

International trade in goods establishes:

$$\frac{q_t}{\beta^t} = \frac{u'(c_t)}{1+\tau_{ct}} = \mu^* \frac{u'(c_t^*)}{1+\tau_{ct}^*},$$

where μ^* is a nonnegative number that is a function of the Lagrange multiplier on the budget constraint for a consumer in country $*$.

We have normalized the Lagrange multiplier on the budget constraint of the domestic country to set the corresponding μ for the domestic country to unity.

```
def compute_rs(c_t, c_tp1, c_s_t, c_s_tp1, τc_t,
               τc_tp1, τc_s_t, τc_s_tp1, model):
    """
    Compute international risk sharing after trade starts.
    """

    return (c_t**(-model.γ)/(1+τc_t)) * ((1+τc_s_t)/c_s_t**(-model.γ)) - (
        c_tp1**(-model.γ)/(1+τc_tp1)) * ((1+τc_s_tp1)/c_s_tp1**(-model.γ))
```

Equilibrium requires that the following two national Euler equations be satisfied for $t \geq 0$:

$$u'(c_t) = \beta u'(c_{t+1}) \left[(1 - \tau_{kt+1})(f'(k_{t+1}) - \delta) + 1\right] \left[\frac{1 + \tau_{ct+1}}{1 + \tau_{ct}}\right],$$

$$u'(c_t^*) = \beta u'(c_{t+1}^*) \left[(1 - \tau_{kt+1}^*)(f'(k_{t+1}^*) - \delta) + 1\right] \left[\frac{1 + \tau_{ct+1}^*}{1 + \tau_{ct}^*}\right].$$

The following code computes both the domestic and foreign Euler equations.

Since they have the same form but use different variables, we can write a single function that handles both cases.

```
def compute_euler(c_t, c_tp1, τc_t,
                  τc_tp1, τk_tp1, k_tp1, model):
    """
    Compute the Euler equation.
    """
    Rbar = (1 - τk_tp1)*(f_prime(k_tp1, model) - model.δ) + 1
    return model.β * (c_tp1/c_t)**(-model.γ) * (1+τc_t)/(1+τc_tp1) * Rbar - 1
```

105.2.2 Initial condition and steady state

For the initial conditions, we choose the pre-trade allocation of capital (k_0, k_0^*) and the initial level B_{-1}^f of international debt owed by the unstarred (domestic) country to the starred (foreign) country.

105.2.3 Equilibrium steady state values

The steady state of the two-country model is characterized by two sets of equations.

First, the following equations determine the steady-state capital-labor ratios $\bar{k}$ and $\bar{k}^*$ in each country:

$$f'(\bar{k}) = \delta + \frac{\rho}{1 - \tau_k} \tag{105.1}$$

$$f'(\bar{k}^*) = \delta + \frac{\rho}{1 - \tau_k^*} \tag{105.2}$$

Given these steady-state capital-labor ratios, the domestic and foreign consumption values $\bar{c}$ and $\bar{c}^*$ are determined by:

$$(\bar{c} + \bar{c}^*) = f(\bar{k}) + f(\bar{k}^*) - \delta(\bar{k} + \bar{k}^*) - (\bar{g} + \bar{g}^*) \tag{105.3}$$

$$\bar{c} = f(\bar{k}) - \delta\bar{k} - \bar{g} - \rho\bar{B}^f \tag{105.4}$$

Equation (105.3) expresses feasibility at the steady state, while equation (105.4) represents the trade balance, including interest payments, at the steady state.

The steady-state level of debt $\bar{B}^f$ from the domestic country to the foreign country influences the consumption allocation between countries but not the total world capital stock.

We assume $\bar{B}^f = 0$ in the steady state, which gives us the following function to compute the steady-state values of capital and consumption

```
def compute_steady_state_global(model, g_ss=0.2):
    """
    Calculate steady state values for capital, consumption, and investment.
    """
    k_ss = ((1/model.β - (1-model.δ)) / (model.A * model.α)) ** (1/(model.α-1))
    c_ss = f(k_ss, model) - model.δ * k_ss - g_ss
    return k_ss, c_ss
```

Now, we can apply the residual minimization method to compute the steady-state values of capital and consumption.

Again, we minimize the residuals of the Euler equation, the global resource constraint, and the no-arbitrage condition.

```
def compute_residuals_global(z, model, shocks, T, k0_ss, k_star, Bf_star):
    """
    Compute residuals for the two-country model.
    """
    k, c, k_s, c_s = z.reshape(T+1, 4).T
    g, gs = shocks['g'], shocks['g_s']
    τc, τk = shocks['τ_c'], shocks['τ_k']
    τc_s, τk_s = shocks['τ_c_s'], shocks['τ_k_s']

    res = [k[0] - k0_ss, k_s[0] - k0_ss]

    for t in range(T):
        e_d = compute_euler(
            c[t], c[t+1],
            τc[t], τc[t+1], τk[t+1],
            k[t+1], model)

        e_f = compute_euler(
            c_s[t], c_s[t+1],
            τc_s[t], τc_s[t+1], τk_s[t+1],
            k_s[t+1], model)

        rs = compute_rs(
            c[t], c[t+1], c_s[t], c_s[t+1],
            τc[t], τc[t+1], τc_s[t], τc_s[t+1],
            model)

        # Global resource constraint
        grc = k[t+1] + k_s[t+1] - (
            f(k[t], model) + f(k_s[t], model) +
            (1-model.δ)*(k[t] + k_s[t]) -
            c[t] - c_s[t] - g[t] - gs[t]
        )

        res.extend([e_d, e_f, rs, grc])

    Bf_term = Bf_path(k, c, shocks['g'], model)[-1]
    res.append(k[T] - k_star)
    res.append(Bf_term - Bf_star)
    return np.array(res)
```

Now we plot the results

```
# Function to plot global two-country model results
def plot_global_results(k, k_s, c, c_s, shocks, model,
                        k0_ss, c0_ss, g_ss, S, T=40, shock='g',
                        # a dictionary storing sequence for lower left panel
                        ll_series='None'):
    """
    Plot results for the two-country model.
    """
    fig, axes = plt.subplots(2, 3, figsize=(10, 8))
    x = np.arange(T)
    τc, τk = shocks['τ_c'], shocks['τ_k']
    Bf = Bf_path(k, c, shocks['g'], model)

    # Compute derived series
    R_ratio = c[:-1]**(-model.γ) / (model.β * c[1:]**(-model.γ)) \
    *(1+τc[:-1])/(1+τc[1:])
    inv = k[1:] - (1-model.δ)*k[:-1]
    inv_s = k_s[1:] - (1-model.δ)*k_s[:-1]

    # Add initial conditions into the series
    R_ratio = np.append(1/model.β, R_ratio)
    c = np.append(c0_ss, c)
    c_s = np.append(c0_ss, c_s)
    k = np.append(k0_ss, k)
    k_s = np.append(k0_ss, k_s)

    # Capital
    axes[0,0].plot(x, k[:T], '-', lw=1.5)
    axes[0,0].plot(x, np.full(T, k0_ss), 'k-.', lw=1.5)
    axes[0,0].plot(x, k_s[:T], '--', lw=1.5)
    axes[0,0].set_title('k')
    axes[0,0].set_xlim(0, T-1)

    # Consumption
    axes[0,1].plot(x, c[:T], '-', lw=1.5)
    axes[0,1].plot(x, np.full(T, c0_ss), 'k-.', lw=1.5)
    axes[0,1].plot(x, c_s[:T], '--', lw=1.5)
    axes[0,1].set_title('c')
    axes[0,1].set_xlim(0, T-1)

    # Interest rate
    axes[0,2].plot(x, R_ratio[:T], '-', lw=1.5)
    axes[0,2].plot(x, np.full(T, 1/model.β), 'k-.', lw=1.5)
    axes[0,2].set_title(r'$\bar{R}$')
    axes[0,2].set_xlim(0, T-1)

    # Investment
    axes[1,0].plot(x, np.full(T, model.δ * k0_ss),
    'k-.', lw=1.5)
    axes[1,0].plot(x, np.append(model.δ*k0_ss, inv[:T-1]),
    '-', lw=1.5)
    axes[1,0].plot(x, np.append(model.δ*k0_ss, inv_s[:T-1]),
    '--', lw=1.5)
    axes[1,0].set_title('x')
    axes[1,0].set_xlim(0, T-1)

    # Shock
```

(continues on next page)

(continued from previous page)

```
    axes[1,1].plot(x, shocks[shock][:T], '-', lw=1.5)
    axes[1,1].plot(x, np.full(T, shocks[shock][0]), 'k-.', lw=1.5)
    axes[1,1].set_title(f'${shock}$')
    axes[1,1].set_ylim(-0.1, 0.5)
    axes[1,1].set_xlim(0, T-1)

    # Capital flow
    axes[1,2].plot(x, np.append(0, Bf[1:T]), lw=1.5)
    axes[1,2].plot(x, np.zeros(T), 'k-.', lw=1.5)
    axes[1,2].set_title(r'$B^{f}$')
    axes[1,2].set_xlim(0, T-1)

    plt.tight_layout()
    return fig, axes
```

As in our in the one-country model in *Cass-Koopmans Model with Distorting Taxes*, we assume a Cobb-Douglas production function:

$$F(k, 1) = Ak^{\alpha}$$

```
def f(k, model, A=1):
    """
    Production function: f(k) = A * k^{α}
    """
    return A * k ** model.α

def f_prime(k, model, A=1):
    """
    Marginal product of capital: f'(k) = α * A * k^{α - 1}
    """
    return model.α * A * k ** (model.α - 1)
```

Similarly, we define the capital rental rate

$$\eta_t = f'(k_t)$$

```
def compute_η_path(k_path, model, S=100, A_path=None):
    """
    Compute η path: η_t = f'(k_t)
    with optional A_path for growth models.
    """
    A = np.ones_like(k_path) if A_path is None else np.asarray(A_path)
    η_path = np.zeros_like(k_path)
    for t in range(S):
        η_path[t] = f_prime(k_path[t], model, A[t])
    return η_path
```

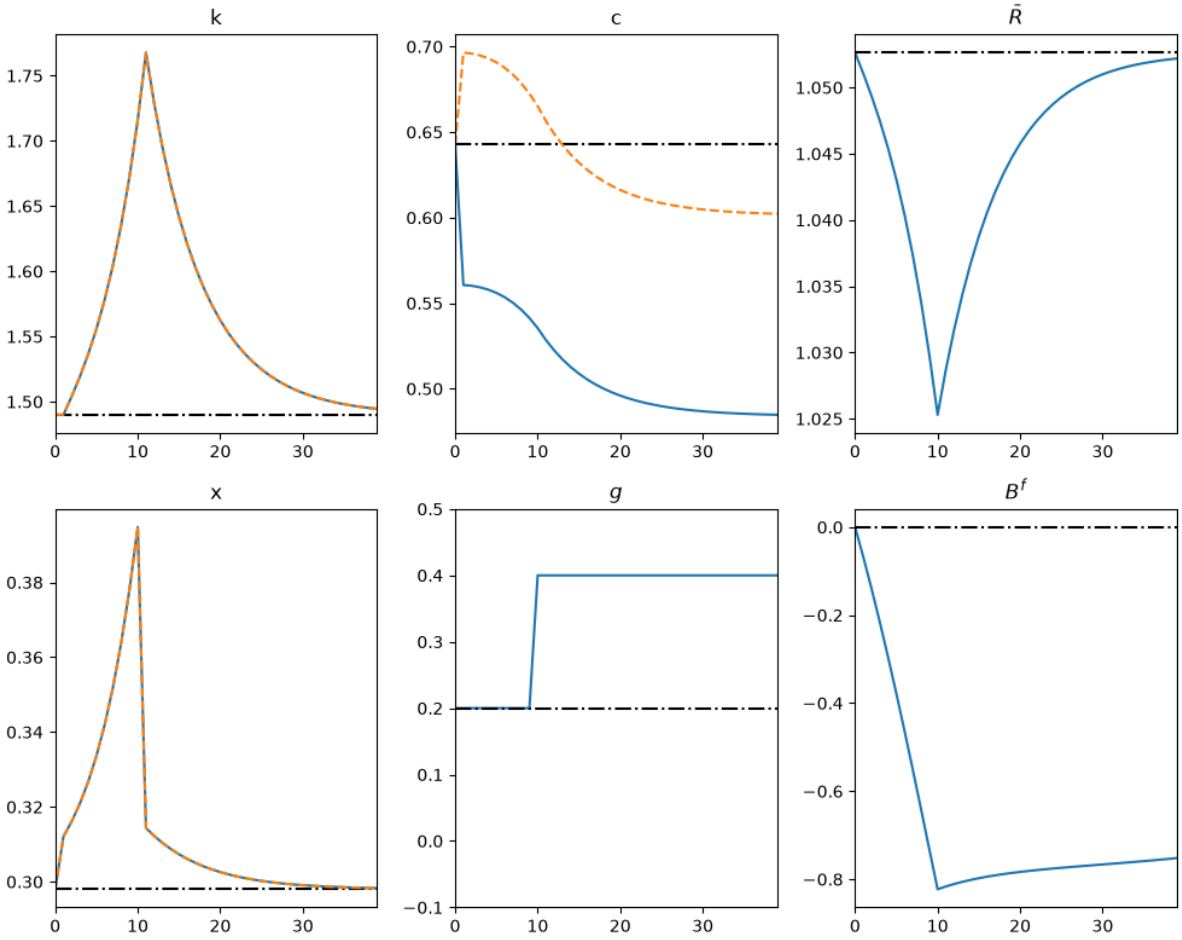
Experiment 1: A foreseen increase in g from 0.2 to 0.4 at t=10

The figure below presents transition dynamics after an increase in g in the domestic economy from 0.2 to 0.4 that is announced ten periods in advance.

We start both economies from a steady state with $B_0^f = 0$.

In the figure below, the blue lines represent the domestic economy and orange dotted lines represent the foreign economy.

```
Model = namedtuple("Model", ["β", "γ", "δ", "α", "A"])
model = Model(β=0.95, γ=2.0, δ=0.2, α=0.33, A=1.0)
S = 100

shocks_global = {
    'g': np.concatenate((np.full(10, 0.2), np.full(S-9, 0.4))),
    'g_s': np.full(S+1, 0.2),
    'τ_c': np.zeros(S+1),
    'τ_k': np.zeros(S+1),
    'τ_c_s': np.zeros(S+1),
    'τ_k_s': np.zeros(S+1)
}
g_ss = 0.2
k0_ss, c0_ss = compute_steady_state_global(model, g_ss)

k_star = k0_ss
Bf_star = Bf_ss(c0_ss, k_star, g_ss, model)

init_glob = np.tile([k0_ss, c0_ss, k0_ss, c0_ss], S+1)
sol_glob = root(
    lambda z: compute_residuals_global(z, model, shocks_global,
                                       S, k0_ss, k_star, Bf_star),
    init_glob, tol=1e-12
)
k, c, k_s, c_s = sol_glob.x.reshape(S+1, 4).T

# Plot global results via function
plot_global_results(k, k_s, c, c_s,
                        shocks_global, model,
                        k0_ss, c0_ss, g_ss,
                        S)
plt.show()
```



At time 1, the government announces that domestic government purchases g will rise ten periods later, cutting into future private resources.

To smooth consumption, domestic households immediately increase saving, offsetting the anticipated hit to their future wealth.

In a closed economy, they would save solely by accumulating extra domestic capital; with open capital markets, they can also lend to foreigners.

Once the capital flow opens up at time 1, the no-arbitrage conditions connect adjustments of both types of saving: the increase in savings by domestic households will reduce the equilibrium return on bonds and capital in the foreign economy to prevent arbitrage opportunities.

Because no-arbitrage equalizes the ratio of marginal utilities, the resulting paths of consumption and capital are synchronized across the two economies.

Up to the date the higher g takes effect, both countries continue to build their capital stocks.

When government spending finally rises 10 periods later, domestic households begin to draw down part of that capital to cushion consumption.

Again by no-arbitrage conditions, when g actually increases, both countries reduce their investment rates.

The domestic economy, in turn, starts running current-account deficits partially to fund the increase in g.

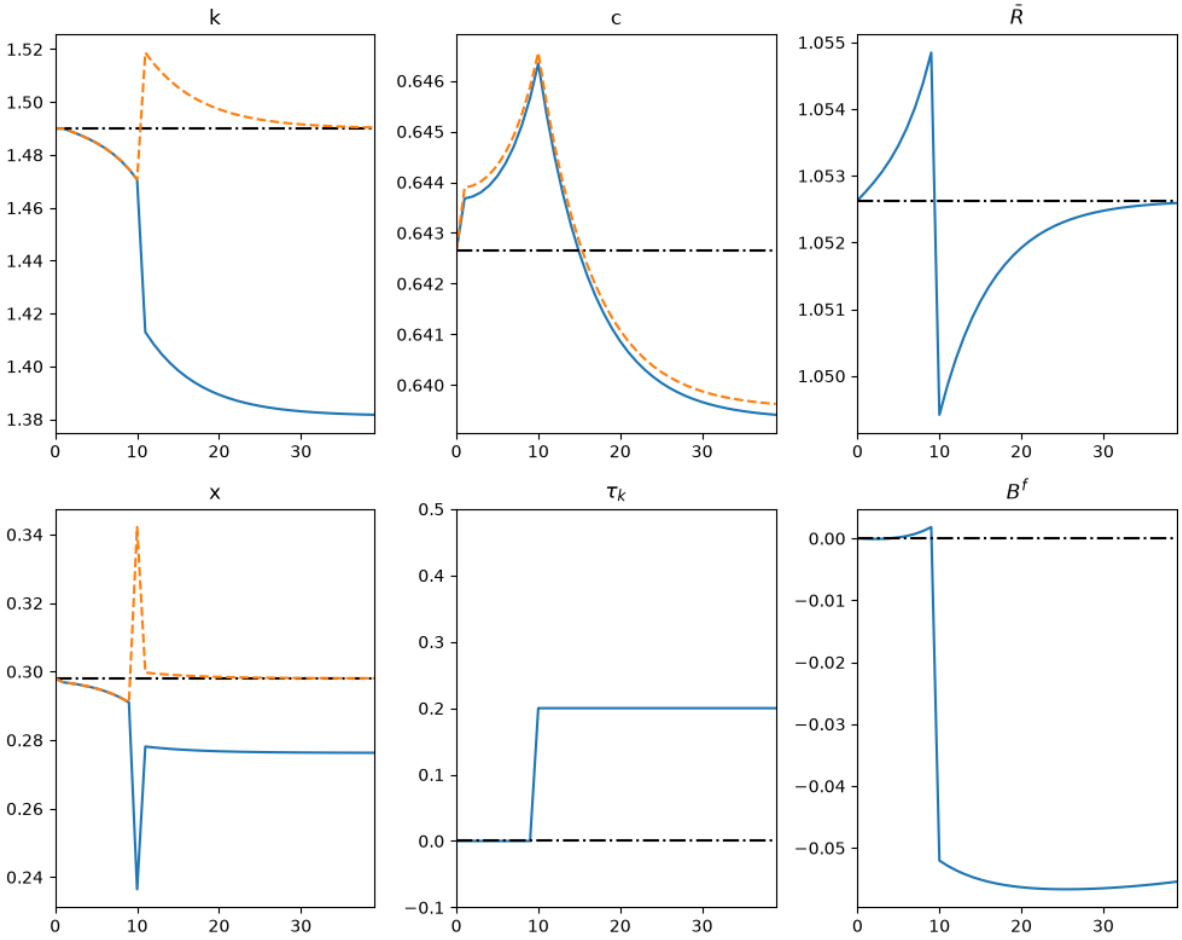
This means that foreign households begin repaying part of their external debt by reducing their capital stock.

Experiment 2: A foreseen increase in g from 0.2 to 0.4 at t=10

We now explore the impact of an increase in capital taxation in the domestic economy 10 periods after its announcement at $t = 1$.

Because the change is anticipated, households in both countries adjust immediately–even though the tax does not take effect until period $t = 11$.

```
shocks_global = {
    'g': np.full(S+1, g_ss),
    'g_s': np.full(S+1, g_ss),
    'τ_c': np.zeros(S+1),
    'τ_k': np.concatenate((np.zeros(10), np.full(S-9, 0.2))),
    'τ_c_s': np.zeros(S+1),
    'τ_k_s': np.zeros(S+1),
}

k0_ss, c0_ss = compute_steady_state_global(model, g_ss)
k_star = k0_ss
Bf_star = Bf_ss(c0_ss, k_star, g_ss, model)

init_glob = np.tile([k0_ss, c0_ss, k0_ss, c0_ss], S+1)

sol_glob = root(
    lambda z: compute_residuals_global(z, model,
            shocks_global, S, k0_ss, k_star, Bf_star),
            init_glob, tol=1e-12)

k, c, k_s, c_s = sol_glob.x.reshape(S+1, 4).T

# plot
fig, axes = plot_global_results(k, k_s, c, c_s, shocks_global, model,
                                k0_ss, c0_ss, g_ss, S, shock='τ_k')
plt.tight_layout()
plt.show()
```



After the tax increase is announced, domestic households foresee lower after-tax returns on capital, so they shift toward higher present consumption and allow the domestic capital stock to decline.

This shrinkage of the world capital supply drives the global real interest rate upward, prompting foreign households to raise current consumption as well.

Prior to the actual tax hike, the domestic economy finances part of its consumption by importing capital, generating a current-account deficit.

When τ_k finally rises, international arbitrage leads investors to reallocate capital quickly toward the untaxed foreign market, compressing the yield on bonds everywhere.

The bond-rate drop reflects the lower after-tax return on domestic capital and the higher foreign capital stock, which depresses its marginal product.

Foreign households fund their capital purchases by borrowing abroad, creating a pronounced current-account deficit and a buildup of external debt.

After the policy change, both countries move smoothly toward a new steady state in which:

- Consumption levels in each economy settle below their pre-announcement paths.
- Capital stocks differ just enough to equalize after-tax returns across borders.

Despite carrying positive net liabilities, the foreign country enjoys higher steady-state consumption because its larger capital stock yields greater output.

The episode demonstrates how open capital markets transmit a domestic tax shock internationally: capital flows and

interest-rate movements share the burden, smoothing consumption adjustments in both the taxed and untaxed economies over time.

Exercise 105.2.1

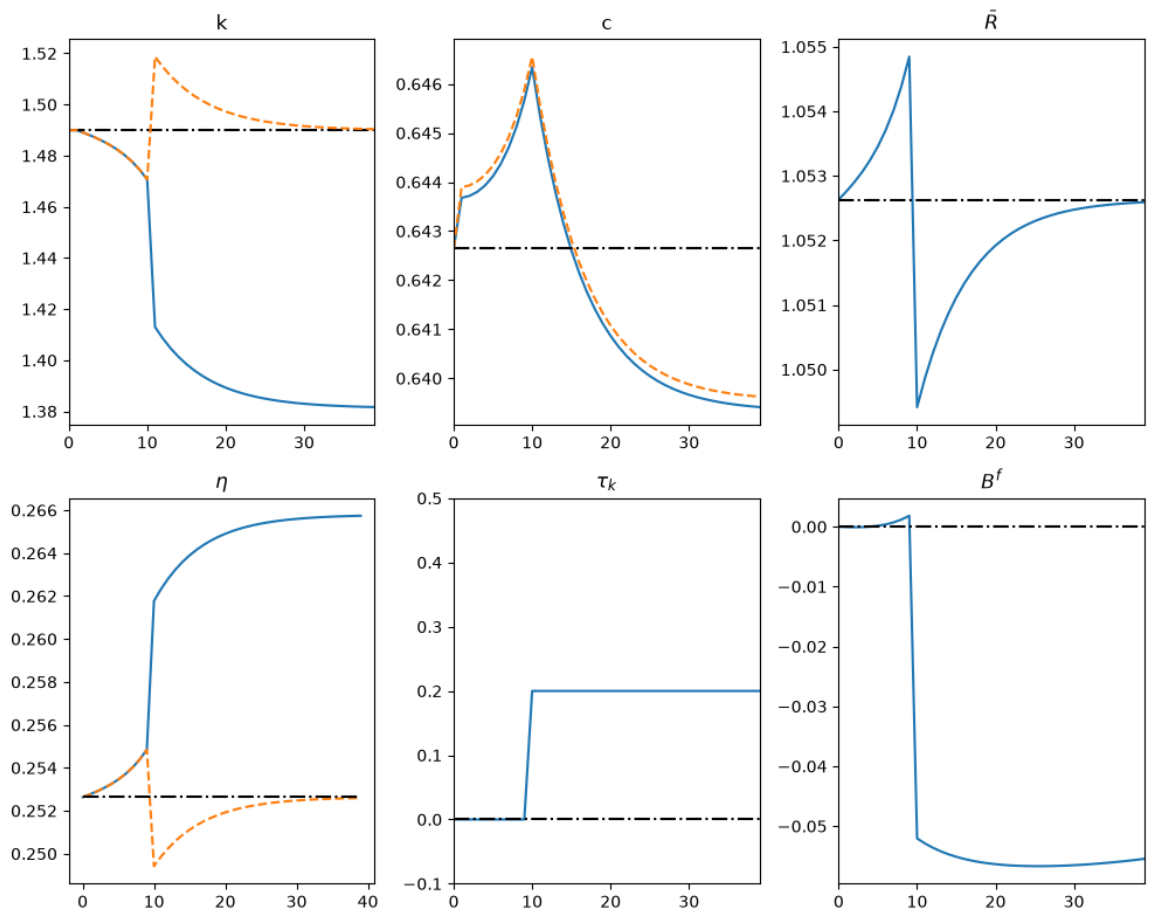
In this exercise, replace the plot for x_t with η_t to replicate the figure in [Ljungqvist and Sargent, 2018].

Compare the figures for k_t and η_t and discuss the economic intuition.

Solution

Here is one solution.

```
fig, axes = plot_global_results(k, k_s, c, c_s, shocks_global, model,
                                k0_ss, c0_ss, g_ss, S, shock='τ_k')

# Clear the plot for x_t
axes[1,0].cla()

# Plot η_t
axes[1,0].plot(compute_η_path(k, model)[:40])
axes[1,0].plot(compute_η_path(k_s, model)[:40], '--')
axes[1,0].plot(np.full(40, f_prime(k_s, model)[0]), 'k-.', lw=1.5)
axes[1,0].set_title(r'$\eta$')

plt.tight_layout()
plt.show()
```



When capital k_t decreases in the domestic country after the tax shock, the rental rate η_t increases in that country.

This happens because when capital becomes scarcer, its marginal product rises.

Part XV

Multiple Agent Models

CHAPTER

SIX

A LAKE MODEL OF EMPLOYMENT AND UNEMPLOYMENT

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

Contents

- *A Lake Model of Employment and Unemployment*
  - *Overview*
  - *The model*
  - *Implementation*
  - *Dynamics of an individual worker*
  - *Exercises*

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install quantecon jax
```

106.1 Overview

This lecture describes what has come to be called a *lake model*.

The lake model is a basic tool for modeling unemployment.

It allows us to analyze

- flows between unemployment and employment
- how these flows influence steady state employment and unemployment rates

It is a good model for interpreting monthly labor department reports on gross and net jobs created and destroyed.

The "lakes" in the model are the pools of employed and unemployed.

The "flows" between the lakes are caused by

- firing and hiring
- entry and exit from the labor force

For the first part of this lecture, the parameters governing transitions into and out of unemployment and employment are exogenous.

Later, we'll determine some of these transition rates endogenously using the *McCall search model*.

We'll also use some nifty concepts like ergodicity, which provides a fundamental link between *cross-sectional* and *long run time series* distributions.

These concepts will help us build an equilibrium model of ex-ante homogeneous workers whose different luck generates variations in their ex post experiences.

Let's start with some imports:

```
import matplotlib.pyplot as plt
import jax
import jax.numpy as jnp
from typing import NamedTuple
from quantecon.distributions import BetaBinomial
from functools import partial
import jax.scipy.stats as stats
```

106.1.1 Prerequisites

Before working through what follows, we recommend you read the *lecture on finite Markov chains*.

You will also need some basic *linear algebra* and probability.

106.2 The model

The economy is inhabited by a very large number of ex-ante identical workers.

The workers live forever, spending their lives moving between unemployment and employment.

Their rates of transition between employment and unemployment are governed by the following parameters:

- λ, the job finding rate for currently unemployed workers
- α, the dismissal rate for currently employed workers
- b, the entry rate into the labor force
- d, the exit rate from the labor force

The growth rate of the labor force evidently equals $g = b - d$.

106.2.1 Aggregate variables

We want to derive the dynamics of the following aggregates:

- E_t, the total number of employed workers at date t
- U_t, the total number of unemployed workers at t
- N_t, the number of workers in the labor force at t

106.2.2 Laws of motion for stock variables

We begin by constructing laws of motion for the aggregate variables E_t, U_t, N_t.

Of the mass of workers E_t who are employed at date t,

- $(1-d)E_t$ will remain in the labor force
- of these, $(1-\alpha)(1-d)E_t$ will remain employed

Of the mass of workers U_t workers who are currently unemployed,

- $(1-d)U_t$ will remain in the labor force
- of these, $(1-d)\lambda U_t$ will become employed

Therefore, the number of workers who will be employed at date $t+1$ will be

$$E_{t+1} = (1-d)(1-\alpha)E_t + (1-d)\lambda U_t$$

A similar analysis implies

$$U_{t+1} = (1-d)\alpha E_t + (1-d)(1-\lambda)U_t + b(E_t + U_t)$$

The value $b(E_t + U_t)$ is the mass of new workers entering the labor force unemployed.

The total stock of workers $N_t = E_t + U_t$ evolves as

$$N_{t+1} = (1+b-d)N_t = (1+g)N_t$$

Letting $X_t := \begin{pmatrix} U_t \\ E_t \end{pmatrix}$, the law of motion for X is

$$X_{t+1} = AX_t \quad \text{where} \quad A := \begin{bmatrix} (1-d)(1-\lambda) + b & (1-d)\alpha + b \\ (1-d)\lambda & (1-d)(1-\alpha) \end{bmatrix}$$

This law tells us how total employment and unemployment evolve over time.

106.2.3 Laws of motion for rates

Now let's derive the law of motion for rates.

We want to track the values of the following objects:

- The employment rate $e_t := E_t/N_t$.
- The unemployment rate $u_t := U_t/N_t$.

(Here and below, capital letters represent aggregates and lowercase letters represent rates)

To get these we can divide both sides of $X_{t+1} = AX_t$ by N_{t+1} to get

$$\begin{bmatrix} U_{t+1}/N_{t+1} \\ E_{t+1}/N_{t+1} \end{bmatrix} = \frac{1}{1+g} A \begin{bmatrix} U_t/N_t \\ E_t/N_t \end{bmatrix}$$

Letting

$$x_t := \begin{pmatrix} u_t \\ e_t \end{pmatrix} = \begin{pmatrix} U_t/N_t \\ E_t/N_t \end{pmatrix}$$

we can also write this as

$$x_{t+1} = Rx_t \quad \text{where} \quad R := \frac{1}{1+g} A$$

You can check that $e_t + u_t = 1$ implies that $e_{t+1} + u_{t+1} = 1$.

This follows from the fact that the columns of R sum to 1.

106.3 Implementation

Let's code up these equations.

106.3.1 Model

To begin, we set up a class called `LakeModel` that stores the primitives α, λ, b, d.

```
class LakeModel(NamedTuple):
    """
    Parameters for the lake model
    """
    λ: float
    α: float
    b: float
    d: float
    A: jnp.ndarray
    R: jnp.ndarray
    g: float


def create_lake_model(
        λ: float = 0.283,      # job finding rate
        α: float = 0.013,      # separation rate
        b: float = 0.0124,     # birth rate
        d: float = 0.00822     # death rate
    ) -> LakeModel:
    """
    Create a LakeModel instance with default parameters.

    Computes and stores the transition matrices A and R,
    and the labor force growth rate g.

    """
    # Compute growth rate
```

(continues on next page)

(continued from previous page)

```
    g = b - d

    # Compute transition matrix A
    A = jnp.array([
        [(1-d) * (1-λ) + b,  (1-d) * α + b],
        [(1-d) * λ,          (1-d) * (1-α)]
    ])

    # Compute normalized transition matrix R
    R = A / (1 + g)

    return LakeModel(λ=λ, α=α, b=b, d=d, A=A, R=R, g=g)
```

The default parameter values are:

- $\alpha = 0.013$ and $\lambda = 0.283$ are based on [Davis *et al.*, 2006]
- $b = 0.0124$ and $d = 0.00822$ are set to match monthly birth and death rates, respectively, in the U.S. population

As an experiment, let's create two instances, one with $\alpha = 0.013$ and another with $\alpha = 0.03$

```
model = create_lake_model()
print(f"Default α: {model.α}")
print(f"A matrix:\n{model.A}")
print(f"R matrix:\n{model.R}")
```

```
Default α: 0.013
A matrix:
[[0.7235063  0.02529314]
 [0.28067374 0.97888684]]
R matrix:
[[0.7204946  0.02518786]
 [0.27950543 0.97481215]]
```

```
model_new = create_lake_model(α=0.03)
print(f"New α: {model_new.α}")
print(f"New A matrix:\n{model_new.A}")
print(f"New R matrix:\n{model_new.R}")
```

```
New α: 0.03
New A matrix:
[[0.7235063  0.0421534 ]
 [0.28067374 0.9620266 ]]
New R matrix:
[[0.7204946  0.04197793]
 [0.27950543 0.9580221 ]]
```

106.3.2 Code for dynamics

We will also use a specialized function to generate time series in an efficient JAX-compatible manner.

Iteratively generating time series is somewhat nontrivial in JAX because arrays are immutable.

Here we use `lax.scan`, which allows the function to be jit-compiled.

Readers who prefer to skip the details can safely continue reading after the function definition.

```
@partial(jax.jit, static_argnames=['f', 'num_steps'])
def generate_path(f, initial_state, num_steps, **kwargs):
    """
    Generate a time series by repeatedly applying an update rule.

    Given a map f, initial state x_0, and model parameters, this
    function computes and returns the sequence {x_t}_{t=0}^{T-1} when

        x_{t+1} = f(x_t, **kwargs)

    Args:
        f: Update function mapping (x_t, **kwargs) -> x_{t+1}
        initial_state: Initial state x_0
        num_steps: Number of time steps T to simulate
        **kwargs: Optional extra arguments passed to f

    Returns:
        Array of shape (dim(x), T) containing the time series path
        [x_0, x_1, x_2, ..., x_{T-1}]
    """

    def update_wrapper(state, t):
        """
        Wrapper function that adapts f for use with JAX scan.
        """
        next_state = f(state, **kwargs)
        return next_state, state

    _, path = jax.lax.scan(update_wrapper,
                    initial_state, jnp.arange(num_steps))
    return path.T
```

Here are functions to update X_t and x_t.

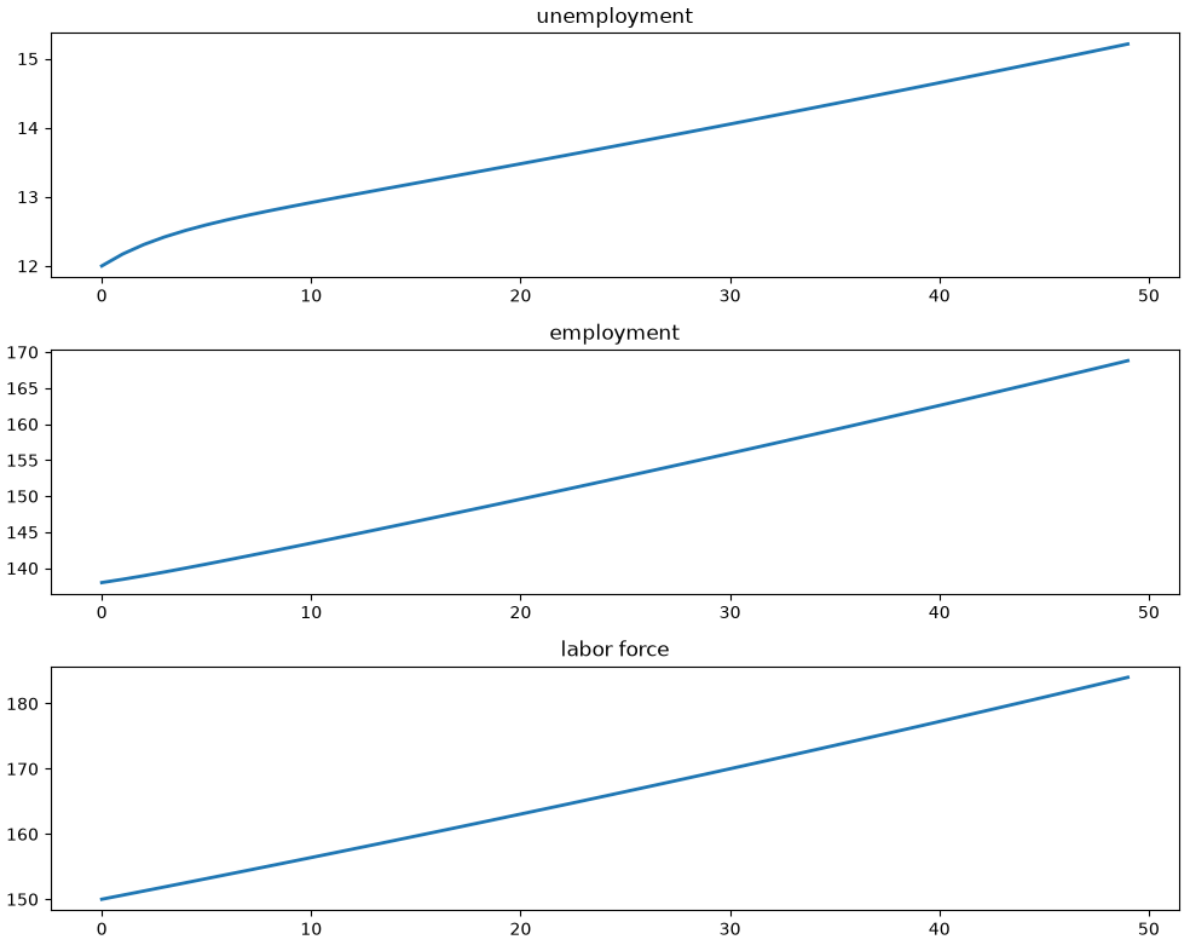
```
def stock_update(X: jnp.ndarray, model: LakeModel) -> jnp.ndarray:
    """Apply transition matrix to get next period's stocks."""
    λ, α, b, d, A, R, g = model
    return A @ X

def rate_update(x: jnp.ndarray, model: LakeModel) -> jnp.ndarray:
    """Apply normalized transition matrix for next period's rates."""
    λ, α, b, d, A, R, g = model
    return R @ x
```

106.3.3 Aggregate dynamics

Let's run a simulation under the default parameters starting from $X_0 = (12, 138)$.

We will plot the sequences $\{E_t\}$, $\{U_t\}$ and $\{N_t\}$.

```
N_0 = 150       # Population
e_0 = 0.92      # Initial employment rate
u_0 = 1 - e_0   # Initial unemployment rate
T = 50          # Simulation length

U_0 = u_0 * N_0
E_0 = e_0 * N_0

# Generate X path
X_0 = jnp.array([U_0, E_0])
X_path = generate_path(stock_update, X_0, T, model=model)

# Plot
fig, axes = plt.subplots(3, 1, figsize=(10, 8))
titles = ['unemployment', 'employment', 'labor force']
data = [X_path[0, :], X_path[1, :], X_path.sum(0)]
for ax, title, series in zip(axes, titles, data):
    ax.plot(series, lw=2)
    ax.set_title(title)
plt.tight_layout()
plt.show()
```



The aggregates E_t and U_t don't converge because their sum $E_t + U_t$ grows at rate g.

106.3.4 Rate dynamics

On the other hand, the vector of employment and unemployment rates x_t can be in a steady state $\bar{x}$ if there exists an $\bar{x}$ such that

- $\bar{x} = R\bar{x}$
- the components satisfy $\bar{e} + \bar{u} = 1$

This equation tells us that a steady state level $\bar{x}$ is an eigenvector of R associated with a unit eigenvalue.

The following function can be used to compute the steady state.

```
@jax.jit
def rate_steady_state(model: LakeModel) -> jnp.ndarray:
    r"""
    Finds the steady state of the system :math:`x_{t+1} = R x_{t}`
    by computing the eigenvector corresponding to the largest eigenvalue.

    By the Perron-Frobenius theorem, since :math:`R` is a non-negative
    matrix with columns summing to 1 (a stochastic matrix), the largest
    eigenvalue equals 1 and the corresponding eigenvector gives the steady state.
    """
```

(continues on next page)

(continued from previous page)

```
    λ, α, b, d, A, R, g = model
    eigenvals, eigenvec = jnp.linalg.eig(R)

    # Find the eigenvector corresponding to the largest eigenvalue
    # (which is 1 for a stochastic matrix by Perron-Frobenius theorem)
    max_idx = jnp.argmax(jnp.abs(eigenvals))

    # Get the corresponding eigenvector
    steady_state = jnp.real(eigenvec[:, max_idx])

    # Normalize to ensure positive values and sum to 1
    steady_state = jnp.abs(steady_state)
    steady_state = steady_state / jnp.sum(steady_state)

    return steady_state
```

We also have $x_t \to \bar{x}$ as $t \to \infty$ provided that the remaining eigenvalue of R has modulus less than 1.

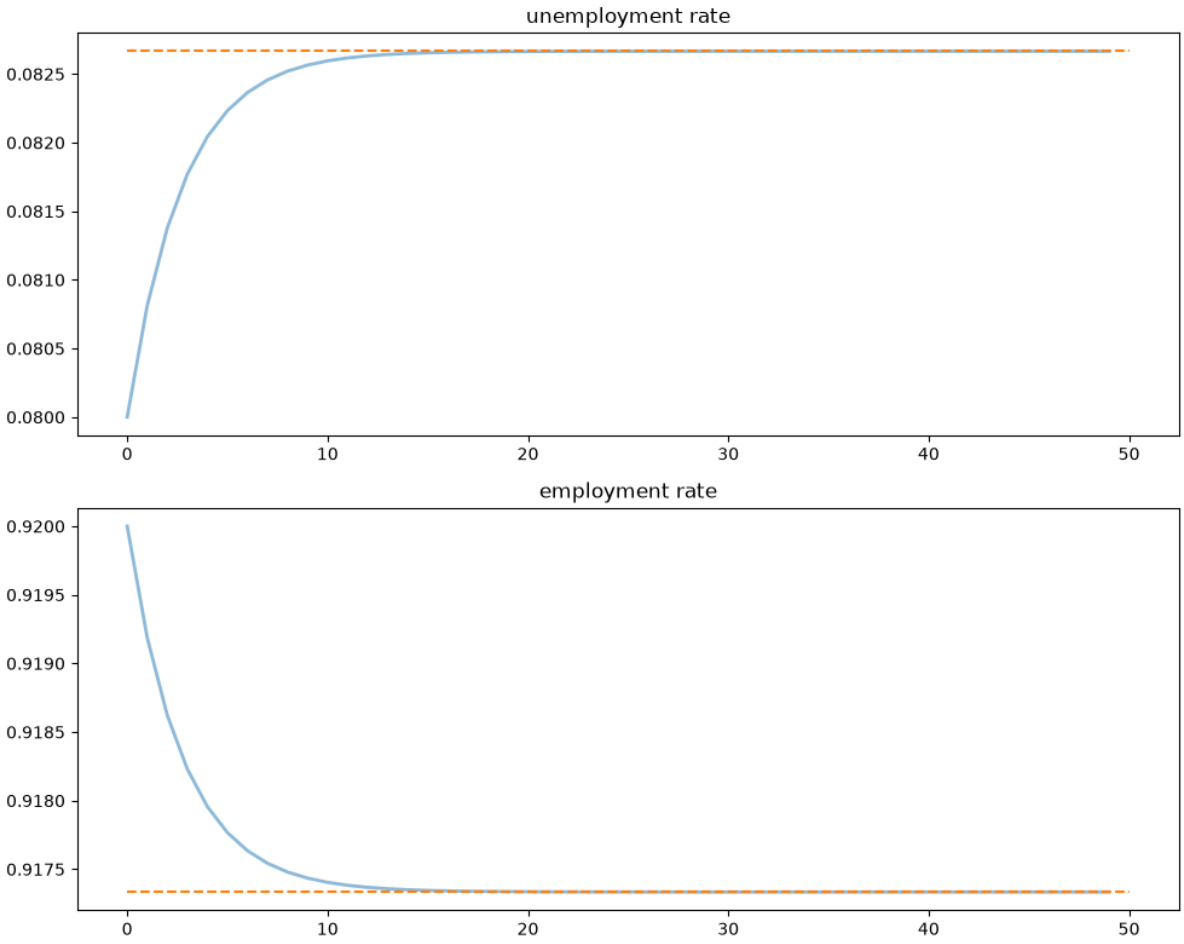
This is the case for our default parameters:

```
model = create_lake_model()
e, f = jnp.linalg.eigvals(model.R)
print(f"Eigenvalue magnitudes: {abs(e):.2f}, {abs(f):.2f}")
```

```
Eigenvalue magnitudes: 0.70, 1.00
```

Let's look at the convergence of the unemployment and employment rates to steady state levels (dashed line)

```
xbar = rate_steady_state(model)

fig, axes = plt.subplots(2, 1, figsize=(10, 8))
x_0 = jnp.array([u_0, e_0])
x_path = generate_path(rate_update, x_0, T, model=model)

titles = ['unemployment rate', 'employment rate']

for i, title in enumerate(titles):
    axes[i].plot(x_path[i, :], lw=2, alpha=0.5)
    axes[i].hlines(xbar[i], 0, T, color='C1', linestyle='--')
    axes[i].set_title(title)

plt.tight_layout()
plt.show()
```



Exercise 106.3.1

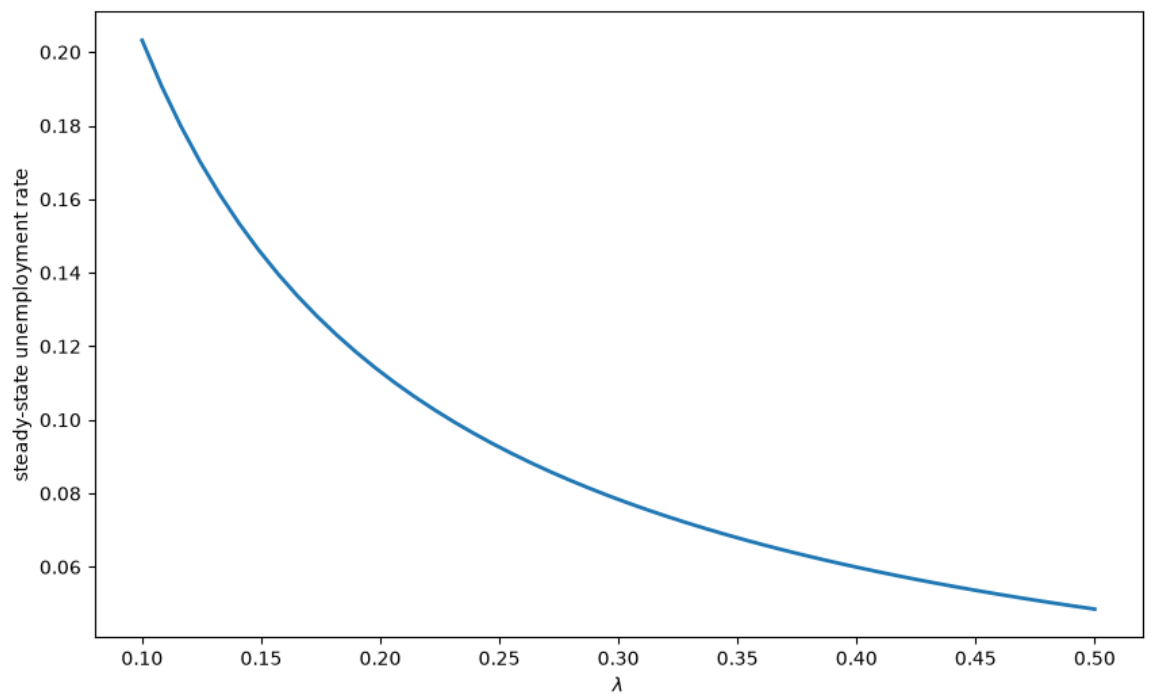
Use JAX's `vmap` to compute steady-state unemployment rates for a range of job finding rates λ (from 0.1 to 0.5), and plot the relationship.

Solution

Here is one solution

```
@jax.jit
def compute_unemployment_rate(λ_val):
    """Computes steady-state unemployment for a given λ"""
    model = create_lake_model(λ=λ_val)
    steady_state = rate_steady_state(model)
    return steady_state[0]

# Use vmap to compute for multiple λ values
λ_values = jnp.linspace(0.1, 0.5, 50)
unemployment_rates = jax.vmap(compute_unemployment_rate)(λ_values)

# Plot the results
fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(λ_values, unemployment_rates, lw=2)
```

```
ax.set_xlabel(r'$\lambda$')
ax.set_ylabel('steady-state unemployment rate')
plt.show()
```



106.4 Dynamics of an individual worker

An individual worker's employment dynamics are governed by a *finite state Markov process*.

The worker can be in one of two states:

- $s_t = 0$ means unemployed
- $s_t = 1$ means employed

Let's start off under the assumption that $b = d = 0$.

The associated transition matrix is then

$$P = \begin{pmatrix} 1 - \lambda & \lambda \\ \alpha & 1 - \alpha \end{pmatrix}$$

Let ψ_t denote the *marginal distribution* over employment/unemployment states for the worker at time t.

As usual, we regard it as a row vector.

We know *from an earlier discussion* that ψ_t follows the law of motion

$$\psi_{t+1} = \psi_t P$$

We also know from the *lecture on finite Markov chains* that if $\alpha \in (0, 1)$ and $\lambda \in (0, 1)$, then P has a unique stationary distribution, denoted here by ψ^*.

The unique stationary distribution satisfies

$$\psi^*[0] = \frac{\alpha}{\alpha + \lambda}$$

Not surprisingly, probability mass on the unemployment state increases with the dismissal rate and falls with the job finding rate.

106.4.1 Ergodicity

Let's look at a typical lifetime of employment-unemployment spells.

We want to compute the average amounts of time an infinitely lived worker would spend employed and unemployed.

Let

$$\bar{s}_{u,T} := \frac{1}{T} \sum_{t=1}^{T} \mathbb{1}\{s_t = 0\}$$

and

$$\bar{s}_{e,T} := \frac{1}{T} \sum_{t=1}^{T} \mathbb{1}\{s_t = 1\}$$

(As usual, $\mathbb{1}\{Q\} = 1$ if statement Q is true and 0 otherwise)

These are the fraction of time a worker spends unemployed and employed, respectively, up until period T.

If $\alpha \in (0, 1)$ and $\lambda \in (0, 1)$, then P is *ergodic*, and hence we have

$$\lim_{T \to \infty} \bar{s}_{u,T} = \psi^*[0] \quad \text{and} \quad \lim_{T \to \infty} \bar{s}_{e,T} = \psi^*[1]$$

with probability one.

Inspection tells us that P is exactly the transpose of R under the assumption $b = d = 0$.

Thus, the percentages of time that an infinitely lived worker spends employed and unemployed equal the fractions of workers employed and unemployed in the steady state distribution.

106.4.2 Convergence rate

How long does it take for time series sample averages to converge to cross-sectional averages?

We can investigate this by simulating the Markov chain.

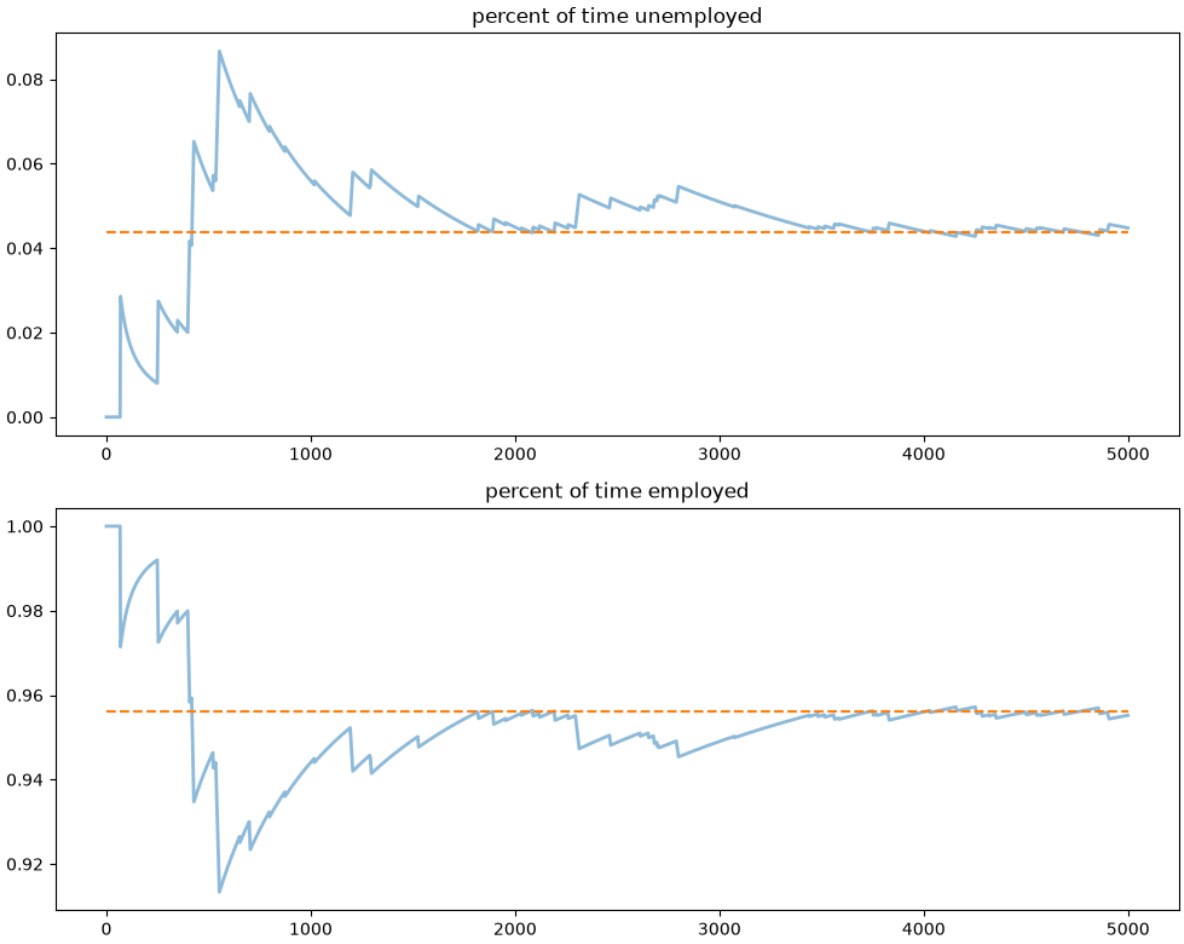
Let's plot the path of the sample averages over 5,000 periods

```
def markov_update(state, P, key):
    """
    Sample next state from transition probabilities.
    """
    probs = P[state]
    state_new = jax.random.choice(key,
                        a=jnp.arange(len(probs)),
                        p=probs)
    return state_new

model_markov = create_lake_model(d=0, b=0)
```

(continues on next page)

(continued from previous page)

```
T = 5000  # Simulation length

α, λ = model_markov.α, model_markov.λ

P = jnp.array([[1 - λ,         λ],
               [    α,     1 - α]])

xbar = rate_steady_state(model_markov)

# Simulate the Markov chain - we need a different approach for random updates
key = jax.random.PRNGKey(0)

def simulate_markov(P, initial_state, T, key):
    """Simulate Markov chain for T periods"""
    keys = jax.random.split(key, T)

    def scan_fn(state, key):
        next_state = markov_update(state, P, key)
        return next_state, state

    _, path = jax.lax.scan(scan_fn, initial_state, keys)
    return path

s_path = simulate_markov(P, 1, T, key)

fig, axes = plt.subplots(2, 1, figsize=(10, 8))
s_bar_e = jnp.cumsum(s_path) / jnp.arange(1, T+1)
s_bar_u = 1 - s_bar_e

to_plot = [s_bar_u, s_bar_e]
titles = ['percent of time unemployed', 'percent of time employed']

for i, plot in enumerate(to_plot):
    axes[i].plot(plot, lw=2, alpha=0.5)
    axes[i].hlines(xbar[i], 0, T, color='C1', linestyle='--')
    axes[i].set_title(titles[i])

plt.tight_layout()
plt.show()
```



The stationary probabilities are given by the dashed line.

In this case it takes much of the sample for these two objects to converge.

This is largely due to the high persistence in the Markov chain.

106.5 Exercises

Exercise 106.5.1

Consider an economy with an initial stock of workers $N_0 = 100$ at the steady state level of employment in the baseline parameterization.

Suppose that in response to new legislation the hiring rate reduces to $\lambda = 0.2$.

Plot the transition dynamics of the unemployment and employment stocks for 50 periods.

Plot the transition dynamics for the rates.

How long does the economy take to converge to its new steady state?

What is the new steady state level of employment?

Solution

We begin by constructing the model with default parameters and finding the initial steady state

```
model_initial = create_lake_model()
x0 = rate_steady_state(model_initial)
print(f"Initial Steady State: {x0}")
```

```
Initial Steady State: [0.08266623 0.9173338 ]
```

Initialize the simulation values

```
N0 = 100
T = 50
```

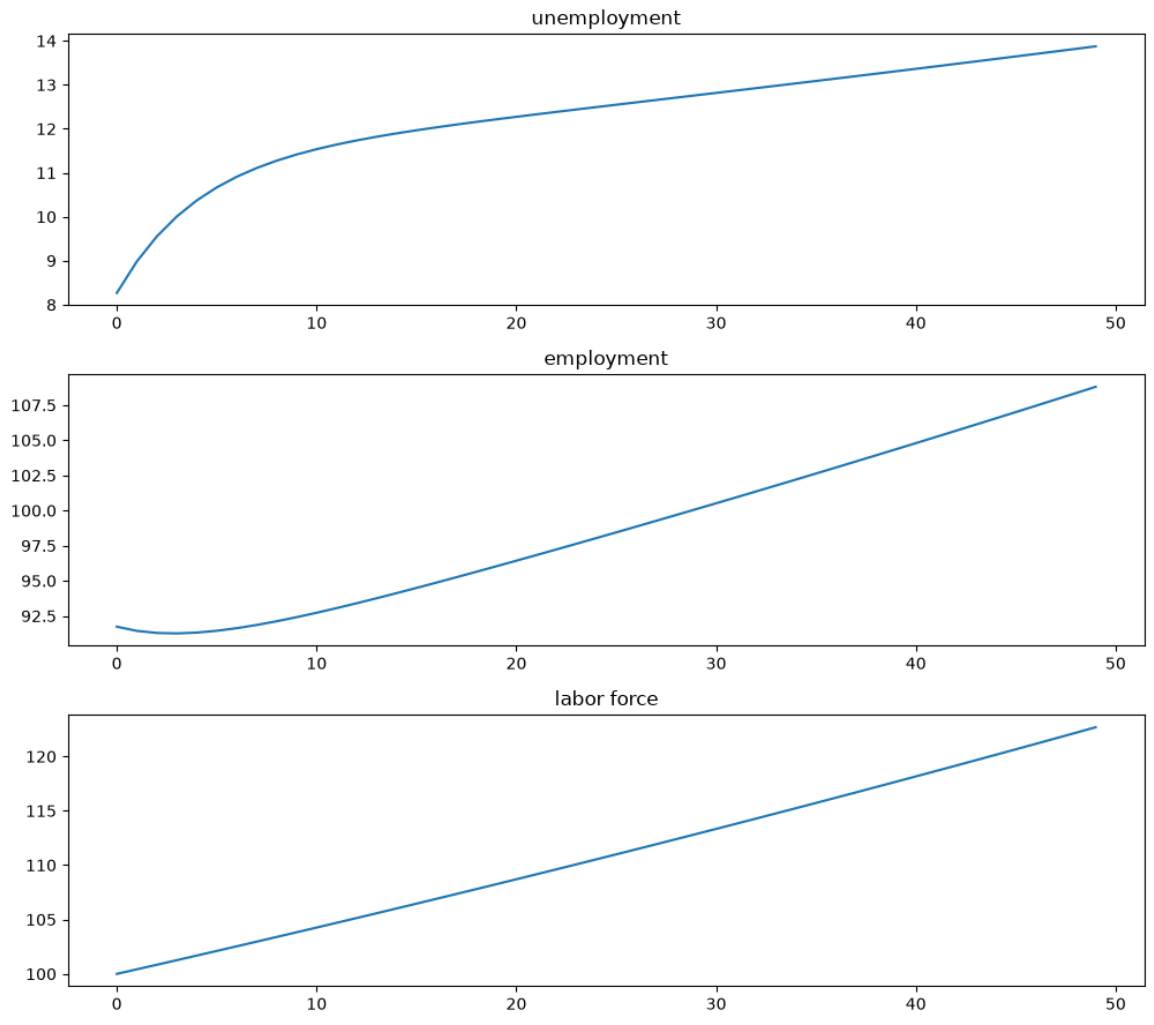
New legislation changes λ to 0.2

```
model_ex2 = create_lake_model(λ=0.2)
xbar = rate_steady_state(model_ex2)  # new steady state

# Simulate paths
X_path = generate_path(stock_update, x0 * N0, T, model=model_ex2)
x_path = generate_path(rate_update, x0, T, model=model_ex2)
print(f"New Steady State: {xbar}")
```

```
New Steady State: [0.11309288 0.8869071 ]
```

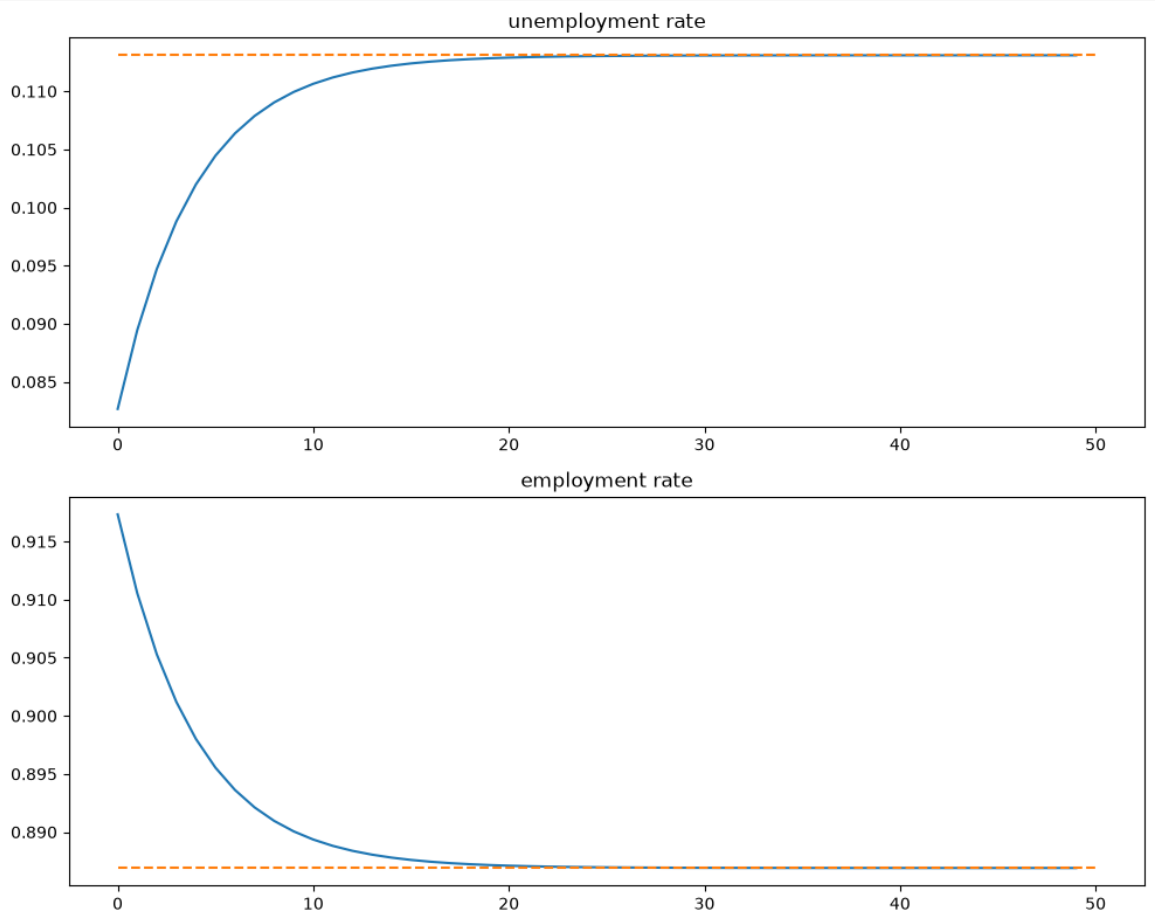
Now plot stocks

```
fig, axes = plt.subplots(3, 1, figsize=[10, 9])

axes[0].plot(X_path[0, :])
axes[0].set_title('unemployment')

axes[1].plot(X_path[1, :])
axes[1].set_title('employment')

axes[2].plot(X_path.sum(0))
axes[2].set_title('labor force')

plt.tight_layout()
plt.show()
```



And how the rates evolve

```
fig, axes = plt.subplots(2, 1, figsize=(10, 8))

titles = ['unemployment rate', 'employment rate']

for i, title in enumerate(titles):
    axes[i].plot(x_path[i, :])
    axes[i].hlines(xbar[i], 0, T, color='C1', linestyle='--')
    axes[i].set_title(title)

plt.tight_layout()
plt.show()
```



We see that it takes 20 periods for the economy to converge to its new steady state levels.

Exercise 106.5.2

Consider an economy with an initial stock of workers $N_0 = 100$ at the steady state level of employment in the baseline parameterization.

Suppose that for 20 periods the birth rate was temporarily high ($b = 0.025$) and then returned to its original level.

Plot the transition dynamics of the unemployment and employment stocks for 50 periods.

Plot the transition dynamics for the rates.

How long does the economy take to return to its original steady state?

Solution

This exercise has the economy experiencing a boom in entrances to the labor market and then later returning to the original levels.

For 20 periods the economy has a new entry rate into the labor market.

Let's start off at the baseline parameterization and record the steady state

```
model_baseline = create_lake_model()
x0 = rate_steady_state(model_baseline)
N0 = 100
T = 50
```

Here are the other parameters:

```
b_hat = 0.025
T_hat = 20
```

Let's increase b to the new value and simulate for 20 periods

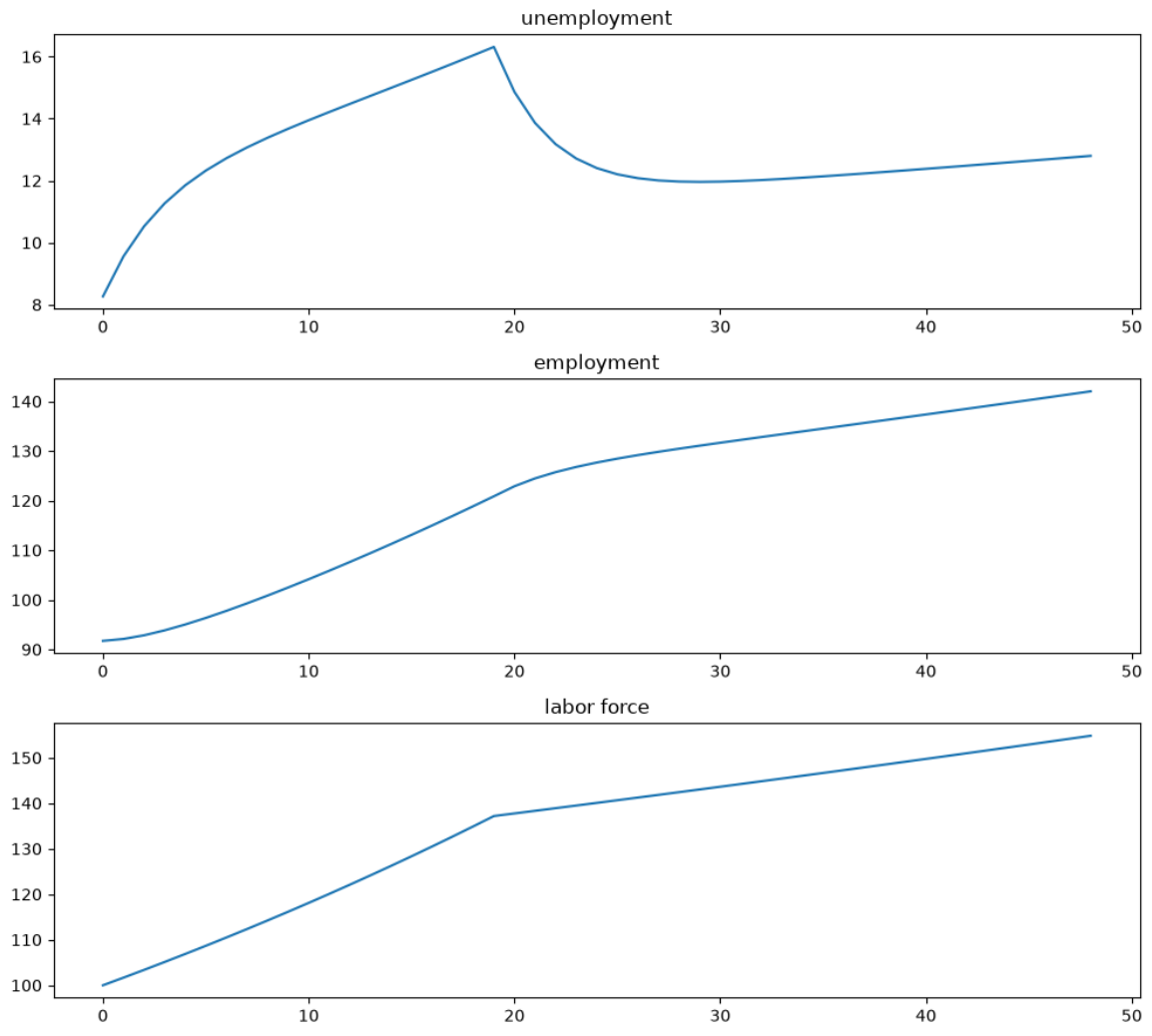
```
model_high_b = create_lake_model(b=b_hat)

# Simulate stocks and rates for first 20 periods
X_path1 = generate_path(stock_update, x0 * N0, T_hat, model=model_high_b)
x_path1 = generate_path(rate_update, x0, T_hat, model=model_high_b)
```

Now we reset b to the original value and then, using the state after 20 periods for the new initial conditions, we simulate for the additional 30 periods

```
# Use final state from period 20 as initial condition
X_path2 = generate_path(stock_update, X_path1[:, -1], T-T_hat,
                        model=model_baseline)
x_path2 = generate_path(rate_update, x_path1[:, -1], T-T_hat,
                        model=model_baseline)
```

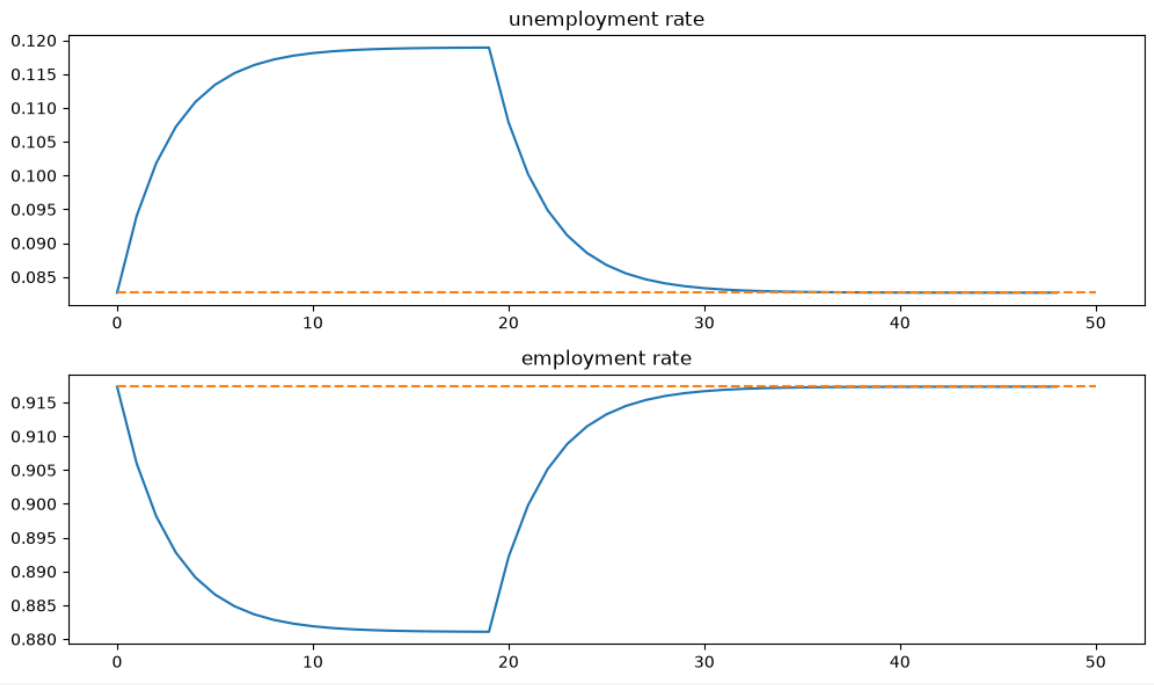
Finally, we combine these two paths and plot

```
# Combine paths
X_path = jnp.hstack([X_path1, X_path2[:, 1:]])
x_path = jnp.hstack([x_path1, x_path2[:, 1:]])

fig, axes = plt.subplots(3, 1, figsize=[10, 9])

axes[0].plot(X_path[0, :])
axes[0].set_title('unemployment')

axes[1].plot(X_path[1, :])
axes[1].set_title('employment')

axes[2].plot(X_path.sum(0))
axes[2].set_title('labor force')

plt.tight_layout()
plt.show()
```



And the rates

```
fig, axes = plt.subplots(2, 1, figsize=[10, 6])

titles = ['unemployment rate', 'employment rate']

for i, title in enumerate(titles):
    axes[i].plot(x_path[i, :])
    axes[i].hlines(x0[i], 0, T, color='C1', linestyle='--')
    axes[i].set_title(title)

plt.tight_layout()
plt.show()
```


unemployment rate
0.120
0.115
0.110
0.105
0.100
0.095
0.090
0.085
0
10
20
30
40
50
employment rate
0.915
0.910
0.905
0.900
0.895
0.890
0.885
0.880
0
10
20
30
40
50

CHAPTER

SEVEN

LAKE MODEL WITH AN ENDOGENOUS JOB FINDING RATE

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

Contents

- *Lake Model with an Endogenous Job Finding Rate*
  - *Overview*
  - *Set Up*
  - *Fiscal policy*
  - *Exercises*

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install quantecon jax
```

107.1 Overview

This lecture is a continuation of the *lake model lecture*.

We recommend you read that lecture first before proceeding with this one.

In the previous lecture, we studied a lake model of unemployment and employment where the transition rates between states were exogenous parameters.

In this lecture, we extend the model by making the job finding rate endogenous.

Specifically, the transition rate from unemployment to employment will be determined by the McCall search model [McCall, 1970].

Let's start with some imports:

```
import matplotlib.pyplot as plt
import jax
import jax.numpy as jnp
from typing import NamedTuple
from quantecon.distributions import BetaBinomial
from functools import partial
import jax.scipy.stats as stats
```

107.2 Set Up

The basic structure of the model will be as discussed in the *lake model lecture*.

The only difference is that the hiring rate is endogenous, determined by the decisions of optimizing agents inhabiting a McCall search model [McCall, 1970] with IID wage offers and job separation at rate α.

107.2.1 Reservation wage

In the model, the optimal policy is characterized by a reservation wage $\bar{w}$

- If the wage offer w in hand is greater than or equal to $\bar{w}$, then the worker accepts.
- Otherwise, the worker rejects.

The reservation wage depends on the wage offer distribution and the parameters

- α, the separation rate
- β, the discount factor
- γ, the offer arrival rate
- c, unemployment compensation

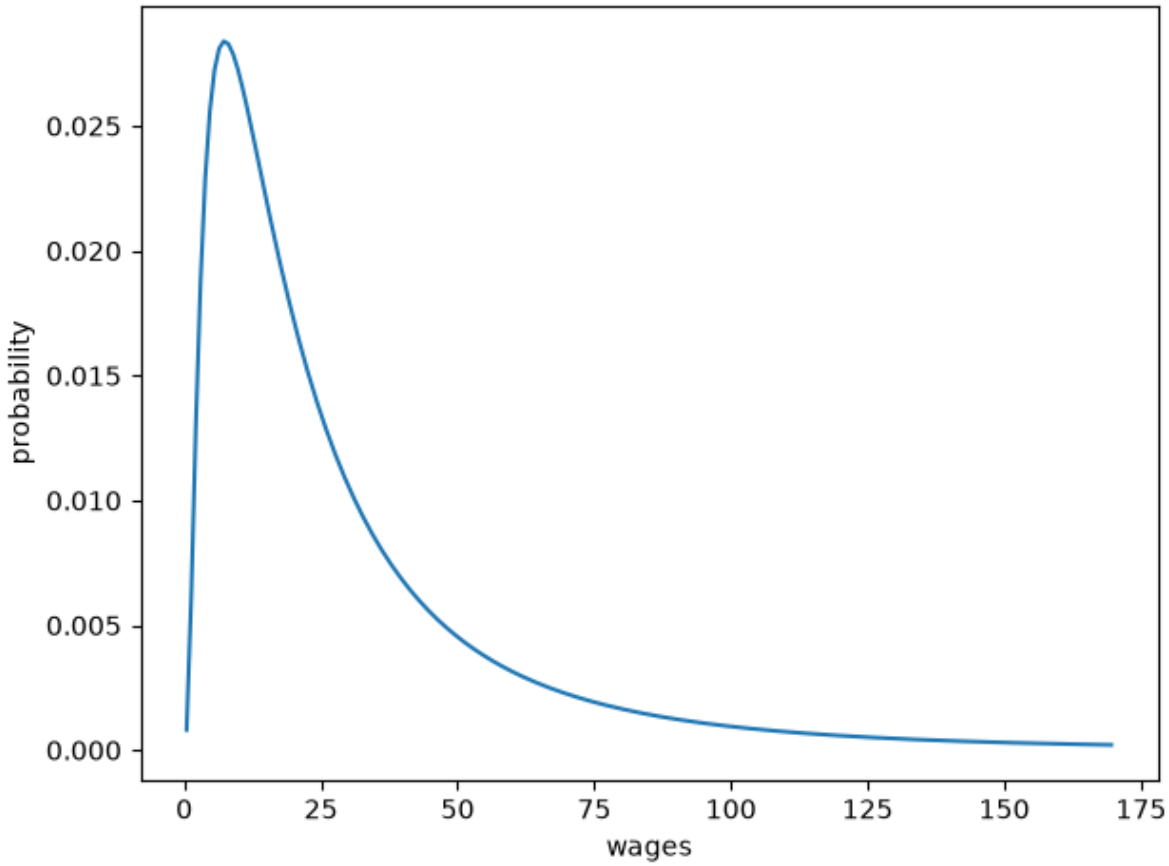
The wage offer distribution will be a discretized version of a lognormal distribution.

We first define a function to create such a discrete distribution.

```
def create_wage_distribution(
        max_wage: float,
        wage_grid_size: int,
        log_wage_mean: float
    ):
    """
    Creates a discretized version of a lognormal density LN(log(m),1), where
    m is log_wage_mean.

    """
    w_vec_temp = jnp.linspace(1e-8, max_wage, wage_grid_size + 1)
    cdf = stats.norm.cdf(
        jnp.log(w_vec_temp), loc=jnp.log(log_wage_mean), scale=1
    )
    pdf = cdf[1:] - cdf[:-1]
    p_vec = pdf / pdf.sum()
    w_vec = (w_vec_temp[1:] + w_vec_temp[:-1]) / 2
    return w_vec, p_vec
```

The cell below creates a discretized $LN(\log(20), 1)$ wage distribution and plots it.

```
w_vec, p_vec = create_wage_distribution(170, 200, 20)

fig, ax = plt.subplots()
ax.plot(w_vec, p_vec)
ax.set_xlabel('wages')
ax.set_ylabel('probability')
plt.tight_layout()
plt.show()
```



Now we organize the code for solving the McCall model, given a set of parameters.

For background on the model and our solution method, see the *lecture on the McCall model with separation*

Our first step is to define the utility function and the McCall model data structure.

```
def u(c, σ=2.0):
    return jnp.where(c > 0, (c**(1 - σ) - 1) / (1 - σ), -10e6)


class McCallModel(NamedTuple):
    """
    Stores the parameters for the McCall search model
    """
    α: float            # Job separation rate
    β: float            # Discount rate
    γ: float            # Job offer rate
```

(continues on next page)

(continued from previous page)

```
    c: float            # Unemployment compensation
    σ: float            # Utility parameter
    w_vec: jnp.ndarray  # Possible wage values
    p_vec: jnp.ndarray  # Probabilities over w_vec


def create_mccall_model(
        α=0.2, β=0.98, γ=0.7, c=6.0, σ=2.0,
        w_vec=None,
        p_vec=None
    ) -> McCallModel:
    if w_vec is None:
        n = 60  # Number of possible outcomes for wage
        # Wages between 10 and 20
        w_vec = jnp.linspace(10, 20, n)
        a, b = 600, 400  # Shape parameters
        dist = BetaBinomial(n-1, a, b)
        p_vec = jnp.array(dist.pdf())
    return McCallModel(
        α=α, β=β, γ=γ, c=c, σ=σ, w_vec=w_vec, p_vec=p_vec
    )
```

Next, we implement the Bellman operator

```
def T(mcm: McCallModel, V, U):
    """
    Update the Bellman equations.
    """
    α, β, γ, c, σ = mcm.α, mcm.β, mcm.γ, mcm.c, mcm.σ
    w_vec, p_vec = mcm.w_vec, mcm.p_vec

    V_new = u(w_vec, σ) + β * ((1 - α) * V + α * U)
    U_new = u(c, σ) + β * (1 - γ) * U + β * γ * (jnp.maximum(U, V) @ p_vec)

    return V_new, U_new
```

Now we define the value function iteration solver.

We'll use a compiled while loop for extra speed.

```
@jax.jit
def solve_mccall_model(mcm: McCallModel, tol=1e-5, max_iter=2000):
    """
    Iterates to convergence on the Bellman equations.
    """
    def cond(state):
        V, U, i, error = state
        return jnp.logical_and(error > tol, i < max_iter)

    def update(state):
        V, U, i, error = state
        V_new, U_new = T(mcm, V, U)
        error_1 = jnp.max(jnp.abs(V_new - V))
        error_2 = jnp.abs(U_new - U)
        error_new = jnp.maximum(error_1, error_2)
        return V_new, U_new, i + 1, error_new
```

(continues on next page)

(continued from previous page)

```
    # Initial state
    V_init = jnp.ones(len(mcm.w_vec))
    U_init = 1.0
    i_init = 0
    error_init = tol + 1

    init_state = (V_init, U_init, i_init, error_init)
    V_final, U_final, _, _ = jax.lax.while_loop(
        cond, update, init_state
    )
    return V_final, U_final
```

107.2.2 Lake model code

We also need the lake model functions from the previous lecture to compute steady state unemployment rates:

```
class LakeModel(NamedTuple):
    """
    Parameters for the lake model
    """
    λ: float
    α: float
    b: float
    d: float
    A: jnp.ndarray
    R: jnp.ndarray
    g: float


def create_lake_model(
        λ: float = 0.283,      # job finding rate
        α: float = 0.013,      # separation rate
        b: float = 0.0124,     # birth rate
        d: float = 0.00822     # death rate
    ) -> LakeModel:
    """
    Create a LakeModel instance with default parameters.

    Computes and stores the transition matrices A and R,
    and the labor force growth rate g.

    """
    # Compute growth rate
    g = b - d

    # Compute transition matrix A
    A = jnp.array([
        [(1-d) * (1-λ) + b, (1-d) * α + b],
        [(1-d) * λ,         (1-d) * (1-α)]
    ])

    # Compute normalized transition matrix R
    R = A / (1 + g)

    return LakeModel(λ=λ, α=α, b=b, d=d, A=A, R=R, g=g)
```

(continues on next page)

(continued from previous page)

```
@jax.jit
def rate_steady_state(model: LakeModel) -> jnp.ndarray:
    r"""
    Finds the steady state of the system :math:`x_{t+1} = R x_{t}`
    by computing the eigenvector corresponding to the largest eigenvalue.

    By the Perron-Frobenius theorem, since :math:`R` is a non-negative
    matrix with columns summing to 1 (a stochastic matrix), the largest
    eigenvalue equals 1 and the corresponding eigenvector gives the steady state.
    """
    λ, α, b, d, A, R, g = model
    eigenvals, eigenvec = jnp.linalg.eig(R)

    # Find the eigenvector corresponding to the largest eigenvalue
    # (which is 1 for a stochastic matrix by Perron-Frobenius theorem)
    max_idx = jnp.argmax(jnp.abs(eigenvals))

    # Get the corresponding eigenvector
    steady_state = jnp.real(eigenvec[:, max_idx])

    # Normalize to ensure positive values and sum to 1
    steady_state = jnp.abs(steady_state)
    steady_state = steady_state / jnp.sum(steady_state)

    return steady_state
```

107.2.3 Linking the McCall search model to the lake model

Suppose that all workers inside a lake model behave according to the McCall search model.

The exogenous probability of leaving employment remains α.

But their optimal decision rules determine the probability λ of leaving unemployment.

This is now

$$\lambda = \gamma \mathbb{P}\{w_t \geq \bar{w}\} = \gamma \sum_{w' \geq \bar{w}} p(w') \tag{107.1}$$

Here

- $\bar{w}$ is the reservation wage determined by the parameters and
- p is the wage offer distribution.

Wage offers across the population of workers are independent draws from p.

Here we calculate λ at the default parameters:

```
mcm = create_mccall_model(w_vec=w_vec, p_vec=p_vec)
V, U = solve_mccall_model(mcm)
w_idx = jnp.searchsorted(V - U, 0)
w_bar = jnp.where(w_idx == len(V), jnp.inf, mcm.w_vec[w_idx])
λ = mcm.γ * jnp.sum(p_vec * (w_vec > w_bar))
print(f"Job finding rate at default paramters ={λ}.")
```

```
Job finding rate at default paramters =0.4510621726512909.
```

107.3 Fiscal policy

In this section, we will put the lake model to work, examining outcomes associated with different levels of unemployment compensation.

Our aim is to find an optimal level of unemployment insurance.

We assume that the government sets unemployment compensation c.

The government imposes a lump-sum tax τ sufficient to finance total unemployment payments.

To attain a balanced budget at a steady state, taxes, the steady state unemployment rate u, and the unemployment compensation rate must satisfy

$$\tau = uc$$

The lump-sum tax applies to everyone, including unemployed workers.

- The post-tax income of an employed worker with wage w is $w - \tau$.
- The post-tax income of an unemployed worker is $c - \tau$.

For each specification (c, τ) of government policy, we can solve for the worker's optimal reservation wage.

This determines λ via (107.1) evaluated at post tax wages, which in turn determines a steady state unemployment rate $u(c, \tau)$.

For a given level of unemployment benefit c, we can solve for a tax that balances the budget in the steady state

$$\tau = u(c, \tau)c$$

To evaluate alternative government tax-unemployment compensation pairs, we require a welfare criterion.

We use a steady state welfare criterion

$$W := e\,\mathbb{E}[V \,|\, \text{employed}] + u\,U$$

where the notation V and U is as defined above and the expectation is at the steady state.

107.3.1 Computing optimal unemployment insurance

Now we set up the infrastructure to compute optimal unemployment insurance levels.

First, we define a container for the economy's parameters:

```
class Economy(NamedTuple):
    """Parameters for the economy"""
    α: float
    b: float
    d: float
    β: float
    γ: float
    σ: float
    log_wage_mean: float
    wage_grid_size: int
```

(continues on next page)

(continued from previous page)

```
    max_wage: float

def create_economy(
        α=0.013,
        b=0.0124,
        d=0.00822,
        β=0.98,
        γ=1.0,
        σ=2.0,
        log_wage_mean=20,
        wage_grid_size=200,
        max_wage=170
    ) -> Economy:
    """
    Create an economy with a set of default values"""
    return Economy(α=α, b=b, d=d, β=β, γ=γ, σ=σ,
                           log_wage_mean=log_wage_mean,
                           wage_grid_size=wage_grid_size,
                           max_wage=max_wage)
```

Next, we define a function that computes optimal worker behavior given policy parameters:

```
@jax.jit
def compute_optimal_quantities(
        c: float,
        τ: float,
        economy: Economy,
        w_vec: jnp.array,
        p_vec: jnp.array
    ):
    """
    Compute the reservation wage, job finding rate and value functions
    of the workers given c and τ.

    """
    mcm = create_mccall_model(
        α=economy.α,
        β=economy.β,
        γ=economy.γ,
        c=c-τ,          # Post tax compensation
        σ=economy.σ,
        w_vec=w_vec-τ,  # Post tax wages
        p_vec=p_vec
    )

    # Compute reservation wage under given parameters
    V, U = solve_mccall_model(mcm)
    w_idx = jnp.searchsorted(V - U, 0)
    w_bar = jnp.where(w_idx == len(V), jnp.inf, mcm.w_vec[w_idx])

    # Compute job finding rate
    λ = economy.γ * jnp.sum(p_vec * (w_vec - τ > w_bar))

    return w_bar, λ, V, U
```

This function computes the steady state outcomes given unemployment insurance and tax levels:

```
@jax.jit
def compute_steady_state_quantities(
        c, τ, economy: Economy, w_vec, p_vec
    ):
    """
    Compute the steady state unemployment rate given c and τ using optimal
    quantities from the McCall model and computing corresponding steady
    state quantities

    """

    # Find optimal values and policies by solving the McCall model, as well
    # as the corresponding job finding rate.
    w_bar, λ, V, U = compute_optimal_quantities(c, τ, economy, w_vec, p_vec)

    # Set up a lake model using the given parameters and the job finding rate.
    model = create_lake_model(λ=λ, α=economy.α, b=economy.b, d=economy.d)

    # Compute steady state employment and unemployment rates from this lake
    # model.
    u, e = rate_steady_state(model)

    # Compute expected lifetime value conditional on being employed.
    mask = (w_vec - τ > w_bar)
    w = jnp.sum(V * p_vec * mask) / jnp.sum(p_vec * mask)
    # Compute steady state welfare.
    welfare = e * w + u * U

    return e, u, welfare
```

We need a function to find the tax rate that balances the government budget:

```
def find_balanced_budget_tax(c, economy: Economy, w_vec, p_vec):
    """
    Find the tax rate that will induce a balanced budget given unemployment
    compensation c.

    """

    def steady_state_budget(t):
        """
        For given tax rate t, compute the budget surplus.

        """
        e, u, w = compute_steady_state_quantities(c, t, economy, w_vec, p_vec)
        return t - u * c

    # Use a simple bisection method to find the tax rate that balances the
    # budget (but setting the surplus to zero

    t_low, t_high = 0.0, 0.9 * c
    tol = 1e-6
    max_iter = 100
    for i in range(max_iter):
        t_mid = (t_low + t_high) / 2
        budget = steady_state_budget(t_mid)
        if abs(budget) < tol:
```

(continues on next page)

(continued from previous page)

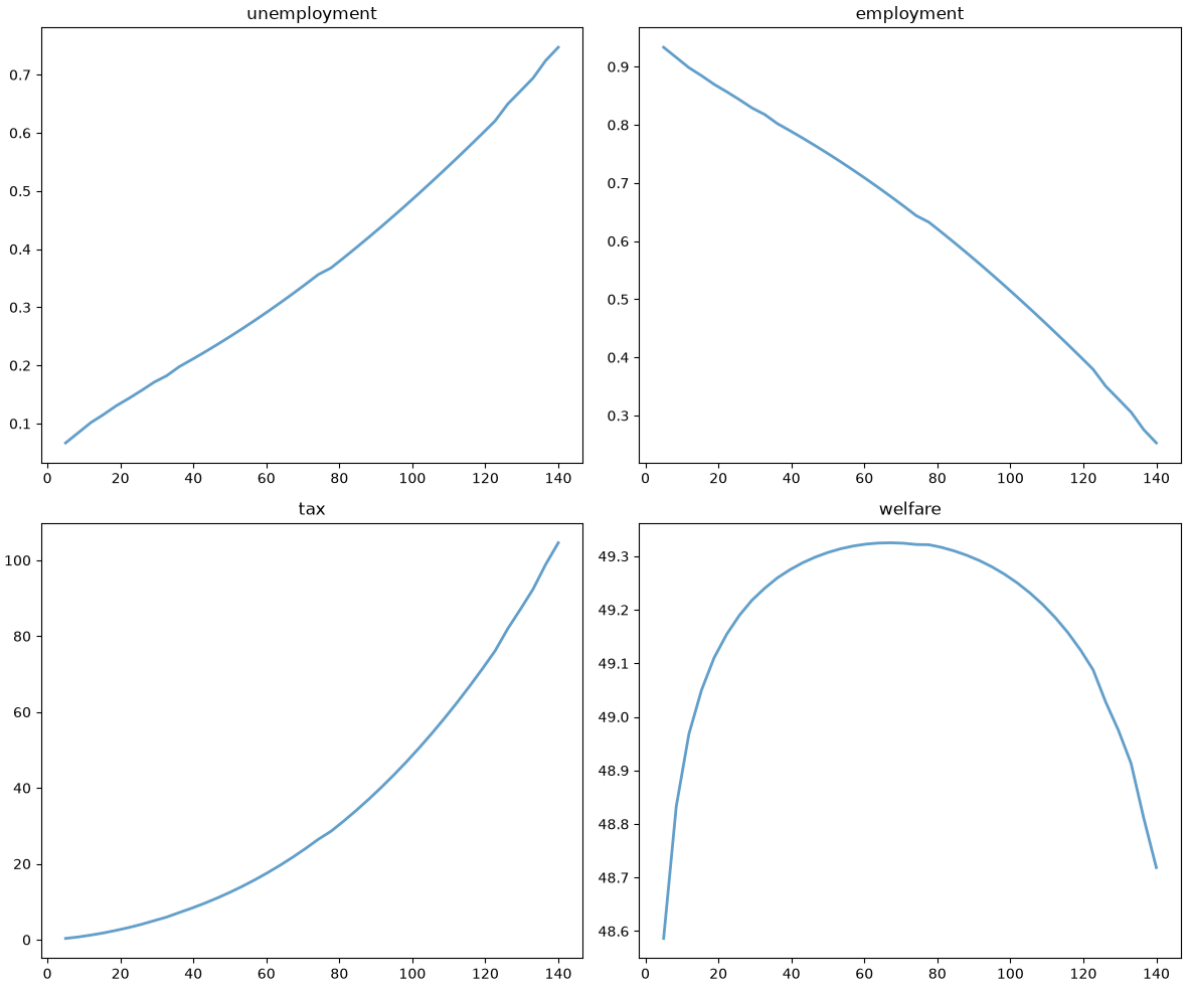
```
            return t_mid
        elif budget < 0:
            t_low = t_mid
        else:
            t_high = t_mid

    return t_mid
```

Now we compute how employment, unemployment, taxes, and welfare vary with the unemployment compensation rate:

```
# Create economy and wage distribution
economy = create_economy()
w_vec, p_vec = create_wage_distribution(
    economy.max_wage, economy.wage_grid_size, economy.log_wage_mean
)

# Levels of unemployment insurance we wish to study
c_vec = jnp.linspace(5, 140, 40)

tax_vec = []
unempl_vec = []
empl_vec = []
welfare_vec = []

for c in c_vec:
    t = find_balanced_budget_tax(c, economy, w_vec, p_vec)
    e_rate, u_rate, welfare = compute_steady_state_quantities(
        c, t, economy, w_vec, p_vec
    )
    tax_vec.append(t)
    unempl_vec.append(u_rate)
    empl_vec.append(e_rate)
    welfare_vec.append(welfare)
```

Let's visualize the results:

```
fig, axes = plt.subplots(2, 2, figsize=(12, 10))

plots = [unempl_vec, empl_vec, tax_vec, welfare_vec]
titles = ['unemployment', 'employment', 'tax', 'welfare']

for ax, plot, title in zip(axes.flatten(), plots, titles):
    ax.plot(c_vec, plot, lw=2, alpha=0.7)
    ax.set_title(title)

plt.tight_layout()
plt.show()
```



Welfare first increases and then decreases as unemployment benefits rise.

The level that maximizes steady state welfare is approximately 62.

107.4 Exercises

Exercise 107.4.1

How does the welfare-maximizing level of unemployment compensation c change with the job separation rate α?

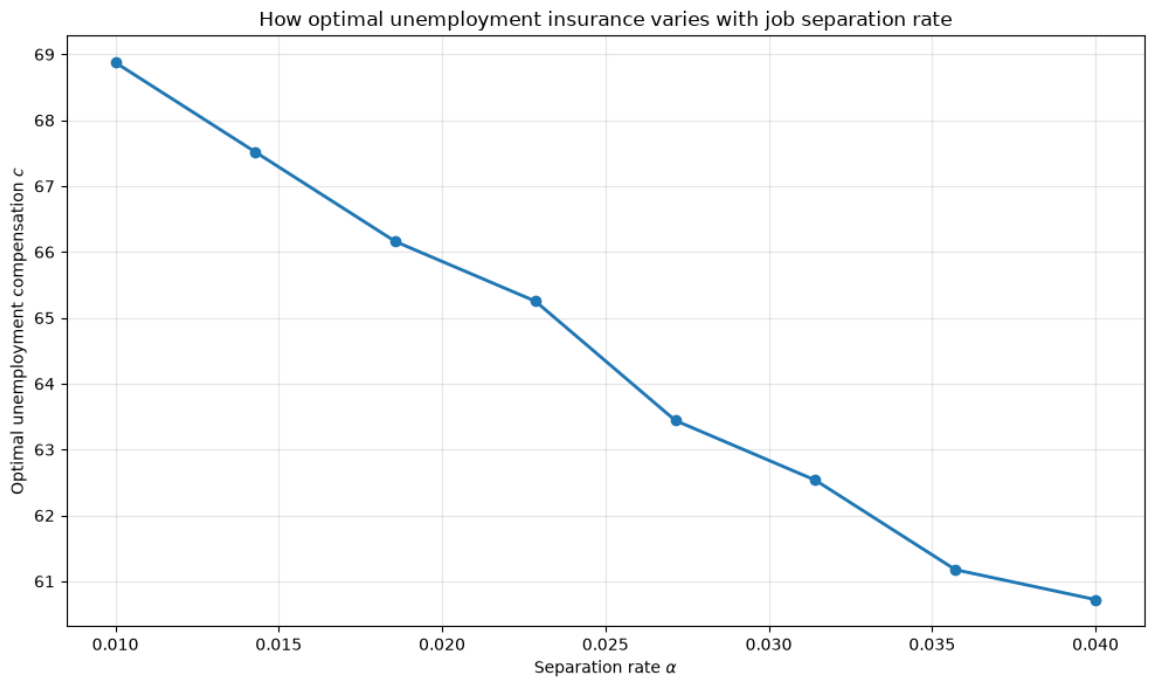
Compute and plot the optimal c (the value that maximizes welfare) for a range of separation rates α from 0.01 to 0.04.

For each α value, find the optimal c by computing welfare across the range of c values and selecting the maximum.

Solution

Here is one solution:

```
# Range of separation rates to explore (wider range, fewer points)
α_values = jnp.linspace(0.01, 0.04, 8)

# We'll store the optimal c for each α
optimal_c_values = []

# Use a finer grid for c values to get better resolution
c_vec_fine = jnp.linspace(5, 140, 150)

for α_val in α_values:
    # Create economy parameters with this α
    params_α = create_economy(α=α_val)

    # Create wage distribution
    w_vec_α, p_vec_α = create_wage_distribution(
        params_α.max_wage, params_α.wage_grid_size, params_α.log_wage_mean
    )

    # Compute welfare for each c value
    welfare_values = []
    for c in c_vec_fine:
        t = find_balanced_budget_tax(c, params_α, w_vec_α, p_vec_α)
        e_rate, u_rate, welfare = compute_steady_state_quantities(
            c, t, params_α, w_vec_α, p_vec_α
        )
        welfare_values.append(welfare)

    # The welfare function is very flat near its maximum.
    # Using argmax on a single point can be unstable due to numerical noise.
    # Instead, we find all c values within 99.9% of maximum welfare and
    # compute their weighted average (centroid). This gives a more stable
    # estimate of the optimal unemployment compensation level.
    welfare_array = jnp.array(welfare_values)
    max_welfare = jnp.max(welfare_array)
    threshold = 0.999 * max_welfare
    near_optimal_mask = welfare_array >= threshold

    # Compute weighted average of c values in the near-optimal region
    optimal_c = jnp.sum(c_vec_fine * near_optimal_mask * welfare_array) / \
                jnp.sum(near_optimal_mask * welfare_array)
    optimal_c_values.append(optimal_c)

# Plot the relationship
fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(α_values, optimal_c_values, lw=2, marker='o')
ax.set_xlabel(r'Separation rate $\alpha$')
ax.set_ylabel('Optimal unemployment compensation $c$')
ax.set_title('How optimal unemployment insurance varies with job separation rate')
ax.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()
```



We see that as the separation rate increases (workers lose their jobs more frequently), the welfare-maximizing level of unemployment compensation decreases.

This occurs because higher separation rates increase steady-state unemployment, which raises the tax burden needed to finance unemployment benefits. The optimal policy balances insurance against distortionary taxation.

CHAPTER

EIGHT

RATIONAL EXPECTATIONS EQUILIBRIUM

Contents

- *Rational Expectations Equilibrium*
  - *Overview*
  - *Rational Expectations Equilibrium*
  - *Computing an Equilibrium*
  - *Exercises*

"If you're so smart, why aren't you rich?"

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install quantecon
```

108.1 Overview

This lecture introduces the concept of a *rational expectations equilibrium*.

To illustrate it, we describe a linear quadratic version of a model due to Lucas and Prescott [Lucas and Prescott, 1971].

That 1971 paper is one of a small number of research articles that ignited a *rational expectations revolution*.

We follow Lucas and Prescott by employing a setting that is readily "Bellmanized" (i.e., susceptible to being formulated as a dynamic programming problems.

Because we use linear quadratic setups for demand and costs, we can deploy the LQ programming techniques described in *this lecture*.

We will learn about how a representative agent's problem differs from a planner's, and how a planning problem can be used to compute quantities and prices in a rational expectations equilibrium.

We will also learn about how a rational expectations equilibrium can be characterized as a fixed point of a mapping from a *perceived law of motion* to an *actual law of motion*.

Equality between a perceived and an actual law of motion for endogenous market-wide objects captures in a nutshell what the rational expectations equilibrium concept is all about.

Finally, we will learn about the important "Big K, little k" trick, a modeling device widely used in macroeconomics.

Except that for us

- Instead of "Big K" it will be "Big Y".
- Instead of "little k" it will be "little y".

Let's start with some standard imports:

```
import matplotlib.pyplot as plt
import numpy as np
```

We'll also use the LQ class from `QuantEcon.py`.

```
from quantecon import LQ
```

108.1.1 The Big Y, little y Trick

This widely used method applies in contexts in which a **representative firm** or agent is a "price taker" operating within a competitive equilibrium.

The following setting justifies the concept of a representative firm that stands in for a large number of other firms too.

There is a uniform unit measure of identical firms named $\omega \in \Omega = [0, 1]$.

The output of firm ω is $y(\omega)$.

The output of all firms is $Y = \int_0^1 y(\omega) d\,\omega$.

All firms end up choosing to produce the same output, so that at the end of the day $y(\omega) = y$ and $Y = y = \int_0^1 y(\omega) d\,\omega$.

This setting allows us to speak of a representative firm that chooses to produce y.

We want to impose that

- The representative firm or individual firm takes *aggregate* Y as given when it chooses individual $y(\omega)$, but ....
- At the end of the day, $Y = y(\omega) = y$, so that the representative firm is indeed representative.

The Big Y, little y trick accomplishes these two goals by

- Taking Y as beyond control when posing the choice problem of who chooses y; but ....
- Imposing $Y = y$ *after* having solved the individual's optimization problem.

Please watch for how this strategy is applied as the lecture unfolds.

We begin by applying the Big Y, little y trick in a very simple static context.

A Simple Static Example of the Big Y, little y Trick

Consider a static model in which a unit measure of firms produce a homogeneous good that is sold in a competitive market.

Each of these firms ends up producing and selling output $y(\omega) = y$.

The price p of the good lies on an inverse demand curve

$$p = a_0 - a_1 Y \tag{108.1}$$

where

- $a_i > 0$ for $i = 0, 1$
- $Y = \int_0^1 y(\omega) d\omega$ is the market-wide level of output

For convenience, we'll often just write y instead of $y(\omega)$ when we are describing the choice problem of an individual firm $\omega \in \Omega$.

Each firm has a total cost function

$$c(y) = c_1 y + 0.5 c_2 y^2, \qquad c_i > 0 \text{ for } i = 1, 2$$

The profits of a representative firm are $py - c(y)$.

Using (108.1), we can express the problem of the representative firm as

$$\max_y \Big[(a_0 - a_1 Y) y - c_1 y - 0.5 c_2 y^2 \Big] \tag{108.2}$$

In posing problem (108.2), we want the firm to be a *price taker*.

We do that by regarding p and therefore Y as exogenous to the firm.

The essence of the Big Y, little y trick is *not* to set $Y = ny$ *before* taking the first-order condition with respect to y in problem (108.2).

This assures that the firm is a price taker.

The first-order condition for problem (108.2) is

$$a_0 - a_1 Y - c_1 - c_2 y = 0 \tag{108.3}$$

At this point, *but not before*, we substitute $Y = y$ into (108.3) to obtain the following linear equation

$$a_0 - c_1 - (a_1 + c_2) Y = 0 \tag{108.4}$$

to be solved for the competitive equilibrium market-wide output Y.

After solving for Y, we can compute the competitive equilibrium price p from the inverse demand curve (108.1).

108.1.2 Related Planning Problem

Define **consumer surplus** as the area under the inverse demand curve:

$$S_c(Y) = \int_0^Y (a_0 - a_1 s) ds = a_o Y - \frac{a_1}{2} Y^2.$$

Define the social cost of production as

$$S_p(Y) = c_1 Y + \frac{c_2}{2} Y^2$$

Consider the planning problem

$$\max_Y [S_c(Y) - S_p(Y)]$$

The first-order necessary condition for the planning problem is equation (108.4).

Thus, a Y that solves (108.4) is a competitive equilibrium output as well as an output that solves the planning problem.

This type of outcome provides an intellectual justification for liking a competitive equilibrium.

108.1.3 Further Reading

References for this lecture include

- [Lucas and Prescott, 1971]
- [Sargent, 1987], chapter XIV
- [Ljungqvist and Sargent, 2018], chapter 7

108.2 Rational Expectations Equilibrium

Our first illustration of a rational expectations equilibrium involves a market with a unit measure of identical firms, each of which seeks to maximize the discounted present value of profits in the face of adjustment costs.

The adjustment costs induce the firms to make gradual adjustments, which in turn requires consideration of future prices.

Individual firms understand that, via the inverse demand curve, the price is determined by the amounts supplied by other firms.

Hence each firm wants to forecast future total industry output.

In our context, a forecast is generated by a belief about the law of motion for the aggregate state.

Rational expectations equilibrium prevails when this belief coincides with the actual law of motion generated by production choices induced by this belief.

We formulate a rational expectations equilibrium in terms of a fixed point of an operator that maps beliefs into optimal beliefs.

108.2.1 Competitive Equilibrium with Adjustment Costs

To illustrate, consider a collection of n firms producing a homogeneous good that is sold in a competitive market.

Each firm sell output $y_t(\omega) = y_t$.

The price p_t of the good lies on the inverse demand curve

$$p_t = a_0 - a_1 Y_t \tag{108.5}$$

where

- $a_i > 0$ for $i = 0, 1$
- $Y_t = \int_0^1 y_t(\omega) d\omega = y_t$ is the market-wide level of output

The Firm's Problem

Each firm is a price taker.

While it faces no uncertainty, it does face adjustment costs

In particular, it chooses a production plan to maximize

$$\sum_{t=0}^{\infty} \beta^t r_t \tag{108.6}$$

where

$$r_t := p_t y_t - \frac{\gamma(y_{t+1} - y_t)^2}{2}, \qquad y_0 \text{ given} \tag{108.7}$$

Regarding the parameters,

- $\beta \in (0, 1)$ is a discount factor
- $\gamma > 0$ measures the cost of adjusting the rate of output

Regarding timing, the firm observes p_t and y_t when it chooses y_{t+1} at time t.

To state the firm's optimization problem completely requires that we specify dynamics for all state variables.

This includes ones that the firm cares about but does not control like p_t.

We turn to this problem now.

Prices and Aggregate Output

In view of (108.5), the firm's incentive to forecast the market price translates into an incentive to forecast aggregate output Y_t.

Aggregate output depends on the choices of other firms.

The output $y_t(\omega)$ of a single firm ω has a negligible effect on aggregate output $\int_0^1 y_t(\omega) d\omega$.

That justifies firms in regarding their forecasts of aggregate output as being unaffected by their own output decisions.

Representative Firm's Beliefs

We suppose the firm believes that market-wide output Y_t follows the law of motion

$$Y_{t+1} = H(Y_t) \tag{108.8}$$

where Y_0 is a known initial condition.

The **belief function** H is an equilibrium object, and hence remains to be determined.

Optimal Behavior Given Beliefs

For now, let's fix a particular belief H in (108.8) and investigate the firm's response to it.

Let v be the optimal value function for the firm's problem given H.

The value function satisfies the Bellman equation

$$v(y, Y) = \max_{y'} \left\{ a_0 y - a_1 y Y - \frac{\gamma(y' - y)^2}{2} + \beta v(y', H(Y)) \right\} \tag{108.9}$$

Let's denote the firm's optimal policy function by h, so that

$$y_{t+1} = h(y_t, Y_t) \tag{108.10}$$

where

$$h(y, Y) := \text{argmax}_{y'} \left\{ a_0 y - a_1 y Y - \frac{\gamma(y' - y)^2}{2} + \beta v(y', H(Y)) \right\} \tag{108.11}$$

Evidently v and h both depend on H.

Characterization with First-Order Necessary Conditions

In what follows it will be helpful to have a second characterization of h, based on first-order conditions.

The first-order necessary condition for choosing y' is

$$-\gamma(y' - y) + \beta v_y(y', H(Y)) = 0 \tag{108.12}$$

An important useful envelope result of Benveniste-Scheinkman [Benveniste and Scheinkman, 1979] implies that to differentiate v with respect to y we can naively differentiate the right side of (108.9), giving

$$v_y(y, Y) = a_0 - a_1 Y + \gamma(y' - y)$$

Substituting this equation into (108.12) gives the **Euler equation**

$$-\gamma(y_{t+1} - y_t) + \beta[a_0 - a_1 Y_{t+1} + \gamma(y_{t+2} - y_{t+1})] = 0 \tag{108.13}$$

The firm optimally sets an output path that satisfies (108.13), taking (108.8) as given, and subject to

- the initial conditions for (y_0, Y_0).
- the terminal condition $\lim_{t\to\infty} \beta^t y_t v_y(y_t, Y_t) = 0$.

This last condition is called the **transversality condition**, and acts as a first-order necessary condition "at infinity".

A representative firm's decision rule solves the difference equation (108.13) subject to the given initial condition y_0 and the transversality condition.

Note that solving the Bellman equation (108.9) for v and then h in (108.11) yields a decision rule that automatically imposes both the Euler equation (108.13) and the transversality condition.

The Actual Law of Motion for Output

As we've seen, a given belief translates into a particular decision rule h.

Recalling that in equilbrium $Y_t = y_t$, the **actual law of motion** for market-wide output is then

$$Y_{t+1} = h(Y_t, Y_t) \tag{108.14}$$

Thus, when firms believe that the law of motion for market-wide output is (108.8), their optimizing behavior makes the actual law of motion be (108.14).

108.2.2 Definition of Rational Expectations Equilibrium

A **rational expectations equilibrium** or **recursive competitive equilibrium** of the model with adjustment costs is a decision rule h and an aggregate law of motion H such that

1. Given belief H, the map h is the firm's optimal policy function.
2. The law of motion H satisfies $H(Y) = h(Y, Y)$ for all Y.

Thus, a rational expectations equilibrium equates the perceived and actual laws of motion (108.8) and (108.14).

Fixed Point Characterization

As we've seen, the firm's optimum problem induces a mapping Φ from a perceived law of motion H for market-wide output to an actual law of motion $\Phi(H)$.

The mapping Φ is the composition of two mappings, the first of which maps a perceived law of motion into a decision rule via (108.9)–(108.11), the second of which maps a decision rule into an actual law via (108.14).

The H component of a rational expectations equilibrium is a fixed point of Φ.

108.3 Computing an Equilibrium

Now let's compute a rational expectations equilibrium.

108.3.1 Failure of Contractivity

Readers accustomed to dynamic programming arguments might try to address this problem by choosing some guess H_0 for the aggregate law of motion and then iterating with Φ.

Unfortunately, the mapping Φ is not a contraction.

Indeed, there is no guarantee that direct iterations on Φ converge[1].

There are examples in which these iterations diverge.

Fortunately, another method works here.

The method exploits a connection between equilibrium and Pareto optimality expressed in the fundamental theorems of welfare economics (see, e.g, [Mas-Colell *et al.*, 1995]).

Lucas and Prescott [Lucas and Prescott, 1971] used this method to construct a rational expectations equilibrium.

Some details follow.

108.3.2 A Planning Problem Approach

Our plan of attack is to match the Euler equations of the market problem with those for a single-agent choice problem.

As we'll see, this planning problem can be solved by LQ control (*linear regulator*).

Optimal quantities from the planning problem are rational expectations equilibrium quantities.

The rational expectations equilibrium price can be obtained as a shadow price in the planning problem.

We first compute a sum of consumer and producer surplus at time t

$$s(Y_t, Y_{t+1}) := \int_0^{Y_t} (a_0 - a_1 x)\, dx - \frac{\gamma (Y_{t+1} - Y_t)^2}{2} \tag{108.15}$$

The first term is the area under the demand curve, while the second measures the social costs of changing output.

[1] A literature that studies whether models populated with agents who learn can converge to rational expectations equilibria features iterations on a modification of the mapping Φ that can be approximated as $\gamma\Phi + (1-\gamma)I$. Here I is the identity operator and $\gamma \in (0, 1)$ is a *relaxation parameter*. See [Marcet and Sargent, 1989] and [Evans and Honkapohja, 2001] for statements and applications of this approach to establish conditions under which collections of adaptive agents who use least squares learning to converge to a rational expectations equilibrium.

The **planning problem** is to choose a production plan $\{Y_t\}$ to maximize

$$\sum_{t=0}^{\infty} \beta^t s(Y_t, Y_{t+1})$$

subject to an initial condition for Y_0.

108.3.3 Solution of Planning Problem

Evaluating the integral in (108.15) yields the quadratic form $a_0 Y_t - a_1 Y_t^2/2$.

As a result, the Bellman equation for the planning problem is

$$V(Y) = \max_{Y'} \left\{ a_0 Y - \frac{a_1}{2} Y^2 - \frac{\gamma (Y' - Y)^2}{2} + \beta V(Y') \right\} \tag{108.16}$$

The associated first-order condition is

$$-\gamma (Y' - Y) + \beta V'(Y') = 0 \tag{108.17}$$

Applying the same Benveniste-Scheinkman formula gives

$$V'(Y) = a_0 - a_1 Y + \gamma (Y' - Y)$$

Substituting this into equation (108.17) and rearranging leads to the Euler equation

$$\beta a_0 + \gamma Y_t - [\beta a_1 + \gamma (1 + \beta)] Y_{t+1} + \gamma \beta Y_{t+2} = 0 \tag{108.18}$$

108.3.4 Key Insight

Return to equation (108.13) and set $y_t = Y_t$ for all t.

A small amount of algebra will convince you that when $y_t = Y_t$, equations (108.18) and (108.13) are identical.

Thus, the Euler equation for the planning problem matches the second-order difference equation that we derived by

1. finding the Euler equation of the representative firm and
2. substituting into it the expression $Y_t = y_t$ that "makes the representative firm be representative".

If it is appropriate to apply the same terminal conditions for these two difference equations, which it is, then we have verified that a solution of the planning problem is also a rational expectations equilibrium quantity sequence.

It follows that for this example we can compute equilibrium quantities by forming the optimal linear regulator problem corresponding to the Bellman equation (108.16).

The optimal policy function for the planning problem is the aggregate law of motion H that the representative firm faces within a rational expectations equilibrium.

Structure of the Law of Motion

As you are asked to show in the exercises, the fact that the planner's problem is an LQ control problem implies an optimal policy — and hence aggregate law of motion — taking the form

$$Y_{t+1} = \kappa_0 + \kappa_1 Y_t \tag{108.19}$$

for some parameter pair κ_0, κ_1.

Now that we know the aggregate law of motion is linear, we can see from the firm's Bellman equation (108.9) that the firm's problem can also be framed as an LQ problem.

As you're asked to show in the exercises, the LQ formulation of the firm's problem implies a law of motion that looks as follows

$$y_{t+1} = h_0 + h_1 y_t + h_2 Y_t \tag{108.20}$$

Hence a rational expectations equilibrium will be defined by the parameters $(\kappa_0, \kappa_1, h_0, h_1, h_2)$ in (108.19)–(108.20).

108.4 Exercises

Exercise 108.4.1

Consider the firm problem *described above*.

Let the firm's belief function H be as given in (108.19).

Formulate the firm's problem as a discounted optimal linear regulator problem, being careful to describe all of the objects needed.

Use the class `LQ` from the QuantEcon.py package to solve the firm's problem for the following parameter values:

$$a_0 = 100, a_1 = 0.05, \beta = 0.95, \gamma = 10, \kappa_0 = 95.5, \kappa_1 = 0.95$$

Express the solution of the firm's problem in the form (108.20) and give the values for each h_j.

If there were a unit measure of identical competitive firms all behaving according to (108.20), what would (108.20) imply for the *actual* law of motion (108.8) for market supply.

Solution

To map a problem into a discounted optimal linear control problem, we need to define

- state vector x_t and control vector u_t
- matrices A, B, Q, R that define preferences and the law of motion for the state

For the state and control vectors, we choose

$$x_t = \begin{bmatrix} y_t \\ Y_t \\ 1 \end{bmatrix}, \qquad u_t = y_{t+1} - y_t$$

For B, Q, R we set

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \kappa_1 & \kappa_0 \\ 0 & 0 & 1 \end{bmatrix}, \quad B = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \quad R = \begin{bmatrix} 0 & a_1/2 & -a_0/2 \\ a_1/2 & 0 & 0 \\ -a_0/2 & 0 & 0 \end{bmatrix}, \quad Q = \gamma/2$$

By multiplying out you can confirm that

- $x_t' R x_t + u_t' Q u_t = -r_t$
- $x_{t+1} = A x_t + B u_t$

We'll use the module `lqcontrol.py` to solve the firm's problem at the stated parameter values.

This will return an LQ policy F with the interpretation $u_t = -Fx_t$, or

$$y_{t+1} - y_t = -F_0 y_t - F_1 Y_t - F_2$$

Matching parameters with $y_{t+1} = h_0 + h_1 y_t + h_2 Y_t$ leads to

$$h_0 = -F_2, \quad h_1 = 1 - F_0, \quad h_2 = -F_1$$

Here's our solution

```
# Model parameters

a0 = 100
a1 = 0.05
β = 0.95
γ = 10.0

# Beliefs

κ0 = 95.5
κ1 = 0.95

# Formulate the LQ problem

A = np.array([[1, 0, 0], [0, κ1, κ0], [0, 0, 1]])
B = np.array([1, 0, 0])
B.shape = 3, 1
R = np.array([[0, a1/2, -a0/2], [a1/2, 0, 0], [-a0/2, 0, 0]])
Q = 0.5 * γ

# Solve for the optimal policy

lq = LQ(Q, R, A, B, beta=β)
P, F, d = lq.stationary_values()
F = F.flatten()
out1 = f"F = [{F[0]:.3f}, {F[1]:.3f}, {F[2]:.3f}]"
h0, h1, h2 = -F[2], 1 - F[0], -F[1]
out2 = f"(h0, h1, h2) = ({h0:.3f}, {h1:.3f}, {h2:.3f})"

print(out1)
print(out2)
```

```
F = [-0.000, 0.046, -96.949]
(h0, h1, h2) = (96.949, 1.000, -0.046)
```

The implication is that

$$y_{t+1} = 96.949 + y_t - 0.046\, Y_t$$

For the case $n > 1$, recall that $Y_t = n y_t$, which, combined with the previous equation, yields

$$Y_{t+1} = n\,(96.949 + y_t - 0.046\, Y_t) = n96.949 + (1 - n0.046) Y_t$$

Exercise 108.4.2

Consider the following κ_0, κ_1 pairs as candidates for the aggregate law of motion component of a rational expectations equilibrium (see (108.19)).

Extending the program that you wrote for Exercise 108.4.1, determine which if any satisfy *the definition* of a rational expectations equilibrium

- (94.0886298678, 0.923409232937)
- (93.2119845412, 0.984323478873)
- (95.0818452486, 0.952459076301)

Describe an iterative algorithm that uses the program that you wrote for Exercise 108.4.1 to compute a rational expectations equilibrium.

(You are not being asked actually to use the algorithm you are suggesting)

Solution

To determine whether a κ_0, κ_1 pair forms the aggregate law of motion component of a rational expectations equilibrium, we can proceed as follows:

- Determine the corresponding firm law of motion $y_{t+1} = h_0 + h_1 y_t + h_2 Y_t$.
- Test whether the associated aggregate law :$Y_{t+1} = nh(Y_t/n, Y_t)$ evaluates to $Y_{t+1} = \kappa_0 + \kappa_1 Y_t$.

In the second step, we can use $Y_t = ny_t = y_t$, so that $Y_{t+1} = nh(Y_t/n, Y_t)$ becomes

$$Y_{t+1} = h(Y_t, Y_t) = h_0 + (h_1 + h_2)Y_t$$

Hence to test the second step we can test $\kappa_0 = h_0$ and $\kappa_1 = h_1 + h_2$.

The following code implements this test

```
candidates = ((94.0886298678, 0.923409232937),
              (93.2119845412, 0.984323478873),
              (95.0818452486, 0.952459076301))

for κ0, κ1 in candidates:

    # Form the associated law of motion
    A = np.array([[1, 0, 0], [0, κ1, κ0], [0, 0, 1]])

    # Solve the LQ problem for the firm
    lq = LQ(Q, R, A, B, beta=β)
    P, F, d = lq.stationary_values()
    F = F.flatten()
    h0, h1, h2 = -F[2], 1 - F[0], -F[1]

    # Test the equilibrium condition
    if np.allclose((κ0, κ1), (h0, h1 + h2)):
        print(f'Equilibrium pair = {κ0}, {κ1}')
        print('f(h0, h1, h2) = {h0}, {h1}, {h2}')
        break
```

```
Equilibrium pair = 95.0818452486, 0.952459076301
f(h0, h1, h2) = {h0}, {h1}, {h2}
```

The output tells us that the answer is pair (iii), which implies $(h_0, h_1, h_2) = (95.0819, 1.0000, -.0475)$.

(Notice we use `np.allclose` to test equality of floating-point numbers, since exact equality is too strict).

Regarding the iterative algorithm, one could loop from a given (κ_0, κ_1) pair to the associated firm law and then to a new (κ_0, κ_1) pair.

This amounts to implementing the operator Φ described in the lecture.

(There is in general no guarantee that this iterative process will converge to a rational expectations equilibrium)

Exercise 108.4.3

Recall the planner's problem *described above*

1. Formulate the planner's problem as an LQ problem.
2. Solve it using the same parameter values in exercise 1
   - $a_0 = 100, a_1 = 0.05, \beta = 0.95, \gamma = 10$
3. Represent the solution in the form $Y_{t+1} = \kappa_0 + \kappa_1 Y_t$.
4. Compare your answer with the results from exercise 2.

Solution

We are asked to write the planner problem as an LQ problem.

For the state and control vectors, we choose

$$x_t = \begin{bmatrix} Y_t \\ 1 \end{bmatrix}, \quad u_t = Y_{t+1} - Y_t$$

For the LQ matrices, we set

$$A = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad B = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad R = \begin{bmatrix} a_1/2 & -a_0/2 \\ -a_0/2 & 0 \end{bmatrix}, \quad Q = \gamma/2$$

By multiplying out you can confirm that

- $x_t' R x_t + u_t' Q u_t = -s(Y_t, Y_{t+1})$
- $x_{t+1} = Ax_t + Bu_t$

By obtaining the optimal policy and using $u_t = -Fx_t$ or

$$Y_{t+1} - Y_t = -F_0 Y_t - F_1$$

we can obtain the implied aggregate law of motion via $\kappa_0 = -F_1$ and $\kappa_1 = 1 - F_0$.

The Python code to solve this problem is below:

```
# Formulate the planner's LQ problem

A = np.array([[1, 0], [0, 1]])
B = np.array([[1], [0]])
R = np.array([[a1 / 2, -a0 / 2], [-a0 / 2, 0]])
Q = γ / 2

# Solve for the optimal policy

lq = LQ(Q, R, A, B, beta=β)
P, F, d = lq.stationary_values()
# Print the results

F = F.flatten()
κ0, κ1 = -F[1], 1 - F[0]
```

```
95.08187459214764 0.952459062703923
```

The output yields the same (κ_0, κ_1) pair obtained as an equilibrium from the previous exercise.

Exercise 108.4.4

A monopolist faces the industry demand curve (108.5) and chooses $\{Y_t\}$ to maximize $\sum_{t=0}^{\infty} \beta^t r_t$ where

$$r_t = p_t Y_t - \frac{\gamma (Y_{t+1} - Y_t)^2}{2}$$

Formulate this problem as an LQ problem.

Compute the optimal policy using the same parameters as Exercise 108.4.2.

In particular, solve for the parameters in

$$Y_{t+1} = m_0 + m_1 Y_t$$

Compare your results with Exercise 108.4.2 – comment.

Solution

The monopolist's LQ problem is almost identical to the planner's problem from the previous exercise, except that

$$R = \begin{bmatrix} a_1 & -a_0/2 \\ -a_0/2 & 0 \end{bmatrix}$$

The problem can be solved as follows

```
A = np.array([[1, 0], [0, 1]])
B = np.array([[1], [0]])
R = np.array([[a1, -a0 / 2], [-a0 / 2, 0]])
Q = γ / 2

lq = LQ(Q, R, A, B, beta=β)
P, F, d = lq.stationary_values()

F = F.flatten()
m0, m1 = -F[1], 1 - F[0]
print(m0, m1)
```

```
73.4729440350284 0.9265270559649701
```

We see that the law of motion for the monopolist is approximately $Y_{t+1} = 73.4729 + 0.9265 Y_t$.

In the rational expectations case, the law of motion was approximately $Y_{t+1} = 95.0818 + 0.9525 Y_t$.

One way to compare these two laws of motion is by their fixed points, which give long-run equilibrium output in each case.

For laws of the form $Y_{t+1} = c_0 + c_1 Y_t$, the fixed point is $c_0/(1 - c_1)$.

If you crunch the numbers, you will see that the monopolist adopts a lower long-run quantity than obtained by the competitive market, implying a higher market price.

This is analogous to the elementary static-case results

CHAPTER

NINE

STABILITY IN LINEAR RATIONAL EXPECTATIONS MODELS

Contents

- *Stability in Linear Rational Expectations Models*
  - *Overview*
  - *Linear Difference Equations*
  - *Illustration: Cagan's Model*
  - *Some Python Code*
  - *Alternative Code*
  - *Another Perspective*
  - *Log money Supply Feeds Back on Log Price Level*
  - *Big P, Little p Interpretation*
  - *Fun with SymPy*

In addition to what's in Anaconda, this lecture deploys the following libraries:

```
!pip install quantecon
```

```
import matplotlib.pyplot as plt
import numpy as np
import quantecon as qe
from sympy import init_printing, symbols, Matrix
init_printing()
```

109.1 Overview

This lecture studies stability in the context of an elementary rational expectations model.

We study a rational expectations version of Philip Cagan's model [Cagan, 1956] linking the price level to the money supply.

Cagan did not use a rational expectations version of his model, but Sargent [Sargent, 1977] did.

We study a rational expectations version of this model because it is intrinsically interesting and because it has a mathematical structure that appears in virtually all linear rational expectations model, namely, that a key endogenous variable equals a mathematical expectation of a geometric sum of future values of another variable.

The model determines the price level or rate of inflation as a function of the money supply or the rate of change in the money supply.

In this lecture, we'll encounter:

- a convenient formula for the expectation of geometric sum of future values of a variable
- a way of solving an expectational difference equation by mapping it into a vector first-order difference equation and appropriately manipulating an eigen decomposition of the transition matrix in order to impose stability
- a way to use a Big K, little k argument to allow apparent feedback from endogenous to exogenous variables within a rational expectations equilibrium
- a use of eigenvector decompositions of matrices that allowed Blanchard and Khan (1981) [Blanchard and Kahn, 1980] and Whiteman (1983) [Whiteman, 1983] to solve a class of linear rational expectations models
- how to use **SymPy** to get analytical formulas for some key objects comprising a rational expectations equilibrium

Matrix decompositions employed here are described in more depth in this lecture *Lagrangian formulations*.

We formulate a version of Cagan's model under rational expectations as an **expectational difference equation** whose solution is a rational expectations equilibrium.

We'll start this lecture with a quick review of deterministic (i.e., non-random) first-order and second-order linear difference equations.

109.2 Linear Difference Equations

We'll use the **backward shift** or **lag** operator L.

The lag operator L maps a sequence $\{x_t\}_{t=0}^{\infty}$ into the sequence $\{x_{t-1}\}_{t=0}^{\infty}$

We'll deploy L by using the equality $Lx_t \equiv x_{t-1}$ in algebraic expressions.

Further, the inverse L^{-1} of the lag operator is the **forward shift** operator.

We'll often use the equality $L^{-1}x_t \equiv x_{t+1}$ below.

The algebra of lag and forward shift operators can simplify representing and solving linear difference equations.

109.2.1 First Order

We want to solve a linear first-order scalar difference equation.

Let $|\lambda| < 1$ and let $\{u_t\}_{t=-\infty}^{\infty}$ be a bounded sequence of scalar real numbers.

Let L be the lag operator defined by $Lx_t \equiv x_{t-1}$ and let L^{-1} be the forward shift operator defined by $L^{-1}x_t \equiv x_{t+1}$.

Then

$$(1 - \lambda L)y_t = u_t, \forall t \tag{109.1}$$

has solutions

$$y_t = (1 - \lambda L)^{-1}u_t + k\lambda^t \tag{109.2}$$

or

$$y_t = \sum_{j=0}^{\infty} \lambda^j u_{t-j} + k\lambda^t$$

for any real number k.

You can verify this fact by applying $(1 - \lambda L)$ to both sides of equation (109.2) and noting that $(1 - \lambda L)\lambda^t = 0$.

To pin down k we need one condition imposed from outside (e.g., an initial or terminal condition) on the path of y.

Now let $|\lambda| > 1$.

Rewrite equation (109.1) as

$$y_{t-1} = \lambda^{-1} y_t - \lambda^{-1} u_t, \forall t \tag{109.3}$$

or

$$(1 - \lambda^{-1} L^{-1}) y_t = -\lambda^{-1} u_{t+1}. \tag{109.4}$$

A solution is

$$y_t = -\lambda^{-1} \left(\frac{1}{1 - \lambda^{-1} L^{-1}} \right) u_{t+1} + k\lambda^t \tag{109.5}$$

for any k.

To verify that this is a solution, check the consequences of operating on both sides of equation (109.5) by $(1 - \lambda L)$ and compare to equation (109.1).

For any bounded $\{u_t\}$ sequence, solution (109.2) exists for $|\lambda| < 1$ because the **distributed lag** in u converges.

Solution (109.5) exists when $|\lambda| > 1$ because the **distributed lead** in u converges.

When $|\lambda| > 1$, the distributed lag in u in (109.2) may diverge, in which case a solution of this form does not exist.

The distributed lead in u in (109.5) need not converge when $|\lambda| < 1$.

109.2.2 Second Order

Now consider the second order difference equation

$$(1 - \lambda_1 L)(1 - \lambda_2 L) y_{t+1} = u_t \tag{109.6}$$

where $\{u_t\}$ is a bounded sequence, y_0 is an initial condition, $|\lambda_1| < 1$ and $|\lambda_2| > 1$.

We seek a bounded sequence $\{y_t\}_{t=0}^{\infty}$ that satisfies (109.6). Using insights from our analysis of the first-order equation, operate on both sides of (109.6) by the forward inverse of $(1 - \lambda_2 L)$ to rewrite equation (109.6) as

$$(1 - \lambda_1 L) y_{t+1} = -\frac{\lambda_2^{-1}}{1 - \lambda_2^{-1} L^{-1}} u_{t+1}$$

or

$$y_{t+1} = \lambda_1 y_t - \lambda_2^{-1} \sum_{j=0}^{\infty} \lambda_2^{-j} u_{t+j+1}. \tag{109.7}$$

Thus, we obtained equation (109.7) by solving a stable root (in this case λ_1) **backward**, and an unstable root (in this case λ_2) **forward**.

Equation (109.7) has a form that we shall encounter often.

- $\lambda_1 y_t$ is called the **feedback part**
- $-\frac{\lambda_2^{-1}}{1 - \lambda_2^{-1} L^{-1}} u_{t+1}$ is called the **feedforward part**

109.3 Illustration: Cagan's Model

Now let's use linear difference equations to represent and solve Sargent's [Sargent, 1977] rational expectations version of Cagan's model [Cagan, 1956] that connects the price level to the public's anticipations of future money supplies.

Cagan did not use a rational expectations version of his model, but Sargent [Sargent, 1977]

Let

- m_t^d be the log of the demand for money
- m_t be the log of the supply of money
- p_t be the log of the price level

It follows that $p_{t+1} - p_t$ is the rate of inflation.

The logarithm of the demand for real money balances $m_t^d - p_t$ is an inverse function of the expected rate of inflation $p_{t+1} - p_t$ for $t \geq 0$:

$$m_t^d - p_t = -\beta(p_{t+1} - p_t), \quad \beta > 0$$

Equate the demand for log money m_t^d to the supply of log money m_t in the above equation and rearrange to deduce that the logarithm of the price level p_t is related to the logarithm of the money supply m_t by

$$p_t = (1 - \lambda)m_t + \lambda p_{t+1} \tag{109.8}$$

where $\lambda \equiv \frac{\beta}{1+\beta} \in (0, 1)$.

(We note that the characteristic polynomial if $1 - \lambda^{-1}z^{-1} = 0$ so that the zero of the characteristic polynomial in this case is $\lambda \in (0, 1)$ which here is **inside** the unit circle.)

Solving the first order difference equation (109.8) forward gives

$$p_t = (1 - \lambda) \sum_{j=0}^{\infty} \lambda^j m_{t+j}, \tag{109.9}$$

which is the unique **stable** solution of difference equation (109.8) among a class of more general solutions

$$p_t = (1 - \lambda) \sum_{j=0}^{\infty} \lambda^j m_{t+j} + c\lambda^{-t} \tag{109.10}$$

that is indexed by the real number $c \in \mathbf{R}$.

Because we want to focus on stable solutions, we set $c = 0$.

Equation (109.10) attributes **perfect foresight** about the money supply sequence to the holders of real balances.

We begin by assuming that the log of the money supply is **exogenous** in the sense that it is an autonomous process that does not feed back on the log of the price level.

In particular, we assume that the log of the money supply is described by the linear state space system

$$\begin{aligned} m_t &= Gx_t \\ x_{t+1} &= Ax_t \end{aligned} \tag{109.11}$$

where x_t is an $n \times 1$ vector that does not include p_t or lags of p_t, A is an $n \times n$ matrix with eigenvalues that are less than λ^{-1} in absolute values, and G is a $1 \times n$ selector matrix.

Variables appearing in the vector x_t contain information that might help predict future values of the money supply.

We'll start with an example in which x_t includes only m_t, possibly lagged values of m, and a constant.

An example of such an $\{m_t\}$ process that fits info state space system (109.11) is one that satisfies the second order linear difference equation

$$m_{t+1} = \alpha + \rho_1 m_t + \rho_2 m_{t-1}$$

where the zeros of the characteristic polynomial $(1 - \rho_1 z - \rho_2 z^2)$ are strictly greater than 1 in modulus.

(Please see *this* QuantEcon lecture for more about characteristic polynomials and their role in solving linear difference equations.)

We seek a stable or non-explosive solution of the difference equation (109.8) that obeys the system comprised of (109.8)-(109.11).

By stable or non-explosive, we mean that neither m_t nor p_t diverges as $t \to +\infty$.

This requires that we shut down the term $c\lambda^{-t}$ in equation (109.10) above by setting $c = 0$

The solution we are after is

$$p_t = F x_t \tag{109.12}$$

where

$$F = (1 - \lambda) G (I - \lambda A)^{-1} \tag{109.13}$$

Note

As mentioned above, an **explosive solution** of difference equation (109.8) can be constructed by adding to the right hand of (109.12) a sequence $c\lambda^{-t}$ where c is an arbitrary positive constant.

109.4 Some Python Code

We'll construct examples that illustrate (109.11).

Our first example takes as the law of motion for the log money supply the second order difference equation

$$m_{t+1} = \alpha + \rho_1 m_t + \rho_2 m_{t-1} \tag{109.14}$$

that is parameterized by ρ_1, ρ_2, α

To capture this parameterization with system (109.9) we set

$$x_t = \begin{bmatrix} 1 \\ m_t \\ m_{t-1} \end{bmatrix}, \quad A = \begin{bmatrix} 1 & 0 & 0 \\ \alpha & \rho_1 & \rho_2 \\ 0 & 1 & 0 \end{bmatrix}, \quad G = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}$$

Here is Python code

```
λ = .9

α = 0
ρ1 = .9
ρ2 = .05

A = np.array([[1,  0,  0],
              [α, ρ1, ρ2],
              [0,  1,  0]])
G = np.array([[0, 1, 0]])
```

The matrix A has one eigenvalue equal to unity.

It is associated with the A_{11} component that captures a constant component of the state x_t.

We can verify that the two eigenvalues of A not associated with the constant in the state x_t are strictly less than unity in modulus.

```
eigvals = np.linalg.eigvals(A)
print(eigvals)
```

```
[-0.05249378  0.95249378  1.        ]
```

```
(abs(eigvals) <= 1).all()
```

```
np.True_
```

Now let's compute F in formulas (109.12) and (109.13).

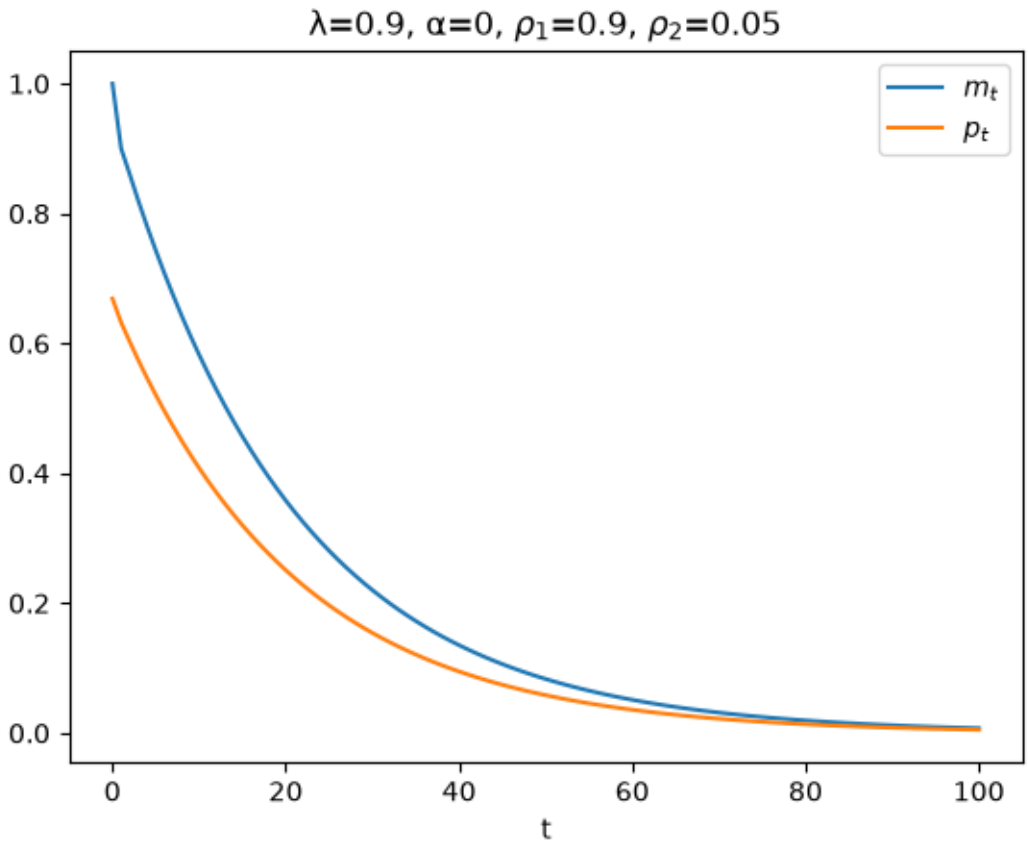
```
# compute the solution, i.e. forumula (3)
F = (1 - λ) * G @ np.linalg.inv(np.eye(A.shape[0]) - λ * A)
print("F= ",F)
```

```
F=  [[0.         0.66889632 0.03010033]]
```

Now let's simulate paths of m_t and p_t starting from an initial value x_0.

```
# set the initial state
x0 = np.array([1, 1, 0])

T = 100 # length of simulation

m_seq = np.empty(T+1)
p_seq = np.empty(T+1)

[m_seq[0]] = G @ x0
[p_seq[0]] = F @ x0

# simulate for T periods
x_old = x0
for t in range(T):

    x = A @ x_old

    [m_seq[t+1]] = G @ x
    [p_seq[t+1]] = F @ x

    x_old = x
```

```
plt.figure()
plt.plot(range(T+1), m_seq, label=r'$m_t$')
plt.plot(range(T+1), p_seq, label=r'$p_t$')
plt.xlabel('t')
plt.title(rf'λ={λ}, α={α}, $ρ_1$={ρ1}, $ρ_2$={ρ2}')
plt.legend()
plt.show()
```



In the above graph, why is the log of the price level always less than the log of the money supply?

Because

- according to equation (109.9), p_t is a geometric weighted average of current and future values of m_t, and
- it happens that in this example future m's are always less than the current m

109.5 Alternative Code

We could also have run the simulation using the quantecon **LinearStateSpace** code.

The following code block performs the calculation with that code.

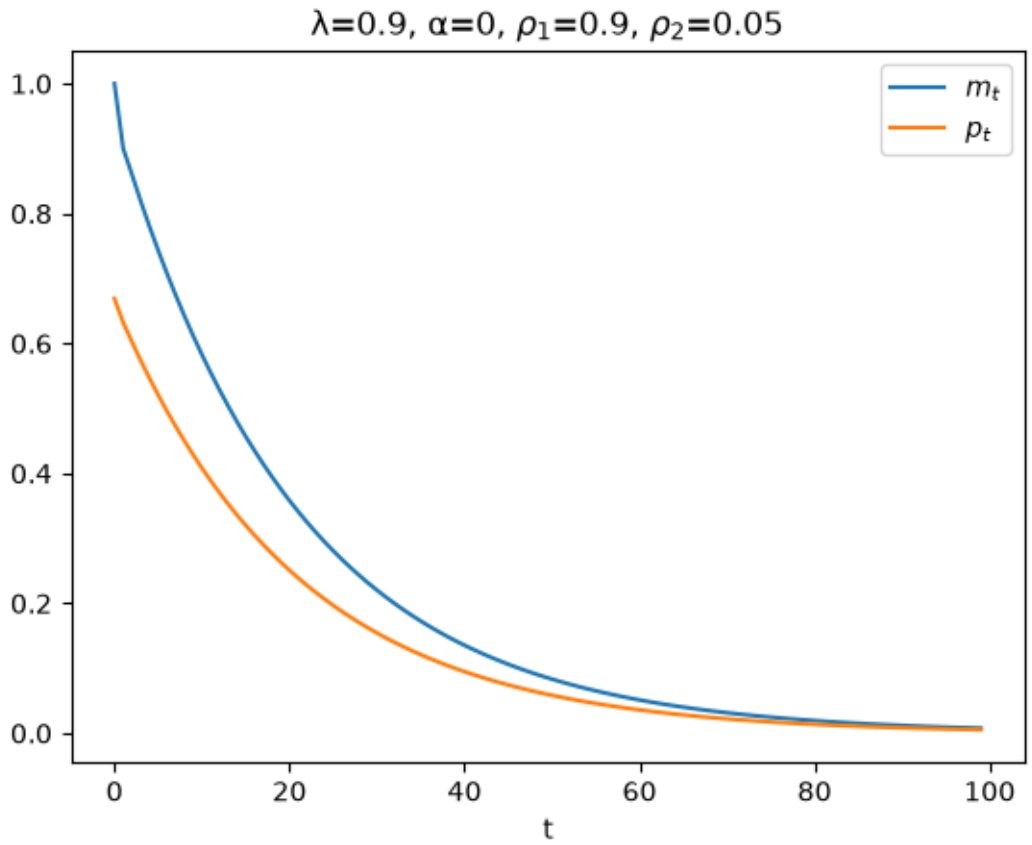
```
# construct a LinearStateSpace instance

# stack G and F
G_ext = np.vstack([G, F])

C = np.zeros((A.shape[0], 1))

ss = qe.LinearStateSpace(A, C, G_ext, mu_0=x0)
```

```
T = 100

# simulate using LinearStateSpace
```

(continues on next page)

(continued from previous page)

```
x, y = ss.simulate(ts_length=T)

# plot
plt.figure()
plt.plot(range(T), y[0,:], label='$m_t$')
plt.plot(range(T), y[1,:], label='$p_t$')
plt.xlabel('t')
plt.title(f'λ={λ}, α={α}, $ρ_1$={ρ1}, $ρ_2$={ρ2}')
plt.legend()
plt.show()
```



109.5.1 Special Case

To simplify our presentation in ways that will let focus on an important idea, in the above second-order difference equation (109.14) that governs m_t, we now set $\alpha = 0$, $\rho_1 = \rho \in (-1, 1)$, and $\rho_2 = 0$ so that the law of motion for m_t becomes

$$m_{t+1} = \rho m_t \tag{109.15}$$

and the state x_t becomes

$$x_t = m_t.$$

Consequently, we can set $G = 1, A = \rho$ making our formula (109.13) for F become

$$F = (1 - \lambda)(1 - \lambda\rho)^{-1}.$$

so that the log the log price level satisfies

$$p_t = Fm_t.$$

Please keep these formulas in mind as we investigate an alternative route to and interpretation of our formula for F.

109.6 Another Perspective

Above, we imposed stability or non-explosiveness on the solution of the key difference equation (109.8) in Cagan's model by solving the unstable root of the characteristic polynomial forward.

To shed light on the mechanics involved in imposing stability on a solution of a potentially unstable system of linear difference equations and to prepare the way for generalizations of our model in which the money supply is allowed to feed back on the price level itself, we stack equations (109.8) and (109.15) to form the system

$$\begin{bmatrix} m_{t+1} \\ p_{t+1} \end{bmatrix} = \begin{bmatrix} \rho & 0 \\ -(1-\lambda)/\lambda & \lambda^{-1} \end{bmatrix} \begin{bmatrix} m_t \\ p_t \end{bmatrix} \tag{109.16}$$

or

$$y_{t+1} = Hy_t, \quad t \geq 0 \tag{109.17}$$

where

$$H = \begin{bmatrix} \rho & 0 \\ -(1-\lambda)/\lambda & \lambda^{-1} \end{bmatrix}. \tag{109.18}$$

Transition matrix H has eigenvalues $\rho \in (0,1)$ and $\lambda^{-1} > 1$.

Because an eigenvalue of H exceeds unity, if we iterate on equation (109.17) starting from an arbitrary initial vector $y_0 = \begin{bmatrix} m_0 \\ p_0 \end{bmatrix}$ with $m_0 > 0, p_0 > 0$, we discover that in general absolute values of both components of y_t diverge toward $+\infty$ as $t \to +\infty$.

To substantiate this claim, we can use the eigenvector matrix decomposition of H that is available to us because the eigenvalues of H are distinct

$$H = Q\Lambda Q^{-1}.$$

Here Λ is a diagonal matrix of eigenvalues of H and Q is a matrix whose columns are eigenvectors associated with the corresponding eigenvalues.

Note that

$$H^t = Q\Lambda^t Q^{-1}$$

so that

$$y_t = Q\Lambda^t Q^{-1} y_0$$

For almost all initial vectors y_0, the presence of the eigenvalue $\lambda^{-1} > 1$ causes both components of y_t to diverge in absolute value to $+\infty$.

To explore this outcome in more detail, we can use the following transformation

$$y_t^* = Q^{-1} y_t$$

that allows us to represent the dynamics in a way that isolates the source of the propensity of paths to diverge:

$$y^*_{t+1} = \Lambda^t y^*_t$$

Staring at this equation indicates that unless

$$y^*_0 = \begin{bmatrix} y^*_{1,0} \\ 0 \end{bmatrix} \tag{109.19}$$

the path of y^*_t and therefore the paths of both components of $y_t = Q y^*_t$ will diverge in absolute value as $t \to +\infty$. (We say that the paths *explode*)

Equation (109.19) also leads us to conclude that there is a unique setting for the initial vector y_0 for which both components of y_t do not diverge.

The required setting of y_0 must evidently have the property that

$$Q y_0 = y^*_0 = \begin{bmatrix} y^*_{1,0} \\ 0 \end{bmatrix}.$$

But note that since $y_0 = \begin{bmatrix} m_0 \\ p_0 \end{bmatrix}$ and m_0 is given to us an initial condition, p_0 has to do all the adjusting to satisfy this equation.

Sometimes this situation is described by saying that while m_0 is truly a **state** variable, p_0 is a **jump** variable that must adjust at $t = 0$ in order to satisfy the equation.

Thus, in a nutshell the unique value of the vector y_0 for which the paths of y_t do not diverge must have second component p_0 that verifies equality (109.19) by setting the second component of y^*_0 equal to zero.

The component p_0 of the initial vector $y_0 = \begin{bmatrix} m_0 \\ p_0 \end{bmatrix}$ must evidently satisfy

$$Q^{\{2\}} y_0 = 0$$

where $Q^{\{2\}}$ denotes the second row of Q^{-1}, a restriction that is equivalent to

$$Q^{21} m_0 + Q^{22} p_0 = 0 \tag{109.20}$$

where Q^{ij} denotes the (i, j) component of Q^{-1}.

Solving this equation for p_0, we find

$$p_0 = -(Q^{22})^{-1} Q^{21} m_0. \tag{109.21}$$

This is the unique **stabilizing value** of p_0 expressed as a function of m_0.

109.6.1 Refining the Formula

We can get an even more convenient formula for p_0 that is cast in terms of components of Q instead of components of Q^{-1}.

To get this formula, first note that because $(Q^{21} \; Q^{22})$ is the second row of the inverse of Q and because $Q^{-1}Q = I$, it follows that

$$\begin{bmatrix} Q^{21} & Q^{22} \end{bmatrix} \begin{bmatrix} Q_{11} \\ Q_{21} \end{bmatrix} = 0$$

which implies that

$$Q^{21}Q_{11} + Q^{22}Q_{21} = 0.$$

Therefore,

$$-(Q^{22})^{-1}Q^{21} = Q_{21}Q_{11}^{-1}.$$

So we can write

$$p_0 = Q_{21}Q_{11}^{-1}m_0. \tag{109.22}$$

It can be verified that this formula replicates itself over time in the sense that

$$p_t = Q_{21}Q_{11}^{-1}m_t. \tag{109.23}$$

To implement formula (109.23), we want to compute Q_1 the eigenvector of Q associated with the stable eigenvalue ρ of Q.

By hand it can be verified that the eigenvector associated with the stable eigenvalue ρ is proportional to

$$Q_1 = \begin{bmatrix} 1 - \lambda\rho \\ 1 - \lambda \end{bmatrix}.$$

Notice that if we set $A = \rho$ and $G = 1$ in our earlier formula for p_t we get

$$p_t = G(I - \lambda A)^{-1}m_t = (1 - \lambda)(1 - \lambda\rho)^{-1}m_t,$$

a formula that is equivalent with

$$p_t = Q_{21}Q_{11}^{-1}m_t,$$

where

$$Q_1 = \begin{bmatrix} Q_{11} \\ Q_{21} \end{bmatrix}.$$

109.6.2 Remarks about Feedback

We have expressed (109.16) in what superficially appears to be a form in which y_{t+1} feeds back on y_t, even though what we actually want to represent is that the component p_t feeds **forward** on p_{t+1}, and through it, on future m_{t+j}, $j = 0, 1, 2, ...$

A tell-tale sign that we should look beyond its superficial "feedback" form is that $\lambda^{-1} > 1$ so that the matrix H in (109.16) is **unstable**

- it has one eigenvalue ρ that is less than one in modulus that does not imperil stability, but ...
- it has a second eigenvalue λ^{-1} that exceeds one in modulus and that makes H an unstable matrix

We'll keep these observations in mind as we turn now to a case in which the log money supply actually does feed back on the log of the price level.

109.7 Log money Supply Feeds Back on Log Price Level

An arrangement of eigenvalues that split around unity, with one being below unity and another being greater than unity, sometimes prevails when there is *feedback* from the log price level to the log money supply.

Let the feedback rule be

$$m_{t+1} = \rho m_t + \delta p_t \tag{109.24}$$

where $\rho \in (0, 1)$ and where we shall now allow $\delta \neq 0$.

Warning: If things are to fit together as we wish to deliver a stable system for some initial value p_0 that we want to determine uniquely, δ cannot be too large.

The forward-looking equation (109.8) continues to describe equality between the demand and supply of money.

We assume that equations (109.8) and (109.24) govern $y_t \equiv \begin{bmatrix} m_t \\ p_t \end{bmatrix}$ for $t \geq 0$.

The transition matrix H in the law of motion

$$y_{t+1} = H y_t$$

now becomes

$$H = \begin{bmatrix} \rho & \delta \\ -(1-\lambda)/\lambda & \lambda^{-1} \end{bmatrix}.$$

We take m_0 as a given initial condition and as before seek an initial value p_0 that stabilizes the system in the sense that y_t converges as $t \to +\infty$.

Our approach is identical with the one followed above and is based on an eigenvalue decomposition in which, cross our fingers, one eigenvalue exceeds unity and the other is less than unity in absolute value.

When $\delta \neq 0$ as we now assume, the eigenvalues of H will no longer be $\rho \in (0, 1)$ and $\lambda^{-1} > 1$

We'll just calculate them and apply the same algorithm that we used above.

That algorithm remains valid so long as the eigenvalues split around unity as before.

Again we assume that m_0 is an initial condition, but that p_0 is not given but to be solved for.

Let's write and execute some Python code that will let us explore how outcomes depend on δ.

```
def construct_H(ρ, λ, δ):
    "contruct matrix H given parameters."

    H = np.empty((2, 2))
    H[0, :] = ρ,δ
    H[1, :] = - (1 - λ) / λ, 1 / λ

    return H

def H_eigvals(ρ=.9, λ=.5, δ=0):
    "compute the eigenvalues of matrix H given parameters."

    # construct H matrix
    H = construct_H(ρ, λ, δ)

    # compute eigenvalues
    eigvals = np.linalg.eigvals(H)

    return eigvals
```

```
H_eigvals()
```

```
array([2. , 0.9])
```

Notice that a negative δ will not imperil the stability of the matrix H, even if it has a big absolute value.

```
# small negative δ
H_eigvals(δ=-0.05)
```

```
array([0.8562829, 2.0437171])
```

```
# large negative δ
H_eigvals(δ=-1.5)
```

```
array([0.10742784, 2.79257216])
```

A sufficiently small positive δ also causes no problem.

```
# sufficiently small positive δ
H_eigvals(δ=0.05)
```

```
array([0.94750622, 1.95249378])
```

But a large enough positive δ makes both eigenvalues of H strictly greater than unity in modulus.

For example,

```
H_eigvals(δ=0.2)
```

```
array([1.12984379, 1.77015621])
```

We want to study systems in which one eigenvalue exceeds unity in modulus while the other is less than unity in modulus, so we avoid values of δ that are too.

That is, we want to avoid too much positive feedback from p_t to m_{t+1}.

```
def magic_p0(m0, ρ=.9, λ=.5, δ=0):
    """
    Use the magic formula (8) to compute the level of p0
    that makes the system stable.
    """

    H = construct_H(ρ, λ, δ)
    eigvals, Q = np.linalg.eig(H)

    # find the index of the smaller eigenvalue
    ind = 0 if eigvals[0] < eigvals[1] else 1

    # verify that the eigenvalue is less than unity
    if eigvals[ind] > 1:

        print("both eigenvalues exceed unity in modulus")

        return None
```

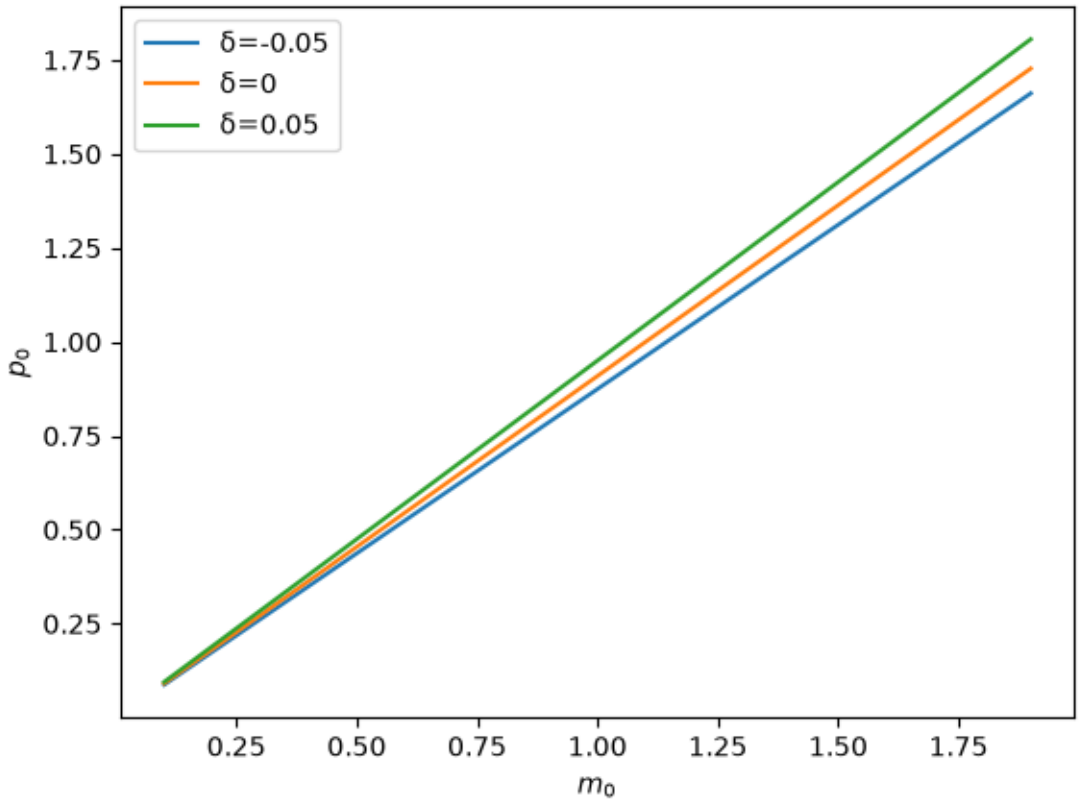
(continues on next page)

(continued from previous page)

```
    p0 = Q[1, ind] / Q[0, ind] * m0

    return p0
```

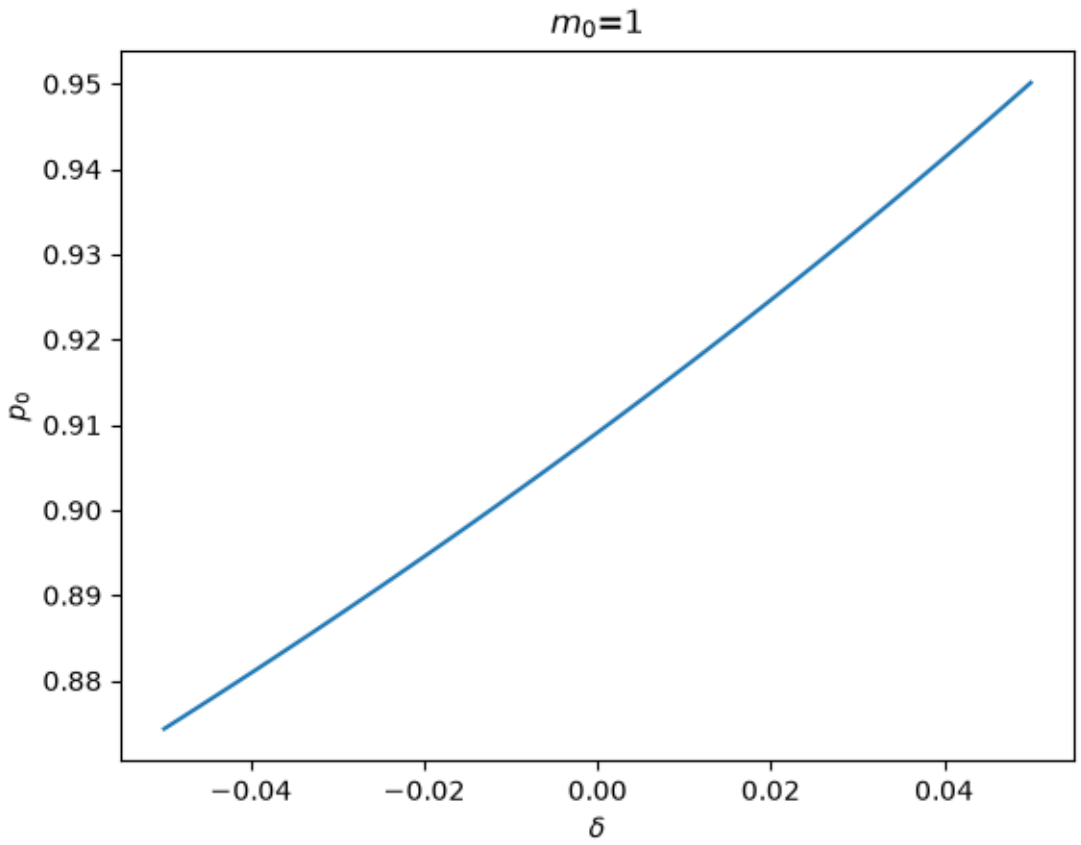
Let's plot how the solution p_0 changes as m_0 changes for different settings of δ.

```
m_range = np.arange(0.1, 2., 0.1)

for δ in [-0.05, 0, 0.05]:
    plt.plot(m_range, [magic_p0(m0, δ=δ) for m0 in m_range], label=f"δ={δ}")
plt.legend()

plt.xlabel(r"$m_0$")
plt.ylabel(r"$p_0$")
plt.show()
```



To look at things from a different angle, we can fix the initial value m_0 and see how p_0 changes as δ changes.

```
m0 = 1

δ_range = np.linspace(-0.05, 0.05, 100)
plt.plot(δ_range, [magic_p0(m0, δ=δ) for δ in δ_range])
plt.xlabel(r'$\delta$')
plt.ylabel(r'$p_0$')
plt.title(rf'$m_0$={m0}')
plt.show()
```



Notice that when δ is large enough, both eigenvalues exceed unity in modulus, causing a stabilizing value of p_0 not to exist.

```
magic_p0(1, δ=0.2)
```

```
both eigenvalues exceed unity in modulus
```

109.8 Big P, Little p Interpretation

It is helpful to view our solutions of difference equations having feedback from the price level or inflation to money or the rate of money creation in terms of the Big K, little k idea discussed in *Rational Expectations Models*.

This will help us sort out what is taken as given by the decision makers who use the difference equation (109.9) to determine p_t as a function of their forecasts of future values of m_t.

Let's write the stabilizing solution that we have computed using the eigenvector decomposition of H as $P_t = F^* m_t$, where

$$F^* = Q_{21} Q_{11}^{-1}.$$

Then from $P_{t+1} = F^* m_{t+1}$ and $m_{t+1} = \rho m_t + \delta P_t$ we can deduce the recursion $P_{t+1} = F^* \rho m_t + F^* \delta P_t$ and create the stacked system

$$\begin{bmatrix} m_{t+1} \\ P_{t+1} \end{bmatrix} = \begin{bmatrix} \rho & \delta \\ F^* \rho & F^* \delta \end{bmatrix} \begin{bmatrix} m_t \\ P_t \end{bmatrix}$$

or

$$x_{t+1} = Ax_t$$

where $x_t = \begin{bmatrix} m_t \\ P_t \end{bmatrix}$.

Apply formula (109.13) for F to deduce that

$$p_t = F \begin{bmatrix} m_t \\ P_t \end{bmatrix} = F \begin{bmatrix} m_t \\ F^* m_t \end{bmatrix}$$

which implies that

$$p_t = \begin{bmatrix} F_1 & F_2 \end{bmatrix} \begin{bmatrix} m_t \\ F^* m_t \end{bmatrix} = F_1 m_t + F_2 F^* m_t$$

so that we can anticipate that

$$F^* = F_1 + F_2 F^*$$

We shall verify this equality in the next block of Python code that implements the following computations.

1. For the system with $\delta \neq 0$ so that there is feedback, we compute the stabilizing solution for p_t in the form $p_t = F^* m_t$ where $F^* = Q_{21} Q_{11}^{-1}$ as above.
2. Recalling the system (109.11), (109.12), and (109.13) above, we define $x_t = \begin{bmatrix} m_t \\ P_t \end{bmatrix}$ and notice that it is Big P_t and not little p_t here. Then we form A and G as $A = \begin{bmatrix} \rho & \delta \\ F^* \rho & F^* \delta \end{bmatrix}$ and $G = \begin{bmatrix} 1 & 0 \end{bmatrix}$ and we compute $\begin{bmatrix} F_1 & F_2 \end{bmatrix} \equiv F$ from equation (109.13) above.
3. We compute $F_1 + F_2 F^*$ and compare it with F^* and check for the anticipated equality.

```
# set parameters
ρ = .9
λ = .5
δ = .05
```

```
# solve for F_star
H = construct_H(ρ, λ, δ)
eigvals, Q = np.linalg.eig(H)

ind = 0 if eigvals[0] < eigvals[1] else 1
F_star = Q[1, ind] / Q[0, ind]
F_star
```

```
0.950124378879 11
```

```
# solve for F_check
A = np.empty((2, 2))
A[0, :] = ρ, δ
A[1, :] = F_star * A[0, :]

G = np.array([1, 0])

F_check= (1 - λ) * G @ np.linalg.inv(np.eye(2) - λ * A)
F_check
```

```
array([0.92755597, 0.02375311])
```

Compare F^* with $F_1 + F_2 F^*$

```
F_check[0] + F_check[1] * F_star, F_star
```

$$(0.950124378879\,11,\ 0.950124378879\,11)$$

109.9 Fun with SymPy

This section is a gift for readers who have made it this far.

It puts SymPy to work on our model.

Thus, we use Sympy to compute some key objects comprising the eigenvector decomposition of H.

We start by generating an H with nonzero δ.

```
λ, δ, ρ = symbols('λ, δ, ρ')
```

```
H1 = Matrix([[ρ,δ], [- (1 - λ) / λ, λ ** -1]])
```

```
H1
```

$$\begin{bmatrix} \rho & \delta \\ \frac{\lambda-1}{\lambda} & \frac{1}{\lambda} \end{bmatrix}$$

```
H1.eigenvals()
```

$$\left\{(\lambda\rho+1)/2\lambda - \sqrt{4\delta\lambda^2 - 4\delta\lambda + \lambda^2\rho^2 - 2\lambda\rho + 1}/2\lambda : 1,\ (\lambda\rho+1)/2\lambda + \sqrt{4\delta\lambda^2 - 4\delta\lambda + \lambda^2\rho^2 - 2\lambda\rho + 1}/2\lambda : 1\right\}$$

```
H1.eigenvects()
```

$$\left[\left(\frac{\lambda\rho+1}{2\lambda} - \frac{\sqrt{4\delta\lambda^2 - 4\delta\lambda + \lambda^2\rho^2 - 2\lambda\rho + 1}}{2\lambda},\ 1,\ \left[\begin{bmatrix} \frac{\lambda\left(\frac{\lambda\rho+1}{2\lambda} - \frac{\sqrt{4\delta\lambda^2-4\delta\lambda+\lambda^2\rho^2-2\lambda\rho+1}}{2\lambda}\right)}{\lambda-1} - \frac{1}{\lambda-1} \\ 1 \end{bmatrix}\right]\right),\ \left(\frac{\lambda\rho+1}{2\lambda} + \frac{\sqrt{4\delta\lambda^2 - 4\delta\lambda}\right.$$

Now let's compute H when δ is zero.

```
H2 = Matrix([[ρ,0], [- (1 - λ) / λ, λ ** -1]])
```

```
H2
```

$$\begin{bmatrix} \rho & 0 \\ \frac{\lambda-1}{\lambda} & \frac{1}{\lambda} \end{bmatrix}$$

```
H2.eigenvals()
```

$$\{1/\lambda : 1,\ \rho : 1\}$$

```
H2.eigenvects()
```

$$\left[\left(\frac{1}{\lambda},\ 1,\ \left[\begin{bmatrix}0\\1\end{bmatrix}\right]\right),\ \left(\rho,\ 1,\ \left[\begin{bmatrix}\frac{\lambda\rho-1}{\lambda-1}\\1\end{bmatrix}\right]\right)\right]$$

Below we do induce SymPy to do the following fun things for us analytically:

1. We compute the matrix Q whose first column is the eigenvector associated with ρ. and whose second column is the eigenvector associated with λ^{-1}.
2. We use SymPy to compute the inverse Q^{-1} of Q (both in symbols).
3. We use SymPy to compute $Q_{21}Q_{11}^{-1}$ (in symbols).
4. Where Q^{ij} denotes the (i,j) component of Q^{-1}, we use SymPy to compute $-(Q^{22})^{-1}Q^{21}$ (again in symbols)

```
# construct Q
vec = []
for i, (eigval, _, eigvec) in enumerate(H2.eigenvects()):

    vec.append(eigvec[0])

    if eigval == ρ:
        ind = i

Q = vec[ind].col_insert(1, vec[1-ind])
```

```
Q
```

$$\begin{bmatrix}\frac{\lambda\rho-1}{\lambda-1} & 0\\ 1 & 1\end{bmatrix}$$

Q^{-1}

```
Q_inv = Q ** (-1)
Q_inv
```

$$\begin{bmatrix}\frac{\lambda-1}{\lambda\rho-1} & 0\\ \frac{1-\lambda}{\lambda\rho-1} & 1\end{bmatrix}$$

$Q_{21}Q_{11}^{-1}$

```
Q[1, 0] / Q[0, 0]
```

$$\frac{\lambda-1}{\lambda\rho-1}$$

$-(Q^{22})^{-1}Q^{21}$

```
- Q_inv[1, 0] / Q_inv[1, 1]
```

$$-\frac{1-\lambda}{\lambda\rho-1}$$

CHAPTER

MARKOV PERFECT EQUILIBRIUM

Contents

- *Markov Perfect Equilibrium*
  - *Overview*
  - *Background*
  - *Linear Markov Perfect Equilibria*
  - *Application*
  - *Exercises*

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install quantecon
```

110.1 Overview

This lecture describes the concept of Markov perfect equilibrium.

Markov perfect equilibrium is a key notion for analyzing economic problems involving dynamic strategic interaction, and a cornerstone of applied game theory.

In this lecture, we teach Markov perfect equilibrium by example.

We will focus on settings with

- two players
- quadratic payoff functions
- linear transition rules for the state

Other references include chapter 7 of [Ljungqvist and Sargent, 2018].

Let's start with some standard imports:

```
import matplotlib.pyplot as plt
import numpy as np
import quantecon as qe
```

110.2 Background

Markov perfect equilibrium is a refinement of the concept of Nash equilibrium.

It is used to study settings where multiple decision-makers interact non-cooperatively over time, each pursuing its own objective.

The agents in the model face a common state vector, the time path of which is influenced by – and influences – their decisions.

In particular, the transition law for the state that confronts each agent is affected by decision rules of other agents.

Individual payoff maximization requires that each agent solve a dynamic programming problem that includes this transition law.

Markov perfect equilibrium prevails when no agent wishes to revise its policy, taking as given the policies of all other agents.

Well known examples include

- Choice of price, output, location or capacity for firms in an industry (e.g., [Ericson and Pakes, 1995], [Ryan, 2012], [Doraszelski and Satterthwaite, 2010]).
- Rate of extraction from a shared natural resource, such as a fishery (e.g., [Levhari and Mirman, 1980], [Van Long, 2011]).

Let's examine a model of the first type.

110.2.1 Example: A Duopoly Model

Two firms are the only producers of a good, the demand for which is governed by a linear inverse demand function

$$p = a_0 - a_1(q_1 + q_2) \tag{110.1}$$

Here $p = p_t$ is the price of the good, $q_i = q_{it}$ is the output of firm $i = 1, 2$ at time t and $a_0 > 0, a_1 > 0$.

In (110.1) and what follows,

- the time subscript is suppressed when possible to simplify notation
- $\hat{x}$ denotes a next period value of variable x

Each firm recognizes that its output affects total output and therefore the market price.

The one-period payoff function of firm i is price times quantity minus adjustment costs:

$$\pi_i = pq_i - \gamma(\hat{q}_i - q_i)^2, \quad \gamma > 0, \tag{110.2}$$

Substituting the inverse demand curve (110.1) into (110.2) lets us express the one-period payoff as

$$\pi_i(q_i, q_{-i}, \hat{q}_i) = a_0 q_i - a_1 q_i^2 - a_1 q_i q_{-i} - \gamma(\hat{q}_i - q_i)^2, \tag{110.3}$$

where q_{-i} denotes the output of the firm other than i.

The objective of the firm is to maximize $\sum_{t=0}^{\infty} \beta^t \pi_{it}$.

Firm i chooses a decision rule that sets next period quantity $\hat{q}_i$ as a function f_i of the current state (q_i, q_{-i}).

An essential aspect of a Markov perfect equilibrium is that each firm takes the decision rule of the other firm as known and given.

Given f_{-i}, the Bellman equation of firm i is

$$v_i(q_i, q_{-i}) = \max_{\hat{q}_i} \{\pi_i(q_i, q_{-i}, \hat{q}_i) + \beta v_i(\hat{q}_i, f_{-i}(q_{-i}, q_i))\} \tag{110.4}$$

Definition 110.2.1 (Markov Perfect Equilibrium)

A **Markov perfect equilibrium** of the duopoly model is a pair of value functions (v_1, v_2) and a pair of policy functions (f_1, f_2) such that, for each $i \in \{1, 2\}$ and each possible state,

- The value function v_i satisfies Bellman equation (110.4).
- The maximizer on the right side of (110.4) equals $f_i(q_i, q_{-i})$.

The adjective "Markov" denotes that the equilibrium decision rules depend only on the current values of the state variables, not other parts of their histories.

"Perfect" means complete, in the sense that the equilibrium is constructed by backward induction and hence builds in optimizing behavior for each firm at all possible future states.

- These include many states that will not be reached when we iterate forward on the pair of equilibrium strategies f_i starting from a given initial state.

110.2.2 Computation

One strategy for computing a Markov perfect equilibrium is iterating to convergence on pairs of Bellman equations and decision rules.

In particular, let v_i^j, f_i^j be the value function and policy function for firm i at the j-th iteration.

Imagine constructing the iterates

$$v_i^{j+1}(q_i, q_{-i}) = \max_{\hat{q}_i} \left\{\pi_i(q_i, q_{-i}, \hat{q}_i) + \beta v_i^j(\hat{q}_i, f_{-i}(q_{-i}, q_i))\right\} \tag{110.5}$$

These iterations can be challenging to implement computationally.

However, they simplify for the case in which one-period payoff functions are quadratic and transition laws are linear — which takes us to our next topic.

110.3 Linear Markov Perfect Equilibria

As we saw in the duopoly example, the study of Markov perfect equilibria in games with two players leads us to an interrelated pair of Bellman equations.

In linear-quadratic dynamic games, these "stacked Bellman equations" become "stacked Riccati equations" with a tractable mathematical structure.

We'll lay out that structure in a general setup and then apply it to some simple problems.

110.3.1 Coupled Linear Regulator Problems

We consider a general linear-quadratic regulator game with two players.

For convenience, we'll start with a finite horizon formulation, where t_0 is the initial date and t_1 is the common terminal date.

Player i takes $\{u_{-it}\}$ as given and minimizes

$$\sum_{t=t_0}^{t_1-1} \beta^{t-t_0} \{x_t' R_i x_t + u_{it}' Q_i u_{it} + u_{-it}' S_i u_{-it} + 2x_t' W_i u_{it} + 2u_{-it}' M_i u_{it}\} \tag{110.6}$$

while the state evolves according to

$$x_{t+1} = Ax_t + B_1 u_{1t} + B_2 u_{2t} \tag{110.7}$$

Here

- x_t is an $n \times 1$ state vector and u_{it} is a $k_i \times 1$ vector of controls for player i
- R_i is $n \times n$
- S_i is $k_{-i} \times k_{-i}$
- Q_i is $k_i \times k_i$
- W_i is $n \times k_i$
- M_i is $k_{-i} \times k_i$
- A is $n \times n$
- B_i is $n \times k_i$

110.3.2 Computing Equilibrium

We formulate a linear Markov perfect equilibrium as follows.

Player i employs linear decision rules $u_{it} = -F_{it} x_t$, where F_{it} is a $k_i \times n$ matrix.

A Markov perfect equilibrium is a pair of sequences $\{F_{1t}, F_{2t}\}$ over $t = t_0, \ldots, t_1 - 1$ such that

- $\{F_{1t}\}$ solves player 1's problem, taking $\{F_{2t}\}$ as given, and
- $\{F_{2t}\}$ solves player 2's problem, taking $\{F_{1t}\}$ as given

If we take $u_{2t} = -F_{2t} x_t$ and substitute it into (110.6) and (110.7), then player 1's problem becomes minimization of

$$\sum_{t=t_0}^{t_1-1} \beta^{t-t_0} \{x_t' \Pi_{1t} x_t + u_{1t}' Q_1 u_{1t} + 2u_{1t}' \Gamma_{1t} x_t\} \tag{110.8}$$

subject to

$$x_{t+1} = \Lambda_{1t} x_t + B_1 u_{1t}, \tag{110.9}$$

where

- $\Lambda_{it} := A - B_{-i} F_{-it}$
- $\Pi_{it} := R_i + F_{-it}' S_i F_{-it}$
- $\Gamma_{it} := W_i' - M_i' F_{-it}$

This is an LQ dynamic programming problem that can be solved by working backwards.

Decision rules that solve this problem are

$$F_{1t} = (Q_1 + \beta B_1' P_{1t+1} B_1)^{-1} (\beta B_1' P_{1t+1} \Lambda_{1t} + \Gamma_{1t}) \tag{110.10}$$

where P_{1t} solves the matrix Riccati difference equation

$$P_{1t} = \Pi_{1t} - (\beta B_1' P_{1t+1} \Lambda_{1t} + \Gamma_{1t})' (Q_1 + \beta B_1' P_{1t+1} B_1)^{-1} (\beta B_1' P_{1t+1} \Lambda_{1t} + \Gamma_{1t}) + \beta \Lambda_{1t}' P_{1t+1} \Lambda_{1t} \tag{110.11}$$

Similarly, decision rules that solve player 2's problem are

$$F_{2t} = (Q_2 + \beta B_2' P_{2t+1} B_2)^{-1} (\beta B_2' P_{2t+1} \Lambda_{2t} + \Gamma_{2t}) \tag{110.12}$$

where P_{2t} solves

$$P_{2t} = \Pi_{2t} - (\beta B_2' P_{2t+1} \Lambda_{2t} + \Gamma_{2t})' (Q_2 + \beta B_2' P_{2t+1} B_2)^{-1} (\beta B_2' P_{2t+1} \Lambda_{2t} + \Gamma_{2t}) + \beta \Lambda_{2t}' P_{2t+1} \Lambda_{2t} \tag{110.13}$$

Here, in all cases $t = t_0, \ldots, t_1 - 1$ and the terminal conditions are $P_{it_1} = 0$.

The solution procedure is to use equations (110.10), (110.11), (110.12), and (110.13), and "work backwards" from time $t_1 - 1$.

Since we're working backward, P_{1t+1} and P_{2t+1} are taken as given at each stage.

Moreover, since

- some terms on the right-hand side of (110.10) contain F_{2t}
- some terms on the right-hand side of (110.12) contain F_{1t}

we need to solve these $k_1 + k_2$ equations simultaneously.

Key Insight

A key insight is that equations (110.10) and (110.12) are linear in F_{1t} and F_{2t}.

After these equations have been solved, we can take F_{it} and solve for P_{it} in (110.11) and (110.13).

Infinite Horizon

We often want to compute the solutions of such games for infinite horizons, in the hope that the decision rules F_{it} settle down to be time-invariant as $t_1 \to +\infty$.

In practice, we usually fix t_1 and compute the equilibrium of an infinite horizon game by driving $t_0 \to -\infty$.

This is the approach we adopt in the next section.

110.3.3 Implementation

We use the function nnash from QuantEcon.py that computes a Markov perfect equilibrium of the infinite horizon linear-quadratic dynamic game in the manner described above.

110.4 Application

Let's use these procedures to treat some applications, starting with the duopoly model.

110.4.1 A Duopoly Model

To map the duopoly model into coupled linear-quadratic dynamic programming problems, define the state and controls as

$$x_t := \begin{bmatrix} 1 \\ q_{1t} \\ q_{2t} \end{bmatrix} \quad \text{and} \quad u_{it} := q_{i,t+1} - q_{it}, \quad i = 1, 2$$

If we write

$$x_t' R_i x_t + u_{it}' Q_i u_{it}$$

where $Q_1 = Q_2 = \gamma$,

$$R_1 := \begin{bmatrix} 0 & -\frac{a_0}{2} & 0 \\ -\frac{a_0}{2} & a_1 & \frac{a_1}{2} \\ 0 & \frac{a_1}{2} & 0 \end{bmatrix} \quad \text{and} \quad R_2 := \begin{bmatrix} 0 & 0 & -\frac{a_0}{2} \\ 0 & 0 & \frac{a_1}{2} \\ -\frac{a_0}{2} & \frac{a_1}{2} & a_1 \end{bmatrix}$$

then we recover the one-period payoffs in expression (110.3).

The law of motion for the state x_t is $x_{t+1} = Ax_t + B_1 u_{1t} + B_2 u_{2t}$ where

$$A := \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad B_1 := \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \quad B_2 := \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

The optimal decision rule of firm i will take the form $u_{it} = -F_i x_t$, inducing the following closed-loop system for the evolution of x in the Markov perfect equilibrium:

$$x_{t+1} = (A - B_1 F_1 - B_1 F_2) x_t \tag{110.14}$$

110.4.2 Parameters and Solution

Consider the previously presented duopoly model with parameter values of:

- $a_0 = 10$
- $a_1 = 2$
- $\beta = 0.96$
- $\gamma = 12$

From these, we compute the infinite horizon MPE using the preceding code

```
import numpy as np
import quantecon as qe

# Parameters
a0 = 10.0
a1 = 2.0
```

(continues on next page)

(continued from previous page)

```
β = 0.96
γ = 12.0

# In LQ form
A = np.eye(3)
B1 = np.array([[0.], [1.], [0.]])
B2 = np.array([[0.], [0.], [1.]])


R1 = [[      0.,     -a0 / 2,           0.],
      [-a0 / 2.,          a1,     a1 / 2.],
      [       0,     a1 / 2.,           0.]]

R2 = [[     0.,           0.,      -a0 / 2],
      [     0.,           0.,      a1 / 2.],
      [-a0 / 2,      a1 / 2.,           a1]]

Q1 = Q2 = γ
S1 = S2 = W1 = W2 = M1 = M2 = 0.0

# Solve using QE's nnash function
F1, F2, P1, P2 = qe.nnash(A, B1, B2, R1, R2, Q1,
                          Q2, S1, S2, W1, W2, M1,
                          M2, beta=β)

# Display policies
print("Computed policies for firm 1 and firm 2:\n")
print(f"F1 = {F1}")
print(f"F2 = {F2}")
print("\n")
```

```
Computed policies for firm 1 and firm 2:

F1 = [[-0.66846615  0.29512482  0.07584666]]
F2 = [[-0.66846615  0.07584666  0.29512482]]
```

Running the code produces the following output.

One way to see that F_i is indeed optimal for firm i taking F_2 as given is to use QuantEcon.py's LQ class.

In particular, let's take F2 as computed above, plug it into (110.8) and (110.9) to get firm 1's problem and solve it using LQ.

We hope that the resulting policy will agree with F1 as computed above

```
beta = 0.96
Λ1 = A - B2 @ F2
lq1 = qe.LQ(Q1, R1, Λ1, B1, beta=beta)
P1_ih, F1_ih, d = lq1.stationary_values()
F1_ih
```

```
array([[-0.66846613,  0.29512482,  0.07584666]])
```

This is close enough for rock and roll, as they say in the trade.

Indeed, np.allclose agrees with our assessment

```
np.allclose(F1, F1_ih)
```

```
True
```

110.4.3 Dynamics

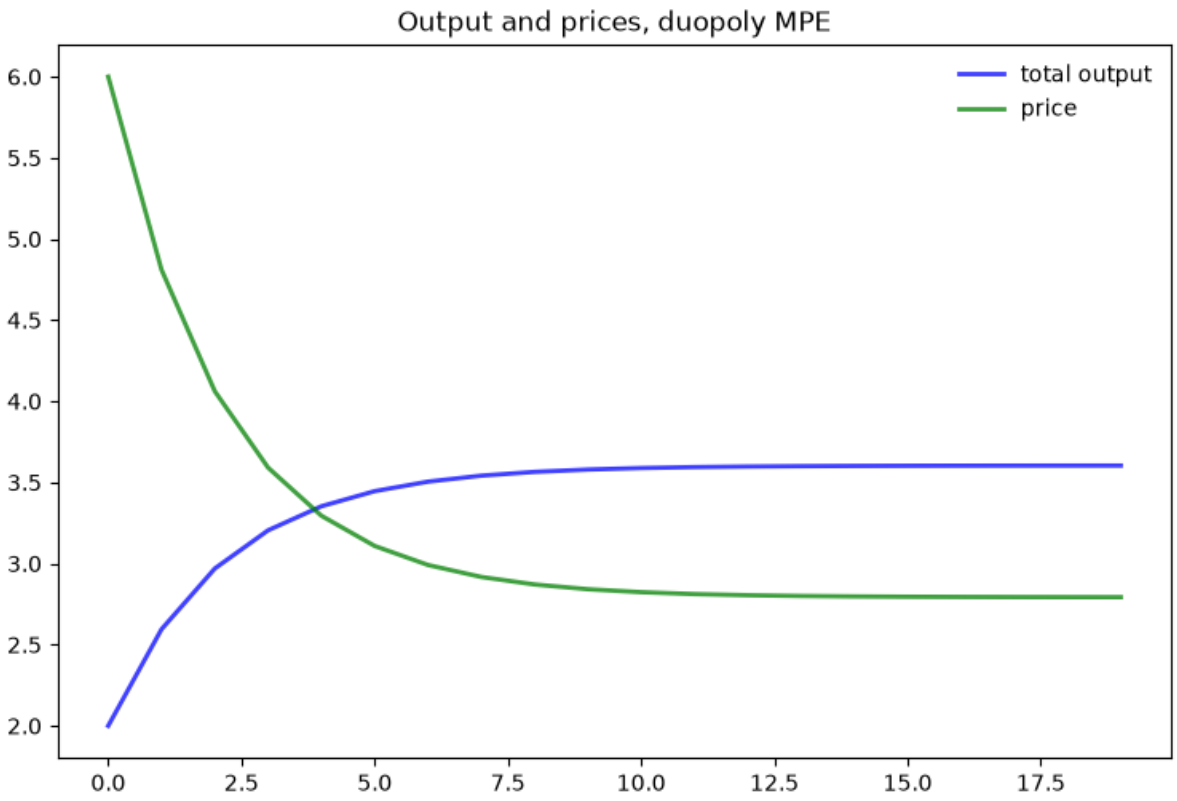
Let's now investigate the dynamics of price and output in this simple duopoly model under the MPE policies.

Given our optimal policies $F1$ and $F2$, the state evolves according to (110.14).

The following program

- imports $F1$ and $F2$ from the previous program along with all parameters.
- computes the evolution of x_t using (110.14).
- extracts and plots industry output $q_t = q_{1t} + q_{2t}$ and price $p_t = a_0 - a_1 q_t$.

```
AF = A - B1 @ F1 - B2 @ F2
n = 20
x = np.empty((3, n))
x[:, 0] = 1, 1, 1
for t in range(n-1):
    x[:, t+1] = AF @ x[:, t]
q1 = x[1, :]
q2 = x[2, :]
q = q1 + q2       # Total output, MPE
p = a0 - a1 * q   # Price, MPE

fig, ax = plt.subplots(figsize=(9, 5.8))
ax.plot(q, 'b-', lw=2, alpha=0.75, label='total output')
ax.plot(p, 'g-', lw=2, alpha=0.75, label='price')
ax.set_title('Output and prices, duopoly MPE')
ax.legend(frameon=False)
plt.show()
```



Note that the initial condition has been set to $q_{10} = q_{20} = 1.0$.

To gain some perspective we can compare this to what happens in the monopoly case.

The first panel in the next figure compares output of the monopolist and industry output under the MPE, as a function of time.

The second panel shows analogous curves for price.

Here parameters are the same as above for both the MPE and monopoly solutions.

The monopolist initial condition is $q_0 = 2.0$ to mimic the industry initial condition $q_{10} = q_{20} = 1.0$ in the MPE case.

As expected, output is higher and prices are lower under duopoly than monopoly.

110.5 Exercises

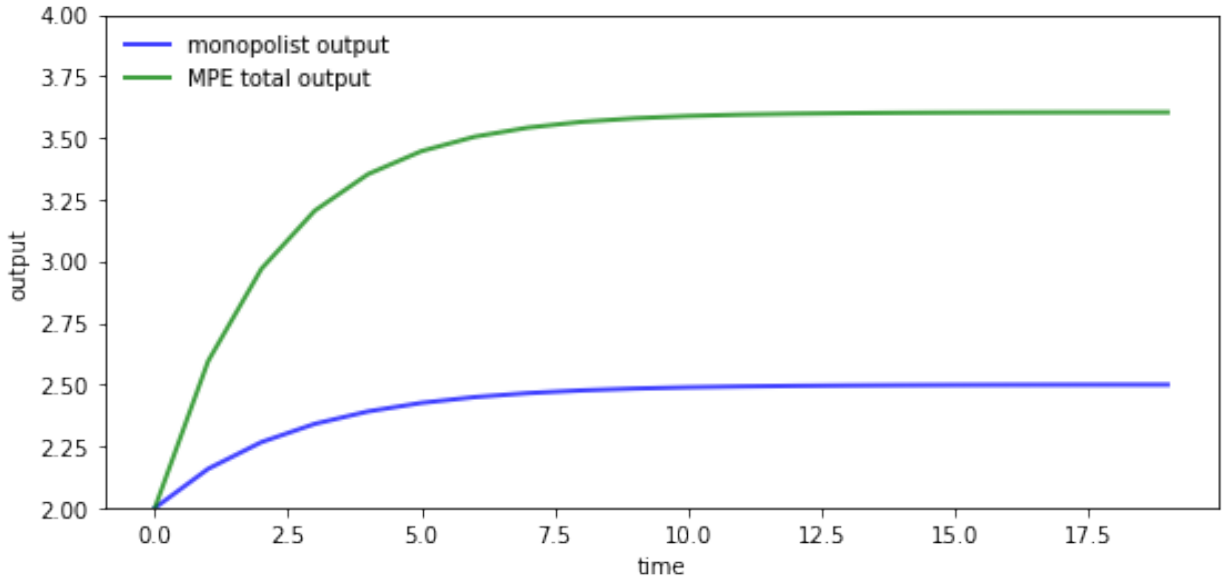
Exercise 110.5.1

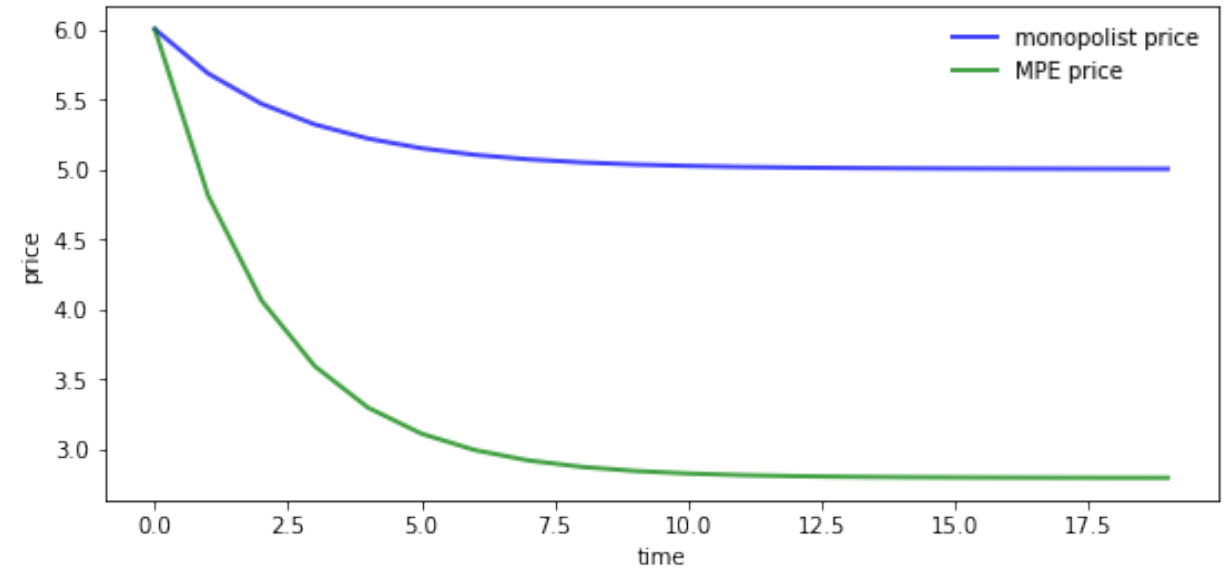
Replicate the *pair of figures* showing the comparison of output and prices for the monopolist and duopoly under MPE.

Parameters are as in duopoly_mpe.py and you can use that code to compute MPE policies under duopoly.

The optimal policy in the monopolist case can be computed using QuantEcon.py's LQ class.

Solution


monopolist output
MPE total output
output
time
4.00
3.75
3.50
3.25
3.00
2.75
2.50
2.25
2.00
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5


monopolist price
MPE price
price
time
6.0
5.5
5.0
4.5
4.0
3.5
3.0
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5

First, let's compute the duopoly MPE under the stated parameters

```
# == Parameters == #
a0 = 10.0
a1 = 2.0
β = 0.96
γ = 12.0

# == In LQ form == #
A  = np.eye(3)
B1 = np.array([[0.], [1.], [0.]])
B2 = np.array([[0.], [0.], [1.]])
R1 = [[      0.,      -a0/2,          0.],
      [-a0 / 2.,         a1,     a1 / 2.],
      [       0,    a1 / 2.,          0.]]

R2 = [[     0.,          0.,     -a0 / 2],
      [     0.,          0.,     a1 / 2.],
      [-a0 / 2,     a1 / 2.,          a1]]

Q1 = Q2 = γ
S1 = S2 = W1 = W2 = M1 = M2 = 0.0

# == Solve using QE's nnash function == #
F1, F2, P1, P2 = qe.nnash(A, B1, B2, R1, R2, Q1,
                          Q2, S1, S2, W1, W2, M1,
                          M2, beta=β)
```

Now we evaluate the time path of industry output and prices given initial condition $q_{10} = q_{20} = 1$.

```
AF = A - B1 @ F1 - B2 @ F2
n = 20
x = np.empty((3, n))
x[:, 0] = 1, 1, 1
for t in range(n-1):
    x[:, t+1] = AF @ x[:, t]
q1 = x[1, :]
q2 = x[2, :]
q = q1 + q2        # Total output, MPE
p = a0 - a1 * q    # Price, MPE
```

Next, let's have a look at the monopoly solution.

For the state and control, we take

$$x_t = q_t - \bar{q} \quad \text{and} \quad u_t = q_{t+1} - q_t$$

To convert to an LQ problem we set

$$R = a_1 \quad \text{and} \quad Q = \gamma$$

in the payoff function $x_t' R x_t + u_t' Q u_t$ and

$$A = B = 1$$

in the law of motion $x_{t+1} = Ax_t + Bu_t$.

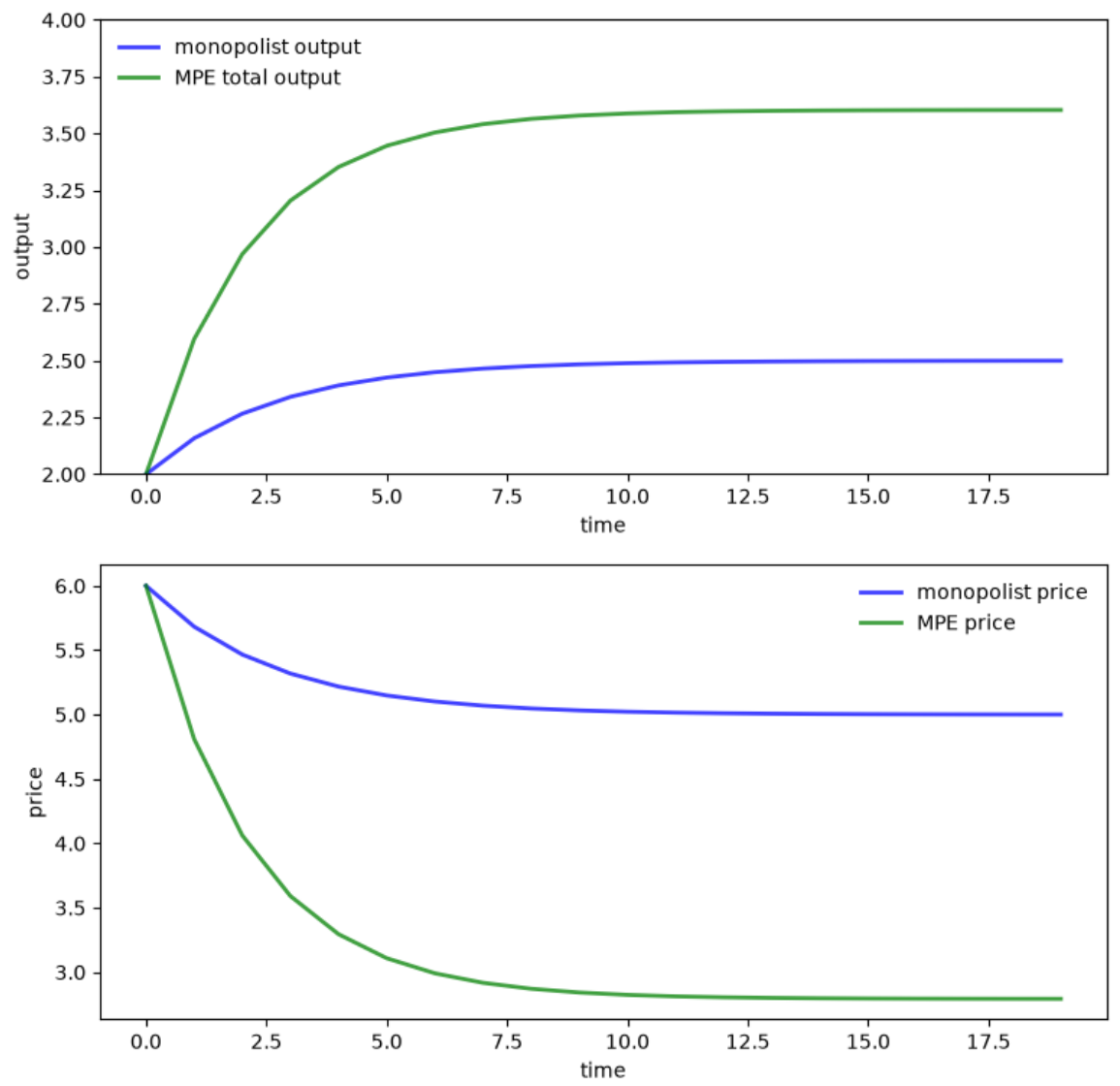
We solve for the optimal policy $u_t = -Fx_t$ and track the resulting dynamics of $\{q_t\}$, starting at $q_0 = 2.0$.

```
R = a1
Q = γ
```

```
A = B = 1
lq_alt = qe.LQ(Q, R, A, B, beta=β)
P, F, d = lq_alt.stationary_values()
q_bar = a0 / (2.0 * a1)
qm = np.empty(n)
qm[0] = 2
x0 = qm[0] - q_bar
x = x0
for i in range(1, n):
    x = A * x - B * F * x
    qm[i] = float(x.item()) + q_bar
pm = a0 - a1 * qm
```

Let's have a look at the different time paths

```
fig, axes = plt.subplots(2, 1, figsize=(9, 9))

ax = axes[0]
ax.plot(qm, 'b-', lw=2, alpha=0.75, label='monopolist output')
ax.plot(q, 'g-', lw=2, alpha=0.75, label='MPE total output')
ax.set(ylabel="output", xlabel="time", ylim=(2, 4))
ax.legend(loc='upper left', frameon=0)

ax = axes[1]
ax.plot(pm, 'b-', lw=2, alpha=0.75, label='monopolist price')
ax.plot(p, 'g-', lw=2, alpha=0.75, label='MPE price')
ax.set(ylabel="price", xlabel="time")
ax.legend(loc='upper right', frameon=0)
plt.show()
```



Exercise 110.5.2

In this exercise, we consider a slightly more sophisticated duopoly problem.

It takes the form of infinite horizon linear-quadratic game proposed by Judd [Judd, 1990].

Two firms set prices and quantities of two goods interrelated through their demand curves.

Relevant variables are defined as follows:

- I_{it} = inventories of firm i at beginning of t
- q_{it} = production of firm i during period t
- p_{it} = price charged by firm i during period t
- S_{it} = sales made by firm i during period t
- E_{it} = costs of production of firm i during period t

- C_{it} = costs of carrying inventories for firm i during t

The firms' cost functions are

- $C_{it} = c_{i1} + c_{i2} I_{it} + 0.5 c_{i3} I_{it}^2$
- $E_{it} = e_{i1} + e_{i2} q_{it} + 0.5 e_{i3} q_{it}^2$ where e_{ij}, c_{ij} are positive scalars

Inventories obey the laws of motion

$$I_{i,t+1} = (1 - \delta) I_{it} + q_{it} - S_{it}$$

Demand is governed by the linear schedule

$$S_t = D p_{it} + b$$

where

- $S_t = \begin{bmatrix} S_{1t} & S_{2t} \end{bmatrix}'$
- D is a 2×2 negative definite matrix and
- b is a vector of constants

Firm i maximizes the undiscounted sum

$$\lim_{T \to \infty} \frac{1}{T} \sum_{t=0}^{T} (p_{it} S_{it} - E_{it} - C_{it})$$

We can convert this to a linear-quadratic problem by taking

$$u_{it} = \begin{bmatrix} p_{it} \\ q_{it} \end{bmatrix} \quad \text{and} \quad x_t = \begin{bmatrix} I_{1t} \\ I_{2t} \\ 1 \end{bmatrix}$$

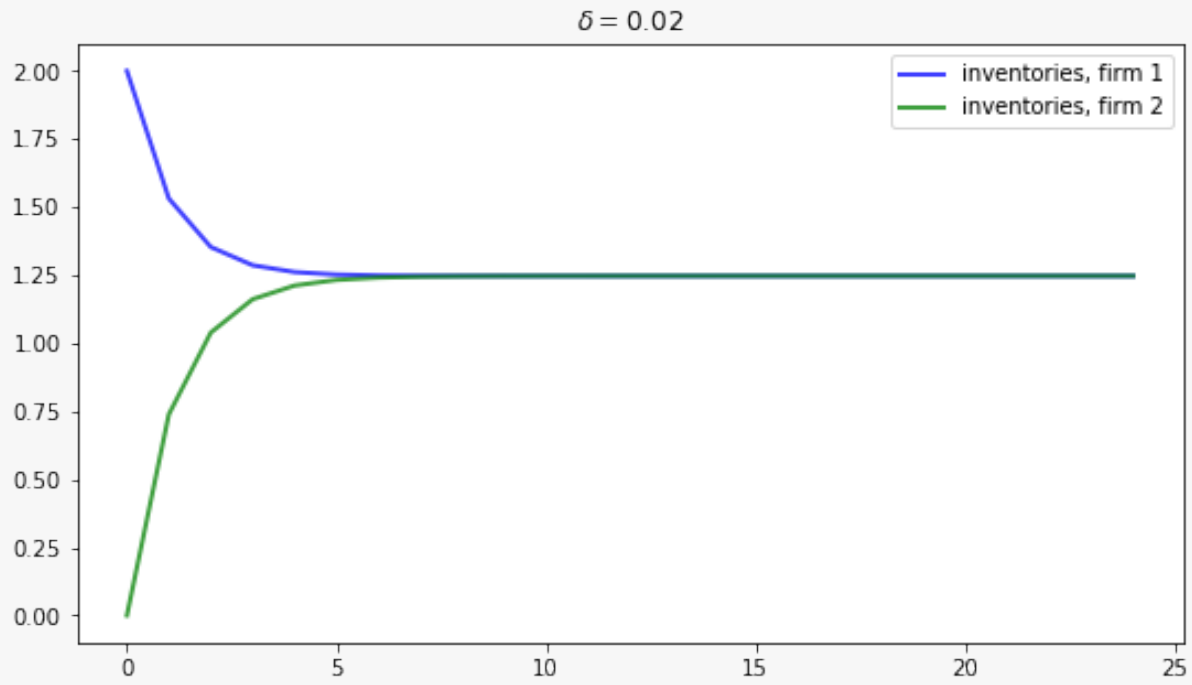
Decision rules for price and quantity take the form $u_{it} = -F_i x_t$.

The Markov perfect equilibrium of Judd's model can be computed by filling in the matrices appropriately.

The exercise is to calculate these matrices and compute the following figures.

The first figure shows the dynamics of inventories for each firm when the parameters are

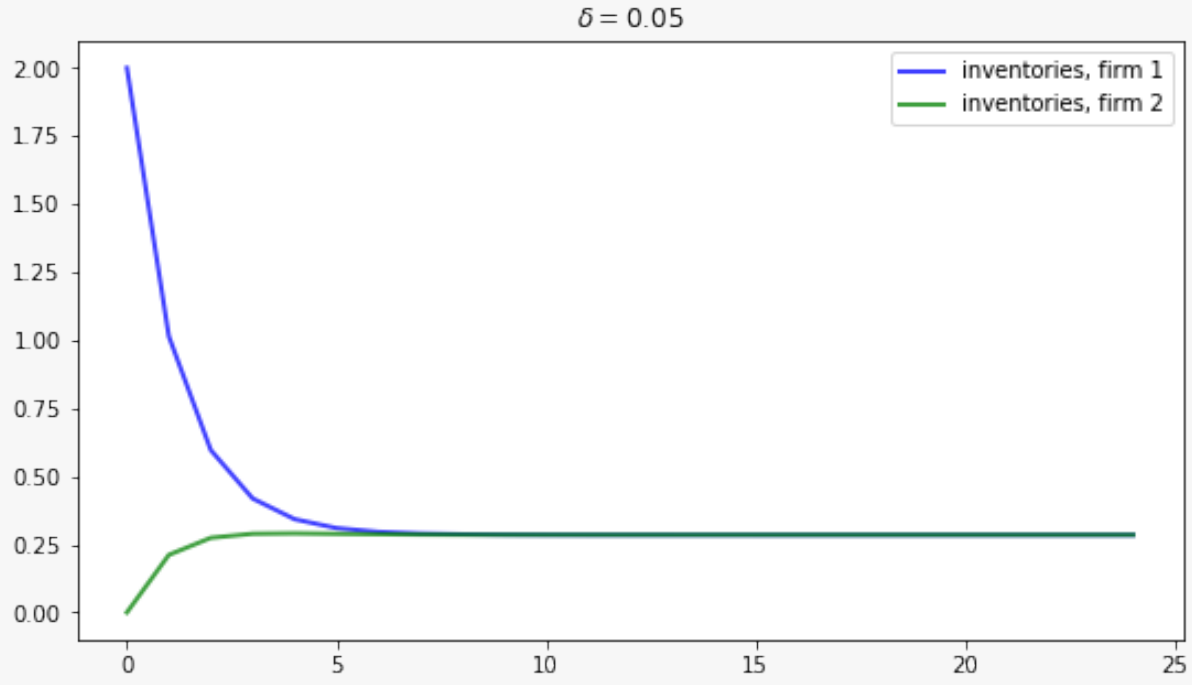
```
δ = 0.02
D = np.array([[-1, 0.5], [0.5, -1]])
b = np.array([25, 25])
c1 = c2 = np.array([1, -2, 1])
e1 = e2 = np.array([10, 10, 3])
```



Inventories trend to a common steady state.

If we increase the depreciation rate to $\delta = 0.05$, then we expect steady state inventories to fall.

This is indeed the case, as the next figure shows



In this exercise, reproduce the figure when $\delta = 0.02$.

Solution

We treat the case $\delta = 0.02$

```
δ = 0.02
D = np.array([[-1, 0.5], [0.5, -1]])
b = np.array([25, 25])
c1 = c2 = np.array([1, -2, 1])
e1 = e2 = np.array([10, 10, 3])

δ_1 = 1 - δ
```

Recalling that the control and state are

$$u_{it} = \begin{bmatrix} p_{it} \\ q_{it} \end{bmatrix} \quad \text{and} \quad x_t = \begin{bmatrix} I_{1t} \\ I_{2t} \\ 1 \end{bmatrix}$$

we set up the matrices as follows:

```
# ==  Create matrices needed to compute the Nash feedback equilibrium == #

A = np.array([[δ_1,      0,     -δ_1 * b[0]],
              [  0,    δ_1,     -δ_1 * b[1]],
              [  0,      0,               1]])

B1 = δ_1 * np.array([[1, -D[0, 0]],
                     [0, -D[1, 0]],
                     [0,       0]])
B2 = δ_1 * np.array([[0, -D[0, 1]],
                     [1, -D[1, 1]],
                     [0,       0]])

R1 = -np.array([[0.5 * c1[2],      0,     0.5 * c1[1]],
                [          0,      0,               0],
                [0.5 * c1[1],      0,          c1[0]]])
R2 = -np.array([[0,                0,               0],
                [0,      0.5 * c2[2],     0.5 * c2[1]],
                [0,      0.5 * c2[1],          c2[0]]])

Q1 = np.array([[-0.5 * e1[2], 0], [0, D[0, 0]]])
Q2 = np.array([[-0.5 * e2[2], 0], [0, D[1, 1]]])

S1 = np.zeros((2, 2))
S2 = np.copy(S1)

W1 = np.array([[           0,              0],
               [           0,              0],
               [-0.5 * e1[1],     b[0] / 2.]])
W2 = np.array([[           0,              0],
               [           0,              0],
               [-0.5 * e2[1],     b[1] / 2.]])

M1 = np.array([[0, 0], [0, D[0, 1] / 2.]])
M2 = np.copy(M1)
```

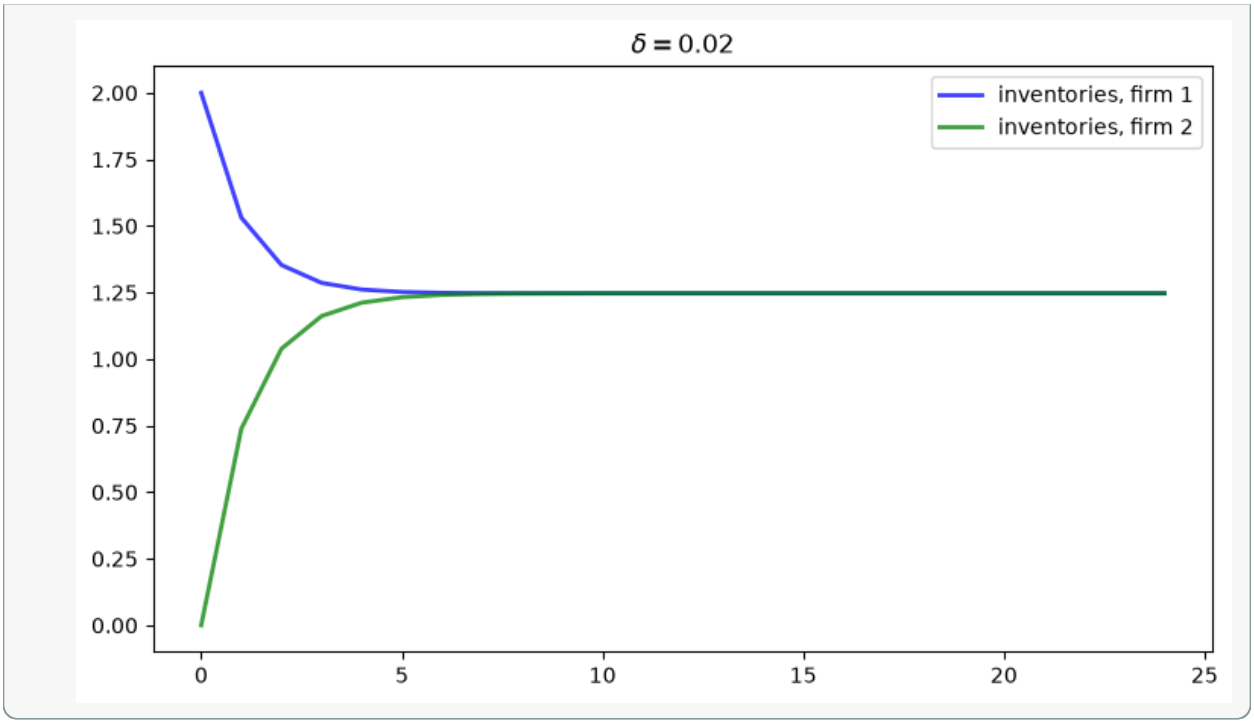
We can now compute the equilibrium using `qe.nnash`

```
F1, F2, P1, P2 = qe.nnash(A, B1, B2, R1,
                          R2, Q1, Q2, S1,
                          S2, W1, W2, M1, M2)

print("\nFirm 1's feedback rule:\n")
print(F1)

print("\nFirm 2's feedback rule:\n")
print(F2)
```

```
Firm 1's feedback rule:

[[ 2.43666582e-01  2.72360627e-02 -6.82788293e+00]
 [ 3.92370734e-01  1.39696451e-01 -3.77341073e+01]]

Firm 2's feedback rule:

[[ 2.72360627e-02  2.43666582e-01 -6.82788293e+00]
 [ 1.39696451e-01  3.92370734e-01 -3.77341073e+01]]
```

Now let's look at the dynamics of inventories, and reproduce the graph corresponding to $\delta = 0.02$

```
AF = A - B1 @ F1 - B2 @ F2
n = 25
x = np.empty((3, n))
x[:, 0] = 2, 0, 1
for t in range(n-1):
    x[:, t+1] = AF @ x[:, t]
I1 = x[0, :]
I2 = x[1, :]
fig, ax = plt.subplots(figsize=(9, 5))
ax.plot(I1, 'b-', lw=2, alpha=0.75, label='inventories, firm 1')
ax.plot(I2, 'g-', lw=2, alpha=0.75, label='inventories, firm 2')
ax.set_title(rf'$\delta = {δ}$')
ax.legend()
plt.show()
```


$\delta = 0.02$
inventories, firm 1
inventories, firm 2
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
0
5
10
15
20
25

CHAPTER

ONE

UNCERTAINTY TRAPS

Contents

- *Uncertainty Traps*
  - *Overview*
  - *The Model*
  - *Implementation*
  - *Results*
  - *Exercises*

111.1 Overview

In this lecture, we study a simplified version of an uncertainty traps model of Fajgelbaum, Schaal and Taschereau-Dumouchel [Fajgelbaum *et al.*, 2015].

The model features self-reinforcing uncertainty that has big impacts on economic activity.

In the model,

- Fundamentals vary stochastically and are not fully observable.
- At any moment there are both active and inactive entrepreneurs; only active entrepreneurs produce.
- Agents – active and inactive entrepreneurs – have beliefs about the fundamentals expressed as probability distributions.
- Greater uncertainty means greater dispersions of these distributions.
- Entrepreneurs are risk-averse and hence less inclined to be active when uncertainty is high.
- The output of active entrepreneurs is observable, supplying a noisy signal that helps everyone inside the model infer fundamentals.
- Entrepreneurs update their beliefs about fundamentals using Bayes' Law, implemented via *Kalman filtering*.

Uncertainty traps emerge because:

- High uncertainty discourages entrepreneurs from becoming active.
- A low level of participation – i.e., a smaller number of active entrepreneurs – diminishes the flow of information about fundamentals.

- Less information translates to higher uncertainty, further discouraging entrepreneurs from choosing to be active, and so on.

Uncertainty traps stem from a positive externality: high aggregate economic activity levels generates valuable information.

Let's start with some standard imports:

```
import matplotlib.pyplot as plt
import numpy as np
```

111.2 The Model

The original model described in [Fajgelbaum *et al.*, 2015] has many interesting moving parts.

Here we examine a simplified version that nonetheless captures many of the key ideas.

111.2.1 Fundamentals

The evolution of the fundamental process $\{\theta_t\}$ is given by

$$\theta_{t+1} = \rho\theta_t + \sigma_\theta w_{t+1}$$

where

- $\sigma_\theta > 0$ and $0 < \rho < 1$
- $\{w_t\}$ is IID and standard normal

The random variable θ_t is not observable at any time.

111.2.2 Output

There is a total $\bar{M}$ of risk-averse entrepreneurs.

Output of the m-th entrepreneur, conditional on being active in the market at time t, is equal to

$$x_m = \theta + \epsilon_m \quad \text{where} \quad \epsilon_m \sim N\left(0, \gamma_x^{-1}\right) \tag{111.1}$$

Here the time subscript has been dropped to simplify notation.

The inverse of the shock variance, γ_x, is called the shock's **precision**.

The higher is the precision, the more informative x_m is about the fundamental.

Output shocks are independent across time and firms.

111.2.3 Information and Beliefs

All entrepreneurs start with identical beliefs about θ_0.

Signals are publicly observable and hence all agents have identical beliefs always.

Dropping time subscripts, beliefs for current θ are represented by the normal distribution $N(\mu, \gamma^{-1})$.

Here γ is the precision of beliefs; its inverse is the degree of uncertainty.

These parameters are updated by Kalman filtering.

Let

- $\mathbb{M} \subset \{1, \ldots, \bar{M}\}$ denote the set of currently active firms.
- $M := |\mathbb{M}|$ denote the number of currently active firms.
- X be the average output $\frac{1}{M}\sum_{m \in \mathbb{M}} x_m$ of the active firms.

With this notation and primes for next period values, we can write the updating of the mean and precision via

$$\mu' = \rho \frac{\gamma\mu + M\gamma_x X}{\gamma + M\gamma_x} \tag{111.2}$$

$$\gamma' = \left(\frac{\rho^2}{\gamma + M\gamma_x} + \sigma_\theta^2 \right)^{-1} \tag{111.3}$$

These are standard Kalman filtering results applied to the current setting.

Exercise 1 provides more details on how (111.2) and (111.3) are derived and then asks you to fill in remaining steps.

The next figure plots the law of motion for the precision in (111.3) as a 45 degree diagram, with one curve for each $M \in \{0, \ldots, 6\}$.

The other parameter values are $\rho = 0.99, \gamma_x = 0.5, \sigma_\theta = 0.5$

Points where the curves hit the 45 degree lines are long-run steady states for precision for different values of M.

Thus, if one of these values for M remains fixed, a corresponding steady state is the equilibrium level of precision

- high values of M correspond to greater information about the fundamental, and hence more precision in steady state
- low values of M correspond to less information and more uncertainty in steady state

In practice, as we'll see, the number of active firms fluctuates stochastically.

111.2.4 Participation

Omitting time subscripts once more, entrepreneurs enter the market in the current period if

$$\mathbb{E}[u(x_m - F_m)] > c \tag{111.4}$$

Here

- the mathematical expectation of x_m is based on (111.1) and beliefs $N(\mu, \gamma^{-1})$ for θ
- F_m is a stochastic but pre-visible fixed cost, independent across time and firms
- c is a constant reflecting opportunity costs

The statement that F_m is pre-visible means that it is realized at the start of the period and treated as a constant in (111.4).

The utility function has the constant absolute risk aversion form

$$u(x) = \frac{1}{a}\left(1 - \exp(-ax)\right) \tag{111.5}$$

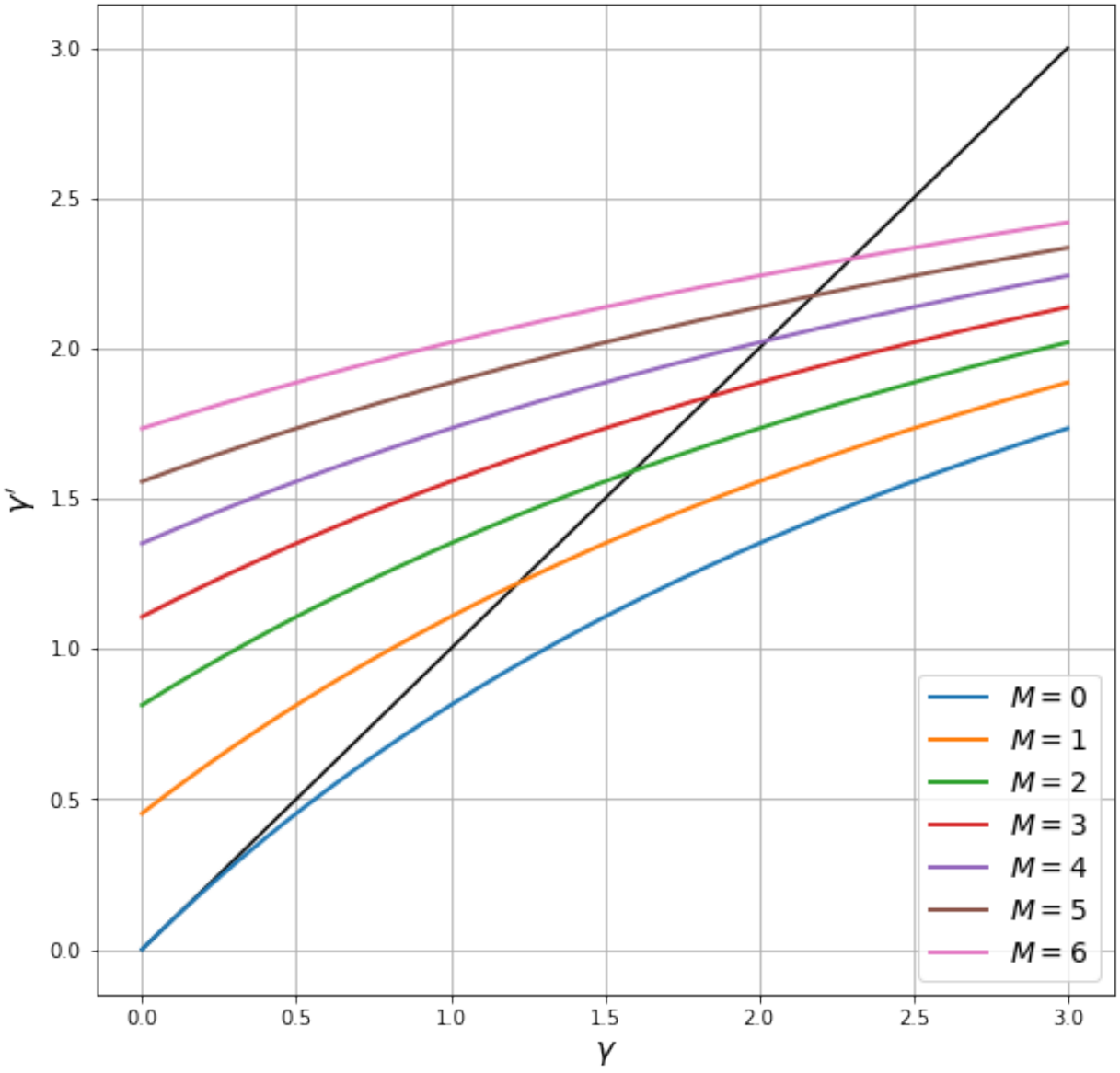
where a is a positive parameter.

Combining (111.4) and (111.5), entrepreneur m participates in the market (or is said to be active) when

$$\frac{1}{a}\left\{1 - \mathbb{E}[\exp\left(-a(\theta + \epsilon_m - F_m)\right)]\right\} > c$$

Using standard formulas for expectations of lognormal random variables, this is equivalent to the condition

$$\psi(\mu, \gamma, F_m) := \frac{1}{a}\left(1 - \exp\left(-a\mu + aF_m + \frac{a^2\left(\frac{1}{\gamma} + \frac{1}{\gamma_x}\right)}{2}\right)\right) - c > 0 \tag{111.6}$$


3.0
2.5
2.0
1.5
1.0
0.5
0.0
y'
0.0
0.5
1.0
1.5
2.0
2.5
3.0
y
$M = 0$
$M = 1$
$M = 2$
$M = 3$
$M = 4$
$M = 5$
$M = 6$

111.3 Implementation

We want to simulate this economy.

As a first step, let's put together a class that bundles

- the parameters, the current value of θ and the current values of the two belief parameters μ and γ
- methods to update θ, μ and γ, as well as to determine the number of active firms and their outputs

The updating methods follow the laws of motion for θ, μ and γ given above.

The method to evaluate the number of active firms generates $F_1, \ldots, F_{\bar{M}}$ and tests condition (111.6) for each firm.

The **init** method encodes as default values the parameters we'll use in the simulations below

```
class UncertaintyTrapEcon:

    def __init__(self,
                 a=1.5,           # Risk aversion
                 γ_x=0.5,         # Production shock precision
                 ρ=0.99,          # Correlation coefficient for θ
                 σ_θ=0.5,         # Standard dev of θ shock
                 num_firms=100,   # Number of firms
                 σ_F=1.5,         # Standard dev of fixed costs
                 c=-420,          # External opportunity cost
                 μ_init=0,        # Initial value for μ
                 γ_init=4,        # Initial value for γ
                 θ_init=0):       # Initial value for θ

        # == Record values == #
        self.a, self.γ_x, self.ρ, self.σ_θ = a, γ_x, ρ, σ_θ
        self.num_firms, self.σ_F, self.c, = num_firms, σ_F, c
        self.σ_x = np.sqrt(1/γ_x)

        # == Initialize states == #
        self.γ, self.μ, self.θ = γ_init, μ_init, θ_init

    def ψ(self, F):
        temp1 = -self.a * (self.μ - F)
        temp2 = self.a**2 * (1/self.γ + 1/self.γ_x) / 2
        return (1 / self.a) * (1 - np.exp(temp1 + temp2)) - self.c

    def update_beliefs(self, X, M):
        """
        Update beliefs (μ, γ) based on aggregates X and M.
        """
        # Simplify names
        γ_x, ρ, σ_θ = self.γ_x, self.ρ, self.σ_θ
        # Update μ
        temp1 = ρ * (self.γ * self.μ + M * γ_x * X)
        temp2 = self.γ + M * γ_x
        self.μ = temp1 / temp2
        # Update γ
        self.γ = 1 / (ρ**2 / (self.γ + M * γ_x) + σ_θ**2)

    def update_θ(self, w):
        """
        Update the fundamental state θ given shock w.
```

(continues on next page)

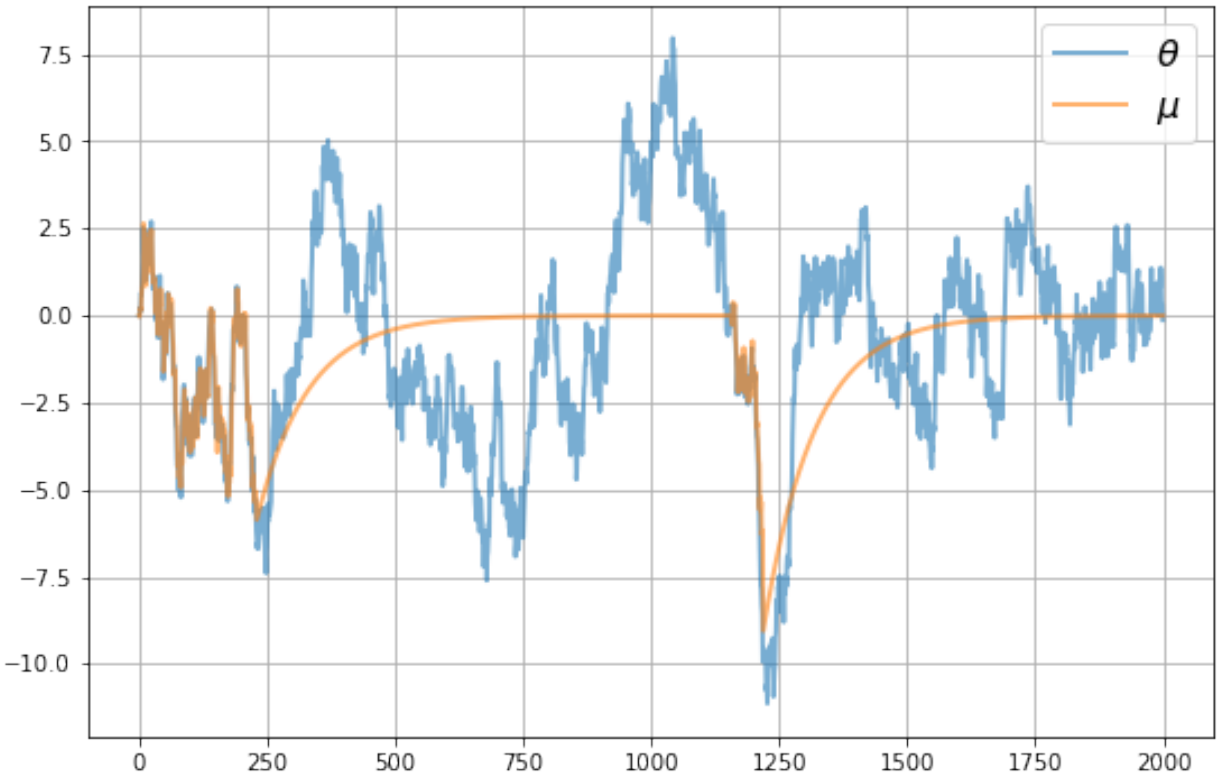
(continued from previous page)

```
        """
        self.θ = self.ρ * self.θ + self.σ_θ * w

    def gen_aggregates(self):
        """
        Generate aggregates based on current beliefs (μ, γ). This
        is a simulation step that depends on the draws for F.
        """
        F_vals = self.σ_F * np.random.randn(self.num_firms)
        M = np.sum(self.ψ(F_vals) > 0)  # Counts number of active firms
        if M > 0:
            x_vals = self.θ + self.σ_x * np.random.randn(M)
            X = x_vals.mean()
        else:
            X = 0
        return X, M
```

In the results below we use this code to simulate time series for the major variables.

111.4 Results

Let's look first at the dynamics of μ, which the agents use to track θ



We see that μ tracks θ well when there are sufficient firms in the market.

However, there are times when μ tracks θ poorly due to insufficient information.

These are episodes where the uncertainty traps take hold.

During these episodes

- precision is low and uncertainty is high
- few firms are in the market

To get a clearer idea of the dynamics, let's look at all the main time series at once, for a given set of shocks

Notice how the traps only take hold after a sequence of bad draws for the fundamental.

Thus, the model gives us a **propagation mechanism** that maps bad random draws into long downturns in economic activity.

111.5 Exercises

Exercise 111.5.1

Fill in the details behind (111.2) and (111.3) based on the following standard result (see, e.g., p. 24 of [Young and Smith, 2005]).

Fact Let $\mathbf{x} = (x_1, \ldots, x_M)$ be a vector of IID draws from common distribution $N(\theta, 1/\gamma_x)$ and let $\bar{x}$ be the sample mean. If γ_x is known and the prior for θ is $N(\mu, 1/\gamma)$, then the posterior distribution of θ given $\mathbf{x}$ is

$$\pi(\theta \,|\, \mathbf{x}) = N(\mu_0, 1/\gamma_0)$$

where

$$\mu_0 = \frac{\mu\gamma + M\bar{x}\gamma_x}{\gamma + M\gamma_x} \quad \text{and} \quad \gamma_0 = \gamma + M\gamma_x$$

Solution

This exercise asked you to validate the laws of motion for γ and μ given in the lecture, based on the stated result about Bayesian updating in a scalar Gaussian setting. The stated result tells us that after observing average output X of the M firms, our posterior beliefs will be

$$N(\mu_0, 1/\gamma_0)$$

where

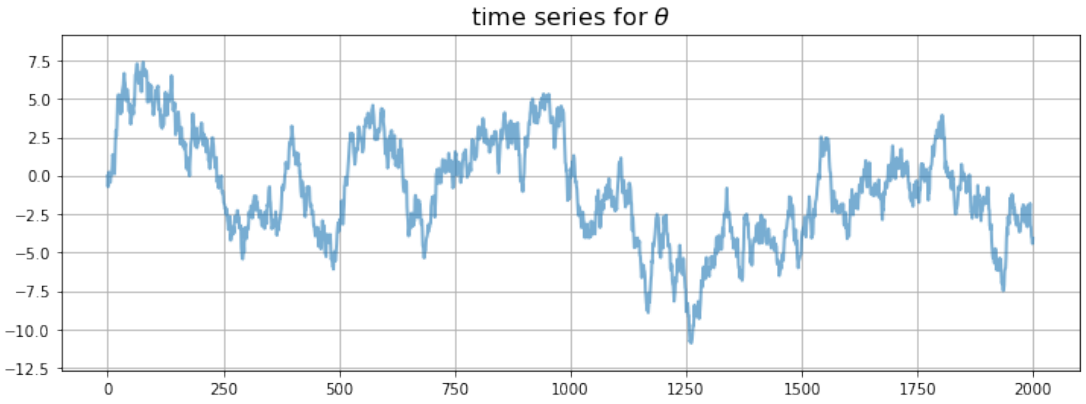
$$\mu_0 = \frac{\mu\gamma + MX\gamma_x}{\gamma + M\gamma_x} \quad \text{and} \quad \gamma_0 = \gamma + M\gamma_x$$

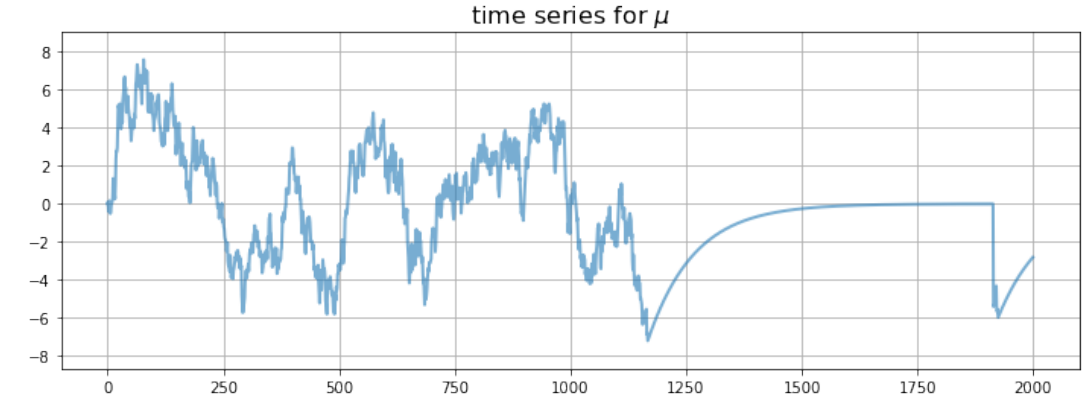
If we take a random variable θ with this distribution and then evaluate the distribution of $\rho\theta + \sigma_\theta w$ where w is independent and standard normal, we get the expressions for μ' and γ' given in the lecture.

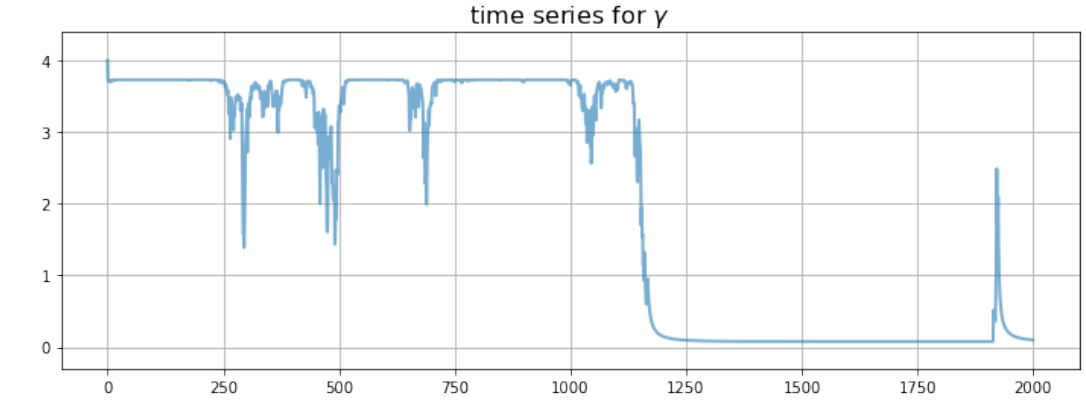
Exercise 111.5.2

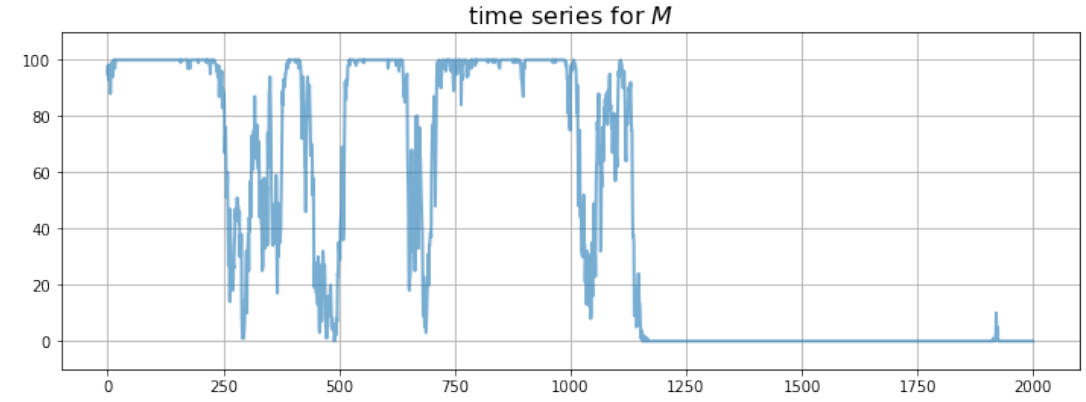
Modulo randomness, replicate the simulation figures shown above.

- Use the parameter values listed as defaults in the **init** method of the UncertaintyTrapEcon class.


time series for θ
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
−12.5
0
250
500
750
1000
1250
1500
1750
2000


time series for μ
8
6
4
2
0
−2
−4
−6
−8
0
250
500
750
1000
1250
1500
1750
2000


time series for γ
4
3
2
1
0
0
250
500
750
1000
1250
1500
1750
2000


time series for M
100
80
60
40
20
0
0
250
500
750
1000
1250
1500
1750
2000

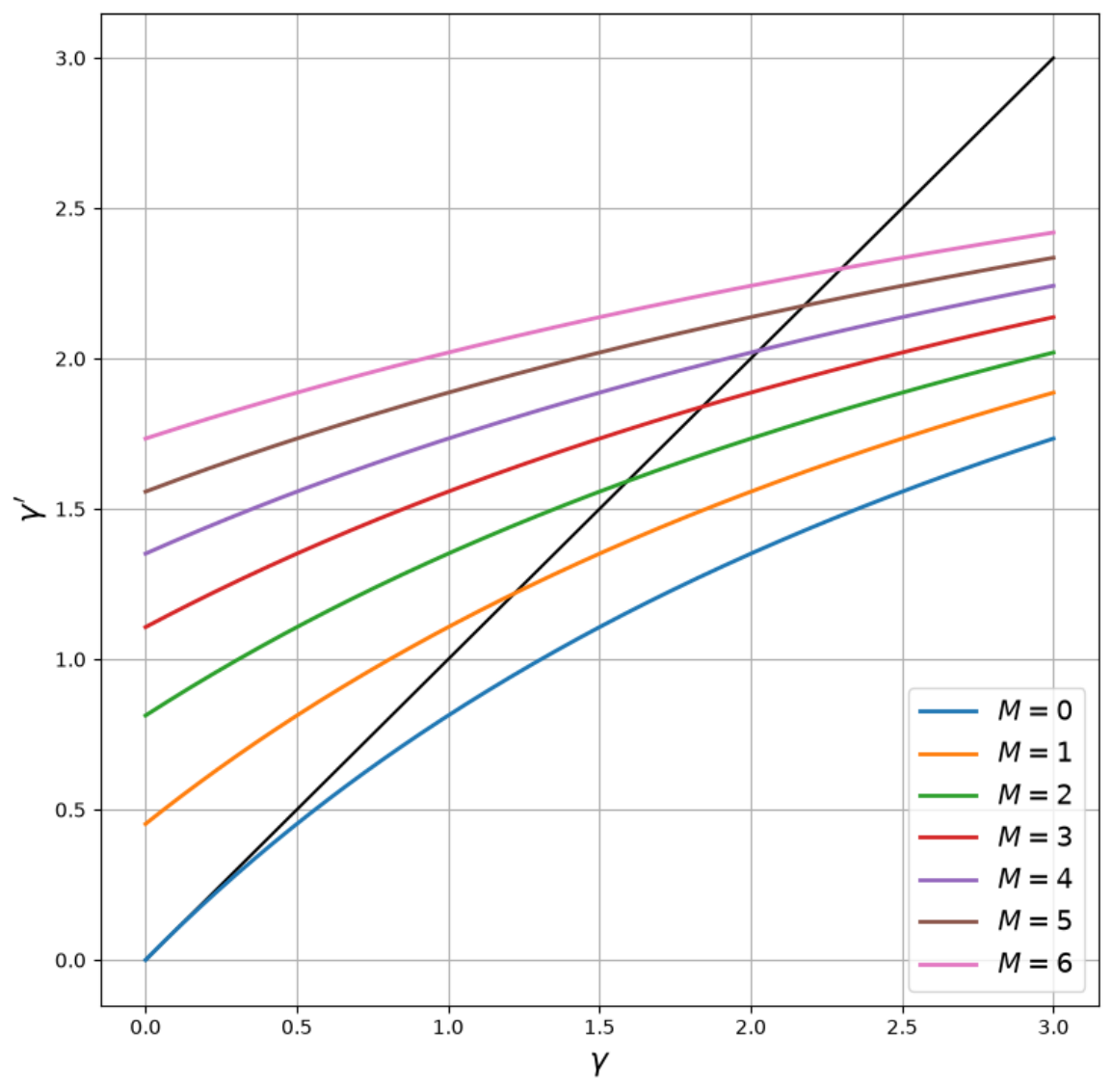
Solution

First, let's replicate the plot that illustrates the law of motion for precision, which is

$$\gamma_{t+1} = \left(\frac{\rho^2}{\gamma_t + M\gamma_x} + \sigma_\theta^2 \right)^{-1}$$

Here M is the number of active firms. The next figure plots γ_{t+1} against γ_t on a 45 degree diagram for different values of M

```
econ = UncertaintyTrapEcon()
ρ, σ_θ, γ_x = econ.ρ, econ.σ_θ, econ.γ_x    # Simplify names
γ = np.linspace(1e-10, 3, 200)              # γ grid
fig, ax = plt.subplots(figsize=(9, 9))
ax.plot(γ, γ, 'k-')                         # 45 degree line

for M in range(7):
    γ_next = 1 / (ρ**2 / (γ + M * γ_x) + σ_θ**2)
    label_string = f"$M = {M}$"
    ax.plot(γ, γ_next, lw=2, label=label_string)
ax.legend(loc='lower right', fontsize=14)
ax.set_xlabel(r'$\gamma$', fontsize=16)
ax.set_ylabel(r"$\gamma'$", fontsize=16)
ax.grid()
plt.show()
```



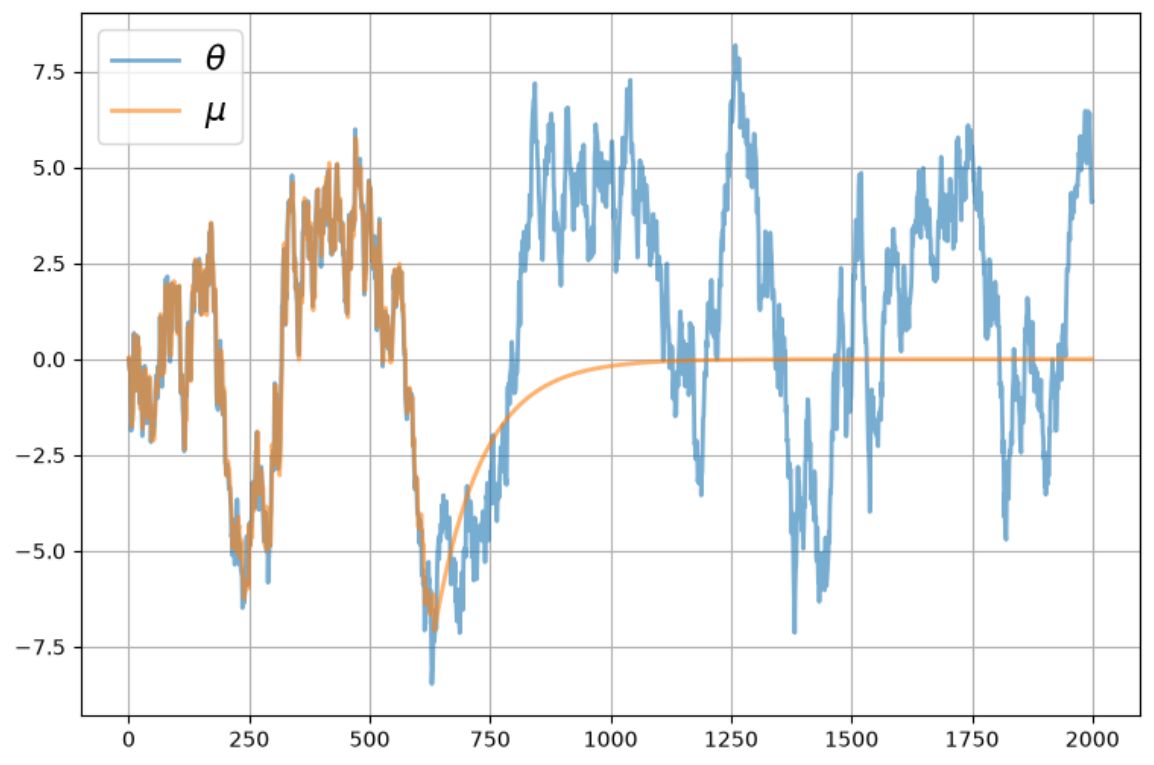
The points where the curves hit the 45 degree lines are the long-run steady states corresponding to each M, if that value of M was to remain fixed. As the number of firms falls, so does the long-run steady state of precision.

Next let's generate time series for beliefs and the aggregates – that is, the number of active firms and average output

```
sim_length=2000

μ_vec = np.empty(sim_length)
θ_vec = np.empty(sim_length)
γ_vec = np.empty(sim_length)
X_vec = np.empty(sim_length)
M_vec = np.empty(sim_length)

μ_vec[0] = econ.μ
γ_vec[0] = econ.γ
θ_vec[0] = 0

w_shocks = np.random.randn(sim_length)

for t in range(sim_length-1):
    X, M = econ.gen_aggregates()
    X_vec[t] = X
    M_vec[t] = M

    econ.update_beliefs(X, M)
    econ.update_θ(w_shocks[t])

    μ_vec[t+1] = econ.μ
    γ_vec[t+1] = econ.γ
    θ_vec[t+1] = econ.θ

# Record final values of aggregates
X, M = econ.gen_aggregates()
X_vec[-1] = X
M_vec[-1] = M
```

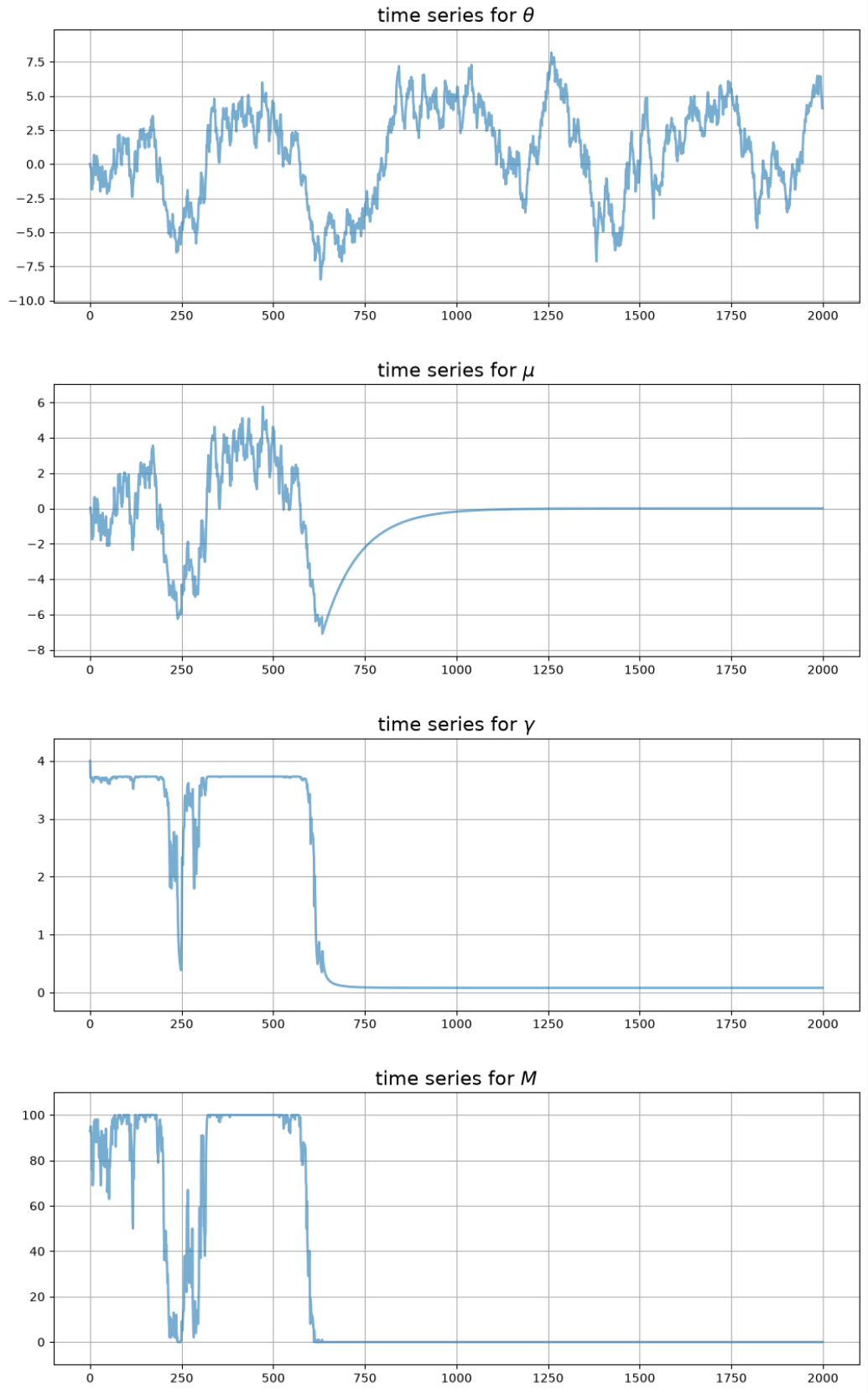
First, let's see how well μ tracks θ in these simulations

```
fig, ax = plt.subplots(figsize=(9, 6))
ax.plot(range(sim_length), θ_vec, alpha=0.6, lw=2, label=r"$\theta$")
ax.plot(range(sim_length), μ_vec, alpha=0.6, lw=2, label=r"$\mu$")
ax.legend(fontsize=16)
ax.grid()
plt.show()
```



Now let's plot the whole thing together

```
fig, axes = plt.subplots(4, 1, figsize=(12, 20))
# Add some spacing
fig.subplots_adjust(hspace=0.3)

series = (θ_vec, μ_vec, γ_vec, M_vec)
names = r'$\theta$', r'$\mu$', r'$\gamma$', r'$M$'

for ax, vals, name in zip(axes, series, names):
    # Determine suitable y limits
    s_max, s_min = max(vals), min(vals)
    s_range = s_max - s_min
    y_max = s_max + s_range * 0.1
    y_min = s_min - s_range * 0.1
    ax.set_ylim(y_min, y_max)
    # Plot series
    ax.plot(range(sim_length), vals, alpha=0.6, lw=2)
    ax.set_title(f"time series for {name}", fontsize=16)
    ax.grid()

plt.show()
```


time series for θ
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
0
250
500
750
1000
1250
1500
1750
2000
time series for μ
6
4
2
0
−2
−4
−6
−8
0
250
500
750
1000
1250
1500
1750
2000
time series for γ
4
3
2
1
0
0
250
500
750
1000
1250
1500
1750
2000
time series for M
100
80
60
40
20
0
0
250
500
750
1000
1250
1500
1750
2000

If you run the code above you'll get different plots, of course.

Try experimenting with different parameters to see the effects on the time series.

(It would also be interesting to experiment with non-Gaussian distributions for the shocks, but this is a big exercise since it takes us outside the world of the standard Kalman filter)

CHAPTER

TWO

THE AIYAGARI MODEL

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

Contents

- *The Aiyagari Model*
  - *Overview*
  - *The Economy*
  - *Implementation*
  - *Exercises*

In addition to what's included in base Anaconda, we need to install JAX

```
!pip install quantecon jax
```

112.1 Overview

In this lecture, we describe the structure of a class of models that build on work by Truman Bewley [Bewley, 1977].

We begin by discussing an example of a Bewley model due to Rao Aiyagari [Aiyagari, 1994].

The model features

- heterogeneous agents
- a single exogenous vehicle for borrowing and lending
- limits on amounts individual agents may borrow

The Aiyagari model has been used to investigate many topics, including

- precautionary savings and the effect of liquidity constraints [Aiyagari, 1994]
- risk sharing and asset pricing [Heaton and Lucas, 1996]
- the shape of the wealth distribution [Benhabib *et al.*, 2015]
- etc., etc., etc.

112.1.1 Preliminaries

We use the following imports:

```
import quantecon as qe
import matplotlib.pyplot as plt
import jax
import jax.numpy as jnp
from typing import NamedTuple
from scipy.optimize import bisect
```

We will use 64-bit floats with JAX in order to increase precision.

```
jax.config.update("jax_enable_x64", True)
```

We will use the following function to compute stationary distributions of stochastic matrices (for a reference to the algorithm, see p. 88 of Economic Dynamics).

```
@jax.jit
def compute_stationary(P):
    n = P.shape[0]
    I = jnp.identity(n)
    O = jnp.ones((n, n))
    A = I - jnp.transpose(P) + O
    return jnp.linalg.solve(A, jnp.ones(n))
```

112.1.2 References

The primary reference for this lecture is [Aiyagari, 1994].

A textbook treatment is available in chapter 18 of [Ljungqvist and Sargent, 2018].

A continuous time version of the model by SeHyoun Ahn and Benjamin Moll can be found here.

112.2 The Economy

112.2.1 Households

Infinitely lived households / consumers face idiosyncratic income shocks.

A unit interval of *ex-ante* identical households face a common borrowing constraint.

The savings problem faced by a typical household is

$$\max \mathbb{E} \sum_{t=0}^{\infty} \beta^t u(c_t)$$

subject to

$$a_{t+1} + c_t \leq wz_t + (1+r)a_t \quad c_t \geq 0, \quad \text{and} \quad a_t \geq -B$$

where

- c_t is current consumption
- a_t is assets
- z_t is an exogenous component of labor income capturing stochastic unemployment risk, etc.
- w is a wage rate
- r is a net interest rate
- B is the maximum amount that the agent is allowed to borrow

The exogenous process $\{z_t\}$ follows a finite state Markov chain with given stochastic matrix P.

The wage and interest rate are fixed over time.

In this simple version of the model, households supply labor inelastically because they do not value leisure.

112.2.2 Firms

Firms produce output by hiring capital and labor.

Firms act competitively and face constant returns to scale.

Since returns to scale are constant, the number of firms does not matter.

Hence we can consider a single (but nonetheless competitive) representative firm.

The firm's output is

$$Y = AK^\alpha N^{1-\alpha}$$

where

- A and α are parameters with $A > 0$ and $\alpha \in (0, 1)$
- K is aggregate capital
- N is total labor supply (which is constant in this simple version of the model)

The firm's problem is

$$\max_{K,N} \left\{ AK^\alpha N^{1-\alpha} - (r+\delta)K - wN \right\}$$

The parameter δ is the depreciation rate.

These parameters are stored in the following namedtuple:

```
class Firm(NamedTuple):
    A: float = 1.0    # Total factor productivity
    N: float = 1.0    # Total labor supply
    α: float = 0.33   # Capital share
    δ: float = 0.05   # Depreciation rate
```

From the first-order condition with respect to capital, the firm's inverse demand for capital is

$$r = A\alpha \left(\frac{N}{K} \right)^{1-\alpha} - \delta \tag{112.1}$$

```
def r_given_k(K, firm):
    """
    Inverse demand curve for capital. The interest rate associated with a
    given demand for capital K.
    """
    A, N, α, δ = firm
    return A * α * (N / K)**(1 - α) - δ
```

Using this expression and the firm's first-order condition for labor, we can pin down the equilibrium wage rate as a function of r as

$$w(r) = A(1-\alpha)(A\alpha/(r+\delta))^{\alpha/(1-\alpha)} \tag{112.2}$$

```
def r_to_w(r, firm):
    """
    Equilibrium wages associated with a given interest rate r.
    """
    A, N, α, δ = firm
    return A * (1 - α) * (A * α / (r + δ))**(α / (1 - α))
```

112.2.3 Equilibrium

We construct a **stationary rational expectations equilibrium (SREE)**.

In such an equilibrium

- prices induce behavior that generates aggregate quantities consistent with the prices
- aggregate quantities and prices are constant over time

In more detail, an SREE lists a set of prices, savings and production policies such that

- households want to choose the specified savings policies taking the prices as given
- firms maximize profits taking the same prices as given
- the resulting aggregate quantities are consistent with the prices; in particular, the demand for capital equals the supply
- aggregate quantities (defined as cross-sectional averages) are constant

112.3 Implementation

Let's look at how we might compute such an equilibrium in practice.

Below we provide code to solve the household problem, taking r and w as fixed.

112.3.1 Primitives and operators

We will solve the household problem using value function iteration.

First we set up a `NamedTuple` to store the parameters that define a household asset accumulation problem, as well as the grids used to solve it

```
class Household(NamedTuple):
    β: float            # Discount factor
    a_grid: jnp.ndarray # Asset grid
    z_grid: jnp.ndarray # Exogenous states
    Π: jnp.ndarray      # Transition matrix

def create_household(β=0.96,                       # Discount factor
                     Π=[[0.9, 0.1], [0.1, 0.9]],   # Markov chain
                     z_grid=[0.1, 1.0],            # Exogenous states
                     a_min=1e-10, a_max=12.5,      # Asset grid
                     a_size=100):
    """
    Create a Household namedtuple with custom grids.
    """
    a_grid = jnp.linspace(a_min, a_max, a_size)
    z_grid, Π = map(jnp.array, (z_grid, Π))
    return Household(β=β, a_grid=a_grid, z_grid=z_grid, Π=Π)
```

For now we assume that $u(c) = \log(c)$

```
u = jnp.log
```

Here's a namedtuple that stores the wage rate and interest rate with default values

```
class Prices(NamedTuple):
    r: float = 0.01  # Interest rate
    w: float = 1.0   # Wages
```

Now we set up a vectorized version of the right-hand side of the Bellman equation (before maximization), which is a 3D array representing

$$B(a, z, a') = u(wz + (1 + r)a - a') + \beta \sum_{z'} v(a', z')\Pi(z, z')$$

for all (a, z, a').

```
def B(v, household, prices):
    # Unpack
    β, a_grid, z_grid, Π = household
    a_size, z_size = len(a_grid), len(z_grid)
    r, w = prices

    # Compute current consumption as array c[i, j, ip]
    a  = jnp.reshape(a_grid, (a_size, 1, 1))    # a[i]   ->  a[i, j, ip]
    z  = jnp.reshape(z_grid, (1, z_size, 1))    # z[j]   ->  z[i, j, ip]
    ap = jnp.reshape(a_grid, (1, 1, a_size))    # ap[ip] -> ap[i, j, ip]
    c = w * z + (1 + r) * a - ap

    # Calculate continuation rewards at all combinations of (a, z, ap)
    v = jnp.reshape(v, (1, 1, a_size, z_size)) # v[ip, jp] -> v[i, j, ip, jp]
    Π = jnp.reshape(Π, (1, z_size, 1, z_size)) # Π[j, jp]  -> Π[i, j, ip, jp]
```

(continues on next page)

(continued from previous page)

```
    EV = jnp.sum(v * Π, axis=-1)               # sum over last index jp

    # Compute the right-hand side of the Bellman equation
    return jnp.where(c > 0, u(c) + β * EV, -jnp.inf)
```

The next function computes greedy policies

```
def get_greedy(v, household, prices):
    """
    Computes a v-greedy policy σ, returned as a set of indices. If
    σ[i, j] equals ip, then a_grid[ip] is the maximizer at i, j.
    """
    # argmax over ap
    return jnp.argmax(B(v, household, prices), axis=-1)
```

We define the Bellman operator T, which takes a value function v and returns Tv as given in the Bellman equation

```
def T(v, household, prices):
    """
    The Bellman operator. Takes a value function v and returns Tv.
    """
    return jnp.max(B(v, household, prices), axis=-1)
```

Here's value function iteration, which repeatedly applies the Bellman operator until convergence

```
@jax.jit
def value_function_iteration(household, prices, tol=1e-4, max_iter=10_000):
    """
    Implements value function iteration using a compiled JAX loop.
    """
    β, a_grid, z_grid, Π = household
    a_size, z_size = len(a_grid), len(z_grid)

    def condition_function(loop_state):
        i, v, error = loop_state
        return jnp.logical_and(error > tol, i < max_iter)

    def update(loop_state):
        i, v, error = loop_state
        v_new = T(v, household, prices)
        error = jnp.max(jnp.abs(v_new - v))
        return i + 1, v_new, error

    # Initial loop state
    v_init = jnp.zeros((a_size, z_size))
    loop_state_init = (0, v_init, tol + 1)

    # Run the fixed point iteration
    i, v, error = jax.lax.while_loop(condition_function, update, loop_state_init)

    return get_greedy(v, household, prices)
```

As a first example of what we can do, let's compute and plot an optimal accumulation policy at fixed prices

```
# Create an instance of Household
household = create_household()
prices = Prices()
```

(continues on next page)

(continued from previous page)

```
r, w = prices
print(f"Interest rate: {r}, Wage: {w}")
```

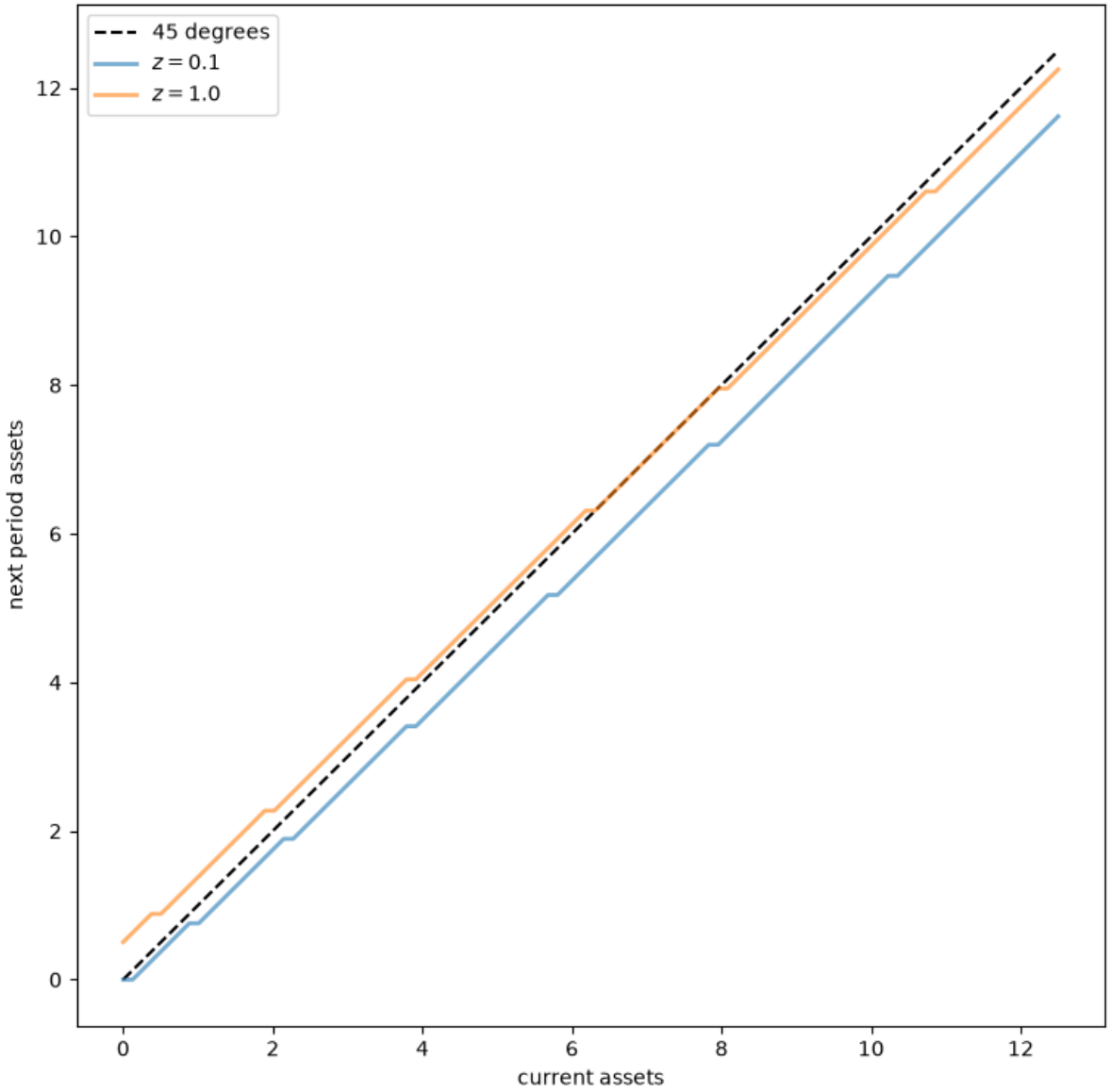
```
Interest rate: 0.01, Wage: 1.0
```

```
with qe.Timer():
    σ_star = value_function_iteration(household, prices).block_until_ready()
```

```
0.2580 seconds elapsed
```

The next plot shows asset accumulation policies at different values of the exogenous state

```
β, a_grid, z_grid, Π = household

fig, ax = plt.subplots(figsize=(9, 9))
ax.plot(a_grid, a_grid, 'k--', label="45 degrees")
for j, z in enumerate(z_grid):
    lb = f'$z = {z:.2}$'
    policy_vals = a_grid[σ_star[:, j]]
    ax.plot(a_grid, policy_vals, lw=2, alpha=0.6, label=lb)
    ax.set_xlabel('current assets')
    ax.set_ylabel('next period assets')
ax.legend(loc='upper left')
plt.show()
```



The plot shows asset accumulation policies at different values of the exogenous state.

112.3.2 Capital supply

To start thinking about equilibrium, we need to know how much capital households supply at a given interest rate r.

This quantity can be calculated by taking the stationary distribution of assets under the optimal policy and computing the mean.

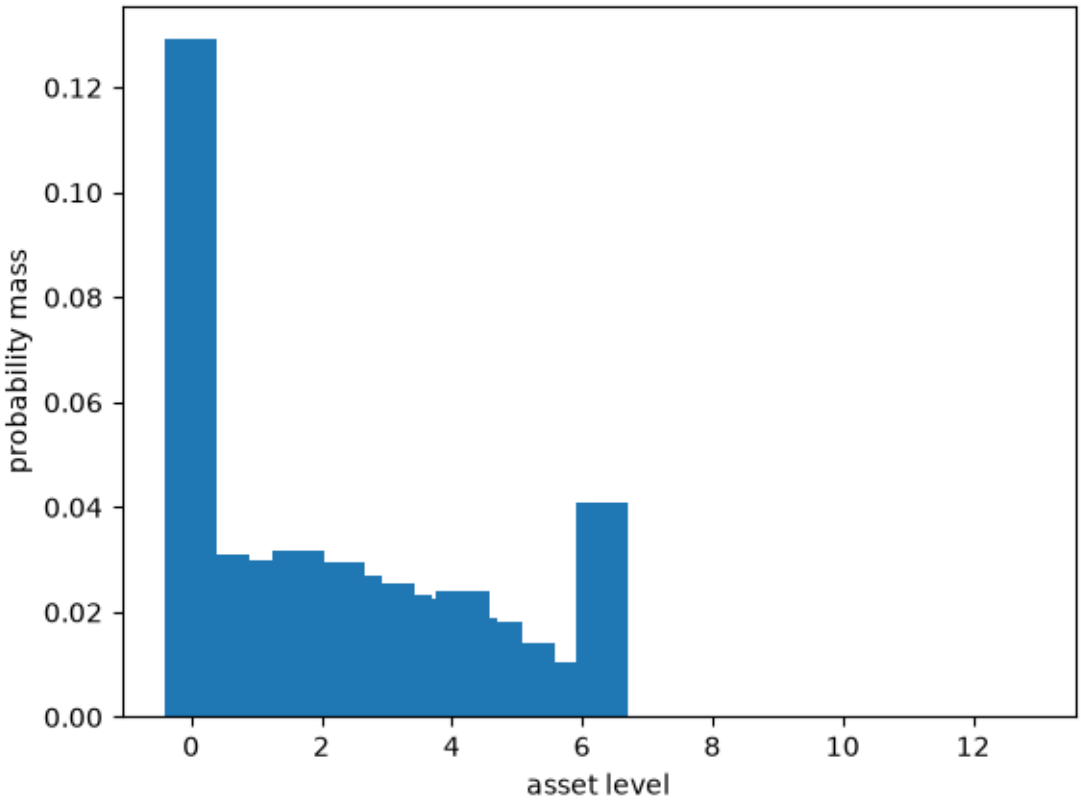
The next function computes the stationary distribution for a given policy σ via the following steps:

- Compute the stationary distribution $\psi = (\psi(a, z))$ of P_σ, which defines the Markov chain of the state (a_t, z_t) under policy σ.
- Sum out z_t to get the marginal distribution for a_t.

```
@jax.jit
def compute_asset_stationary(σ, household):
    # Unpack
    β, a_grid, z_grid, Π = household
    a_size, z_size = len(a_grid), len(z_grid)

    # Construct P_σ as an array of the form P_σ[i, j, ip, jp]
    ap_idx = jnp.arange(a_size)
    ap_idx = jnp.reshape(ap_idx, (1, 1, a_size, 1))
    σ = jnp.reshape(σ, (a_size, z_size, 1, 1))
    A = jnp.where(σ == ap_idx, 1, 0)
    Π = jnp.reshape(Π, (1, z_size, 1, z_size))
    P_σ = A * Π

    # Reshape P_σ into a matrix
    n = a_size * z_size
    P_σ = jnp.reshape(P_σ, (n, n))

    # Get stationary distribution and reshape back onto [i, j] grid
    ψ = compute_stationary(P_σ)
    ψ = jnp.reshape(ψ, (a_size, z_size))

    # Sum along the rows to get the marginal distribution of assets
    ψ_a = jnp.sum(ψ, axis=1)
    return ψ_a
```

Let's give this a test run.

```
ψ_a = compute_asset_stationary(σ_star, household)

fig, ax = plt.subplots()
ax.bar(household.a_grid, ψ_a)
ax.set_xlabel("asset level")
ax.set_ylabel("probability mass")
plt.show()
```



The distribution should sum to one:

```
ψ_a.sum()
```

```
Array(1., dtype=float64)
```

The next function computes aggregate capital supply by households under policy σ, given wages and interest rates

```
def capital_supply(σ, household):
    """
    Induced level of capital stock under the policy, taking r and w as given.
    """
    β, a_grid, z_grid, Π = household
    ψ_a = compute_asset_stationary(σ, household)
    return float(jnp.sum(ψ_a * a_grid))
```

112.3.3 Equilibrium

We compute a SREE as follows:

1. Set $n = 0$ and start with an initial guess K_0 for aggregate capital.
2. Determine prices r, w from the firm decision problem, given K_n.
3. Compute the optimal savings policy of households given these prices.
4. Compute aggregate capital K_{n+1} as the mean of steady-state capital given this savings policy.
5. If $K_{n+1} \approx K_n$, stop; otherwise, go to step 2.

We can write the sequence of operations in steps 2-4 as

$$K_{n+1} = G(K_n)$$

If K_{n+1} agrees with K_n then we have a SREE.

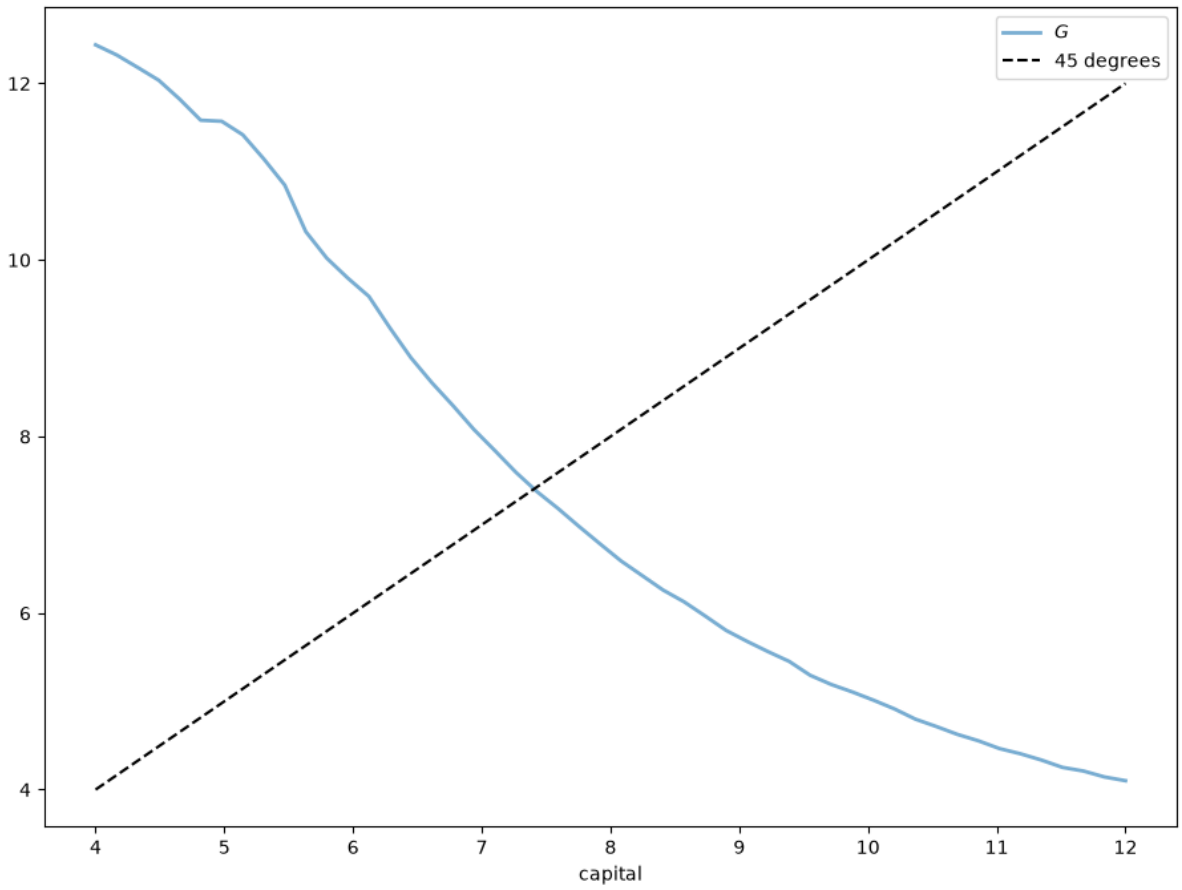
In other words, our problem is to find the fixed point of the one-dimensional map G.

Here's G expressed as a Python function

```
def G(K, firm, household):
    # Get prices r, w associated with K
    r = r_given_k(K, firm)
    w = r_to_w(r, firm)

    # Generate a household object with these prices, compute
    # aggregate capital.
    prices = Prices(r=r, w=w)
    σ_star = value_function_iteration(household, prices)
    return capital_supply(σ_star, household)
```

Let's inspect visually as a first pass

```
num_points = 50
firm = Firm()
household = create_household()
k_vals = jnp.linspace(4, 12, num_points)
out = [G(k, firm, household) for k in k_vals]

fig, ax = plt.subplots(figsize=(11, 8))
ax.plot(k_vals, out, lw=2, alpha=0.6, label='$G$')
ax.plot(k_vals, k_vals, 'k--', label="45 degrees")
ax.set_xlabel('capital')
ax.legend()
plt.show()
```



Now let's compute the equilibrium.

Looking at the figure above, we see that a simple iteration scheme $K_{n+1} = G(K_n)$ will cycle from high to low values, leading to slow convergence.

As a result, we use a damped iteration scheme of the form

$$K_{n+1} = \alpha K_n + (1 - \alpha) G(K_n)$$

```
def compute_equilibrium(firm, household,
                        K0=6, α=0.99, max_iter=1_000, tol=1e-4,
                        print_skip=10, verbose=False):
    n = 0
    K = K0
    error = tol + 1
    while error > tol and n < max_iter:
        new_K = α * K + (1 - α) * G(K, firm, household)
        error = abs(new_K - K)
        K = new_K
        n += 1
        if verbose and n % print_skip == 0:
            print(f"At iteration {n} with error {error}")
    return K, n
```

```
firm = Firm()
household = create_household()
```

(continues on next page)

(continued from previous page)

```
print("\nComputing equilibrium capital stock")
with qe.Timer():
    K_star, n = compute_equilibrium(firm, household, K0=6.0)
print(f"Computed equilibrium {K_star:.5} in {n} iterations")
```

```
Computing equilibrium capital stock
2.2801 seconds elapsed
Computed equilibrium 7.4095 in 259 iterations
```

This convergence is not very fast, given how quickly we can solve the household problem.

You can try varying α, but usually this parameter is hard to set a priori.

In the exercises below you will be asked to use bisection instead, which generally performs better.

112.3.4 Supply and demand curves

We can visualize the equilibrium using supply and demand curves.

The following code draws the aggregate supply and demand curves.

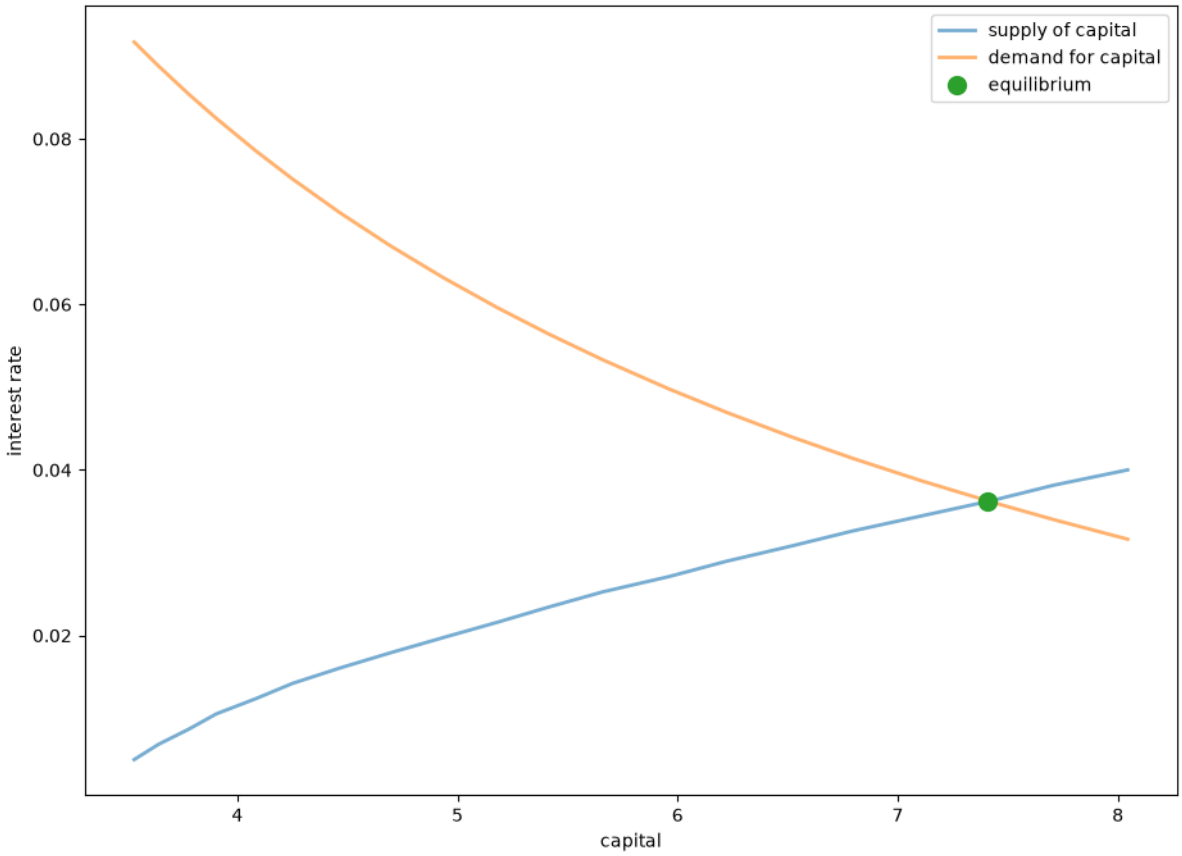
The intersection gives the equilibrium interest rate and capital

```
def prices_to_capital_stock(household, r, firm):
    """
    Map prices to the induced level of capital stock.
    """
    w = r_to_w(r, firm)
    prices = Prices(r=r, w=w)

    # Compute the optimal policy
    σ_star = value_function_iteration(household, prices)

    # Compute capital supply
    return capital_supply(σ_star, household)

# Create a grid of r values to compute demand and supply of capital
num_points = 20
r_vals = jnp.linspace(0.005, 0.04, num_points)

# Compute supply of capital
k_vals = []
for r in r_vals:
    k_vals.append(prices_to_capital_stock(household, r, firm))

# Plot against demand for capital by firms
fig, ax = plt.subplots(figsize=(11, 8))
ax.plot(k_vals, r_vals, lw=2, alpha=0.6,
        label='supply of capital')
ax.plot(k_vals, r_given_k(
        jnp.array(k_vals), firm), lw=2, alpha=0.6,
        label='demand for capital')

# Add marker at equilibrium
r_star = r_given_k(K_star, firm)
ax.plot(K_star, r_star, 'o', markersize=10, label='equilibrium')
```

(continues on next page)

(continued from previous page)

```
ax.set_xlabel('capital')
ax.set_ylabel('interest rate')
ax.legend(loc='upper right')

plt.show()
```



112.4 Exercises

Exercise 112.4.1

Write a new version of `compute_equilibrium` that uses `bisect` from `scipy.optimize` instead of damped iteration.

See if you can make it faster than the previous version.

In `bisect`,

- you should set `xtol=1e-4` to have the same error tolerance as the previous version.
- for the lower and upper bounds of the bisection routine try `a = 1.0` and `b = 20.0`.

Solution

We use bisection to find the zero of the function $h(k) = k - G(k)$

```
def compute_equilibrium_bisect(firm, household, a=1.0, b=20.0):
    K = bisect(lambda k: k - G(k, firm, household), a, b, xtol=1e-4)
    return K

firm = Firm()
household = create_household()
print("\nComputing equilibrium capital stock using bisection")
with qe.Timer():
    K_star = compute_equilibrium_bisect(firm, household)
print(f"Computed equilibrium capital stock {K_star:.5}")
```

```
Computing equilibrium capital stock using bisection
0.1780 seconds elapsed
Computed equilibrium capital stock 7.4178
```

The bisection method is faster than the damped iteration scheme.

Exercise 112.4.2

Show how equilibrium capital stock changes with β.

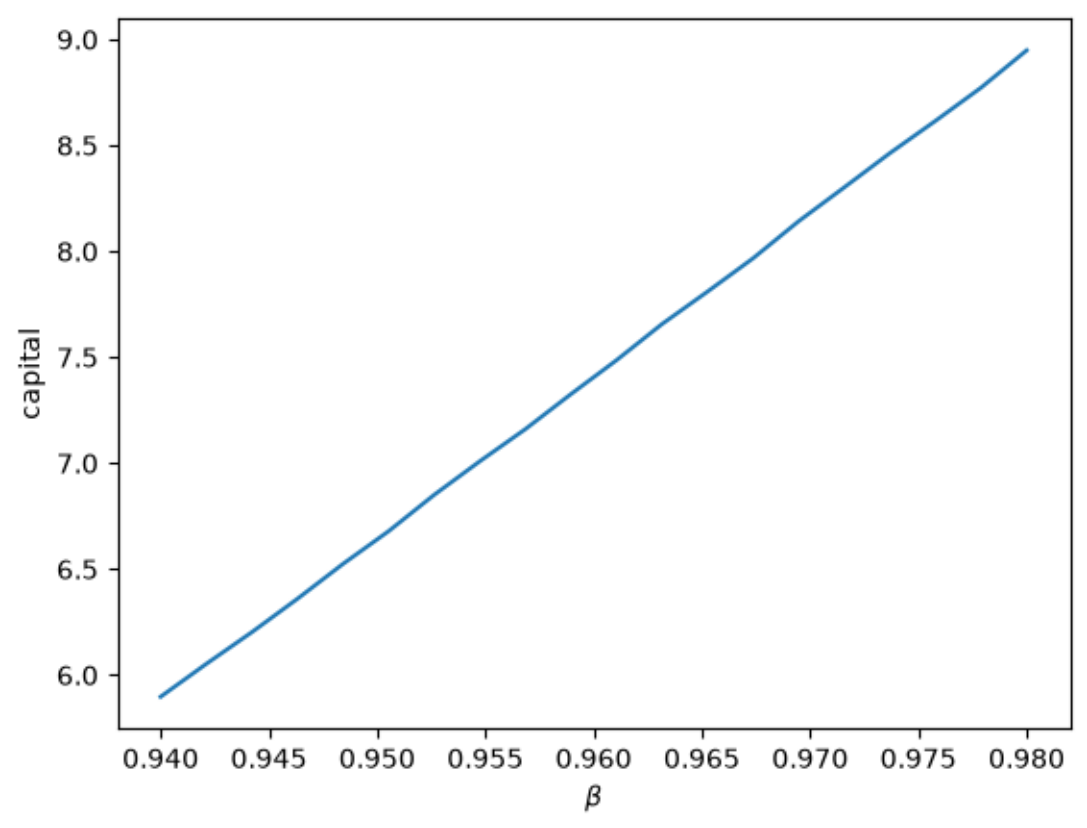
Use the following values of β and plot the relationship you find.

```
β_vals = jnp.linspace(0.94, 0.98, 20)
```

Solution

```
K_vals = []
K = 6.0  # initial guess

for β in β_vals:
    household = create_household(β=β)
    K = compute_equilibrium_bisect(firm, household, 0.5 * K, 1.5 * K)
    print(f"Computed equilibrium {K:.4} at β = {β}")
    K_vals.append(K)

fig, ax = plt.subplots()
ax.plot(β_vals, K_vals, ms=2)
ax.set_xlabel(r'$\beta$')
ax.set_ylabel('capital')
plt.show()
```

```
Computed equilibrium 5.897 at β = 0.94
Computed equilibrium 6.052 at β = 0.9421052631578948
Computed equilibrium 6.202 at β = 0.9442105263157894
Computed equilibrium 6.36 at β = 0.9463157894736841
Computed equilibrium 6.524 at β = 0.9484210526315789
Computed equilibrium 6.679 at β = 0.9505263157894737
Computed equilibrium 6.85 at β = 0.9526315789473683
Computed equilibrium 7.008 at β = 0.9547368421052631
Computed equilibrium 7.16 at β = 0.9568421052631579
Computed equilibrium 7.325 at β = 0.9589473684210527
Computed equilibrium 7.486 at β = 0.9610526315789474
Computed equilibrium 7.657 at β = 0.963157894736842
Computed equilibrium 7.81 at β = 0.9652631578947368
Computed equilibrium 7.967 at β = 0.9673684210526317
Computed equilibrium 8.142 at β = 0.9694736842105263
Computed equilibrium 8.302 at β = 0.971578947368421
Computed equilibrium 8.463 at β = 0.9736842105263158
Computed equilibrium 8.616 at β = 0.9757894736842105
Computed equilibrium 8.773 at β = 0.9778947368421053
Computed equilibrium 8.948 at β = 0.98
```



Exercise 112.4.3

In this lecture, we used value function iteration to solve the household problem.

An alternative is Howard policy iteration (HPI), which is discussed in detail in Dynamic Programming.

HPI can be faster than VFI for some problems because it uses fewer but more computationally intensive iterations.

Your task is to implement Howard policy iteration and compare the results with value function iteration.

Key concepts you'll need:

Howard policy iteration requires computing the value v_σ of a policy σ, defined as:

$$v_\sigma = (I - \beta P_\sigma)^{-1} r_\sigma$$

where r_σ is the reward vector under policy σ, and P_σ is the transition matrix induced by σ.

To solve this, you'll need to:

1. Compute current rewards $r_\sigma(a, z) = u((1 + r)a + wz - \sigma(a, z))$
2. Set up the linear operator R_σ where $(R_\sigma v)(a, z) = v(a, z) - \beta \sum_{z'} v(\sigma(a, z), z')\Pi(z, z')$
3. Solve $v_\sigma = R_\sigma^{-1} r_\sigma$ using `jax.scipy.sparse.linalg.bicgstab`

You can use the `get_greedy` function that's already defined in this lecture.

Implement the following Howard policy iteration routine:

```
def howard_policy_iteration(household, prices,
                            tol=1e-4, max_iter=10_000, verbose=False):
    """
    Howard policy iteration routine.
    """
    # Your code here
    pass
```

Once implemented, compute the equilibrium capital stock using HPI and verify that it produces approximately the same result as VFI at the default parameter values.

Solution

First, we need to implement the helper functions for Howard policy iteration.

The following function computes the array r_σ which gives current rewards given policy σ:

```
def compute_r_σ(σ, household, prices):
    """
    Compute current rewards at each i, j under policy σ. In particular,

        r_σ[i, j] = u((1 + r)a[i] + wz[j] - a'[ip])

    when ip = σ[i, j].
    """
    # Unpack
    β, a_grid, z_grid, Π = household
    a_size, z_size = len(a_grid), len(z_grid)
    r, w = prices

    # Compute r_σ[i, j]
    a = jnp.reshape(a_grid, (a_size, 1))
    z = jnp.reshape(z_grid, (1, z_size))
    ap = a_grid[σ]
    c = (1 + r) * a + w * z - ap
    r_σ = u(c)

    return r_σ
```

The linear operator R_σ is defined as:

```
def R_σ(v, σ, household):
    # Unpack
    β, a_grid, z_grid, Π = household
    a_size, z_size = len(a_grid), len(z_grid)

    # Set up the array v[σ[i, j], jp]
    zp_idx = jnp.arange(z_size)
    zp_idx = jnp.reshape(zp_idx, (1, 1, z_size))
    σ = jnp.reshape(σ, (a_size, z_size, 1))
    V = v[σ, zp_idx]

    # Expand Π[j, jp] to Π[i, j, jp]
    Π = jnp.reshape(Π, (1, z_size, z_size))

    # Compute and return v[i, j] - β Σ_jp v[σ[i, j], jp] * Π[j, jp]
    return v - β * jnp.sum(V * Π, axis=-1)
```

The next function computes the lifetime value of a given policy:

```
def get_value(σ, household, prices):
    """
    Get the lifetime value of policy σ by computing

        v_σ = R_σ^{-1} r_σ
    """
    r_σ = compute_r_σ(σ, household, prices)

    # Reduce R_σ to a function in v
    _R_σ = lambda v: R_σ(v, σ, household)

    # Compute v_σ = R_σ^{-1} r_σ using an iterative routine.
    return jax.scipy.sparse.linalg.bicgstab(_R_σ, r_σ)[0]
```

Now we can implement Howard policy iteration:

```
@jax.jit
def howard_policy_iteration(household, prices, tol=1e-4, max_iter=10_000):
    """
    Howard policy iteration routine using a compiled JAX loop.
    """
    β, a_grid, z_grid, Π = household
    a_size, z_size = len(a_grid), len(z_grid)

    def condition_function(loop_state):
        i, σ, v_σ, error = loop_state
        return jnp.logical_and(error > tol, i < max_iter)

    def update(loop_state):
        i, σ, v_σ, error = loop_state
        σ_new = get_greedy(v_σ, household, prices)
        v_σ_new = get_value(σ_new, household, prices)
        error = jnp.max(jnp.abs(v_σ_new - v_σ))
        return i + 1, σ_new, v_σ_new, error

    # Initial loop state
    σ_init = jnp.zeros((a_size, z_size), dtype=int)
    v_σ_init = get_value(σ_init, household, prices)
    loop_state_init = (0, σ_init, v_σ_init, tol + 1)
```

```
    # Run the fixed point iteration
    i, σ, v_σ, error = jax.lax.while_loop(condition_function, update, loop_state_
↪init)

    return σ
```

Now let's create a modified version of the G function that uses HPI:

```
def G_hpi(K, firm, household):
    # Get prices r, w associated with K
    r = r_given_k(K, firm)
    w = r_to_w(r, firm)

    # Generate prices and compute aggregate capital using HPI.
    prices = Prices(r=r, w=w)
    σ_star = howard_policy_iteration(household, prices)
    return capital_supply(σ_star, household)
```

And compute the equilibrium using HPI:

```
def compute_equilibrium_bisect_hpi(firm, household, a=1.0, b=20.0):
    K = bisect(lambda k: k - G_hpi(k, firm, household), a, b, xtol=1e-4)
    return K

firm = Firm()
household = create_household()
print("\nComputing equilibrium capital stock using HPI")
with qe.Timer():
    K_star_hpi = compute_equilibrium_bisect_hpi(firm, household)
print(f"Computed equilibrium capital stock with HPI: {K_star_hpi:.5}")
print(f"Previous equilibrium capital stock with VFI: {K_star:.5}")
print(f"Difference: {abs(K_star_hpi - K_star):.6}")
```

```
Computing equilibrium capital stock using HPI
0.8912 seconds elapsed
Computed equilibrium capital stock with HPI: 7.4172
Previous equilibrium capital stock with VFI: 7.4178
Difference: 0.000579834
```

The results show that both methods produce approximately the same equilibrium, confirming that HPI is a valid alternative to VFI.

CHAPTER

THREE

THE AIYAGARI MODEL WITH ENDOGENOUS GRID METHOD

Contents

- *The Aiyagari Model with Endogenous Grid Method*
  - *Overview*
  - *The Economy*
  - *Implementation with EGM*
  - *Computing Aggregate Capital by Simulation*
  - *Computing Equilibrium*
  - *Wealth Distribution*
  - *Summary and Comparison*
  - *Exercises*

In addition to what's included in base Anaconda, we need to install QuantEcon's Python library and JAX.

```
!pip install quantecon jax
```

113.1 Overview

This lecture combines two important computational methods in macroeconomics:

1. The **Aiyagari model** [Aiyagari, 1994] – a heterogeneous agent model with incomplete markets
2. The **endogenous grid method** (EGM) [Carroll, 2006] – an efficient algorithm for solving dynamic programming problems

In the *standard Aiyagari lecture*, we solved the household problem using discretization and value function iteration.

We then computed aggregate capital at a given set of prices using the stationary distribution of the finite Markov chain.

In this lecture, we take a different approach:

1. We use the **endogenous grid method** to solve the household problem via the Euler equation and linear interpolation.
2. We compute aggregate capital by **simulation** rather than an algebraic technique (which only works for the finite case).

These modifications make the solution method faster and more flexible, especially when dealing with more complex models.

We use JAX throughout, so that the EGM operator, the solver and the simulation are all JIT-compiled and vectorized.

113.1.1 References

The primary references for this lecture are:

- our *previous Aiyagari lecture* for the key ideas
- [Aiyagari, 1994] for the economic model
- [Carroll, 2006] for the endogenous grid method
- Chapter 18 of [Ljungqvist and Sargent, 2018] for textbook treatment

113.1.2 Preliminaries

We use the following imports:

```
import quantecon as qe
import matplotlib.pyplot as plt
import jax
import jax.numpy as jnp
import numpy as np
from typing import NamedTuple
from functools import partial
from scipy.optimize import bisect
```

We will use 64-bit floats with JAX in order to increase precision.

```
jax.config.update("jax_enable_x64", True)
```

113.2 The Economy

The economy consists of households and a representative firm.

113.2.1 Households

Infinitely lived households face idiosyncratic income shocks and a borrowing constraint.

The savings problem faced by a typical household is

$$\max \mathbb{E} \sum_{t=0}^{\infty} \beta^t u(c_t)$$

subject to

$$a_{t+1} + c_t \leq w z_t + (1+r) a_t \quad c_t \geq 0, \quad \text{and} \quad a_t \geq -B$$

where

- c_t is current consumption

- a_t is assets
- z_t is an exogenous component of labor income (stochastic employment status)
- w is the wage rate
- r is the interest rate
- B is the maximum amount that the agent is allowed to borrow

The exogenous process $\{z_t\}$ follows a finite state Markov chain with stochastic matrix Π.

Optimal interior consumption choices satisfy the Euler equation

$$u'(c) = \beta \mathbb{E}_z[(1+r)u'(c')]$$

(We use $'$ symbols for both derivatives and future values, which is not ideal but convenient and common.)

In terms of assets, this is

$$u'(wz + (1+r)a - a') = \beta(1+r)\sum_{z'} u'(wz' + (1+r)a' - s(a', z'))\Pi(z, z')$$

where s is the optimal savings policy function.

113.2.2 Firms

Firms produce output by hiring capital and labor under constant returns to scale.

The representative firm's output is

$$Y = AK^\alpha N^{1-\alpha}$$

where

- A and α are parameters with $A > 0$ and $\alpha \in (0, 1)$
- K is aggregate capital
- N is total labor supply (normalized to 1)

These parameters are stored in the following namedtuple:

```
class Firm(NamedTuple):
    A: float = 1.0    # Total factor productivity
    N: float = 1.0    # Total labor supply
    α: float = 0.33   # Capital share
    δ: float = 0.05   # Depreciation rate
```

From the firm's first-order condition, the inverse demand for capital is

$$r = A\alpha \left(\frac{N}{K}\right)^{1-\alpha} - \delta \tag{113.1}$$

```
def r_given_k(K, firm):
    """
    Inverse demand curve for capital.
    """
    A, N, α, δ = firm
    return A * α * (N / K)**(1 - α) - δ
```

The equilibrium wage rate as a function of r is

$$w(r) = A(1-\alpha)(A\alpha/(r+\delta))^{\alpha/(1-\alpha)} \tag{113.2}$$

```
def r_to_w(r, firm):
    """
    Equilibrium wages associated with a given interest rate r.
    """
    A, N, α, δ = firm
    return A * (1 - α) * (A * α / (r + δ))**(α / (1 - α))
```

113.2.3 Equilibrium

A **stationary rational expectations equilibrium (SREE)** consists of prices and policies such that:

- Households optimize given prices
- Firms maximize profits given prices
- Markets clear: aggregate capital supply equals aggregate capital demand
- Aggregate quantities are constant over time

113.3 Implementation with EGM

113.3.1 Household primitives

First we set up the household parameters and grids:

```
class Household(NamedTuple):
    β: float            # Discount factor
    a_grid: jnp.ndarray # Asset grid
    z_grid: jnp.ndarray # Exogenous states
    Π: jnp.ndarray      # Transition matrix

def create_household(β=0.96,                       # Discount factor
                     Π=[[0.9, 0.1], [0.1, 0.9]],   # Markov chain
                     z_grid=[0.1, 1.0],            # Exogenous states
                     a_min=1e-10, a_max=50.0,      # Asset grid
                     a_size=200):
    """
    Create a Household namedtuple with custom grids.
    """
    a_grid = jnp.linspace(a_min, a_max, a_size)
    z_grid, Π = map(jnp.array, (z_grid, Π))
    return Household(β=β, a_grid=a_grid, z_grid=z_grid, Π=Π)
```

For utility, we assume $u(c) = \log(c)$, which gives us $u'(c) = 1/c$ and $(u')^{-1}(x) = 1/x$.

```
@jax.jit
def u_prime(c):
    return 1 / c

@jax.jit
```

(continues on next page)

(continued from previous page)

```
def u_prime_inv(x):
    return 1 / x
```

Here's a namedtuple for prices:

```
class Prices(NamedTuple):
    r: float = 0.01  # Interest rate
    w: float = 1.0   # Wages
```

113.3.2 The EGM operator

The key insight of EGM is to avoid root-finding by choosing the asset grid exogenously and computing the consumption values directly from the Euler equation.

The Coleman-Reffett operator using EGM works as follows:

1. Start with a consumption policy σ represented on an exogenous grid of (next-period) assets $\{a_i\}$.
2. For each asset level a_i and current employment state z_j:
   - Compute the right-hand side of the Euler equation: \$RHS $= \beta(1+r)\sum_{z'} \Pi(z_j, z')\, u'(\sigma(a_i, z'))$\$
   - Use the inverse marginal utility to get current consumption: \$$c_{ij} = (u')^{-1}$(RHS)\$
   - Recover the implied current asset level from the budget constraint: \$$a_{ij} = \frac{c_{ij}+a_i-wz_j}{1+r}$\$
3. Reconstruct the new policy $K\sigma$ on the original asset grid by interpolating (a_{ij}, c_{ij}), handling the borrowing constraint where it binds.

The whole operation vectorizes cleanly, so we write it as a single JIT-compiled function and use `vmap` over the employment states:

```
@jax.jit
def K_egm(σ, household, prices):
    """
    The Coleman-Reffett operator using EGM for the Aiyagari model.

    Here σ[i, j] is consumption when next-period assets are a_grid[i]
    and the current employment state is z_grid[j].
    """
    β, a_grid, z_grid, Π = household
    r, w = prices
    z_size = len(z_grid)

    # Expectation E[u'(c(a', z')) | z] over the next-period shock
    Eu_prime = (Π @ u_prime(σ).T).T                          # (a_size, z_size)

    # Euler equation -> consumption on the endogenous grid
    c_endo = u_prime_inv(β * (1 + r) * Eu_prime)             # (a_size, z_size)

    # Implied current assets: a = (c + a' - w z) / (1 + r)
    a_endo = (c_endo + a_grid[:, None] - w * z_grid[None, :]) / (1 + r)

    # Interpolate back onto the exogenous grid for each employment state
    def interp_policy(j):
        # Where today's assets fall below the endogenous grid the borrowing
        # constraint binds, so the household saves a_grid[0] and consumes
```

(continues on next page)

(continued from previous page)

```
        # the rest of current income.
        return jnp.where(
            a_grid < a_endo[0, j],
            w * z_grid[j] + (1 + r) * a_grid - a_grid[0],
            jnp.interp(a_grid, a_endo[:, j], c_endo[:, j])
        )

    σ_new = jax.vmap(interp_policy)(jnp.arange(z_size))     # (z_size, a_size)
    return σ_new.T                                          # (a_size, z_size)
```

113.3.3 Solving the household problem

We solve for the optimal policy by iterating the EGM operator to convergence.

The solver is fully JIT-compiled and uses `jax.lax.while_loop` for the iteration:

```
@jax.jit
def solve_household(household, prices, tol=1e-6, max_iter=10_000):
    """
    Solve the household problem by iterating the EGM operator.

    Returns the optimal consumption policy σ[i, j], where i indexes
    next-period assets and j indexes employment states.
    """
    β, a_grid, z_grid, Π = household
    r, w = prices

    # Initial guess: consume half of current income
    income = w * z_grid[None, :] + (1 + r) * a_grid[:, None]
    σ_init = 0.5 * income

    def condition(state):
        i, σ, error = state
        return (error > tol) & (i < max_iter)

    def body(state):
        i, σ, error = state
        σ_new = K_egm(σ, household, prices)
        error = jnp.max(jnp.abs(σ_new - σ))
        return i + 1, σ_new, error

    i, σ, error = jax.lax.while_loop(condition, body, (0, σ_init, tol + 1))
    return σ
```

Let's test this on an example:

```
household = create_household()
prices = Prices(r=0.01, w=1.0)

with qe.Timer():
    σ_star = solve_household(household, prices)
    jax.block_until_ready(σ_star)
```

```
0.4592 seconds elapsed
```

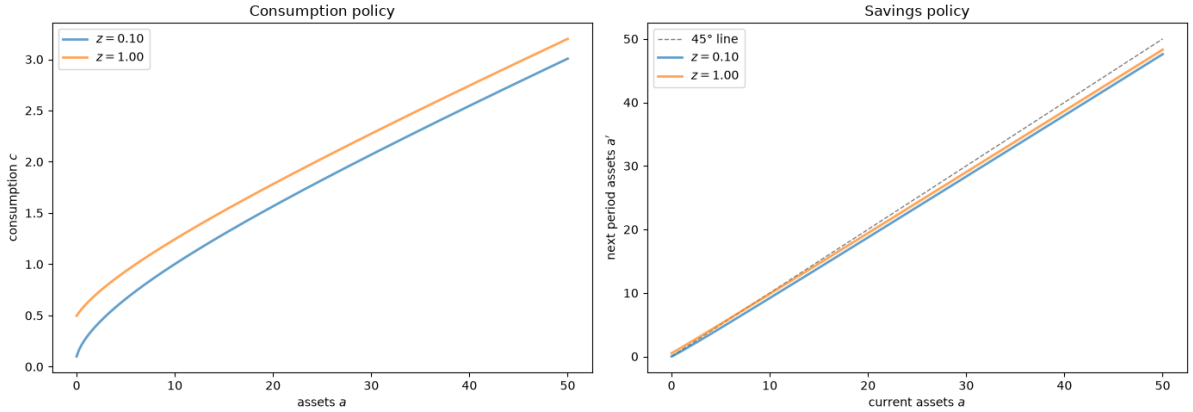
We can check that the policy is a fixed point of the EGM operator by measuring the residual:

```
residual = jnp.max(jnp.abs(K_egm(σ_star, household, prices) - σ_star))
print(f"Final Euler residual: {residual:.2e}")
```

```
Final Euler residual: 8.34e-07
```

Let's plot the resulting policy functions:

```
β, a_grid, z_grid, Π = household
r, w = prices

# Convert consumption policy to savings policy
income = w * z_grid[None, :] + (1 + r) * a_grid[:, None]
savings = income - σ_star

fig, axes = plt.subplots(1, 2, figsize=(14, 5))

# Plot consumption policy
ax = axes[0]
for j, z in enumerate(z_grid):
    ax.plot(a_grid, σ_star[:, j], label=f'$z={z:.2f}$', lw=2, alpha=0.7)
ax.set_xlabel('assets $a$')
ax.set_ylabel('consumption $c$')
ax.set_title('Consumption policy')
ax.legend()

# Plot savings policy
ax = axes[1]
ax.plot(a_grid, a_grid, 'k--', lw=1, alpha=0.5, label='45° line')
for j, z in enumerate(z_grid):
    ax.plot(a_grid, savings[:, j], label=f'$z={z:.2f}$', lw=2, alpha=0.7)
ax.set_xlabel('current assets $a$')
ax.set_ylabel("next period assets $a'$")
ax.set_title('Savings policy')
ax.legend()

plt.tight_layout()
plt.show()
```



113.4 Computing Aggregate Capital by Simulation

Instead of computing the stationary distribution of the Markov chain analytically, we compute aggregate capital by simulating a large cross-section of households.

This approach:

- is more flexible (works with continuous shocks, non-linear policies, etc.)
- avoids storing and manipulating large transition matrices
- is conceptually simpler

The simulation is fully vectorized: we advance all households simultaneously with `jax.lax.fori_loop`, drawing the employment transitions by inverse-CDF sampling and looking up consumption with a `vmap`-ed interpolation.

```
@partial(jax.jit, static_argnames=('num_households', 'num_periods'))
def simulate_cross_section(σ, household, prices, key,
                           num_households=50_000, num_periods=1_000):
    """
    Simulate a panel of households forward and return the terminal
    cross-section of assets and employment states.
    """
    β, a_grid, z_grid, Π = household
    r, w = prices

    # CDF of each row of Π, used for inverse-CDF sampling of transitions
    Π_cdf = jnp.cumsum(Π, axis=1)

    # Vectorized consumption lookup: interpolate σ along assets for each z
    @jax.vmap
    def consume(a, j):
        return jnp.interp(a, a_grid, σ[:, j])

    # Initial conditions: everyone at the middle of the grid, in state 0
    assets = jnp.full(num_households, a_grid[len(a_grid) // 2])
    z_idx = jnp.zeros(num_households, dtype=jnp.int32)

    def step(t, state):
        assets, z_idx, key = state
        key, subkey = jax.random.split(key)
        unif = jax.random.uniform(subkey, (num_households,))
        # Markov transition via inverse CDF
        z_idx = (unif[:, None] > Π_cdf[z_idx]).sum(axis=1).astype(jnp.int32)
        # Budget constraint: consume, then carry assets to next period
        income = w * z_grid[z_idx] + (1 + r) * assets
        assets = income - consume(assets, z_idx)
        # Enforce the asset grid bounds
        assets = jnp.clip(assets, a_grid[0], a_grid[-1])
        return assets, z_idx, key

    assets, z_idx, key = jax.lax.fori_loop(
        0, num_periods, step, (assets, z_idx, key)
    )
    return assets, z_idx
```

Now we can compute capital supply for given prices by solving the household problem and averaging assets across the simulated cross-section:

```
def capital_supply(household, prices, key,
                   num_households=50_000, num_periods=1_000):
    """
    Compute aggregate capital supply by simulation.
    """
    σ = solve_household(household, prices)
    assets, _ = simulate_cross_section(
        σ, household, prices, key,
        num_households=num_households, num_periods=num_periods
    )
    return float(jnp.mean(assets))
```

Let's test it:

```
household = create_household()
prices = Prices(r=0.01, w=1.0)
key = jax.random.PRNGKey(42)

with qe.Timer():
    K_supply = capital_supply(household, prices, key)

print(f"Capital supply: {K_supply:.4f}")
```

```
0.9646 seconds elapsed
Capital supply: 2.5863
```

113.5 Computing Equilibrium

Now we can compute the equilibrium by finding the capital stock at which capital supply equals capital demand.

Given K, the equilibrium mapping G computes:

1. prices (r, w) from the firm's first-order conditions,
2. the household's optimal policy given those prices,
3. aggregate capital supply via simulation.

```
def G(K, firm, household, key,
      num_households=50_000, num_periods=1_000):
    """
    The equilibrium mapping K -> capital supply.
    """
    r = r_given_k(K, firm)
    w = r_to_w(r, firm)
    prices = Prices(r=r, w=w)
    return capital_supply(household, prices, key,
                          num_households=num_households,
                          num_periods=num_periods)
```

We compute the equilibrium by applying bisection to the excess demand $K - G(K)$.

We pass a fixed random key to every evaluation so that the excess demand function is deterministic, as required by the root finder.

```
def compute_equilibrium(firm, household, key,
                        K_min=4.0, K_max=12.0,
                        num_households=50_000, num_periods=1_000,
                        xtol=1e-2):
    """
    Compute the equilibrium capital stock using bisection.
    """
    def excess_demand(K):
        return K - G(K, firm, household, key,
                     num_households=num_households,
                     num_periods=num_periods)

    return bisect(excess_demand, K_min, K_max, xtol=xtol)
```

Let's compute the equilibrium:

```
firm = Firm()
household = create_household()
key = jax.random.PRNGKey(42)

with qe.Timer():
    K_star = compute_equilibrium(firm, household, key)

r_star = r_given_k(K_star, firm)
w_star = r_to_w(r_star, firm)

print(f"\nEquilibrium capital:       {K_star:.4f}")
print(f"Equilibrium interest rate: {r_star:.4f}")
print(f"Equilibrium wage:          {w_star:.4f}")
```

```
1.9025 seconds elapsed

Equilibrium capital:       8.1484
Equilibrium interest rate: 0.0309
Equilibrium wage:          1.3388
```

113.5.1 Visualizing equilibrium

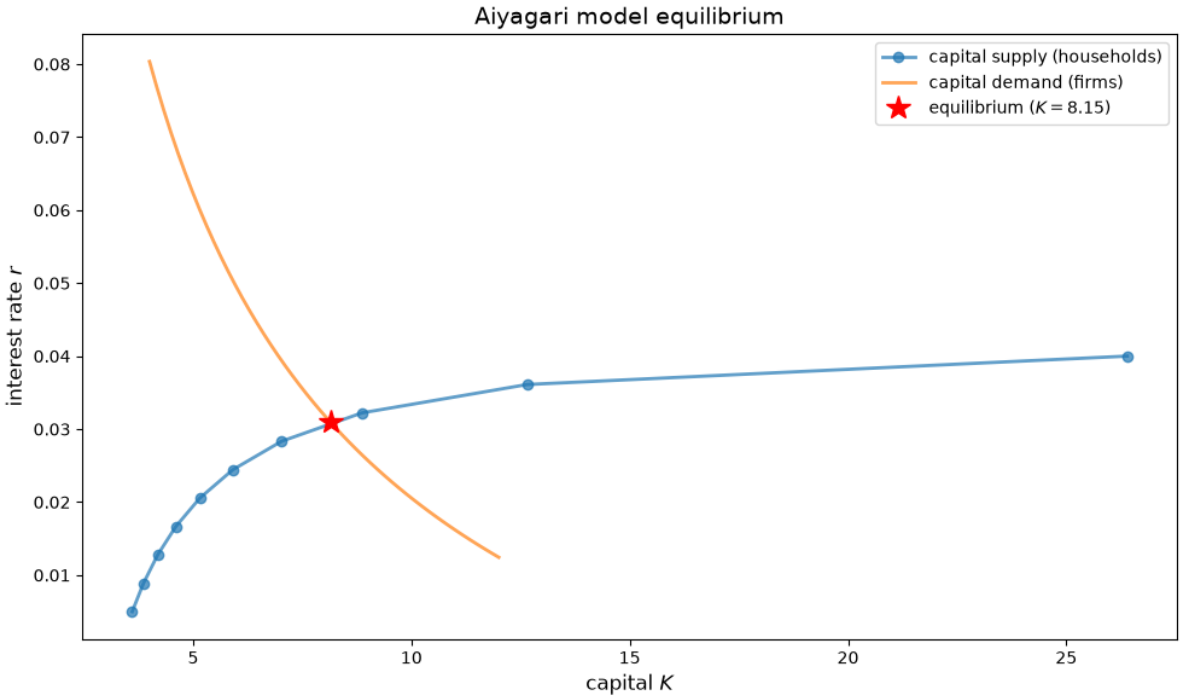
Let's plot the supply and demand curves:

```
# Supply curve: capital supplied by households as a function of r
r_vals = np.linspace(0.005, 0.04, 10)
K_supply_vals = []

for r in r_vals:
    w = r_to_w(r, firm)
    prices = Prices(r=r, w=w)
    K_supply_vals.append(capital_supply(household, prices, key))

# Demand curve: capital demanded by firms as a function of r
K_vals = np.linspace(4, 12, 50)
r_demand_vals = r_given_k(K_vals, firm)

fig, ax = plt.subplots(figsize=(10, 6))
```

(continues on next page)

(continued from previous page)

```
ax.plot(K_supply_vals, r_vals, 'o-', lw=2, alpha=0.7,
        label='capital supply (households)', markersize=6)
ax.plot(K_vals, r_demand_vals, lw=2, alpha=0.7,
        label='capital demand (firms)')
ax.plot(K_star, r_star, 'r*', markersize=15, zorder=5,
        label=f'equilibrium ($K={K_star:.2f}$)')

ax.set_xlabel('capital $K$', fontsize=12)
ax.set_ylabel('interest rate $r$', fontsize=12)
ax.set_title('Aiyagari model equilibrium', fontsize=14)
ax.legend(fontsize=10)

plt.tight_layout()
plt.show()
```



113.6 Wealth Distribution

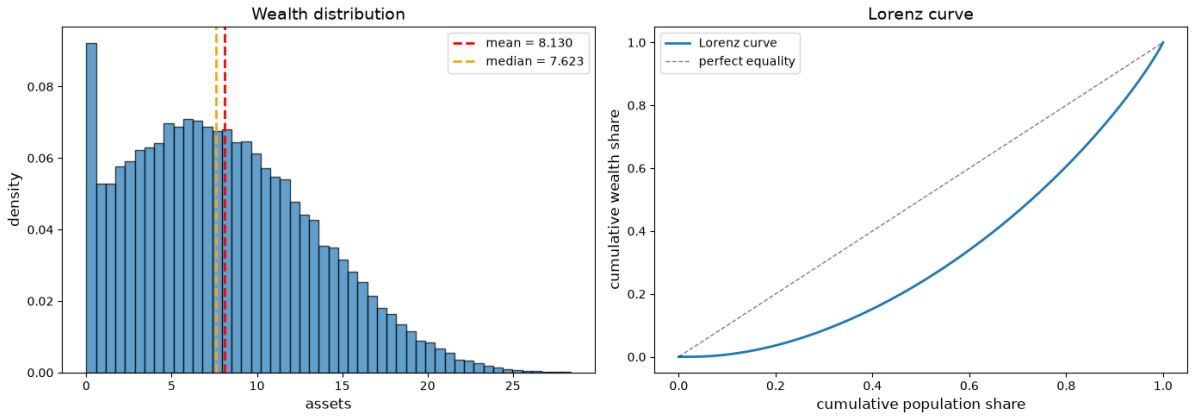
One advantage of the simulation approach is that we can easily examine the wealth distribution.

We reuse the cross-section simulated at equilibrium prices:

```
prices_star = Prices(r=r_star, w=w_star)
σ_star = solve_household(household, prices_star)
assets_dist, z_dist = simulate_cross_section(σ_star, household, prices_star, key)

assets_dist = np.asarray(assets_dist)
```

We use QuantEcon's `lorenz_curve` and `gini_coefficient` to summarize inequality:

```
fig, axes = plt.subplots(1, 2, figsize=(14, 5))

# Histogram
ax = axes[0]
ax.hist(assets_dist, bins=50, density=True, alpha=0.7, edgecolor='black')
ax.axvline(np.mean(assets_dist), color='red', linestyle='--', linewidth=2,
           label=f'mean = {np.mean(assets_dist):.3f}')
ax.axvline(np.median(assets_dist), color='orange', linestyle='--', linewidth=2,
           label=f'median = {np.median(assets_dist):.3f}')
ax.set_xlabel('assets', fontsize=12)
ax.set_ylabel('density', fontsize=12)
ax.set_title('Wealth distribution', fontsize=14)
ax.legend()

# Lorenz curve
ax = axes[1]
cum_pop, cum_wealth = qe.lorenz_curve(assets_dist)
ax.plot(cum_pop, cum_wealth, lw=2, label='Lorenz curve')
ax.plot([0, 1], [0, 1], 'k--', lw=1, alpha=0.5, label='perfect equality')
ax.set_xlabel('cumulative population share', fontsize=12)
ax.set_ylabel('cumulative wealth share', fontsize=12)
ax.set_title('Lorenz curve', fontsize=14)
ax.legend()

plt.tight_layout()
plt.show()

gini = qe.gini_coefficient(assets_dist)
print(f"\nGini coefficient: {gini:.4f}")
```




```
Gini coefficient: 0.3645
```

113.7 Summary and Comparison

This lecture demonstrated how to solve the Aiyagari model using:

1. **Endogenous Grid Method (EGM)** for the household problem
    - avoids costly root-finding by working backwards from the Euler equation
    - directly computes consumption from marginal utility
    - more efficient than value function iteration
2. **Simulation** for computing aggregate capital
    - simulates a large cross-section of households
    - more flexible than analytical stationary distributions
    - allows easy computation of wealth inequality measures

113.7.1 Comparison with standard approach

Compared to the *standard Aiyagari lecture*:

Advantages:

- EGM avoids the root-finding required by value function iteration
- simulation is more flexible (works with continuous shocks, non-linear policies)
- it is easy to compute distributional statistics (Gini, percentiles, etc.)
- the approach is simpler to extend to more complex models

Disadvantages:

- simulation requires a large number of households for accuracy
- equilibrium computation is subject to Monte Carlo noise
- it is less precise than the analytical stationary distribution

113.7.2 Extensions

This framework can be easily extended to:

- continuous income shocks (e.g., lognormal)
- more complex preference specifications
- aggregate shocks and heterogeneous agent New Keynesian (HANK) models
- life-cycle models with age-dependent policies

113.8 Exercises

Exercise 113.8.1

Compare the speed and accuracy of EGM against the value function iteration approach used in the *standard Aiyagari lecture*.

1. Solve the household problem with both methods at the same prices.
2. Time both methods and compare the resulting policies.
3. Which method is faster? Are the policies close?

Exercise 113.8.2

Study how the wealth distribution changes with the discount factor β.

1. Compute equilibria for $\beta \in \{0.94, 0.95, 0.96, 0.97\}$.
2. For each β, compute and plot the wealth distribution.
3. How does the Gini coefficient change with β?
4. Explain the economic intuition.

Exercise 113.8.3

The simulation method introduced in this lecture uses a fixed number of periods. Investigate the impact of this choice:

1. Vary `num_periods` from 200 to 2000.
2. For each value, compute the mean assets multiple times with different random keys.
3. Plot the standard deviation of the capital estimate as a function of `num_periods`.
4. What is the trade-off between accuracy and computational cost?

Exercise 113.8.4

Extend the model to include a third employment state (e.g., unemployed, part-time, full-time):

1. Set up a 3-state Markov chain with appropriate transition probabilities.
2. Define income levels for each state.
3. Re-compute the equilibrium.
4. How does the additional heterogeneity affect the wealth distribution?

CHAPTER

FOUR

TRANSITIONS IN AN OVERLAPPING GENERATIONS MODEL

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install --upgrade quantecon
```

114.1 Introduction

This lecture presents a life-cycle model consisting of overlapping generations of two-period lived people proposed by Peter Diamond [Diamond, 1965].

We'll present the version that was analyzed in chapter 2 of Auerbach and Kotlikoff (1987) [Auerbach and Kotlikoff, 1987].

Auerbach and Kotlikoff (1987) used their two period model as a warm-up for their analysis of overlapping generation models of long-lived people that is the main topic of their book.

Their model of two-period lived overlapping generations is a useful starting point because

- it sets forth the structure of interactions between generations of different agents who are alive at a given date
- it activates forces and tradeoffs confronting the government and successive generations of people
- it is good laboratory for studying connections between government tax and subsidy programs and for policies for issuing and servicing government debt
- some interesting experiments involving transitions from one steady state to another can be computed by hand
- it is a good setting for illustrating a **shooting method** for solving a system of non-linear difference equations with initial and terminal condition

Note

Auerbach and Kotlikoff use computer code to calculate transition paths for their models with long-lived people.

We take the liberty of extending Auerbach and Kotlikoff's chapter 2 model to study some arrangements for redistributing resources across generations

- these take the form of a sequence of age-specific lump sum taxes and transfers

We study how these arrangements affect capital accumulation and government debt

114.2 Setting

Time is discrete and is indexed by $t = 0, 1, 2, ...$

The economy lives forever, but the people inside it do not.

At each time $t \geq 0$ a representative old person and a representative young person are alive.

At time t a representative old person coexists with a representative young person who will become an old person at time $t + 1$.

We assume that the population size is constant over time.

A young person works, saves, and consumes.

An old person dissaves and consumes, but does not work,

A government lives forever, i.e., at $t = 0, 1, 2, ...$

Each period $t \geq 0$, the government taxes, spends, transfers, and borrows.

Initial conditions set outside the model at time $t = 0$ are

- K_0 – initial capital stock brought into time $t = 0$ by a representative initial old person
- D_0 – government debt falling due at $t = 0$ and owned by a representative old person at time $t = 0$

K_0 and D_0 are both measured in units of time 0 goods.

A government **policy** consists of five sequences $\{G_t, D_t, \tau_t, \delta_{ot}, \delta_{yt}\}_{t=0}^{\infty}$ whose components are

- τ_t – flat rate tax at time t on wages and earnings from capital and government bonds
- D_t – one-period government bond principal due at time t, per capita
- G_t – government purchases of goods at time t, per capita
- δ_{yt} – a lump sum tax on each young person at time t
- δ_{ot} – a lump sum tax on each old person at time t

An **allocation** is a collection of sequences $\{C_{yt}, C_{ot}, K_{t+1}, L_t, Y_t, G_t\}_{t=0}^{\infty}$; constituents of the sequences include

- K_t – physical capital per capita
- L_t – labor per capita
- Y_t – output per capita

and also

- C_{yt} – consumption of young person at time $t \geq 0$
- C_{ot} – consumption of old person at time $t \geq 0$
- $K_{t+1} - K_t \equiv I_t$ – investment in physical capital at time $t \geq 0$
- G_t – government purchases

National income and product accounts consist of a sequence of equalities

- $Y_t = C_{yt} + C_{ot} + (K_{t+1} - K_t) + G_t, \quad t \geq 0$

A **price system** is a pair of sequences $\{W_t, r_t\}_{t=0}^{\infty}$; constituents of a price sequence include rental rates for the factors of production

- W_t – rental rate for labor at time $t \geq 0$
- r_t – rental rate for capital at time $t \geq 0$

114.3 Production

There are two factors of production, physical capital K_t and labor L_t.

Capital does not depreciate.

The initial capital stock K_0 is owned by the representative initial old person, who rents it to the firm at time 0.

Net investment rate I_t at time t is

$$I_t = K_{t+1} - K_t$$

The capital stock at time t emerges from cumulating past rates of investment:

$$K_t = K_0 + \sum_{s=0}^{t-1} I_s$$

A Cobb-Douglas technology converts physical capital K_t and labor services L_t into output Y_t

$$Y_t = K_t^\alpha L_t^{1-\alpha}, \quad \alpha \in (0,1) \tag{114.1}$$

114.4 Government

At time $t-1$, the government issues one-period risk-free debt that promises to pay D_t time t goods per capita at time t.

Young people at time t purchase government debt D_{t+1} that matures at time $t+1$.

Government debt issued at t bears a before-tax net rate of interest rate of r_t at time $t+1$.

The government budget constraint at time $t \geq 0$ is

$$D_{t+1} - D_t = r_t D_t + G_t - T_t$$

or

$$D_{t+1} = (1 + r_t) D_t + G_t - T_t. \tag{114.2}$$

Total tax collections net of transfers equal T_t and satisfy

$$T_t = \tau_t W_t L_t + \tau_t r_t (D_t + K_t) + \delta_{yt} + \delta_{ot}$$

114.5 Activities in Factor Markets

Old people: At each $t \geq 0$, a representative old person

- brings K_t and D_t into the period,
- rents capital to a representative firm for $r_t K_t$,
- pays taxes $\tau_t r_t (K_t + D_t)$ on its rental and interest earnings,
- pays a lump sum tax δ_{ot} to the government,
- sells K_t to a young person.

Young people: At each $t \geq 0$, a representative young person

- sells one unit of labor services to a representative firm for W_t in wages,
- pays taxes $\tau_t W_t$ on its labor earnings
- pays a lump sum tax δ_{yt} to the goverment,
- spends C_{yt} on consumption,
- acquires non-negative assets A_{t+1} consisting of a sum of physical capital K_{t+1} and one-period government bonds D_{t+1} that mature at $t+1$.

Note

If a lump-sum tax is negative, it means that the government pays the person a subsidy.

114.6 Representative firm's problem

The representative firm hires labor services from young people at competitive wage rate W_t and hires capital from old people at competitive rental rate r_t.

The rental rate on capital r_t equals the interest rate on government one-period bonds.

Units of the rental rates are:

- for W_t, output at time t per unit of labor at time t
- for r_t, output at time t per unit of capital at time t

We take output at time t as **numeraire**, so the price of output at time t is one.

The firm's profits at time t are

$$K_t^\alpha L_t^{1-\alpha} - r_t K_t - W_t L_t.$$

To maximize profits a firm equates marginal products to rental rates:

$$\begin{aligned} W_t &= (1-\alpha) K_t^\alpha L_t^{-\alpha} \\ r_t &= \alpha K_t^\alpha L_t^{1-\alpha} \end{aligned} \tag{114.3}$$

Output can be consumed either by old people or young people; or sold to young people who use it to augment the capital stock; or sold to the government for uses that do not generate utility for the people in the model (i.e., "it is thrown into the ocean").

The firm thus sells output to old people, young people, and the government.

114.7 Individuals' problems

114.7.1 Initial old person

At time $t=0$, a representative initial old person is endowed with $(1+r_0(1-\tau_0))A_0$ in initial assets.

It must pay a lump sum tax to (if positive) or receive a subsidy from (if negative) δ_{ot} the government.

An old person's budget constraint is

$$C_{o0} = (1+r_0(1-\tau_0))A_0 - \delta_{ot}. \tag{114.4}$$

An initial old person's utility function is C_{o0}, so the person's optimal consumption plan is provided by equation (114.4).

114.7.2 Young person

At each $t \geq 0$, a young person inelastically supplies one unit of labor and in return receives pre-tax labor earnings of W_t units of output.

A young person's post-tax-and-transfer earnings are $W_t(1-\tau_t) - \delta_{yt}$.

At each $t \geq 0$, a young person chooses a consumption plan C_{yt}, C_{ot+1} to maximize the Cobb-Douglas utility function

$$U_t = C_{yt}^{\beta} C_{o,t+1}^{1-\beta}, \quad \beta \in (0,1) \tag{114.5}$$

subject to the following budget constraints at times t and $t+1$:

$$\begin{aligned} C_{yt} + A_{t+1} &= W_t(1-\tau_t) - \delta_{yt} \\ C_{ot+1} &= (1 + r_{t+1}(1-\tau_{t+1}))A_{t+1} - \delta_{ot} \end{aligned} \tag{114.6}$$

Solving the second equation of (114.6) for savings A_{t+1} and substituting it into the first equation implies the present value budget constraint

$$C_{yt} + \frac{C_{ot+1}}{1 + r_{t+1}(1-\tau_{t+1})} = W_t(1-\tau_t) - \delta_{yt} - \frac{\delta_{ot}}{1 + r_{t+1}(1-\tau_{t+1})} \tag{114.7}$$

To solve the young person's choice problem, form a Lagrangian

$$\begin{aligned} \mathcal{L} &= C_{yt}^{\beta} C_{o,t+1}^{1-\beta} \\ &\quad + \lambda \Big[C_{yt} + \frac{C_{ot+1}}{1 + r_{t+1}(1-\tau_{t+1})} - W_t(1-\tau_t) + \delta_{yt} + \frac{\delta_{ot}}{1 + r_{t+1}(1-\tau_{t+1})} \Big], \end{aligned} \tag{114.8}$$

where λ is a Lagrange multiplier on the intertemporal budget constraint (114.7).

After several lines of algebra, the intertemporal budget constraint (114.7) and the first-order conditions for maximizing $\mathcal{L}$ with respect to C_{yt}, C_{ot+1} imply that an optimal consumption plan satisfies

$$\begin{aligned} C_{yt} &= \beta \Big[W_t(1-\tau_t) - \delta_{yt} - \frac{\delta_{ot}}{1 + r_{t+1}(1-\tau_{t+1})} \Big] \\ \frac{C_{0t+1}}{1 + r_{t+1}(1-\tau_{t+1})} &= (1-\beta) \Big[W_t(1-\tau_t) - \delta_{yt} - \frac{\delta_{ot}}{1 + r_{t+1}(1-\tau_{t+1})} \Big] \end{aligned} \tag{114.9}$$

The first-order condition for minimizing Lagrangian (114.8) with respect to the Lagrange multipler λ recovers the budget constraint (114.7), which, using (114.9) gives the optimal savings plan

$$A_{t+1} = (1-\beta)[(1-\tau_t)W_t - \delta_{yt}] + \beta \frac{\delta_{ot}}{1 + r_{t+1}(1-\tau_{t+1})} \tag{114.10}$$

114.8 Equilbrium

Definition: An equilibrium is an allocation, a government policy, and a price system with the properties that

- given the price system and the government policy, the allocation solves
  - representative firms' problems for $t \geq 0$
  - individual persons' problems for $t \geq 0$
- given the price system and the allocation, the government budget constraint is satisfied for all $t \geq 0$.

114.9 Next steps

To begin our analysis of equilibrium outcomes, we'll study the special case of the model with which Auerbach and Kotlikoff (1987) [Auerbach and Kotlikoff, 1987] began their analysis in chapter 2.

It can be solved by hand.

We shall do that next.

After we derive a closed form solution, we'll pretend that we don't know and will compute equilibrium outcome paths.

We'll do that by first formulating an equilibrium as a fixed point of a mapping from sequences of factor prices and tax rates to sequences of factor prices and tax rates.

We'll compute an equilibrium by iterating to convergence on that mapping.

114.10 Closed form solution

To get the special chapter 2 case of Auerbach and Kotlikoff (1987) [Auerbach and Kotlikoff, 1987], we set both δ_{ot} and δ_{yt} to zero.

As our special case of (114.9), we compute the following consumption-savings plan for a representative young person:

$$
\begin{aligned}
C_{yt} &= \beta(1-\tau_t)W_t \\
A_{t+1} &= (1-\beta)(1-\tau_t)W_t
\end{aligned}
$$

Using (114.3) and $A_t = K_t + D_t$, we obtain the following closed form transition law for capital:

$$
K_{t+1} = K_t^{\alpha}\left(1-\tau_t\right)\left(1-\alpha\right)\left(1-\beta\right) - D_t \tag{114.11}
$$

114.10.1 Steady states

From (114.11) and the government budget constraint (114.2), we compute **time-invariant** or **steady state values** $\hat{K}, \hat{D}, \hat{T}$:

$$
\begin{aligned}
\hat{K} &= \hat{K}\left(1-\hat{\tau}\right)\left(1-\alpha\right)\left(1-\beta\right) - \hat{D} \\
\hat{D} &= (1+\hat{r})\hat{D} + \hat{G} - \hat{T} \\
\hat{T} &= \hat{\tau}\hat{Y} + \hat{\tau}\hat{r}\hat{D}.
\end{aligned} \tag{114.12}
$$

These imply

$$
\begin{aligned}
\hat{K} &= \left[\left(1-\hat{\tau}\right)\left(1-\alpha\right)\left(1-\beta\right)\right]^{\frac{1}{1-\alpha}} \\
\hat{\tau} &= \frac{\hat{G} + \hat{r}\hat{D}}{\hat{Y} + \hat{r}\hat{D}}
\end{aligned}
$$

Let's take an example in which

1. there is no initial government debt, $D_t = 0$,
2. government consumption G_t equals 15% of output Y_t

Our formulas for steady-state values tell us that

$$
\begin{aligned}
\hat{D} &= 0 \\
\hat{G} &= 0.15\hat{Y} \\
\hat{\tau} &= 0.15
\end{aligned}
$$

114.10.2 Implementation

```
import numpy as np
import matplotlib.pyplot as plt
from numba import jit
from quantecon.optimize import brent_max
```

For parameters $\alpha = 0.3$ and $\beta = 0.5$, let's compute $\hat{K}$:

```
# parameters
α = 0.3
β = 0.5

# steady states of τ and D
τ_hat = 0.15
D_hat = 0.

# solve for steady state of K
K_hat = ((1 - τ_hat) * (1 - α) * (1 - β)) ** (1 / (1 - α))
K_hat
```

```
0.17694509514972878
```

Knowing $\hat{K}$, we can calculate other equilibrium objects.

Let's first define some Python helper functions.

```
@jit
def K_to_Y(K, α):

    return K ** α

@jit
def K_to_r(K, α):

    return α * K ** (α - 1)

@jit
def K_to_W(K, α):

    return (1 - α) * K ** α

@jit
def K_to_C(K, D, τ, r, α, β):

    # optimal consumption for the old when δ=0
    A = K + D
    Co = A * (1 + r * (1 - τ))

    # optimal consumption for the young when δ=0
    W = K_to_W(K, α)
    Cy = β * W * (1 - τ)

    return Cy, Co
```

We can use these helper functions to obtain steady state values $\hat{Y}$, $\hat{r}$, and $\hat{W}$ associated with steady state values $\hat{K}$ and $\hat{r}$.

```
Y_hat, r_hat, W_hat = K_to_Y(K_hat, α), K_to_r(K_hat, α), K_to_W(K_hat, α)
Y_hat, r_hat, W_hat
```

```
(0.5947734290747186, 1.0084033613445376, 0.41634140035230305)
```

Since steady state government debt $\hat{D}$ is 0, all taxes are used to pay for government expenditures

```
G_hat = τ_hat * Y_hat
G_hat
```

```
0.0892160143612078
```

We use the optimal consumption plans to find steady state consumptions for young and old

```
Cy_hat, Co_hat = K_to_C(K_hat, D_hat, τ_hat, r_hat, α, β)
Cy_hat, Co_hat
```

```
(0.17694509514972878, 0.32861231956378195)
```

Let's store the steady state quantities and prices using an array called `init_ss`

```
init_ss = np.array([K_hat, Y_hat, Cy_hat, Co_hat,      # quantities
                    W_hat, r_hat,                      # prices
                    τ_hat, D_hat, G_hat                # policies
                    ])
```

114.10.3 Transitions

We have computed a steady state in which the government policy sequences are each constant over time.

We'll use this steady state as an initial condition at time $t = 0$ for another economy in which government policy sequences are with time-varying sequences.

To make sense of our calculation, we'll treat $t = 0$ as time when a huge unanticipated shock occurs in the form of

- a time-varying government policy sequences that disrupts an original steady state
- new government policy sequences are eventually time-invariant in the sense that after some date $T > 0$, each sequence is constant over time.
- sudden revelation of a new government policy in the form of sequences starting at time $t = 0$

We assume that everyone, including old people at time $t = 0$, knows the new government policy sequence and chooses accordingly.

As the capital stock and other aggregates adjust to the fiscal policy change over time, the economy will approach a new steady state.

We can find a transition path from an old steady state to a new steady state by employing a fixed-point algorithm in a space of sequences.

But in our special case with its closed form solution, we have available a simpler and faster approach.

Here we define a Python class `ClosedFormTrans` that computes length T transition path in response to a particular fiscal policy change.

We choose T large enough so that we have gotten very close to a new steady state after T periods.

The class takes three keyword arguments, `τ_pol`, `D_pol`, and `G_pol`.

These are sequences of tax rate, government debt level, and government purchases, respectively.

In each policy experiment below, we will pass two out of three as inputs required to depict a fiscal policy.

We'll then compute the single remaining undetermined policy variable from the government budget constraint.

When we simulate transition paths, it is useful to distinguish **state variables** at time t such as K_t, Y_t, D_t, W_t, r_t from **control variables** that include $C_{yt}, C_{ot}, \tau_t, G_t$.

```
class ClosedFormTrans:
    """
    This class simulates length T transitional path of a economy
    in response to a fiscal policy change given its initial steady
    state. The simulation is based on the closed form solution when
    the lump sum taxations are absent.

    """

    def __init__(self, α, β):

        self.α, self.β = α, β

    def simulate(self,
                 T,            # length of transitional path to simulate
                 init_ss,      # initial steady state
                 τ_pol=None,   # sequence of tax rates
                 D_pol=None,   # sequence of government debt levels
                 G_pol=None): # sequence of government purchases

        α, β = self.α, self.β

        # unpack the steady state variables
        K_hat, Y_hat, Cy_hat, Co_hat = init_ss[:4]
        W_hat, r_hat = init_ss[4:6]
        τ_hat, D_hat, G_hat = init_ss[6:9]

        # initialize array containers
        # K, Y, Cy, Co
        quant_seq = np.empty((T+1, 4))

        # W, r
        price_seq = np.empty((T+1, 2))

        # τ, D, G
        policy_seq = np.empty((T+2, 3))

        # t=0, starting from steady state
        K0, Y0 = K_hat, Y_hat
        W0, r0 = W_hat, r_hat
        D0 = D_hat

        # fiscal policy
        if τ_pol is None:
            D1 = D_pol[1]
            G0 = G_pol[0]
            τ0 = (G0 + (1 + r0) * D0 - D1) / (Y0 + r0 * D0)
        elif D_pol is None:
            τ0 = τ_pol[0]
            G0 = G_pol[0]
```

(continues on next page)

(continued from previous page)

```
            D1 = (1 + r0) * D0 + G0 - τ0 * (Y0 + r0 * D0)
        elif G_pol is None:
            D1 = D_pol[1]
            τ0 = τ_pol[0]
            G0 = τ0 * (Y0 + r0 * D0) + D1 - (1 + r0) * D0

        # optimal consumption plans
        Cy0, Co0 = K_to_C(K0, D0, τ0, r0, α, β)

        # t=0 economy
        quant_seq[0, :] = K0, Y0, Cy0, Co0
        price_seq[0, :] = W0, r0
        policy_seq[0, :] = τ0, D0, G0
        policy_seq[1, 1] = D1

        # starting from t=1 to T
        for t in range(1, T+1):

            # transition of K
            K_old, τ_old = quant_seq[t-1, 0], policy_seq[t-1, 0]
            D = policy_seq[t, 1]
            K = K_old ** α * (1 - τ_old) * (1 - α) * (1 - β) - D

            # output, capital return, wage
            Y, r, W = K_to_Y(K, α), K_to_r(K, α), K_to_W(K, α)

            # to satisfy the government budget constraint
            if τ_pol is None:
                D = D_pol[t]
                D_next = D_pol[t+1]
                G = G_pol[t]
                τ = (G + (1 + r) * D - D_next) / (Y + r * D)
            elif D_pol is None:
                τ = τ_pol[t]
                G = G_pol[t]
                D = policy_seq[t, 1]
                D_next = (1 + r) * D + G - τ * (Y + r * D)
            elif G_pol is None:
                D = D_pol[t]
                D_next = D_pol[t+1]
                τ = τ_pol[t]
                G = τ * (Y + r * D) + D_next - (1 + r) * D

            # optimal consumption plans
            Cy, Co = K_to_C(K, D, τ, r, α, β)

            # store time t economy aggregates
            quant_seq[t, :] = K, Y, Cy, Co
            price_seq[t, :] = W, r
            policy_seq[t, 0] = τ
            policy_seq[t+1, 1] = D_next
            policy_seq[t, 2] = G

        self.quant_seq = quant_seq
        self.price_seq = price_seq
        self.policy_seq = policy_seq
```

(continues on next page)

(continued from previous page)

```
        return quant_seq, price_seq, policy_seq

    def plot(self):

        quant_seq = self.quant_seq
        price_seq = self.price_seq
        policy_seq = self.policy_seq

        fig, axs = plt.subplots(3, 3, figsize=(14, 10))

        # quantities
        for i, name in enumerate(['K', 'Y', 'Cy', 'Co']):
            ax = axs[i//3, i%3]
            ax.plot(range(T+1), quant_seq[:T+1, i], label=name)
            ax.hlines(init_ss[i], 0, T+1, color='r', linestyle='--')
            ax.legend()
            ax.set_xlabel('t')

        # prices
        for i, name in enumerate(['W', 'r']):
            ax = axs[(i+4)//3, (i+4)%3]
            ax.plot(range(T+1), price_seq[:T+1, i], label=name)
            ax.hlines(init_ss[i+4], 0, T+1, color='r', linestyle='--')
            ax.legend()
            ax.set_xlabel('t')

        # policies
        for i, name in enumerate(['τ', 'D', 'G']):
            ax = axs[(i+6)//3, (i+6)%3]
            ax.plot(range(T+1), policy_seq[:T+1, i], label=name)
            ax.hlines(init_ss[i+6], 0, T+1, color='r', linestyle='--')
            ax.legend()
            ax.set_xlabel('t')
```

We can create an instance `closed` for model parameters $\{\alpha, \beta\}$ and use it for various fiscal policy experiments.

```
closed = ClosedFormTrans(α, β)
```

114.10.4 Experiment 1: Tax cut

To illustrate the power of `ClosedFormTrans`, let's first experiment with the following fiscal policy change:

1. at $t = 0$, the government unexpectedly announces a one-period tax cut, $\tau_0 = (1 - \frac{1}{3})\hat{\tau}$, by issuing government debt $\bar{D}$
2. from $t = 1$, the government will keep $D_t = \bar{D}$ and adjust τ_t to collect taxation to pay for the government consumption and interest payments on the debt
3. government consumption G_t will be fixed at $0.15\hat{Y}$

The following equations completely characterize the equilibrium transition path originating from the initial steady state

$$
\begin{aligned}
K_{t+1} &= K_t^{\alpha}\left(1-\tau_t\right)\left(1-\alpha\right)\left(1-\beta\right)-\bar{D} \\
\tau_0 &= (1-\frac{1}{3})\hat{\tau} \\
\bar{D} &= \hat{G}-\tau_0\hat{Y} \\
\tau_t &= \frac{\hat{G}+r_t\bar{D}}{\hat{Y}+r_t\bar{D}}
\end{aligned}
$$

We can simulate the transition for 20 periods, after which the economy will be close to a new steady state.

The first step is to prepare sequences of policy variables that describe fiscal policy.

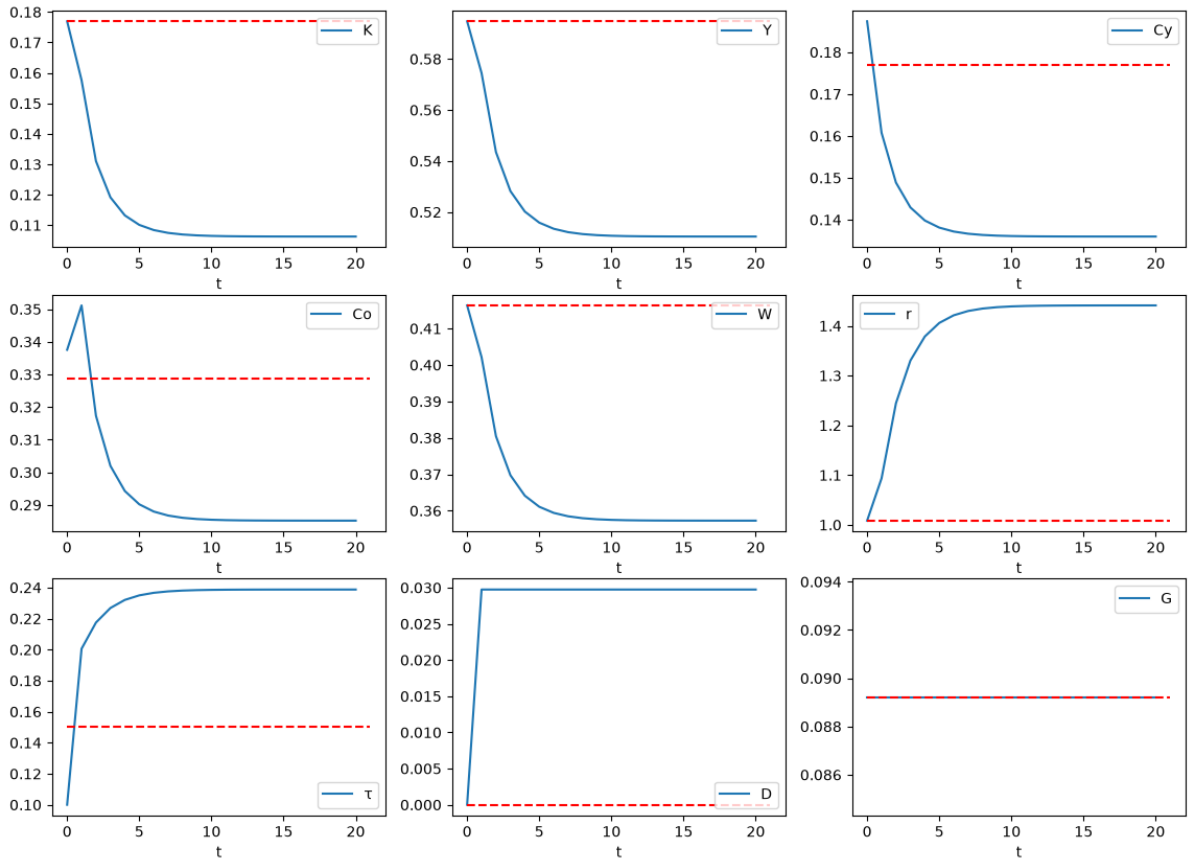
We must define sequences of government expenditure $\{G_t\}_{t=0}^{T}$ and debt level $\{D_t\}_{t=0}^{T+1}$ in advance, then pass them to the solver.

```
T = 20

# tax cut
τ0 = τ_hat * (1 - 1/3)

# sequence of government purchase
G_seq = τ_hat * Y_hat * np.ones(T+1)

# sequence of government debt
D_bar = G_hat - τ0 * Y_hat
D_seq = np.ones(T+2) * D_bar
D_seq[0] = D_hat
```

Let's use the `simulate` method of `closed` to compute dynamic transitions.

Note that we leave `τ_pol` as `None`, since the tax rates need to be determined to satisfy the government budget constraint.

```
quant_seq1, price_seq1, policy_seq1 = closed.simulate(T, init_ss,
                                                      D_pol=D_seq,
                                                      G_pol=G_seq)
closed.plot()
```



We can also experiment with a lower tax cut rate, such as 0.2.

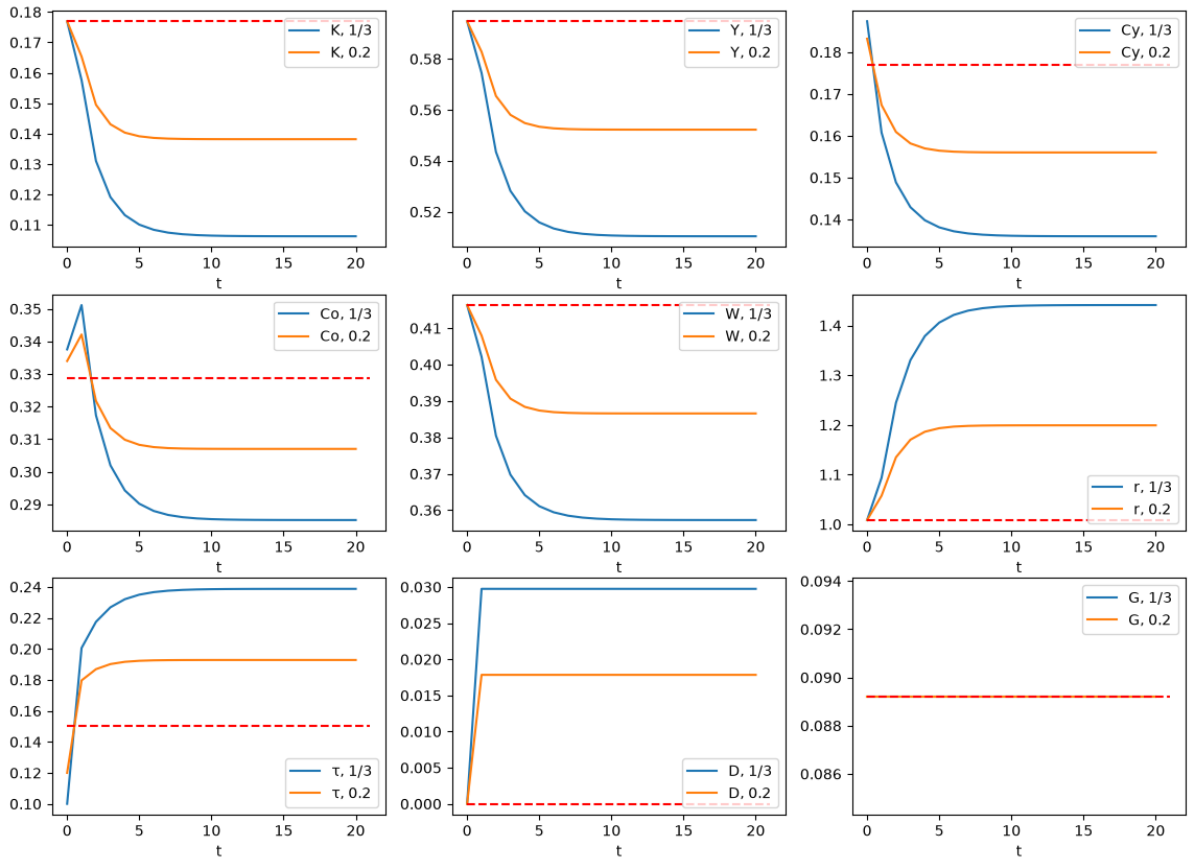
```
# lower tax cut rate
τ0 = 0.15 * (1 - 0.2)

# the corresponding debt sequence
D_bar = G_hat - τ0 * Y_hat
D_seq = np.ones(T+2) * D_bar
D_seq[0] = D_hat

quant_seq2, price_seq2, policy_seq2 = closed.simulate(T, init_ss,
                                                      D_pol=D_seq,
                                                      G_pol=G_seq)
```

```
fig, axs = plt.subplots(3, 3, figsize=(14, 10))

# quantities
for i, name in enumerate(['K', 'Y', 'Cy', 'Co']):
    ax = axs[i//3, i%3]
    ax.plot(range(T+1), quant_seq1[:T+1, i], label=f'{name}, 1/3')
    ax.plot(range(T+1), quant_seq2[:T+1, i], label=f'{name}, 0.2')
    ax.hlines(init_ss[i], 0, T+1, color='r', linestyle='--')
    ax.legend()
    ax.set_xlabel('t')

# prices
for i, name in enumerate(['W', 'r']):
```

(continues on next page)

(continued from previous page)

```
    ax = axs[(i+4)//3, (i+4)%3]
    ax.plot(range(T+1), price_seq1[:T+1, i], label=f'{name}, 1/3')
    ax.plot(range(T+1), price_seq2[:T+1, i], label=f'{name}, 0.2')
    ax.hlines(init_ss[i+4], 0, T+1, color='r', linestyle='--')
    ax.legend()
    ax.set_xlabel('t')

# policies
for i, name in enumerate(['τ', 'D', 'G']):
    ax = axs[(i+6)//3, (i+6)%3]
    ax.plot(range(T+1), policy_seq1[:T+1, i], label=f'{name}, 1/3')
    ax.plot(range(T+1), policy_seq2[:T+1, i], label=f'{name}, 0.2')
    ax.hlines(init_ss[i+6], 0, T+1, color='r', linestyle='--')
    ax.legend()
    ax.set_xlabel('t')
```



The economy with lower tax cut rate at $t = 0$ has the same transitional pattern, but is less distorted, and it converges to a new steady state with higher physical capital stock.

114.10.5 Experiment 2: Government asset accumulation

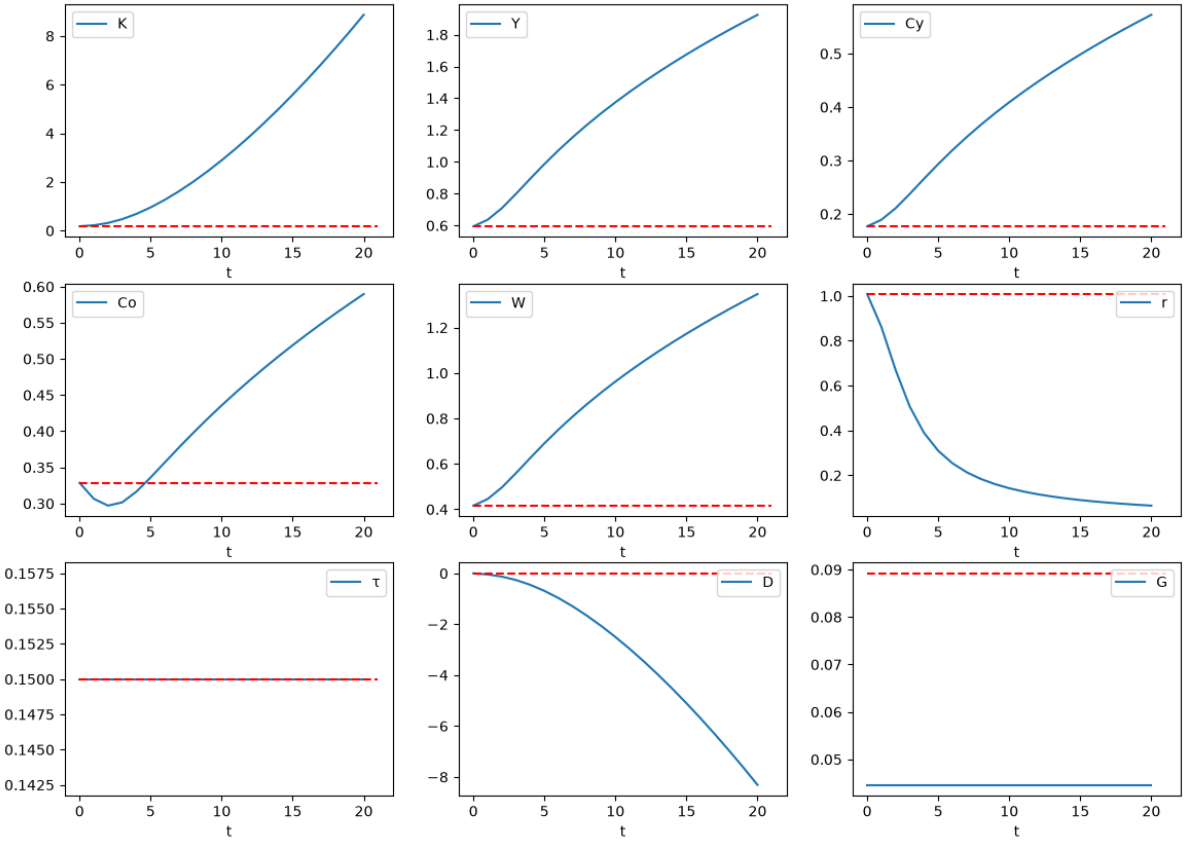
Assume that the economy is initially in the same steady state.

Now the government promises to cut its spending on services and goods by half $\forall t \geq 0$.

The government targets the same tax rate $\tau_t = \hat{\tau}$ and to accumulate assets $-D_t$ over time.

To conduct this experiment, we pass `τ_seq` and `G_seq` as inputs and let `D_pol` be determined along the path by satisfying the government budget constraint.

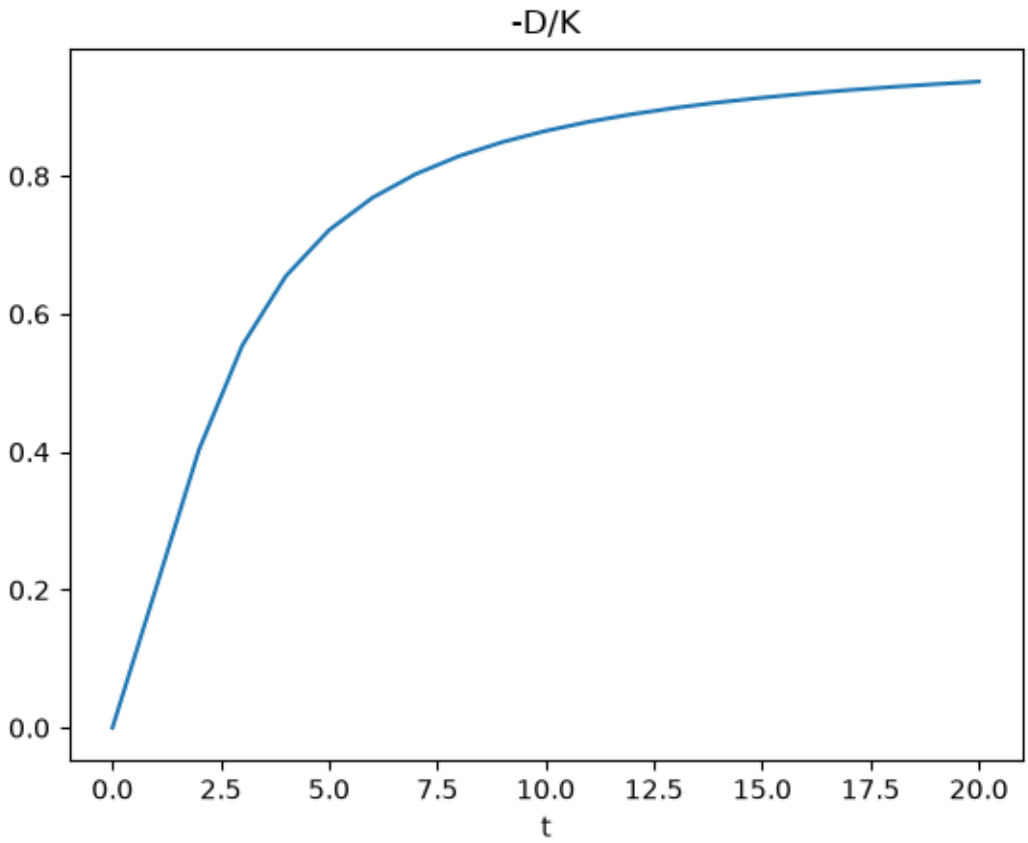
```
# government expenditure cut by a half
G_seq = τ_hat * 0.5 * Y_hat * np.ones(T+1)

# targeted tax rate
τ_seq = τ_hat * np.ones(T+1)

closed.simulate(T, init_ss, τ_pol=τ_seq, G_pol=G_seq);
closed.plot()
```



As the government accumulates the asset and uses it in production, the rental rate on capital falls and private investment falls.

As a result, the ratio $-\frac{D_t}{K_t}$ of the government asset to physical capital used in production will increase over time

```
plt.plot(range(T+1), -closed.policy_seq[:-1, 1] / closed.quant_seq[:, 0])
plt.xlabel('t')
plt.title('-D/K');
```



We want to know how this policy experiment affects individuals.

In the long run, future cohorts will enjoy higher consumption throughout their lives because they will earn higher labor income when they work.

However, in the short run, old people suffer because increases in their labor income are not big enough to offset their losses of capital income.

Such distinct long run and short run effects motivate us to study transition paths.

Note

Although the consumptions in the new steady state are strictly higher, it is at a cost of fewer public services and goods.

114.10.6 Experiment 3: Temporary expenditure cut

Let's now investigate a scenario in which the government also cuts its spending by half and accumulates the asset.

But now let the government cut its expenditures only at $t = 0$.

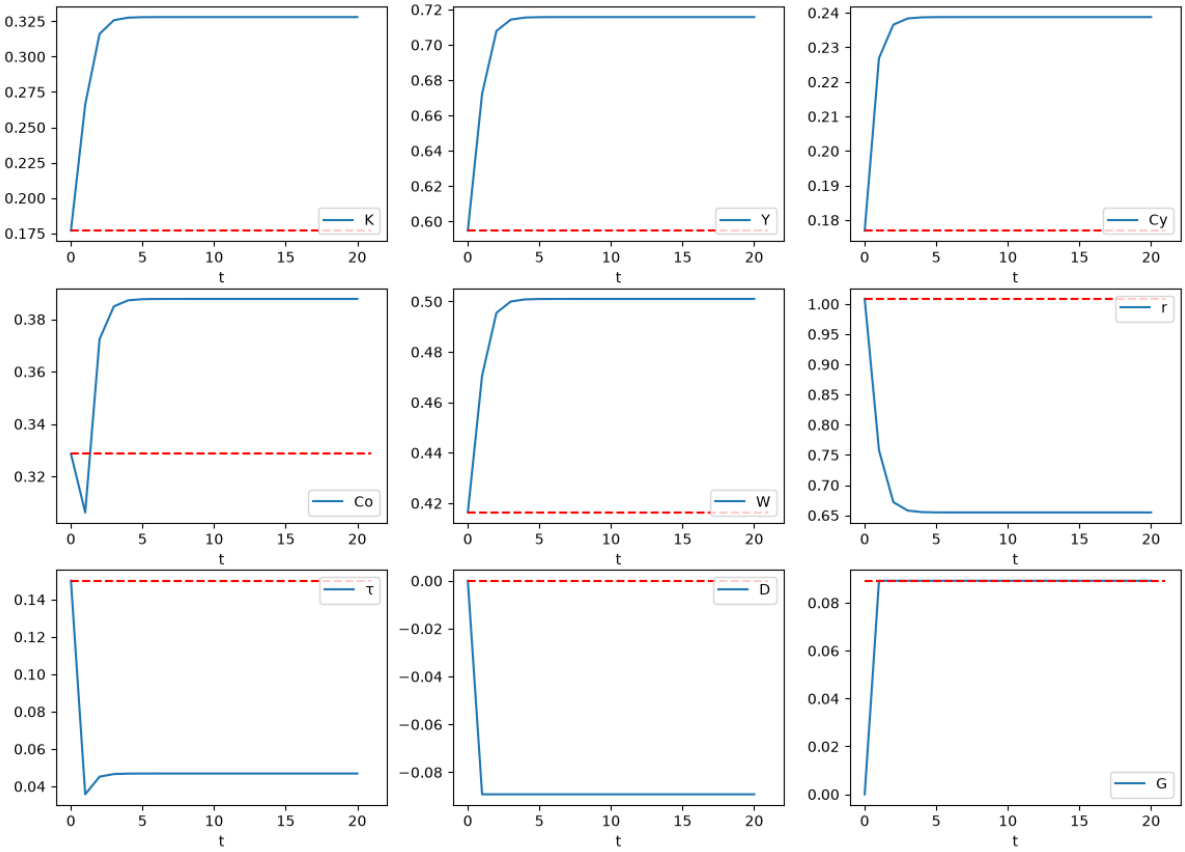
From $t \geq 1$, the government expeditures return to $\hat{G}$ and τ_t adjusts to maintain the asset level $-D_t = -D_1$.

```
# sequence of government purchase
G_seq = τ_hat * Y_hat * np.ones(T+1)
G_seq[0] = 0
```

(continues on next page)

(continued from previous page)

```
# sequence of government debt
D_bar = G_seq[0] - τ_hat * Y_hat
D_seq = D_bar * np.ones(T+2)
D_seq[0] = D_hat

closed.simulate(T, init_ss, D_pol=D_seq, G_pol=G_seq);
closed.plot()
```



The economy quickly converges to a new steady state with higher physical capital stock, lower interest rate, higher wage rate, and higher consumptions for both the young and the old.

Even though government expenditure G_t returns to its high initial level from $t \geq 1$, the government can balance the budget at a lower tax rate because it gathers additional revenue $-r_t D_t$ from the asset accumulated during the temporary cut in the spendings.

As in *Experiment 2: Government asset accumulation*, old perople early in the transition periods suffer from this policy shock.

114.11 A computational strategy

With the preceding caluations, we studied dynamic transitions instigated by alternative fiscal policies.

In all these experiments, we maintained the assumption that lump sum taxes were absent so that $\delta_{yt} = 0, \delta_{ot} = 0$.

In this section, we investigate the transition dynamics when the lump sum taxes are present.

The government will use lump sum taxes and transfers to redistribute resources across successive generations.

Including lump sum taxes disrupts closed form solution because of how they make optimal consumption and saving plans depend on future prices and tax rates.

Therefore, we compute equilibrium transitional paths by finding a fixed point of a mapping from sequences to sequences.

- that fixed point pins down an equilibrium

To set the stage for the entry of the mapping whose fixed point we seek, we return to concepts introduced in section *Equilbrium*.

Definition: Given parameters $\{\alpha, \beta\}$, a competitive equilibrium consists of

- sequences of optimal consumptions $\{C_{yt}, C_{ot}\}$
- sequences of prices $\{W_t, r_t\}$
- sequences of capital stock and output $\{K_t, Y_t\}$
- sequences of tax rates, government assets (debt), government purchases $\{\tau_t, D_t, G_t \, \delta_{yt}, \delta_{ot}\}$

with the properties that

- given the price system and government fiscal policy, consumption plans are optimal
- the government budget constraints are satisfied for all t

An equilibrium transition path can be computed by "guessing and verifying" some endogenous sequences.

In our *Experiment 1: Tax cut* example, sequences $\{D_t\}_{t=0}^T$ and $\{G_t\}_{t=0}^T$ are exogenous.

In addition, we assume that the lump sum taxes $\{\delta_{yt}, \delta_{ot}\}_{t=0}^T$ are given and known to everybody inside the model.

We can solve for sequences of other equilibrium sequences following the steps below

1. guess prices $\{W_t, r_t\}_{t=0}^T$ and tax rates $\{\tau_t\}_{t=0}^T$
2. solve for optimal consumption and saving plans $\{C_{yt}, C_{ot}\}_{t=0}^T$, treating the guesses of future prices and taxes as true
3. solve for transition of the capital stock $\{K_t\}_{t=0}^T$
4. update the guesses for prices and tax rates with the values implied by the equilibrium conditions
5. iterate until convergence

Let's implement this "guess and verify" approach

We start by defining the Cobb-Douglas utility function

```
@jit
def U(Cy, Co, β):

    return (Cy ** β) * (Co ** (1-β))
```

We use `Cy_val` to compute the lifetime value of an arbitrary consumption plan, C_y, given the intertemporal budget constraint.

Note that it requires knowing future prices r_{t+1} and tax rate τ_{t+1}.

```
@jit
def Cy_val(Cy, W, r_next, τ, τ_next, δy, δo_next, β):

    # Co given by the budget constraint
    Co = (W * (1 - τ) - δy - Cy) * (1 + r_next * (1 - τ_next)) - δo_next

    return U(Cy, Co, β)
```

An optimal consumption plan C_y^* can be found by maximizing `Cy_val`.

Here is an example that computes optimal consumption $C_y^* = \hat{C}_y$ in the steady state with $\delta_{yt} = \delta_{ot} = 0$, like one that we studied earlier

```
W, r_next, τ, τ_next = W_hat, r_hat, τ_hat, τ_hat
δy, δo_next = 0, 0

Cy_opt, U_opt, _ = brent_max(Cy_val,             # maximand
                             1e-6,               # lower bound
                             W*(1-τ)-δy-1e-6,    # upper bound
                             args=(W, r_next, τ, τ_next, δy, δo_next, β))

Cy_opt, U_opt
```

```
(0.17694509514972878, 0.241135518231111)
```

Let's define a Python class `AK2` that computes the transition paths with the fixed-point algorithm.

It can handle nonzero lump sum taxes

```
class AK2():
    """
    This class simulates length T transitional path of a economy
    in response to a fiscal policy change given its initial steady
    state. The transitional path is found by employing a fixed point
    algorithm to satisfy the equilibrium conditions.

    """

    def __init__(self, α, β):

        self.α, self.β = α, β

    def simulate(self,
                 T,           # length of transitional path to simulate
                 init_ss,     # initial steady state
                 δy_seq,      # sequence of lump sum tax for the young
                 δo_seq,      # sequence of lump sum tax for the old
                 τ_pol=None,  # sequence of tax rates
                 D_pol=None,  # sequence of government debt levels
                 G_pol=None,  # sequence of government purchases
                 verbose=False,
                 max_iter=500,
                 tol=1e-5):
```

(continues on next page)

(continued from previous page)

```
        α, β = self.α, self.β

        # unpack the steady state variables
        K_hat, Y_hat, Cy_hat, Co_hat = init_ss[:4]
        W_hat, r_hat = init_ss[4:6]
        τ_hat, D_hat, G_hat = init_ss[6:9]

        # K, Y, Cy, Co
        quant_seq = np.empty((T+2, 4))

        # W, r
        price_seq = np.empty((T+2, 2))

        # τ, D, G
        policy_seq = np.empty((T+2, 3))
        policy_seq[:, 1] = D_pol
        policy_seq[:, 2] = G_pol

        # initial guesses of prices
        price_seq[:, 0] = np.ones(T+2) * W_hat
        price_seq[:, 1] = np.ones(T+2) * r_hat

        # initial guesses of policies
        policy_seq[:, 0] = np.ones(T+2) * τ_hat

        # t=0, starting from steady state
        quant_seq[0, :2] = K_hat, Y_hat

        if verbose:
            # prepare to plot iterations until convergence
            fig, axs = plt.subplots(1, 3, figsize=(14, 4))

        # containers for checking convergence
        price_seq_old = np.empty_like(price_seq)
        policy_seq_old = np.empty_like(policy_seq)

        # start iteration
        i_iter = 0
        while True:

            if verbose:
                # plot current prices at ith iteration
                for i, name in enumerate(['W', 'r']):
                    axs[i].plot(range(T+1), price_seq[:T+1, i])
                    axs[i].set_title(name)
                    axs[i].set_xlabel('t')
                axs[2].plot(range(T+1), policy_seq[:T+1, 0],
                            label=f'{i_iter}th iteration')
                axs[2].legend(bbox_to_anchor=(1.05, 1), loc='upper left')
                axs[2].set_title('τ')
                axs[2].set_xlabel('t')

            # store old prices from last iteration
            price_seq_old[:] = price_seq
            policy_seq_old[:] = policy_seq
```

(continues on next page)

(continued from previous page)

```
            # start updating quantities and prices
            for t in range(T+1):
                K, Y = quant_seq[t, :2]
                W, r = price_seq[t, :]
                r_next = price_seq[t+1, 1]
                τ, D, G = policy_seq[t, :]
                τ_next, D_next, G_next = policy_seq[t+1, :]
                δy, δo = δy_seq[t], δo_seq[t]
                δy_next, δo_next = δy_seq[t+1], δo_seq[t+1]

                # consumption for the old
                Co = (1 + r * (1 - τ)) * (K + D) - δo

                # optimal consumption for the young
                out = brent_max(Cy_val, 1e-6, W*(1-τ)-δy-1e-6,
                                args=(W, r_next, τ, τ_next,
                                      δy, δo_next, β))
                Cy = out[0]

                quant_seq[t, 2:] = Cy, Co
                τ_num = ((1 + r) * D + G - D_next - δy - δo)
                τ_denom = (Y + r * D)
                policy_seq[t, 0] = τ_num / τ_denom

                # saving of the young
                A_next = W * (1 - τ) - δy - Cy

                # transition of K
                K_next = A_next - D_next
                Y_next = K_to_Y(K_next, α)
                W_next, r_next = K_to_W(K_next, α), K_to_r(K_next, α)

                quant_seq[t+1, :2] = K_next, Y_next
                price_seq[t+1, :] = W_next, r_next

            i_iter += 1

            if (np.max(np.abs(price_seq_old - price_seq)) < tol) & \
               (np.max(np.abs(policy_seq_old - policy_seq)) < tol):
                if verbose:
                    print(f"Converge using {i_iter} iterations")
                break

            if i_iter > max_iter:
                if verbose:
                    print(f"Fail to converge using {i_iter} iterations")
                break

        self.quant_seq = quant_seq
        self.price_seq = price_seq
        self.policy_seq = policy_seq

        return quant_seq, price_seq, policy_seq

    def plot(self):

        quant_seq = self.quant_seq
```

(continues on next page)

(continued from previous page)

```
        price_seq = self.price_seq
        policy_seq = self.policy_seq

        fig, axs = plt.subplots(3, 3, figsize=(14, 10))

        # quantities
        for i, name in enumerate(['K', 'Y', 'Cy', 'Co']):
            ax = axs[i//3, i%3]
            ax.plot(range(T+1), quant_seq[:T+1, i], label=name)
            ax.hlines(init_ss[i], 0, T+1, color='r', linestyle='--')
            ax.legend()
            ax.set_xlabel('t')

        # prices
        for i, name in enumerate(['W', 'r']):
            ax = axs[(i+4)//3, (i+4)%3]
            ax.plot(range(T+1), price_seq[:T+1, i], label=name)
            ax.hlines(init_ss[i+4], 0, T+1, color='r', linestyle='--')
            ax.legend()
            ax.set_xlabel('t')

        # policies
        for i, name in enumerate(['τ', 'D', 'G']):
            ax = axs[(i+6)//3, (i+6)%3]
            ax.plot(range(T+1), policy_seq[:T+1, i], label=name)
            ax.hlines(init_ss[i+6], 0, T+1, color='r', linestyle='--')
            ax.legend()
            ax.set_xlabel('t')
```

We can initialize an instance of class `AK2` with model parameters $\{\alpha, \beta\}$ and then use it to conduct fiscal policy experiments.

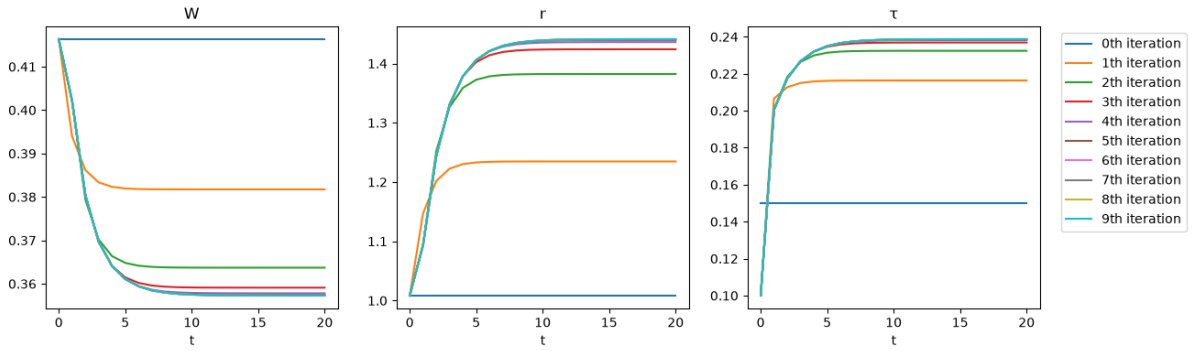
```
ak2 = AK2(α, β)
```

We first examine that the "guess and verify" method leads to the same numerical results as we obtain with the closed form solution when lump sum taxes are muted

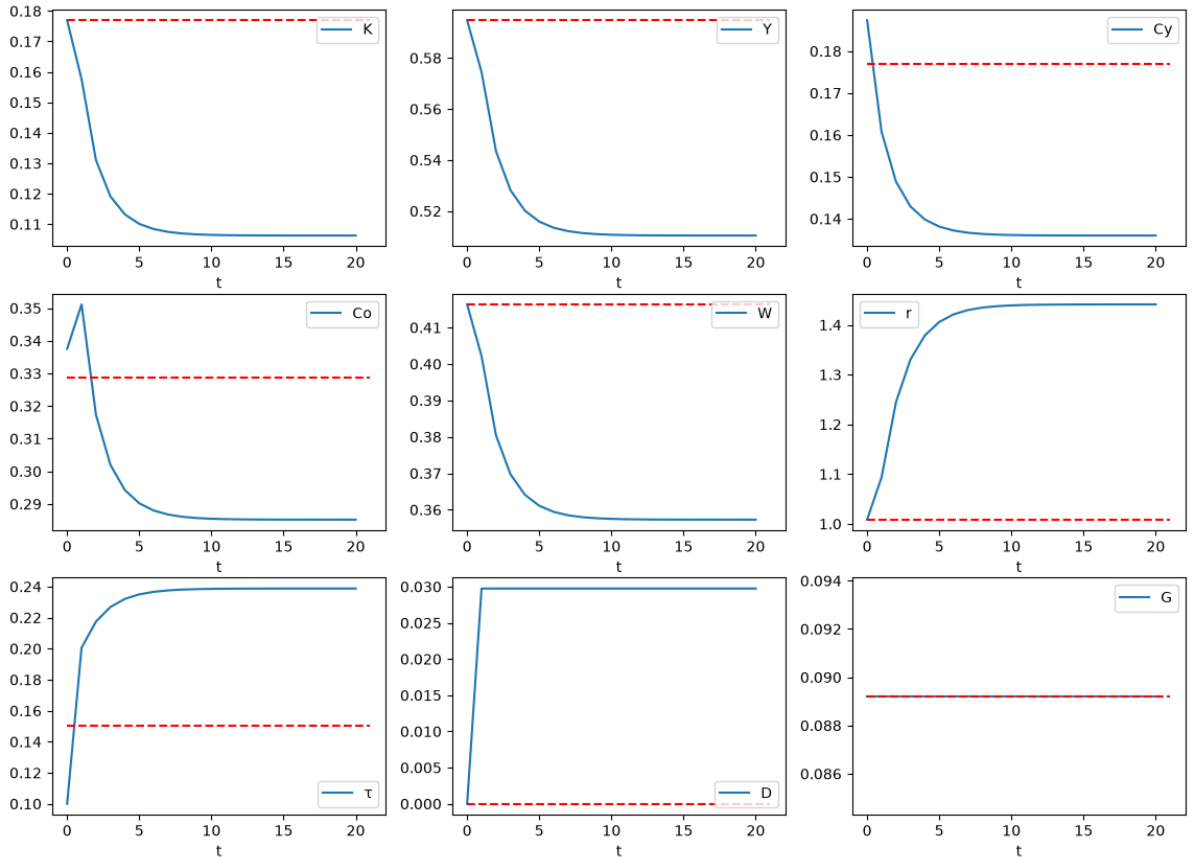
```
δy_seq = np.ones(T+2) * 0.
δo_seq = np.ones(T+2) * 0.

D_pol = np.zeros(T+2)
G_pol = np.ones(T+2) * G_hat

# tax cut
τ0 = τ_hat * (1 - 1/3)
D1 = D_hat * (1 + r_hat * (1 - τ0)) + G_hat - τ0 * Y_hat - δy_seq[0] - δo_seq[0]
D_pol[0] = D_hat
D_pol[1:] = D1
```

```
quant_seq3, price_seq3, policy_seq3 = ak2.simulate(T, init_ss,
                                                   δy_seq, δo_seq,
                                                   D_pol=D_pol, G_pol=G_pol,
                                                   verbose=True)
```

```
Converge using 10 iterations
```




```
ak2.plot()
```



Next, we activate lump sum taxes.

Let's alter our *Experiment 1: Tax cut* fiscal policy experiment by assuming that the government also increases lump sum taxes for both young and old people $\delta_{yt} = \delta_{ot} = 0.005, t \geq 0$.

```
δy_seq = np.ones(T+2) * 0.005
δo_seq = np.ones(T+2) * 0.005

D1 = D_hat * (1 + r_hat * (1 - τ0)) + G_hat - τ0 * Y_hat - δy_seq[0] - δo_seq[0]
D_pol[1:] = D1

quant_seq4, price_seq4, policy_seq4 = ak2.simulate(T, init_ss,
```

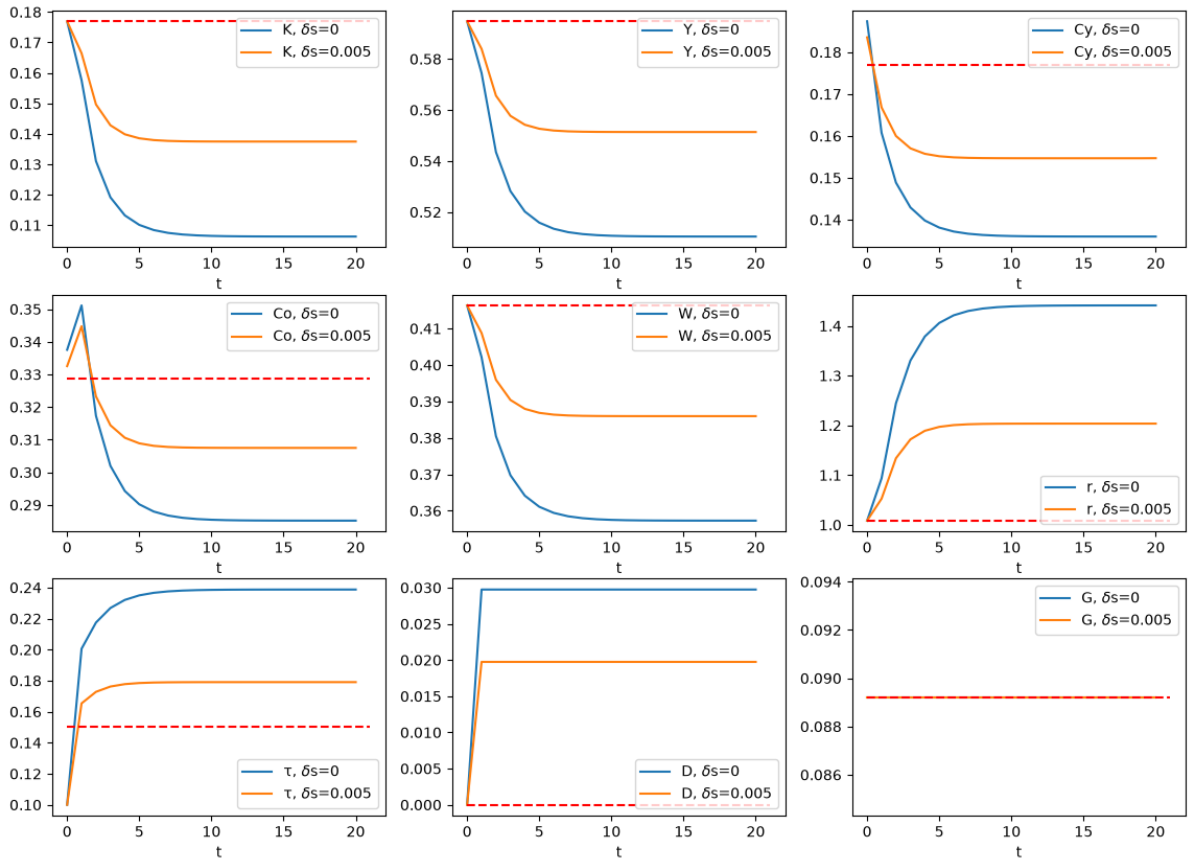
(continues on next page)

(continued from previous page)

```
                                                δy_seq, δo_seq,
                                                D_pol=D_pol, G_pol=G_pol)
```

Note how “crowding out” has been mitigated.

```
fig, axs = plt.subplots(3, 3, figsize=(14, 10))

# quantities
for i, name in enumerate(['K', 'Y', 'Cy', 'Co']):
    ax = axs[i//3, i%3]
    ax.plot(range(T+1), quant_seq3[:T+1, i], label=rf'{name}, $\delta$s=0')
    ax.plot(range(T+1), quant_seq4[:T+1, i], label=rf'{name}, $\delta$s=0.005')
    ax.hlines(init_ss[i], 0, T+1, color='r', linestyle='--')
    ax.legend()
    ax.set_xlabel('t')

# prices
for i, name in enumerate(['W', 'r']):
    ax = axs[(i+4)//3, (i+4)%3]
    ax.plot(range(T+1), price_seq3[:T+1, i], label=rf'{name}, $\delta$s=0')
    ax.plot(range(T+1), price_seq4[:T+1, i], label=rf'{name}, $\delta$s=0.005')
    ax.hlines(init_ss[i+4], 0, T+1, color='r', linestyle='--')
    ax.legend()
    ax.set_xlabel('t')

# policies
for i, name in enumerate(['τ', 'D', 'G']):
    ax = axs[(i+6)//3, (i+6)%3]
    ax.plot(range(T+1), policy_seq3[:T+1, i], label=rf'{name}, $\delta$s=0')
    ax.plot(range(T+1), policy_seq4[:T+1, i], label=rf'{name}, $\delta$s=0.005')
    ax.hlines(init_ss[i+6], 0, T+1, color='r', linestyle='--')
    ax.legend()
    ax.set_xlabel('t')
```



Comparing to *Experiment 1: Tax cut*, the government raises lump-sum taxes to finance the increasing debt interest payment, which is less distortionary comparing to raising the capital income tax rate.

114.11.1 Experiment 4: Unfunded Social Security System

In this experiment, lump-sum taxes are of equal magnitudes for old and the young, but of opposite signs.

A negative lump-sum tax is a subsidy.

Thus, in this experiment we tax the young and subsidize the old.

We start the economy at the same initial steady state that we assumed in several earlier experiments.

The government sets the lump sum taxes $\delta_{y,t} = -\delta_{o,t} = 10\% \hat{C}_y$ starting from $t = 0$.

It keeps debt levels and expenditures at their steady state levels $\hat{D}$ and $\hat{G}$.

In effect, this experiment amounts to launching an unfunded social security system.

We can use our code to compute the transition ignited by launching this system.

Let's compare the results to the *Experiment 1: Tax cut*.

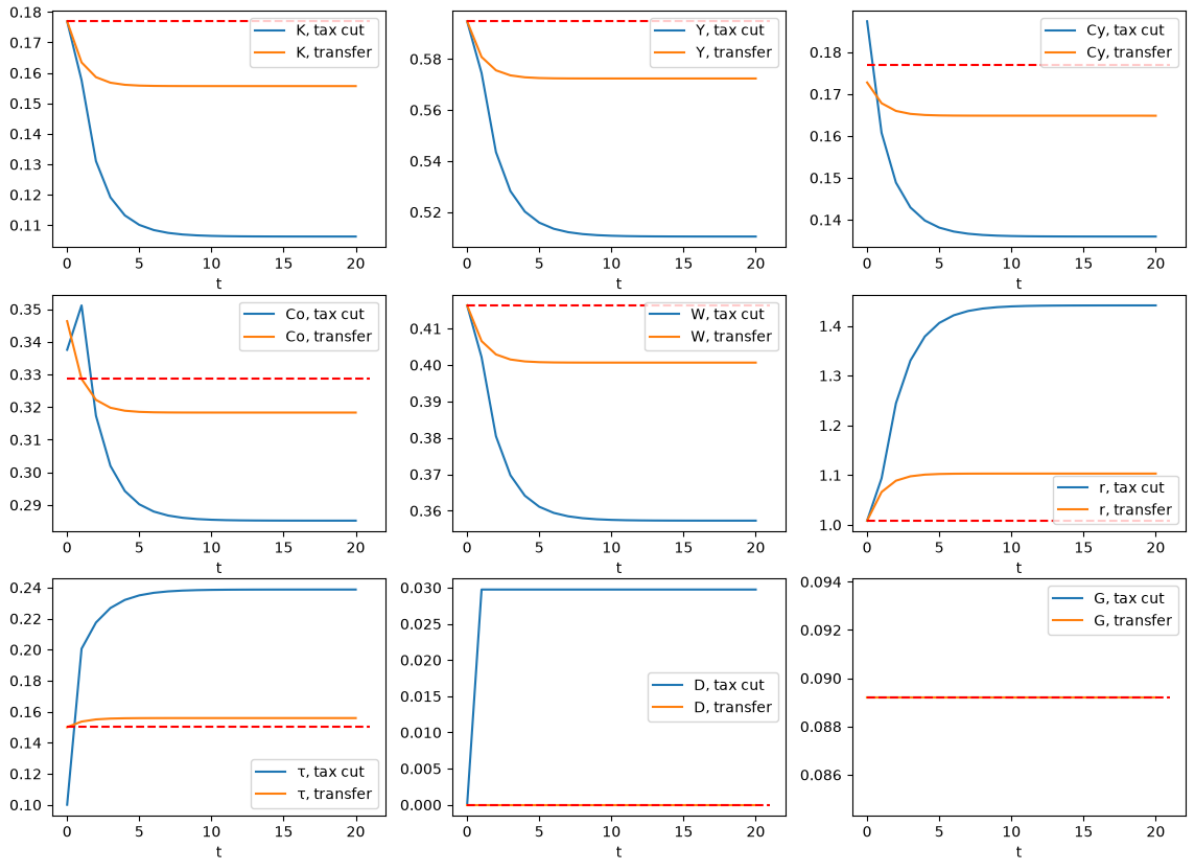
```
δy_seq = np.ones(T+2) * Cy_hat * 0.1
δo_seq = np.ones(T+2) * -Cy_hat * 0.1

D_pol[:] = D_hat

quant_seq5, price_seq5, policy_seq5 = ak2.simulate(T, init_ss,
```

(continues on next page)

(continued from previous page)

```
                                                    δy_seq, δo_seq,
                                                    D_pol=D_pol, G_pol=G_pol)
```

```
fig, axs = plt.subplots(3, 3, figsize=(14, 10))

# quantities
for i, name in enumerate(['K', 'Y', 'Cy', 'Co']):
    ax = axs[i//3, i%3]
    ax.plot(range(T+1), quant_seq3[:T+1, i], label=f'{name}, tax cut')
    ax.plot(range(T+1), quant_seq5[:T+1, i], label=f'{name}, transfer')
    ax.hlines(init_ss[i], 0, T+1, color='r', linestyle='--')
    ax.legend()
    ax.set_xlabel('t')

# prices
for i, name in enumerate(['W', 'r']):
    ax = axs[(i+4)//3, (i+4)%3]
    ax.plot(range(T+1), price_seq3[:T+1, i], label=f'{name}, tax cut')
    ax.plot(range(T+1), price_seq5[:T+1, i], label=f'{name}, transfer')
    ax.hlines(init_ss[i+4], 0, T+1, color='r', linestyle='--')
    ax.legend()
    ax.set_xlabel('t')

# policies
for i, name in enumerate(['τ', 'D', 'G']):
    ax = axs[(i+6)//3, (i+6)%3]
    ax.plot(range(T+1), policy_seq3[:T+1, i], label=f'{name}, tax cut')
    ax.plot(range(T+1), policy_seq5[:T+1, i], label=f'{name}, transfer')
    ax.hlines(init_ss[i+6], 0, T+1, color='r', linestyle='--')
    ax.legend()
    ax.set_xlabel('t')
```



An initial old person benefits especially when the social security system is launched because he receives a transfer but pays nothing for it.

But in the long run, consumption rates of both young and old people decrease because the the social security system decreases incentives to save.

That lowers the stock of physical capital and consequently lowers output.

The government must then raise tax rate in order to pay for its expenditures.

The higher rate on capital income further distorts incentives to save.

CHAPTER

FIVE

A LONG-LIVED, HETEROGENEOUS AGENTS, OVERLAPPING GENERATIONS MODEL

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

In addition to what's in Anaconda, this lecture will need the following library

```
!pip install jax
```

115.1 Overview

This lecture describes an overlapping generations model with these features:

- A competitive equilibrium with incomplete markets determines prices and quantities
- Agents live many periods as in [Auerbach and Kotlikoff, 1987]
- Agents receive idiosyncratic labor productivity shocks that cannot be fully insured as in [Aiyagari, 1994]
- Government fiscal policy instruments include tax rates, debt, and transfers as in chapter 2 of [Auerbach and Kotlikoff, 1987] and *Transitions in an Overlapping Generations Model*
- Among other equilibrium objects, a competitive equilibrium determines a sequence of cross-section densities of heterogeneous agents' consumptions, labor incomes, and savings

We use the model to study:

- How fiscal policies affect different generations
- How market incompleteness promotes precautionary savings
- How life-cycle savings and buffer-stock savings motives interact
- How fiscal policies redistribute resources across and within generations

As prerequisites for this lecture, we recommend two quantecon lectures:

1. Discrete State Dynamic Programming
2. *Transitions in an Overlapping Generations Model*

as well as the optional reading *The Aiyagari Model*

As usual, let's start by importing some Python modules

```
from collections import namedtuple
import numpy as np
import matplotlib.pyplot as plt
import jax.numpy as jnp
import jax.scipy as jsp
import jax
```

115.2 Environment

We start by introducing the economic environment we are operating in.

115.2.1 Demographics and time

We work in discrete time indexed by $t = 0, 1, 2, ...$

Each agent lives for $J = 50$ periods and faces no mortality risk.

We index age by $j = 0, 1, ..., 49$, and the population size remains fixed at $1/J$.

115.2.2 Individuals' state variables

Each agent i of age j at time t is characterized by two state variables: asset holdings $a_{i,j,t}$ and idiosyncratic labor productivity $\gamma_{i,j,t}$.

The idiosyncratic labor productivity process follows a two-state Markov chain that takes values γ_l and γ_h with transition matrix Π.

Newborn agents begin with an initial distribution $\pi = [0.5, 0.5]$ over these productivity states.

115.2.3 Labor supply

An agent with productivity $\gamma_{i,j,t}$ supplies $l(j)\gamma_{i,j,t}$ efficiency units of labor.

$l(j)$ is a deterministic age-specific labor efficiency units profile.

An agent's effective labor supply depends on a life-cycle efficiency profile and an idiosyncratic stochastic process.

115.2.4 Initial conditions

Newborns start with zero assets $a_{i,0,t} = 0$.

Initial idiosyncratic productivities are drawn from distribution π.

Agents leave no bequests and have terminal value function $V_J(a) = 0$.

115.3 Production

A representative firm operates a constant returns to scale Cobb-Douglas production:

$$Y_t = Z_t K_t^\alpha L_t^{1-\alpha}$$

where:

- K_t is aggregate capital
- L_t is aggregate efficiency units of labor
- Z_t is total factor productivity
- α is the capital share

115.4 Government

The government follows a fiscal policy that includes debt, taxes, transfers, and government spending.

The government issues one-period debt D_t to finance its operations and collects revenues through a flat-rate tax τ_t on both labor and capital income.

The government also implements age-specific lump-sum taxes or transfers $\delta_{j,t}$ that can redistribute resources across different age groups.

Additionally, it makes government purchases G_t for public goods and services.

The government budget constraint at time t is

$$D_{t+1} - D_t = r_t D_t + G_t - T_t$$

where total tax revenues T_t satisfy

$$T_t = \tau_t w_t L_t + \tau_t r_t (D_t + K_t) + \sum_j \delta_{j,t}$$

115.5 Activities in factor markets

At each time $t \geq 0$, agents supply labor and capital.

115.5.1 Age-specific labor supplies

Agents of age $j \in \{0, 1, ..., J-1\}$ supply labor according to:

- Their deterministic age-efficiency profile $l(j)$
- Their current idiosyncratic productivity shock $\gamma_{i,j,t}$

Each agent supplies $l(j)\gamma_{i,j,t}$ effective units of labor and earns a competitive wage w_t per effective unit, subject to a flat tax rate τ_t on labor earnings.

115.5.2 Asset market participation

Summarizing activities in the asset market, all agents, regardless of age $j \in \{0, 1, ..., J-1\}$, can:

- Hold assets $a_{i,j,t}$ (subject to borrowing constraints)
- Earn a risk-free one-period return r_t on savings
- Pay capital income taxes at flat rate τ_t
- Receive or pay age-specific transfers $\delta_{j,t}$

115.5.3 Key features

Lifecycle patterns shape economic behavior across ages:

- Labor productivity varies systematically with age according to the profile $l(j)$, while asset holdings typically follow a lifecycle pattern of accumulation during working years and decumulation during retirement.
- Age-specific fiscal transfers $\delta_{j,t}$ redistribute resources across generations.

Within-cohort heterogeneity creates dispersion among agents of the same age:

- Agents of the same age differ in their asset holdings $a_{i,j,t}$ due to different histories of idiosyncratic productivity shocks, their current productivities $\gamma_{i,j,t}$, and consequently their labor incomes and financial wealth.

Cross-cohort interactions determine equilibrium outcomes through market aggregation:

- All cohorts participate together in factor markets, with asset supplies from all cohorts determining aggregate capital and effective labor supplies from all cohorts determining aggregate labor.
- Equilibrium prices reflect both lifecycle and redistributive forces.

115.6 Representative firm's problem

A representative firm chooses capital and effective labor to maximize profits

$$\max_{K,L} Z_t K_t^\alpha L_t^{1-\alpha} - r_t K_t - w_t L_t$$

First-order necessary conditions imply that

$$w_t = (1-\alpha) Z_t (K_t / L_t)^\alpha$$

and

$$r_t = \alpha Z_t (K_t / L_t)^{\alpha-1}$$

115.7 Households' problems

A household's value function satisfies a Bellman equation

$$V_{j,t}(a,\gamma) = \max_{c,a'}\{u(c) + \beta\mathbb{E}[V_{j+1,t+1}(a',\gamma')]\}$$

where maximization is subject to

$$c + a' = (1 + r_t(1-\tau_t))a + (1-\tau_t)w_t l(j)\gamma - \delta_{j,t}$$

$$c \geq 0$$

and a terminal condition $V_{J,t}(a,\gamma) = 0$

115.8 Population dynamics

The joint probability density function $\mu_{j,t}(a,\gamma)$ of asset holdings and idiosyncratic labor productivity evolves according to

- For newborns ($j = 0$):

$$\mu_{0,t+1}(a',\gamma') = \begin{cases} \pi(\gamma') & \text{if } a' = 0, \\ 0, & \text{otherwise} \end{cases}$$

- For other cohorts:

$$\mu_{j+1,t+1}(a',\gamma') = \int \mathbf{1}_{\sigma_{j,t}(a,\gamma)=a'}\Pi(\gamma,\gamma')\mu_{j,t}(a,\gamma)d(a,\gamma)$$

where $\sigma_{j,t}(a,\gamma)$ is the optimal saving policy function.

115.9 Equilibrium

An equilibrium consists of:

- Value functions $V_{j,t}$
- Policy functions $\sigma_{j,t}$
- Joint probability distributions $\mu_{j,t}$
- Prices r_t, w_t
- Government policies $\tau_t, D_t, \delta_{j,t}, G_t$

that satisfy the following conditions

- Given prices and government policies, value and policy functions solve households' problems
- Given prices, the representative firm maximizes profits
- Government budget constraints are satisfied
- Markets clear:
  - Asset market: $K_t = \sum_j \int a\mu_{j,t}(a,\gamma)d(a,\gamma) - D_t$
  - Labor market: $L_t = \sum_j \int l(j)\gamma\mu_{j,t}(a,\gamma)d(a,\gamma)$

Relative to the model presented in *Transitions in an Overlapping Generations Model*, the present model adds

- Heterogeneity within generations due to productivity shocks
- A precautionary savings motive
- More re-distributional effects
- More complicated transition dynamics

115.10 Implementation

Using tools in Discrete State Dynamic Programming, we solve our model by combining value function iteration with equilibrium price determination.

A sensible approach is to nest a discrete DP solver inside an outer loop that searches for market-clearing prices.

For a candidate sequence of prices interest rates r_t and wages w_t, we can solve individual households' dynamic programming problems using either value function iteration or policy iteration to obtain optimal policy functions.

We then deduce associated stationary joint probability distributions of asset holdings and idiosyncratic labor efficiency units for each age cohort.

This will give us an aggregate capital supply (from household savings) and a labor supply (from the age-efficiency profile and productivity shocks).

We can then compare these with capital and labor demand from firms, compute deviations between factor market supplies and demands, then update price guesses until we find market-clearing prices.

To construct transition dynamics, we can compute sequences of time-varying prices by using *backward induction* to compute value and policy functions, and *forward iteration* for the distributions of agents across states:

1. Outer loop (market clearing)
    - Guess initial prices (r_t, w_t)
    - Iterate until asset and labor markets clear
    - Use firms' first-order necessary conditions to update prices
2. Inner loop (individual dynamic programming)
    - For each age cohort:
        - Discretize asset and productivity state space
        - Use value function iteration or policy iteration
        - Solve for optimal savings policies
        - Compute stationary distributions
3. Aggregation
    - Sum across individual states within each cohort
    - Sum across cohorts both
        - Aggregate capital supply, and
        - Aggregate effective labor supply
    - Take into account population weights $1/J$
4. Transition dynamics

- Backward induction:
  - Start from final steady state
  - Solve sequence of value functions
- Forward iteration:
  - Start from initial distribution
  - Track cohort distributions over time
- Market clearing in each period:
  - Solve for price sequences
  - Update until all markets clear in all periods

We start coding by defining helper functions that describe preferences, firms, and government budget constraints.

```
ϕ, k_bar = 0., 0.

@jax.jit
def V_bar(a):
    "Terminal value function depending on the asset holding."

    return - ϕ * (a - k_bar) ** 2
```

```
ν = 0.5

@jax.jit
def u(c):
    "Utility from consumption."

    return c ** (1 - ν) / (1 - ν)

l1, l2, l3 = 0.5, 0.05, -0.0008

@jax.jit
def l(j):
    "Age-specific wage profile."

    return l1 + l2 * j + l3 * j ** 2
```

Let's define a `Firm` namedtuple that contains parameters governing the production technology.

```
Firm = namedtuple("Firm", ("α", "Z"))

def create_firm(α=0.3, Z=1):

    return Firm(α=α, Z=Z)
```

```
firm = create_firm()
```

The following helper functions link aggregates (K, L) and prices (w, r) that emerge from the representative firm's first-order necessary conditions.

```
@jax.jit
def KL_to_r(K, L, firm):
```

(continues on next page)

(continued from previous page)

```
    α, Z = firm

    return Z * α * (K / L) ** (α - 1)

@jax.jit
def KL_to_w(K, L, firm):

    α, Z = firm

    return Z * (1 - α) * (K / L) ** α
```

We use a function `find_τ` to find flat tax rates that balance the government budget constraint given other policy variables that include debt levels, government spending, and transfers.

```
@jax.jit
def find_τ(policy, price, aggs):

    D, D_next, G, δ = policy
    r, w = price
    K, L = aggs

    num = r * D + G - D_next + D - δ.sum(axis=-1)
    denom = w * L + r * (D + K)

    return num / denom
```

We use a namedtuple `Household` to store parameters that characterize households' problems.

```
Household = namedtuple("Household", ("j_grid", "a_grid", "y_grid",
                                     "Π", "β", "init_μ", "VJ"))

def create_household(
        a_min=0., a_max=10, a_size=200,
        Π=[[0.9, 0.1], [0.1, 0.9]],
        y_grid=[0.5, 1.5],
        β=0.96, J=50
    ):

    j_grid = jnp.arange(J)

    a_grid = jnp.linspace(a_min, a_max, a_size)

    y_grid, Π = map(jnp.array, (y_grid, Π))
    y_size = len(y_grid)

    # Population distribution of new borns
    init_μ = jnp.zeros((a_size * y_size))

    # Newborns are endowed with zero asset
    # and equal probability of y
    init_μ = init_μ.at[:y_size].set(1 / y_size)

    # Terminal value V_bar(a)
    VJ = jnp.empty(a_size * y_size)
    for a_i in range(a_size):
        a = a_grid[a_i]
        VJ = VJ.at[a_i*y_size:(a_i+1)*y_size].set(V_bar(a))
```

(continues on next page)

(continued from previous page)

```
    return Household(j_grid=j_grid, a_grid=a_grid, y_grid=y_grid,
                     Π=Π, β=β, init_μ=init_μ, VJ=VJ)
```

```
hh = create_household()
```

We apply discrete state dynamic programming tools.

Initial steps involve preparing rewards and transition matrices R and Q for our discretized Bellman equations.

```
@jax.jit
def populate_Q(household):

    j_grid, a_grid, y_grid, Π, β, init_μ, VJ = household

    num_state = a_grid.size * y_grid.size
    num_action = a_grid.size

    Q = jsp.linalg.block_diag(*[Π]*a_grid.size)
    Q = Q.reshape((num_state, num_action, y_grid.size))
    Q = jnp.tile(Q, a_grid.size).T

    return Q

@jax.jit
def populate_R(j, r, w, τ, δ, household):

    j_grid, a_grid, y_grid, Π, β, init_μ, VJ = household

    num_state = a_grid.size * y_grid.size
    num_action = a_grid.size

    a = jnp.reshape(a_grid, (a_grid.size, 1, 1))
    y = jnp.reshape(y_grid, (1, y_grid.size, 1))
    ap = jnp.reshape(a_grid, (1, 1, a_grid.size))
    c = (1 + r*(1-τ)) * a + (1-τ) * w * l(j) * y - δ[j] - ap

    return jnp.reshape(jnp.where(c > 0, u(c), -jnp.inf),
                       (num_state, num_action))
```

115.11 Computing a steady state

We first compute a steady state.

Given guesses of prices and taxes, we can use backwards induction to solve for value functions and optimal consumption and saving policies at all ages.

The function `backwards_opt` solves for optimal values by applying the discretized bellman operator backwards.

We use `jax.lax.scan` to facilitate sequential and recurrent computations efficiently.

```
@jax.jit
def backwards_opt(prices, taxes, household, Q):

    r, w = prices
```

(continues on next page)

(continued from previous page)

```
    τ, δ = taxes

    j_grid, a_grid, y_grid, Π, β, init_μ, VJ = household
    J = j_grid.size

    num_state = a_grid.size * y_grid.size
    num_action = a_grid.size

    def bellman_operator_j(V_next, j):
        "Solve household optimization problem at age j given Vj+1"

        Rj = populate_R(j, r, w, τ, δ, household)
        vals = Rj + β * Q.dot(V_next)
        σ_j = jnp.argmax(vals, axis=1)
        V_j = vals[jnp.arange(num_state), σ_j]

        return V_j, (V_j, σ_j)

    js = jnp.arange(J-1, -1, -1)
    init_V = VJ

    # Iterate from age J to 1
    _, outputs = jax.lax.scan(bellman_operator_j, init_V, js)
    V, σ = outputs
    V = V[::-1]
    σ = σ[::-1]

    return V, σ
```

```
r, w = 0.05, 1
τ, δ = 0.15, np.zeros(hh.j_grid.size)

Q = populate_Q(hh)
```

```
V, σ = backwards_opt([r, w], [τ, δ], hh, Q)
```

Let's time the computation with `block_until_ready()` to ensure that all JAX operations are complete

```
%time backwards_opt([r, w], [τ, δ], hh, Q)[0].block_until_ready();
```

```
CPU times: user 1.61 ms, sys: 0 ns, total: 1.61 ms
Wall time: 51 ms
```

From the optimal consumption and saving choices by each cohort, we can compute a joint probability distribution of asset levels and idiosyncratic productivity levels in a steady state.

```
@jax.jit
def popu_dist(σ, household, Q):

    j_grid, a_grid, y_grid, Π, β, init_μ, VJ = household

    J = hh.j_grid.size
    num_state = hh.a_grid.size * hh.y_grid.size

    def update_popu_j(μ_j, j):
```

(continues on next page)

(continued from previous page)

```
        "Update population distribution from age j to j+1"

        Qσ = Q[jnp.arange(num_state), σ[j]]
        μ_next = μ_j @ Qσ

        return μ_next, μ_next

    js = jnp.arange(J-1)

    # iterate from age 1 to J
    _, μ = jax.lax.scan(update_popu_j, init_μ, js)
    μ = jnp.concatenate([init_μ[jnp.newaxis], μ], axis=0)

    return μ
```

```
μ = popu_dist(σ, hh, Q)
```

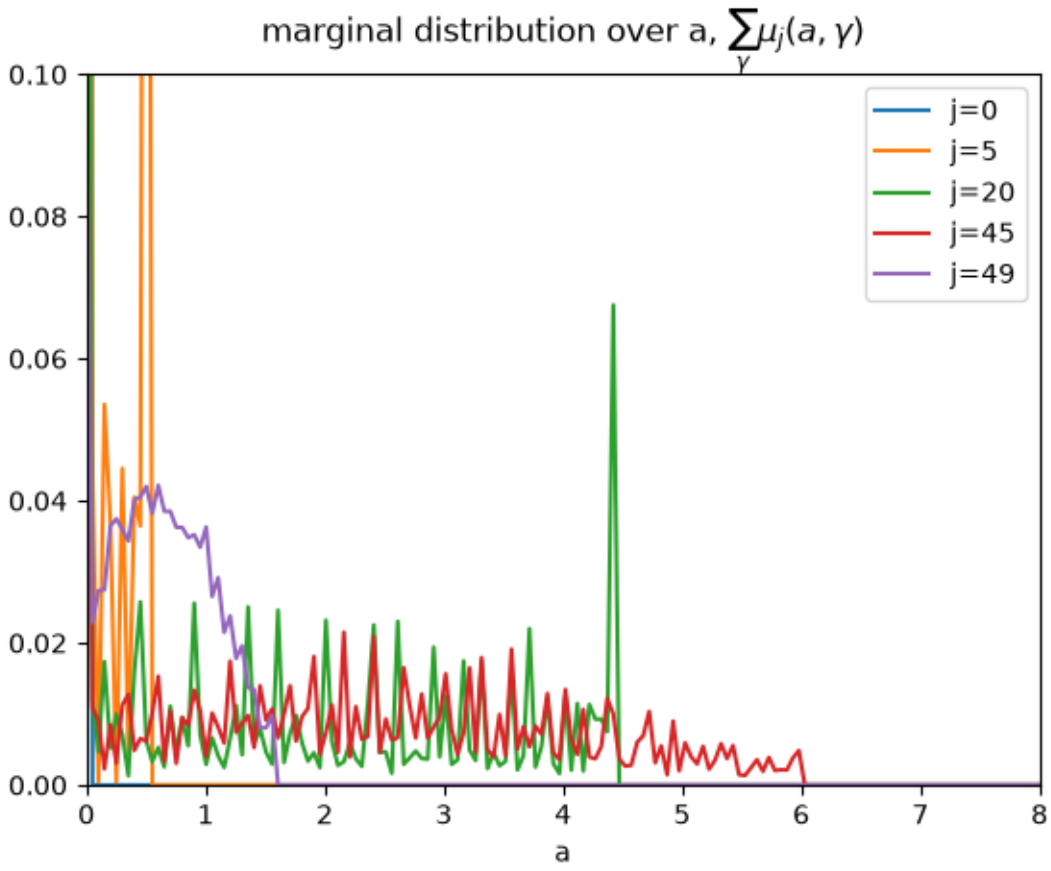
Let's time the computation

```
%time popu_dist(σ, hh, Q)[0].block_until_ready();
```

```
CPU times: user 44.8 ms, sys: 786 µs, total: 45.6 ms
Wall time: 58.8 ms
```

Below we plot the marginal distribution of savings for each age group.

```
for j in [0, 5, 20, 45, 49]:
    plt.plot(hh.a_grid, jnp.sum(μ[j].reshape((hh.a_grid.size, hh.y_grid.size)),␣
 ↪axis=1), label=f'j={j}')

plt.legend()
plt.xlabel('a')

plt.title(r'marginal distribution over a, $\sum_\gamma \mu_j(a, \gamma)$')
plt.xlim([0, 8])
plt.ylim([0, 0.1])

plt.show()
```



These marginal distributions confirm that new agents enter the economy with no asset holdings.

- the blue $j = 0$ distribution has mass only at $a = 0$.

As agents age, at first they gradually accumulate assets.

- the orange $j = 5$ distribution puts positive mass on positive but low asset levels
- the green $j = 20$ distribution puts positive mass on a much wider range of asset levels.
- the red $j = 45$ distribution is even wider

At a later age, they gradually deplete their asset holdings.

- the purple $j = 49$ distribution illustrates this

At the end of life, they will have drawn down all of their assets.

Let's now look at age-specific optimal saving policies that generate the preceding marginal distributions of assets at different ages.

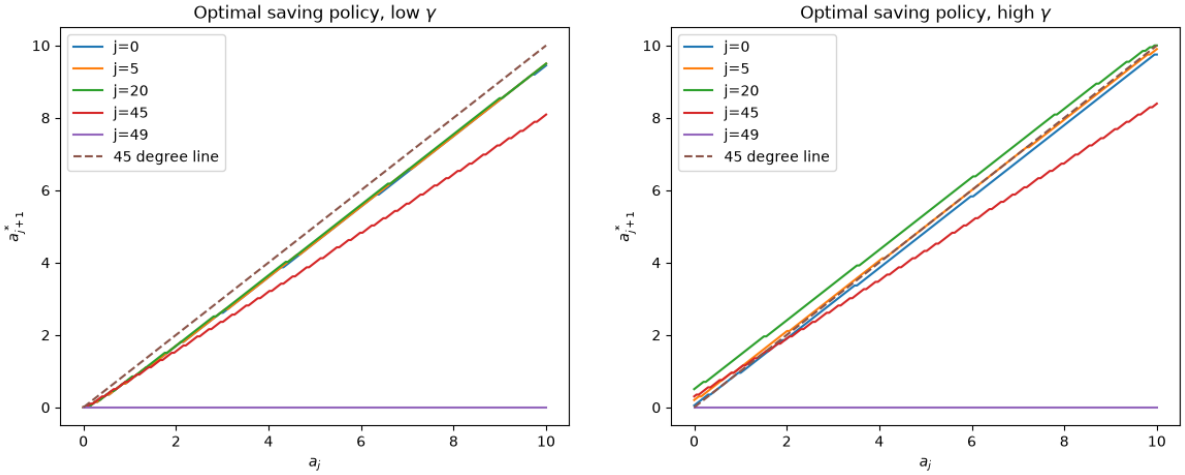
We'll plot some saving functions with the following Python code.

```
σ_reshaped = σ.reshape(hh.j_grid.size, hh.a_grid.size, hh.y_grid.size)
j_labels = [f'j={j}' for j in [0, 5, 20, 45, 49]]

fig, axs = plt.subplots(1, 2, figsize=(14, 5))

axs[0].plot(hh.a_grid, hh.a_grid[σ_reshaped[[0, 5, 20, 45, 49], :, 0].T])
axs[0].plot(hh.a_grid, hh.a_grid, '--')
```

(continues on next page)

(continued from previous page)

```
axs[0].set_xlabel("$a_{j}$")
axs[0].set_ylabel("$a^*_{j+1}$")
axs[0].legend(j_labels+['45 degree line'])
axs[0].set_title(r"Optimal saving policy, low $\gamma$")

axs[1].plot(hh.a_grid, hh.a_grid[σ_reshaped[[0, 5, 20, 45, 49], :, 1].T])
axs[1].plot(hh.a_grid, hh.a_grid, '--')
axs[1].set_xlabel("$a_{j}$")
axs[1].set_ylabel("$a^*_{j+1}$")
axs[1].legend(j_labels+['45 degree line'])
axs[1].set_title(r"Optimal saving policy, high $\gamma$")

plt.show()
```



From an implied stationary population distribution, we can compute the aggregate labor supply L and private savings A.

```
@jax.jit
def compute_aggregates(μ, household):

    j_grid, a_grid, y_grid, Π, β, init_μ, VJ = household

    J, a_size, y_size = j_grid.size, a_grid.size, y_grid.size

    μ = μ.reshape((J, hh.a_grid.size, hh.y_grid.size))

    # Compute private savings
    a = a_grid.reshape((1, a_size, 1))
    A = (a * μ).sum() / J

    y = y_grid.reshape((1, 1, y_size))
    lj = l(j_grid).reshape((J, 1, 1))
    L = (lj * y * μ).sum() / J

    return A, L
```

```
A, L = compute_aggregates(μ, hh)
A, L
```

```
(Array(1.8594263, dtype=float32), Array(1.0781993, dtype=float32))
```

The capital stock in this economy equals $A - D$.

```
D = 0
K = A - D
```

The firm's optimality conditions imply interest rate r and wage rate w.

```
KL_to_r(K, L, firm), KL_to_w(K, L, firm)
```

```
(Array(0.20485441, dtype=float32), Array(0.8243317, dtype=float32))
```

The implied prices (r, w) differ from our guesses, so we must update our guesses and iterate until we find a fixed point.

This is our outer loop.

```
@jax.jit
def find_ss(household, firm, pol_target, Q, tol=1e-6, verbose=False):

    j_grid, a_grid, y_grid, Π, β, init_μ, VJ = household
    J = j_grid.size
    num_state = a_grid.size * y_grid.size

    D, G, δ = pol_target

    # Initial guesses of prices
    r, w = 0.05, 1.

    # Initial guess of τ
    τ = 0.15

    def cond_fn(state):
        "The convergence criteria."

        V, σ, μ, K, L, r, w, τ, D, G, δ, r_old, w_old = state

        error = (r - r_old) ** 2 + (w - w_old) ** 2

        return error > tol

    def body_fn(state):
        "The main body of iteration."

        V, σ, μ, K, L, r, w, τ, D, G, δ, r_old, w_old = state
        r_old, w_old, τ_old = r, w, τ

        # Household optimal decisions and values
        V, σ = backwards_opt([r, w], [τ, δ], hh, Q)

        # Compute the stationary distribution
        μ = popu_dist(σ, hh, Q)

        # Compute aggregates
        A, L = compute_aggregates(μ, hh)
        K = A - D
```

(continues on next page)

(continued from previous page)

```
        # Update prices
        r, w = KL_to_r(K, L, firm), KL_to_w(K, L, firm)

        # Find τ
        D_next = D
        τ = find_τ([D, D_next, G, δ],
                   [r, w],
                   [K, L])

        r = (r + r_old) / 2
        w = (w + w_old) / 2

        return V, σ, μ, K, L, r, w, τ, D, G, δ, r_old, w_old

    # Initial state
    V = jnp.empty((J, num_state), dtype=float)
    σ = jnp.empty((J, num_state), dtype=int)
    μ = jnp.empty((J, num_state), dtype=float)

    K, L = 1., 1.
    initial_state = (V, σ, μ, K, L, r, w, τ, D, G, δ, r-1, w-1)
    V, σ, μ, K, L, r, w, τ, D, G, δ, _, _ = jax.lax.while_loop(
                                    cond_fn, body_fn, initial_state)

    return V, σ, μ, K, L, r, w, τ, D, G, δ
```

```
ss1 = find_ss(hh, firm, [0, 0.1, np.zeros(hh.j_grid.size)], Q, verbose=True)
```

Let's time the computation

```
%time find_ss(hh, firm, [0, 0.1, np.zeros(hh.j_grid.size)], Q)[0].block_until_ready();
```

```
CPU times: user 842 ms, sys: 26.8 ms, total: 869 ms
Wall time: 535 ms
```

```
hh_out_ss1 = ss1[:3]
quant_ss1 = ss1[3:5]
price_ss1 = ss1[5:7]
policy_ss1 = ss1[7:11]
```

```
# V, σ, μ
V_ss1, σ_ss1, μ_ss1 = hh_out_ss1
```

```
# K, L
K_ss1, L_ss1 = quant_ss1

K_ss1, L_ss1
```

```
(Array(6.6221957, dtype=float32), Array(1.0781994, dtype=float32))
```

```
# r, w
r_ss1, w_ss1 = price_ss1

r_ss1, w_ss1
```

```
(Array(0.08430456, dtype=float32), Array(1.2056923, dtype=float32))
```

```
# τ, D, G, δ
τ_ss1, D_ss1, G_ss1, δ_ss1 = policy_ss1

τ_ss1, D_ss1, G_ss1, δ_ss1
```

```
(Array(0.05380344, dtype=float32),
 Array(0, dtype=int32, weak_type=True),
 Array(0.1, dtype=float32, weak_type=True),
 Array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],      ␣
↪dtype=float32))
```

115.12 Transition dynamics

We compute transition dynamics using a function `path_iteration`.

In an outer loop, we iterate over guesses of prices and taxes.

In an inner loop, we compute the optimal consumption and saving choices by each cohort j in each time t, then find the implied evolution of the joint distribution of assets and productivities.

We then update our guesses of prices and taxes given the aggregate labor supply and capital stock in the economy.

We use `solve_backwards` to solve for optimal saving choices given price and tax sequences and `simulate_forward` to compute the evolution of the joint distributions.

We require two steady states as inputs: the initial steady state to provide the initial condition for `simulate_forward`, and the final steady state to provide continuation values for `solve_backwards`.

```
@jax.jit
def bellman_operator(prices, taxes, V_next, household, Q):

    r, w = prices
    τ, δ = taxes

    j_grid, a_grid, y_grid, Π, β, init_μ, VJ = household
    J = j_grid.size

    num_state = a_grid.size * y_grid.size
    num_action = a_grid.size

    def bellman_operator_j(j):
        Rj = populate_R(j, r, w, τ, δ, household)
        vals = Rj + β * Q.dot(V_next[j+1])
        σ_j = jnp.argmax(vals, axis=1)
        V_j = vals[jnp.arange(num_state), σ_j]

        return V_j, σ_j

    V, σ = jax.vmap(bellman_operator_j, (0,))(jnp.arange(J-1))

    # The last life stage
```

(continues on next page)

(continued from previous page)

```
    j = J-1
    Rj = populate_R(j, r, w, τ, δ, household)
    vals = Rj + β * Q.dot(VJ)
    σ = jnp.concatenate([σ, jnp.argmax(vals, axis=1)[jnp.newaxis]])
    V = jnp.concatenate([V, vals[jnp.arange(num_state), σ[j]][jnp.newaxis]])

    return V, σ
```

```
@jax.jit
def solve_backwards(V_ss2, σ_ss2, household, firm, price_seq, pol_seq, Q):

    j_grid, a_grid, y_grid, Π, β, init_μ, VJ = household
    J = j_grid.size
    num_state = a_grid.size * y_grid.size

    τ_seq, D_seq, G_seq, δ_seq = pol_seq
    r_seq, w_seq = price_seq

    T = r_seq.size

    def solve_backwards_t(V_next, t):

        prices = (r_seq[t], w_seq[t])
        taxes = (τ_seq[t], δ_seq[t])
        V, σ = bellman_operator(prices, taxes, V_next, household, Q)

        return V, (V,σ)

    ts = jnp.arange(T-2, -1, -1)
    init_V = V_ss2

    _, outputs = jax.lax.scan(solve_backwards_t, init_V, ts)
    V_seq, σ_seq = outputs
    V_seq = V_seq[::-1]
    σ_seq = σ_seq[::-1]

    V_seq = jnp.concatenate([V_seq, V_ss2[jnp.newaxis]])
    σ_seq = jnp.concatenate([σ_seq, σ_ss2[jnp.newaxis]])

    return V_seq, σ_seq
```

```
@jax.jit
def population_evolution(σt, μt, household, Q):

    j_grid, a_grid, y_grid, Π, β, init_μ, VJ = household

    J = hh.j_grid.size
    num_state = hh.a_grid.size * hh.y_grid.size

    def population_evolution_j(j):

        Qσ = Q[jnp.arange(num_state), σt[j]]
        μ_next = μt[j] @ Qσ

        return μ_next
```

(continues on next page)

(continued from previous page)

```
    μ_next = jax.vmap(population_evolution_j, (0,))(jnp.arange(J-1))
    μ_next = jnp.concatenate([init_μ[jnp.newaxis], μ_next])

    return μ_next
```

```
@jax.jit
def simulate_forwards(σ_seq, D_seq, μ_ss1, K_ss1, L_ss1, household, Q):

    j_grid, a_grid, y_grid, Π, β, init_μ, VJ = household

    J, num_state = μ_ss1.shape

    T = σ_seq.shape[0]

    def simulate_forwards_t(μ, t):

        μ_next = population_evolution(σ_seq[t], μ, household, Q)

        A, L = compute_aggregates(μ_next, household)
        K = A - D_seq[t+1]

        return μ_next, (μ_next, K, L)

    ts = jnp.arange(T-1)
    init_μ = μ_ss1

    _, outputs = jax.lax.scan(simulate_forwards_t, init_μ, ts)
    μ_seq, K_seq, L_seq = outputs

    μ_seq = jnp.concatenate([μ_ss1[jnp.newaxis], μ_seq])
    K_seq = jnp.concatenate([K_ss1[jnp.newaxis], K_seq])
    L_seq = jnp.concatenate([L_ss1[jnp.newaxis], L_seq])

    return μ_seq, K_seq, L_seq
```

The following algorithm describes the path iteration procedure:

Algorithm 115.12.1 (AK-Aiyagari transition path algorithm)

Inputs Given initial steady state ss_1, final steady state ss_2, time horizon T, and policy sequences (D, G, δ)

Output Compute equilibrium transition paths for value functions V, policy functions σ, distributions μ, and prices (r, w, τ)

1. Initialize from steady states:
   - $(V_1, \sigma_1, \mu_1) \leftarrow ss_1$ *(Initial steady state)*
   - $(V_2, \sigma_2, \mu_2) \leftarrow ss_2$ *(Final steady state)*
   - $(r, w, \tau) \leftarrow initialize_prices(T)$ *(Linear interpolation)*
   - $error \leftarrow \infty, i \leftarrow 0$
2. **While** $error > \varepsilon$ or $i \leq max_iter$:
   1. $i \leftarrow i + 1$
   2. $(r_{\text{old}}, w_{\text{old}}, \tau_{\text{old}}) \leftarrow (r, w, \tau)$

3. **Backward induction:** For $t \in [T, 1]$:
    - For $j \in [0, J-1]$ *(age groups)*:
        - $V[t, j] \leftarrow \max_{a'}\{u(c) + \beta\mathbb{E}[V[t+1, j+1]]\}$
        - $\sigma[t, j] \leftarrow \arg\max_{a'}\{u(c) + \beta\mathbb{E}[V[t+1, j+1]]\}$
4. **Forward simulation:** For $t \in [1, T]$:
    - $\mu[t] \leftarrow \Gamma(\sigma[t], \mu[t-1])$ *(Distribution evolution)*
    - $K[t] \leftarrow \int a\, d\mu[t] - D[t]$ *(Aggregate capital)*
    - $L[t] \leftarrow \int l(j)\gamma\, d\mu[t]$ *(Aggregate labor)*
    - $r[t] \leftarrow \alpha Z(K[t]/L[t])^{\alpha-1}$ *(Interest rate)*
    - $w[t] \leftarrow (1-\alpha)Z(K[t]/L[t])^{\alpha}$ *(Wage rate)*
    - $\tau[t] \leftarrow solve_budget(r[t], w[t], K[t], L[t], D[t], G[t])$
5. Compute convergence metric:
    - $error \leftarrow \|r - r_{\text{old}}\| + \|w - w_{\text{old}}\| + \|\tau - \tau_{\text{old}}\|$
6. Update prices with dampening:
    - $r \leftarrow \lambda r + (1-\lambda)r_{\text{old}}$
    - $w \leftarrow \lambda w + (1-\lambda)w_{\text{old}}$
    - $\tau \leftarrow \lambda\tau + (1-\lambda)\tau_{\text{old}}$

3. **Return** $(V, \sigma, \mu, r, w, \tau)$

```
def path_iteration(ss1, ss2, pol_target, household, firm, Q, tol=1e-4, verbose=False):

    # Starting point: initial steady state
    V_ss1, σ_ss1, μ_ss1 = ss1[:3]
    K_ss1, L_ss1 = ss1[3:5]
    r_ss1, w_ss1 = ss1[5:7]
    τ_ss1, D_ss1, G_ss1, δ_ss1 = ss1[7:11]

    # Ending point: converging new steady state
    V_ss2, σ_ss2, μ_ss2 = ss2[:3]
    K_ss2, L_ss2 = ss2[3:5]
    r_ss2, w_ss2 = ss2[5:7]
    τ_ss2, D_ss2, G_ss2, δ_ss2 = ss2[7:11]

    # The given policies: D, G, δ
    D_seq, G_seq, δ_seq = pol_target
    T = G_seq.shape[0]

    # Initial guesses of prices
    r_seq = jnp.linspace(0, 1, T) * (r_ss2 - r_ss1) + r_ss1
    w_seq = jnp.linspace(0, 1, T) * (w_ss2 - w_ss1) + w_ss1

    # Initial guess of policy
    τ_seq = jnp.linspace(0, 1, T) * (τ_ss2 - τ_ss1) + τ_ss1

    error = 1
    num_iter = 0
```

(continues on next page)

(continued from previous page)

```
    if verbose:
        fig, axs = plt.subplots(1, 3, figsize=(14, 3))
        axs[0].plot(jnp.arange(T), r_seq)
        axs[1].plot(jnp.arange(T), w_seq)
        axs[2].plot(jnp.arange(T), τ_seq, label=f'iter {num_iter}')

    while error > tol:
        # Repeat until finding the fixed point

        r_old, w_old, τ_old = r_seq, w_seq, τ_seq

        pol_seq = (τ_seq, D_seq, G_seq, δ_seq)
        price_seq = (r_seq, w_seq)

        # Solve optimal policies backwards
        V_seq, σ_seq = solve_backwards(
            V_ss2, σ_ss2, hh, firm, price_seq, pol_seq, Q)

        # Compute population evolution forwards
        μ_seq, K_seq, L_seq = simulate_forwards(
            σ_seq, D_seq, μ_ss1, K_ss1, L_ss1, household, Q)

        # Update prices by aggregate capital and labor supply
        r_seq = KL_to_r(K_seq, L_seq, firm)
        w_seq = KL_to_w(K_seq, L_seq, firm)

        # Find taxes that balance the government budget constraint
        τ_seq = find_τ([D_seq[:-1], D_seq[1:], G_seq, δ_seq],
                       [r_seq, w_seq],
                       [K_seq, L_seq])

        # Distance between new and old guesses
        error = jnp.sum((r_old - r_seq) ** 2) + \
                jnp.sum((w_old - w_seq) ** 2) + \
                jnp.sum((τ_old - τ_seq) ** 2)

        num_iter += 1
        if verbose:
            print(f"Iteration {num_iter:3d}: error = {error:.6e}")
            axs[0].plot(jnp.arange(T), r_seq)
            axs[1].plot(jnp.arange(T), w_seq)
            axs[2].plot(jnp.arange(T), τ_seq, label=f'iter {num_iter}')

        r_seq = (r_seq + r_old) / 2
        w_seq = (w_seq + w_old) / 2
        τ_seq = (τ_seq + τ_old) / 2

    if verbose:
        axs[0].set_xlabel('t')
        axs[1].set_xlabel('t')
        axs[2].set_xlabel('t')

        axs[0].set_title('r')
        axs[1].set_title('w')
        axs[2].set_title('τ')
```

(continues on next page)

(continued from previous page)

```
        axs[2].legend(loc='center left', bbox_to_anchor=(1, 0.5))

    return V_seq, σ_seq, μ_seq, K_seq, L_seq, r_seq, w_seq, \
           τ_seq, D_seq, G_seq, δ_seq
```

We can now compute equilibrium transitions that are ignited by fiscal policy reforms.

115.13 Experiment 1: Immediate tax cut

Assume that the government cuts the tax rate and immediately balances its budget by issuing debt.

At $t = 0$, the government unexpectedly announces an immediate tax cut.

From $t = 0$ to 19, the government issues debt, so debt D_{t+1} increases linearly for 20 periods.

The government sets a target for its new debt level $D_{20} = D_0 + 1 = \bar{D} + 1$.

Government spending $\bar{G}$ and transfers $\bar{\delta}_j$ remain constant.

The government adjusts τ_t to balance the budget along the transition.

We want to compute the equilibrium transition path.

Our first step is to prepare appropriate policy variable arrays `D_seq`, `G_seq`, `δ_seq`

We'll compute a `τ_seq` that balances government budgets.

```
T = 150

D_seq = jnp.ones(T+1) * D_ss1
D_seq = D_seq.at[:21].set(D_ss1 + jnp.linspace(0, 1, 21))
D_seq = D_seq.at[21:].set(D_seq[20])

G_seq = jnp.ones(T) * G_ss1

δ_seq = jnp.repeat(δ_ss1, T).reshape((T, δ_ss1.size))
```

In order to iterate the path, we need to first find its destination, which is the new steady state under the new fiscal policy.

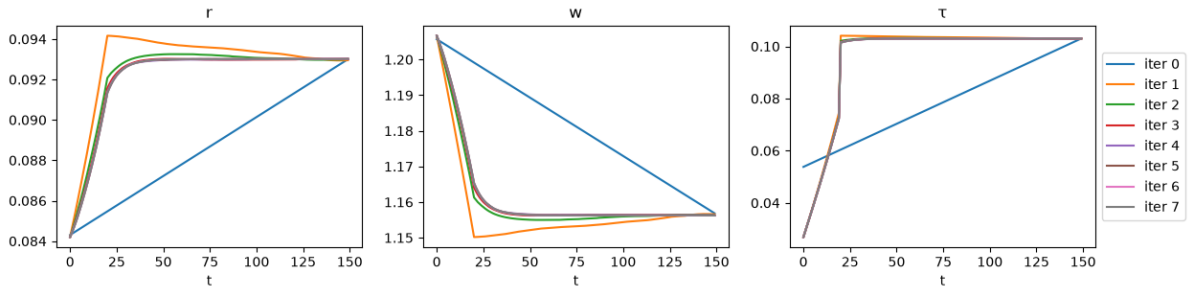
```
ss2 = find_ss(hh, firm, [D_seq[-1], G_seq[-1], δ_seq[-1]], Q)
```

We can use `path_iteration` to find equilibrium transition dynamics.

Setting the key argument `verbose=True` tells the function `path_iteration` to display convergence information.

```
paths = path_iteration(ss1, ss2, [D_seq, G_seq, δ_seq], hh, firm, Q, verbose=True)
```

```
Iteration    1: error = 2.075169e-01
Iteration    2: error = 3.733911e-02
Iteration    3: error = 7.672485e-03
Iteration    4: error = 1.819943e-03
Iteration    5: error = 4.734877e-04
Iteration    6: error = 1.216737e-04
Iteration    7: error = 2.955517e-05
```



Having successfully computed transition dynamics, let's study them.

```
V_seq, σ_seq, μ_seq = paths[:3]
K_seq, L_seq = paths[3:5]
r_seq, w_seq = paths[5:7]
τ_seq, D_seq, G_seq, δ_seq = paths[7:11]
```

```
ap = hh.a_grid[σ_seq[0]]
```

```
j = jnp.reshape(hh.j_grid, (hh.j_grid.size, 1, 1))
lj = l(j)
a = jnp.reshape(hh.a_grid, (1, hh.a_grid.size, 1))
y = jnp.reshape(hh.y_grid, (1, 1, hh.y_grid.size))
```

```
t = 0

ap = hh.a_grid[σ_seq[t]]
δ = δ_seq[t].reshape((hh.j_grid.size, 1, 1))

inc = (1 + r_seq[t]*(1-τ_seq[t])) * a \
        + (1-τ_seq[t]) * w_seq[t] * lj * y - δ
inc = inc.reshape((hh.j_grid.size, hh.a_grid.size * hh.y_grid.size))

c = inc - ap

c_mean0 = (c * μ_seq[t]).sum(axis=1)
```

We care about how the policy change affects consumption across cohorts and across time.

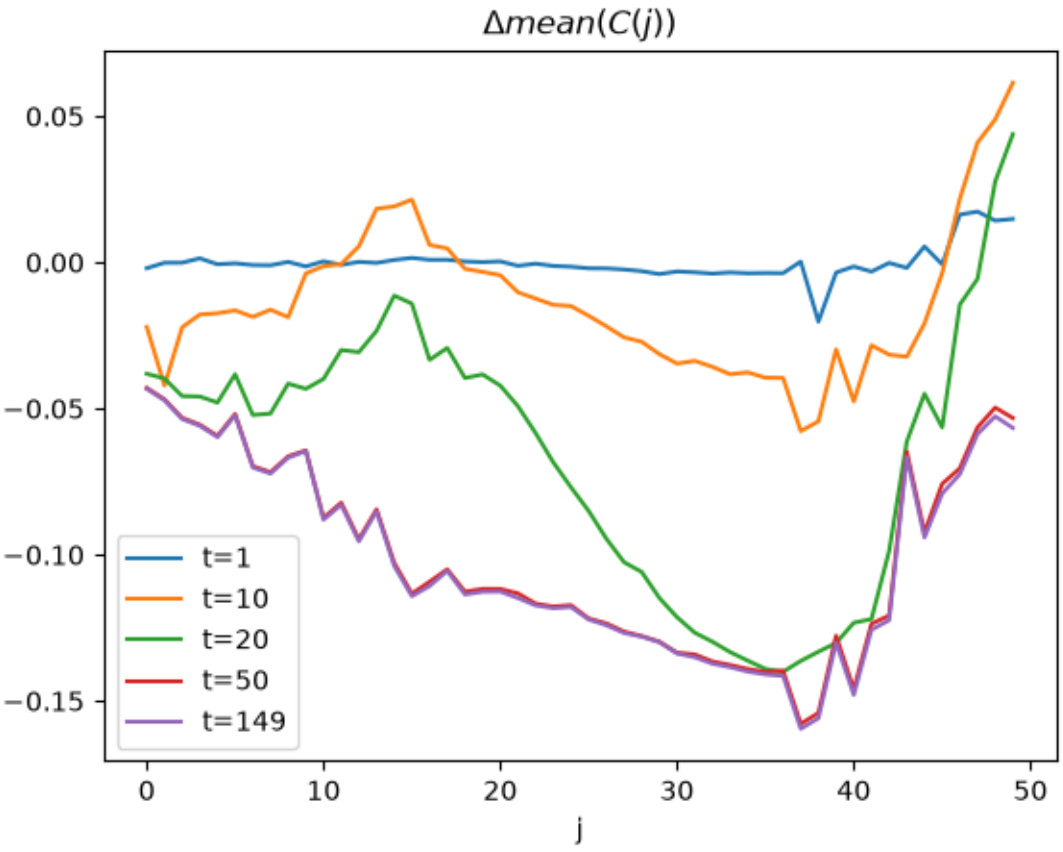
We can study age-specific average consumption levels.

```
for t in [1, 10, 20, 50, 149]:

    ap = hh.a_grid[σ_seq[t]]
    δ = δ_seq[t].reshape((hh.j_grid.size, 1, 1))

    inc = (1 + r_seq[t]*(1-τ_seq[t])) * a + (1-τ_seq[t]) * w_seq[t] * lj * y - δ
    inc = inc.reshape((hh.j_grid.size, hh.a_grid.size * hh.y_grid.size))

    c = inc - ap

    c_mean = (c * μ_seq[t]).sum(axis=1)

    plt.plot(range(hh.j_grid.size), c_mean-c_mean0, label=f't={t}')

plt.legend()
```

(continues on next page)

(continued from previous page)

```
plt.xlabel(r'j')
plt.title(r'$\Delta mean(C(j))$')
plt.show()
```



To summarize the transition, we can plot paths as we did in *Transitions in an Overlapping Generations Model*.

But unlike the setup in that two-period lived overlapping generations model, we no longer have representative old and young agents.

- now we have 50 cohorts of different ages at each time

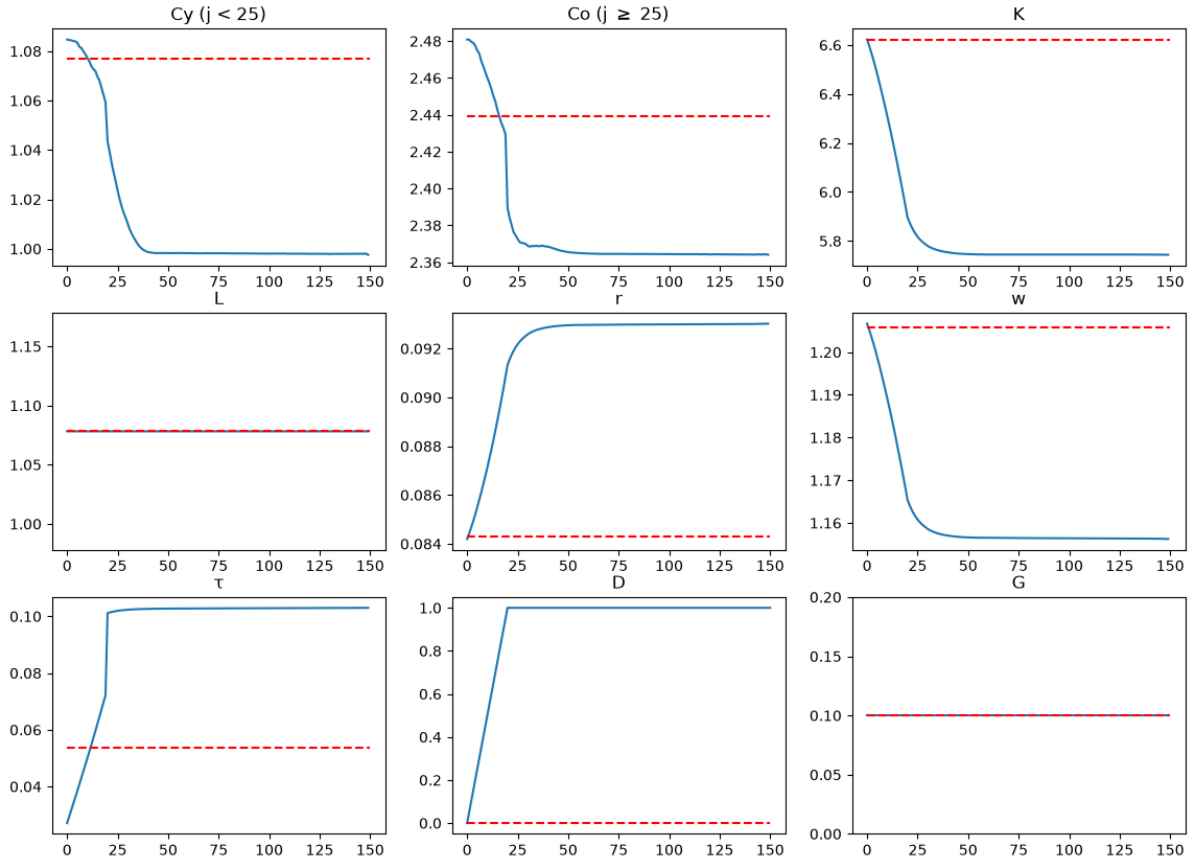
To proceed, we construct two age groups of equal size – young and old.

- at age 25, someone moves from being young to becoming old

```
ap = hh.a_grid[σ_ss1]
J = hh.j_grid.size
δ = δ_ss1.reshape((hh.j_grid.size, 1, 1))

inc = (1 + r_ss1*(1-τ_ss1)) * a + (1-τ_ss1) * w_ss1 * lj * y - δ
inc = inc.reshape((hh.j_grid.size, hh.a_grid.size * hh.y_grid.size))

c = inc - ap

Cy_ss1 = (c[:J//2] * μ_ss1[:J//2]).sum() / (J // 2)
Co_ss1 = (c[J//2:] * μ_ss1[J//2:]).sum() / (J // 2)
```

```
T = σ_seq.shape[0]
J = σ_seq.shape[1]

Cy_seq = np.empty(T)
Co_seq = np.empty(T)

for t in range(T):
    ap = hh.a_grid[σ_seq[t]]
    δ = δ_seq[t].reshape((hh.j_grid.size, 1, 1))

    inc = (1 + r_seq[t]*(1-τ_seq[t])) * a + (1-τ_seq[t]) * w_seq[t] * lj * y - δ
    inc = inc.reshape((hh.j_grid.size, hh.a_grid.size * hh.y_grid.size))

    c = inc - ap

    Cy_seq[t] = (c[:J//2] * μ_seq[t, :J//2]).sum() / (J // 2)
    Co_seq[t] = (c[J//2:] * μ_seq[t, J//2:]).sum() / (J // 2)
```

```
fig, axs = plt.subplots(3, 3, figsize=(14, 10))

# Cy (j=0-24)
axs[0, 0].plot(Cy_seq)
axs[0, 0].hlines(Cy_ss1, 0, T, color='r', linestyle='--')
axs[0, 0].set_title('Cy (j < 25)')

# Cy (j=25-49)
axs[0, 1].plot(Co_seq)
axs[0, 1].hlines(Co_ss1, 0, T, color='r', linestyle='--')
axs[0, 1].set_title(r'Co (j $\geq$ 25)')

names = ['K', 'L', 'r', 'w', 'τ', 'D', 'G']
for i in range(len(names)):
    i_var = i + 3
    i_axes = i + 2

    row_i = i_axes // 3
    col_i = i_axes % 3

    axs[row_i, col_i].plot(paths[i_var])
    axs[row_i, col_i].hlines(ss1[i_var], 0, T, color='r', linestyle='--')
    axs[row_i, col_i].set_title(names[i])

# ylims
axs[1, 0].set_ylim([ss1[4]-0.1, ss1[4]+0.1])
axs[2, 2].set_ylim([ss1[9]-0.1, ss1[9]+0.1])

plt.show()
```



Now let's compute the mean and variance of consumption conditional on age at each time t.

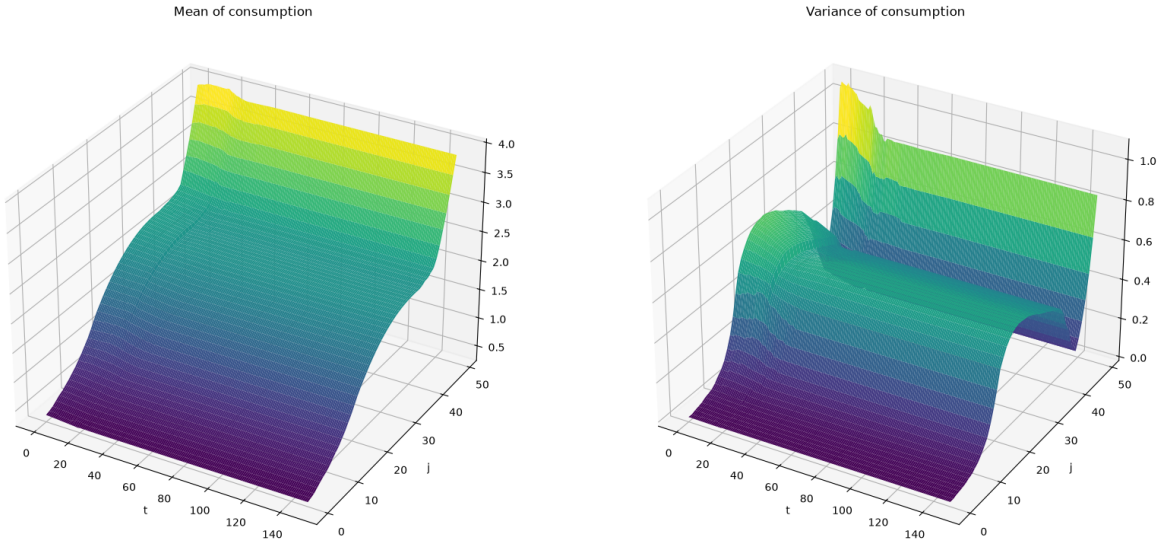
```
Cmean_seq = np.empty((T, J))
Cvar_seq = np.empty((T, J))

for t in range(T):
    ap = hh.a_grid[σ_seq[t]]
    δ = δ_seq[t].reshape((hh.j_grid.size, 1, 1))

    inc = (1 + r_seq[t]*(1-τ_seq[t])) * a + (1-τ_seq[t]) * w_seq[t] * lj * y - δ
    inc = inc.reshape((hh.j_grid.size, hh.a_grid.size * hh.y_grid.size))

    c = inc - ap

    Cmean_seq[t] = (c * μ_seq[t]).sum(axis=1)
    Cvar_seq[t] = ((c - Cmean_seq[t].reshape((J, 1))) ** 2 * μ_seq[t]).sum(axis=1)
```

```
J_seq, T_range = np.meshgrid(np.arange(J), np.arange(T))

fig = plt.figure(figsize=[20, 20])

# Plot the consumption mean over age and time
ax1 = fig.add_subplot(121, projection='3d')
ax1.plot_surface(T_range, J_seq, Cmean_seq, rstride=1, cstride=1,
                 cmap='viridis', edgecolor='none')
ax1.set_title(r"Mean of consumption")
ax1.set_xlabel(r"t")
```

(continues on next page)

(continued from previous page)

```
ax1.set_ylabel(r"j")

# plot the consumption variance over age and time
ax2 = fig.add_subplot(122, projection='3d')
ax2.plot_surface(T_range, J_seq, Cvar_seq, rstride=1, cstride=1,
                 cmap='viridis', edgecolor='none')
ax2.set_title(r"Variance of consumption")
ax2.set_xlabel(r"t")
ax2.set_ylabel(r"j")

plt.show()
```



115.14 Experiment 2: Preannounced tax cut

Now the government announces a permanent tax rate cut at time 0 but implements it only after 20 periods.

We will use the same key toolkit `path_iteration`.

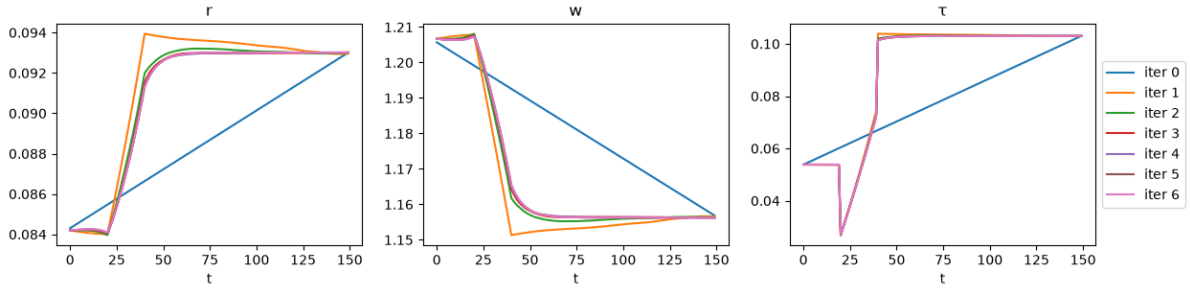
We must specify `D_seq` appropriately.

```
T = 150

D_t = 20
D_seq = jnp.ones(T+1) * D_ss1
D_seq = D_seq.at[D_t:D_t+21].set(D_ss1 + jnp.linspace(0, 1, 21))
D_seq = D_seq.at[D_t+21:].set(D_seq[D_t+20])

G_seq = jnp.ones(T) * G_ss1

δ_seq = jnp.repeat(δ_ss1, T).reshape((T, δ_ss1.size))
```

```
ss2 = find_ss(hh, firm, [D_seq[-1], G_seq[-1], δ_seq[-1]], Q)
```

```
paths = path_iteration(ss1, ss2, [D_seq, G_seq, δ_seq],
                       hh, firm, Q, verbose=True)
```

```
Iteration    1: error = 1.300627e-01
Iteration    2: error = 2.349870e-02
Iteration    3: error = 4.931191e-03
Iteration    4: error = 1.196040e-03
Iteration    5: error = 3.122933e-04
Iteration    6: error = 7.865898e-05
```




```
V_seq, σ_seq, μ_seq = paths[:3]
K_seq, L_seq = paths[3:5]
r_seq, w_seq = paths[5:7]
τ_seq, D_seq, G_seq, δ_seq = paths[7:11]
```

```
T = σ_seq.shape[0]
J = σ_seq.shape[1]

Cy_seq = np.empty(T)
Co_seq = np.empty(T)

for t in range(T):
    ap = hh.a_grid[σ_seq[t]]
    δ = δ_seq[t].reshape((hh.j_grid.size, 1, 1))

    inc = (1 + r_seq[t]*(1-τ_seq[t])) * a + (1-τ_seq[t]) * w_seq[t] * lj * y - δ
    inc = inc.reshape((hh.j_grid.size, hh.a_grid.size * hh.y_grid.size))

    c = inc - ap

    Cy_seq[t] = (c[:J//2] * μ_seq[t, :J//2]).sum() / (J // 2)
    Co_seq[t] = (c[J//2:] * μ_seq[t, J//2:]).sum() / (J // 2)
```

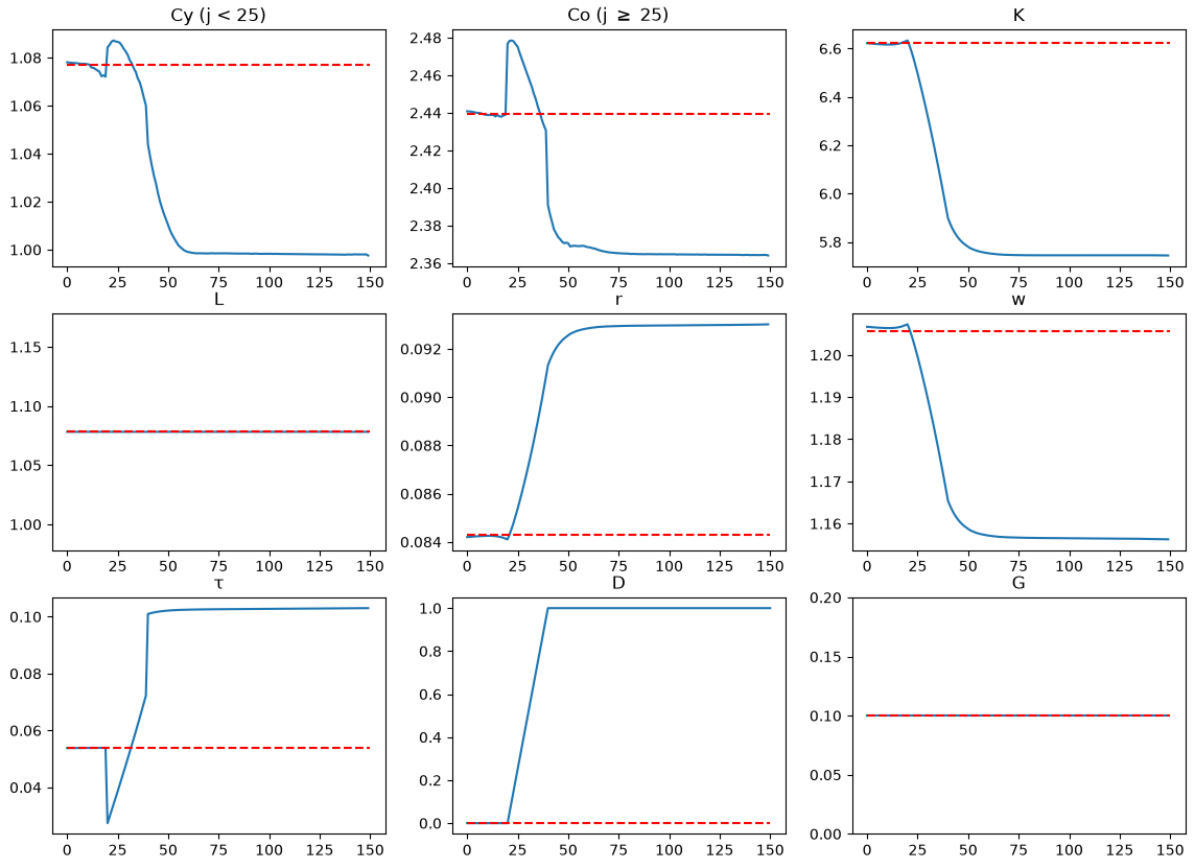
Below we plot the transition paths of the economy.

```
fig, axs = plt.subplots(3, 3, figsize=(14, 10))

# Cy (j=0-24)
axs[0, 0].plot(Cy_seq)
axs[0, 0].hlines(Cy_ss1, 0, T, color='r', linestyle='--')
axs[0, 0].set_title('Cy (j < 25)')

# Cy (j=25-49)
axs[0, 1].plot(Co_seq)
axs[0, 1].hlines(Co_ss1, 0, T, color='r', linestyle='--')
```

(continues on next page)

(continued from previous page)

```
axs[0, 1].set_title(r'Co (j $\geq$ 25)')

names = ['K', 'L', 'r', 'w', 'τ', 'D', 'G']
for i in range(len(names)):
    i_var = i + 3
    i_axes = i + 2

    row_i = i_axes // 3
    col_i = i_axes % 3

    axs[row_i, col_i].plot(paths[i_var])
    axs[row_i, col_i].hlines(ss1[i_var], 0, T, color='r', linestyle='--')
    axs[row_i, col_i].set_title(names[i])

# ylims
axs[1, 0].set_ylim([ss1[4]-0.1, ss1[4]+0.1])
axs[2, 2].set_ylim([ss1[9]-0.1, ss1[9]+0.1])

plt.show()
```



Notice how prices and quantities respond immediately to the anticipated tax rate increase.

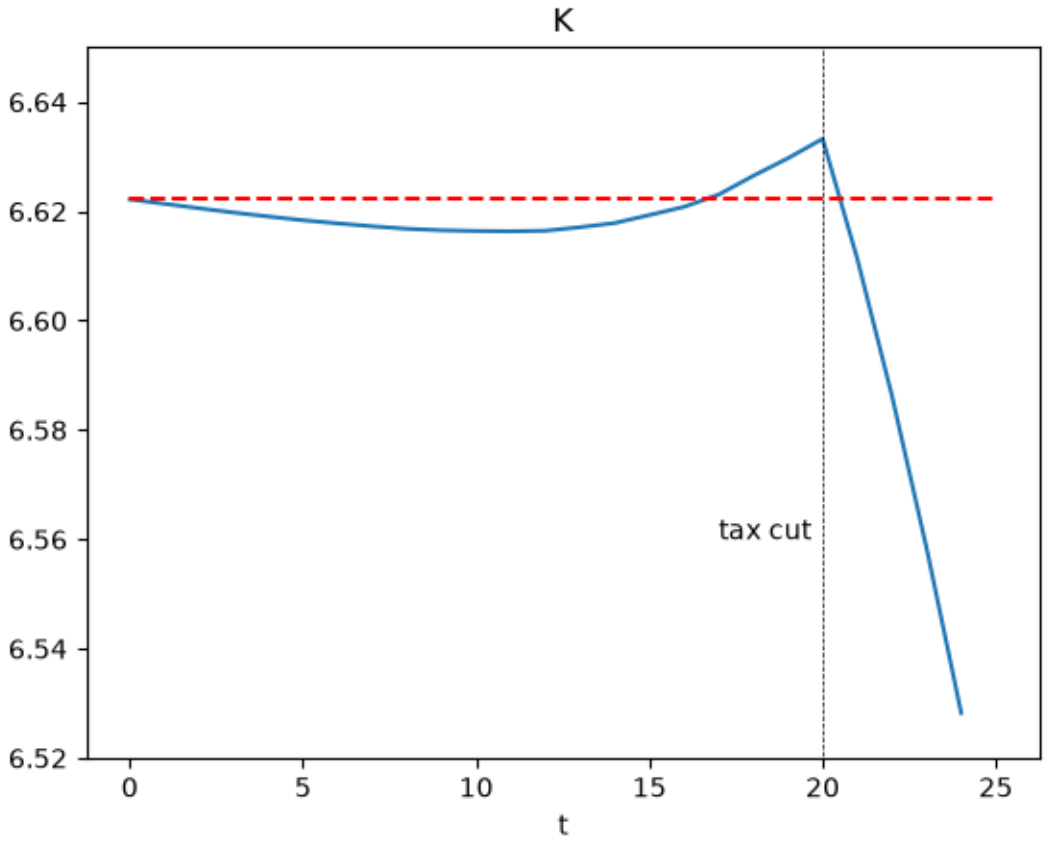
Let's zoom in on how the capital stock responds.

```
# K
i_var = 3
```

(continues on next page)

(continued from previous page)

```
plt.plot(paths[i_var][:25])
plt.hlines(ss1[i_var], 0, 25, color='r', linestyle='--')
plt.vlines(20, 6, 7, color='k', linestyle='--', linewidth=0.5)
plt.text(17, 6.56, r'tax cut')
plt.ylim([6.52, 6.65])
plt.title("K")
plt.xlabel("t")
plt.show()
```



After the tax cut policy is implemented at $t = 20$, the aggregate capital will decrease because of the crowding out effect.

Having foreseen an increase in the interest rate, individuals a few periods before $t = 20$ start saving more.

Because that increases the capital, a temporary decrease in the interest rate ensues.

For agents living in much earlier periods, that lower interest rate causes them to save less.

We can also plot evolutions of means and variances of consumption by different cohorts along a transition path.

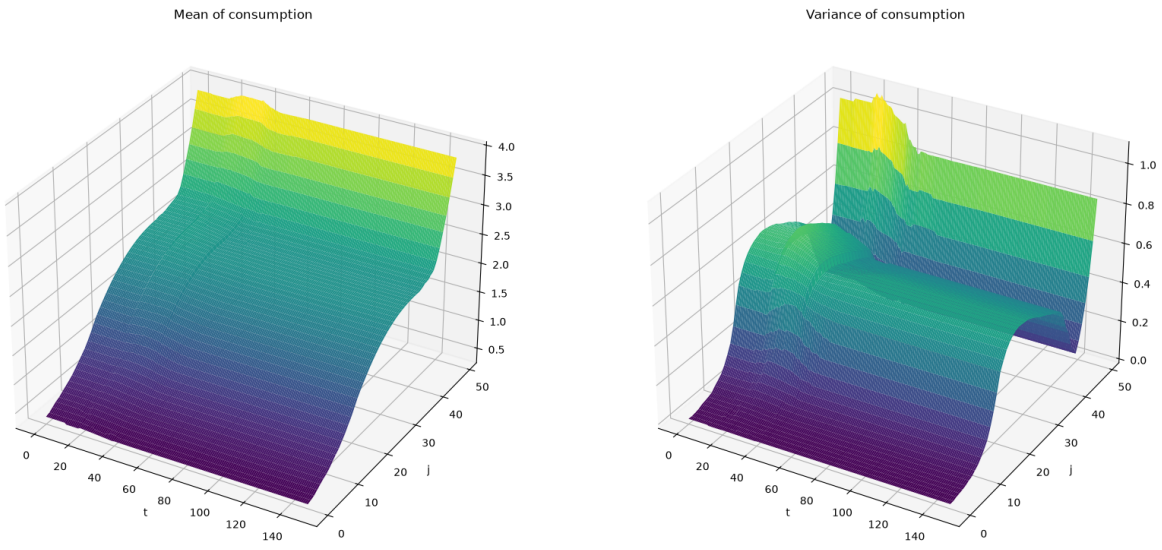
```
Cmean_seq = np.empty((T, J))
Cvar_seq = np.empty((T, J))

for t in range(T):
    ap = hh.a_grid[σ_seq[t]]
    δ = δ_seq[t].reshape((hh.j_grid.size, 1, 1))

    inc = (1 + r_seq[t]*(1-τ_seq[t])) * a + (1-τ_seq[t]) * w_seq[t] * lj * y - δ
    inc = inc.reshape((hh.j_grid.size, hh.a_grid.size * hh.y_grid.size))
```

(continues on next page)

(continued from previous page)

```
    c = inc - ap

    Cmean_seq[t] = (c * μ_seq[t]).sum(axis=1)
    Cvar_seq[t] = (
        (c - Cmean_seq[t].reshape((J, 1))) ** 2 * μ_seq[t]).sum(axis=1)
```

```
J_seq, T_range = np.meshgrid(np.arange(J), np.arange(T))

fig = plt.figure(figsize=[20, 20])

# Plot the consumption mean over age and time
ax1 = fig.add_subplot(121, projection='3d')
ax1.plot_surface(T_range, J_seq, Cmean_seq, rstride=1, cstride=1,
                 cmap='viridis', edgecolor='none')
ax1.set_title(r"Mean of consumption")
ax1.set_xlabel(r"t")
ax1.set_ylabel(r"j")

# Plot the consumption variance over age and time
ax2 = fig.add_subplot(122, projection='3d')
ax2.plot_surface(T_range, J_seq, Cvar_seq, rstride=1, cstride=1,
                 cmap='viridis', edgecolor='none')
ax2.set_title(r"Variance of consumption")
ax2.set_xlabel(r"t")
ax2.set_ylabel(r"j")

plt.show()
```



CHAPTER

SIX

TWO COMPUTATIONS TO FUND SOCIAL SECURITY

Contents

- *Two Computations to Fund Social Security*
  - *Overview*
  - *Model*
  - *Two experiments*
  - *Computation strategy*
  - *Calibration*
  - *Individual optimality*
  - *Transition path computation*
  - *Experiment 1: compensation through debt*
  - *Experiment 2: government capital accumulation*
  - *Distribution surfaces*

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

In addition to what's in Anaconda, this lecture will need the following library

```
!pip install jax
```

116.1 Overview

This lecture describes two computational experiments about alternative ways to move gradually from an unfunded (pay-as-you-go) to a fully funded social security system, following Huang *et al.* [1997].

As populations age, pay-as-you-go social security systems have faced financial difficulties.

This situation has led some of today's policy makers and policy advisors to think that today's citizens would be better off if earlier policy makers had set up a fully-funded retirement system.

But starting from where we are today, a transition to a fully funded system creates distributional challenges because older generations who contributed to the unfunded system could lose benefits.

To study possibilities quantitatively, this lecture employs a general equilibrium overlapping generations model that modifies the Auerbach and Kotlikoff [1987] environment by incorporating

- risk-sensitive preferences
- uncertainty about lifetimes
- uninsurable labor income risk
- a theory of consumption distributions within and across cohorts

By employing the discounted risk-sensitive linear-quadratic preferences of Hansen and Sargent [1995], the model yields linear decision rules for individual consumption and savings.

This makes it computationally feasible to track the joint distribution of consumption and wealth across cohorts.

This lecture relates to two other lectures:

- *Transitions in an Overlapping Generations Model* studies how taxes, transfers, and debt affect capital accumulation in a two-period OLG model, introducing the Auerbach and Kotlikoff [1987] framework that this lecture extends.
- *A Long-Lived, Heterogeneous Agents, Overlapping Generations Model* studies how fiscal policy interacts with precautionary savings in a long-lived OLG model, using discrete dynamic programming rather than the linear-quadratic approach adopted here.

We use the following imports and configurations

```
import jax
import jax.numpy as jnp
from jax import jit, vmap
import jax.lax as lax
import numpy as np
import matplotlib.pyplot as plt
from collections import namedtuple

# Enable 64-bit precision
jax.config.update("jax_enable_x64", True)
```

116.2 Model

116.2.1 Environment

The economy consists of overlapping generations of finitely lived individuals who may live up to $T_0 + 1$ years and an infinitely lived government.

Individual consumers and the government can invest at a constant risk-free gross rate of return.

During the first $T_1 + 1$ periods of life, consumers receive labor income that they allocate among consumption, taxes, and asset accumulation.

During the final $T_0 - T_1$ periods of life (retirement), consumers receive social security benefits and dissave by drawing down their assets.

The government taxes income from capital and labor, issues debt, purchases goods, and pays retirement benefits.

For any variable z, we use subscript t to denote age, argument s in parentheses to denote calendar time, and superscript $s - t$ to denote date of birth, so that $z^{s-t}(s) \equiv z_t(s) \equiv z_t^{s-t}(s)$.

116.2.2 Demographics

At date s, a cohort of measure $N_0(s)$ consumers is born who live during periods $s, s + 1, \ldots, s + T_0$.

As a cohort ages, its members face random survival according to age-to-age survival probabilities $\{\alpha_t\}_{t=0}^{T_0}$, where α_t is the probability of surviving from age t to $t + 1$.

Let $N_t(s)$ be the number of age-t people alive at time s and let n be the constant gross rate of population growth.

The size of age group t at time s satisfies

$$N_t(s) = \lambda_t \cdot N_0(s - t)$$

where $\lambda_t = \prod_{j=0}^{t-1} \alpha_j$ for $t = 0, \ldots, T_0$, with $\lambda_0 = 1$, and births follow $N_0(s) = n^s \cdot N_0(0)$.

The population fraction of cohort t at each time s is

$$f_t = \frac{\lambda_t \cdot n^{-t}}{\sum_{\tau=0}^{T_0} \lambda_\tau \cdot n^{-\tau}}$$

and total population at time s is

$$N(s) = N_0(0) \cdot n^s \cdot \sum_{t=0}^{T_0} n^{-t} \cdot \lambda_t.$$

116.2.3 Distributions and aggregates

Individuals face life span uncertainty and labor income shocks.

They self-insure by accumulating risk-free assets (government bonds and physical capital), while properly taking into account the social security benefits that they anticipate receiving.

Let ϵ_0^t denote the history of random shocks that an individual has received from birth to age t.

The state vector $x_t(s) = x_t(s; \epsilon_0^t, x_0)$ measures the stock of assets as well as information variables a consumer uses to forecast future preferences or opportunities.

The model delivers consumption as a time- and age-dependent linear function of the state vector

$$c_t(s;\epsilon_0^t,x_0) = \eta_{ct}(s)\cdot x_t(s;\epsilon_0^t,x_0)$$

where the state vector follows a linear law of motion

$$x_{t+1}(s+1;\epsilon_0^{t+1},x_0) = A_t(s)\cdot x_t(s;\epsilon_0^t,x_0) + C_t(s)\cdot\epsilon_{t+1}$$

with ϵ_{t+1} a martingale difference sequence satisfying $E(\epsilon_{t+1}|J_t)=0$, $E(\epsilon_{t+1}\cdot\epsilon_{t+1}'|J_t)=I$, and $J_t=(\epsilon_0^t,x_0)$.

The model delivers probability distributions for state vectors.

Let $\mu_t(s)=E[x_t(s)]$ and $\Sigma_t(s)=E[(x_t(s)-\mu_t(s))(x_t(s)-\mu_t(s))']$.

These moments satisfy

$$\mu_{t+1}(s+1) = A_t(s)\cdot\mu_t(s)$$

$$\Sigma_{t+1}(s+1) = A_t(s)\cdot\Sigma_t(s)\cdot A_t(s)' + C_t(s)\cdot C_t(s)'$$

Per capita aggregate consumption is

$$c(s)/N(s) = \sum_{t=0}^{T_0}\mu_{ct}(s)\cdot f_t$$

where $\mu_{ct}(s)$ is the mean consumption of age-t people at time s.

The distribution of consumption within age cohort t has mean $\mu_{ct}(s)=\eta_{ct}(s)\cdot\mu_t(s)$ and variance $\Sigma_{ct}(s)=\eta_{ct}(s)\cdot\Sigma_t(s)\cdot\eta_{ct}(s)'$.

116.2.4 Resource constraint

The economy-wide physical resource constraint is

$$g(s)\cdot N(s) + \sum_{t=0}^{T_0} c_t(s)\cdot N_t^{s-t} + K(s) = R(s-1)\cdot K(s-1) + w(s)\cdot\sum_{t=0}^{T_1}\varepsilon_t\cdot N_t^{s-t} + N_0(s)\cdot k_{-1}(s)$$

where $g(s)$ is per capita government purchases, $K(s-1)$ is physical capital, $R(s-1)=1+r(s-1)-\delta$ is the gross return on assets, ε_t is the exogenous efficiency endowment of age-t people, $w(s)$ is the base wage rate, $N_0(s)\cdot k_{-1}(s)$ is capital brought by newborns, and δ is the depreciation rate.

116.2.5 Factor prices

We consider two alternative assumptions about factor prices:

- *Small open economy:* $r(s-1)=r$ and $w(s)=w$ are exogenous and constant.
- *Closed economy:* Factor prices are determined by marginal products from a Cobb-Douglas production function:

$$r(s-1) = \tilde{A}\cdot\tilde{\alpha}\cdot(K(s-1)/\tilde{N}(s))^{\tilde{\alpha}-1}, \qquad w(s) = (1-\tilde{\alpha})\cdot\tilde{A}\cdot(K(s-1)/\tilde{N}(s))^{\tilde{\alpha}}$$

where $\tilde{N}(s)=\sum_{t=0}^{T_1}\varepsilon_t\cdot N_t^{s-t}$ is aggregate labor input in efficiency units and $\tilde{\alpha}$ is capital's share of income.

116.2.6 Consumers' problems

Individual consumers face an overlapping-generations version of a classic consumption-saving problem (see *The Permanent Income Model*).

Working-age consumers ($t \leq T_1$) receive labor income $w(s) \cdot \varepsilon_t + d_t$, where d_t is an AR(1) process

$$d_t = \rho_d \cdot d_{t-1} + \xi_t$$

with ξ_t being Gaussian white noise with variance σ_d^2.

The budget constraint at age t and time s is

$$c_t(s) + a_t(s) = R(s-1) \cdot a_{t-1}(s-1) + w(s) \cdot \varepsilon_t + S_t(s) - T_t(s) + d_t \tag{116.1}$$

where $a_{t-1}(s-1)$ is asset holdings at the beginning of age t, $S_t(s)$ is social security benefits (zero while working, $S(s)$ when retired), and taxes are $T_t(s) = \tau_0(s) + \tau_\ell(s)(w(s) \cdot \varepsilon_t + d_t) + \tau_a(s)(R(s-1)-1) \cdot a_{t-1}(s-1)$.

Following Hansen and Sargent [1995], preferences over stochastic consumption processes are defined recursively by

$$U_t = -(\pi \cdot c_t - \gamma_t)^2/2 + \beta_t \cdot \mathcal{R}_t(U_{t+1})$$

where $\mathcal{R}_t(U_{t+1}) = (2/\sigma) \cdot \log E[\exp(\sigma \cdot U_{t+1}/2)|J_t]$, σ is the risk-sensitivity parameter, and $\beta_t = \bar{\beta} \cdot \alpha_t$ is the survival-adjusted discount factor.

This preference specification delivers linear decision rules while allowing a form of risk-sensitivity that induces a type of precautionary savings.

When $\sigma < 0$, the consumer prefers early resolution of uncertainty, and decision rules depend partly on noise statistics.

116.2.7 Government

The government purchases goods, pays social security benefits, taxes capital and labor income, confiscates accidental bequests, and issues one-period bonds.

The government budget constraint is

$$g(s) \cdot N(s) + \sum_{t=T_1+1}^{T_0} S_t(s) \cdot N_t^{s-t} + R(s-1) \cdot \sum_{t=1}^{T_0} b_{t-1}(s-1) \cdot N_t^{s-t} =$$

$$\sum_{t=0}^{T_0} N_t^{s-t}\{\tau_a(s)[R(s-1)-1] \cdot a_{t-1}(s-1) + \tau_\ell(s) \cdot w(s) \cdot \varepsilon_t\} + \tau_0 \cdot N(s) +$$

$$\sum_{t=0}^{T_0} b_t(s) \cdot N_t^{s-t} + R(s-1) \cdot \sum_{t=0}^{T_0} (1-\alpha_t) \cdot k_t(s-1) \cdot N_t^{s-t-1}$$

where $b_t(s)$ is government debt held by age-t individuals at time s and the last term represents the bequest tax.

A transition between fiscal regimes is described by dates $0 \leq s_1 < s_2 < s_3$:

- Before $s = 0$: initial stationary equilibrium
- At $s = 0$: government announces a policy change
- $s \in [s_1, s_2)$: fiscal parameters and the social security system change
- $s \geq s_2$: new constant policy parameters
- $s \geq s_3$: final stationary equilibrium

116.2.8 Equilibrium

We define the following objects:

- An **allocation** is a stochastic process for $\{c_t(s), a_t(s)\}_{s=0}^{s_3}$ for $t = 0, \ldots, T_0$, and a sequence $\{K(s)\}_{s=0}^{s_3}$.
- A **government policy** is a sequence $\{b(s), g(s), \tau_\ell(s), S(s), \tau_a(s)\}_{s=0}^{s_3}$.
- A **price system** is a sequence $\{w(s), r(s-1)\}_{s=0}^{s_3}$.

An **equilibrium** is an allocation, a price system, and a government policy such that

1. given the price sequence and government policy, the allocation solves households' optimum problems, and
2. the allocation and government policy satisfy the government budget constraint at each date s.

In a **stationary equilibrium**, all variables are independent of calendar time s, which simplifies the government budget constraint to

$$g + \sum_{t=T_1+1}^{T_0} S_t \cdot f_t + [R/n - 1] \cdot \bar{b} = \tau_a(R-1) \cdot \sum_{t=0}^{T_0} a_{t-1} \cdot f_t + \tau_\ell \cdot \sum_{t=0}^{T_1} w \cdot \varepsilon_t \cdot f_t + \frac{R}{n} \cdot \sum_{t=0}^{T_0} (1 - \alpha_t) \cdot a_t \cdot f_t$$

where $\bar{b}$ is per capita government debt in steady state.

116.2.9 Transition dynamics

The following diagram shows the age-time structure during the transition.

Each horizontal line represents the lifetime of one cohort, and the vertical dashed lines mark the policy change dates s_1 and s_2.

```
fig, ax = plt.subplots(figsize=(8, 5))

birth_range = np.arange(-60, 41, 5)

ax.hlines(birth_range, birth_range, birth_range + 60, 'k', linewidth=1)
ax.vlines([0, 40], -60, 40, 'k', linestyle='--', linewidth=0.5)

ax.set_ylabel("Date born")
ax.set_xlabel("Time")

ax.text(-20, 25, r"$s_1=0$")
ax.text(43, -50, r"$s_2$")

ax.invert_yaxis()
plt.show()
```

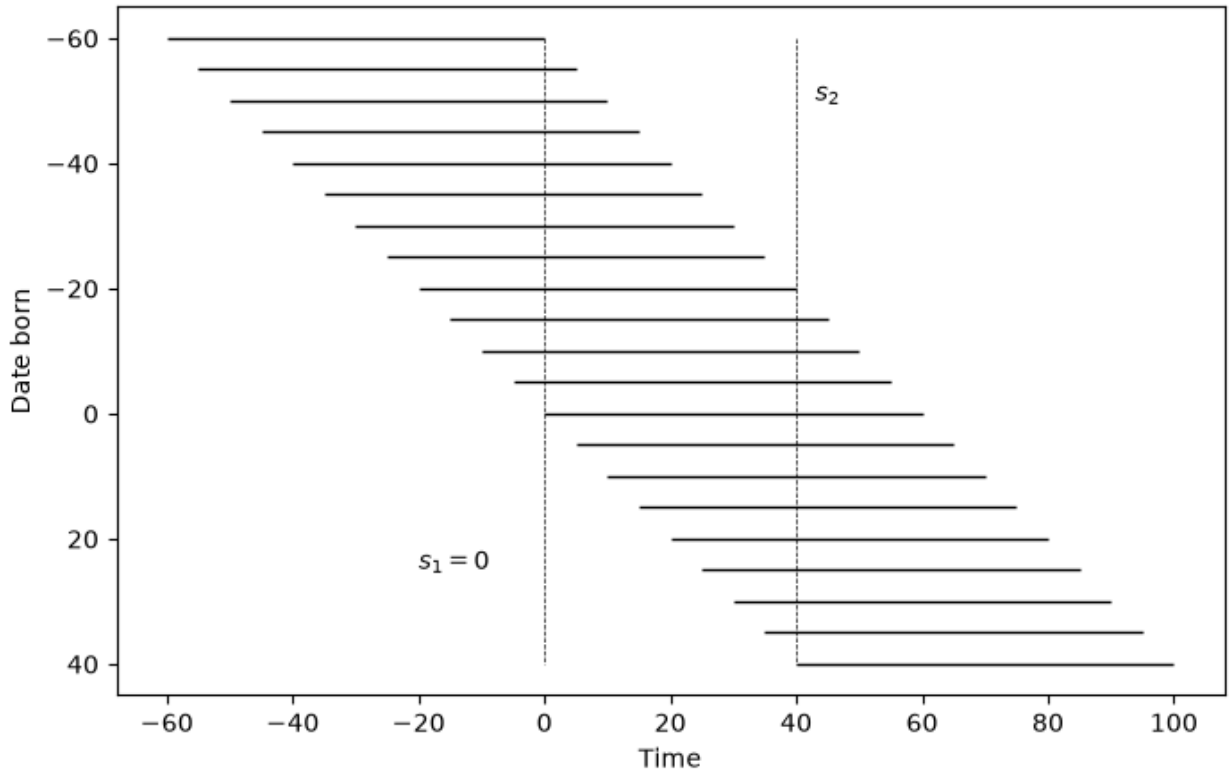


Fig. 116.1: Age-time diagram for overlapping generations

Small open economy

With fixed factor prices, the transition has a clear structure:

- Cohorts born before $s_1 - T_0$ die before the policy change and are unaffected.
- Cohorts alive at s_1 must recalculate their consumption-saving plans for their remaining lifetimes.
- Cohorts born between s_1 and s_2 face time-varying tax and benefit rates, while those born after s_2 face constant parameters.
- The transition ends at $s_3 = s_2 + T_0$ when the last cohort that experienced the policy change has died.

Because factor prices are fixed, we can compute the transition by solving decision rules for cohorts born at dates $s_1 - T_0 - 1, \ldots, s_2$.

For any date s, aggregate consumption is computed by summing across all living cohorts (along a vertical line in the age-time diagram), weighted by their population fractions.

Closed economy

With endogenous factor prices, the transition is more complex:

- Factor prices continue to evolve after policy parameters stabilize at s_2, so we follow Auerbach and Kotlikoff [1987] and truncate at $s_3 = s_2 + 2T_0$.
- The computation requires nested iteration: an inner loop determines labor income tax rates, and an outer loop adjusts interest rates to clear factor markets.
- Changes in saving behavior affect capital accumulation, which alters marginal products and feeds back into household decisions.
- Lower interest rates benefit young workers through higher wages but hurt retirees through lower returns on savings.

116.3 Two experiments

We explore two strategies for transitioning to a fully funded social security system.

In experiment 1, the government terminates social security benefits but compensates entitled generations through a one-time increase in government debt.

In experiment 2, the government retains social security benefits but temporarily raises taxes to accumulate physical capital, the returns from which eventually finance social security payments.

Both proposals finance a transition to fully funded social security while maintaining welfare across generations, but they entail different amounts of intergenerational risk-sharing.

We compute both experiments under fixed and endogenous factor prices and compare outcomes below.

116.4 Computation strategy

116.4.1 Dynamic program

An individual consumer's problem can be formulated as a discounted risk-sensitive linear control problem (see *LQ Control: Foundations*).

Let $x_t = [a_{t-1}, z_t]'$ where z_t is the vector of shocks.

The optimal value function takes the form $U_t = x_t' \cdot P_t \cdot x_t + \xi_t$.

The recursive problem is

$$U_t = \max_{u_t} \left\{ u_t' Q_t u_t + x_t' R_t x_t + \frac{2\beta_t}{\sigma} \log E_t[\exp(\sigma U_{t+1}/2)] \right\}$$

subject to $x_{t+1} = A_t x_t + B_t u_t + C_t w_{t+1}$.

We deploy two operators

$$\begin{aligned} T_t(P) &= P + \sigma P C_t \left(I - \sigma C_t' P C_t\right)^{-1} C_t' P \\ D_t(W) &= R_t + A_t' \left[\beta_t W - \beta_t^2 W B_t \left(Q_t + \beta_t B_t' W B_t\right)^{-1} B_t' W\right] A_t \\ \mathcal{S}_t(k, P) &= \beta_t k - (\beta_t/\sigma) \log\det\left(I - \sigma C_t' P C_t\right) \end{aligned}$$

that we use to construct a value function recursion $P_t = (D_t \circ T_t) P_{t+1}$, $\xi_t = \mathcal{S}_t(\xi_{t+1}, P_{t+1})$, and an optimal control

$$u_t = -F_t x_t, \qquad F_t = \beta_t \left[Q_t + \beta_t B_t' T_t(P_{t+1}) B_t\right]^{-1} B_t' T_t(P_{t+1}) A_t.$$

Operators T_t, D_t, $\mathcal{S}_t$ and decision rule F_t are constructed in `solve_riccati_step`.

Given value function parameters (P_{t+1}, ξ_{t+1}) at the next age, it constructs the state-space matrix A_t, applies the cross-product trick, evaluates the Riccati operators, and returns the optimal decision rule F_t, the closed-loop matrix $A_t^o = A_t - BF_t$, and the updated (P_t, ξ_t)

```
def solve_riccati_step(
        ε_t, β_t, Ind_work_t,
        RR, w, τ_l, τ_a, τ_0, benef,
        P_next, ξ_next,
        ρ_d, σ, B, C, R, Q, H):
    """One backward step of the risk-sensitive Riccati recursion."""

    A = jnp.array([
        [RR * (1.0 - τ_a) + τ_a,
         (1.0 - τ_l) * w * ε_t
         - τ_0 + benef * (1.0 - Ind_work_t),
         (1.0 - τ_l) * Ind_work_t],
        [0.0, 1.0, 0.0],
        [0.0, 0.0, ρ_d]
    ])

    Q_scalar = Q[0, 0]
    Q_inv_scalar = 1.0 / Q_scalar
    Q_inv = jnp.array([[Q_inv_scalar]])

    # Cross-product trick: A* = A - B Q^{-1} H
    A = A - B @ Q_inv @ H
```

(continues on next page)

(continued from previous page)

```
    # T_t operator
    CTP = C.T @ P_next @ C
    PP_scalar = 1.0 - σ * CTP[0, 0]
    PP_inv_scalar = 1.0 / PP_scalar
    PC = P_next @ C
    CP = C.T @ P_next
    TP = P_next + σ * PP_inv_scalar * (PC @ CP)

    # D_t operator and decision rule F_t
    BTB_scalar = (B.T @ TP @ B)[0, 0]
    Q_BTB_scalar = Q_scalar + β_t * BTB_scalar
    Q_BTB_inv_scalar = 1.0 / Q_BTB_scalar
    BT_TP = B.T @ TP
    BT_TP_A = BT_TP @ A
    F = β_t * Q_BTB_inv_scalar * BT_TP_A

    TP_B = TP @ B
    middle = (β_t * TP
              - β_t**2 * Q_BTB_inv_scalar
              * (TP_B @ BT_TP))
    P = R + A.T @ middle @ A

    # S_t operator
    log_det_PP = jnp.log(PP_scalar)
    ξ = jnp.where(
        σ != 0.0,
        β_t * (ξ_next - log_det_PP / σ),
        β_t * (ξ_next + CTP[0, 0])
    )

    Ao = A - B @ F
    F = F + Q_inv @ H

    return F.squeeze(), Ao, P, ξ
```

116.4.2 State space preparation

The budget constraint (116.1) and the income process can be written in state-space form.

Let $x_t = [a_{t-1}(s-1),\ 1,\ d_t]'$, $u_t = c_t(s)$, and $w_{t+1} = \epsilon_{t+1}$, so that

$$x_{t+1} = A_t x_t + B u_t + C w_{t+1}$$

where

$$A_t = \begin{bmatrix} R(s-1)(1-\tau_t^a(s)) + \tau_t^a(s) & (1-\tau_\ell(s))w(s)\varepsilon_t - \tau_0(s) + S_t(s) & \mathbf{1}_t^{\text{work}}(1-\tau_\ell(s)) \\ 0 & 1 & 0 \\ 0 & 0 & \rho_d \end{bmatrix},$$

$$B = \begin{bmatrix} -1 \\ 0 \\ 0 \end{bmatrix},$$

$$C = \begin{bmatrix} 0 \\ 0 \\ \sigma_d \end{bmatrix}$$

and $\mathbf{1}_t^{\text{work}}$ indicates whether the agent is of working age.

The per-period return $-\frac{1}{2}(\pi c_t - \gamma)^2$ introduces a cross-product term H between the control and the state.

This is eliminated using the cross-product trick (see *Eliminating Cross Products*):

$$A^* = A - BQ^{-1}H, \qquad R^* = R - H'Q^{-1}H.$$

116.4.3 Means and covariances

Define $A_o = A - BF$ as the closed-loop transition matrix, so that $x_{t+1} = A_o x_t + C w_{t+1}$ and unconditional moments satisfy

$$\mu_{t+1} = A_{o,t}\mu_t, \qquad \Sigma_{t+1} = A_{o,t}\Sigma_t A_{o,t}' + CC'.$$

The moment recursion is implemented as `forward_moment_step`, which propagates the mean vector and covariance matrix by one age step and computes consumption statistics as by-products.

```
def forward_moment_step(μx_t, Σx_t, Ao_t, F_t, CCT):
    """One step of the forward moment recursion."""

    μx_next = Ao_t @ μx_t
    μc_t = -F_t @ μx_t
    Σx_next = CCT + Ao_t @ Σx_t @ Ao_t.T
    Vc_t = F_t @ Σx_t @ F_t.T
    return μx_next, μc_t, Σx_next, Vc_t
```

116.4.4 Computing transitions

A cohort born at s lives during $s, s+1, \ldots, s+T_0$ and works during $s, s+1, \ldots, s+T_1$.

Let $0 \leq s_1 < s_2 < s_3$.

- At $s = 0$, the government announces a policy change between $s = s_1$ and $s = s_2$.
- From $s = s_2$, government policies are constant forever.
- From $s = s_3$, convergence to the final stationary equilibrium is achieved (in the small open economy, $s_3 = s_2 + T_0$).

The affected cohorts are those born at $s = s_1 - T_0, s_1 - T_0 + 1, \ldots, s_2$.

In all exercises, we set $T_0 = 65$, $T_1 = 43$, $s_1 = 0$, and $s_2 = 40$.

116.5 Calibration

The model parameters are set as follows.

116.5.1 Preference parameters

Parameter	Description	Value
$\{\alpha_t\}_{t=0}^{T_0}$	Age-to-age survival probabilities	Faber [1982]
π	Consumption preference parameter	1.0
σ	Risk-sensitivity parameter	-0.05
$\bar{\gamma}$	Preference shock parameter	7.0
$\tilde{\beta}$	Discount factor	0.986
T_0	Maximum age	65
T_1	Retirement age	43
n	Gross population growth rate	1.012

116.5.2 Technology parameters

Parameter	Description	Value
k_{-1}	Initial capital endowment	4.0
σ_d	Standard deviation of income shock	0.85
ρ_d	Persistence of income shock	0.8
δ	Depreciation rate	0.06
$\{\varepsilon_t\}_{t=0}^{T_1}$	Age-efficiency profile	Hansen [1993]
w	Base wage rate (exogenous)	5.0147
r	Return on capital (exogenous)	0.1275
$\tilde{A}$	Production function scaling (endogenous)	2.2625
$\tilde{\alpha}$	Capital share (endogenous)	0.40

```
N_GRID_SS = 10
TOL_SS = 1e-10

T0 = 65   # maximum lifespan (ages 21 to 86)
T1 = 43   # working life length (retire at 65)

UNIT_GRID = jnp.linspace(0.0, 1.0, N_GRID_SS)
AGE_INDICES = jnp.arange(T0 + 2)
```

The hidden code cell below defines the age-efficiency profile $\{\varepsilon_t\}$ and the survival probabilities $\{\alpha_t\}$ based on Faber [1982] and Hansen [1993].

```
fig, axs = plt.subplots(1, 2, figsize=(10, 6))

axs[0].plot(ε_arr)
axs[0].set_title("Working efficiency")
axs[0].set_xlabel("Age")

axs[1].plot(α_arr)
axs[1].set_title("Survival probability")
axs[1].set_xlabel("Age")

plt.tight_layout()
plt.show()
```

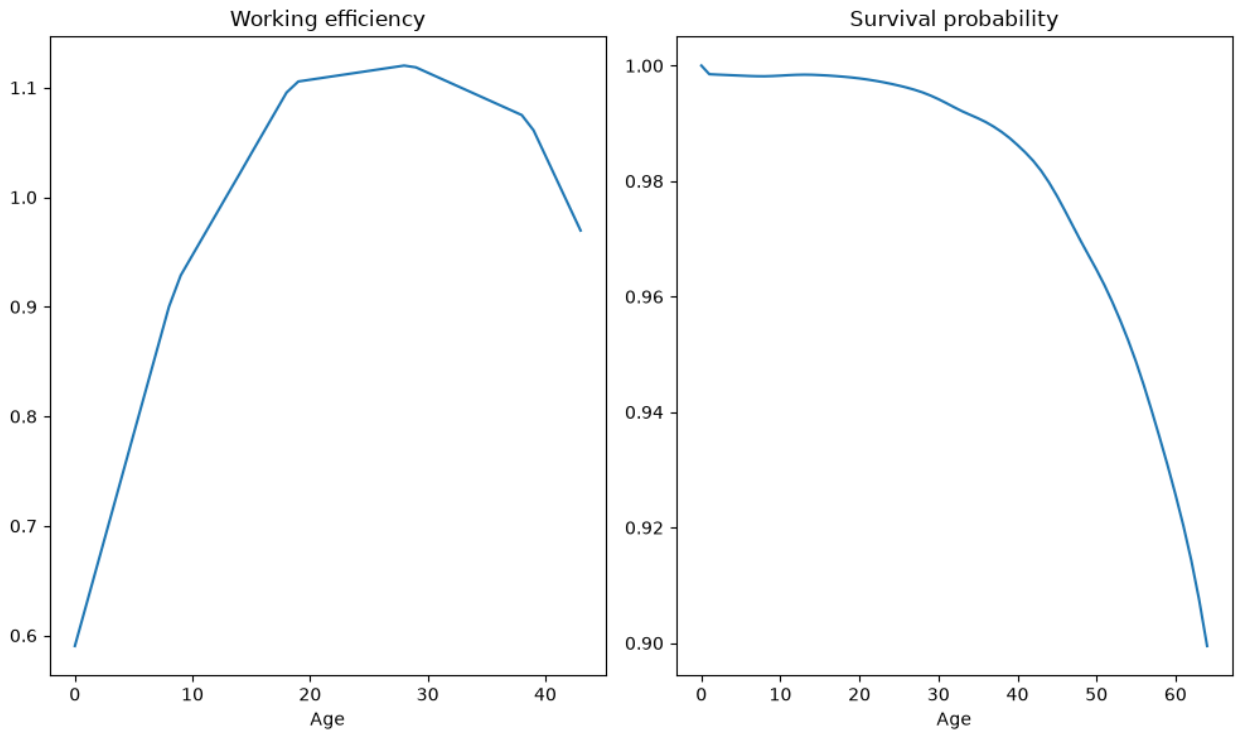


Fig. 116.2: Age-efficiency profile and survival probabilities

We impose a large penalty on terminal asset holdings to enforce the end-of-life condition, and set the initial state to $x_0 = [k_{-1},\ 1,\ 0]'$.

```
P_end = jnp.zeros((3, 3))
P_end = P_end.at[0, 0].set(-2000000.0)
ξ_end = 0.0

x0 = jnp.array([4.0, 1.0, 0.0])
Σ0 = jnp.zeros((3, 3))
```

All household parameters are collected into a named tuple.

```
Household = namedtuple('Household', (
    'α_arr', 'frac', 'n', 'π', 'σ', 'k_init', 'ε_arr', 'Ind_work',
    'σ_d', 'ρ_d', 'y_bar', 'β_arr',
    'T0', 'T1', 'T2', 'n_x', 'n_w',
    'P_end', 'ξ_end', 'x0', 'Σ0',
    'B', 'C', 'R', 'Q', 'H'
))

def create_household(α_arr=α_arr,     # Age-to-age survival probabilities
                     n=1.012,         # Gross population growth rate
                     π=1,             # Consumption preference parameter
                     σ=-0.05,         # Risk-sensitivity parameter
                     k_init=4,        # initial capital endowment
                     ε_arr=ε_arr,     # age-efficiency profile
                     σ_d=0.85,        # std of income shock
                     ρ_d=0.8,         # persistence of income shock
```

(continues on next page)

(continued from previous page)

```
                    γ_bar=7,         # Preference shock parameter
                    β_tilde=0.986,   # Discount factor
                    T0=65,           # Maximum age
                    T1=43,           # Retirement age
                    n_x=3,           # Number of states
                    n_w=1,           # Number of shocks
                    P_end=P_end,     # Terminal value
                    ξ_end=ξ_end,     # Terminal value
                    x0=x0,           # Initial mean
                    Σ0=Σ0):          # Initial variance
    """Create a Household named tuple with derived arrays."""

    α_arr = np.concatenate([α_arr, np.array([0])])
    T2 = T0 - T1

    frac = np.ones(T0 + 1)
    frac[1:] = np.cumprod(α_arr / n)[:-1]
    frac = frac / frac.sum()

    ε_arr = np.concatenate([ε_arr, np.zeros(T0 + 1 - ε_arr.size)])

    # Indicator for working ages: 1 if working (ε > 0), 0 if retired
    Ind_work = (ε_arr != 0).astype(np.float64)

    β_arr = β_tilde * α_arr
    β_arr[-1] = β_tilde

    B = jnp.array([[-1.0, 0.0, 0.0]]).T
    C = jnp.array([[0.0, 0.0, σ_d]]).T

    Q = jnp.array([[-0.5 * π**2]])
    H = jnp.array([[0.0, 0.5 * π * γ_bar, 0.0]])

    # Apply cross-product trick: R* = R - H'Q^{-1}H
    R_base = np.array([[0.0, 0.0, 0.0],
                       [0.0, -0.5 * γ_bar**2, 0.0],
                       [0.0, 0.0, 0.0]])
    H_np = np.array([[0.0, 0.5 * π * γ_bar, 0.0]])
    Q_inv_np = np.array([[1.0 / (-0.5 * π**2)]])
    R = jnp.array(R_base - H_np.T @ Q_inv_np @ H_np)

    return Household(
        α_arr=jnp.array(α_arr), frac=jnp.array(frac), n=n, π=π, σ=σ,
        k_init=k_init, ε_arr=jnp.array(ε_arr), Ind_work=jnp.array(Ind_work),
        σ_d=σ_d, ρ_d=ρ_d, γ_bar=γ_bar, β_arr=jnp.array(β_arr),
        T0=T0, T1=T1, T2=T2, n_x=n_x, n_w=n_w,
        P_end=P_end, ξ_end=ξ_end, x0=x0, Σ0=Σ0,
        B=B, C=C, R=R, Q=Q, H=H
    )
```

```
hh = create_household()
```

The stationary population distribution follows.

```
fig, ax = plt.subplots()
ax.plot(hh.frac)
```

(continues on next page)

(continued from previous page)

```
ax.set_xlabel("Age")
ax.set_ylabel("Population fraction")
ax.set_title("Population distribution over age")
plt.show()
```

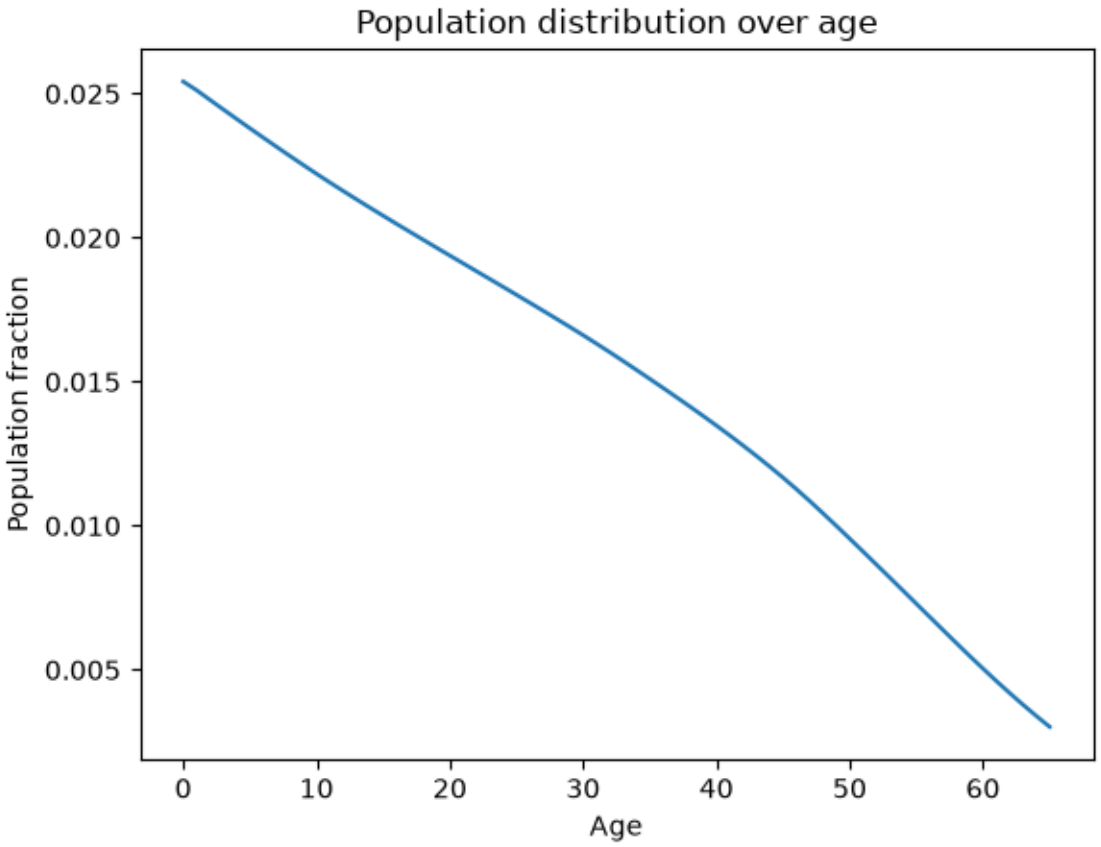


Fig. 116.3: Stationary population distribution over age

Mortality causes the population fraction to decline with age, a demographic pattern central to the intergenerational redistribution that social security reform entails.

Under the small open economy assumption, factor prices are fixed at calibrated values; under the closed economy assumption, they are determined by Cobb-Douglas marginal products

```
Tech = namedtuple('Tech', ('δ', 'w', 'r', 'RR', 'A', 'α_tilde'))

def create_Tech(δ=0.06, w=5.0147, r=0.1275,
                A=2.2625, α_tilde=0.40):
    """Create a Tech named tuple with factor price parameters."""

    RR = 1 + r - δ

    return Tech(δ=δ, w=w, r=r, RR=RR, A=A, α_tilde=α_tilde)
```

```
tech = create_Tech()
```

116.6 Individual optimality

116.6.1 Steady-state computation

With `solve_riccati_step` and `forward_moment_step` in hand, the steady-state computation proceeds in three phases.

Algorithm 116.6.1 (Steady-state computation)

1. **Backward recursion:** scan from age T_0 down to 0. At each age t, call `solve_riccati_step` to obtain decision rule F_t, closed-loop matrix A_t^o, value-function matrix P_t, and certainty-equivalent ξ_t.
2. **Forward simulation:** scan from age 0 up to T_0. At each age t, call `forward_moment_step` to propagate mean $\mu_{x,t}$ and covariance $\Sigma_{x,t}$ of the state vector, and record mean consumption $\mu_{c,t}$ and its variance $V_{c,t}$.
3. **Budget imbalance:** aggregate across cohorts. Sum tax revenues (labor, capital, lump-sum), subtract benefit payments, add accidental bequests, and return the government budget gap.

Phase 1. The backward recursion scans from age T_0 to 0, applying `solve_riccati_step` at each age

```
def _ss_backward_recursion(
        ε_arr, β_arr, Ind_work,
        RR, w, τ_l, τ_a, τ_0, benef,
        P_end, ξ_end,
        ρ_d, σ, B, C, R, Q, H):
    """Backward Riccati scan over all ages."""

    ε_rev = ε_arr[::-1]
    β_rev = β_arr[::-1]
    Ind_work_rev = Ind_work[::-1]

    def backward_step(carry, inputs):
        P_next, ξ_next = carry
        ε_t, β_t, Ind_work_t = inputs
        F, Ao, P, ξ = solve_riccati_step(
            ε_t, β_t, Ind_work_t,
            RR, w, τ_l, τ_a, τ_0, benef,
            P_next, ξ_next,
            ρ_d, σ, B, C, R, Q, H
        )
        return (P, ξ), (F, Ao, P, ξ)

    init_carry = (P_end, ξ_end)
    _, (F_rev, Ao_rev, P_rev, ξ_rev) = lax.scan(
        backward_step, init_carry,
        (ε_rev, β_rev, Ind_work_rev)
    )

    F_arr = F_rev[::-1]
    Ao_arr = Ao_rev[::-1]
    P_inner = P_rev[::-1]
```

(continues on next page)

(continued from previous page)

```
    ξ_inner = ξ_rev[::-1]

    P_arr = jnp.concatenate(
        [P_inner, P_end[None, :, :]], axis=0
    )
    ξ_arr = jnp.concatenate(
        [ξ_inner, jnp.array([ξ_end])]
    )

    return F_arr, Ao_arr, P_arr, ξ_arr
```

Phase 2. The forward recursion propagates means and covariances from age 0 to T_0

```
def _ss_forward_simulation(
        Ao_arr, F_arr, x0, Σ0, C):
    """Forward moment scan using forward_moment_step."""

    CCT = C @ C.T

    def forward_step(carry, inputs):
        μx_t, Σx_t = carry
        Ao_t, F_t = inputs
        result = forward_moment_step(
            μx_t, Σx_t, Ao_t, F_t, CCT
        )
        μx_next, μc_t, Σx_next, Vc_t = result
        return (μx_next, Σx_next), \
            (μx_next, μc_t, Σx_next, Vc_t)

    init_carry = (x0, Σ0)
    _, (μx_scn, μc_arr, Σx_scn, Vc_arr) = lax.scan(
        forward_step, init_carry, (Ao_arr, F_arr)
    )

    μx_arr = jnp.concatenate(
        [x0[None, :], μx_scn], axis=0
    )
    Σx_arr = jnp.concatenate(
        [Σ0[None, :, :], Σx_scn], axis=0
    )

    return μx_arr, μc_arr, Σx_arr, Vc_arr
```

Phase 3. Aggregating tax revenues, benefit payments, and accidental bequests across cohorts gives the government budget gap

```
def _ss_budget_imbalance(
        μx_arr, ε_arr, frac, n, α_arr,
        RR, w, τ_l, τ_a, τ_0, benef,
        G, Gb, Ind_work):
    """Aggregate tax revenues and expenditures."""

    μa_arr = μx_arr[1:, 0]
    μa_last_arr = μx_arr[:-1, 0]

    τ_l_tot = jnp.sum(τ_l * ε_arr * w * frac)
    τ_a_tot = jnp.sum(
```

(continues on next page)

(continued from previous page)

```
        τ_a * (RR - 1.0) * μa_last_arr * frac
    )
    τ_0_tot = jnp.sum(τ_0 * frac)

    retired_mask = 1.0 - Ind_work
    benef_tot = jnp.sum(benef * frac * retired_mask)

    Beq = jnp.sum(
        RR * (1.0 - α_arr) * frac * μa_arr / n
    )

    T_tot = τ_l_tot + τ_a_tot + τ_0_tot + Beq
    diff = (G + benef_tot - T_tot
            + (RR / n - 1.0) * Gb)

    return diff
```

A steady state is found when the budget gap equals zero.

`ss_imbalance` chains the three phases into a single JIT-compiled function: backward recursion, forward simulation, and budget gap

```
@jit
def ss_imbalance(price, policy, α_arr, ε_arr, frac,
                 n, β_arr, ρ_d, σ, B, C, R, Q, H,
                 P_end, ξ_end, x0, Σ0, Ind_work):
    """Backward solve, forward simulate, and return budget gap."""

    RR, w = price
    τ_l, τ_a, τ_0, benef, G, Gb = policy

    F_arr, Ao_arr, P_arr, ξ_arr = \
        _ss_backward_recursion(
            ε_arr, β_arr, Ind_work,
            RR, w, τ_l, τ_a, τ_0, benef,
            P_end, ξ_end,
            ρ_d, σ, B, C, R, Q, H
        )

    μx_arr, μc_arr, Σx_arr, Vc_arr = \
        _ss_forward_simulation(
            Ao_arr, F_arr, x0, Σ0, C
        )

    diff = _ss_budget_imbalance(
        μx_arr, ε_arr, frac, n, α_arr,
        RR, w, τ_l, τ_a, τ_0, benef,
        G, Gb, Ind_work
    )

    return (diff, P_arr, ξ_arr, Ao_arr, F_arr,
            μx_arr, μc_arr, Σx_arr, Vc_arr)
```

A named tuple `SteadyState` collects value-function parameters, decision rules, moments, and aggregate statistics into a single object that the transition solver can unpack

```
SteadyState = namedtuple("SteadyState", (
    "P_arr",      # Value function matrices by age
    "ξ_arr",      # Certainty equivalent adjustments by age
    "Ao_arr",     # Closed-loop transition matrices by age
    "F_arr",      # Decision rule matrices by age
    "μx_arr",     # Mean state vectors by age
    "μc_arr",     # Mean consumption by age
    "Σx_arr",     # Covariance matrices by age
    "Vc_arr",     # Consumption variances by age
    "debt2gdp",   # Government debt to GDP ratio
    "τ_l",        # Labor income tax rate
    "benef",      # Social security benefit level
    "Gb",         # Per-capita government debt
    "k_bar",      # Per-capita capital stock
    "RR",         # Gross return on assets
    "w",          # Wage rate
    "r",          # Interest rate (before depreciation)
    "k2gdp"       # Capital to GDP ratio
))
```

Given all other fiscal instruments, the equilibrium τ_ℓ is the value that zeroes the budget gap.

We find it by iterative grid refinement: evaluate the gap on a coarse grid, zoom into the best interval, and repeat

```
def _grid_refine(eval_fn, a_init, b_init, unit_grid, tol, max_iter):
    """Iterative grid-refinement root search.

    Must be called inside @jit functions.
    """

    n_grid = unit_grid.shape[0]

    def cond_fn(state):
        a, b, best_val, i = state
        return (jnp.abs(best_val) > tol) & (i < max_iter)

    def body_fn(state):
        a, b, _, i = state
        grid = a + (b - a) * unit_grid
        diffs = vmap(eval_fn)(grid)
        best_idx = jnp.argmin(jnp.abs(diffs))
        best_val = diffs[best_idx]
        idx_lo = jnp.maximum(best_idx - 1, 0)
        idx_hi = jnp.minimum(best_idx + 1, n_grid - 1)
        return (grid[idx_lo], grid[idx_hi], best_val, i + 1)

    grid = a_init + (b_init - a_init) * unit_grid
    diffs = vmap(eval_fn)(grid)
    best_idx = jnp.argmin(jnp.abs(diffs))
    best_val = diffs[best_idx]
    idx_lo = jnp.maximum(best_idx - 1, 0)
    idx_hi = jnp.minimum(best_idx + 1, n_grid - 1)

    init_state = (grid[idx_lo], grid[idx_hi], best_val, 0)
    final_state = lax.while_loop(cond_fn, body_fn, init_state)
    a_final, b_final, _, _ = final_state
    return (a_final + b_final) / 2.0
```

`_ss_diff_for_tau_l` evaluates the budget gap at a given τ_ℓ, and `_find_ss_tau_l` wraps it inside the grid-refine loop.

```
def _ss_diff_for_tau_l(τ_l, price_arr, policy_no_τl, α_arr, ε_arr, frac, n,
                       β_arr, ρ_d, σ, B, C, R, Q, H, P_end, ξ_end, x0, Σ0,
                       Ind_work):
    """Budget imbalance for a given τ_l."""

    τ_a, τ_0, benef, G, Gb = policy_no_τl
    policy_arr = jnp.array([τ_l, τ_a, τ_0, benef, G, Gb])
    diff, *_ = ss_imbalance(
        price_arr, policy_arr,
        α_arr, ε_arr, frac, n,
        β_arr, ρ_d, σ, B, C, R, Q, H, P_end, ξ_end, x0, Σ0,
        Ind_work
    )
    return diff
```

```
@jit
def _find_ss_tau_l(price_arr, policy_no_τl, α_arr, ε_arr, frac, n,
                   β_arr, ρ_d, σ, B, C, R, Q, H, P_end, ξ_end, x0, Σ0,
                   Ind_work, unit_grid):
    """Find τ_l that balances the steady-state budget."""

    def compute_diff(τ_l):
        return _ss_diff_for_tau_l(
            τ_l, price_arr, policy_no_τl, α_arr, ε_arr, frac, n,
            β_arr, ρ_d, σ, B, C, R, Q, H, P_end, ξ_end, x0, Σ0,
            Ind_work
        )

    return _grid_refine(compute_diff, -0.5, 1.0 - 1e-5, unit_grid, TOL_SS, 10)
```

GDP is the sum of capital and labor income shares

```
def _compute_gdp(μa_arr, frac, ε_arr, Gb, r, w, n, x0_0, frac_0):
    """Compute GDP from aggregates."""

    ε_agg = jnp.sum(frac * ε_arr)
    a_agg = jnp.sum(frac * μa_arr)
    k_agg = a_agg - Gb
    k_share = r * (k_agg / n + frac_0 * x0_0)
    l_share = w * ε_agg
    return k_agg, k_share + l_share
```

`find_ss_exo` ties the pieces together: it solves for τ_ℓ, evaluates the full steady state, and returns a `SteadyState` named tuple

```
def find_ss_exo(price, policy_target, hh, tech):
    """Find steady state with exogenous prices by solving for τ_l."""

    frac, ε_arr, n, x0 = hh.frac, hh.ε_arr, hh.n, hh.x0
    RR, w = price
    r = RR - 1 + tech.δ

    τ_a, τ_0, benef, G, Gb = policy_target
```

(continues on next page)

(continued from previous page)

```
    price_arr = jnp.array([RR, w])
    policy_no_τl = jnp.array([τ_a, τ_0, benef, G, Gb])

    τ_l = _find_ss_tau_l(
        price_arr, policy_no_τl,
        hh.α_arr, hh.ε_arr, hh.frac, hh.n,
        hh.β_arr, hh.ρ_d, hh.σ, hh.B, hh.C, hh.R, hh.Q, hh.H,
        hh.P_end, hh.ξ_end, hh.x0, hh.Σ0,
        hh.Ind_work, UNIT_GRID
    )

    price_arr = jnp.array([RR, w])
    policy_arr = jnp.array([float(τ_l), τ_a, τ_0, benef, G, Gb])
    diff, P_arr, ξ_arr, Ao_arr, F_arr, μx_arr, μc_arr, Σx_arr, Vc_arr = \
        ss_imbalance(
            price_arr, policy_arr,
            hh.α_arr, hh.ε_arr, hh.frac, hh.n,
            hh.β_arr, hh.ρ_d, hh.σ,
            hh.B, hh.C, hh.R, hh.Q, hh.H,
            hh.P_end, hh.ξ_end, hh.x0, hh.Σ0,
            hh.Ind_work
        )

    k_agg, gdp = _compute_gdp(
        μx_arr[1:, 0], frac, ε_arr,
        Gb, r, w, n, x0[0], frac[0]
    )
    debt2gdp = Gb / gdp
    k2gdp = k_agg / gdp

    return SteadyState(
        P_arr=P_arr, ξ_arr=ξ_arr, Ao_arr=Ao_arr, F_arr=F_arr,
        μx_arr=μx_arr, μc_arr=μc_arr, Σx_arr=Σx_arr, Vc_arr=Vc_arr,
        debt2gdp=float(debt2gdp), τ_l=float(τ_l), benef=benef, Gb=Gb,
        k_bar=float(k_agg), RR=RR, w=w, r=float(r), k2gdp=float(k2gdp)
    )
```

The initial fiscal policy sets a social security replacement rate of $\theta = 0.6$

```
aveinc = tech.w * sum(hh.ε_arr) / (hh.T1 + 1)
θ = 0.6
benef_0 = aveinc * θ

G_0 = 1.44          # government purchases
Gb_0 = 2.8 * G_0    # government debt
τ_l_0 = 0.3385      # labor income tax
τ_a_0 = 0.30        # capital income tax
τ_0_0 = 0           # lump-sum tax

RR, w = tech.RR, tech.w
```

116.6.2 Initial and terminal steady states

The initial steady state features a calibrated replacement rate ($\theta = 0.6$), positive social security benefits, and government expenditure and debt set to match targets.

The transition dates are $s_1 = 0$ and $s_2 = 40$, with horizons $S = 140$ (exogenous prices) and $S = 200$ (endogenous prices).

```
S_exo = 140
S_endo = 200
S1, S2 = 0, 40
S3 = S2 + 2 * hh.T0

RR_exo, w_exo = tech.RR, tech.w
```

Two helper functions build the price and policy arrays that the transition solver expects.

Under the small open economy assumption, prices are constant over time.

```
def make_exo_price_seq(S, RR, w):
    """Construct constant price sequence for small open economy."""

    return jnp.column_stack([jnp.full(S + 2, RR), jnp.full(S + 2, w)])
```

The policy sequence sets τ_ℓ to the initial steady-state value before s_1 and to the terminal value after s_2, while holding all other fiscal instruments constant.

```
def make_policy_seq(S, ss0_τl, ss1_τl, S1, S2, τ_a, τ_0, benef, G, Gb):
    """Construct policy sequence with initial/terminal
    τ_l and constant other policies."""

    policy_seq = jnp.empty((S + 2, 6))
    policy_seq = policy_seq.at[:S1 + 1, 0].set(ss0_τl)
    policy_seq = policy_seq.at[S2 + 1:, 0].set(ss1_τl)
    policy_seq = policy_seq.at[:, 1].set(τ_a)
    policy_seq = policy_seq.at[:, 2].set(τ_0)
    policy_seq = policy_seq.at[:, 3].set(benef)
    policy_seq = policy_seq.at[:, 4].set(G)
    policy_seq = policy_seq.at[:, 5].set(Gb)
    return policy_seq
```

```
ss0 = find_ss_exo((RR, w), (τ_a_0, τ_0_0, benef_0, G_0, Gb_0), hh, tech)

print(f"Initial Steady State (s < 0):")
print(f"  Labor tax τ_l = {ss0.τ_l:.4f}")
print(f"  Interest rate r - δ = {ss0.r - tech.δ:.4f}")
print(f"  Capital/GDP = {ss0.k2gdp:.4f}")
print(f"  Debt/GDP = {ss0.debt2gdp:.4f}")
```

```
Initial Steady State (s < 0):
  Labor tax τ_l = 0.3383
  Interest rate r - δ = 0.0675
  Capital/GDP = 3.1615
  Debt/GDP = 0.5899
```

The following figure traces how the equilibrium labor tax rate varies with government debt in the terminal steady state (no social security)

```
Gb_arr = np.linspace(0.5 * Gb_0, 1.5 * Gb_0, 20)
τl_arr = np.empty_like(Gb_arr)
debt2gdp_arr = np.empty_like(Gb_arr)

for i, Gb in enumerate(Gb_arr):
    ss = find_ss_exo((RR, w), (τ_a_0, τ_0_0, 0, G_0, Gb), hh, tech)
    τl_arr[i] = ss.τ_l
    debt2gdp_arr[i] = ss.debt2gdp

fig, ax = plt.subplots()
ax.plot(τl_arr, debt2gdp_arr)
ax.hlines(ss0.debt2gdp, τl_arr.min(),
          np.maximum(τl_arr.max(), ss0.τ_l), linestyle='--', color='r')
ax.scatter(ss0.τ_l, ss0.debt2gdp)
ax.text(ss0.τ_l * 0.95, ss0.debt2gdp * 0.95, "ss0")
ax.text(0.07, 0.4, r"ss1($G_b$)")
ax.set_xlabel(r'$\tau_\ell$')
ax.set_ylabel('Debt/GDP')
plt.show()
```

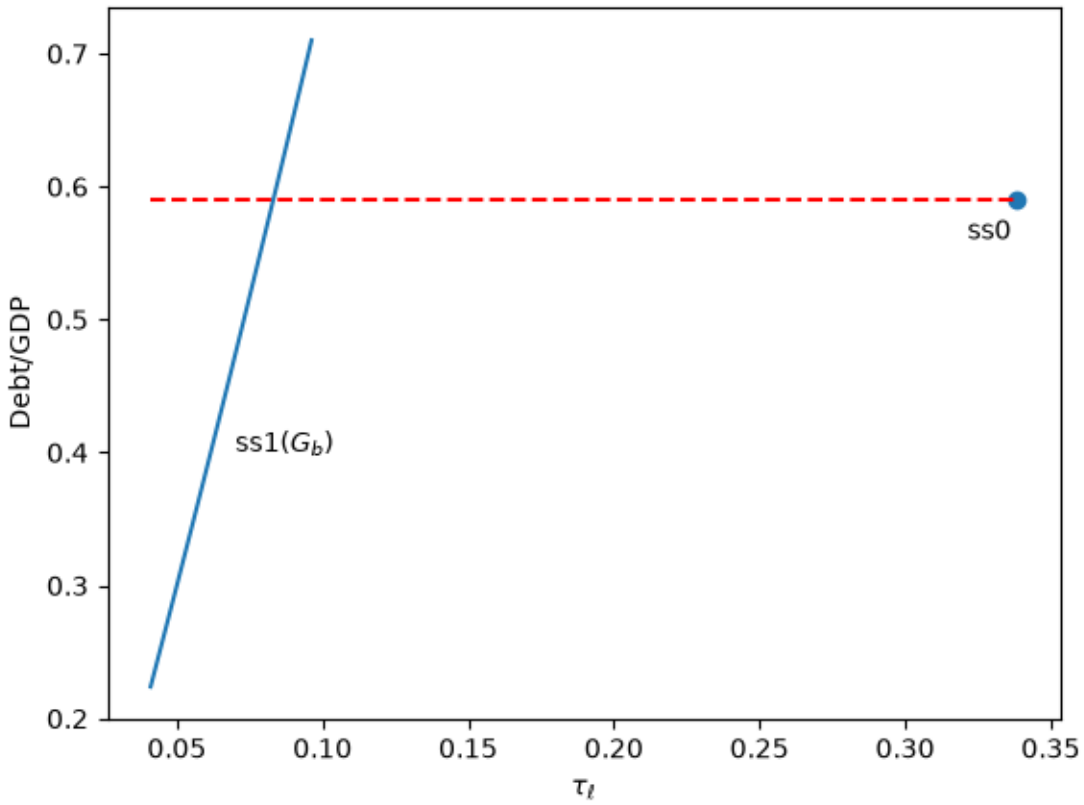


Fig. 116.4: Debt-to-GDP ratio as a function of the labor tax rate

Higher government debt requires larger interest payments, so the equilibrium labor tax rises.

The marked point shows the initial steady state.

To set the terminal steady state, we need to invert this relationship: given a target debt-to-GDP ratio, find the debt level

$\bar{b}$ and the associated τ_ℓ.

```
def _compute_debt2gdp_for_Gb(
        Gb, price_arr, policy_no_Gb,
        α_arr, ε_arr, frac, n,
        β_arr, ρ_d, σ, B, C, R, Q, H,
        P_end, ξ_end, x0, Σ0,
        δ, Ind_work, unit_grid):
    """Compute debt-to-GDP ratio for a given Gb."""

    RR, w = price_arr
    τ_a, τ_0, benef, G = policy_no_Gb
    r = RR - 1 + δ

    policy_no_τl = jnp.array([τ_a, τ_0, benef, G, Gb])

    # Reuse _find_ss_tau_l instead of duplicating grid search
    τ_l = _find_ss_tau_l(
        price_arr, policy_no_τl, α_arr, ε_arr, frac, n,
        β_arr, ρ_d, σ, B, C, R, Q, H, P_end, ξ_end, x0, Σ0,
        Ind_work, unit_grid
    )

    policy_arr = jnp.array([τ_l, τ_a, τ_0, benef, G, Gb])
    _, _, _, _, _, μx_arr, _, _, _ = ss_imbalance(
        price_arr, policy_arr,
        α_arr, ε_arr, frac, n,
        β_arr, ρ_d, σ, B, C, R, Q, H, P_end, ξ_end, x0, Σ0,
        Ind_work
    )

    _, gdp = _compute_gdp(
        μx_arr[1:, 0], frac, ε_arr,
        Gb, r, w, n, x0[0], frac[0]
    )
    return Gb / gdp
```

`_find_Gb_for_debt2gdp` searches over $\bar{b}$ values via grid refinement to match a target debt-to-GDP ratio.

```
@jit
def _find_Gb_for_debt2gdp(
        debt2gdp_target, price_arr, policy_no_Gb,
        α_arr, ε_arr, frac, n,
        β_arr, ρ_d, σ, B, C, R, Q, H,
        P_end, ξ_end, x0, Σ0,
        δ, Ind_work, unit_grid, unit_grid_tau):
    """Find Gb consistent with a target debt-to-GDP ratio."""

    RR, w = price_arr
    τ_a, τ_0, benef, G = policy_no_Gb

    def compute_diff_coarse(Gb):
        debt2gdp = _compute_debt2gdp_for_Gb(
            Gb, price_arr, policy_no_Gb, α_arr, ε_arr, frac, n,
            β_arr, ρ_d, σ, B, C, R, Q, H, P_end, ξ_end, x0, Σ0,
            δ, Ind_work, unit_grid_tau
        )
        return debt2gdp - debt2gdp_target
```

(continues on next page)

(continued from previous page)

```
    return _grid_refine(compute_diff_coarse, -40.0, 20.0, unit_grid, TOL_SS, 5)
```

`ss_target_debt2gdp_exo` finds the debt level consistent with the target ratio, then computes the full steady state.

```
def ss_target_debt2gdp_exo(debt2gdp_target, policy_target, price, hh, tech):
    """Find steady state with target debt-to-GDP ratio."""

    τ_a, τ_0, benef, G = policy_target
    RR, w = price

    price_arr = jnp.array([RR, w])
    policy_no_Gb = jnp.array([τ_a, τ_0, benef, G])

    Gb = _find_Gb_for_debt2gdp(
        float(debt2gdp_target), price_arr, policy_no_Gb,
        hh.α_arr, hh.ε_arr, hh.frac, hh.n,
        hh.β_arr, hh.ρ_d, hh.σ, hh.B, hh.C, hh.R, hh.Q, hh.H,
        hh.P_end, hh.ξ_end, hh.x0, hh.Ξ0, tech.δ,
        hh.Ind_work, UNIT_GRID, UNIT_GRID
    )

    return find_ss_exo((RR, w), (τ_a, τ_0, benef, G, float(Gb)), hh, tech)
```

The terminal steady state eliminates social security ($\theta = 0$) while matching the initial debt-to-GDP ratio

```
ss1 = ss_target_debt2gdp_exo(
    ss0.debt2gdp, (τ_a_0, τ_0_0, 0, G_0), (RR_exo, w_exo), hh, tech
)

print(f"\nTerminal Steady State (s >= s3):")
print(f"  Labor tax τ_l = {ss1.τ_l:.4f}")
print(f"  Benefits θ = 0")
print(f"  Capital/GDP = {ss1.k2gdp:.4f}")
print(f"  Debt/GDP = {ss1.debt2gdp:.4f}")
```

```
Terminal Steady State (s >= s3):
  Labor tax τ_l = 0.0831
  Benefits θ = 0
  Capital/GDP = 4.1567
  Debt/GDP = 0.5901
```

116.7 Transition path computation

The transition path describes how the economy moves from the initial steady state (with social security) to the terminal steady state (after reform).

This is more complex than the steady-state computation because prices and policies change over time, so each cohort faces a unique lifetime sequence of tax and benefit rates.

`solve_backwards` solves the household problem backward in time during the transition, computing optimal decision rules $F_t(s)$ and closed-loop transition matrices $A_t^o(s)$ at each calendar date s and age t.

```
@jit
def solve_backwards(
        price_seq, policy_seq,
        P_arr_ss1, ξ_arr_ss1,
        ε_arr, β_arr, Ind_work,
        ρ_d, σ, B, C, R, Q, H,
        P_end, ξ_end,
        s_indices, ages):
    """Backward Riccati scan over all dates and ages."""

    # Infer dimensions from input arrays
    n_x = P_end.shape[0]
    S = s_indices.shape[0] - 1

    def solve_all_ages(P_next_all, ξ_next_all, RR_s, w_s, τ_l, τ_a, τ_0, benef):
        def solve_one_age(t, P_next, ξ_next):
            ε_t = ε_arr[t]
            β_t = β_arr[t]
            Ind_work_t = Ind_work[t]
            F, Ao, P, ξ = solve_riccati_step(
                ε_t, β_t, Ind_work_t, RR_s, w_s, τ_l, τ_a, τ_0, benef,
                P_next, ξ_next, ρ_d, σ, B, C, R, Q, H
            )
            return F, Ao, P, ξ

        P_next_shifted = P_next_all[1:]
        ξ_next_shifted = ξ_next_all[1:]

        F_all, Ao_all, P_all, ξ_all = vmap(
            solve_one_age
        )(ages, P_next_shifted, ξ_next_shifted)
        return F_all, Ao_all, P_all, ξ_all

    def scan_body(carry, s_inv):
        P_next_seq, ξ_next_seq = carry
        s = S - s_inv

        RR_s = price_seq[s, 0]
        w_s = price_seq[s, 1]
        τ_l = policy_seq[s, 0]
        τ_a = policy_seq[s, 1]
        τ_0 = policy_seq[s, 2]
        benef = policy_seq[s, 3]

        F_s, Ao_s, P_s, ξ_s = solve_all_ages(
            P_next_seq, ξ_next_seq, RR_s, w_s, τ_l, τ_a, τ_0, benef
        )

        # Build P_curr and ξ_curr using the known shapes from input arrays
        P_curr = jnp.zeros_like(P_arr_ss1)
        P_curr = P_curr.at[:T0+1].set(P_s)
        P_curr = P_curr.at[-1].set(P_end)

        ξ_curr = jnp.zeros_like(ξ_arr_ss1)
        ξ_curr = ξ_curr.at[:T0+1].set(ξ_s)
        ξ_curr = ξ_curr.at[-1].set(ξ_end)
```

(continues on next page)

(continued from previous page)

```
        output = (F_s, Ao_s, P_s, ξ_s)
        new_carry = (P_curr, ξ_curr)

        return new_carry, output

    init_carry = (P_arr_ss1, ξ_arr_ss1)

    # s_indices already has the right length
    _, outputs = lax.scan(scan_body, init_carry, s_indices)

    F_seq, Ao_seq, P_seq_inner, ξ_seq_inner = outputs

    F_seq = jnp.flip(F_seq, axis=0)
    Ao_seq = jnp.flip(Ao_seq, axis=0)

    # Build output arrays using shapes from price_seq
    P_seq = jnp.zeros((price_seq.shape[0], P_arr_ss1.shape[0], n_x, n_x))
    ξ_seq = jnp.zeros((price_seq.shape[0], ξ_arr_ss1.shape[0]))

    P_seq_inner = jnp.flip(P_seq_inner, axis=0)
    ξ_seq_inner = jnp.flip(ξ_seq_inner, axis=0)
    P_seq = P_seq.at[:S+1, :T0+1].set(P_seq_inner)
    ξ_seq = ξ_seq.at[:S+1, :T0+1].set(ξ_seq_inner)

    P_seq = P_seq.at[:, -1].set(P_end)
    ξ_seq = ξ_seq.at[:, -1].set(ξ_end)
    P_seq = P_seq.at[-1, :].set(P_arr_ss1)
    ξ_seq = ξ_seq.at[-1, :].set(ξ_arr_ss1)

    return F_seq, Ao_seq, P_seq, ξ_seq
```

`simulate_forwards` takes the computed decision rules and simulates the economy forward from the initial distribution, tracking the evolution of asset means and variances across cohorts through the transition.

```
@jit
def simulate_forwards(
        Ao_seq, F_seq, μx_init, Σx_init,
        C, x0, Σ0, s_indices, ages):
    """Forward moment scan over all dates and ages."""

    # Infer dimensions from input arrays
    n_x = x0.shape[0]
    CCT = C @ C.T
    S = s_indices.shape[0] - 1

    def simulate_all_ages(μx_curr, Σx_curr, Ao_s, F_s):
        def simulate_one_age(t, μx_t, Σx_t, Ao_t, F_t):
            return forward_moment_step(
                μx_t, Σx_t, Ao_t, F_t, CCT
            )

        μx_next_all, μc_all, Σx_next_all, Vc_all = vmap(simulate_one_age)(
            ages, μx_curr[:T0+1], Σx_curr[:T0+1], Ao_s, F_s
        )
        return μx_next_all, μc_all, Σx_next_all, Vc_all

    def scan_body(carry, s):
```

(continues on next page)

(continued from previous page)

```
        μx_curr, Σx_curr = carry

        Ao_s = Ao_seq[s]
        F_s = F_seq[s]

        μx_next_inner, μc_s, Σx_next_inner, Vc_s = simulate_all_ages(
            μx_curr, Σx_curr, Ao_s, F_s
        )

        # Use shapes from μx_init
        μx_next = jnp.zeros_like(μx_init)
        μx_next = μx_next.at[0].set(x0)
        μx_next = μx_next.at[1:T0+2].set(μx_next_inner)

        Σx_next = jnp.zeros_like(Σx_init)
        Σx_next = Σx_next.at[0].set(Σ0)
        Σx_next = Σx_next.at[1:T0+2].set(Σx_next_inner)

        output = (μx_curr, μc_s, Σx_curr, Vc_s)
        new_carry = (μx_next, Σx_next)

        return new_carry, output

    init_carry = (μx_init, Σx_init)
    final_carry, outputs = lax.scan(scan_body, init_carry, s_indices)

    μx_seq_inner, μc_seq, Σx_seq_inner, Vc_seq = outputs

    # Build output arrays using inferred sizes
    μx_seq = jnp.zeros((S + 2, T0 + 2, n_x))
    Σx_seq = jnp.zeros((S + 2, T0 + 2, n_x, n_x))

    μx_seq = μx_seq.at[:S+1].set(μx_seq_inner)
    Σx_seq = Σx_seq.at[:S+1].set(Σx_seq_inner)

    μx_seq = μx_seq.at[S+1].set(final_carry[0])
    Σx_seq = Σx_seq.at[S+1].set(final_carry[1])

    return μx_seq, μc_seq, Σx_seq, Vc_seq
```

Given a candidate transition tax rate τ_ℓ^{trans}, the function `transition_paths` constructs the complete policy sequence, solves backward, simulates forward, computes the capital and debt paths by aggregating across cohorts, and returns the terminal debt carryover that we seek to drive to zero.

```
@jit
def _transition_paths(
        τ_l_trans, price_seq, policy_seq,
        ss1_P_arr, ss1_ξ_arr, ss1_Gb,
        μx_init, Σx_init, k_bar_init,
        s_indices, age_range, S1, S2,
        ε_arr, β_arr, Ind_work,
        ρ_d, σ, B, C, R, Q, H,
        P_end, ξ_end, x0, Σ0,
        frac, n):
    """Solve backward and simulate forward for a given τ_l_trans."""
```

(continues on next page)

(continued from previous page)

```
    # Infer dimensions from input arrays
    n_x = x0.shape[0]
    S = s_indices.shape[0] - 2

    # Derive variants via slicing
    s_indices_scan = s_indices[:-1]      # arange(S+1)
    ages = age_range[:-1]                 # arange(T0+1)
    capital_indices = s_indices[1:-1]    # arange(1, S+1)

    # Update policy sequence with transition tax using dynamic indexing
    mask = (s_indices >= S1 + 1) & (s_indices <= S2)
    τ_l_col = jnp.where(mask, τ_l_trans, policy_seq[:, 0])
    policy_seq = policy_seq.at[:, 0].set(τ_l_col)

    # Solve backwards
    F_seq, Ao_seq, P_seq, ξ_seq = solve_backwards(
        price_seq, policy_seq, ss1_P_arr, ss1_ξ_arr,
        ε_arr, β_arr, Ind_work,
        ρ_d, σ, B, C, R, Q, H,
        P_end, ξ_end,
        s_indices_scan, ages
    )

    # Simulate forwards
    μx_seq, μc_seq, Σx_seq, Vc_seq = simulate_forwards(
        Ao_seq, F_seq, μx_init, Σx_init, C, x0, Σ0, s_indices_scan, ages
    )

    # Compute capital path
    ε_agg = jnp.sum(ε_arr * frac)
    frac0_x0 = frac[0] * x0[0]

    def capital_step(k_prev, s):
        RR = price_seq[s, 0]
        w = price_seq[s, 1]
        G = policy_seq[s, 4]
        c_agg = jnp.sum(μc_seq[s] * frac)
        k_new = RR * (frac0_x0 + k_prev / n) - G - c_agg + w * ε_agg
        return k_new, k_new

    # capital_indices is pre-created arange(1, S+1)
    _, k_path = lax.scan(capital_step, k_bar_init, capital_indices)
    k_seq = jnp.concatenate([jnp.array([k_bar_init]), k_path])

    # Compute debt path
    a_seq = jnp.sum(μx_seq[1:, 1:, 0] * frac, axis=1)
    Gb_seq = a_seq - k_seq

    carryover = Gb_seq[-1] - ss1_Gb

    return carryover, μx_seq, μc_seq, k_seq, Gb_seq, F_seq, Ao_seq
```

`transition_paths` unpacks the steady-state and household objects and calls the JIT-compiled inner function.

```
def transition_paths(
        τ_l_trans, price_seq, policy_seq,
        ss0, ss1, hh, tech,
```

(continues on next page)

(continued from previous page)

```
        S, S1, S2, μx_init, Σx_init):
    """Compute transition path."""

    policy_seq = jnp.asarray(policy_seq)
    price_seq = jnp.asarray(price_seq)

    # Pre-create iteration arrays (use slicing for variants)
    s_indices = jnp.arange(S + 2)

    carryover, μx_seq, μc_seq, k_seq, Gb_seq, F_seq, Ao_seq = _transition_paths(
        float(τ_l_trans), price_seq, policy_seq,
        ss1.P_arr, ss1.ξ_arr, float(ss1.Gb),
        μx_init, Σx_init, float(ss0.k_bar),
        s_indices, AGE_INDICES,
        S1, S2,
        hh.ε_arr, hh.β_arr, hh.Ind_work,
        hh.ρ_d, hh.σ, hh.B, hh.C,
        hh.R, hh.Q, hh.H,
        hh.P_end, hh.ξ_end, hh.x0, hh.Σ0,
        hh.frac, hh.n
    )

    return (float(carryover), μx_seq, μc_seq,
            k_seq, Gb_seq, F_seq, Ao_seq)
```

116.7.1 Shooting method

To find the correct transition tax rate, we use a shooting method.

If the tax rate is too low, debt explodes; if it is too high, debt falls below the target.

The equilibrium tax rate is found where the terminal debt exactly hits the target.

We start by computing two transition paths with different trial tax rates to illustrate the shooting method

```
price_seq = make_exo_price_seq(S_exo, RR, w)
policy_seq_base = make_policy_seq(S_exo, ss0.τ_l, ss1.τ_l, S1, S2,
                                  τ_a_0, τ_0_0, 0, G_0, Gb_0)

τ_l_low = 0.14
τ_l_high = 0.17

_, μx_seq1, μc_seq1, k_seq1, Gb_seq1, _, _ = transition_paths(
    τ_l_low, price_seq, policy_seq_base, ss0, ss1, hh, tech,
    S_exo, S1, S2, ss0.μx_arr, ss0.Σx_arr)

_, μx_seq2, μc_seq2, k_seq2, Gb_seq2, _, _ = transition_paths(
    τ_l_high, price_seq, policy_seq_base, ss0, ss1, hh, tech,
    S_exo, S1, S2, ss0.μx_arr, ss0.Σx_arr)
```

We can plot the resulting debt paths to see how they differ under the two trial tax rates

```
fig, ax = plt.subplots(figsize=(10, 5))
ax.plot(Gb_seq1, 'b-', linewidth=2,
        label=f'$\\tau_\\ell$ = {τ_l_low:.2f} (too low)')
ax.plot(Gb_seq2, 'r-', linewidth=2,
```

(continues on next page)

(continued from previous page)

```
        label=f'$\\tau_\\ell$ = {τ_l_high:.2f} (too high)')
ax.axhline(ss1.Gb, color='k', linestyle='--',
           label=f'Target $G_b$ = {ss1.Gb:.2f}')
ax.axvspan(S1, S2, alpha=0.1, color='yellow', label='Transition period')
ax.set_xlabel('Time')
ax.set_ylabel('Government debt')
ax.legend()
plt.show()
```

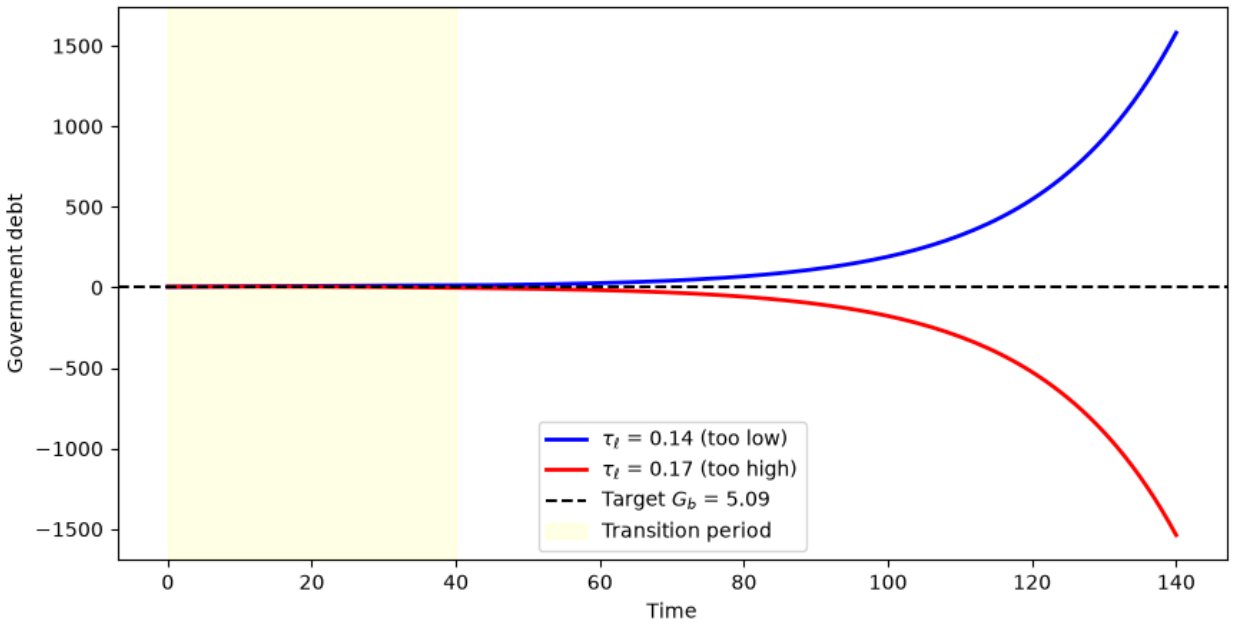


Fig. 116.5: Shooting method for finding the transition tax rate

The blue curve shows government debt increasing over time when the tax rate is too low, while the red curve shows debt falling below the target when the tax rate is too high.

The equilibrium transition tax rate lies between these two extremes, at the value where terminal debt exactly matches the target G_b.

Bisection automates this shooting procedure.

For Experiment 1, each cohort alive at s_1 must also be compensated by the present value of the social security benefits it would have received under the original system.

`_compute_compensation` evaluates this present value for a single cohort identified by its remaining lifetime.

```
def _compute_compensation(
        death_time, τ_l_seq, τ_a_seq,
        benef_diff, RR_seq, w_seq,
        ε_arr, indices, ss0_τ_l):
    """Present value of lost benefits for one cohort."""

    n_periods = death_time + 1
    age_at_0 = T0 - death_time

```

(continues on next page)

(continued from previous page)

```
    time_mask = indices < n_periods
    age_mask = indices >= age_at_0

    τ_l_cohort = jnp.where(time_mask, τ_l_seq[:T0 + 1], 0.0)
    τ_a_cohort = jnp.where(time_mask, τ_a_seq[:T0 + 1], 0.0)
    RR_cohort = jnp.where(time_mask, RR_seq[:T0 + 1], 1.0)
    w_cohort = jnp.where(time_mask, w_seq[:T0 + 1], 0.0)

    ε_masked = jnp.where(age_mask, ε_arr, 0.0)
    benef_masked = jnp.where(age_mask, benef_diff, 0.0)
    benef_masked = jnp.where(ε_masked != 0, 0.0, benef_masked)

    age_idx = jnp.clip(age_at_0 + indices, 0, T0)
    ε_cohort = jnp.where(
        time_mask, ε_arr[age_idx], 0.0
    )
    benef_cohort = jnp.where(
        time_mask, benef_diff[age_idx], 0.0
    )
    benef_cohort = jnp.where(ε_cohort != 0, 0.0, benef_cohort)

    RR_tilde_seq = RR_cohort - τ_a_cohort * (RR_cohort - 1)
    RR_tilde_seq = jnp.where(time_mask, RR_tilde_seq, 1.0)
    discount_factors = jnp.cumprod(RR_tilde_seq)

    labor_loss = w_cohort * ε_cohort * (τ_l_cohort - ss0_τ_l)

    pv_seq = jnp.where(
        time_mask,
        (benef_cohort + labor_loss) / discount_factors,
        0.0
    )

    valid = (death_time >= 0) & (death_time < T0)
    return jnp.where(valid, jnp.sum(pv_seq), 0.0)
```

`apply_compensation` vectorizes this calculation across all cohorts with `vmap` and adds the result to each cohort's initial asset holdings.

```
@jit
def apply_compensation(
        μx_arr_ss0, Σx_arr_ss0,
        τ_l_seq, τ_a_seq, benef_diff,
        RR_seq, w_seq, ε_arr, ss0_τ_l,
        ages_full, ages, x0, Σ0,
        comp_mult):
    """Vectorize compensation across cohorts and adjust initial assets."""

    def compute_comp_for_age(age):
        death_time = T0 - age
        comp = _compute_compensation(
            death_time,
            τ_l_seq, τ_a_seq, benef_diff,
            RR_seq, w_seq, ε_arr,
            ages, ss0_τ_l
        )
        valid = (age >= 1) & (age <= T0)
```

(continues on next page)

(continued from previous page)

```
        return jnp.where(valid, comp, 0.0)

    compensations = vmap(compute_comp_for_age)(ages_full)

    μx_init = jnp.zeros_like(μx_arr_ss0)
    Σx_init = jnp.zeros_like(Σx_arr_ss0)

    μx_init = μx_init.at[0].set(x0)
    Σx_init = Σx_init.at[0].set(Σ0)
    μx_init = μx_init.at[-1].set(x0)
    Σx_init = Σx_init.at[-1].set(Σ0)

    μx_init = μx_init.at[1:-1].set(μx_arr_ss0[1:-1])
    Σx_init = Σx_init.at[1:-1].set(Σx_arr_ss0[1:-1])

    # comp_mult: 0.0 = no compensation, 1.0 = full
    μx_init = μx_init.at[:, 0].add(comp_mult * compensations)

    return μx_init, Σx_init
```

`_transition_carryover` applies compensation, solves the transition, and returns the terminal debt carryover – the scalar the bisection drives to zero.

```
def _transition_carryover(
        τ_l_trans, price_seq, policy_seq,
        ss1_P_arr, ss1_ξ_arr, ss1_Gb,
        ss0_μx_arr, ss0_Σx_arr, k_bar_init,
        benef_diff, ss0_τ_l, comp_mult,
        s_indices, age_range, S1, S2,
        ε_arr, β_arr, Ind_work,
        ρ_d, σ, B, C, R, Q, H,
        P_end, ξ_end, x0, Σ0,
        frac, n):
    """Terminal debt carryover for a given transition τ_l."""

    ages = age_range[:-1]  # arange(T0+1)

    # Update policy sequence with transition tax
    mask = (s_indices >= S1 + 1) & (s_indices <= S2)
    τ_l_col = jnp.where(mask, τ_l_trans, policy_seq[:, 0])
    policy_seq_updated = policy_seq.at[:, 0].set(τ_l_col)

    # Compute initial conditions (compensation zeroed when comp_mult=0.0)
    μx_init, Σx_init = apply_compensation(
        ss0_μx_arr, ss0_Σx_arr,
        policy_seq_updated[:, 0], policy_seq_updated[:, 1], benef_diff,
        price_seq[:, 0], price_seq[:, 1],
        ε_arr, ss0_τ_l,
        age_range, ages, x0, Σ0,
        comp_mult
    )

    carryover, *_ = _transition_paths(
        τ_l_trans, price_seq, policy_seq,
        ss1_P_arr, ss1_ξ_arr, ss1_Gb,
        μx_init, Σx_init, k_bar_init,
        s_indices, age_range, S1, S2,
```

(continues on next page)

(continued from previous page)

```
        ε_arr, β_arr, Ind_work,
        ρ_d, σ, B, C, R, Q, H,
        P_end, ξ_end, x0, Σ0,
        frac, n
    )
    return carryover
```

We implement the bisection search in `_find_transition_tau_l`, which repeatedly evaluates `_transition_carryover` at the midpoint of a shrinking interval until the carryover is driven to zero.

```
@jit
def _find_transition_tau_l(
        price_seq, policy_seq, bounds,
        ss1_P_arr, ss1_ξ_arr, ss1_Gb,
        ss0_μx_arr, ss0_Σx_arr, k_bar_init,
        benef_diff, ss0_τ_l, comp_mult,
        s_indices, age_range, S1, S2,
        ε_arr, β_arr, Ind_work,
        ρ_d, σ, B, C, R, Q, H,
        P_end, ξ_end, x0, Σ0,
        frac, n):
    """Find transition τ_l using bisection.

    comp_mult controls compensation (0 or 1).
    """

    a, b = bounds[0], bounds[1]

    def compute_carryover(τ_l_trans):
        return _transition_carryover(
            τ_l_trans, price_seq, policy_seq,
            ss1_P_arr, ss1_ξ_arr, ss1_Gb,
            ss0_μx_arr, ss0_Σx_arr, k_bar_init,
            benef_diff, ss0_τ_l, comp_mult,
            s_indices, age_range,
            S1, S2,
            ε_arr, β_arr, Ind_work,
            ρ_d, σ, B, C, R, Q, H,
            P_end, ξ_end, x0, Σ0,
            frac, n
        )

    def cond_fn(state):
        a, b, fa, fb, i = state
        return (jnp.abs(b - a) > 1e-10) & (i < 100)

    def body_fn(state):
        a, b, fa, fb, i = state
        c = (a + b) / 2.0
        fc = compute_carryover(c)
        a_new = jnp.where(fa * fc > 0, c, a)
        b_new = jnp.where(fa * fc > 0, b, c)
        fa_new = jnp.where(fa * fc > 0, fc, fa)
        fb_new = jnp.where(fa * fc > 0, fb, fc)
        return (a_new, b_new, fa_new, fb_new, i + 1)

    fa, fb = compute_carryover(a), compute_carryover(b)
```

(continues on next page)

(continued from previous page)

```
    init_state = (a, b, fa, fb, 0)
    final_state = lax.while_loop(cond_fn, body_fn, init_state)
    a_final, b_final, _, _, _ = final_state

    return (a_final + b_final) / 2.0
```

The top-level wrapper `find_transition_exo` sets up the compensation parameters and calls the bisection solver, then recomputes the full transition path at the equilibrium tax rate.

```
def find_transition_exo(price_seq, policy_seq_base, ss0, ss1,
                        hh, tech, S, S1, S2,
                        compensation_data=None,
                        τl_bounds=(0.01, 0.6)):
    """Find transition tax rate under exogenous prices."""

    policy_seq = jnp.asarray(policy_seq_base)
    price_seq = jnp.asarray(price_seq)
    bounds = jnp.array([τl_bounds[0], τl_bounds[1]])
    s_indices = jnp.arange(S + 2)

    # Set up compensation parameters (default zeros when not using compensation)
    if compensation_data is not None:
        benef_diff, ss0_τ_l = compensation_data
        comp_mult = 1.0
    else:
        benef_diff = jnp.zeros(hh.T0 + 1)
        ss0_τ_l = ss0.τ_l
        comp_mult = 0.0

    # Find transition tax using unified function
    τ_l_trans = _find_transition_tau_l(
        price_seq, policy_seq, bounds,
        ss1.P_arr, ss1.ξ_arr, float(ss1.Gb),
        ss0.μx_arr, ss0.Σx_arr, float(ss0.k_bar),
        benef_diff, float(ss0_τ_l), comp_mult,
        s_indices, AGE_INDICES,
        S1, S2,
        hh.ε_arr, hh.β_arr, hh.Ind_work,
        hh.ρ_d, hh.σ, hh.B, hh.C,
        hh.R, hh.Q, hh.H,
        hh.P_end, hh.ξ_end, hh.x0, hh.Σ0,
        hh.frac, hh.n
    )
    τ_l_trans = float(τ_l_trans)

    # Compute final results with initial conditions
    mask = (s_indices >= S1 + 1) & (s_indices <= S2)
    τ_l_col = jnp.where(
        mask, τ_l_trans, policy_seq[:, 0]
    )
    policy_seq_final = policy_seq.at[:, 0].set(
        τ_l_col
    )

    μx_init, Σx_init = apply_compensation(
        ss0.μx_arr, ss0.Σx_arr,
        policy_seq_final[:, 0],
```

(continues on next page)

(continued from previous page)

```
        policy_seq_final[:, 1],
        benef_diff,
        price_seq[:, 0], price_seq[:, 1],
        hh.ε_arr, float(ss0_τ_l),
        AGE_INDICES, AGE_INDICES[:-1],
        hh.x0, hh.Σ0, comp_mult
    )

    results = transition_paths(
        τ_l_trans, price_seq, policy_seq,
        ss0, ss1, hh, tech, S, S1, S2,
        μx_init, Σx_init
    )

    return τ_l_trans, results
```

Bisection over the transition tax rate produces the equilibrium path.

```
τ_l_trans, results = find_transition_exo(
    price_seq, policy_seq_base, ss0, ss1,
    hh, tech, S_exo, S1, S2)

carryover, μx_seq, μc_seq, k_seq, Gb_seq, F_seq, Ao_seq = results
```

In the baseline case (no compensation), social security benefits are simply terminated.

```
τ_l_seq = np.zeros(S_exo + 1)
τ_l_seq[:S1 + 1] = ss0.τ_l
τ_l_seq[S1 + 1:S2 + 1] = τ_l_trans
τ_l_seq[S2 + 1:] = ss1.τ_l

fig, axes = plt.subplots(1, 2, figsize=(12, 4))

axes[0].plot(τ_l_seq, 'b-', linewidth=2)
axes[0].axhline(ss0.τ_l, color='r', linestyle=':',
                label=f'Initial $\\tau_\\ell$ = {ss0.τ_l:.4f}')
axes[0].axhline(ss1.τ_l, color='g', linestyle=':',
                label=f'Terminal $\\tau_\\ell$ = {ss1.τ_l:.4f}')
axes[0].axvspan(S1, S2, alpha=0.1, color='yellow')
axes[0].set_xlabel('Time')
axes[0].set_ylabel('Labor tax rate')
axes[0].set_title('Labor tax rate path')
axes[0].legend()


axes[1].plot(Gb_seq, 'b-', linewidth=2)
axes[1].axhline(ss0.Gb, color='r', linestyle=':',
                label=f'Initial $G_b$ = {ss0.Gb:.2f}')
axes[1].axhline(ss1.Gb, color='g', linestyle=':',
                label=f'Terminal $G_b$ = {ss1.Gb:.2f}')
axes[1].axvspan(S1, S2, alpha=0.1, color='yellow')
axes[1].set_xlabel('Time')
axes[1].set_ylabel('Government debt')
axes[1].set_title('Government debt path')
axes[1].legend()
```

(continues on next page)

(continued from previous page)

```
plt.tight_layout()
plt.show()
```

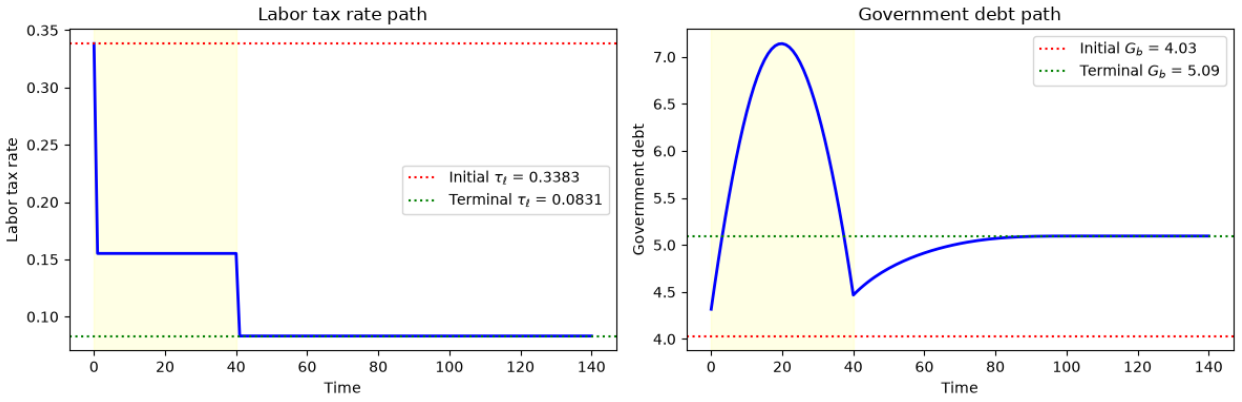


Fig. 116.6: Baseline transition path (no compensation)

The left panel shows the labor tax rate dropping during the transition because the government no longer needs to fund social security benefits.

The right panel shows government debt converging from its initial level to the new steady-state level, with the shaded region marking the transition period $[s_1, s_2)$.

116.8 Experiment 1: compensation through debt

In this experiment, the government terminates social security benefits but compensates affected generations.

Each cohort receives a transfer equal to the present value of the benefits they would have received – an actuarially fair buy-out.

Algorithm 116.8.1 (Fixed factor prices – Experiment 1 (buy-out))

Here $s_3 = s_2 + T_0 = 105$.

Step 1. Set up parameters.

Step 2. Solve the initial stationary equilibrium with constant social security benefit S: fix $\tau_a, \tau_0, S, G, \bar{b}$ and solve for $\tau_\ell = \tau_{\ell,0}$ such that the government budget balances:

```
F(τ_ℓ) = government budget imbalance
Find root of F(τ_ℓ) = 0.
```

Step 3. Solve the terminal stationary equilibrium with no social security: search over $\bar{b}$ so that the debt-to-GDP ratio matches a target:

```
H(Gb) = debt-to-GDP given Gb
       (internally solves F(τ_ℓ; Gb) = 0)
Find root of H(Gb) = target.
```

The associated labor tax is $\tau_{\ell,2}$.

Step 4. Solve the transition path: at $s = 0 = s_1$, all cohorts alive lose benefits and a cohort of age t receives a one-time compensation equal to the present value of lost benefits, discounted at the after-tax return $\tilde{R}(s) = R(s)[1-\tau_a(s)]+\tau_a(s)$:

$$\text{comp}_t = S \sum_{j=\max(T_1-t,\,0)}^{T_0-t} \prod_{i=0}^{j} \tilde{R}(s+i)^{-1}.$$

The government sets $\tau_{\ell,1}$ during $[s_1, s_2)$ and $\tau_{\ell,2}$ from s_2 onwards, with a one-time expenditure increase of $\sum f_t \,\text{comp}_t$ at s_1.

Find $\tau_{\ell,1}$ such that terminal government debt matches the target:

```
J(τ_ℓ) = terminal debt carryover
Find root of J(τ_ℓ) = Gb_terminal.
```

116.8.1 Fixed prices

We hold factor prices fixed and construct the price and policy sequences for the transition.

The benefit difference vector `benef_diff_exp1` records the per-period benefit loss for each age: retirees lose their old-regime benefits, while workers are unaffected.

```
ss1_exp1_exo = ss1

price_seq_exp1_exo = make_exo_price_seq(S_exo, RR_exo, w_exo)
policy_seq_exp1_exo = make_policy_seq(S_exo, ss0.τ_l, ss1.τ_l, S1, S2,
                                      τ_a_0, τ_0_0, 0, G_0, Gb_0)

benef_diff_exp1 = jnp.zeros(hh.T0 + 1)
benef_diff_exp1 = benef_diff_exp1.at[hh.T1 + 1:].set(ss0.benef)
```

The function `buyout_compensation_exp1_exo` computes the present-value compensation for each cohort alive at the reform date and adds it to their initial assets.

We then solve for the transition tax rate with and without the buy-out, so that we can compare the two paths.

```
def buyout_compensation_exp1_exo(τ_l_trans, policy_seq_base, price_seq):
    """Compute buy-out compensation under exogenous prices."""

    policy_seq = policy_seq_base.copy()
    policy_seq[S1 + 1:S2 + 1, 0] = τ_l_trans
    return apply_compensation(
        ss0.μx_arr, ss0.Σx_arr,
        policy_seq[:, 0], policy_seq[:, 1], benef_diff_exp1,
        price_seq[:, 0], price_seq[:, 1], hh.ε_arr, ss0.τ_l,
        AGE_INDICES, AGE_INDICES[:-1],
        hh.x0, hh.Σ0,
        1.0  # comp_mult = 1.0 for full compensation
    )

# Solve with buyout
τ_l_exp1_exo_bo, results_exp1_exo_bo = find_transition_exo(
    price_seq_exp1_exo, policy_seq_exp1_exo, ss0, ss1_exp1_exo,
    hh, tech, S_exo, S1, S2,
    compensation_data=(benef_diff_exp1, ss0.τ_l)
```

(continues on next page)

(continued from previous page)

```
)

# Solve without buyout (for comparison)
τ_l_exp1_exo_nb, results_exp1_exo_nb = find_transition_exo(
    price_seq_exp1_exo, policy_seq_exp1_exo, ss0, ss1_exp1_exo,
    hh, tech, S_exo, S1, S2
)
```

We compare the transition paths with and without buy-out compensation.

```
exp1_exo = {
    'ss0': ss0, 'ss1': ss1_exp1_exo,
    'τ_l_buyout': τ_l_exp1_exo_bo, 'τ_l_no_buyout': τ_l_exp1_exo_nb,
    'results_buyout': results_exp1_exo_bo,
    'results_no_buyout': results_exp1_exo_nb,
    'hh': hh, 'tech': tech
}
```

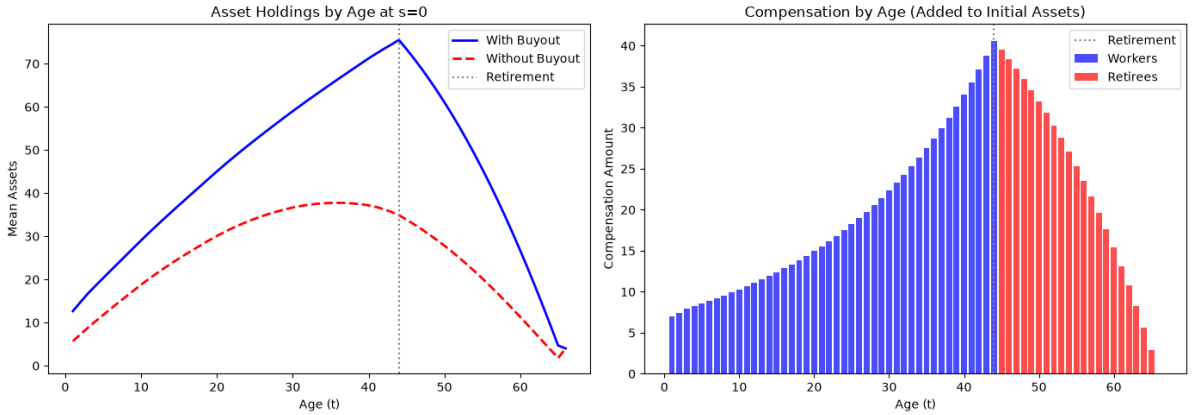
The following figure shows how the buy-out reshapes initial asset holdings across cohorts.

```
# Extract results
_, μx_seq_bo, μc_seq_bo, k_seq_bo, Gb_seq_bo, _, _ = exp1_exo['results_buyout']
results_nb = exp1_exo['results_no_buyout']
_, μx_seq_nb, μc_seq_nb, k_seq_nb, Gb_seq_nb, _, _ = results_nb

# Mean assets by age at time s=0 (with vs without buyout)
μa_bo = μx_seq_bo[0, 1:, 0]  # Assets at s=0 with buyout
μa_nb = μx_seq_nb[0, 1:, 0]  # Assets at s=0 without buyout

# Compensation = difference in initial assets
compensation_by_age = μa_bo - μa_nb

fig, axes = plt.subplots(1, 2, figsize=(14, 5))

# Asset profiles
ages = np.arange(1, hh.T0 + 2)
axes[0].plot(ages, μa_bo, 'b-', linewidth=2, label='With Buyout')
axes[0].plot(ages, μa_nb, 'r--', linewidth=2, label='Without Buyout')
axes[0].axvline(hh.T1 + 1, color='gray', linestyle=':', label='Retirement')
axes[0].set_xlabel('Age (t)')
axes[0].set_ylabel('Mean Assets')
axes[0].set_title('Asset Holdings by Age at s=0')
axes[0].legend()


# Compensation histogram
working_ages = ages[ages <= hh.T1 + 1]
retired_ages = ages[ages > hh.T1 + 1]
comp_working = compensation_by_age[:hh.T1 + 1]
comp_retired = compensation_by_age[hh.T1 + 1:]

axes[1].bar(working_ages, comp_working,
            color='blue', alpha=0.7, label='Workers')
axes[1].bar(retired_ages, comp_retired,
            color='red', alpha=0.7, label='Retirees')
axes[1].axhline(0, color='k', linewidth=0.5)
axes[1].axvline(hh.T1 + 1, color='gray', linestyle=':', label='Retirement')
```

(continues on next page)

(continued from previous page)

```
axes[1].set_xlabel('Age (t)')
axes[1].set_ylabel('Compensation Amount')
axes[1].set_title('Compensation by Age (Added to Initial Assets)')
axes[1].legend()


plt.tight_layout()
plt.show()
```



Retirees receive the largest compensation because they were expecting benefits for the remainder of their lives.

Older workers receive significant compensation, while young workers receive little because they have their entire working lives to adjust.

The declining profile among retirees reflects the actuarial calculation: older retirees have fewer remaining years of expected benefits.

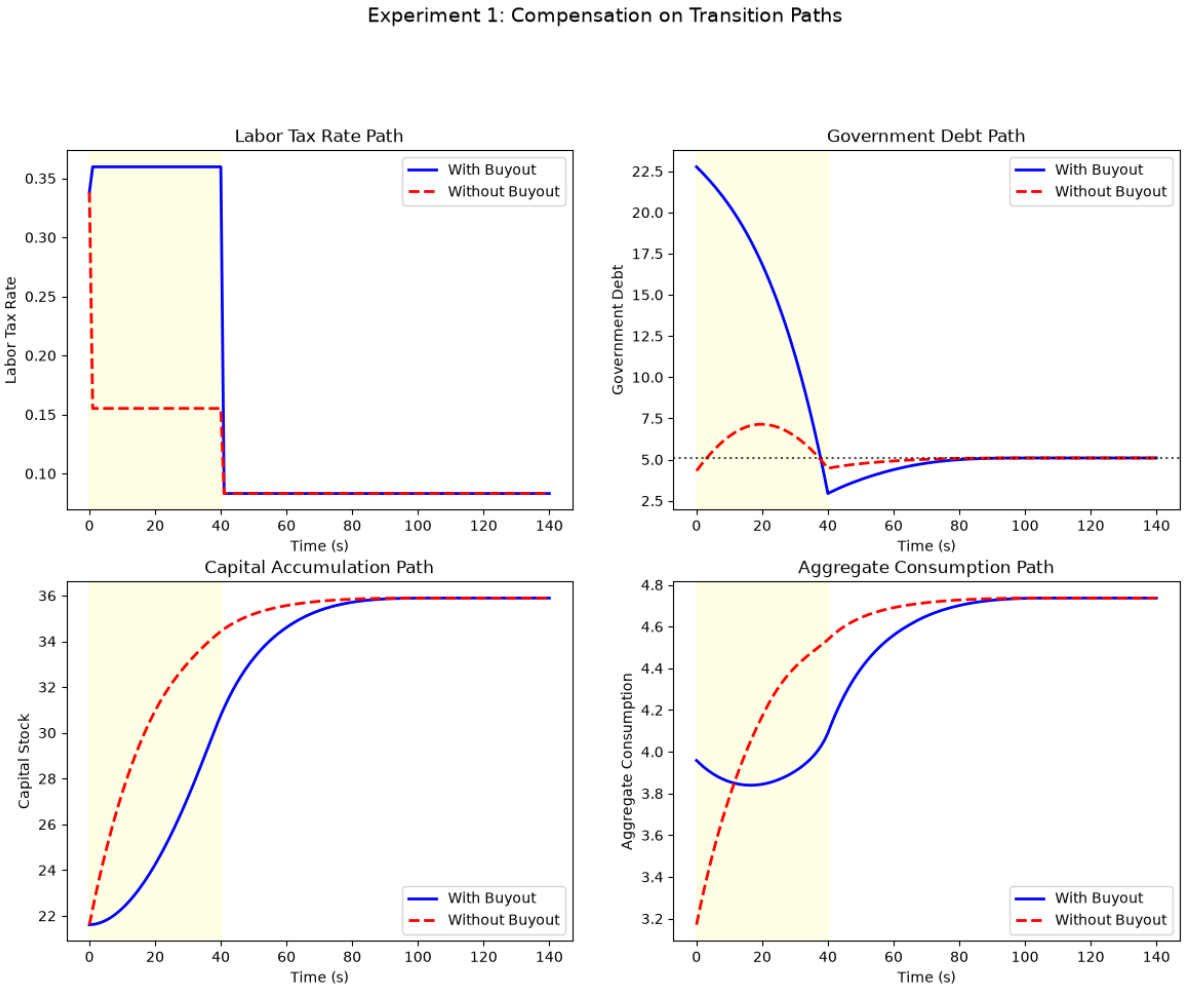
We now plot the aggregate transition paths for the labor tax, government debt, capital, and consumption under both schemes.

```
# hh, tech, ss0, ss1 already in scope -- just alias from dict for readability
ss0_exp1 = exp1_exo['ss0']
ss1_exp1 = exp1_exo['ss1']

# Construct τ_l sequences
τ_l_seq_bo = np.zeros(S_exo + 1)
τ_l_seq_bo[:S1 + 1] = ss0_exp1.τ_l
τ_l_seq_bo[S1 + 1:S2 + 1] = exp1_exo['τ_l_buyout']
τ_l_seq_bo[S2 + 1:] = ss1_exp1.τ_l

τ_l_seq_nb = np.zeros(S_exo + 1)
τ_l_seq_nb[:S1 + 1] = ss0_exp1.τ_l
τ_l_seq_nb[S1 + 1:S2 + 1] = exp1_exo['τ_l_no_buyout']
τ_l_seq_nb[S2 + 1:] = ss1_exp1.τ_l

fig, axes = plt.subplots(2, 2, figsize=(14, 10))

# τ_l comparison
axes[0, 0].plot(τ_l_seq_bo, 'b-', linewidth=2, label='With Buyout')
axes[0, 0].plot(τ_l_seq_nb, 'r--', linewidth=2, label='Without Buyout')
axes[0, 0].axvspan(S1, S2, alpha=0.1, color='yellow')
axes[0, 0].set_xlabel('Time (s)')
```

(continues on next page)

(continued from previous page)

```
axes[0, 0].set_ylabel('Labor Tax Rate')
axes[0, 0].set_title('Labor Tax Rate Path')
axes[0, 0].legend()

# Gb comparison
axes[0, 1].plot(Gb_seq_bo, 'b-', linewidth=2, label='With Buyout')
axes[0, 1].plot(Gb_seq_nb, 'r--', linewidth=2, label='Without Buyout')
axes[0, 1].axhline(ss1_exp1.Gb, color='k', linestyle=':', alpha=0.7)
axes[0, 1].axvspan(S1, S2, alpha=0.1, color='yellow')
axes[0, 1].set_xlabel('Time (s)')
axes[0, 1].set_ylabel('Government Debt')
axes[0, 1].set_title('Government Debt Path')
axes[0, 1].legend()

# Capital path
axes[1, 0].plot(k_seq_bo, 'b-', linewidth=2, label='With Buyout')
axes[1, 0].plot(k_seq_nb, 'r--', linewidth=2, label='Without Buyout')
axes[1, 0].axvspan(S1, S2, alpha=0.1, color='yellow')
axes[1, 0].set_xlabel('Time (s)')
axes[1, 0].set_ylabel('Capital Stock')
axes[1, 0].set_title('Capital Accumulation Path')
axes[1, 0].legend()

# Aggregate consumption
c_agg_bo = np.array(μc_seq_bo[:S_exo + 1]) @ np.array(hh.frac)
c_agg_nb = np.array(μc_seq_nb[:S_exo + 1]) @ np.array(hh.frac)

axes[1, 1].plot(c_agg_bo, 'b-', linewidth=2, label='With Buyout')
axes[1, 1].plot(c_agg_nb, 'r--', linewidth=2, label='Without Buyout')
axes[1, 1].axvspan(S1, S2, alpha=0.1, color='yellow')
axes[1, 1].set_xlabel('Time (s)')
axes[1, 1].set_ylabel('Aggregate Consumption')
axes[1, 1].set_title('Aggregate Consumption Path')
axes[1, 1].legend()

plt.suptitle(
    'Experiment 1: Compensation on Transition Paths',
    fontsize=14, y=1.02
)
plt.show()
```



The buy-out scheme leads to a slower initial rise in private capital because the government must make large transfers.

Both schemes converge to the same terminal steady state.

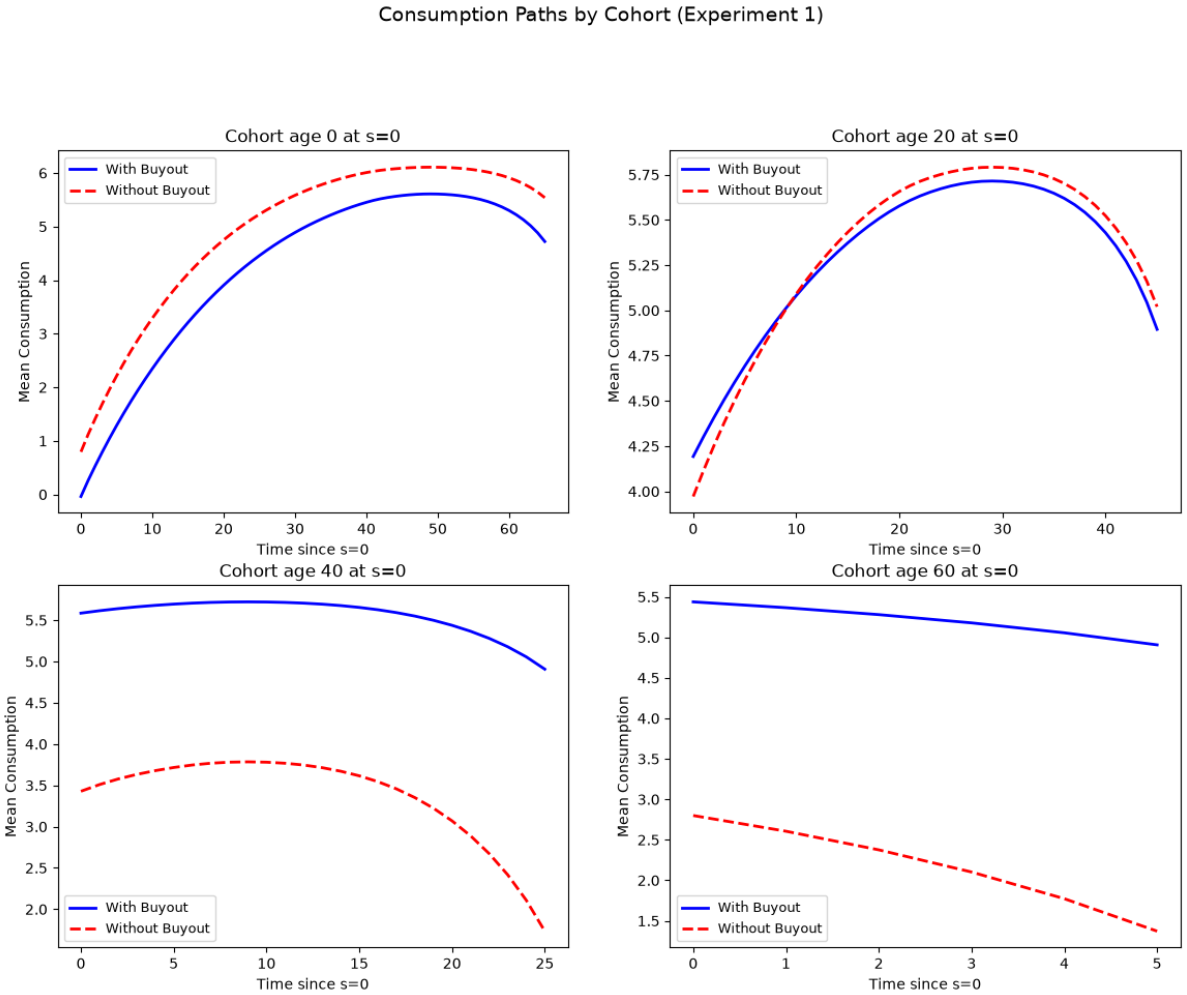
We now examine consumption paths for cohorts at different ages when the reform occurs.

```
fig, axes = plt.subplots(2, 2, figsize=(14, 10))

selected_ages = [0, 20, 40, 60]  # Cohorts at different ages at s=0

for idx, age_at_0 in enumerate(selected_ages):
    ax = axes[idx // 2, idx % 2]

    remaining_life = hh.T0 - age_at_0
    max_time = min(remaining_life + 1, S_exo + 1)

    c_bo = [μc_seq_bo[s, age_at_0 + s]
            for s in range(max_time)
            if age_at_0 + s <= hh.T0]
    c_nb = [μc_seq_nb[s, age_at_0 + s]
            for s in range(max_time)
            if age_at_0 + s <= hh.T0]

    ax.plot(c_bo, 'b-', linewidth=2, label='With Buyout')
```

(continues on next page)

(continued from previous page)

```
    ax.plot(c_nb, 'r--', linewidth=2, label='Without Buyout')
    ax.set_xlabel('Time since s=0')
    ax.set_ylabel('Mean Consumption')
    ax.set_title(f'Cohort age {age_at_0} at s=0')
    ax.legend(fontsize=9)


plt.suptitle(
    'Consumption Paths by Cohort (Experiment 1)',
    fontsize=14, y=1.02
)
plt.show()
```



Young workers (age 0) show nearly identical consumption paths because they receive little compensation and have decades to adjust their savings.

Near-retirement workers (age 40) exhibit more noticeable differences as the buyout compensation partially offsets their lost benefits.

Retirees (age 60) show the most dramatic difference: without compensation, their consumption drops sharply when benefits end, whereas the buyout scheme maintains higher consumption by replacing the lost income.

116.8.2 Endogenous prices

With endogenous factor prices, changes in saving behavior affect capital accumulation, which alters marginal products and feeds back into household decisions.

Algorithm 116.8.2 (Endogenous factor prices)

Here $s_3 = s_2 + 2T_0 = 170$.

Steps 1–3 are the same as *Algorithm 116.8.1*, with an outer fixed-point loop over factor prices to clear factor markets in steady state.

Step 4. The factor price sequences are now endogenous.

Wrap the fixed-price transition solver in a relaxation loop:

```
T(R_seq):
  1. Compute the wage sequence from R_seq via Cobb-Douglas.
  2. Taking prices as given, find root of J(τ_ℓ) = Gb_terminal.
  3. Compute the implied capital path, then the implied R*.
  Return R* - R_seq.

Iterate T(R_seq) = 0 with relaxation until convergence.
```

We iterate on the price sequence until factor markets clear.

```
@jit
def compute_factor_prices(k_prod, ε_bar, A, α, δ):
    """Compute factor prices from Cobb-Douglas."""

    k_per_eff = k_prod / ε_bar
    r = A * α * (k_per_eff ** (α - 1))
    w = A * (1 - α) * (k_per_eff ** α)
    RR = 1 + r - δ
    return r, w, RR
```

With endogenous prices, finding a steady state requires an outer fixed-point loop: solve the household problem at given prices, compute the implied capital stock, update prices via Cobb-Douglas marginal products, and repeat until convergence.

```
def find_ss_endo(
        debt2gdp_target, policy_target,
        hh, tech, RR_init=None, w_init=None,
        max_iter=50, tol=1e-5, verbose=False):
    """Find steady state with endogenous factor prices."""

    τ_a, τ_0, benef, G = policy_target
    ε_bar = float(jnp.sum(hh.frac * hh.ε_arr))

    RR = RR_init if RR_init else tech.RR
    w = w_init if w_init else tech.w

    relaxation = 0.3

    for iteration in range(max_iter):
        try:
            ss = ss_target_debt2gdp_exo(
```

(continues on next page)

(continued from previous page)

```
                debt2gdp_target,
                (τ_a, τ_0, benef, G),
                (RR, w), hh, tech
            )
        except ValueError:
            RR = RR * 0.99
            continue

        K_eff = ss.k_bar / hh.n + float(hh.frac[0] * hh.x0[0])
        r_new, w_new, RR_new = compute_factor_prices(
            K_eff, ε_bar,
            tech.A, tech.α_tilde, tech.δ
        )
        r_new = float(r_new)
        w_new = float(w_new)
        RR_new = float(RR_new)
        price_diff = abs(RR_new - RR) + abs(w_new - w)

        if verbose and iteration % 5 == 0:
            print(f"    SS iter {iteration}: "
                  f"RR={RR:.6f}, w={w:.4f}, "
                  f"k_bar={ss.k_bar:.4f}")

        if price_diff < tol:
            if verbose:
                print(f"    Converged at iteration {iteration}")
            break

        RR = RR + relaxation * (RR_new - RR)
        w = w + relaxation * (w_new - w)

    return ss_target_debt2gdp_exo(
        debt2gdp_target,
        (τ_a, τ_0, benef, G),
        (RR, w), hh, tech
    )
```

The price iteration also needs an initial guess for the transition price path and a way to update it after each inner solve.

The function `init_price_seq_interp` linearly interpolates between the two steady-state price vectors, while `_update_prices_from_capital` recomputes factor prices from the capital path via Cobb-Douglas marginal products.

```
def init_price_seq_interp(S, S1, S3, ss0_RR, ss0_w, ss1_RR, ss1_w):
    """Linearly interpolate price sequence between steady states."""

    s_indices = jnp.arange(S + 2)
    t_frac = jnp.clip((s_indices - S1) / (S3 - S1), 0.0, 1.0)

    RR_seq = ss0_RR + t_frac * (ss1_RR - ss0_RR)
    w_seq = ss0_w + t_frac * (ss1_w - ss0_w)

    RR_seq = jnp.where(s_indices <= S1, ss0_RR, RR_seq)
    w_seq = jnp.where(s_indices <= S1, ss0_w, w_seq)
    RR_seq = jnp.where(s_indices >= S3, ss1_RR, RR_seq)
    w_seq = jnp.where(s_indices >= S3, ss1_w, w_seq)
```

(continues on next page)

(continued from previous page)

```
    return jnp.column_stack([RR_seq, w_seq])


@jit
def _update_prices_from_capital(
        k_seq, k_bar_ss0, n, frac0_x0,
        ε_bar, A, α, δ,
        s_indices_full, ss1_RR, ss1_w, S3):
    """Compute new price sequence from the capital path."""

    k_prev = jnp.concatenate([jnp.array([k_bar_ss0]), k_seq[:-1]])
    K_eff = k_prev / n + frac0_x0

    k_per_eff = K_eff / ε_bar
    r_new = A * α * (k_per_eff ** (α - 1))
    w_new = A * (1 - α) * (k_per_eff ** α)
    RR_new = 1 + r_new - δ

    price_seq_new = jnp.column_stack([RR_new, w_new])
    price_seq_new = jnp.concatenate([price_seq_new, price_seq_new[-1:]], axis=0)

    terminal_prices = jnp.array([[ss1_RR, ss1_w]])
    mask = s_indices_full >= S3
    price_seq_new = jnp.where(mask[:, None], terminal_prices, price_seq_new)

    return price_seq_new
```

The top-level function `find_transition_endo` wraps everything in a relaxation loop: at each iteration it solves the transition under current prices, computes the implied capital path, updates prices, and checks for convergence.

```
def find_transition_endo(price_seq, policy_seq_base,
                         ss0, ss1, hh, tech, S, S1, S2, S3,
                         compensation_data=None,
                         max_iter=50, tol=1e-3,
                         relaxation=0.5, verbose=False):
    """Find transition with endogenous prices."""

    ε_bar = float(jnp.sum(hh.frac * hh.ε_arr))
    frac0_x0 = float(hh.frac[0] * hh.x0[0])

    price_seq = jnp.asarray(price_seq)
    policy_seq_base = jnp.asarray(policy_seq_base)

    # Pre-create iteration arrays for price update
    s_indices_full = jnp.arange(S + 2)

    if verbose:
        print("  Starting price iteration...")

    for iteration in range(max_iter):
        try:
            τ_l_trans, results = find_transition_exo(
                price_seq, policy_seq_base, ss0, ss1,
                hh, tech, S, S1, S2,
                compensation_data=compensation_data
            )
        except ValueError:
```

(continues on next page)

(continued from previous page)

```
            τ_l_trans = 0.35
            results = transition_paths(
                τ_l_trans, price_seq, policy_seq_base,
                ss0, ss1, hh, tech, S, S1, S2,
                ss0.μx_arr, ss0.Σx_arr
            )

        _, μx_seq, μc_seq, k_seq, Gb_seq, F_seq, Ao_seq = results

        price_seq_new = _update_prices_from_capital(
            k_seq, float(ss0.k_bar), hh.n, float(frac0_x0), float(ε_bar),
            tech.A, tech.α_tilde, tech.δ,
            s_indices_full,
            float(ss1.RR), float(ss1.w), S3
        )

        price_diff = float(jnp.max(jnp.abs(price_seq_new - price_seq)))

        if verbose:
            print(f"  Iter {iteration}: "
                  f"τ_l={τ_l_trans:.4f}, "
                  f"price_diff={price_diff:.6f}")

        if price_diff < tol:
            if verbose:
                print(f"  Converged at iteration {iteration}")
            break

        price_seq = price_seq + relaxation * (price_seq_new - price_seq)

    return τ_l_trans, price_seq, results
```

We now compute the initial and terminal steady states under endogenous prices and solve the transition with price iteration.

```
# Compute endogenous prices for initial SS
ε_bar = float(jnp.sum(hh.frac * hh.ε_arr))
K_eff_0 = ss0.k_bar / hh.n + float(hh.frac[0] * hh.x0[0])
r0_endo, w0_endo, RR0_endo = compute_factor_prices(
    K_eff_0, ε_bar,
    tech.A, tech.α_tilde, tech.δ
)
r0_endo = float(r0_endo)
w0_endo = float(w0_endo)
RR0_endo = float(RR0_endo)

ss0_exp1_endo = SteadyState(
    P_arr=ss0.P_arr, ξ_arr=ss0.ξ_arr, Ao_arr=ss0.Ao_arr, F_arr=ss0.F_arr,
    μx_arr=ss0.μx_arr, μc_arr=ss0.μc_arr, Σx_arr=ss0.Σx_arr, Vc_arr=ss0.Vc_arr,
    debt2gdp=ss0.debt2gdp, τ_l=ss0.τ_l, benef=ss0.benef, Gb=ss0.Gb,
    k_bar=ss0.k_bar, RR=RR0_endo, w=w0_endo, r=r0_endo, k2gdp=ss0.k2gdp
)

ss1_exp1_endo = find_ss_endo(
    ss0.debt2gdp, (τ_a_0, τ_0_0, 0, G_0), hh, tech,
    RR_init=tech.RR, w_init=tech.w, verbose=True
)
```

```
    SS iter 0: RR=1.067500, w=5.0147, k_bar=35.8906
    SS iter 5: RR=1.044957, w=5.7366, k_bar=30.4338
    SS iter 10: RR=1.044407, w=5.7330, k_bar=30.1611
    SS iter 15: RR=1.044405, w=5.7291, k_bar=30.1587
    SS iter 20: RR=1.044406, w=5.7284, k_bar=30.1588
    SS iter 25: RR=1.044406, w=5.7283, k_bar=30.1588
    Converged at iteration 28
```

The initial price guess interpolates linearly between the two steady states.

```
# Initialize price sequence
price_seq_exp1_endo = init_price_seq_interp(
    S_endo, S1, S3,
    float(ss0_exp1_endo.RR), float(ss0_exp1_endo.w),
    float(ss1_exp1_endo.RR), float(ss1_exp1_endo.w)
)

# Policy sequence
policy_seq_exp1_endo = make_policy_seq(
    S_endo,
    ss0_exp1_endo.τ_l, ss1_exp1_endo.τ_l,
    S1, S2,
    τ_a_0, τ_0_0, 0, G_0,
    ss0_exp1_endo.Gb
)
```

The benefit difference vector records the benefit loss at each age, and the price iteration finds the equilibrium transition path with buy-out compensation.

```
# Buyout compensation
benef_diff_exp1_endo = jnp.zeros(hh.T0 + 1)
benef_diff = ss0_exp1_endo.benef - ss1_exp1_endo.benef
benef_diff_exp1_endo = benef_diff_exp1_endo.at[
    hh.T1 + 1:
].set(benef_diff)

# Solve with price iteration
print("\n  Solving transition with endogenous prices...")
endo_result = find_transition_endo(
    price_seq_exp1_endo, policy_seq_exp1_endo,
    ss0_exp1_endo, ss1_exp1_endo,
    hh, tech, S_endo, S1, S2, S3,
    compensation_data=(
        benef_diff_exp1_endo,
        ss0_exp1_endo.τ_l
    ),
    verbose=True
)
τ_l_exp1_endo_bo = endo_result[0]
price_seq_exp1_endo_conv = endo_result[1]
results_exp1_endo = endo_result[2]
(_, μx_seq_exp1_endo, μc_seq_exp1_endo,
 k_seq_exp1_endo, Gb_seq_exp1_endo,
 _, _) = results_exp1_endo
```

```
  Solving transition with endogenous prices...
  Starting price iteration...
```

(continues on next page)

(continued from previous page)

```
  Iter 0: τ_l=0.3391, price_diff=0.490311
  Iter 1: τ_l=0.3549, price_diff=0.184939
  Iter 2: τ_l=0.3637, price_diff=0.064441
  Iter 3: τ_l=0.3676, price_diff=0.021013
  Iter 4: τ_l=0.3692, price_diff=0.006973
  Iter 5: τ_l=0.3698, price_diff=0.002371
  Iter 6: τ_l=0.3701, price_diff=0.000828
  Converged at iteration 6
```

The endogenous-price results are stored for comparison with the fixed-price case.

```
exp1_endo = {
    'ss0': ss0_exp1_endo, 'ss1': ss1_exp1_endo,
    'τ_l_buyout': τ_l_exp1_endo_bo,
    'price_seq': price_seq_exp1_endo_conv,
    'k_seq': k_seq_exp1_endo, 'Gb_seq': Gb_seq_exp1_endo,
    'results': results_exp1_endo,
    'μc_seq': μc_seq_exp1_endo, 'μx_seq': μx_seq_exp1_endo
}
```

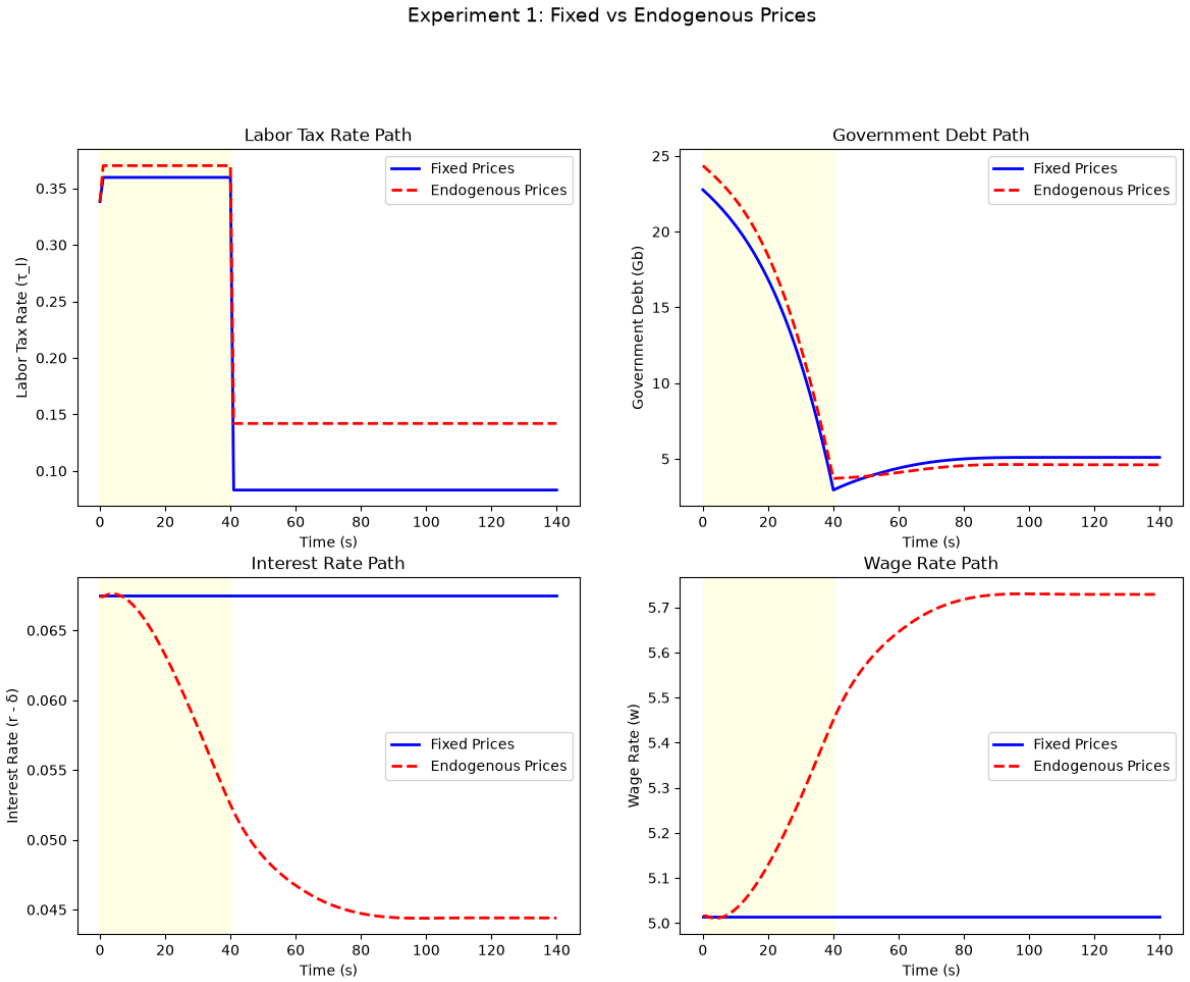
The following figure compares the transition paths under fixed and endogenous factor prices, showing how general equilibrium effects alter the tax, debt, interest rate, and wage paths.

```
# Get endogenous price sequences
price_seq_endo = exp1_endo['price_seq']
S_endo = price_seq_endo.shape[0] - 2

# Construct fixed price sequences for comparison
RR_fixed = tech.RR
w_fixed = tech.w

# For fixed prices, construct τ_l sequence
τ_l_seq_fixed = np.zeros(S_exo + 1)
τ_l_seq_fixed[:S1 + 1] = ss0_exp1.τ_l
τ_l_seq_fixed[S1 + 1:S2 + 1] = exp1_exo['τ_l_buyout']
τ_l_seq_fixed[S2 + 1:] = ss1_exp1.τ_l

# For endogenous prices
τ_l_seq_endo = np.zeros(S_endo + 1)
τ_l_seq_endo[:S1 + 1] = exp1_endo['ss0'].τ_l
τ_l_seq_endo[S1 + 1:S2 + 1] = exp1_endo['τ_l_buyout']
τ_l_seq_endo[S2 + 1:] = exp1_endo['ss1'].τ_l

fig, axes = plt.subplots(2, 2, figsize=(14, 10))

# Labor tax comparison
axes[0, 0].plot(τ_l_seq_fixed, 'b-', linewidth=2, label='Fixed Prices')
axes[0, 0].plot(τ_l_seq_endo[:len(τ_l_seq_fixed)],
                'r--', linewidth=2,
                label='Endogenous Prices')
axes[0, 0].axvspan(S1, S2, alpha=0.1, color='yellow')
axes[0, 0].set_xlabel('Time (s)')
axes[0, 0].set_ylabel('Labor Tax Rate (τ_l)')
axes[0, 0].set_title('Labor Tax Rate Path')
axes[0, 0].legend()

# Government debt comparison
```

(continues on next page)

(continued from previous page)

```
Gb_seq_fixed = Gb_seq_bo
Gb_seq_endo_exp1 = exp1_endo['Gb_seq']
axes[0, 1].plot(Gb_seq_fixed, 'b-', linewidth=2, label='Fixed Prices')
axes[0, 1].plot(
    Gb_seq_endo_exp1[:len(Gb_seq_fixed)],
    'r--', linewidth=2,
    label='Endogenous Prices'
)
axes[0, 1].axvspan(S1, S2, alpha=0.1, color='yellow')
axes[0, 1].set_xlabel('Time (s)')
axes[0, 1].set_ylabel('Government Debt (Gb)')
axes[0, 1].set_title('Government Debt Path')
axes[0, 1].legend()

# Interest rate comparison
r_fixed = np.full(S_exo + 1, tech.r - tech.δ)
r_endo = price_seq_endo[:-1, 0] - 1  # RR - 1 = r - δ
axes[1, 0].plot(r_fixed, 'b-', linewidth=2, label='Fixed Prices')
axes[1, 0].plot(
    r_endo[:len(r_fixed)],
    'r--', linewidth=2,
    label='Endogenous Prices'
)
axes[1, 0].axvspan(S1, S2, alpha=0.1, color='yellow')
axes[1, 0].set_xlabel('Time (s)')
axes[1, 0].set_ylabel('Interest Rate (r - δ)')
axes[1, 0].set_title('Interest Rate Path')
axes[1, 0].legend()

# Wage rate comparison
w_fixed_seq = np.full(S_exo + 1, tech.w)
w_endo = price_seq_endo[:-1, 1]
axes[1, 1].plot(w_fixed_seq, 'b-', linewidth=2, label='Fixed Prices')
axes[1, 1].plot(
    w_endo[:len(w_fixed_seq)],
    'r--', linewidth=2,
    label='Endogenous Prices'
)
axes[1, 1].axvspan(S1, S2, alpha=0.1, color='yellow')
axes[1, 1].set_xlabel('Time (s)')
axes[1, 1].set_ylabel('Wage Rate (w)')
axes[1, 1].set_title('Wage Rate Path')
axes[1, 1].legend()

plt.suptitle(
    'Experiment 1: Fixed vs Endogenous Prices',
    fontsize=14, y=1.02
)
plt.show()
```



The top-left panel shows that the transition tax rate is similar under both price assumptions, but the bottom panels reveal important differences in factor prices.

As the capital stock rises during the transition, the interest rate falls and the wage rises under endogenous pricing, while these remain constant under the small open economy assumption.

These price effects create additional redistributive consequences beyond those intended by the policy change: lower interest rates benefit young workers through higher wages, but hurt retirees through lower returns on savings.

116.9 Experiment 2: government capital accumulation

In Experiment 2, the government maintains social security benefits but temporarily raises taxes to accumulate physical capital.

The returns from this capital eventually finance the social security payments, so that in the terminal steady state the government is a net creditor rather than a debtor.

Unlike Experiment 1, which eliminates benefits, this approach preserves the social insurance function of social security – namely, insurance against life span risk and partial insurance against labor income volatility.

By having the government save on behalf of households, the economy can achieve higher capital accumulation while maintaining the intergenerational insurance that social security provides.

Algorithm 116.9.1 (Fixed factor prices – Experiment 2 (government funding))

Here $s_3 = s_2 + T_0 = 105$.

Steps 1–3 are the same as *Algorithm 116.8.1*, except that social security benefits are maintained and the target debt-to-GDP ratio is negative (the government becomes a net creditor).

The right target induces the government to accumulate sufficient capital to finance benefits from asset returns.

Step 4 is the same root-finding procedure over $\tau_{\ell,1}$, but without compensation payments.

We first compute the terminal steady state under fixed prices, targeting a negative debt-to-GDP ratio that makes the government a net creditor.

```
debt2gdp_target_exp2_exo = -1.1785
ss1_exp2_exo = ss_target_debt2gdp_exo(
    debt2gdp_target_exp2_exo,
    (τ_a_0, τ_0_0, benef_0, G_0),
    (RR_exo, w_exo), hh, tech
)
```

With the terminal steady state in hand, bisection over the transition tax rate gives the equilibrium path under fixed prices.

```
# Price and policy sequences
price_seq_exp2_exo = make_exo_price_seq(S_exo, RR_exo, w_exo)
policy_seq_exp2_exo = make_policy_seq(
    S_exo, ss0.τ_l, ss1_exp2_exo.τ_l,
    S1, S2,
    τ_a_0, τ_0_0, benef_0, G_0, Gb_0
)

# Solve (no compensation)
τ_l_exp2_exo, results_exp2_exo = find_transition_exo(
    price_seq_exp2_exo, policy_seq_exp2_exo, ss0, ss1_exp2_exo,
    hh, tech, S_exo, S1, S2
)
```

The results are packaged for the cross-experiment comparison below.

```
(carryover_exp2, μx_seq_exp2_exo,
 μc_seq_exp2_exo, k_seq_exp2_exo,
 Gb_seq_exp2_exo, F_seq_exp2_exo,
 Ao_seq_exp2_exo) = results_exp2_exo
exp2_exo = {
    'ss0': ss0, 'ss1': ss1_exp2_exo,
    'τ_l_trans': τ_l_exp2_exo,
    'results': results_exp2_exo,
    'k_seq': k_seq_exp2_exo, 'Gb_seq': Gb_seq_exp2_exo,
    'μc_seq': μc_seq_exp2_exo, 'μx_seq': μx_seq_exp2_exo
}
```

We repeat the computation under endogenous factor prices, using the same initial steady state as Experiment 1.

```
# Compute endogenous prices for initial SS (reuse from Exp 1)
ss0_exp2_endo = ss0_exp1_endo  # Same initial SS

# Terminal steady state with endogenous prices
```

(continues on next page)

(continued from previous page)

```
ss1_exp2_endo = find_ss_endo(
    debt2gdp_target=-1.925,
    policy_target=(τ_a_0, τ_0_0, benef_0, G_0),
    hh=hh, tech=tech,
    RR_init=tech.RR, w_init=tech.w,
    verbose=True
)
```

```
    SS iter 0: RR=1.067500, w=5.0147, k_bar=43.5570
    SS iter 5: RR=1.044472, w=5.7664, k_bar=30.0436
    SS iter 10: RR=1.044623, w=5.7272, k_bar=30.0407
    SS iter 15: RR=1.044648, w=5.7206, k_bar=30.0403
    SS iter 20: RR=1.044651, w=5.7195, k_bar=30.0409
    SS iter 25: RR=1.044651, w=5.7193, k_bar=30.0409
    Converged at iteration 29
```

Price iteration produces the endogenous-price transition path.

```
# Initialize price sequence
price_seq_exp2_endo = init_price_seq_interp(
    S_endo, S1, S3,
    float(ss0_exp2_endo.RR), float(ss0_exp2_endo.w),
    float(ss1_exp2_endo.RR), float(ss1_exp2_endo.w)
)

# Policy sequence
policy_seq_exp2_endo = make_policy_seq(
    S_endo,
    ss0_exp2_endo.τ_l, ss1_exp2_endo.τ_l,
    S1, S2,
    τ_a_0, τ_0_0, benef_0, G_0,
    ss0_exp2_endo.Gb
)

# Solve with price iteration (no compensation)
endo2 = find_transition_endo(
    price_seq_exp2_endo, policy_seq_exp2_endo,
    ss0_exp2_endo, ss1_exp2_endo,
    hh, tech, S_endo, S1, S2, S3,
    verbose=True
)
τ_l_exp2_endo = endo2[0]
price_seq_exp2_endo_conv = endo2[1]
results_exp2_endo = endo2[2]
(_, μx_seq_exp2_endo, μc_seq_exp2_endo,
 k_seq_exp2_endo, Gb_seq_exp2_endo,
 _, _) = results_exp2_endo
```

```
   Starting price iteration...
   Iter 0: τ_l=0.3660, price_diff=0.394320
   Iter 1: τ_l=0.3769, price_diff=0.202210
   Iter 2: τ_l=0.3845, price_diff=0.103709
   Iter 3: τ_l=0.3891, price_diff=0.050465
   Iter 4: τ_l=0.3916, price_diff=0.022852
   Iter 5: τ_l=0.3928, price_diff=0.009652
   Iter 6: τ_l=0.3933, price_diff=0.003839
```

(continues on next page)

(continued from previous page)

```
  Iter 7: τ_l=0.3935, price_diff=0.001457
  Iter 8: τ_l=0.3936, price_diff=0.000537
  Converged at iteration 8
```

```
exp2_endo = {
    'ss0': ss0_exp2_endo, 'ss1': ss1_exp2_endo,
    'τ_l_trans': τ_l_exp2_endo,
    'price_seq': price_seq_exp2_endo_conv,
    'k_seq': k_seq_exp2_endo, 'Gb_seq': Gb_seq_exp2_endo,
    'results': results_exp2_endo,
    'μc_seq': μc_seq_exp2_endo, 'μx_seq': μx_seq_exp2_endo
}
```

We now compare all four reform scenarios: the buy-out scheme and the government funding scheme, each under fixed and endogenous factor prices.

```
# Get debt sequences for all cases
Gb_buyout_fixed = Gb_seq_bo
Gb_buyout_endo = exp1_endo['Gb_seq']
Gb_accum_fixed = exp2_exo['Gb_seq']
Gb_accum_endo = exp2_endo['Gb_seq']

# Get capital sequences
k_buyout_fixed = k_seq_bo
k_buyout_endo = exp1_endo['k_seq']
k_accum_fixed = exp2_exo['k_seq']
k_accum_endo = exp2_endo['k_seq']

# Common time horizon for plotting
T_plot = min(len(Gb_buyout_fixed), len(Gb_buyout_endo),
             len(Gb_accum_fixed), len(Gb_accum_endo))

fig, axes = plt.subplots(2, 3, figsize=(18, 10))

# Labels for four scenarios
lb = ['Buyout (Fixed)', 'Buyout (Endo)',
      'Gov Funding (Fixed)', 'Gov Funding (Endo)']
ls = ['b-', 'b--', 'r-', 'r--']

# Government Debt paths
ax = axes[0, 0]
for d, s, l in zip(
    [Gb_buyout_fixed, Gb_buyout_endo,
     Gb_accum_fixed, Gb_accum_endo],
    ls, lb
):
    ax.plot(d[:T_plot], s, linewidth=2, label=l)
ax.axhline(0, color='k', linestyle=':', alpha=0.5)
ax.axvspan(0, 40, alpha=0.1, color='yellow')
ax.set_xlabel('Time (s)')
ax.set_ylabel('Government Debt (Gb)')
ax.set_title('Government Debt Paths')
ax.legend(fontsize=9)


# Capital paths
```

(continues on next page)

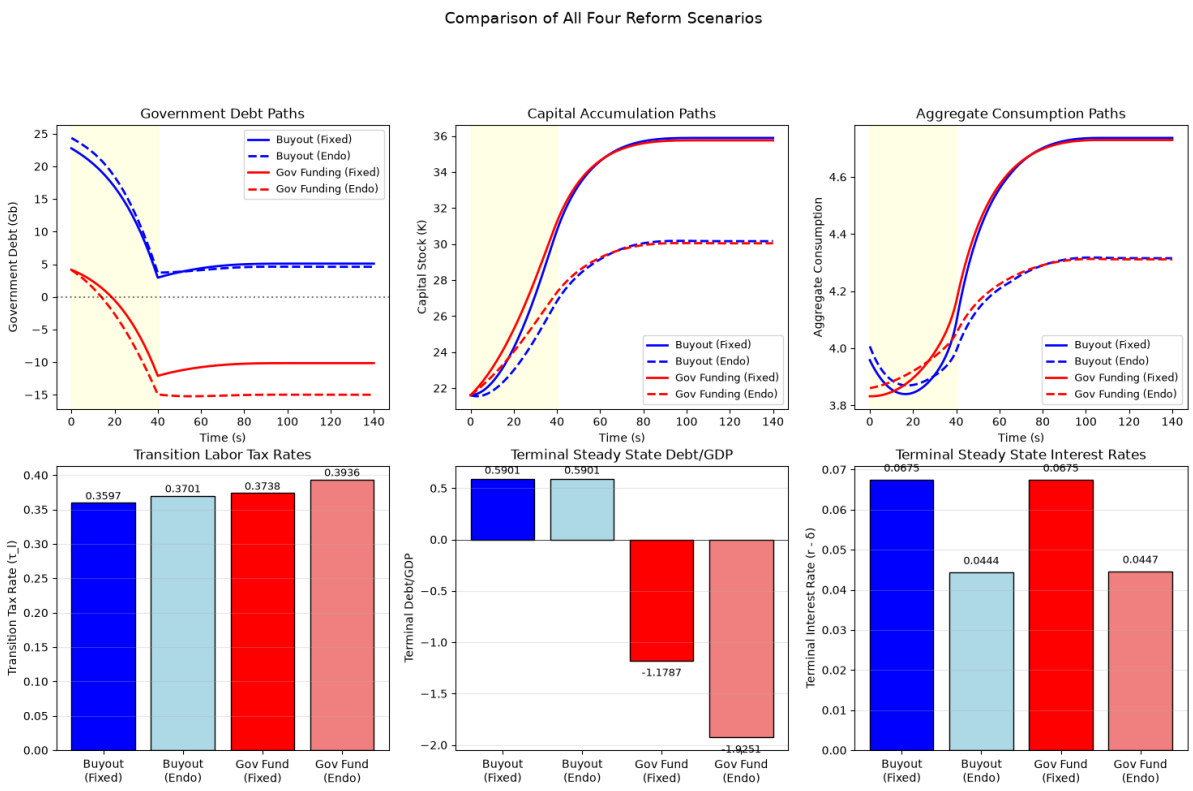
(continued from previous page)

```
ax = axes[0, 1]
for d, s, l in zip(
    [k_buyout_fixed, k_buyout_endo,
     k_accum_fixed, k_accum_endo],
    ls, lb
):
    ax.plot(d[:T_plot], s, linewidth=2, label=l)
ax.axvspan(0, 40, alpha=0.1, color='yellow')
ax.set_xlabel('Time (s)')
ax.set_ylabel('Capital Stock (K)')
ax.set_title('Capital Accumulation Paths')
ax.legend(fontsize=9)


# Aggregate consumption
res2_exo = exp2_exo['results']
res1_endo = exp1_endo['results']
res2_endo = exp2_endo['results']
_, _, μc_exp2_exo, _, _, _, _ = res2_exo
_, _, μc_exp1_endo, _, _, _, _ = res1_endo
_, _, μc_exp2_endo, _, _, _, _ = res2_endo

def _agg_c(μc, T):
    """Aggregate per-capita consumption across cohorts."""

    n = min(T, μc.shape[0])
    return np.array([
        np.sum(μc[s] * hh.frac) for s in range(n)
    ])

c_agg_buyout_fixed = c_agg_bo[:T_plot]
c_agg_buyout_endo = _agg_c(μc_exp1_endo, T_plot)
c_agg_accum_fixed = _agg_c(μc_exp2_exo, T_plot)
c_agg_accum_endo = _agg_c(μc_exp2_endo, T_plot)

ax = axes[0, 2]
for d, s, l in zip(
    [c_agg_buyout_fixed, c_agg_buyout_endo,
     c_agg_accum_fixed, c_agg_accum_endo],
    ls, lb
):
    ax.plot(d[:T_plot], s, linewidth=2, label=l)
axes[0, 2].axvspan(0, 40, alpha=0.1, color='yellow')
axes[0, 2].set_xlabel('Time (s)')
axes[0, 2].set_ylabel('Aggregate Consumption')
axes[0, 2].set_title('Aggregate Consumption Paths')
axes[0, 2].legend(fontsize=9)

# Bar chart: Transition tax rates
cases = ['Buyout\n(Fixed)', 'Buyout\n(Endo)',
         'Gov Fund\n(Fixed)', 'Gov Fund\n(Endo)']
τ_l_values = [exp1_exo['τ_l_buyout'], exp1_endo['τ_l_buyout'],
              exp2_exo['τ_l_trans'], exp2_endo['τ_l_trans']]
colors = ['blue', 'lightblue', 'red', 'lightcoral']
axes[1, 0].bar(cases, τ_l_values, color=colors, edgecolor='black')
axes[1, 0].set_ylabel('Transition Tax Rate (τ_l)')
axes[1, 0].set_title('Transition Labor Tax Rates')
```

(continues on next page)

(continued from previous page)

```
axes[1, 0].grid(True, alpha=0.3, axis='y')
for i, v in enumerate(τ_l_values):
    axes[1, 0].text(i, v + 0.005, f'{v:.4f}', ha='center', fontsize=9)

# Bar chart: Terminal debt/GDP
debt2gdp_values = [exp1_exo['ss1'].debt2gdp, exp1_endo['ss1'].debt2gdp,
                   exp2_exo['ss1'].debt2gdp, exp2_endo['ss1'].debt2gdp]
axes[1, 1].bar(cases, debt2gdp_values, color=colors, edgecolor='black')
axes[1, 1].axhline(0, color='k', linestyle='-', linewidth=0.5)
axes[1, 1].set_ylabel('Terminal Debt/GDP')
axes[1, 1].set_title('Terminal Steady State Debt/GDP')
axes[1, 1].grid(True, alpha=0.3, axis='y')
for i, v in enumerate(debt2gdp_values):
    y = v + 0.05 if v > 0 else v - 0.15
    axes[1, 1].text(
        i, y, f'{v:.4f}',
        ha='center', fontsize=9
    )

# Bar chart: Terminal interest rate
r_values = [exp1_exo['ss1'].r - tech.δ, exp1_endo['ss1'].r - tech.δ,
            exp2_exo['ss1'].r - tech.δ, exp2_endo['ss1'].r - tech.δ]
axes[1, 2].bar(cases, r_values, color=colors, edgecolor='black')
axes[1, 2].set_ylabel('Terminal Interest Rate (r - δ)')
axes[1, 2].set_title('Terminal Steady State Interest Rates')
axes[1, 2].grid(True, alpha=0.3, axis='y')
for i, v in enumerate(r_values):
    axes[1, 2].text(i, v + 0.002, f'{v:.4f}', ha='center', fontsize=9)

plt.suptitle('Comparison of All Four Reform Scenarios', fontsize=14, y=1.02)
plt.show()
```



The top row compares transition dynamics: the buy-out scheme (blue) accumulates higher debt during the transition due to compensation payments, while the government funding scheme (red) leads to large negative debt as the government becomes a net creditor.

The bottom-left bar chart shows that government funding requires higher transition tax rates than the buy-out scheme because benefit payments continue alongside capital accumulation.

Under endogenous pricing, the larger capital stock reduces the marginal product of capital and hence interest rates, as shown in the bottom-right panel.

The government-funded scheme (Experiment 2) delivers larger long-run efficiency gains because it preserves insurance against life span risk and labor income volatility that would be lost under privatization.

Higher labor income tax rates during the transition also provide implicit insurance against earnings risk, amplifying the efficiency advantage under endogenous prices.

116.10 Distribution surfaces

The 3D surface plots below show how assets and consumption evolve over both the age dimension and calendar time.

```
# Compute variances for 3D plotting
def compute_variances(results, ss0, hh):
    """Compute variance sequences from transition results."""

    _, μx_seq, μc_seq, k_seq, Gb_seq, F_seq, Ao_seq = results

    # Convert to numpy
    μx_seq = np.array(μx_seq)
    μc_seq_full = np.array(μc_seq)
```

(continues on next page)

(continued from previous page)

```
    F_seq = np.array(F_seq)
    Ao_seq = np.array(Ao_seq)
    Σx_arr_ss0 = np.array(ss0.Σx_arr)
    Σ0 = np.array(hh.Σ0)
    C = np.array(hh.C)

    # Get actual dimensions from data
    S_plus_1 = Ao_seq.shape[0]  # S+1
    T0_plus_1 = Ao_seq.shape[1]  # T0+1

    Σx_seq = np.empty((S_plus_1 + 1, T0_plus_1 + 1, hh.n_x, hh.n_x))
    Vc_seq = np.empty((S_plus_1, T0_plus_1))
    Va_seq = np.empty((S_plus_1, T0_plus_1))

    Σx_seq[:, 0] = Σ0
    Σx_seq[0, :] = Σx_arr_ss0[:T0_plus_1 + 1]

    CCT = C @ C.T
    for s in range(S_plus_1):
        Ao_s = Ao_seq[s]                    # (T0+1, n_x, n_x)
        Σx_s = Σx_seq[s, :T0_plus_1]        # (T0+1, n_x, n_x)
        F_s = F_seq[s]                       # (T0+1, n_x)
        Σx_seq[s + 1, 1:] = CCT + Ao_s @ Σx_s @ Ao_s.transpose(0, 2, 1)
        Vc_seq[s] = np.einsum('ti,tij,tj->t', F_s, Σx_s, F_s)
        Va_seq[s] = Σx_s[:, 0, 0]

    # Extract mean assets - match dimensions with Ao_seq
    μa_seq = μx_seq[:S_plus_1, :T0_plus_1, 0]
    μc_seq_out = μc_seq_full[:S_plus_1, :T0_plus_1]

    return μa_seq, Va_seq, μc_seq_out, Vc_seq

# Compute variances for each case
μa_bf, Va_bf, μc_bf, Vc_bf = compute_variances(
    exp1_exo['results_buyout'], exp1_exo['ss0'], hh
)
μa_be, Va_be, μc_be, Vc_be = compute_variances(
    exp1_endo['results'], exp1_endo['ss0'], hh
)
μa_af, Va_af, μc_af, Vc_af = compute_variances(
    exp2_exo['results'], exp2_exo['ss0'], hh
)
μa_ae, Va_ae, μc_ae, Vc_ae = compute_variances(
    exp2_endo['results'], exp2_endo['ss0'], hh
)

case_names = [
    'Buyout (Fixed)', 'Buyout (Endo)',
    'Gov Funding (Fixed)', 'Gov Funding (Endo)'
]


def plot_surface_grid(
        data_cases, case_names, zlabel,
        suptitle, cmap='viridis',
        transform=None):
    """Plot 2x2 grid of 3D surfaces for age-time data."""
```

(continues on next page)

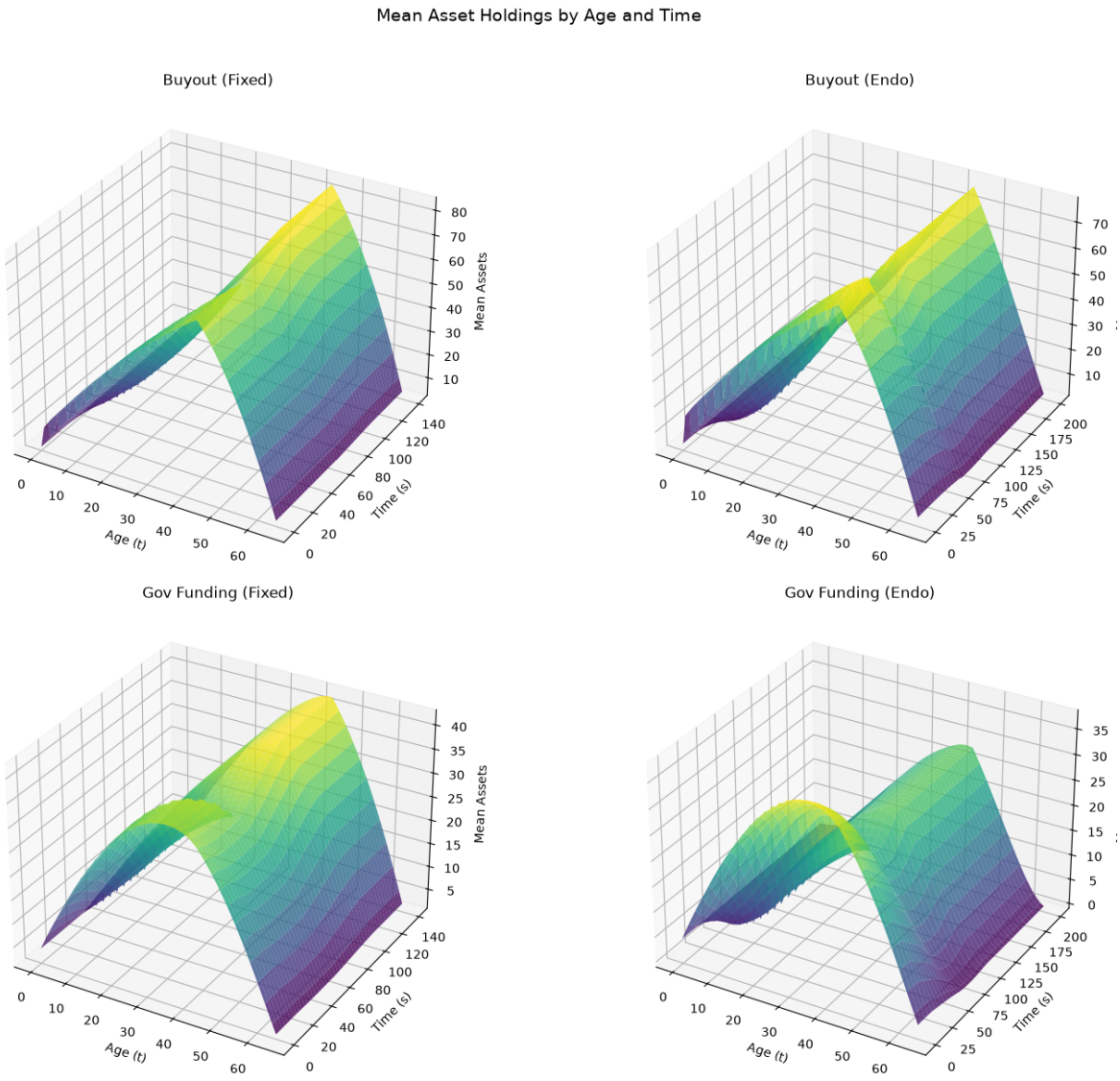
(continued from previous page)

```
    fig = plt.figure(figsize=(16, 12))
    for i, (data, name) in enumerate(zip(data_cases, case_names)):
        Z = transform(data) if transform is not None else data
        n_time, n_age = Z.shape
        X, Y = np.meshgrid(np.arange(n_age), np.arange(n_time))
        ax = fig.add_subplot(2, 2, i + 1, projection='3d')
        ax.plot_surface(X, Y, Z, cmap=cmap, edgecolor='none', alpha=0.8)
        ax.set_xlabel('Age (t)')
        ax.set_ylabel('Time (s)')
        ax.set_zlabel(zlabel)
        ax.set_title(name)
    plt.suptitle(suptitle, fontsize=14, y=1.02)
    plt.tight_layout()
    plt.show()
```

Each surface shows the joint distribution across age (t) and calendar time (s), revealing life-cycle patterns, transition dynamics, and cross-cohort heterogeneity.

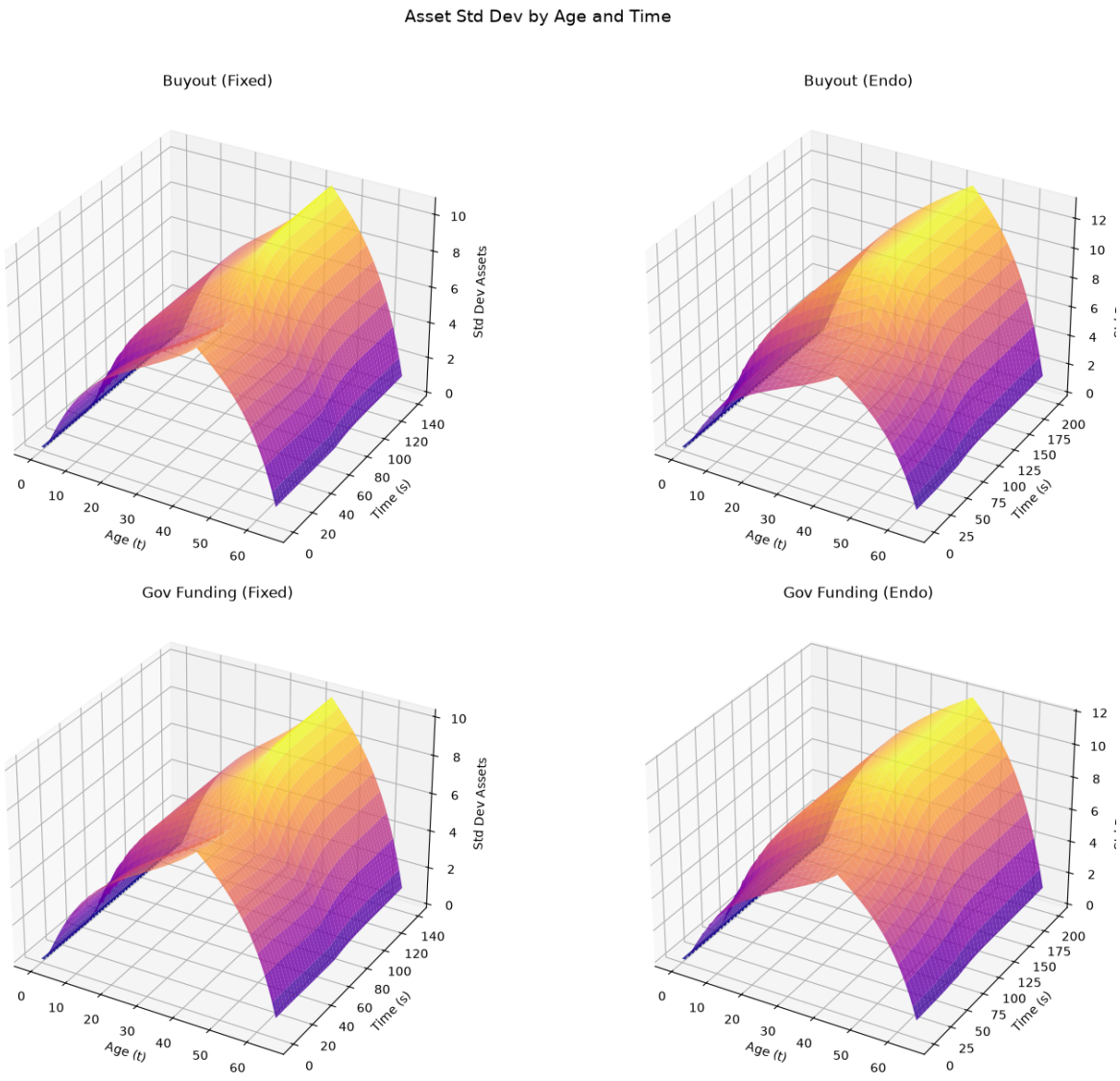
The mean asset surfaces display the hump-shaped life-cycle profile of asset holdings, with peak assets shifting as working generations increase their saving in response to the reform.

```
plot_surface_grid(
    [μa_bf, μa_be, μa_af, μa_ae],
    case_names, 'Mean Assets',
    'Mean Asset Holdings by Age and Time'
)
```



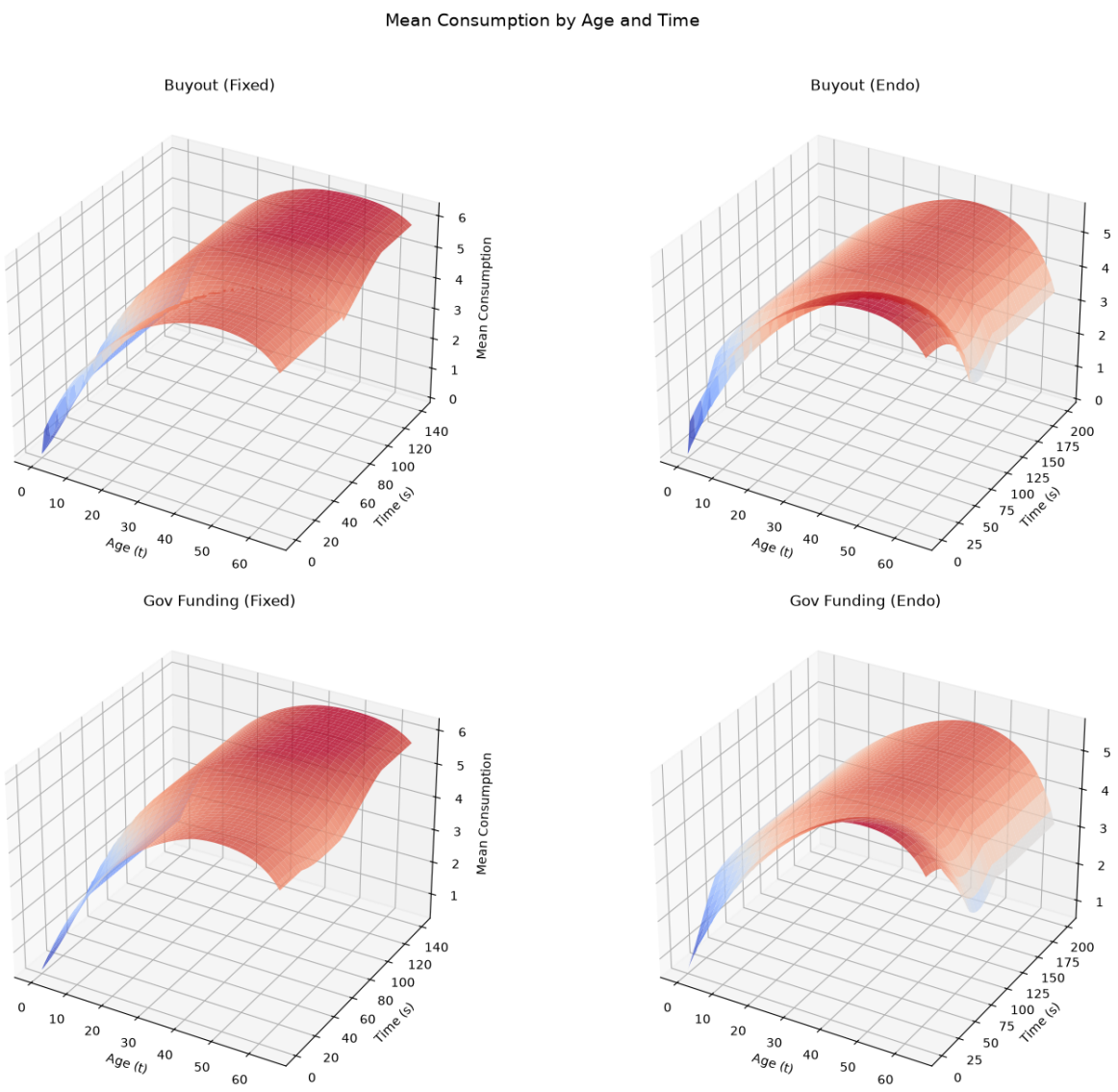
The asset variance surfaces show how cumulative income shocks cause dispersion to increase with age, with the transition potentially altering the rate of dispersion growth.

```
plot_surface_grid(
    [Va_bf, Va_be, Va_af, Va_ae],
    case_names, 'Std Dev Assets',
    'Asset Std Dev by Age and Time',
    cmap='plasma', transform=np.sqrt
)
```



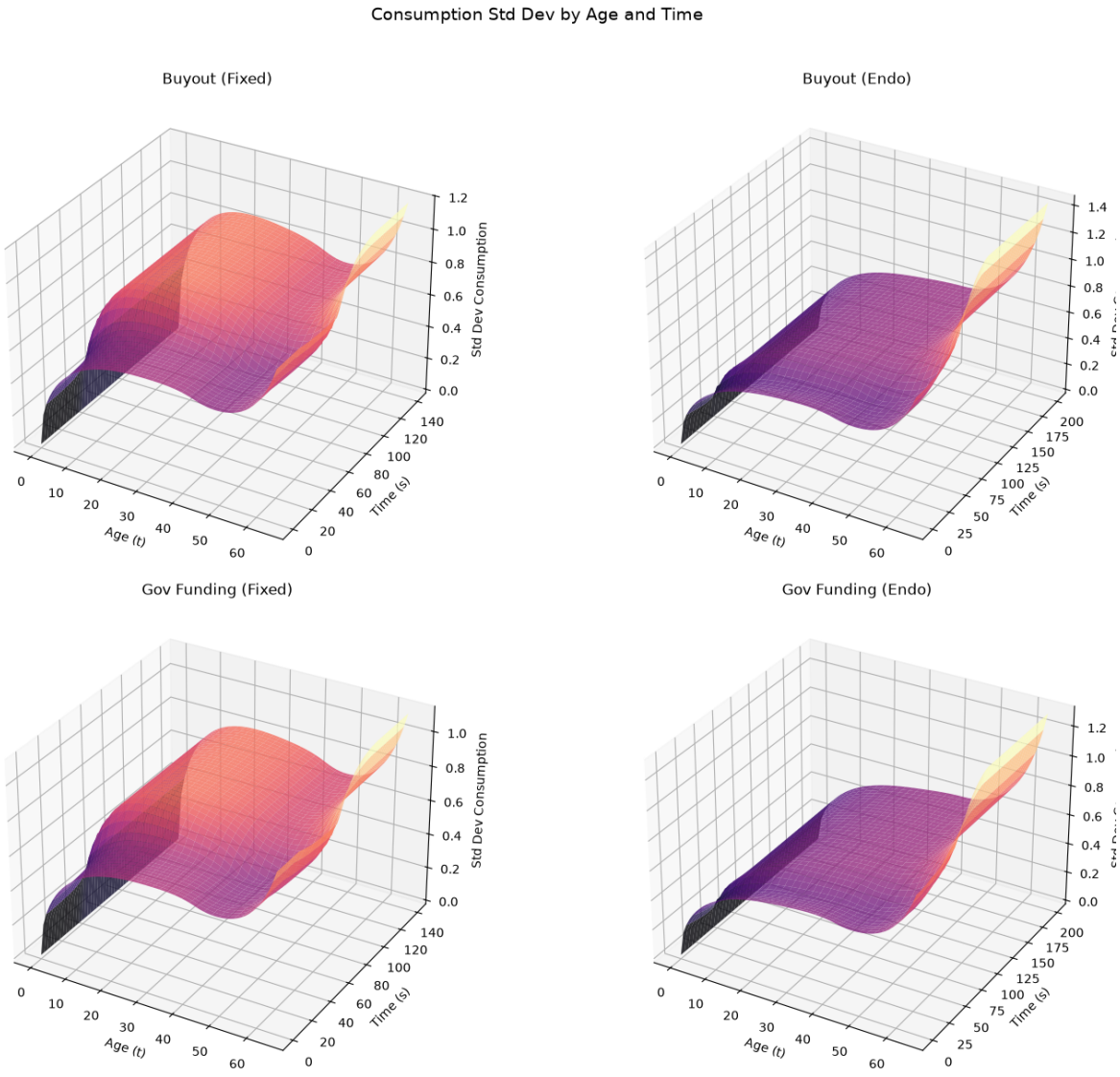
The mean consumption surfaces reflect the optimal consumption path, which should be smooth across ages due to the permanent income hypothesis underlying the model.

```
plot_surface_grid(
    [μc_bf, μc_be, μc_af, μc_ae],
    case_names, 'Mean Consumption',
    'Mean Consumption by Age and Time',
    cmap='coolwarm'
)
```



The consumption variance surfaces reveal how the certainty-equivalence property of the LQ framework shapes the within-cohort distribution of consumption over time.

```
plot_surface_grid(
    [Vc_bf, Vc_be, Vc_af, Vc_ae],
    case_names, 'Std Dev Consumption',
    'Consumption Std Dev by Age and Time',
    cmap='magma', transform=np.sqrt
)
```


Consumption Std Dev by Age and Time
Buyout (Fixed)
Buyout (Endo)
Gov Funding (Fixed)
Gov Funding (Endo)
Std Dev Consumption
Age (t)
Time (s)

Part XVI

Asset Pricing and Finance

CHAPTER

SEVEN

ASSET PRICING: FINITE STATE MODELS

Contents

- *Asset Pricing: Finite State Models*
  - *Overview*
  - *Pricing Models*
  - *Prices in the Risk-Neutral Case*
  - *Risk Aversion and Asset Prices*
  - *Exercises*

> "A little knowledge of geometric series goes a long way" – Robert E. Lucas, Jr.
>
> "Asset pricing is all about covariances" – Lars Peter Hansen

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install quantecon
```

117.1 Overview

An asset is a claim on one or more future payoffs.

The spot price of an asset depends primarily on

- the anticipated income stream
- attitudes about risk
- rates of time preference

In this lecture, we consider some standard pricing models and dividend stream specifications.

We study how prices and dividend-price ratios respond in these different scenarios.

We also look at creating and pricing *derivative* assets that repackage income streams.

Key tools for the lecture are

- Markov processses
- formulas for predicting future values of functions of a Markov state

- a formula for predicting the discounted sum of future values of a Markov state

Let's start with some imports:

```
import matplotlib.pyplot as plt
import numpy as np
import quantecon as qe
from numpy.linalg import eigvals, solve
```

117.2 Pricing Models

Let $\{d_t\}_{t \geq 0}$ be a stream of dividends

- A time-t **cum-dividend** asset is a claim to the stream $d_t, d_{t+1}, ...$
- A time-t **ex-dividend** asset is a claim to the stream $d_{t+1}, d_{t+2}, ...$

Let's look at some equations that we expect to hold for prices of assets under ex-dividend contracts (we will consider cum-dividend pricing in the exercises).

117.2.1 Risk-Neutral Pricing

Our first scenario is risk-neutral pricing.

Let $\beta = 1/(1 + \rho)$ be an intertemporal discount **factor**, where ρ is the **rate** at which agents discount the future.

The basic risk-neutral asset pricing equation for pricing one unit of an ex-dividend asset is

$$p_t = \beta \mathbb{E}_t[d_{t+1} + p_{t+1}] \tag{117.1}$$

This is a simple "cost equals expected benefit" relationship.

Here $\mathbb{E}_t[y]$ denotes the best forecast of y, conditioned on information available at time t.

More precisely, $\mathbb{E}_t[y]$ is the mathematical expectation of y conditional on information available at time t.

117.2.2 Pricing with Random Discount Factor

What happens if for some reason traders discount payouts differently depending on the state of the world?

Michael Harrison and David Kreps [Harrison and Kreps, 1979] and Lars Peter Hansen and Scott Richard [Hansen and Richard, 1987] showed that in quite general settings the price of an ex-dividend asset obeys

$$p_t = \mathbb{E}_t \left[m_{t+1} (d_{t+1} + p_{t+1}) \right] \tag{117.2}$$

for some **stochastic discount factor** m_{t+1}.

Here the fixed discount factor β in (117.1) has been replaced by the random variable m_{t+1}.

How anticipated future payoffs are evaluated now depends on statistical properties of m_{t+1}.

The stochastic discount factor can be specified to capture the idea that assets that tend to have good payoffs in bad states of the world are valued more highly than other assets whose payoffs don't behave that way.

This is because such assets pay well when funds are more urgently wanted.

We give examples of how the stochastic discount factor has been modeled below.

117.2.3 Asset Pricing and Covariances

Recall that, from the definition of a conditional covariance $\text{cov}_t(x_{t+1}, y_{t+1})$, we have

$$\mathbb{E}_t(x_{t+1}y_{t+1}) = \text{cov}_t(x_{t+1}, y_{t+1}) + \mathbb{E}_t x_{t+1}\mathbb{E}_t y_{t+1} \tag{117.3}$$

If we apply this definition to the asset pricing equation (117.2) we obtain

$$p_t = \mathbb{E}_t m_{t+1}\mathbb{E}_t(d_{t+1} + p_{t+1}) + \text{cov}_t(m_{t+1}, d_{t+1} + p_{t+1}) \tag{117.4}$$

It is useful to regard equation (117.4) as a generalization of equation (117.1)

- In equation (117.1), the stochastic discount factor $m_{t+1} = \beta$, a constant.
- In equation (117.1), the covariance term $\text{cov}_t(m_{t+1}, d_{t+1} + p_{t+1})$ is zero because $m_{t+1} = \beta$.
- In equation (117.1), $\mathbb{E}_t m_{t+1}$ can be interpreted as the reciprocal of the one-period risk-free gross interest rate.
- When m_{t+1} covaries more negatively with the payout $p_{t+1} + d_{t+1}$, the price of the asset is lower.

Equation (117.4) asserts that the covariance of the stochastic discount factor with the one period payout $d_{t+1} + p_{t+1}$ is an important determinant of the price p_t.

We give examples of some models of stochastic discount factors that have been proposed later in this lecture and also in a later lecture.

117.2.4 The Price-Dividend Ratio

Aside from prices, another quantity of interest is the **price-dividend ratio** $v_t := p_t/d_t$.

Let's write down an expression that this ratio should satisfy.

We can divide both sides of (117.2) by d_t to get

$$v_t = \mathbb{E}_t\left[m_{t+1}\frac{d_{t+1}}{d_t}(1 + v_{t+1})\right] \tag{117.5}$$

Below we'll discuss the implication of this equation.

117.3 Prices in the Risk-Neutral Case

What can we say about price dynamics on the basis of the models described above?

The answer to this question depends on

1. the process we specify for dividends
2. the stochastic discount factor and how it correlates with dividends

For now we'll study the risk-neutral case in which the stochastic discount factor is constant.

We'll focus on how an asset price depends on a dividend process.

117.3.1 Example 1: Constant Dividends

The simplest case is risk-neutral price of a constant, non-random dividend stream $d_t = d > 0$.

Removing the expectation from (117.1) and iterating forward gives

$$\begin{aligned} p_t &= \beta(d + p_{t+1}) \\ &= \beta(d + \beta(d + p_{t+2})) \\ &\quad \vdots \\ &= \beta(d + \beta d + \beta^2 d + \cdots + \beta^{k-2} d + \beta^{k-1} p_{t+k}) \end{aligned}$$

If $\lim_{k \to +\infty} \beta^{k-1} p_{t+k} = 0$, this sequence converges to

$$\bar{p} := \frac{\beta d}{1 - \beta} \tag{117.6}$$

This is the equilibrium price in the constant dividend case.

Indeed, simple algebra shows that setting $p_t = \bar{p}$ for all t satisfies the difference equation $p_t = \beta(d + p_{t+1})$.

117.3.2 Example 2: Dividends with Deterministic Growth Paths

Consider a growing, non-random dividend process $d_{t+1} = g d_t$ where $0 < g\beta < 1$.

While prices are not usually constant when dividends grow over time, a price dividend-ratio can be.

If we guess this, substituting $v_t = v$ into (117.5) as well as our other assumptions, we get $v = \beta g(1 + v)$.

Since $\beta g < 1$, we have a unique positive solution:

$$v = \frac{\beta g}{1 - \beta g}$$

The price is then

$$p_t = \frac{\beta g}{1 - \beta g} d_t$$

If, in this example, we take $g = 1 + \kappa$ and let $\rho := 1/\beta - 1$, then the price becomes

$$p_t = \frac{1 + \kappa}{\rho - \kappa} d_t$$

This is called the **Gordon formula**.

117.3.3 Example 3: Markov Growth, Risk-Neutral Pricing

Next, we consider a dividend process

$$d_{t+1} = g_{t+1} d_t \tag{117.7}$$

The stochastic growth factor $\{g_t\}$ is given by

$$g_t = g(X_t), \quad t = 1, 2, \ldots$$

where

1. $\{X_t\}$ is a finite Markov chain with state space S and transition probabilities

$$P(x, y) := \mathbb{P}\{X_{t+1} = y \,|\, X_t = x\} \qquad (x, y \in S)$$

2. g is a given function on S taking nonnegative values

You can think of

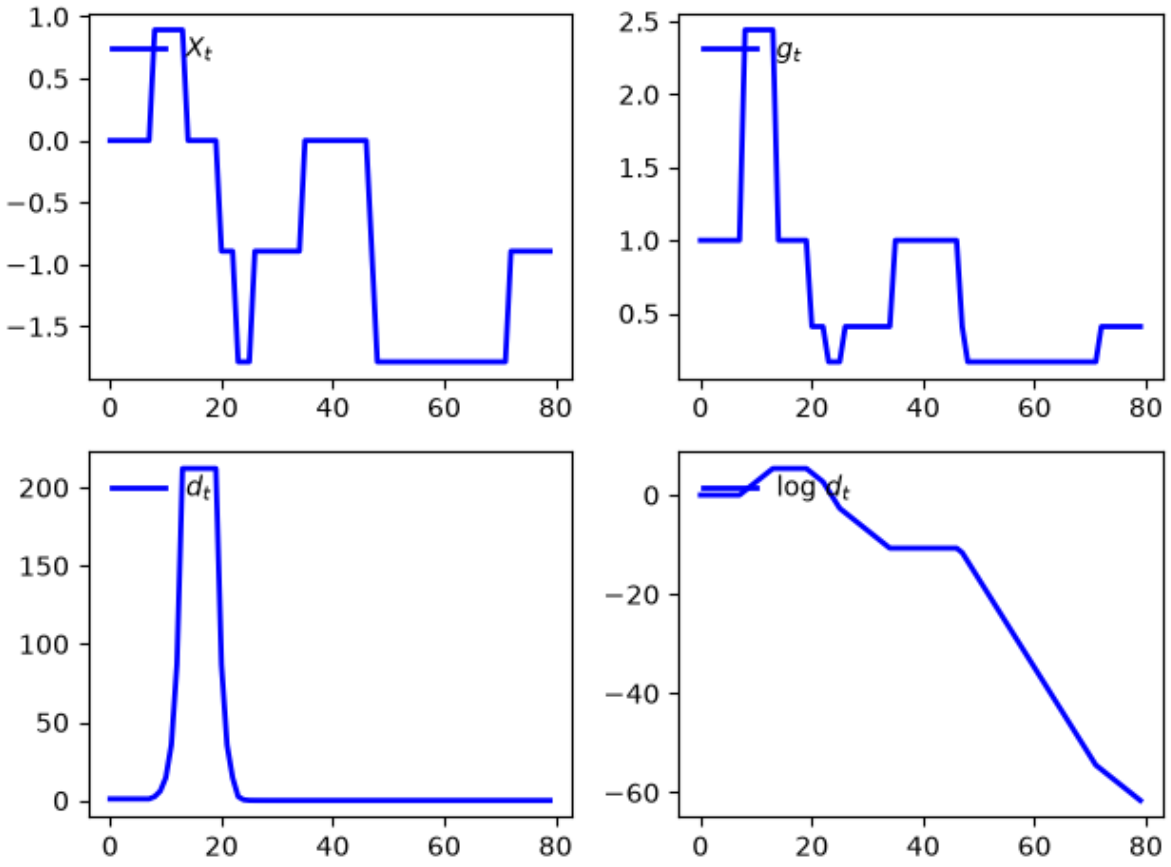
- S as n possible "states of the world" and X_t as the current state.
- g as a function that maps a given state X_t into a growth of dividends factor $g_t = g(X_t)$.
- $\ln g_t = \ln(d_{t+1}/d_t)$ is the growth rate of dividends.

(For a refresher on notation and theory for finite Markov chains see *this lecture*)

The next figure shows a simulation, where

- $\{X_t\}$ evolves as a discretized AR1 process produced using *Tauchen's method*.
- $g_t = \exp(X_t)$, so that $\ln g_t = X_t$ is the growth rate.

```
n = 7
mc = qe.tauchen(n, 0.96, 0.25)
sim_length = 80

x_series = mc.simulate(sim_length, init=np.median(mc.state_values))
g_series = np.exp(x_series)
d_series = np.cumprod(g_series) # Assumes d_0 = 1

series = [x_series, g_series, d_series, np.log(d_series)]
labels = ['$X_t$', '$g_t$', '$d_t$', r'$\log \, d_t$']

fig, axes = plt.subplots(2, 2)
for ax, s, label in zip(axes.flatten(), series, labels):
    ax.plot(s, 'b-', lw=2, label=label)
    ax.legend(loc='upper left', frameon=False)
plt.tight_layout()
plt.show()
```



Pricing Formula

To obtain asset prices in this setting, let's adapt our analysis from the case of deterministic growth.

In that case, we found that v is constant.

This encourages us to guess that, in the current case, v_t is a fixed function of the state X_t.

We seek a function v such that the price-dividend ratio satisfies $v_t = v(X_t)$.

We can substitute this guess into (117.5) to get

$$v(X_t) = \beta \mathbb{E}_t[g(X_{t+1})(1 + v(X_{t+1}))]$$

If we condition on $X_t = x$, this becomes

$$v(x) = \beta \sum_{y \in S} g(y)(1 + v(y))P(x, y)$$

or

$$v(x) = \beta \sum_{y \in S} K(x, y)(1 + v(y)) \quad \text{where} \quad K(x, y) := g(y)P(x, y) \tag{117.8}$$

Suppose that there are n possible states $x_1, \ldots, x_n$.

We can then think of (117.8) as n stacked equations, one for each state, and write it in matrix form as

$$v = \beta K(\mathbb{1} + v) \tag{117.9}$$

Here

- v is understood to be the column vector $(v(x_1), \ldots, v(x_n))'$.
- K is the matrix $(K(x_i, x_j))_{1 \leq i,j \leq n}$.
- $\mathbb{1}$ is a column vector of ones.

When does equation (117.9) have a unique solution?

From the *Neumann series lemma* and Gelfand's formula, equation (117.9) has a unique solution when βK has spectral radius strictly less than one.

Thus, we require that the eigenvalues of K be strictly less than β^{-1} in modulus.

The solution is then

$$v = (I - \beta K)^{-1} \beta K \mathbb{1} \tag{117.10}$$

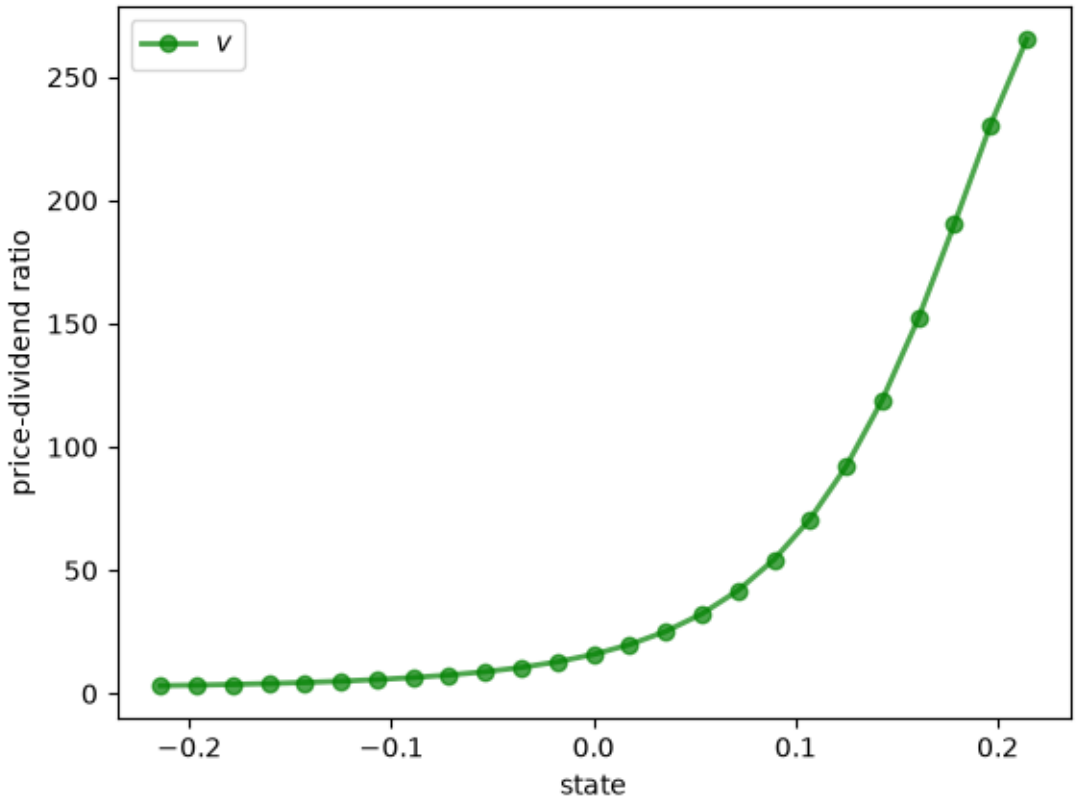
117.3.4 Code

Let's calculate and plot the price-dividend ratio at some parameters.

As before, we'll generate $\{X_t\}$ as a *discretized AR1 process* and set $g_t = \exp(X_t)$.

Here's the code, including a test of the spectral radius condition

```
n = 25  # Size of state space
β = 0.9
mc = qe.tauchen(n, 0.96, 0.02)

K = mc.P * np.exp(mc.state_values)

warning_message = "Spectral radius condition fails"
assert np.max(np.abs(eigvals(K))) < 1 / β,  warning_message

I = np.identity(n)
v = solve(I - β * K, β * K @ np.ones(n))

fig, ax = plt.subplots()
ax.plot(mc.state_values, v, 'g-o', lw=2, alpha=0.7, label='$v$')
ax.set_ylabel("price-dividend ratio")
ax.set_xlabel("state")
ax.legend(loc='upper left')
plt.show()
```



Why does the price-dividend ratio increase with the state?

The reason is that this Markov process is positively correlated, so high current states suggest high future states.

Moreover, dividend growth is increasing in the state.

The anticipation of high future dividend growth leads to a high price-dividend ratio.

117.4 Risk Aversion and Asset Prices

Now let's turn to the case where agents are risk averse.

We'll price several distinct assets, including

- An endowment stream
- A consol (a type of bond issued by the UK government in the 19th century)
- Call options on a consol

117.4.1 Pricing a Lucas Tree

Let's start with a version of the celebrated asset pricing model of Robert E. Lucas, Jr. [Lucas, 1978].

Lucas considered an abstract pure exchange economy with these features:

- a single non-storable consumption good
- a Markov process that governs the total amount of the consumption good available each period
- a single *tree* that each period yields *fruit* that equals the total amount of consumption available to the economy
- a competitive market in *shares* in the tree that entitles their owners to corresponding shares of the *dividend* stream, i.e., the *fruit* stream, yielded by the tree
- a representative consumer who in a competitive equilibrium
  - consumes the economy's entire endowment each period
  - owns 100 percent of the shares in the tree

As in [Lucas, 1978], we suppose that the stochastic discount factor takes the form

$$m_{t+1} = \beta \frac{u'(c_{t+1})}{u'(c_t)} \tag{117.11}$$

where u is a concave utility function and c_t is time t consumption of a representative consumer.

(A derivation of this expression is given in a later lecture)

Assume the existence of an endowment that follows growth process (117.7).

The asset being priced is a claim on the endowment process, i.e., the **Lucas tree** described above.

Following [Lucas, 1978], we suppose that in equilibrium the representative consumer's consumption equals the aggregate endowment, so that $d_t = c_t$ for all t.

For utility, we'll assume the **constant relative risk aversion** (CRRA) specification

$$u(c) = \frac{c^{1-\gamma}}{1-\gamma} \text{ with } \gamma > 0 \tag{117.12}$$

When $\gamma = 1$ we let $u(c) = \ln c$.

Inserting the CRRA specification into (117.11) and using $c_t = d_t$ gives

$$m_{t+1} = \beta \left(\frac{c_{t+1}}{c_t} \right)^{-\gamma} = \beta g_{t+1}^{-\gamma} \tag{117.13}$$

Substituting this into (117.5) gives the price-dividend ratio formula

$$v(X_t) = \beta \mathbb{E}_t \left[g(X_{t+1})^{1-\gamma} (1 + v(X_{t+1})) \right] \tag{117.14}$$

Conditioning on $X_t = x$, we can write this as

$$v(x) = \beta \sum_{y \in S} g(y)^{1-\gamma} (1 + v(y)) P(x, y)$$

If we let

$$J(x, y) := g(y)^{1-\gamma} P(x, y)$$

then we can rewrite equation (117.14) in vector form as

$$v = \beta J(\mathbb{1} + v)$$

Assuming that the spectral radius of J is strictly less than β^{-1}, this equation has the unique solution

$$v = (I - \beta J)^{-1} \beta J \mathbb{1} \tag{117.15}$$

We will define a function tree_price to compute v given parameters stored in the class AssetPriceModel

```
class AssetPriceModel:
    """
    A class that stores the primitives of the asset pricing model.

    Parameters
    ----------
    β : scalar, float
        Discount factor
    mc : MarkovChain
        Contains the transition matrix and set of state values for the state
        process
    γ : scalar(float)
        Coefficient of risk aversion
    g : callable
        The function mapping states to growth rates

    """
    def __init__(self, β=0.96, mc=None, γ=2.0, g=np.exp):
        self.β, self.γ = β, γ
        self.g = g

        # A default process for the Markov chain
        if mc is None:
            self.ρ = 0.9
            self.σ = 0.02
            self.mc = qe.tauchen(n, self.ρ, self.σ)
        else:
            self.mc = mc

        self.n = self.mc.P.shape[0]

    def test_stability(self, Q):
        """
        Stability test for a given matrix Q.
        """
        sr = np.max(np.abs(eigvals(Q)))
        if not sr < 1 / self.β:
            msg = f"Spectral radius condition failed with radius = {sr}"
            raise ValueError(msg)


def tree_price(ap):
    """
    Computes the price-dividend ratio of the Lucas tree.

    Parameters
    ----------
    ap: AssetPriceModel
        An instance of AssetPriceModel containing primitives

    Returns
    -------
```

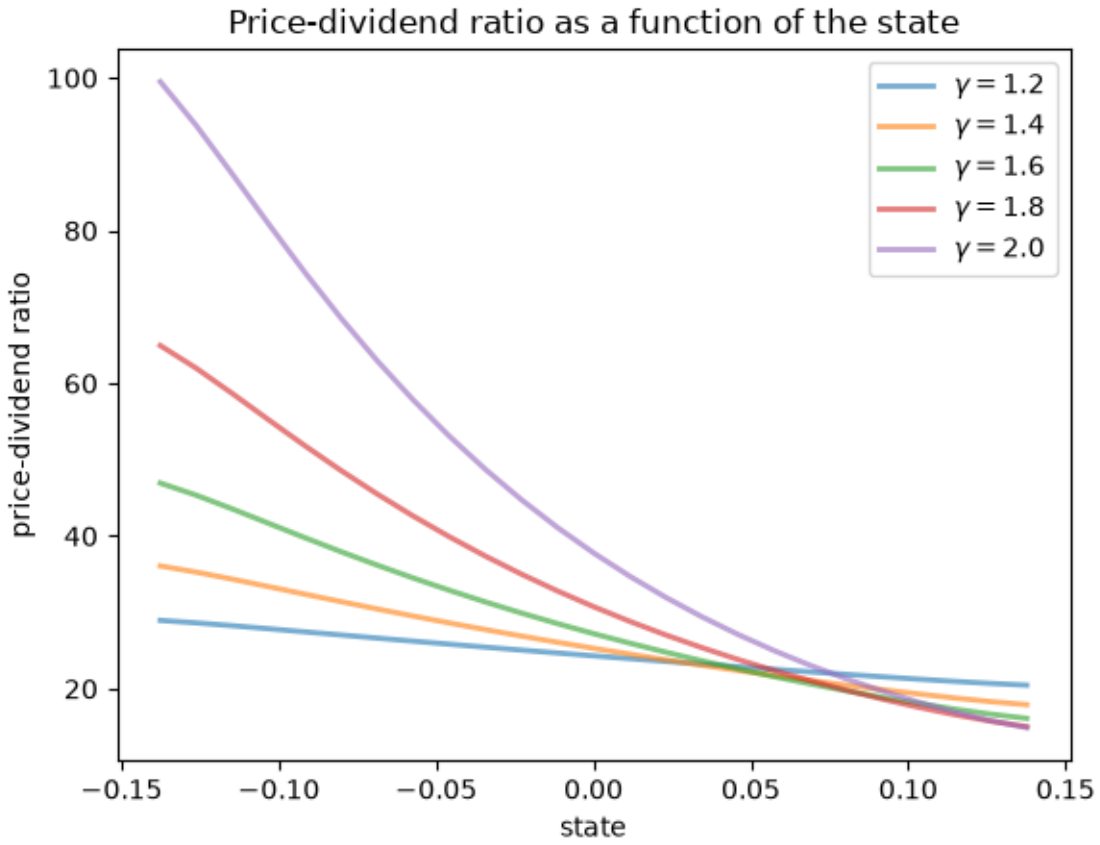
(continues on next page)

(continued from previous page)

```
    v : array_like(float)
        Lucas tree price-dividend ratio

    """
    # Simplify names, set up matrices
    β, γ, P, y = ap.β, ap.γ, ap.mc.P, ap.mc.state_values
    J = P * ap.g(y)**(1 - γ)

    # Make sure that a unique solution exists
    ap.test_stability(J)

    # Compute v
    I = np.identity(ap.n)
    Ones = np.ones(ap.n)
    v = solve(I - β * J, β * J @ Ones)

    return v
```

Here's a plot of v as a function of the state for several values of γ, with a positively correlated Markov process and $g(x) = \exp(x)$

```
ys = [1.2, 1.4, 1.6, 1.8, 2.0]
ap = AssetPriceModel()
states = ap.mc.state_values

fig, ax = plt.subplots()

for γ in ys:
    ap.γ = γ
    v = tree_price(ap)
    ax.plot(states, v, lw=2, alpha=0.6, label=rf"$\gamma = {γ}$")

ax.set_title('Price-dividend ratio as a function of the state')
ax.set_ylabel("price-dividend ratio")
ax.set_xlabel("state")
ax.legend(loc='upper right')
plt.show()
```



Notice that v is decreasing in each case.

This is because, with a positively correlated state process, higher states indicate higher future consumption growth.

With the stochastic discount factor (117.13), higher growth decreases the discount factor, lowering the weight placed on future dividends.

Special Cases

In the special case $\gamma = 1$, we have $J = P$.

Recalling that $P^i \mathbb{1} = \mathbb{1}$ for all i and applying *Neumann's geometric series lemma*, we are led to

$$v = \beta(I - \beta P)^{-1} \mathbb{1} = \beta \sum_{i=0}^{\infty} \beta^i P^i \mathbb{1} = \beta \frac{1}{1-\beta} \mathbb{1}$$

Thus, with log preferences, the price-dividend ratio for a Lucas tree is constant.

Alternatively, if $\gamma = 0$, then $J = K$ and we recover the risk-neutral solution (117.10).

This is as expected, since $\gamma = 0$ implies $u(c) = c$ (and hence agents are risk-neutral).

117.4.2 A Risk-Free Consol

Consider the same pure exchange representative agent economy.

A risk-free consol promises to pay a constant amount $\zeta > 0$ each period.

Recycling notation, let p_t now be the price of an ex-coupon claim to the consol.

An ex-coupon claim to the consol entitles an owner at the end of period t to

- ζ in period $t+1$, plus
- the right to sell the claim for p_{t+1} next period

The price satisfies (117.2) with $d_t = \zeta$, or

$$p_t = \mathbb{E}_t \left[m_{t+1} (\zeta + p_{t+1}) \right]$$

With the stochastic discount factor (117.13), this becomes

$$p_t = \mathbb{E}_t \left[\beta g_{t+1}^{-\gamma} (\zeta + p_{t+1}) \right] \tag{117.16}$$

Guessing a solution of the form $p_t = p(X_t)$ and conditioning on $X_t = x$, we get

$$p(x) = \beta \sum_{y \in S} g(y)^{-\gamma} (\zeta + p(y)) P(x, y)$$

Letting $M(x, y) = P(x, y) g(y)^{-\gamma}$ and rewriting in vector notation yields the solution

$$p = (I - \beta M)^{-1} \beta M \zeta \mathbb{1} \tag{117.17}$$

The above is implemented in the function consol_price.

```
def consol_price(ap, ζ):
    """
    Computes price of a consol bond with payoff ζ

    Parameters
    ----------
    ap: AssetPriceModel
        An instance of AssetPriceModel containing primitives

    ζ : scalar(float)
        Coupon of the console

    Returns
    -------
    p : array_like(float)
        Console bond prices

    """
    # Simplify names, set up matrices
    β, γ, P, y = ap.β, ap.γ, ap.mc.P, ap.mc.state_values
    M = P * ap.g(y)**(- γ)

    # Make sure that a unique solution exists
    ap.test_stability(M)

    # Compute price
    I = np.identity(ap.n)
```

(continues on next page)

(continued from previous page)

```
    Ones = np.ones(ap.n)
    p = solve(I - β * M, β * ζ * M @ Ones)

    return p
```

117.4.3 Pricing an Option to Purchase the Consol

Let's now price options of various maturities.

We'll study an option that gives the owner the right to purchase a consol at a price p_S.

An Infinite Horizon Call Option

We want to price an **infinite horizon** option to purchase a consol at a price p_S.

The option entitles the owner at the beginning of a period either

1. to purchase the bond at price p_S now, or
2. not to exercise the option to purchase the asset now but to retain the right to exercise it later

Thus, the owner either *exercises* the option now or chooses *not to exercise* and wait until next period.

This is termed an infinite-horizon **call option** with **strike price** p_S.

The owner of the option is entitled to purchase the consol at price p_S at the beginning of any period, after the coupon has been paid to the previous owner of the bond.

The fundamentals of the economy are identical with the one above, including the stochastic discount factor and the process for consumption.

Let $w(X_t, p_S)$ be the value of the option when the time t growth state is known to be X_t but *before* the owner has decided whether to exercise the option at time t (i.e., today).

Recalling that $p(X_t)$ is the value of the consol when the initial growth state is X_t, the value of the option satisfies

$$w(X_t, p_S) = \max \left\{ \beta \, \mathbb{E}_t \frac{u'(c_{t+1})}{u'(c_t)} w(X_{t+1}, p_S), \; p(X_t) - p_S \right\}$$

The first term on the right is the value of waiting, while the second is the value of exercising now.

We can also write this as

$$w(x, p_S) = \max \left\{ \beta \sum_{y \in S} P(x, y) g(y)^{-\gamma} w(y, p_S), \; p(x) - p_S \right\} \tag{117.18}$$

With $M(x, y) = P(x, y) g(y)^{-\gamma}$ and w as the vector of values $(w(x_i), p_S)_{i=1}^n$, we can express (117.18) as the nonlinear vector equation

$$w = \max\{\beta M w, \; p - p_S \mathbb{1}\} \tag{117.19}$$

To solve (117.19), form an operator T that maps vector w into vector Tw via

$$Tw = \max\{\beta M w, \; p - p_S \mathbb{1}\}$$

Start at some initial w and iterate with T to convergence .

We can find the solution with the following function call_option

```
def call_option(ap, ζ, p_s, ϵ=1e-7):
    """
    Computes price of a call option on a consol bond.

    Parameters
    ----------
    ap: AssetPriceModel
        An instance of AssetPriceModel containing primitives

    ζ : scalar(float)
        Coupon of the console

    p_s : scalar(float)
        Strike price

    ϵ : scalar(float), optional(default=1e-8)
        Tolerance for infinite horizon problem

    Returns
    -------
    w : array_like(float)
        Infinite horizon call option prices

    """
    # Simplify names, set up matrices
    β, γ, P, y = ap.β, ap.γ, ap.mc.P, ap.mc.state_values
    M = P * ap.g(y)**(- γ)

    # Make sure that a unique consol price exists
    ap.test_stability(M)

    # Compute option price
    p = consol_price(ap, ζ)
    w = np.zeros(ap.n)
    error = ϵ + 1
    while error > ϵ:
        # Maximize across columns
        w_new = np.maximum(β * M @ w, p - p_s)
        # Find maximal difference of each component and update
        error = np.amax(np.abs(w - w_new))
        w = w_new

    return w
```

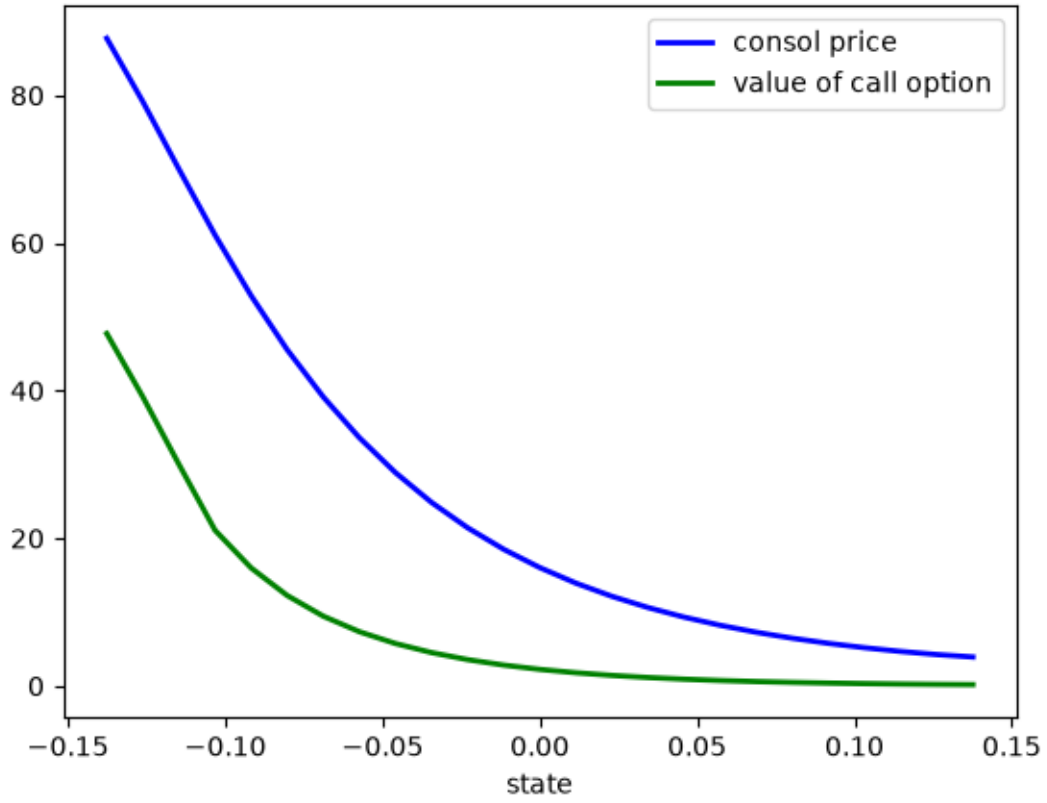
Here's a plot of w compared to the consol price when $P_S = 40$

```
ap = AssetPriceModel(β=0.9)
ζ = 1.0
strike_price = 40

x = ap.mc.state_values
p = consol_price(ap, ζ)
w = call_option(ap, ζ, strike_price)

fig, ax = plt.subplots()
ax.plot(x, p, 'b-', lw=2, label='consol price')
ax.plot(x, w, 'g-', lw=2, label='value of call option')
```

(continues on next page)

(continued from previous page)

```
ax.set_xlabel("state")
ax.legend(loc='upper right')
plt.show()
```



In high values of the Markov growth state, the value of the option is close to zero.

This is despite the facts that the Markov chain is irreducible and that low states — where the consol prices are high — will be visited recurrently.

The reason for low valuations in high Markov growth states is that $\beta = 0.9$, so future payoffs are discounted substantially.

117.4.4 Risk-Free Rates

Let's look at risk-free interest rates over different periods.

The One-period Risk-free Interest Rate

As before, the stochastic discount factor is $m_{t+1} = \beta g_{t+1}^{-\gamma}$.

It follows that the reciprocal R_t^{-1} of the gross risk-free interest rate R_t in state x is

$$\mathbb{E}_t m_{t+1} = \beta \sum_{y \in S} P(x,y) g(y)^{-\gamma}$$

We can write this as

$$m_1 = \beta M \mathbb{1}$$

where the i-th element of m_1 is the reciprocal of the one-period gross risk-free interest rate in state x_i.

Other Terms

Let m_j be an $n \times 1$ vector whose i th component is the reciprocal of the j -period gross risk-free interest rate in state x_i.

Then $m_1 = \beta M$, and $m_{j+1} = M m_j$ for $j \geq 1$.

117.5 Exercises

Exercise 117.5.1

In the lecture, we considered **ex-dividend assets**.

A **cum-dividend** asset is a claim to the stream $d_t, d_{t+1}, \ldots$.

Following (117.1), find the risk-neutral asset pricing equation for one unit of a cum-dividend asset.

With a constant, non-random dividend stream $d_t = d > 0$, what is the equilibrium price of a cum-dividend asset?

With a growing, non-random dividend process $d_t = g d_t$ where $0 < g\beta < 1$, what is the equilibrium price of a cum-dividend asset?

Solution

For a cum-dividend asset, the basic risk-neutral asset pricing equation is

$$p_t = d_t + \beta \mathbb{E}_t[p_{t+1}]$$

With constant dividends, the equilibrium price is

$$p_t = \frac{1}{1-\beta} d_t$$

With a growing, non-random dividend process, the equilibrium price is

$$p_t = \frac{1}{1-\beta g} d_t$$

Exercise 117.5.2

Consider the following primitives

```
n = 5  # Size of State Space
P = np.full((n, n), 0.0125)
P[range(n), range(n)] += 1 - P.sum(1)
# State values of the Markov chain
s = np.array([0.95, 0.975, 1.0, 1.025, 1.05])
γ = 2.0
β = 0.94
```

Let g be defined by $g(x) = x$ (that is, g is the identity map).

Compute the price of the Lucas tree.

Do the same for

- the price of the risk-free consol when $\zeta = 1$
- the call option on the consol when $\zeta = 1$ and $p_S = 150.0$

Solution

First, let's enter the parameters:

```
n = 5
P = np.full((n, n), 0.0125)
P[range(n), range(n)] += 1 - P.sum(1)
s = np.array([0.95, 0.975, 1.0, 1.025, 1.05])  # State values
mc = qe.MarkovChain(P, state_values=s)

γ = 2.0
β = 0.94
ζ = 1.0
p_s = 150.0
```

Next, we'll create an instance of `AssetPriceModel` to feed into the functions

```
apm = AssetPriceModel(β=β, mc=mc, γ=γ, g=lambda x: x)
```

Now we just need to call the relevant functions on the data:

```
tree_price(apm)
```

```
array([29.47401578, 21.93570661, 17.57142236, 14.72515002, 12.72221763])
```

```
consol_price(apm, ζ)
```

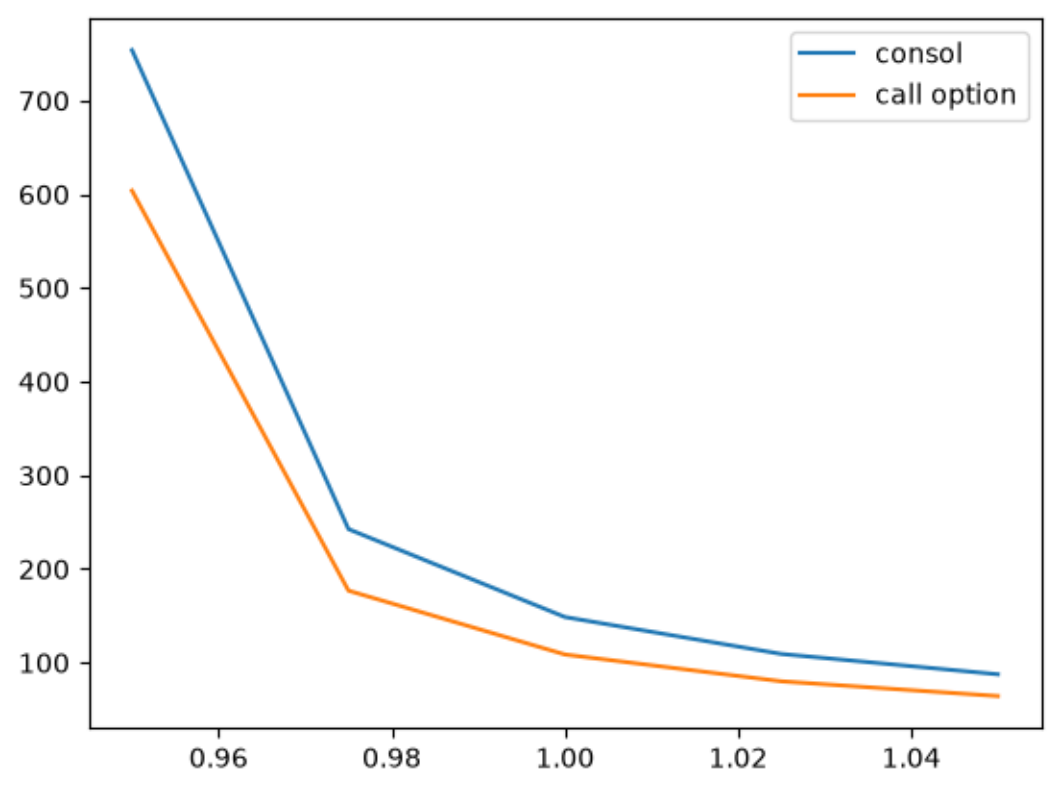
```
array([753.87100476, 242.55144082, 148.67554548, 109.25108965,
       87.56860139])
```

```
call_option(apm, ζ, p_s)
```

```
array([603.87100476, 176.8393343 , 108.67734499,  80.05179254,
       64.30843748])
```

Let's show the last two functions as a plot

```
fig, ax = plt.subplots()
ax.plot(s, consol_price(apm, ζ), label='consol')
ax.plot(s, call_option(apm, ζ, p_s), label='call option')
ax.legend()
plt.show()
```


consol
call option
700
600
500
400
300
200
100
0.96
0.98
1.00
1.02
1.04

Exercise 117.5.3

Let's consider finite horizon call options, which are more common than infinite horizon ones.

Finite horizon options obey functional equations closely related to (117.18).

A k period option expires after k periods.

If we view today as date zero, a k period option gives the owner the right to exercise the option to purchase the risk-free consol at the strike price p_S at dates $0, 1, \ldots, k-1$.

The option expires at time k.

Thus, for $k = 1, 2, \ldots$, let $w(x, k)$ be the value of a k-period option.

It obeys

$$w(x,k) = \max\left\{\beta \sum_{y \in S} P(x,y) g(y)^{-\gamma} w(y, k-1),\ p(x) - p_S\right\}$$

where $w(x, 0) = 0$ for all x.

We can express this as a sequence of nonlinear vector equations

$$w_k = \max\{\beta M w_{k-1},\ p - p_S \mathbb{1}\} \quad k = 1, 2, \ldots \quad \text{with } w_0 = 0$$

Write a function that computes w_k for any given k.

Compute the value of the option with `k = 5` and `k = 25` using parameter values as in Exercise 117.5.1.

Is one higher than the other? Can you give intuition?

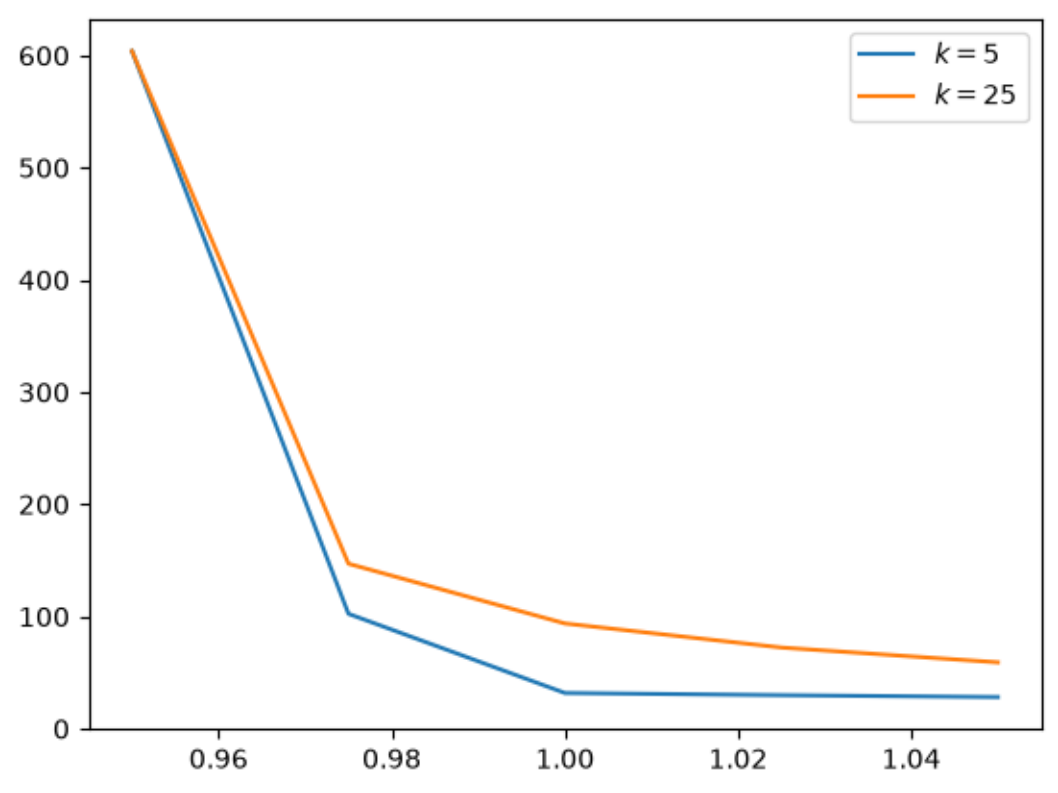
Solution

Here's a suitable function:

```
def finite_horizon_call_option(ap, ζ, p_s, k):
    """
    Computes k period option value.
    """
    # Simplify names, set up matrices
    β, γ, P, y = ap.β, ap.γ, ap.mc.P, ap.mc.state_values
    M = P * ap.g(y)**(- γ)

    # Make sure that a unique solution exists
    ap.test_stability(M)


    # Compute option price
    p = consol_price(ap, ζ)
    w = np.zeros(ap.n)
    for i in range(k):
        # Maximize across columns
        w = np.maximum(β * M @ w, p - p_s)

    return w
```

Now let's compute the option values at `k=5` and `k=25`

```
fig, ax = plt.subplots()
for k in [5, 25]:
    w = finite_horizon_call_option(apm, ζ, p_s, k)
    ax.plot(s, w, label=rf'$k = {k}$')
ax.legend()
plt.show()
```



Not surprisingly, options with larger k are worth more.

This is because an owner has a longer horizon over which the option can be exercised.

CHAPTER

EIGHT

STOCHASTIC CONSUMPTION, RISK AVERSION, AND THE TEMPORAL BEHAVIOR OF ASSET RETURNS

Contents

- *Stochastic Consumption, Risk Aversion, and the Temporal Behavior of Asset Returns*
  - *Overview*
  - *Euler equation*
  - *The Euler equation under lognormality*
  - *The restricted system and its likelihood*
  - *Likelihood implementation*
  - *Preference parameters and likelihood ratio tests*
  - *Predictability and the R-squared restriction*
  - *Return-difference tests*
  - *Empirical MLE estimation*
  - *Residual diagnostics*
  - *Connection to the equity premium puzzle*
  - *Another approach*

Evans and Honkapohja: What were the profession's most important responses to the Lucas Critique?

Sargent: There were two. The first and most optimistic response was complete rational expectations econometrics. A rational expectations equilibrium is a likelihood function. Maximize it.

– An Interview with Thomas J. Sargent [Evans and Honkapohja, 2005]

118.1 Overview

This lecture describes how Hansen and Singleton [1983] formulated a complete statistical model of asset returns and consumption growth, then estimated its parameters by the method of maximum likelihood.

They detect a defects in their model, one of which Mehra and Prescott [1985] later called the **equity premium puzzle**.

Hansen and Singleton [1983] study a consumption-based asset pricing model in which a representative consumer with CRRA preferences chooses how to allocate wealth across traded assets.

First-order conditions for asset holdings are stochastic Euler equations that connect consumption growth, asset returns, and preference parameters.

Hansen and Singleton [1983] assume that consumption growth and asset returns are *jointly lognormal*.

Then Euler equations imply a set of restrictions on a the joint distribution of asset prices and returns.

These restrict a linear time-series model for logarithms of consumption growth and returns in which predictable movements in log returns are proportional to predictable movements in log consumption growth.

Hansen and Singleton estimated their model by *Maximum Likelihood Estimation*.

The empirical findings of Hansen and Singleton [1983] constitute what Mehra and Prescott [1985] would later call the **equity premium puzzle**.

To keep lecture this lecture narrowly focused, we estimate one return at a time (either a market proxy or a T-bill) rather than the full multi-asset systems studied by Hansen and Singleton [1983].

- we use only monthly nondurable consumption (`ND`).

```
from itertools import combinations

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from IPython.display import Latex
from scipy import stats
from scipy.linalg import LinAlgError, cholesky, solve_triangular
from scipy.optimize import minimize
from statsmodels.stats.stattools import durbin_watson
```

We also define a helper to display DataFrames as LaTeX arrays in the hidden cell below

118.2 Euler equation

Consider a single-good economy of identical consumers whose utility functions are in the CRRA form

$$U(c_t) = c_t^\gamma / \gamma, \quad \gamma < 1, \tag{118.1}$$

where c_t is aggregate real per capita consumption and $U(\cdot)$ is the period utility function.

The representative consumer chooses a stochastic consumption plan to maximize the expected value of a time-additive utility function,

$$E_0 \sum_{t=0}^{\infty} \beta^t U(c_t), \quad 0 < \beta < 1. \tag{118.2}$$

Consumers substitute present for future consumption by trading the ownership rights of N financial and capital assets.

Let $\mathbf{w}_t$ denote the holdings of the N assets at date t, $\mathbf{q}_t$ the vector of asset prices, $\mathbf{d}_t$ the vector of dividends, and y_t real labor income.

A feasible consumption and investment plan $\{c_t, \mathbf{w}_t\}$ must satisfy the sequence of budget constraints

$$c_t + \mathbf{q}_t \cdot \mathbf{w}_{t+1} \leq (\mathbf{q}_t + \mathbf{d}_t) \cdot \mathbf{w}_t + y_t, \tag{118.3}$$

where $(\mathbf{q}_t + \mathbf{d}_t) \cdot \mathbf{w}_t$ is the cum-dividend value of the portfolio carried into period t.

To derive the first-order conditions, attach a Lagrange multiplier λ_t to the budget constraint (118.3) at each date t and form the Lagrangian

$$\mathcal{L} = E_0 \sum_{t=0}^{\infty} \beta^t \left\{ U(c_t) + \lambda_t \left[(\mathbf{q}_t + \mathbf{d}_t) \cdot \mathbf{w}_t + y_t - c_t - \mathbf{q}_t \cdot \mathbf{w}_{t+1} \right] \right\}. \tag{118.4}$$

Differentiating $\mathcal{L}$ with respect to c_t gives

$$\frac{\partial \mathcal{L}}{\partial c_t} = 0 \implies \lambda_t = U'(c_t).$$

Differentiating with respect to $w_{i,t+1}$, the holdings of asset i carried from date t into date $t+1$, collects two terms: $w_{i,t+1}$ appears in the date-t budget constraint as well as in the date-$(t+1)$ constraint:

$$\frac{\partial \mathcal{L}}{\partial w_{i,t+1}} = 0 \implies E_0 \left[\beta^t \left(-\lambda_t \, q_{it} + \beta \, \lambda_{t+1} (q_{i,t+1} + d_{i,t+1}) \right) \right] = 0.$$

By the law of iterated expectations, this becomes

$$E_0 \left[\beta^t \, E_t \left(-\lambda_t \, q_{it} + \beta \, \lambda_{t+1} (q_{i,t+1} + d_{i,t+1}) \right) \right] = 0.$$

Hence,

$$\lambda_t \, q_{it} = \beta \, E_t \left[\lambda_{t+1} (q_{i,t+1} + d_{i,t+1}) \right].$$

Substituting $\lambda_t = U'(c_t)$ and $\lambda_{t+1} = U'(c_{t+1})$ yields

$$q_{it} \, U'(c_t) = \beta \, E_t \left[U'(c_{t+1}) (q_{i,t+1} + d_{i,t+1}) \right].$$

Dividing both sides by q_{it} and defining the gross return $r_{it+1} := (q_{i,t+1} + d_{i,t+1})/q_{it}$ yields the stochastic Euler equation:

$$U'(c_t) = \beta E_t [U'(c_{t+1}) \, r_{it+1}], \quad i = 1, \ldots, N, \tag{118.5}$$

where r_{it+1} is the gross real return on asset i.

Substituting the CRRA marginal utility $U'(c_t) = c_t^{\gamma-1} = c_t^{\alpha}$ with $\alpha := \gamma - 1$ into (118.5) and rearranging gives

$$E_t \left[\beta \left(\frac{c_{t+1}}{c_t} \right)^{\alpha} r_{it+1} \right] = 1, \quad i = 1, \ldots, N. \tag{118.6}$$

The coefficient of relative risk aversion is $-\alpha$.

118.3 The Euler equation under lognormality

Using the Euler equation (118.6) derived above, we now impose the distributional assumptions of Hansen and Singleton [1983].

Let $x_t = c_t / c_{t-1}$ denote the consumption ratio, and define $u_{it} = x_t^{\alpha} r_{it}$ where r_{it} is the gross real return on asset i.

The Euler equation (118.6) states $E_{t-1}[u_{it}] = 1/\beta$.

Define log variables $X_t = \log x_t$, $R_{it} = \log r_{it}$, and $U_{it} = \log u_{it}$, so that

$$U_{i,t} = \alpha X_t + R_{i,t}. \tag{118.7}$$

Assume the vector process $\{Y_t\} = \{(X_t, R_{1t}, \dots, R_{nt})^\top\}$ is jointly stationary and Gaussian.

Under this assumption, U_{it} conditional on the information set ψ_{t-1} is normal with constant variance σ_i^2 and a conditional mean $\mu_{i,t-1}$ that is a linear function of past Y's.

Because $u_{it} = \exp(U_{it})$ is conditionally lognormal,

$$E_{t-1}[u_{it}] = \exp\left(\mu_{i,t-1} + \tfrac{1}{2}\sigma_i^2\right). \tag{118.8}$$

Setting $E_{t-1}[u_{it}] = 1/\beta$ and taking logs gives $\mu_{i,t-1} + \sigma_i^2/2 = -\log\beta$.

Now define the innovation

$$V_{i,t} := U_{i,t} - \mu_{i,t-1} = \alpha X_t + R_{i,t} + \log\beta + \frac{\sigma_i^2}{2}, \quad i = 1, \dots, N. \tag{118.9}$$

Then $E_{t-1}[V_{i,t}] = 0$, where $\sigma_i^2 = \mathrm{Var}_{t-1}(\alpha X_t + R_{it})$ is constant under the stationarity and Gaussian assumptions.

Setting $E_{t-1}[V_{i,t}] = 0$ gives the conditional mean restriction

$$E_{t-1}[R_{i,t}] = -\alpha\, E_{t-1}[X_t] - \log\beta - \frac{\sigma_i^2}{2}. \tag{118.10}$$

Equation (118.10) is the central result of Hansen and Singleton [1983].

The predictable component of each asset's log return is proportional to the predictable component of log consumption growth, with proportionality factor $-\alpha$.

The intercept absorbs the discount factor β and the lognormal variance term $\sigma_i^2/2$.

Let's consider three special cases to better understand the implications of (118.10):

- Risk neutrality ($\alpha = 0$): Each asset's log return equals a constant plus a serially uncorrelated error, so returns are serially uncorrelated.
- Log utility ($\alpha = -1$): The difference $R_{it} - X_t$ has no time-varying predictable component, so returns and consumption growth share the same predictable component.
- Risk aversion ($\alpha < 0$): The time-varying part of $E_{t-1}[R_{it}]$ equals $-\alpha$ times $E_{t-1}[X_t]$, so a larger $|\alpha|$ amplifies the sensitivity of expected returns to expected consumption growth.

For a given consumption–return covariance structure, higher relative risk aversion ($-\alpha$) widens the gap between risky-asset returns and the risk-free return.

The equity premium puzzle arises because the observed spread is large but estimated $|\alpha|$ is small as we will see soon in our estimation.

118.4 The restricted system and its likelihood

To build a likelihood, we need to parameterize the conditional expectation $E_{t-1}[X_t]$.

In the single-return case, write $\mathbf{Y}_t = (X_t, R_t)^\top$ and assume that the predictable component of X_t is a finite-order linear function of past observations:

$$E(X_t \mid \psi_{t-1}) = \mathbf{a}(L)^\top \mathbf{Y}_{t-1} + \mu_x, \tag{118.11}$$

where $\mathbf{a}(L)$ is a vector of lag polynomial coefficients in past (X, R) and μ_x is a constant.

The consumption-growth equation is unrestricted, so X_t depends freely on its own lags and on lagged returns.

The return equation, however, is restricted by the Euler equation.

Combining (118.10) with (118.11) forces the predictable part of R_t to be $-\alpha$ times the predictable part of X_t, plus a constant.

This gives the restricted system

$$\mathbf{A}_0 \mathbf{Y}_t = \mathbf{A}_1(L)\mathbf{Y}_{t-1} + \mu + \mathbf{V}_t, \tag{118.12}$$

with

$$\mathbf{A}_0 = \begin{bmatrix} 1 & 0 \\ \alpha & 1 \end{bmatrix}, \quad \mathbf{A}_1(L) = \begin{bmatrix} \mathbf{a}(L)^\top \\ 0 \end{bmatrix}, \quad \mu = \begin{bmatrix} \mu_x \\ -\log\beta - \sigma_U^2/2 \end{bmatrix}, \tag{118.13}$$

where $\sigma_U^2 := \mathrm{Var}_{t-1}(\alpha X_t + R_t) = \alpha^2\sigma_{XX} + \sigma_{RR} + 2\alpha\sigma_{XR}$ under conditional homoskedasticity.

The sign in the second element of μ follows directly from (118.10).

The innovations $\mathbf{V}_t$ are assumed to be Gaussian with density $f_V(\mathbf{v})$.

Since $\mathbf{V}_t = \mathbf{A}_0\mathbf{Y}_t - \mathbf{A}_1(L)\mathbf{Y}_{t-1} - \mu$, the observables $\mathbf{Y}_t$ are a linear transformation of $\mathbf{V}_t$ with $\mathbf{Y}_t = \mathbf{A}_0^{-1}(\mathbf{V}_t + \mathbf{A}_1(L)\mathbf{Y}_{t-1} + \mu)$.

The change-of-variables formula for densities gives

$$f_Y(\mathbf{y}_t \mid \mathbf{Y}_{t-1}) = f_V(\mathbf{A}_0\mathbf{y}_t - \mathbf{A}_1(L)\mathbf{Y}_{t-1} - \mu) \left|\det\left(\frac{\partial \mathbf{V}_t}{\partial \mathbf{Y}_t}\right)\right| = f_V(\mathbf{v}_t) \,|\det(\mathbf{A}_0)|.$$

Because $\mathbf{A}_0$ is unit lower triangular, $\det(\mathbf{A}_0) = 1$, so the Jacobian term disappears and the log-likelihood of $\mathbf{Y}_t$ equals the log-likelihood of $\mathbf{V}_t$ evaluated at the residuals.

Since the Jacobian is unity, the conditional density of $\mathbf{Y}_t$ is the Gaussian density of $\mathbf{V}_t$ evaluated at $\mathbf{v}_t = \mathbf{A}_0\mathbf{Y}_t - \mathbf{A}_1(L)\mathbf{Y}_{t-1} - \mu$.

For a single observation, $\mathbf{V}_t \sim N(\mathbf{0}, \Sigma)$ has density

$$f_V(\mathbf{v}_t) = (2\pi)^{-1}|\Sigma|^{-1/2} \exp\!\left(-\tfrac{1}{2}\,\mathbf{v}_t^\top \Sigma^{-1}\mathbf{v}_t\right).$$

Taking logarithms gives

$$\ell_t(\theta) = -\log(2\pi) - \tfrac{1}{2}\log|\Sigma| - \tfrac{1}{2}\,\mathbf{v}_t^\top\Sigma^{-1}\,\mathbf{v}_t.$$

Summing over T observations and dropping the constant $-T\log(2\pi)$ yields the log-likelihood function

$$L(\theta) = -\frac{T}{2}\log|\Sigma| - \frac{1}{2}\sum_{t=1}^{T}(\mathbf{A}_0\mathbf{Y}_t - \mathbf{A}_1(L)\mathbf{Y}_{t-1} - \mu)^\top\Sigma^{-1}(\mathbf{A}_0\mathbf{Y}_t - \mathbf{A}_1(L)\mathbf{Y}_{t-1} - \mu), \tag{118.14}$$

where Σ is the covariance matrix of the innovation $\mathbf{V}_t$ and θ collects all free parameters including α, β, the covariance parameters, the first-row intercept μ_x, and the first-row lag coefficients.

The restrictions imposed by (118.6) enter (118.14) through the structure of $\mathbf{A}_0$, $\mathbf{A}_1(L)$, and μ in (118.13).

The second row of (118.12) has no free lag coefficients and its intercept is determined by α, β, and Σ.

An alternative unrestricted bivariate VAR(p) would estimate both rows of (118.12) freely.

Each row has 1 intercept plus $2p$ lag coefficients (p lags $\times$ 2 variables), giving $2(1 + 2p)$ mean parameters.

Adding 3 free covariance parameters ($\sigma_{XX}, \sigma_{RR}, \sigma_{XR}$) gives a total of $5 + 4p$.

The restricted system (118.12) has only $6 + 2p$ free parameters: the first row contributes $1 + 2p$ (its intercept μ_x and $2p$ lag coefficients), plus α, β, and the 3 covariance parameters.

The second row adds nothing because its lag structure and intercept are pinned down by α, β, and Σ via (118.10).

The difference $(5 + 4p) - (6 + 2p) = 2p - 1$ gives the degrees of freedom for the likelihood ratio tests we will perform soon.

118.5 Likelihood implementation

Now let's implement the likelihood (118.14).

The building blocks are:

- a function to construct lagged data matrices $(\mathbf{Y}_t, \mathbf{Y}_{t-1}, \dots, \mathbf{Y}_{t-p})$,
- a function to map the parameter vector into the matrices $\mathbf{A}_0$, $\mathbf{A}_1$, μ, Σ,
- a function to compute the restricted-system residuals $\mathbf{V}_t = \mathbf{A}_0 \mathbf{Y}_t - \mathbf{A}_1(L)\mathbf{Y}_{t-1} - \mu$, and
- the Gaussian log-likelihood itself.

Since we are working with log-transformed data, we define a helper for the transformation below

```
def to_mle_array(data):
    valid = (data[:, 0] > 0.0) & (data[:, 1] > 0.0)
    return np.column_stack(
        [np.log(data[valid, 1]), np.log(data[valid, 0])])
```

First we build the lagged data matrices, which are the inputs to the likelihood function

```
def build_lagged_data(data, n_lags):
    """
    Build Y_t and lag stacks [Y_{t-1}, ..., Y_{t-p}] for bivariate data.
    """
    if data.ndim != 2 or data.shape[1] != 2:
        raise ValueError("data must be T x 2.")
    if data.shape[0] <= n_lags:
        raise ValueError("Sample size must exceed n_lags.")

    t_obs = data.shape[0]
    n_obs = t_obs - n_lags
    y_t = data[n_lags:, :]
    y_lag = np.empty((n_obs, 2 * n_lags))

    for lag in range(1, n_lags + 1):
        y_lag[:, 2 * (lag - 1) : 2 * lag] = data[n_lags - lag : t_obs - lag, :]

    return y_t, y_lag
```

Next, we validate and unpack the parameter vector while enforcing feasibility conditions

```
def unpack_parameters(params, n_lags):
    """
    Validate and unpack parameter vector.
    """
    if len(params) != 6 + 2 * n_lags:
        return None
```

(continues on next page)

(continued from previous page)

```
    α, β, σ_x, σ_r, cov_xr, μ_x = params[:6]
    a_lags = params[6:]

    tol = 1e-8
    if not np.isfinite(α):
        return None
    if not (tol < β):
        return None
    if not (σ_x > tol and σ_r > tol):
        return None

    Σ = np.array(
        [
            [σ_x ** 2, cov_xr],
            [cov_xr, σ_r ** 2],
        ]
    )

    try:
        cholesky(Σ, lower=True)
    except (LinAlgError, ValueError):
        return None

    return {
        "α": np.array(α),
        "β": np.array(β),
        "σ_x": np.array(σ_x),
        "σ_r": np.array(σ_r),
        "cov_xr": np.array(cov_xr),
        "μ_x": np.array(μ_x),
        "a_lags": a_lags,
    }
```

The next step maps parameters and lagged data into restricted-system residuals

```
def restricted_residuals(
    params,
    y_t,
    y_lag,
    n_lags,
):
    """
    Compute V_t implied by A0 Y_t - A1 Y_{t-1} - mu.
    """
    parsed = unpack_parameters(params, n_lags)
    if parsed is None:
        return None

    α = float(parsed["α"])
    β = float(parsed["β"])
    σ_x = float(parsed["σ_x"])
    σ_r = float(parsed["σ_r"])
    cov_xr = float(parsed["cov_xr"])
    μ_x = float(parsed["μ_x"])
    a_lags = np.asarray(parsed["a_lags"])
```

(continues on next page)

(continued from previous page)

```
    A0 = np.array([[1.0, 0.0], [α, 1.0]])
    A1 = np.zeros((2, 2 * n_lags))
    A1[0, :] = a_lags
    σ_u2 = α ** 2 * σ_x ** 2 + σ_r ** 2 + 2.0 * α * cov_xr
    μ = np.array([μ_x, -np.log(β) - 0.5 * σ_u2])

    resid = y_t @ A0.T - y_lag @ A1.T - μ[None, :]
    if np.any(np.abs(resid) > 1e10):
        return None
    return resid
```

The recursive structure also lets us simulate data from the model.

We can generate data from the model by drawing innovations $\mathbf{V}_t \sim N(\mathbf{0}, \Sigma)$ and compute $\mathbf{Y}_t = \mathbf{A}_0^{-1}(\mathbf{A}_1(L)\mathbf{Y}_{t-1} + \mu + \mathbf{V}_t)$ forward in time.

This is useful for checking the likelihood implementation through Monte Carlo

```
def simulate_restricted_var(
    params,
    n_obs,
    n_lags,
    burn_in=200,
    seed=0,
):
    """
    Simulate [log consumption growth, log return] from the restricted model.
    """
    if seed is not None:
        np.random.seed(seed)

    if len(params) != 6 + 2 * n_lags:
        raise ValueError("Parameter vector length must be 6 + 2 * n_lags.")

    α, β, σ_x, σ_r, cov_xr, μ_x = params[:6]
    a_lags = params[6:]

    Σ_e = np.array(
        [
            [σ_x ** 2, cov_xr],
            [cov_xr, σ_r ** 2],
        ]
    )

    A0 = np.array([[1.0, 0.0], [α, 1.0]])
    Σ_v = A0 @ Σ_e @ A0.T

    eigvals = np.linalg.eigvals(Σ_v)
    if np.min(eigvals) <= 0.0:
        Σ_v += np.eye(2) * 1e-6

    A1 = np.zeros((2, 2 * n_lags))
    A1[0, :] = a_lags
    σ_u2 = α ** 2 * σ_x ** 2 + σ_r ** 2 + 2.0 * α * cov_xr
    μ = np.array([μ_x, -np.log(β) - 0.5 * σ_u2])

    total_n = n_obs + burn_in
```

(continues on next page)

(continued from previous page)

```
    y = np.zeros((total_n, 2))

    for t in range(n_lags, total_n):
        lag_stack = []
        for lag in range(1, n_lags + 1):
            lag_stack.append(y[t - lag, :])
        lag_vec = np.concatenate(lag_stack)
        shock = np.random.multivariate_normal(np.zeros(2), Σ_v)
        y[t, :] = np.linalg.solve(A0, A1 @ lag_vec + μ + shock)

    return y[burn_in:, :]
```

Next, we encode the Gaussian log-likelihood implied by the residual covariance matrix

```
def log_likelihood_mle(
    params,
    y_t,
    y_lag,
    n_lags,
    include_const=True,
):
    """
    Evaluate Gaussian log-likelihood for the restricted system.
    """
    parsed = unpack_parameters(params, n_lags)
    if parsed is None:
        return -np.inf

    resid = restricted_residuals(params, y_t, y_lag, n_lags)
    if resid is None:
        return -np.inf

    α = float(parsed["α"])
    σ_x = float(parsed["σ_x"])
    σ_r = float(parsed["σ_r"])
    cov_xr = float(parsed["cov_xr"])

    Σ_e = np.array(
        [
            [σ_x ** 2, cov_xr],
            [cov_xr, σ_r ** 2],
        ]
    )

    A0 = np.array([[1.0, 0.0], [α, 1.0]])
    Σ_v = A0 @ Σ_e @ A0.T

    try:
        chol = cholesky(Σ_v, lower=True)
        log_det = 2.0 * np.sum(np.log(np.diag(chol) + 1e-16))
        std_resid = solve_triangular(chol, resid.T, lower=True).T
        quad_form = np.sum(std_resid ** 2)
    except (LinAlgError, ValueError):
        return -np.inf

    sample_size = y_t.shape[0]
    ll = -0.5 * sample_size * log_det - 0.5 * quad_form
```

(continues on next page)

(continued from previous page)

```
    if include_const:
        ll -= sample_size * np.log(2.0 * np.pi)

    if np.isnan(ll) or np.isinf(ll):
        return -np.inf
    return float(ll)
```

To pass this into a numerical optimizer, we wrap the log-likelihood as a minimization objective

```
def negative_log_likelihood(
    params,
    y_t,
    y_lag,
    n_lags,
):
    """
    Return negative log-likelihood for minimization.
    """
    ll = log_likelihood_mle(params, y_t, y_lag, n_lags, include_const=False)
    if np.isfinite(ll):
        return -ll
    return 1e20
```

We set parameter bounds and generate data-driven starting values for multi-start optimization.

We keep β positive but do not force $\beta < 1$ in estimation

```
def parameter_bounds(n_lags):
    """
    Bounds for optimization.
    """
    bounds = [
        (-200.0, 200.0),    # α (= -risk aversion)
        (1e-8, 2.0),      # β (discount factor)
        (1e-8, None),     # σ_x (std dev of consumption innovation)
        (1e-8, None),     # σ_r (std dev of return innovation)
        (None, None),     # cov_xr (covariance)
        (None, None),     # μ_x (consumption growth intercept)
    ]
    bounds += [(-0.99, 0.99)] * (2 * n_lags)  # VAR lag coefficients
    return bounds
```

We use multiple starting vectors to help local solvers escape poor initializations

```
def starting_values(y_t, y_lag, n_lags, n_starts=10):
    """
    Generate multiple starting values.
    """
    rng = np.random.default_rng(123)
    starts = []
    n_params = 6 + 2 * n_lags

    base = np.zeros(n_params)
    base[0] = -0.5
    base[1] = 0.999
    base[2] = max(float(np.std(y_t[:, 0])), 1e-3)
    base[3] = max(float(np.std(y_t[:, 1])), 1e-3)
```

(continues on next page)

(continued from previous page)

```
    base[4] = float(np.cov(y_t.T)[0, 1])
    base[5] = float(np.mean(y_t[:, 0]))
    base[6:] = 0.1
    starts.append(base.copy())

    # OLS seed from unrestricted VAR
    n_obs = y_t.shape[0]
    x = np.column_stack([np.ones(n_obs), y_lag])
    coef = np.linalg.lstsq(x, y_t, rcond=None)[0]
    resid = y_t - x @ coef
    Σ_e = resid.T @ resid / max(1, n_obs)

    a_lags_ols = coef[1:, 0]
    r_lags_ols = coef[1:, 1]
    denom = float(a_lags_ols @ a_lags_ols)
    if denom > 1e-10:
        α_ols = -float((a_lags_ols @ r_lags_ols) / denom)
    else:
        α_ols = -0.5

    μ_x_ols = float(coef[0, 0])
    μ_r_ols = float(coef[0, 1])
    σ_x_ols = float(np.sqrt(max(Σ_e[0, 0], 1e-8)))
    σ_r_ols = float(np.sqrt(max(Σ_e[1, 1], 1e-8)))
    cov_xr_ols = float(Σ_e[0, 1])
    σ_u2_ols = (
        α_ols ** 2 * σ_x_ols ** 2
        + σ_r_ols ** 2
        + 2.0 * α_ols * cov_xr_ols
    )
    β_ols = float(np.exp(-(μ_r_ols + α_ols * μ_x_ols + 0.5 * σ_u2_ols)))
    β_ols = float(np.clip(β_ols, 1e-6, 2.0))

    ols_seed = np.zeros(n_params)
    ols_seed[0] = α_ols
    ols_seed[1] = β_ols
    ols_seed[2] = σ_x_ols
    ols_seed[3] = σ_r_ols
    ols_seed[4] = cov_xr_ols
    ols_seed[5] = μ_x_ols
    ols_seed[6:] = a_lags_ols
    starts.append(ols_seed.copy())

    seeds = [base, ols_seed]
    while len(starts) < n_starts:
        seed = seeds[len(starts) % len(seeds)]
        trial = seed.copy()
        trial[:2] += rng.normal(0.0, 0.2, 2)
        trial[2:6] *= 1.0 + rng.normal(0.0, 0.15, 4)
        trial[6:] += rng.normal(0.0, 0.08, 2 * n_lags)
        trial[1] = max(trial[1], 1e-6)
        trial[2] = max(trial[2], 1e-6)
        trial[3] = max(trial[3], 1e-6)
        starts.append(trial)

    return starts
```

Standard errors come from an outer-product-of-gradients (OPG) approximation to the information matrix, computed by finite differences of per-observation log-likelihood contributions.

This tends to be more numerically stable than finite-difference Hessians in this application.

```
def log_likelihood_contributions(
    params,
    y_t,
    y_lag,
    n_lags,
    include_const=False,
):
    """
    Vector of per-observation Gaussian log-likelihood contributions.

    Returns an array of length T, or None if the parameter vector is infeasible.
    """
    parsed = unpack_parameters(params, n_lags)
    if parsed is None:
        return None

    resid = restricted_residuals(params, y_t, y_lag, n_lags)
    if resid is None:
        return None

    α = float(parsed["α"])
    σ_x = float(parsed["σ_x"])
    σ_r = float(parsed["σ_r"])
    cov_xr = float(parsed["cov_xr"])

    Σ_e = np.array(
        [
            [σ_x ** 2, cov_xr],
            [cov_xr, σ_r ** 2],
        ]
    )
    A0 = np.array([[1.0, 0.0], [α, 1.0]])
    Σ_v = A0 @ Σ_e @ A0.T

    try:
        chol = cholesky(Σ_v, lower=True)
        log_det = 2.0 * np.sum(np.log(np.diag(chol) + 1e-16))
        std_resid = solve_triangular(chol, resid.T, lower=True).T
    except (LinAlgError, ValueError):
        return None

    quad = np.sum(std_resid ** 2, axis=1)
    ll_t = -0.5 * log_det - 0.5 * quad
    if include_const:
        ll_t -= np.log(2.0 * np.pi)
    if not np.all(np.isfinite(ll_t)):
        return None
    return ll_t


def opg_standard_errors(
    params,
    y_t,
```

(continues on next page)

(continued from previous page)

```
    y_lag,
    n_lags,
    step=1e-6,
    max_step_shrink=12,
    eig_floor=1e-12,
):
    """
    Standard errors via OPG approximation to the information matrix.
    """
    n = len(params)
    ll0 = log_likelihood_contributions(params, y_t, y_lag, n_lags, include_
↪const=False)
    if ll0 is None:
        return np.full(n, np.nan)

    n_obs = int(ll0.shape[0])
    scores = np.empty((n_obs, n))

    for i in range(n):
        base_step = step * (abs(params[i]) + 1.0)
        hi = base_step
        ll_plus = None
        ll_minus = None

        for _ in range(max_step_shrink + 1):
            p_plus = params.copy()
            p_minus = params.copy()
            p_plus[i] += hi
            p_minus[i] -= hi
            ll_plus = log_likelihood_contributions(
                p_plus, y_t, y_lag, n_lags, include_const=False
            )
            ll_minus = log_likelihood_contributions(
                p_minus, y_t, y_lag, n_lags, include_const=False
            )
            if ll_plus is not None and ll_minus is not None:
                break
            hi *= 0.5

        if ll_plus is None or ll_minus is None:
            return np.full(n, np.nan)

        scores[:, i] = (ll_plus - ll_minus) / (2.0 * hi)

    if not np.all(np.isfinite(scores)):
        return np.full(n, np.nan)

    # Center scores to mitigate numerical drift away
    scores = scores - scores.mean(axis=0, keepdims=True)

    opg = scores.T @ scores
    if not np.all(np.isfinite(opg)):
        return np.full(n, np.nan)
    opg = 0.5 * (opg + opg.T)

    try:
        eigvals, eigvecs = np.linalg.eigh(opg)
```

(continues on next page)

(continued from previous page)

```
    except (LinAlgError, ValueError):
        return np.full(n, np.nan)

    floor = float(eig_floor) * max(1.0, float(np.max(eigvals)))
    eigvals = np.clip(eigvals, floor, None)
    cov = eigvecs @ np.diag(1.0 / eigvals) @ eigvecs.T
    se = np.sqrt(np.maximum(np.diag(cov), 0.0))
    se[~np.isfinite(se)] = np.nan
    return se
```

The multi-start MLE estimator below combines these pieces and returns parameters, fit criteria, and residuals

```
def estimate_mle(data, n_lags, verbose=False):
    """
    Estimate the restricted model by multi-start local optimization.
    """
    y_t, y_lag = build_lagged_data(data, n_lags)
    bounds = parameter_bounds(n_lags)
    starts = starting_values(y_t, y_lag, n_lags)

    best_result = None
    best_ll = -np.inf

    for i, x0 in enumerate(starts):
        try:
            result = minimize(
                negative_log_likelihood,
                x0=x0,
                args=(y_t, y_lag, n_lags),
                method="L-BFGS-B",
                bounds=bounds,
            )
        except Exception:
            continue

        if np.isfinite(result.fun):
            ll_val = log_likelihood_mle(result.x, y_t, y_lag, n_lags)
            if np.isfinite(ll_val) and ll_val > best_ll:
                best_ll = ll_val
                best_result = result
                if verbose:
                    print(
                        f"start={i}, success={result.success}, "
                        f"status={result.status}, loglike={ll_val:.2f}"
                    )

    n_params = 6 + 2 * n_lags

    if best_result is None:
        return {
            "params": np.full(n_params, np.nan),
            "se": np.full(n_params, np.nan),
            "loglike": -np.inf,
            "converged": False,
            "optimizer_success": False,
            "residuals": None,
            "n_obs": int(y_t.shape[0]),
```

(continues on next page)

(continued from previous page)

```
        }

    params = best_result.x
    se = opg_standard_errors(params, y_t, y_lag, n_lags)
    resid = restricted_residuals(params, y_t, y_lag, n_lags)
    ll_val = log_likelihood_mle(params, y_t, y_lag, n_lags)

    return {
        "params": params,
        "se": se,
        "loglike": ll_val,
        "converged": bool(np.isfinite(ll_val)),
        "optimizer_success": bool(best_result.success),
        "residuals": resid,
        "n_obs": int(y_t.shape[0]),
    }
```

Residual diagnostics below summarize normality and serial-correlation checks

```
def residual_diagnostics(resid):
    """
    Compute basic residual diagnostics.
    """
    out = {}

    for i, label in enumerate(["consumption", "return"]):
        jb_stat, jb_pval = stats.jarque_bera(resid[:, i])
        out[f"{label}_jb_stat"] = float(jb_stat)
        out[f"{label}_jb_pval"] = float(jb_pval)
        out[f"{label}_dw"] = float(durbin_watson(resid[:, i]))

    return out
```

Finally, a lag-loop wrapper runs MLE across different lag lengths and collects results in a DataFrame

```
def run_mle_by_lag(
    data,
    lags=(2, 4, 6),
    verbose=False,
):
    """
    Estimate restricted MLE models by lag length.
    """
    rows = []
    fits = {}

    for lag in lags:
        fit = estimate_mle(data, n_lags=lag, verbose=verbose)
        fits[lag] = fit

        rows.append(
            {
                "NLAG": lag,
                "α_hat": fit["params"][0],
                "se_α": fit["se"][0],
                "β_hat": fit["params"][1],
                "se_β": fit["se"][1],
```

(continues on next page)

(continued from previous page)

```
                "loglike": fit["loglike"],
                "n_obs": fit["n_obs"],
            }
        )

    table = pd.DataFrame(rows).set_index("NLAG")
    return table, fits
```

118.5.1 Simulation

Before applying the likelihood to real data, we verify that it recovers known parameters from simulated data generated by the restricted system.

For `n_lags = 1`, the parameter vector is

$$\theta = (\alpha,\ \beta,\ \sigma_x,\ \sigma_r,\ \sigma_{xr},\ \mu_x,\ a_{x,1},\ a_{r,1}),$$

where the last two entries are the coefficients on X_{t-1} and R_{t-1} in the first-row regression for X_t.

More generally, for `n_lags = p` we pack `a_lags` in the order $[a_{x,1}, a_{r,1}, \dots, a_{x,p}, a_{r,p}]$, so the full parameter vector has length $6 + 2p$.

The covariance parameters $(\sigma_x, \sigma_r, \sigma_{xr})$ describe the reduced-form shocks to (X_t, R_t): if $\varepsilon_t = (\varepsilon_{x,t}, \varepsilon_{r,t})^\top \sim N(0, \Sigma_\varepsilon)$, then $\Sigma_\varepsilon = \begin{bmatrix} \sigma_x^2 & \sigma_{xr} \\ \sigma_{xr} & \sigma_r^2 \end{bmatrix}$.

The restricted-system innovation is $\mathbf{V}_t = \mathbf{A}_0 \varepsilon_t$, so its covariance is $\Sigma_V = \mathbf{A}_0 \Sigma_\varepsilon \mathbf{A}_0^\top$, which is what enters the likelihood.

In the simulation recursion we draw $\mathbf{V}_t \sim N(0, \Sigma_V)$ and compute $\mathbf{Y}_t = \mathbf{A}_0^{-1}(\mathbf{A}_1(L)\mathbf{Y}_{t-1} + \mu + \mathbf{V}_t)$ forward.

The following table compares the true parameters to the MLE estimates from a large simulated sample

```
α_true = -1.00
β_true = 0.993
σ_x_true = 0.015
σ_r_true = 0.020
cov_xr_true = 0.0001
μ_x_true = 0.002
a_x1_true = 0.40
a_r1_true = 0.10

true_params = np.array(
    [
        α_true,
        β_true,
        σ_x_true,
        σ_r_true,
        cov_xr_true,
        μ_x_true,
        a_x1_true,
        a_r1_true,
    ]
)

sim_mle_data = simulate_restricted_var(
    params=true_params,
```

(continues on next page)

(continued from previous page)

```
    n_obs=50000,
    n_lags=1,
    burn_in=5000,
    seed=0,
)

fit_sim = estimate_mle(sim_mle_data, n_lags=1, verbose=False)
```

	true	estimate	se	t (H_0: true)
α	-1.0000	-0.9934	0.011671	0.56
β	0.9930	0.9928	0.000110	-1.57

The point estimates are close to the true parameters, and the t-statistics for the null hypothesis that the true value is correct are small in magnitude, consistent with sampling variation.

118.6 Preference parameters and likelihood ratio tests

The restricted system embeds preference parameters through cross-equation restrictions.

The parameter α links predictable variation in returns to predictable variation in consumption growth.

Under the model, the return equation's dependence on lagged variables is entirely determined by $-\alpha$ times the consumption equation's lag coefficients.

The parameter β shifts the return-equation intercept through $-\log\beta - \sigma_U^2/2$.

Hansen and Singleton [1983] test these restrictions by comparing the restricted system to an unrestricted bivariate VAR estimated on the same sample.

If the restrictions imposed by the Euler-equation are correct, the restricted model should fit nearly as well as the unrestricted model.

The standard test is the likelihood ratio statistic

$$LR = 2(\ell_u - \ell_r) \Rightarrow \chi^2_d, \tag{118.15}$$

where ℓ_u and ℓ_r are the maximized log-likelihoods of the unrestricted and restricted models, and d is the difference in the number of free parameters.

Both likelihoods must be evaluated on the same effective sample for the LR distribution to be valid.

Hansen and Singleton [1983] report that for the value-weighted aggregate stock return, the χ^2 test statistics provide little evidence against the model.

In the tables below we report both `chi2.cdf(LR, df)` (corresponding to what HS report in parentheses) and the usual right-tail `p(LR) = 1 - chi2.cdf`

```
def likelihood_ratio_test(
    fit_restricted,
    fit_unrestricted,
    df_diff,
):
    """
    Compare nested specifications with an LR test.
    """
```

(continues on next page)

(continued from previous page)

```
    if not (fit_restricted["converged"] and fit_unrestricted["converged"]):
        return {"lr_stat": np.nan, "p_value": np.nan, "chi2_cdf": np.nan}
    if fit_restricted.get("n_obs") != fit_unrestricted.get("n_obs"):
        return {"lr_stat": np.nan, "p_value": np.nan, "chi2_cdf": np.nan}

    lr_stat = 2.0 * (fit_unrestricted["loglike"] - fit_restricted["loglike"])
    chi2_cdf = stats.chi2.cdf(lr_stat, df=df_diff)
    p_value = 1.0 - chi2_cdf
    return {
        "lr_stat": float(lr_stat),
        "p_value": float(p_value),
        "chi2_cdf": float(chi2_cdf),
    }
```

The unrestricted benchmark is a Gaussian VAR for $\mathbf{Y}_t = (X_t, R_t)^\top$ with free coefficients on all lags in both equations.

```
def estimate_unrestricted_var(data, n_lags):
    """
    Estimate an unrestricted Gaussian VAR for Y_t = [X_t, R_t].
    """
    y_t, y_lag = build_lagged_data(data, n_lags)
    n_obs = y_t.shape[0]
    x = np.column_stack([np.ones(n_obs), y_lag])
    coef = np.linalg.lstsq(x, y_t, rcond=None)[0]
    resid = y_t - x @ coef
    Σ = resid.T @ resid / n_obs

    try:
        chol = cholesky(Σ, lower=True)
        log_det = 2.0 * np.sum(np.log(np.diag(chol) + 1e-16))
        std_resid = solve_triangular(chol, resid.T, lower=True).T
        quad_form = np.sum(std_resid ** 2)
    except (LinAlgError, ValueError):
        return {
            "coef": coef,
            "Σ": np.full((2, 2), np.nan),
            "residuals": resid,
            "loglike": -np.inf,
            "converged": False,
            "n_obs": int(n_obs),
        }

    d = y_t.shape[1]
    loglike = float(-0.5 * n_obs * d * np.log(2.0 * np.pi)
            - 0.5 * n_obs * log_det - 0.5 * quad_form)

    return {
        "coef": coef,
        "Σ": Σ,
        "residuals": resid,
        "loglike": loglike,
        "converged": True,
        "n_obs": int(n_obs),
    }
```

```
def run_unrestricted_var_by_lag(data, lags=(2, 4, 6)):
```

(continues on next page)

(continued from previous page)

```
    """
    Estimate unrestricted VAR models by lag length.
    """
    rows = []
    fits = {}

    for lag in lags:
        fit = estimate_unrestricted_var(data, n_lags=lag)
        fits[lag] = fit
        rows.append(
            {
                "NLAG": lag,
                "loglike": fit["loglike"],
                "n_obs": fit["n_obs"],
            }
        )

    table = pd.DataFrame(rows).set_index("NLAG")
    return table, fits
```

The following function computes the LR statistic at each lag length, replicating the testing strategy of Table 1 in Hansen and Singleton [1983].

```
def restricted_vs_unrestricted_lr(mle_fits, unrestricted_fits, lags=(2, 4, 6)):
    """
    Compute LR tests of restricted model versus unrestricted VAR.
    """
    rows = []

    for lag in lags:
        fit_r = mle_fits[lag]
        fit_u = unrestricted_fits[lag]
        df_diff = (2 * (1 + 2 * lag) + 3) - (6 + 2 * lag)
        lr = likelihood_ratio_test(fit_restricted=fit_r, fit_unrestricted=fit_u, df_
↪diff=df_diff)
        rows.append(
            {
                "NLAG": lag,
                "lr_stat": lr["lr_stat"],
                "p_value": lr["p_value"],
                "chi2_cdf": lr["chi2_cdf"],
                "df": df_diff,
                "T": fit_r.get("n_obs", np.nan),
            }
        )

    return pd.DataFrame(rows).set_index("NLAG")
```

118.7 Predictability and the R-squared restriction

Section II of Hansen and Singleton [1983] emphasizes an implication of the restricted system for return predictability.

From (118.10) and (118.11), the predictable component of the log return is

$$E(R_t \mid \psi_{t-1}) = -\alpha\, E(X_t \mid \psi_{t-1}) - \log\beta - \frac{\sigma_U^2}{2}. \tag{118.16}$$

Since the predictable return is a linear function of the predictable consumption growth, the variance of the predictable return component is exactly α^2 times the variance of the predictable consumption-growth component:

$$\operatorname{Var}[E(R_t \mid \psi_{t-1})] = \alpha^2 \operatorname{Var}[E(X_t \mid \psi_{t-1})]. \tag{118.17}$$

Hansen and Singleton [1983] derive the implied R^2 of the return projection onto ψ_{t-1}:

$$R_R^2 = \frac{\alpha^2 \operatorname{Var}[E(X_t \mid \psi_{t-1})]}{\operatorname{Var}(R_t \mid \psi_{t-1}) + \alpha^2 \operatorname{Var}[E(X_t \mid \psi_{t-1})]}. \tag{118.18}$$

If $\alpha = 0$ (risk neutrality), then $R_R^2 = 0$ and asset returns are unpredictable.

If $\alpha = -1$ (log utility), then $R_t - X_t$ is unpredictable, so returns and consumption growth share the same predictable component.

More generally, the R_R^2 for returns will be small whenever the variance of the unpredictable return component $\operatorname{Var}(R_t \mid \psi_{t-1})$ is large relative to the predictable variance, which is the case for stock returns.

The function below reports:

- restriction-side predictable-variance terms implied by the Euler equation, and
- R_X^2 and R_R^2 from the unrestricted VAR.

```
def predictability_metrics(data, restricted_fit, unrestricted_fit, n_lags):
    """
    Compute predictable-component metrics and unrestricted VAR R^2 values.
    """
    y_t, y_lag = build_lagged_data(data, n_lags)
    parsed = unpack_parameters(restricted_fit["params"], n_lags)
    α = float(parsed["α"])
    β = float(parsed["β"])
    σ_x = float(parsed["σ_x"])
    σ_r = float(parsed["σ_r"])
    cov_xr = float(parsed["cov_xr"])
    μ_x = float(parsed["μ_x"])
    a_lags = np.asarray(parsed["a_lags"])

    pred_x = y_lag @ a_lags + μ_x
    σ_u2 = α ** 2 * σ_x ** 2 + σ_r ** 2 + 2.0 * α * cov_xr
    pred_r = -α * pred_x - np.log(β) - 0.5 * σ_u2

    x = y_t[:, 0]
    r = y_t[:, 1]
    resid_x = x - pred_x
    resid_r = r - pred_r

    r2_x = 1.0 - np.var(resid_x) / np.var(x)
    r2_r = 1.0 - np.var(resid_r) / np.var(r)
```

(continues on next page)

(continued from previous page)

```
    if unrestricted_fit["converged"] and unrestricted_fit.get("coef") is not None:
        n_obs = y_t.shape[0]
        x_u = np.column_stack([np.ones(n_obs), y_lag])
        pred_u = x_u @ unrestricted_fit["coef"]
        resid_u = y_t - pred_u
        var_x = np.var(y_t[:, 0])
        var_r = np.var(y_t[:, 1])
        r2_x_unres = np.nan if var_x <= 0.0 else 1.0 - np.var(resid_u[:, 0]) / var_x
        r2_r_unres = np.nan if var_r <= 0.0 else 1.0 - np.var(resid_u[:, 1]) / var_r
    else:
        r2_x_unres = np.nan
        r2_r_unres = np.nan

    return {
        "alpha_hat": α,
        "var_pred_x": float(np.var(pred_x)),
        "var_pred_r": float(np.var(pred_r)),
        "alpha2_var_pred_x": float(α ** 2 * np.var(pred_x)),
        "r2_x_restricted": float(r2_x),
        "r2_r_restricted": float(r2_r),
        "r2_x_unrestricted": float(r2_x_unres),
        "r2_r_unrestricted": float(r2_r_unres),
        "T": int(y_t.shape[0]),
    }
```

118.8 Return-difference tests

Hansen and Singleton [1983] also propose tests based on differences in log returns across assets.

From (118.10), the conditional mean of asset i's log return is $E_{t-1}[R_{it}] = -\alpha\, E_{t-1}[X_t] - \log\beta - \sigma_i^2/2$.

The term $-\alpha\, E_{t-1}[X_t] - \log\beta$ is common to all assets, so it cancels in the difference:

$$E_{t-1}[R_{it} - R_{jt}] = \frac{\sigma_j^2 - \sigma_i^2}{2},$$

which is a constant that does not depend on time-$(t-1)$ information.

Return differences should therefore be unpredictable if the model is correct, regardless of the values of α and β.

These tests avoid the need to measure consumption, at the cost of losing the ability to identify α and β.

Hansen and Singleton [1983] report that the return-difference restrictions are strongly rejected for models with multiple stock returns, providing substantial evidence against the CRRA-lognormal specification even when consumption measurement problems are eliminated.

The code below is an illustration of this logic on simulated data.

Reproducing the paper's empirical return-difference tables requires estimating multi-asset systems that are outside this lecture's scope

```
def simulate_multi_asset_nominal_returns(
    n_obs,
    n_assets=3,
    α_true=-1.0,
    β_true=0.993,
```

(continues on next page)

(continued from previous page)

```
    seed=0,
):
    """
    Simulate log nominal returns satisfying E_t[beta * exp(alpha X) * r_i] = 1.
    """
    if n_assets < 2:
        raise ValueError("n_assets must be at least 2.")

    rng = np.random.default_rng(seed)
    x = np.empty(n_obs)
    x[0] = 0.001
    for t in range(1, n_obs):
        x[t] = 0.001 + 0.4 * (x[t - 1] - 0.001) + 0.006 * rng.standard_normal()

    sigmas = np.linspace(0.03, 0.06, n_assets)
    eps = rng.standard_normal((n_obs, n_assets)) * sigmas
    log_returns = -np.log(β_true) - α_true * x[:, None] + eps \
            - 0.5 * sigmas[None, :] ** 2
    return x, log_returns


def return_difference_test(log_returns, n_lags=2):
    """
    Test predictability of pairwise log-return differences.
    """
    if log_returns.ndim != 2 or log_returns.shape[1] < 2:
        raise ValueError("log_returns must be T x m with m >= 2.")
    if log_returns.shape[0] <= n_lags + 1:
        raise ValueError("Sample size must exceed n_lags + 1.")

    t_obs, n_assets = log_returns.shape
    pairs = list(combinations(range(n_assets), 2))
    n_obs = t_obs - n_lags - 1
    z = np.empty((n_obs, 1 + n_assets * n_lags))
    z[:, 0] = 1.0

    for j in range(n_lags):
        z[:, 1 + j * n_assets : 1 + (j + 1) * n_assets] = log_returns[
            n_lags - j : t_obs - 1 - j, :
        ]

    rows = []
    for i, j in pairs:
        y = log_returns[n_lags + 1 :, i] - log_returns[n_lags + 1 :, j]
        coef = np.linalg.lstsq(z, y, rcond=None)[0]
        resid = y - z @ coef
        sigma2 = float((resid @ resid) / max(1, n_obs - z.shape[1]))
        cov = sigma2 * np.linalg.pinv(z.T @ z)
        slopes = coef[1:]
        cov_slopes = cov[1:, 1:]
        stat = float(slopes @ np.linalg.pinv(cov_slopes) @ slopes)
        p_value = float(1.0 - stats.chi2.cdf(stat, df=slopes.shape[0]))
        rows.append(
            {
                "pair": f"{i+1}-{j+1}",
                "wald_chi2": stat,
                "p_value": p_value,
```

(continues on next page)

(continued from previous page)

```
                "mean_spread": float(np.mean(y)),
            }
        )

    return pd.DataFrame(rows).set_index("pair")
```

We run `return_difference_test` on simulated data with $m = 3$ assets, giving $\binom{3}{2} = 3$ pairs.

For each pair, the function regresses the return spread on a constant and `n_lags` lags of all asset returns, then tests whether the slope coefficients are jointly zero using a Wald χ^2 statistic

```
_, sim_log_returns = simulate_multi_asset_nominal_returns(
    n_obs=1500,
    n_assets=3,
    α_true=-1.0,
    β_true=0.993,
    seed=0,
)

spread_test = return_difference_test(sim_log_returns, n_lags=2)
spread_pretty = spread_test.rename(columns={
    "wald_chi2": r"\chi^2", "p_value": "p", "mean_spread": r"\overline{\Delta R}",
})
display_table(spread_pretty, fmt={
    r"\chi^2": "{:.3f}",
    "p": "{:.3f}",
    r"\overline{\Delta R}": "{:.5f}",
})
```

pair	χ^2	p	$\overline{\Delta R}$
$1-2$	9.445	0.150	0.00132
$1-3$	8.338	0.214	0.00154
$2-3$	4.272	0.640	0.00023

Large p-values confirm that return differences are unpredictable in this simulation, exactly as the model predicts when $\alpha = -1$.

118.9 Empirical MLE estimation

We now apply the maximum likelihood estimator of Hansen and Singleton [1983] to real data.

118.9.1 Data

Both this lecture and the companion lecture *Estimating Euler Equations by Generalized Method of Moments* use the same data construction.

Hansen and Singleton [1982] and Hansen and Singleton [1983] use monthly data on real per capita consumption (nondurables) and stock returns from CRSP for the period 1959:2 through 1978:12.

To align with the paper, we set the default sample to 1959:2–1978:12.

You can pass different `start` and `end` dates to study later periods.

This lecture pulls stock-market and one-month bill returns from the Ken French data library (`F-F_Research_Data_Factors`) and constructs gross nominal returns as `1 + (Mkt-RF + RF)/100` for the market and `1 + RF/100` for T-bills.

`Mkt-RF` is the value-weighted return on all CRSP firms minus the risk-free rate, and `RF` is the one-month Treasury bill return (see here for details).

While Hansen-Singleton use CRSP value-weighted NYSE returns, we use the Ken French market factor as the closest open-access alternative.

The consumption series is constructed from consumption of nondurables (`ND`) with the nondurables deflator.

The hidden cell below loads a vendored monthly dataset of returns and consumption series. The data are built from the FRED and Ken French data libraries by the maintenance script at `_static/lecture_specific/hansen_singleton_1983/make_data.py` and read here directly from GitHub.

118.9.2 MLE estimation and predictability summaries

With the data in hand, we can run the MLE estimation and compute the predictability summaries.

Lognormality makes the MLE tractable when the assumption is correct.

As Hansen and Singleton [1983] stress, however, α is estimated with relatively large standard errors, and precise inference on risk aversion cannot be expected from aggregate monthly data alone.

```
lags = (2, 4, 6)

emp_frame, emp_data = get_estimation_data()
```

The following table reports the restricted MLE estimates of $\hat{\alpha}$ and $\hat{\beta}$ by lag length

```
emp_log_data = to_mle_array(emp_data)
mle_table, mle_fits = run_mle_by_lag(
    emp_log_data, lags=lags, verbose=False
)
mle_pretty = mle_table.rename(columns={
    "α_hat": r"\hat{\alpha}", "se_α": r"\mathrm{se}(\hat{\alpha})",
    "β_hat": r"\hat{\beta}", "se_β": r"\mathrm{se}(\hat{\beta})",
    "loglike": "logL", "n_obs": "T",
})
display_table(mle_pretty, fmt={
    r"\hat{\alpha}": "{:.4f}", r"\mathrm{se}(\hat{\alpha})": "{:.4f}",
    r"\hat{\beta}": "{:.4f}", r"\mathrm{se}(\hat{\beta})": "{:.4f}",
    "logL": "{:.1f}", "T": "{:.0f}",
})
```

NLAG	$\hat{\alpha}$	se($\hat{\alpha}$)	$\hat{\beta}$	se($\hat{\beta}$)	logL	T
2	−0.3228	0.7916	0.9976	0.0030	1209.3	237
4	−0.6402	0.6902	0.9980	0.0028	1206.4	235
6	−0.6155	0.6448	0.9979	0.0028	1199.2	233

The table reports $\hat{\alpha}$ and $\hat{\beta}$ by lag length for the sample used in the code cell above.

For comparison, Hansen and Singleton [1983] report $\hat{\alpha}$ values of -0.32 to -1.25 (standard errors 0.65 to 0.83) for the value-weighted return with nondurables consumption.

Our numbers fall into those ranges.

In risk-aversion units, this corresponds to $-\hat{\alpha}$ between 0.32 and 1.25.

We now compute the predictability summaries

```
unres_table, unres_fits = run_unrestricted_var_by_lag(
    emp_log_data,
    lags=lags,
)
```

NLAG	$\hat{\alpha}$	$\mathrm{Var}(\hat{E}[X_t \mid \psi_{t-1}])$	$\mathrm{Var}(\hat{E}[R_t \mid \psi_{t-1}])$	$\hat{\alpha}^2\,\mathrm{Var}(\hat{E}[X_t \mid \psi_{t-1}])$	R^2_X	R^2_R	T
2	−0.3228	0.000014	0.000001	0.000001	0.1630	0.0246	237
4	−0.6402	0.000016	0.000007	0.000007	0.1966	0.0459	235
6	−0.6155	0.000018	0.000007	0.000007	0.2270	0.0602	233

The column $\hat{\alpha}^2\,\mathrm{Var}(\hat{E}[X_t \mid \psi_{t-1}])$ should equal $\mathrm{Var}(\hat{E}[R_t \mid \psi_{t-1}])$ if the restriction (118.17) holds.

The columns R^2_X and R^2_R match the paper convention of reporting the values from the unrestricted VAR projections.

In Hansen and Singleton [1983], R^2_R is small (0.02 to 0.06) even when R^2_X is nontrivial: most stock-return variation is unpredictable.

Our estimates show the same pattern.

We now test the Euler-equation restrictions by comparing the restricted system to an unrestricted VAR.

```
lr_hs83 = restricted_vs_unrestricted_lr(mle_fits, unres_fits, lags=lags)
lr_pretty = lr_hs83.rename(columns={
    "lr_stat": "LR",
    "chi2_cdf": "chi2.cdf(LR,df)",
    "p_value": "p(LR)",
    "df": "df",
    "T": "T",
})
display_table(lr_pretty, fmt={
    "LR": "{:.3f}",
    "chi2.cdf(LR,df)": "{:.3f}",
    "p(LR)": "{:.3f}",
    "df": "{:.0f}",
    "T": "{:.0f}",
})

sig_level = 0.05
rejected_lags = [int(lag) for lag in lr_hs83.index if lr_hs83.loc[lag, "p_value"] <␣
↪sig_level]
not_rejected_lags = [int(lag) for lag in lr_hs83.index if lr_hs83.loc[lag, "p_value"]␣
↪>= sig_level]
print(f"Not rejected at 5%: {not_rejected_lags if not_rejected_lags else 'none'}")
```

NLAG	LR	p(LR)	chi2.cdf(LR,df)	df	T
2	5.977	0.113	0.887	3	237
4	10.023	0.187	0.813	7	235
6	12.519	0.326	0.674	11	233

```
Not rejected at 5%: [2, 4, 6]
```

The LR test does not reject the model for the value-weighted return, consistent with Hansen and Singleton [1983].

118.9.3 Treasury bill estimation

We now repeat the estimation using the 1-month Treasury bill return in place of the stock return.

Hansen and Singleton [1983] find that the model is strongly rejected for Treasury bills (Table 4 of their paper).

Because the nominally risk-free T-bill return is much more predictable than stock returns, the proportionality restriction has more bite, and the LR test has greater power to detect violations

```
tbill_frame, tbill_data = get_tbill_estimation_data()
```

The following table reports the restricted MLE estimates for Treasury bill returns by lag length.

```
tbill_log_data = to_mle_array(tbill_data)
tbill_mle_table, tbill_mle_fits = run_mle_by_lag(
    tbill_log_data, lags=lags, verbose=False
)
```

NLAG	$\hat{\alpha}$	se($\hat{\alpha}$)	$\hat{\beta}$	se($\hat{\beta}$)	logL	T
2	−0.3060	0.0600	0.9996	0.0003	1817.5	237
4	−0.2882	0.0612	0.9996	0.0003	1804.9	235
6	−0.1991	0.0571	0.9996	0.0003	1791.4	233

The LR test below remains the main specification check in the paper

```
tbill_unres_table, tbill_unres_fits = run_unrestricted_var_by_lag(
    tbill_log_data,
    lags=lags,
)

tbill_unres_pretty = tbill_unres_table.rename(columns={"loglike": "logL", "n_obs": "T
↪"})
display_table(tbill_unres_pretty, fmt={
    "logL": "{:.1f}", "T": "{:.0f}",
})
```

NLAG	logL	T
2	1838.8	237
4	1828.2	235
6	1824.8	233

NLAG	$\hat{\alpha}$	Var($\hat{E}[X_t \mid \psi_{t-1}]$)	Var($\hat{E}[R_t \mid \psi_{t-1}]$)	$\hat{\alpha}^2$ Var($\hat{E}[X_t \mid \psi_{t-1}]$)	R^2_X	R^2_R	T
2	−0.3060	0.000019	0.000002	0.000002	0.2309	0.1618	237
4	−0.2882	0.000018	0.000001	0.000001	0.2223	0.1870	235
6	−0.1991	0.000019	0.000001	0.000001	0.2333	0.2560	233

The following table reports the likelihood ratio tests for the Treasury bill model.

```
tbill_lr = restricted_vs_unrestricted_lr(tbill_mle_fits, tbill_unres_fits, lags=lags)
```

NLAG	LR	p(LR)	chi2.cdf(LR,df)	df	T
2	42.503	0.000	1.000	3	237
4	46.617	0.000	1.000	7	235
6	66.656	0.000	1.000	11	233

```
T-bill model rejected at 5% for lags: [2, 4, 6]
```

Hansen and Singleton [1983] find the same qualitative pattern in their 1959–1978 sample: the Treasury bill model is rejected much more strongly than the value-weighted stock-return model.

Why does the LR test not reject the model for stocks?

As we hinted earlier, one reason may be limited test power when return predictability is small (as reflected in the low R_R^2 for stocks).

When aggregate stock returns are nearly unpredictable, there is almost no predictable variation to constrain.

118.10 Residual diagnostics

As a final check, let's inspect the residual paths, histograms, and diagnostic statistics from the restricted model to assess whether the normality and serial independence assumptions hold.

```
diag_lag = 2
diag_fit = mle_fits[diag_lag]
resid = diag_fit["residuals"]

if diag_fit["converged"] and resid is not None:
    fig, axes = plt.subplots(2, 2, figsize=(12, 8))
    axes[0, 0].plot(resid[:, 0], lw=2)
    axes[0, 0].axhline(0.0, color="black", lw=2)
    axes[0, 0].set_ylabel("consumption residual")
    axes[0, 0].set_xlabel("observation")

    axes[0, 1].plot(resid[:, 1], lw=2)
    axes[0, 1].axhline(0.0, color="black", lw=2)
    axes[0, 1].set_ylabel("return residual")
    axes[0, 1].set_xlabel("observation")

    axes[1, 0].hist(resid[:, 0], bins=30, edgecolor="white")
    axes[1, 0].set_xlabel("consumption residual")
    axes[1, 0].set_ylabel("count")

    axes[1, 1].hist(resid[:, 1], bins=30, edgecolor="white")
    axes[1, 1].set_xlabel("return residual")
    axes[1, 1].set_ylabel("count")
    plt.tight_layout()
    plt.show()
```

The following table reports Jarque-Bera normality tests and Durbin-Watson serial correlation statistics for the restricted-model residuals.

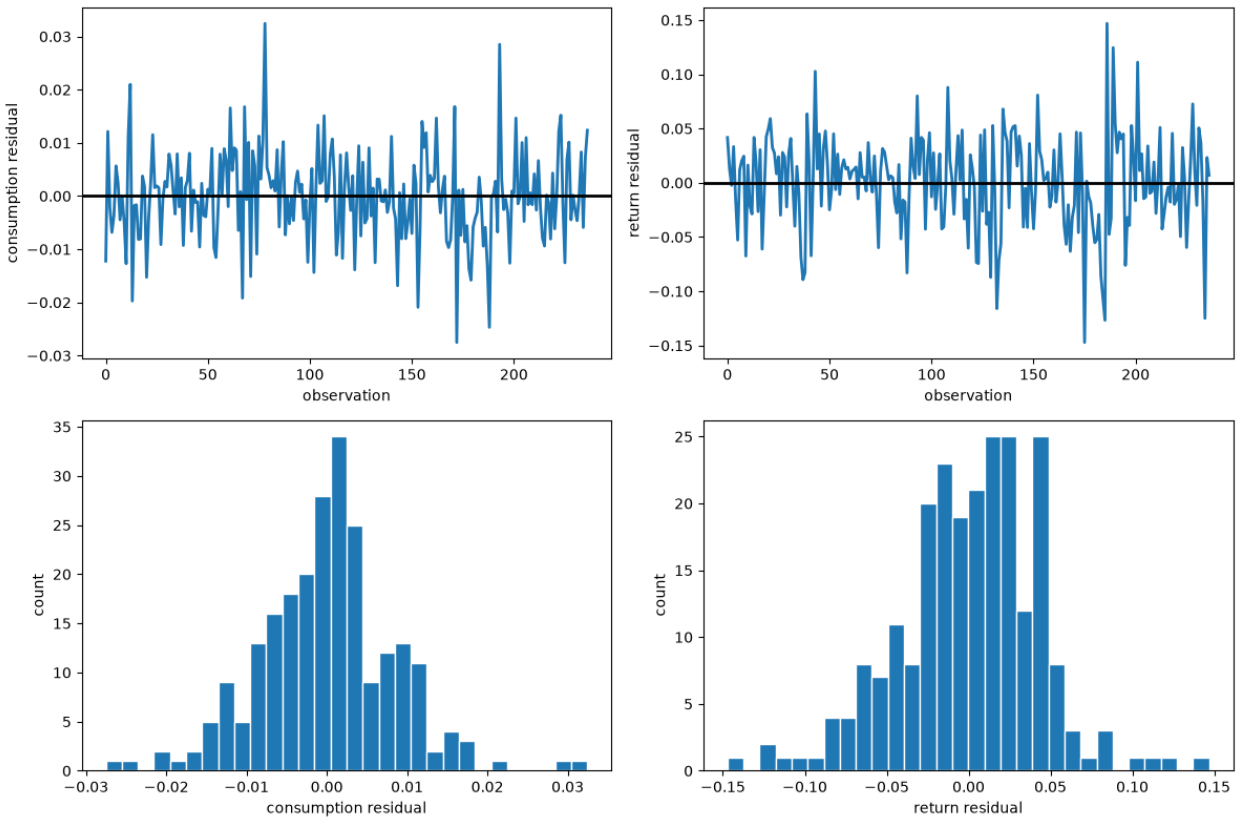


Fig. 118.1: Restricted-model residual diagnostics

```
if diag_fit["converged"] and resid is not None:
    diag = residual_diagnostics(resid)
    diag_df = pd.DataFrame({
        "JB stat": [diag["consumption_jb_stat"], diag["return_jb_stat"]],
        "JB p-val": [diag["consumption_jb_pval"], diag["return_jb_pval"]],
        "DW": [diag["consumption_dw"], diag["return_dw"]],
    }, index=pd.Index(["consumption", "return"], name="series"))
    display_table(diag_df, fmt={
        "JB stat": "{:.2f}", "JB p-val": "{:.4f}", "DW": "{:.3f}",
    })
```

series	JB stat	JB p-val	DW
consumption	18.16	0.0001	1.969
return	13.27	0.0013	1.766

The residual time-series plots reveal periods of volatility clustering, while the histograms show departures from bell curve with fatter tails.

The Jarque-Bera statistics are large for both series, rejecting the normality assumption that underlies the likelihood.

The Durbin-Watson statistics are close to 2 for both series, so serial correlation is not a concern.

This motivates the companion lecture *Estimating Euler Equations by Generalized Method of Moments* where we discuss how GMM avoids the normality assumption.

118.11 Connection to the equity premium puzzle

Our estimates reproduce the pattern that Mehra and Prescott [1985] later called the **equity premium puzzle**.

- *Low estimated risk aversion:* The estimated $\hat{\alpha}$ values (and thus risk aversion $-\hat{\alpha}$) from the table above are similar to those in Hansen and Singleton [1983], who report $\hat{\alpha}$ between -0.32 and -1.25.
- *Tiny return predictability:* The unrestricted-VAR R^2_R values are comparable to the 0.02 to 0.06 range in Hansen and Singleton [1983] – the predictable component of stock returns is small relative to the unpredictable component.
- *Strong rejection for Treasury bills:* The Euler-equation restrictions are decisively rejected for the nominally risk-free Treasury bill return, just as in Table 4 of Hansen and Singleton [1983].

The restrictions have more bite when the return series is more predictable (the unrestricted-VAR R^2_R for bills is much larger than for stocks), a precursor to what Weil [1989] would later call the **risk-free rate puzzle**.

Mehra and Prescott [1985] presented closely related findings.

In a calibrated version of the consumption-based model with CRRA utility, they showed that for relative risk aversion γ_{MP} in the range of 0 to 10, the model could not simultaneously match:

1. the average annual equity premium of about 6%,
2. the average annual risk-free rate of about 1%.

Matching the equity premium requires γ_{MP} well above 10, while matching the risk-free rate requires low γ_{MP}.

The same difficulty appears in our estimates: the implied risk aversion $-\hat{\alpha}$ is too low to generate a large equity premium.

118.12 Another approach

A companion lecture *Estimating Euler Equations by Generalized Method of Moments* describes a closely related paper in which Hansen and Singleton specified less about the joint distribution of returns and consumption growth.

They formulated an incomplete probability model that stops short of specifying a likelihood function.

To proceed, they used a generalized methods of moments (GMM) estimator to estimate key parameters that appear in the Euler equation.

CHAPTER

NINE

ESTIMATING EULER EQUATIONS BY GENERALIZED METHOD OF MOMENTS

Contents

- *Estimating Euler Equations by Generalized Method of Moments*
  - *Overview*
  - *The economic model*
  - *From conditional to unconditional moments*
  - *GMM criterion and asymptotic theory*
  - *Covariance estimation and the choice of instruments*
  - *Empirical GMM estimation*
  - *Summary*

119.1 Overview

This lecture implements the generalized instrumental variables estimator of Hansen and Singleton [1982] for nonlinear rational expectations models.

The preceding lecture *Stochastic Consumption, Risk Aversion, and the Temporal Behavior of Asset Returns* derives the consumption Euler equation from the representative consumer's problem with CRRA preferences and estimates it by maximum likelihood under normality assumptions.

That approach requires specifying the joint distribution of consumption and returns, and its validity depends on lognormality being correct.

However, as we saw in *Stochastic Consumption, Risk Aversion, and the Temporal Behavior of Asset Returns*, the lognormal model is rejected by the data.

Moreover, outside of linear-quadratic environments, closed-form solutions for equilibrium typically require strong assumptions about the stochastic properties of forcing variables, the nature of preferences, or the production technology.

Hansen and Singleton [1982] propose an estimation strategy that circumvents this requirement.

The key idea is that the Euler equations from economic agents' optimization problems imply a set of population orthogonality conditions that depend on observable variables and unknown preference parameters.

By making sample counterparts of these orthogonality conditions close to zero, the parameters can be estimated without explicitly solving for the stochastic equilibrium and without specifying the distribution of the observable variables.

Though maximum likelihood estimators (such as the MLE in *Stochastic Consumption, Risk Aversion, and the Temporal Behavior of Asset Returns*) will be asymptotically more efficient when the distributional assumptions are correctly specified.

Relative to the full paper, we only estimate one return at a time (value-weighted stock returns), using only monthly nondurable consumption (`ND`), and omitting their maximum-likelihood comparison (Table II) and multi-return systems (Table III).

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from IPython.display import Latex
from numba import njit
from scipy import stats
from scipy.optimize import minimize
from statsmodels.sandbox.regression import gmm
from statsmodels.tsa.stattools import acf
```

We also define a helper to display DataFrames as LaTeX arrays in the hidden cell below

119.2 The economic model

We consider a single-good economy with a representative consumer whose preferences are of the CRRA type, following Hansen and Singleton [1982] and Hansen and Singleton [1983].

Note

The following discussion is very close to the setup in *Stochastic Consumption, Risk Aversion, and the Temporal Behavior of Asset Returns*, but it is more general in that it allows for multiple assets with different maturities and does not assume lognormality.

The representative consumer chooses stochastic consumption and investment plans to maximize

$$\max E_0 \sum_{t=0}^{\infty} \beta^t u(C_t) \tag{119.1}$$

where C_t is consumption in period t, $\beta \in (0,1)$ is a subjective discount factor, and $u(\cdot)$ is a strictly concave period utility function.

The consumer trades N assets with potentially different maturities.

Let Q_{jt} denote the quantity of asset j held at the end of date t, P_{jt} its price at date t, and R_{jt} the date-t payoff from holding one unit of an M_j-period asset purchased at date $t - M_j$.

Feasible consumption and investment plans must satisfy the sequence of budget constraints

$$C_t + \sum_{j=1}^{N} P_{jt} Q_{jt} \leq \sum_{j=1}^{N} R_{jt} Q_{jt-M_j} + W_t, \tag{119.2}$$

where W_t is real labor income at date t.

We specialize to CRRA preferences

$$u(C_t) = \frac{C_t^{1-\gamma}}{1-\gamma}, \quad \gamma > 0, \tag{119.3}$$

where γ is the coefficient of relative risk aversion.

The maximization of (119.1) subject to (119.2) gives the first-order necessary conditions (see Lucas [1978], Brock [1982], Prescott and Mehra [1980], and *Stochastic Consumption, Risk Aversion, and the Temporal Behavior of Asset Returns*):

$$P_{jt} u'(C_t) = \beta^{M_j} E_t \left[R_{jt+M_j} u'(C_{t+M_j}) \right], \quad j = 1, \ldots, N. \tag{119.4}$$

When asset j is a one-period stock ($M_j = 1$) with payoff $R_{jt+1} = P_{jt+1} + D_{jt+1}$ where D_{jt} is the dividend, the **gross real return** is $R^i_{t+1} = (P_{i,t+1} + D_{i,t+1})/P_{i,t}$.

Substituting the CRRA marginal utility into (119.4) and dividing both sides by $P_{jt} u'(C_t)$ yields the Euler equation

$$E_t \left[\beta \left(\frac{C_{t+1}}{C_t} \right)^{-\gamma} R^i_{t+1} \right] = 1, \tag{119.5}$$

where R^i_{t+1} is the gross real return on asset i.

We define the **stochastic discount factor** $M_{t+1}(\theta) = \beta (C_{t+1}/C_t)^{-\gamma}$ with parameter vector $\theta = (\gamma, \beta)$.

In this notation the Euler equation becomes $E_t[M_{t+1}(\theta) R^i_{t+1} - 1] = 0$.

As we have seen and will see, equation (119.5) is the central object of both *Stochastic Consumption, Risk Aversion, and the Temporal Behavior of Asset Returns* and this lecture.

It holds for every traded asset for which the agent's optimality conditions apply (interior solution, no binding portfolio constraints or transaction costs).

It depends on observable quantities (consumption growth and returns) and unknown preference parameters (γ and β).

The challenge that Hansen and Singleton [1982] address is how to estimate θ from (119.5) without specifying the rest of the economic environment.

119.3 From conditional to unconditional moments

A natural approach to estimating θ from (119.5) would be to specify the entire economic environment, solve for equilibrium, and apply maximum likelihood.

Just like what we did together in *Stochastic Consumption, Risk Aversion, and the Temporal Behavior of Asset Returns*,

Hansen and Singleton [1982] argue that this is impractical for most nonlinear models because it requires strong assumptions about the stochastic properties of "forcing variables" and the production technology.

Their alternative is to work directly with the Euler equation's implications for observable moments.

The Euler equation (119.5) states that $E_t[M_{t+1}(\theta_0) R^i_{t+1} - 1] = 0$.

Let z_t denote any q-dimensional vector of variables that are in the agent's time-t information set and observed by the econometrician.

Because z_t is known at date t,

$$E\left[(M_{t+1}(\theta_0) R^i_{t+1} - 1) \otimes z_t \right] = E\left[z_t \otimes \underbrace{E_t[M_{t+1}(\theta_0) R^i_{t+1} - 1]}_{=0 \text{ by the Euler equation}} \right] = 0.$$

This yields the moment restriction

$$E\left[(M_{t+1}(\theta_0)R^i_{t+1} - 1) \otimes z_t\right] = 0, \tag{119.6}$$

where $\otimes$ denotes the Kronecker product.

The vector z_t plays the role of **instruments**.

The conditional Euler equation $E_t[M_{t+1}R^i_{t+1} - 1] = 0$ says that the pricing error is unpredictable given *everything* in the agent's time-t information set.

That is a very strong restriction – it says the pricing error is orthogonal to every time-t measurable random variable.

We cannot use the entire information set in practice, but we can pick any finite collection of time-t observable variables z_t and the orthogonality must still hold.

Each variable we include in z_t gives us one sample moment condition $\frac{1}{T}\sum_t (M_{t+1}R^i_{t+1} - 1)\, z_{kt} \approx 0$ that we can compute from data.

More instruments means more orthogonality conditions to match, which can improve efficiency and provides more overidentifying restrictions to test the model against.

With q instruments and m Euler equations we obtain mq moment restrictions for estimating the parameter vector θ.

For one return and p lags of instruments, Hansen and Singleton [1982] use

$$z_t = \left[1, R_t, g_t, R_{t-1}, g_{t-1}, \ldots, R_{t-p+1}, g_{t-p+1}\right]^\top, \tag{119.7}$$

where $g_t = C_t / C_{t-1}$ is gross consumption growth.

Note

Hansen and Singleton [1982] describe z_t as "lagged values of x_{t+1}" where $x_{t+1} = (R_{t+1}, g_{t+1})$.

The constant 1 is not stated explicitly but is implied by the degrees of freedom reported in their Table I.

More generally, Hansen and Singleton [1982] write the first-order condition as $E_t[h(x_{t+n}, b_0)] = 0$, where x_{t+n} is a vector of observables dated $t+n$ and b_0 is the true parameter vector.

The disturbance $u_{t+1} = h(x_{t+1}, b_0)$ is serially uncorrelated in the one-period stock case.

Stacking moment conditions via the Kronecker product gives $f(x_{t+n}, z_t, b) = h(x_{t+n}, b) \otimes z_t$, a vector of dimension mq.

The resulting unconditional restriction $E[f(x_{t+n}, z_t, b_0)] = 0$ nests both the single-return one-period Euler equation and the multi-maturity asset pricing restrictions.

The orthogonality condition and lagged instrument vector follow equations (119.6) and (119.7)

```
def euler_error_horizon(params, exog, horizon=1):
    """
    Compute Euler-equation pricing errors for (γ, β) at a given horizon.
    """
    if horizon < 1:
        raise ValueError("horizon must be at least one.")
    γ, β = params
    gross_return = exog[:, 0]
    gross_cons_growth = exog[:, 1]
    return (β ** horizon) * gross_cons_growth ** (-γ) * gross_return - 1.0
```

(continues on next page)

(continued from previous page)

```
def euler_error_grad_horizon(params, exog, horizon=1):
    """
    Gradient of the Euler error wrt (γ, β) at a given horizon.
    """
    if horizon < 1:
        raise ValueError("horizon must be at least one.")
    γ, β = params
    gross_return = exog[:, 0]
    gross_cons_growth = exog[:, 1]

    g_pow = gross_cons_growth ** (-γ)
    common = (β ** horizon) * g_pow * gross_return

    dγ = -common * np.log(gross_cons_growth)
    dβ = horizon * (β ** (horizon - 1)) * g_pow * gross_return
    return np.column_stack([dγ, dβ])


def euler_error(params, exog):
    """
    One-period Euler-equation pricing errors for (γ, β).
    """
    return euler_error_horizon(params, exog, horizon=1)
```

A helper aligns outcomes and lagged instruments for nonlinear IV-GMM

```
def build_gmm_arrays(data, n_lags):
    """
    Build endog, exog, and instruments for nonlinear IV-GMM.
    """
    if n_lags < 1:
        raise ValueError("n_lags must be at least one.")
    if data.shape[0] <= n_lags:
        raise ValueError("Sample size must exceed n_lags.")

    t_obs = data.shape[0]
    exog = data[n_lags:, :]
    endog = np.zeros(exog.shape[0])
    n_obs = t_obs - n_lags
    n_instr = 2 * n_lags + 1

    instruments = np.empty((n_obs, n_instr))
    instruments[:, 0] = 1.0

    for j in range(n_lags):
        left = 2 * j + 1
        right = left + 2
        instruments[:, left:right] = data[n_lags - 1 - j : t_obs - 1 - j, :]

    return endog, exog, instruments
```

When the asset has maturity $n > 1$, the Euler equation involves n-period compounded returns and consumption growth.

For CRRA preferences, the n-period Euler restriction is

$$E_t\left[\beta^n \left(\frac{C_{t+n}}{C_t}\right)^{-\gamma} R^i_{t,t+n}\right] = 1. \tag{119.8}$$

For estimation, the n-period exog can either use directly observed n-period returns/payoffs, or be constructed by compounding one-period returns and consumption growth over n consecutive periods.

Again, a key requirement from Hansen and Singleton [1982] is that instruments z_t must lie in the agent's time-t information set $\mathcal{I}_t$.

For the multi-period case, instruments must be measurable with respect to $\mathcal{I}_t$.

In particular, one should avoid any lagged multi-period aggregates that still include periods after t, since these would embed realizations not in $\mathcal{I}_t$.

The n-period exog is constructed by compounding one-period data, and instruments are timed to lie in $\mathcal{I}_t$.

```
def build_gmm_arrays_horizon(one_period_data, n_lags, horizon):
    """
    Build endog, exog, and instruments for multi-period GMM.

    Exog contains n-period compounded returns and consumption growth.
    """
    if horizon < 1:
        raise ValueError("horizon must be at least one.")
    if n_lags < 1:
        raise ValueError("n_lags must be at least one.")
    T = one_period_data.shape[0]
    if T <= n_lags + horizon:
        raise ValueError("Sample size too small for given n_lags and horizon.")

    # Each observation starts at index t (the first period in the window)
    # The window spans one_period_data[t : t + horizon]
    # Instruments use one_period_data[t - 1], ..., one_period_data[t - n_lags]
    starts = np.arange(n_lags, T - horizon + 1)
    n_obs = len(starts)

    exog = np.empty((n_obs, 2))
    n_instr = 2 * n_lags + 1
    instruments = np.empty((n_obs, n_instr))
    instruments[:, 0] = 1.0

    for i, t in enumerate(starts):
        window = one_period_data[t : t + horizon, :]
        exog[i, 0] = np.prod(window[:, 0])    # n-period return
        exog[i, 1] = np.prod(window[:, 1])    # n-period consumption growth
        for j in range(n_lags):
            instruments[
                i, 2 * j + 1 : 2 * j + 3] = one_period_data[t - 1 - j, :]

    endog = np.zeros(n_obs)
    return endog, exog, instruments
```

When $n > 1$, the Euler equation involves variables dated $t + n$, and the disturbance $u_{t+n} = h(x_{t+n}, b_0)$ will generally be serially correlated.

As Hansen and Singleton [1982] note, if the m assets are all one-period stocks, then u is serially uncorrelated because observations on x_{t-s}, $s \geq 0$, are in the agent's time-t information set and $E_t[h(x_{t+1}, b_0)] = 0$.

But if $n_j > 1$ for some asset j, the condition $E_t[h(x_{t+n}, b_0)] = 0$ does not preclude serial correlation in u, since x_{t+n-1} is not necessarily in I_t when $n > 1$.

The number of population autocovariances in the long-run covariance S_0 is determined by n, the order of the moving average disturbance term u_t.

We implement this directly as a finite-order covariance estimator

```
def finite_ma_covariance(moment_series, ma_order):
    """
    Estimate
    S = Gamma_0 + sum_{j=1}^{ma_order}(Gamma_j + Gamma_j.T) for moment vectors.
    """
    if ma_order < 0:
        raise ValueError("ma_order must be nonnegative.")
    if moment_series.ndim != 2:
        raise ValueError("moment_series must be 2D.")

    t_obs, n_mom = moment_series.shape
    if t_obs <= ma_order:
        raise ValueError("Need more observations than ma_order.")

    # Use the *uncentered* cross products
    # T^{-1} sum_t f_t f_{t-j}' and then add the symmetric lag terms.
    s_hat = moment_series.T @ moment_series / t_obs

    for j in range(1, ma_order + 1):
        gamma_j = moment_series[j:, :].T @ moment_series[:-j, :] / t_obs
        s_hat += gamma_j + gamma_j.T

    ridge = 1e-8 * np.eye(n_mom)
    return s_hat + ridge
```

The estimation procedure in Hansen and Singleton [1982] is a two-step generalized instrumental variables procedure.

In the first step, we minimize the GMM criterion with a suboptimal weighting matrix (the identity) to obtain a consistent preliminary estimate b_T.

In the second step, we use b_T to estimate the covariance matrix of the sample moment conditions and invert it to form the optimal weighting matrix, then re-minimize the criterion.

Let's implement this algorithm

```
def two_step_gmm(data, n_lags, ma_order=0, horizon=1, start_params=None):
    """
    Two-step GMM with finite-order covariance.

    The Euler error uses β**horizon.
    """
    if start_params is None:
        start_params = np.array([1.0, 0.99])
    else:
        start_params = np.asarray(start_params, dtype=float)

    if horizon == 1:
        _, exog, instruments = build_gmm_arrays(data, n_lags)
    else:
        _, exog, instruments = build_gmm_arrays_horizon(data, n_lags, horizon)
    n_obs = exog.shape[0]

    def sample_moments(params):
        err = euler_error_horizon(params, exog, horizon=horizon)
        return err[:, None] * instruments

    def objective(params, weight_matrix):
```

(continues on next page)

(continued from previous page)

```
    g_bar = sample_moments(params).mean(axis=0)
    return float(g_bar @ weight_matrix @ g_bar)

def objective_grad(params, weight_matrix):
    g_bar = sample_moments(params).mean(axis=0)
    grad_err = euler_error_grad_horizon(params, exog, horizon=horizon)
    d_bar = (instruments.T @ grad_err) / n_obs
    return 2.0 * d_bar.T @ weight_matrix @ g_bar

q = instruments.shape[1]
w_identity = np.eye(q)

bounds = [(-2.0, 10.0), (0.85, 1.5)]

def coarse_starts(weight_matrix, n_best=5):
    γ_grid = np.linspace(bounds[0][0], bounds[0][1], 33)
    β_grid = np.linspace(bounds[1][0], bounds[1][1], 33)
    scored = []
    for γ0 in γ_grid:
        for β0 in β_grid:
            params0 = np.array([γ0, β0])
            val = objective(params0, weight_matrix)
            if np.isfinite(val):
                scored.append((val, params0))
    scored.sort(key=lambda item: item[0])
    return [params for _, params in scored[:n_best]] or [start_params]

def best_local_minimize(weight_matrix, starts):
    best = None
    for x0 in starts:
        res = minimize(
            objective,
            x0=x0,
            args=(weight_matrix,),
            jac=objective_grad,
            method="L-BFGS-B",
            bounds=bounds,
            options={"maxiter": 25_000},
        )
        if not np.isfinite(res.fun):
            continue
        if best is None or (res.fun < best.fun):
            best = res
    return best if best is not None else minimize(
        objective,
        x0=start_params,
        args=(weight_matrix,),
        jac=objective_grad,
        method="L-BFGS-B",
        bounds=bounds,
        options={"maxiter": 25_000},
    )

step1_starts = [start_params] + coarse_starts(w_identity, n_best=5)
step1 = best_local_minimize(w_identity, step1_starts)
params1 = step1.x
```

(continues on next page)

(continued from previous page)

```
    m1 = sample_moments(params1)
    s_hat = finite_ma_covariance(m1, ma_order=ma_order)
    w_opt = np.linalg.pinv(s_hat)

    step2_starts = [params1] + coarse_starts(w_opt, n_best=5)
    step2 = best_local_minimize(w_opt, step2_starts)
    params2 = step2.x

    m2 = sample_moments(params2)
    s_hat2 = finite_ma_covariance(m2, ma_order=ma_order)
    w_opt2 = np.linalg.pinv(s_hat2)
    g2 = m2.mean(axis=0)
    j_stat = float(n_obs * (g2 @ w_opt2 @ g2))
    df = instruments.shape[1] - len(params2)
    j_prob = float(stats.chi2.cdf(j_stat, df=df)) if df > 0 else np.nan
    p_value = float(1.0 - j_prob) if df > 0 else np.nan

    # Asymptotic covariance under optimal weighting: (D' S^{-1} D)^{-1} / T.
    grad_err = euler_error_grad_horizon(params2, exog, horizon=horizon)
    d_hat = (instruments.T @ grad_err) / n_obs
    cov_hat = np.linalg.pinv(d_hat.T @ w_opt2 @ d_hat) / n_obs
    se_hat = np.sqrt(np.diag(cov_hat))

    return {
        "params_step1": params1,
        "params_step2": params2,
        "se_step2": se_hat,
        "weight_opt": w_opt2,
        "j_stat": j_stat,
        "j_df": int(df),
        "j_prob": j_prob,
        "j_pval": p_value,
        "n_obs": int(n_obs),
        "success": bool(step2.success),
    }
```

119.4 GMM criterion and asymptotic theory

We now formalize the estimation procedure.

Let $m_t(\theta) = (M_{t+1}(\theta) R^i_{t+1} - 1) \otimes z_t$ denote the vector of moment conditions at date t, and define the sample mean

$$g_T(\theta) = \frac{1}{T} \sum_{t=1}^{T} m_t(\theta). \tag{119.9}$$

If the model is correctly specified, $g_T(\theta_0)$ should be close to zero for large T.

We estimate θ by choosing the parameter vector that makes g_T as close to zero as possible, measured by a quadratic form:

$$\hat{\theta} = \arg\min_{\theta} g_T(\theta)^\top W_T g_T(\theta) \tag{119.10}$$

where W_T is a symmetric positive-definite weighting matrix.

Under regularity conditions given in Hansen [1982], the GMM estimator is consistent, asymptotically normal, and has the sandwich covariance matrix

$$\sqrt{T}(\hat{\theta} - \theta_0) \Rightarrow N\left(0, (D^\top W D)^{-1} D^\top W S W D (D^\top W D)^{-1}\right), \tag{119.11}$$

where $D = E[\partial m_t(\theta_0)/\partial\theta^\top]$ is the Jacobian of the moment conditions, S is the long-run covariance matrix of $m_t(\theta_0)$, and W is the probability limit of W_T.

Hansen [1982] shows that the optimal weighting matrix is $W^* = S^{-1}$, which yields the smallest asymptotic covariance matrix among all choices of W.

Under $W = S^{-1}$ the sandwich simplifies to $(D^\top S^{-1} D)^{-1}$.

When the number of moment conditions r (e.g., $r = mq$ for m Euler equations and q instruments) exceeds the number of parameters k, the model is overidentified and we can test whether the data are consistent with the maintained restrictions.

Hansen and Singleton [1982] test the overidentifying restrictions using a result from Hansen [1982]:

$$J_T = T\, g_T(\hat{\theta})^\top \hat{S}^{-1} g_T(\hat{\theta}) \Rightarrow \chi^2_{r-k}, \tag{119.12}$$

where $\hat{S}$ is a consistent estimator of S.

A large J_T relative to χ^2_{r-k} critical values leads to rejection of the model's overidentifying restrictions.

For the multi-period case ($n > 1$), the disturbance is at most MA($n-1$) as discussed above, so Hansen and Singleton [1982] obtain the optimal weighting matrix by setting $W_0 = S_0^{-1}$ where

$$S_0 = \sum_{j=-n+1}^{n-1} E\left[f(x_{t+n}, z_t, b_0)\, f(x_{t+n-j}, z_{t-j}, b_0)^\top\right]. \tag{119.13}$$

119.5 Covariance estimation and the choice of instruments

For the one-period Euler equation ($n = 1$), the disturbance $u_{t+1} = M_{t+1}(\theta_0) R_{t+1} - 1$ is a martingale difference sequence.

In this case the moment vector $m_t(\theta_0) = u_{t+1} \otimes z_t$ is serially uncorrelated, and the long-run covariance S equals the contemporaneous variance $E[m_t m_t^\top]$.

The covariance estimator therefore requires no kernel or bandwidth choice.

We simply use the sample analogue $\hat{S} = T^{-1} \sum_t m_t m_t^\top$.

In our implementation below, we use a HAC (heteroskedasticity and autocorrelation consistent) estimator against possible mild serial dependence from time aggregation or measurement timing.

This is a modern precaution, not part of the original Hansen and Singleton [1982] procedure, which exploits the known MA order directly as in (119.13).

The number of instrument lags p determines how many orthogonality conditions we use and hence the power of the J test.

Hansen and Singleton [1982] report results for NLAG $= 1, 2, 4, 6$ and note (footnote 12) that using more orthogonality conditions may lead to estimators with less desirable small-sample properties.

Below we implement the estimation procedure and allow the user to choose the number of lags and whether to use a HAC estimator for the covariance matrix

```
def estimate_gmm(
    data,
    n_lags,
    start_params=None,
    use_hac=True,
    hac_maxlag=None,
    maxiter=2,
```

(continues on next page)

(continued from previous page)

```
):
    """
    Estimate Euler-equation parameters with nonlinear IV-GMM.
    """
    if start_params is None:
        start_params = np.array([1.0, 0.99])

    endog, exog, instruments = build_gmm_arrays(data, n_lags)
    model = gmm.NonlinearIVGMM(endog, exog, instruments, euler_error)

    if use_hac:
        if hac_maxlag is None:
            hac_maxlag = max(
                1, int(
                    np.floor(4.0 * (endog.shape[0] / 100.0) ** (2.0 / 9.0))))
        result = model.fit(
            start_params=start_params,
            maxiter=maxiter,
            optim_method="bfgs",
            optim_args={"disp": False},
            weights_method="hac",
            wargs={"maxlag": hac_maxlag},
        )
    else:
        result = model.fit(
            start_params=start_params,
            maxiter=maxiter,
            optim_method="bfgs",
            optim_args={"disp": False},
        )

    return result
```

Next we include a helper to run GMM estimation across lag lengths and summarize results in a table

```
def run_gmm_by_lag(
    data,
    lags=(1, 2, 4, 6),
    use_hac=True,
    hac_maxlag=None,
):
    """
    Estimate GMM models by lag length and return a summary table.
    """
    rows = []
    results = {}

    for lag in lags:
        res = estimate_gmm(data, n_lags=lag, use_hac=use_hac, hac_maxlag=hac_maxlag)
        results[lag] = res
        j_stat, j_pval, j_df = res.jtest()
        j_prob = float(stats.chi2.cdf(j_stat, df=j_df)) if j_df > 0 else np.nan
        rows.append(
            {
                "NLAG": lag,
                "y_hat": res.params[0],
                "se_y": res.bse[0],
```

(continues on next page)

(continued from previous page)

```
                "β_hat": res.params[1],
                "se_β": res.bse[1],
                "j_stat": j_stat,
                "j_prob": j_prob,
                "j_pval": j_pval,
                "j_df": int(j_df),
                "n_obs": int(res.nobs),
            }
        )

    table = pd.DataFrame(rows).set_index("NLAG")
    return table, results


def run_two_step_by_lag(
    data,
    lags=(1, 2, 4, 6),
    horizon=1,
):
    """
    Two-step GMM with exact S0 (MA order 0) across lag lengths.
    """
    rows = []
    start_params = None
    for lag in lags:
        res = two_step_gmm(
            data,
            n_lags=lag,
            ma_order=0,
            horizon=horizon,
            start_params=start_params,
        )
        start_params = res["params_step2"]
        rows.append(
            {
                "NLAG": lag,
                "γ_hat": res["params_step2"][0],
                "se_γ": res["se_step2"][0],
                "β_hat": res["params_step2"][1],
                "se_β": res["se_step2"][1],
                "j_stat": res["j_stat"],
                "j_prob": res["j_prob"],
                "j_pval": res["j_pval"],
                "j_df": res["j_df"],
                "n_obs": res["n_obs"],
            }
        )
    return pd.DataFrame(rows).set_index("NLAG")
```

119.5.1 Simulation

Before applying the estimator to real data, we verify that GMM recovers known parameters from simulated data.

We build a simulator that generates synthetic return-growth pairs satisfying the Euler equation by construction.

We generate log consumption growth from a stationary AR(1), compute the stochastic discount factor at known true parameters, and construct gross returns as $R_{t+1} = \xi_{t+1}/M_{t+1}(\theta_0)$ where ξ_{t+1} is an iid lognormal shock with mean one.

```
@njit
def _ar1_simulate(mu_c, phi_c, sigma_c, shocks_c, total_n):
    """
    Simulate AR(1) log consumption growth.
    """
    delta_c = np.empty(total_n)
    delta_c[0] = mu_c
    for t in range(1, total_n):
        delta_c[t] = mu_c * (1.0 - phi_c) + phi_c * delta_c[t - 1] + sigma_c * shocks_
↪c[t]
    return delta_c


def simulate_euler_sample(
    n_obs,
    γ_true=0.8,
    β_true=0.993,
    seed=0,
):
    """
    Simulate [gross real return, gross consumption growth].
    """
    rng = np.random.default_rng(seed)
    mu_c = 0.0015
    sigma_c = 0.006
    phi_c = 0.4
    sigma_eta = 0.02
    burn_in = 200

    total_n = n_obs + burn_in
    shocks_c = rng.standard_normal(total_n)
    delta_c = _ar1_simulate(mu_c, phi_c, sigma_c, shocks_c, total_n)

    cons_growth = np.exp(delta_c[burn_in:])
    sdf = β_true * cons_growth ** (-γ_true)

    # Positive mean-one return shock: E[ξ]=1 so E[M R]=1 by construction.
    eps = rng.standard_normal(n_obs)
    xi = np.exp(sigma_eta * eps - 0.5 * sigma_eta**2)
    gross_return = xi / sdf

    return np.column_stack([gross_return, cons_growth])
```

We set $\gamma = 2$ and $\beta = 0.995$ as the true parameters and generate 700 monthly observations from the Euler-consistent DGP.

```
γ_true = 2.0
β_true = 0.995
sim_data = simulate_euler_sample(
```

(continues on next page)

(continued from previous page)

```
    n_obs=5000,
    γ_true=γ_true,
    β_true=β_true,
    seed=0,
)

print(f"Simulation sample size: {sim_data.shape[0]}")
print(f"True γ: {γ_true:.3f}")
print(f"True β: {β_true:.3f}")
```

```
Simulation sample size: 5000
True γ: 2.000
True β: 0.995
```

We now estimate GMM across lag lengths, following the format of Table I in Hansen and Singleton [1982].

```
sim_table = run_two_step_by_lag(sim_data, lags=(1, 2, 4, 6), horizon=1)
```

NLAG	$\hat{\gamma}$	se($\hat{\gamma}$)	$\hat{\beta}$	se($\hat{\beta}$)	J	df	Prob(J)
1	2.0806	0.1551	0.9948	0.0004	0.135	1	0.286
2	2.0927	0.1546	0.9949	0.0004	2.493	3	0.523
4	2.0930	0.1471	0.9949	0.0004	3.936	7	0.213
6	2.0991	0.1443	0.9949	0.0004	8.125	11	0.298

GMM recovers the true γ and β across lag specifications pretty closely.

For hypothesis testing, the right-tail p value is $1 - \text{Prob}(J)$.

The large standard errors visible in both papers' tables suggest that the preference parameters γ and β can be weakly identified.

To visualize this, we plot the GMM criterion over a (γ, β) grid using the simulated data

```
def gmm_objective_surface(
    data,
    n_lags=2,
    γ_grid=None,
    β_grid=None,
):
    """
    Compute identity-weighted GMM objective on a parameter grid.
    """
    _, exog, instruments = build_gmm_arrays(data, n_lags)

    if γ_grid is None:
        γ_grid = np.linspace(-1.0, 8.0, 70)
    if β_grid is None:
        β_grid = np.linspace(0.96, 1.02, 70)

    objective = np.empty((len(β_grid), len(γ_grid)))

    for i, β_val in enumerate(β_grid):
        for j, γ_val in enumerate(γ_grid):
            err = euler_error(np.array([γ_val, β_val]), exog)
            moments = (err[:, None] * instruments).mean(axis=0)
```

(continues on next page)

(continued from previous page)

```
            objective[i, j] = moments @ moments

    return γ_grid, β_grid, objective
```

```
γ_grid, β_grid, objective = gmm_objective_surface(sim_data, n_lags=2)
log_obj = np.log10(objective + 1e-12)

fig, ax = plt.subplots()
contours = ax.contourf(γ_grid, β_grid, log_obj, levels=30, cmap="viridis")
ax.set_xlabel(r"$\gamma$")
ax.set_ylabel(r"$\beta$")
ax.plot(γ_true, β_true, "k*", ms=12, lw=2, label="true values")
ax.legend()
plt.colorbar(contours, ax=ax)
plt.tight_layout()
plt.show()
```

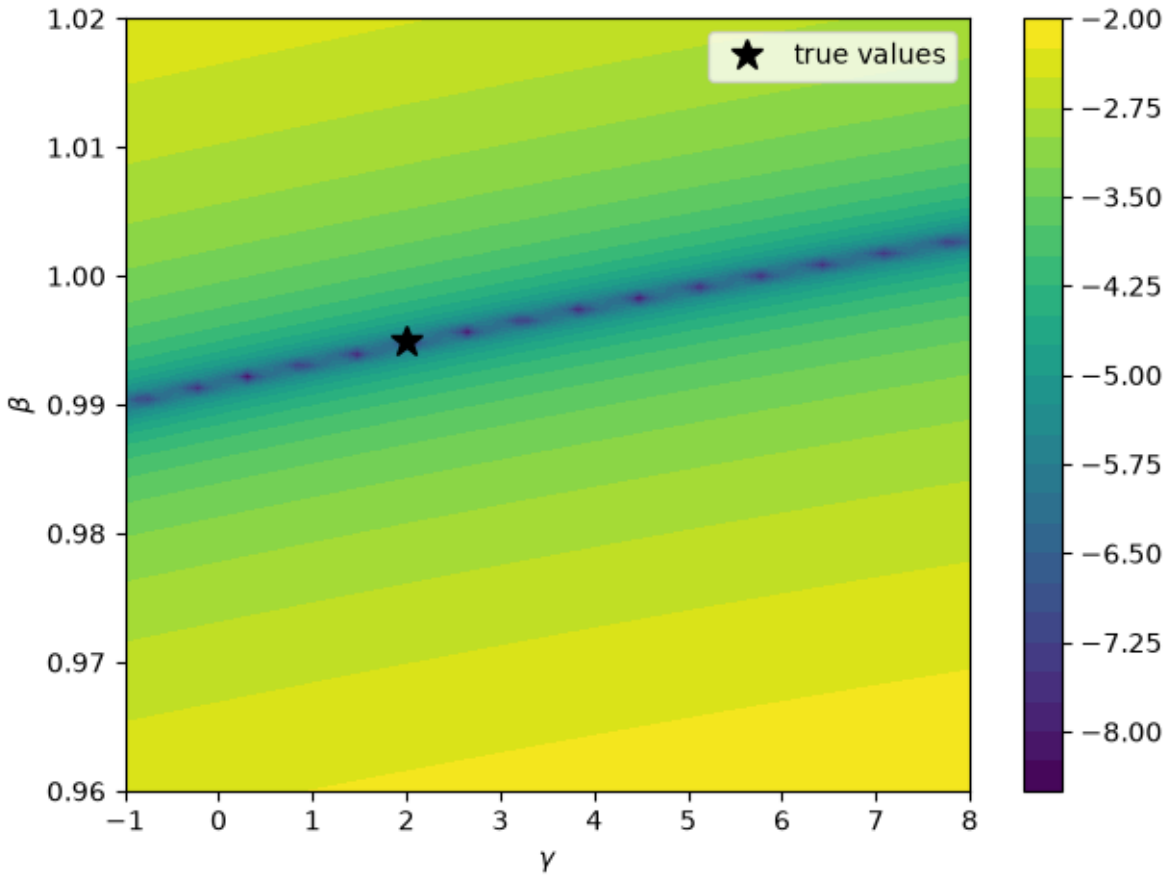


Fig. 119.1: GMM objective contour surface (simulated data)

The criterion surface may have elongated valleys where many parameter combinations fit the moments nearly equally well.

To illustrate the multi-period case from Section 2 of Hansen and Singleton [1982], we estimate the three-period Euler

restriction using overlapping-horizon returns and consumption growth, with instruments formed from one-period data dated t or earlier and the finite-order covariance appropriate for MA(2) disturbances.

```
horizon_n = 3
two_step = two_step_gmm(
    sim_data,
    n_lags=2,
    ma_order=horizon_n - 1,
    horizon=horizon_n,
)

print(f"Horizon n: {horizon_n}")
print(f"Step-2 converged: {two_step['success']}")
print(f"Step-2 gamma: {two_step['params_step2'][0]:.4f}")
print(f"Step-2 beta (one-period): {two_step['params_step2'][1]:.4f}")
print(
    f"J({two_step['j_df']}): {two_step['j_stat']:.3f}, "
    f"Prob={two_step['j_prob']:.3f}, p={two_step['j_pval']:.3f}"
)

_, exog_n, _ = build_gmm_arrays_horizon(sim_data, n_lags=2, horizon=horizon_n)
acf_n = acf(
    euler_error_horizon(two_step["params_step2"], exog_n, horizon=horizon_n),
    nlags=6,
    fft=True,
)
print("Euler-error ACF lags 1-3:", ", ".join([f"{v:.3f}" for v in acf_n[1:4]]))
```

```
Horizon n: 3
Step-2 converged: True
Step-2 gamma: 2.1095
Step-2 beta (one-period): 0.9949
J(3): 2.518, Prob=0.528, p=0.472
Euler-error ACF lags 1-3: 0.678, 0.356, 0.023
```

The low-lag ACF is consistent with the MA(2) dependence implied by the 3-period horizon.

We now run a Monte Carlo exercise with 500 replications to visualize the finite-sample distribution of $\hat{\gamma}$, $\hat{\beta}$, and the J statistic and verify that the asymptotic theory from Section 3 of Hansen and Singleton [1982] provides a reasonable approximation.

```
n_rep = 500
estimates = []
j_stats = []

for rep in range(n_rep):
    rep_data = simulate_euler_sample(
        n_obs=900,
        γ_true=γ_true,
        β_true=β_true,
        seed=rep,
    )
    rep_res = estimate_gmm(rep_data, n_lags=2, use_hac=True, maxiter=2)
    estimates.append(rep_res.params)
    j_stats.append(rep_res.jval)

estimates = np.asarray(estimates)
j_stats = np.asarray(j_stats)
```

(continues on next page)

(continued from previous page)

```
fig, axes = plt.subplots(1, 3, figsize=(15, 4))
axes[0].hist(estimates[:, 0], bins=20, edgecolor="white")
axes[0].axvline(γ_true, color="red", ls="--", lw=2)
axes[0].set_xlabel(r"$\hat{\gamma}$")
axes[1].hist(estimates[:, 1], bins=20, edgecolor="white")
axes[1].axvline(β_true, color="red", ls="--", lw=2)
axes[1].set_xlabel(r"$\hat{\beta}$")

df_j = 2 * 2 + 1 - 2
axes[2].hist(j_stats, bins=20, density=True, edgecolor="white")
grid = np.linspace(0.0, max(j_stats.max(), 1.0), 200)
axes[2].plot(grid, stats.chi2.pdf(grid, df_j), "r-", lw=2)
axes[2].set_xlabel("j-statistic")
plt.tight_layout()
plt.show()
```

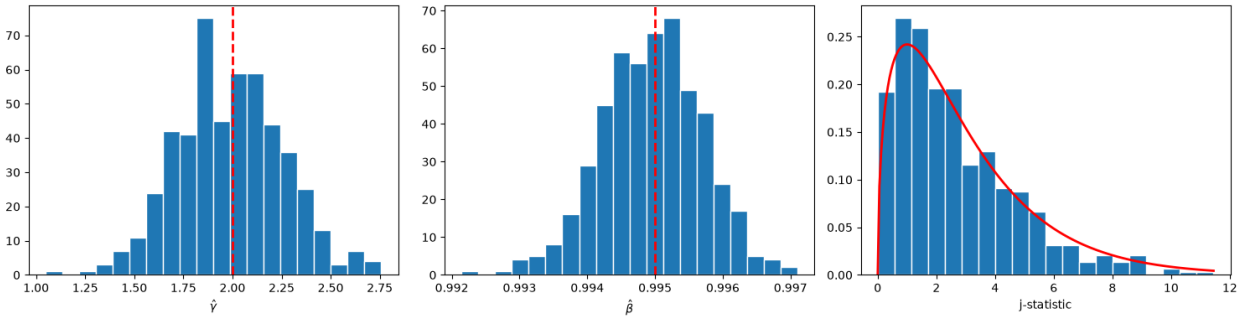


Fig. 119.2: Monte Carlo GMM sampling distributions

Both $\hat{\gamma}$ and $\hat{\beta}$ are centered near their true values, and the J histogram tracks the χ^2 density, supporting the asymptotic approximation at this sample size.

119.6 Empirical GMM estimation

We now apply GMM to observed data, following the empirical strategy of Section 5 in Hansen and Singleton [1982].

Hansen and Singleton [1982] use monthly per capita consumption of nondurables (ND) and nondurables plus services (NDS) paired with the equally-weighted (EWR) and value-weighted (VWR) aggregate stock returns from CRSP, for 1959:2 through 1978:12.

We focus on their ND+VWR specification using FRED nondurables consumption and the Ken French value-weighted market return as a proxy for CRSP, on the same 1959:2–1978:12 sample period.

Because the Ken French return is not identical to the original CRSP NYSE value-weighted return, we only want to match the paper qualitatively.

Both this lecture and the companion lecture *Stochastic Consumption, Risk Aversion, and the Temporal Behavior of Asset Returns* use the same data construction.

The hidden cell below loads a vendored monthly dataset of gross real returns and gross consumption growth. The data are built from the FRED and Ken French data libraries by the maintenance script at `_static/lecture_specific/hansen_singleton_1982/make_data.py` and read here directly from GitHub.

We first examine the raw data moments.

```
LAGS = (1, 2, 4, 6)

emp_frame, emp_data = get_estimation_data()

print(f"Mean net real return: {(emp_data[:, 0].mean() - 1.0) * 100:.3f}%")
print(f"Std net real return: {emp_data[:, 0].std() * 100:.3f}%")
print(f"Mean net consumption growth: {(emp_data[:, 1].mean() - 1.0) * 100:.3f}%")
print(f"Std net consumption growth: {emp_data[:, 1].std() * 100:.3f}%")
print(f"Std log consumption growth: {np.log(emp_data[:, 1]).std() * 100:.3f}%")
```

```
Mean net real return: 0.308%
Std net real return: 4.335%
Mean net consumption growth: 0.093%
Std net consumption growth: 0.915%
Std log consumption growth: 0.915%
```

One feature of these data is the large gap between the volatility of returns and the volatility of consumption growth.

This is again an empirical fact underlying the equity premium puzzle of Mehra and Prescott [1985]: matching the observed equity premium with CRRA preferences requires implausibly high risk aversion.

We now estimate the Euler equation using the two-step generalized instrumental variables (GIV) / GMM procedure in Hansen and Singleton [1982].

For the one-period stock-return Euler equation ($n = 1$), the disturbance is a martingale difference sequence, so the optimal weighting matrix uses the contemporaneous covariance $S_0 = E[m_t m_t^\top]$.

To match Table I, we report the paper's exponent parameter α in $E_t[\beta(C_{t+1}/C_t)^\alpha R_{t+1} - 1] = 0$.

The two-step GMM estimates of $\hat{\alpha}$ and $\hat{\beta}$ by lag length are

```
gmm_raw = run_two_step_by_lag(emp_data, lags=LAGS, horizon=1)
gmm_raw.index.name = "NLAG"

table_i = pd.DataFrame(index=gmm_raw.index)
table_i.index.name = "NLAG"
table_i[r"\hat{\alpha}"] = -gmm_raw["γ_hat"]
table_i[r"SE(\hat{\alpha})"] = gmm_raw["se_γ"]
table_i[r"\beta"] = gmm_raw["β_hat"]
table_i[r"\mathrm{SE}(\beta)"] = gmm_raw["se_β"]
table_i[r"\chi^2"] = gmm_raw["j_stat"]
table_i["DF"] = gmm_raw["j_df"]
table_i["Prob"] = gmm_raw["j_prob"]

display_table(
    table_i,
    fmt={
        r"\hat{\alpha}": "{:.4f}",
        r"SE(\hat{\alpha})": "{:.4f}",
        r"\beta": "{:.4f}",
        r"\mathrm{SE}(\beta)": "{:.4f}",
        r"\chi^2": "{:.4f}",
        "DF": "{:.0f}",
        "Prob": "{:.4f}",
    },
)
```

NLAG	$\hat{\alpha}$	$SE(\hat{\alpha})$	β	SE(β)	χ^2	DF	Prob
1	−1.1974	0.8428	0.9966	0.0026	1.7644	1	0.8159
2	−0.5494	0.7844	0.9969	0.0026	6.6667	3	0.9167
4	−0.5441	0.7040	0.9968	0.0026	9.0237	7	0.7490
6	−0.8718	0.6457	0.9973	0.0026	11.0602	11	0.5618

For comparison, Table I of Hansen and Singleton [1982] (as corrected in the 1984 *Econometrica* errata) reports the following ND+VWR values for 1959:2–1978:12:

NLAG	α	$SE(\alpha)$	β	SE(β)	χ^2	DF	Prob
1	−1.2028	0.7789	0.9976	0.0027	1.4570	1	0.7726
2	−0.5761	0.7067	0.9975	0.0027	5.8190	3	0.8792
4	−0.6565	0.6896	0.9978	0.0027	7.9230	7	0.6606
6	−0.9638	0.6425	0.9985	0.0027	10.5220	11	0.5159

They are very close to our estimates.

119.7 Summary

The GMM estimator requires only the orthogonality conditions implied by the Euler equation and a set of conditioning variables.

It does not require assumptions about the joint distribution of consumption and returns, the production technology, or any other part of the economic environment beyond the representative agent's first-order conditions.

So GMM is provides a way to estimate some objects of interest while not estimating all of the parameters that Hansen and Singleton [1983] estimated in their loglinear model of consumption growth and returns.

- If the complete model of Hansen and Singleton [1983] is correctly specified, then their maximum likelihood estimators promise to be more efficient that are the GMM estimators described in this lecture.
- The theme of estimating *something* while not estimating *everything* runs through much of Lars Peter Hansen's work. See Hansen [2014]

COMPETITIVE EQUILIBRIA WITH ARROW SECURITIES

120.1 Introduction

This lecture presents Python code for experimenting with competitive equilibria of an infinite-horizon pure exchange economy with

- Heterogeneous agents
- Endowments of a single consumption that are person-specific functions of a common Markov state
- Complete markets in one-period Arrow state-contingent securities
- Discounted expected utility preferences of a kind often used in macroeconomics and finance
- Common expected utility preferences across agents
- Common beliefs among agents
- A constant relative risk aversion (CRRA) one-period utility function that implies the existence of a representative consumer whose consumption process can be plugged into a formula for the pricing kernel for one-step Arrow securities and thereby determine equilibrium prices before determining an equilibrium distribution of wealth
- Differences in their endowments make individuals want to reallocate consumption goods across time and Markov states

We impose restrictions that allow us to **Bellmanize** competitive equilibrium prices and quantities

We use Bellman equations to describe

- asset prices
- continuation wealth levels for each person
- state-by-state natural debt limits for each person

In the course of presenting the model we shall encounter these important ideas

- a **resolvent operator** widely used in this class of models
- absence of **borrowing limits** in finite horizon economies
- state-by-state **borrowing limits** required in infinite horizon economies
- a counterpart of the **law of iterated expectations** known as a **law of iterated values**
- a **state-variable degeneracy** that prevails within a competitive equilibrium and that opens the way to various appearances of resolvent operators

120.2 The setting

In effect, this lecture implements a Python version of the model presented in section 9.3.3 of Ljungqvist and Sargent [Ljungqvist and Sargent, 2018].

120.2.1 Preferences and endowments

In each period $t \geq 0$, a stochastic event $s_t \in \mathbf{S}$ is realized.

Let the history of events up until time t be denoted $s^t = [s_0, s_1, \ldots, s_{t-1}, s_t]$.

(Sometimes we inadvertently reverse the recording order and denote a history as $s^t = [s_t, s_{t-1}, \ldots, s_1, s_0]$.)

The unconditional probability of observing a particular sequence of events s^t is given by a probability measure $\pi_t(s^t)$.

For $t > \tau$, we write the probability of observing s^t conditional on the realization of s^τas $\pi_t(s^t|s^\tau)$.

We assume that trading occurs after observing s_0, which we capture by setting $\pi_0(s_0) = 1$ for the initially given value of s_0.

In this lecture we shall follow much macroeconomics and econometrics and assume that $\pi_t(s^t)$ is induced by a Markov process.

There are K consumers named $k = 1, \ldots, K$.

Consumer k owns a stochastic endowment of one good $y_t^k(s^t)$ that depends on the history s^t.

The history s^t is publicly observable.

Consumer k purchases a history-dependent consumption plan $c^k = \{c_t^k(s^t)\}_{t=0}^\infty$

Consumer k orders consumption plans by

$$U_k(c^k) = \sum_{t=0}^{\infty} \sum_{s^t} \beta^t u_k[c_t^k(s^t)]\pi_t(s^t),$$

where $0 < \beta < 1$.

The right side is equal to $E_0 \sum_{t=0}^\infty \beta^t u_k(c_t^k)$, where E_0 is the mathematical expectation operator, conditioned on s_0.

Here $u_k(c)$ is an increasing, twice continuously differentiable, strictly concave function of consumption $c \geq 0$ of one good.

The utility function of person k satisfies the Inada condition

$$\lim_{c \downarrow 0} u_k'(c) = +\infty.$$

This condition implies that each agent chooses strictly positive consumption for every date-history pair (t, s^t).

Those interior solutions enable us to confine our analysis to Euler equations that hold with equality and also guarantee that **natural debt limits** don't bind in economies like ours with sequential trading of Arrow securities.

We adopt the assumption, routinely employed in much of macroeconomics, that consumers share probabilities $\pi_t(s^t)$ for all t and s^t.

A **feasible allocation** satisfies

$$\sum_i c_t^k(s^t) \leq \sum_i y_t^k(s^t)$$

for all t and for all s^t.

120.3 Recursive Formulation

Following descriptions in section 9.3.3 of Ljungqvist and Sargent [Ljungqvist and Sargent, 2018] chapter 9, we set up a competitive equilibrium of a pure exchange economy with complete markets in one-period Arrow securities.

When endowments $y^k(s)$ are all functions of a common Markov state s, the pricing kernel takes the form $Q(s'|s)$, where $Q(s'|s)$ is the price of one unit of consumption in state s' at date $t+1$ when the Markov state at date t is s.

These enable us to provide a recursive formulation of a consumer's optimization problem.

Consumer k's state at time t is its financial wealth a_t^k and Markov state s_t.

Let $v^k(a,s)$ be the optimal value of consumer k's problem starting from state (a,s).

- $v^k(a,s)$ is the maximum expected discounted utility that consumer k with current financial wealth a can attain in Markov state s.

The optimal value function satisfies the Bellman equation

$$v^k(a,s) = \max_{c,\hat{a}(s')} \left\{ u_k(c) + \beta \sum_{s'} v^k[\hat{a}(s'), s'] \pi(s'|s) \right\}$$

where maximization is subject to the budget constraint

$$c + \sum_{s'} \hat{a}(s') Q(s'|s) \leq y^k(s) + a$$

and also the constraints

$$\begin{aligned} c &\geq 0, \\ -\hat{a}(s') &\leq \bar{A}^k(s'), \qquad \forall s' \in \mathbf{S} \end{aligned}$$

with the second constraint evidently being a set of state-by-state debt limits.

Note that the value function and decision rule that solve the Bellman equation implicitly depend on the pricing kernel $Q(\cdot|\cdot)$ because it appears in the agent's budget constraint.

Use the first-order conditions for the problem on the right of the Bellman equation and a Benveniste-Scheinkman formula and rearrange to get

$$Q(s_{t+1}|s_t) = \frac{\beta u_k'(c_{t+1}^k) \pi(s_{t+1}|s_t)}{u_k'(c_t^k)},$$

where it is understood that $c_t^k = c^k(s_t)$ and $c_{t+1}^k = c^k(s_{t+1})$.

A **recursive competitive equilibrium** is an initial distribution of wealth $\vec{a}_0$, a set of borrowing limits $\{\bar{A}^k(s)\}_{k=1}^K$, a pricing kernel $Q(s'|s)$, sets of value functions $\{v^k(a,s)\}_{k=1}^K$, and decision rules $\{c^k(s), \hat{a}^k(s)\}_{k=1}^K$ such that

- The state-by-state borrowing constraints satisfy the recursion

$$\bar{A}^k(s) = y^k(s) + \sum_{s'} Q(s'|s) \bar{A}^k(s')$$

- For all k, given a_0^k, $\bar{A}^k(s)$, and the pricing kernel, the value functions and decision rules solve the consumers' problems;
- For all realizations of $\{s_t\}_{t=0}^\infty$, the consumption and asset portfolios $\{\{c_t^k, \{\hat{a}_{t+1}^k(s')\}_{s'}\}_k\}_t$ satisfy $\sum_k c_t^k = \sum_k y^k(s_t)$ and $\sum_k \hat{a}_{t+1}^k(s') = 0$ for all t and s'.
- The initial financial wealth vector $\vec{a}_0$ satisfies $\sum_{k=1}^K a_0^k = 0$.

The third condition asserts that there are zero net aggregate claims in all Markov states.

The fourth condition asserts that the economy is closed and starts from a situation in which there are zero net aggregate claims.

120.4 State Variable Degeneracy

Please see Ljungqvist and Sargent [Ljungqvist and Sargent, 2018] for a description of timing protocol for trades consistent with an Arrow-Debreu vision in which

- at time 0 there are complete markets in a complete menu of history s^t-contingent claims on consumption at all dates that all trades occur at time zero
- all trades occur once and for all at time 0

If an allocation and pricing kernel Q in a recursive competitive equilibrium are to be consistent with the equilibrium allocation and price system that prevail in a corresponding complete markets economy with such history-contingent commodities and all trades occurring at time 0, we must impose that $a_0^k = 0$ for $k = 1, \ldots, K$.

That is what assures that at time 0 the present value of each agent's consumption equals the present value of his endowment stream, the single budget constraint in arrangement with all trades occurring at time 0.

Starting the system with $a_0^k = 0$ for all i has a striking implication that we call **state variable degeneracy**.

Here is what we mean by **state variable degeneracy**:

Although two state variables a, s appear in the value function $v^k(a, s)$, within a recursive competitive equilibrium starting from $a_0^k = 0 \ \forall i$ at initial Markov state s_0, two outcomes prevail:

- $a_0^k = 0$ for all i whenever the Markov state s_t returns to s_0.
- Financial wealth a is an exact function of the Markov state s.

The first finding asserts that each household recurrently visits the zero financial wealth state with which it began life.

The second finding asserts that within a competitive equilibrium the exogenous Markov state is all we require to track an individual.

Financial wealth turns out to be redundant because it is an exact function of the Markov state for each individual.

This outcome depends critically on there being complete markets in Arrow securities.

For example, it does not prevail in the incomplete markets setting of this lecture *The Aiyagari Model*

120.5 Markov Asset Prices

Let's start with a brief summary of formulas for computing asset prices in a Markov setting.

The setup assumes the following infrastructure

- Markov states: $s \in S = [\bar{s}_1, \ldots, \bar{s}_n]$ governed by an n-state Markov chain with transition probability
$$P_{ij} = \Pr\left\{s_{t+1} = \bar{s}_j \mid s_t = \bar{s}_i\right\}$$
- A collection $h = 1, \ldots, H$ of $n \times 1$ vectors of H assets that pay off $d^h(s)$ in state s
- An $n \times n$ matrix pricing kernel Q for one-period Arrow securities, where Q_{ij} = price at time t in state $s_t = \bar{s}_i$ of one unit of consumption when $s_{t+1} = \bar{s}_j$ at time $t+1$:
$$Q_{ij} = \text{Price}\left\{s_{t+1} = \bar{s}_j \mid s_t = \bar{s}_i\right\}$$
- The price of risk-free one-period bond in state i is $R_i^{-1} = \sum_j Q_{i,j}$
- The gross rate of return on a one-period risk-free bond Markov state $\bar{s}_i$ is $R_i = (\sum_j Q_{i,j})^{-1}$

120.5.1 Exogenous Pricing Kernel

At this point, we'll take the pricing kernel Q as exogenous, i.e., determined outside the model

Two examples would be

- $Q = \beta P$ where $\beta \in (0, 1)$
- $Q = SP$ where S is an $n \times n$ matrix of *stochastic discount factors*

We'll write down implications of Markov asset pricing in a nutshell for two types of assets

- the price in Markov state s at time t of a **cum dividend** stock that entitles the owner at the beginning of time t to the time t dividend and the option to sell the asset at time $t+1$. The price evidently satisfies $p^h(\bar{s}_i) = d^h(\bar{s}_i) + \sum_j Q_{ij} p^h(\bar{s}_j)$, which implies that the vector p^h satisfies $p^h = d^h + Qp^h$ which implies the formula

$$p^h = (I - Q)^{-1} d^h$$

- the price in Markov state s at time t of an **ex dividend** stock that entitles the owner at the end of time t to the time $t+1$ dividend and the option to sell the stock at time $t+1$. The price is

$$p^h = (I - Q)^{-1} Q d^h$$

Note

The matrix geometric sum $(I - Q)^{-1} = I + Q + Q^2 + \cdots$ is an example of a **resolvent operator**.

Below, we describe an equilibrium model with trading of one-period Arrow securities in which the pricing kernel is endogenous.

In constructing our model, we'll repeatedly encounter formulas that remind us of our asset pricing formulas.

120.5.2 Multi-Step-Forward Transition Probabilities and Pricing Kernels

The (i, j) component of the ℓ-step ahead transition probability P^ℓ is

$$Prob(s_{t+\ell} = \bar{s}_j | s_t = \bar{s}_i) = P^\ell_{i,j}$$

The (i, j) component of the ℓ-step ahead pricing kernel Q^ℓ is

$$Q^{(\ell)}(s_{t+\ell} = \bar{s}_j | s_t = \bar{s}_i) = Q^\ell_{i,j}$$

We'll use these objects to state a useful property in asset pricing theory.

120.5.3 Laws of Iterated Expectations and Iterated Values

A **law of iterated values** has a mathematical structure that parallels a **law of iterated expectations**

We can describe its structure readily in the Markov setting of this lecture

Recall the following recursion satisfied by j step ahead transition probabilites for our finite state Markov chain:

$$P_j(s_{t+j}|s_t) = \sum_{s_{t+1}} P_{j-1}(s_{t+j}|s_{t+1}) P(s_{t+1}|s_t)$$

We can use this recursion to verify the law of iterated expectations applied to computing the conditional expectation of a random variable $d(s_{t+j})$ conditioned on s_t via the following string of equalities

$$\begin{aligned} E\left[Ed(s_{t+j})|s_{t+1}\right]|s_t &= \sum_{s_{t+1}}\left[\sum_{s_{t+j}} d(s_{t+j})P_{j-1}(s_{t+j}|s_{t+1})\right]P(s_{t+1}|s_t) \\ &= \sum_{s_{t+j}} d(s_{t+j})\left[\sum_{s_{t+1}} P_{j-1}(s_{t+j}|s_{t+1})P(s_{t+1}|s_t)\right] \\ &= \sum_{s_{t+j}} d(s_{t+j})P_j(s_{t+j}|s_t) \\ &= Ed(s_{t+j})|s_t \end{aligned}$$

The pricing kernel for j step ahead Arrow securities satisfies the recursion

$$Q_j(s_{t+j}|s_t) = \sum_{s_{t+1}} Q_{j-1}(s_{t+j}|s_{t+1})Q(s_{t+1}|s_t)$$

The time t **value** in Markov state s_t of a time $t+j$ payout $d(s_{t+j})$ is

$$V(d(s_{t+j})|s_t) = \sum_{s_{t+j}} d(s_{t+j})Q_j(s_{t+j}|s_t)$$

The **law of iterated values** states

$$V\left[V(d(s_{t+j})|s_{t+1})\right]|s_t = V(d(s_{t+j}))|s_t$$

We verify it by pursuing the following a string of inequalities that are counterparts to those we used to verify the law of iterated expectations:

$$\begin{aligned} V\left[V(d(s_{t+j})|s_{t+1})\right]|s_t &= \sum_{s_{t+1}}\left[\sum_{s_{t+j}} d(s_{t+j})Q_{j-1}(s_{t+j}|s_{t+1})\right]Q(s_{t+1}|s_t) \\ &= \sum_{s_{t+j}} d(s_{t+j})\left[\sum_{s_{t+1}} Q_{j-1}(s_{t+j}|s_{t+1})Q(s_{t+1}|s_t)\right] \\ &= \sum_{s_{t+j}} d(s_{t+j})Q_j(s_{t+j}|s_t) \\ &= EV(d(s_{t+j}))|s_t \end{aligned}$$

120.6 General Equilibrium

Now we are ready to do some fun calculations.

We find it interesting to think in terms of analytical **inputs** into and **outputs** from our general equilibrium theorizing.

120.6.1 Inputs

- Markov states: $s \in S = [\bar{s}_1, \ldots, \bar{s}_n]$ governed by an n-state Markov chain with transition probability

$$P_{ij} = \Pr\left\{s_{t+1} = \bar{s}_j \mid s_t = \bar{s}_i\right\}$$

- A collection of $K \times 1$ vectors of individual k endowments: $y^k(s), k = 1, \ldots, K$

- An $n \times 1$ vector of aggregate endowment: $y(s) \equiv \sum_{k=1}^{K} y^k(s)$
- A collection of $K \times 1$ vectors of individual k consumptions: $c^k(s), k = 1, \ldots, K$
- A collection of restrictions on feasible consumption allocations for $s \in S$:

$$c(s) = \sum_{k=1}^{K} c^k(s) \leq y(s)$$

- Preferences: a common utility functional across agents $E_0 \sum_{t=0}^{\infty} \beta^t u(c_t^k)$ with CRRA one-period utility function $u(c)$ and discount factor $\beta \in (0,1)$

The one-period utility function is

$$u(c) = \frac{c^{1-\gamma}}{1-\gamma}$$

so that

$$u'(c) = c^{-\gamma}$$

120.6.2 Outputs

- An $n \times n$ matrix pricing kernel Q for one-period Arrow securities, where Q_{ij} = price at time t in state $s_t = \bar{s}_i$ of one unit of consumption when $s_{t+1} = \bar{s}_j$ at time $t+1$
- pure exchange so that $c(s) = y(s)$
- a $K \times 1$ vector distribution of wealth vector α, $\alpha_k \geq 0, \sum_{k=1}^{K} \alpha_k = 1$
- A collection of $n \times 1$ vectors of individual k consumptions: $c^k(s), k = 1, \ldots, K$

120.6.3 Q is the Pricing Kernel

For any agent $k \in [1, \ldots, K]$, at the equilibrium allocation, the one-period Arrow securities pricing kernel satisfies

$$Q_{ij} = \beta \left(\frac{c^k(\bar{s}_j)}{c^k(\bar{s}_i)} \right)^{-\gamma} P_{ij}$$

where Q is an $n \times n$ matrix

This follows from agent k's first-order necessary conditions.

But with the CRRA preferences that we have assumed, individual consumptions vary proportionately with aggregate consumption and therefore with the aggregate endowment.

- This is a consequence of our preference specification implying that **Engle curves** are affine in wealth and therefore satisfy conditions for **Gorman aggregation**

Thus,

$$c^k(s) = \alpha_k c(s) = \alpha_k y(s)$$

for an arbitrary **distribution of wealth** in the form of an $K \times 1$ vector α that satisfies

$$\alpha_k \in (0,1), \quad \sum_{k=1}^{K} \alpha_k = 1$$

This means that we can compute the pricing kernel from

$$Q_{ij} = \beta \left(\frac{y_j}{y_i} \right)^{-\gamma} P_{ij} \tag{120.1}$$

Note that Q_{ij} is independent of vector α.

Key finding: We can compute competitive equilibrium **prices** prior to computing a **distribution of wealth**.

120.6.4 Values

Having computed an equilibrium pricing kernel Q, we can compute several **values** that are required to pose or represent the solution of an individual household's optimum problem.

We denote an $K \times 1$ vector of state-dependent values of agents' endowments in Markov state s as

$$A(s) = \begin{bmatrix} A^1(s) \\ \vdots \\ A^K(s) \end{bmatrix}, \quad s \in [\bar{s}_1, \ldots, \bar{s}_n]$$

and an $n \times 1$ vector of continuation endowment values for each individual k as

$$A^k = \begin{bmatrix} A^k(\bar{s}_1) \\ \vdots \\ A^k(\bar{s}_n) \end{bmatrix}, \quad k \in [1, \ldots, K]$$

A^k of consumer k satisfies

$$A^k = [I - Q]^{-1} [y^k]$$

where

$$y^k = \begin{bmatrix} y^k(\bar{s}_1) \\ \vdots \\ y^k(\bar{s}_n) \end{bmatrix} \equiv \begin{bmatrix} y_1^k \\ \vdots \\ y_n^k \end{bmatrix}$$

In a competitive equilibrium of an **infinite horizon** economy with sequential trading of one-period Arrow securities, $A^k(s)$ serves as a state-by-state vector of **debt limits** on the quantities of one-period Arrow securities paying off in state s at time $t+1$ that individual k can issue at time t.

These are often called **natural debt limits**.

Evidently, they equal the maximum amount that it is feasible for individual k to repay even if he consumes zero goods forevermore.

Remark: If we have an Inada condition at zero consumption or just impose that consumption be nonnegative, then in a **finite horizon** economy with sequential trading of one-period Arrow securities there is no need to impose natural debt limits. See the section on a Finite Horizon Economy below.

120.6.5 Continuation Wealth

Continuation wealth plays an important role in Bellmanizing a competitive equilibrium with sequential trading of a complete set of one-period Arrow securities.

We denote an $K \times 1$ vector of state-dependent continuation wealths in Markov state s as

$$\psi(s) = \begin{bmatrix} \psi^1(s) \\ \vdots \\ \psi^K(s) \end{bmatrix}, \quad s \in [\bar{s}_1, \ldots, \bar{s}_n]$$

and an $n \times 1$ vector of continuation wealths for each individual k as

$$\psi^k = \begin{bmatrix} \psi^k(\bar{s}_1) \\ \vdots \\ \psi^k(\bar{s}_n) \end{bmatrix}, \quad k \in [1, \ldots, K]$$

Continuation wealth ψ^k of consumer k satisfies

$$\psi^k = [I - Q]^{-1} [\alpha_k y - y^k] \tag{120.2}$$

where

$$y^k = \begin{bmatrix} y^k(\bar{s}_1) \\ \vdots \\ y^k(\bar{s}_n) \end{bmatrix}, \quad y = \begin{bmatrix} y(\bar{s}_1) \\ \vdots \\ y(\bar{s}_n) \end{bmatrix}$$

Note that $\sum_{k=1}^K \psi^k = 0_{n \times 1}$.

Remark: At the initial state $s_0 \in [\bar{s}_1, \ldots, \bar{s}_n]$, the continuation wealth $\psi^k(s_0) = 0$ for all agents $k = 1, \ldots, K$. This indicates that the economy begins with all agents being debt-free and financial-asset-free at time 0, state s_0.

Remark: Note that all agents' continuation wealths recurrently return to zero when the Markov state returns to whatever value s_0 it had at time 0.

120.6.6 Optimal Portfolios

A nifty feature of the model is that an optimal portfolio of a type k agent equals the continuation wealth that we just computed.

Thus, agent k's state-by-state purchases of Arrow securities next period depend only on next period's Markov state and equal

$$a_k(s) = \psi^k(s), \quad s \in [\bar{s}_1, \ldots, \bar{s}_n] \tag{120.3}$$

120.6.7 Equilibrium Wealth Distribution α

With the initial state being a particular state $s_0 \in [\bar{s}_1, \ldots, \bar{s}_n]$, we must have

$$\psi^k(s_0) = 0, \quad k = 1, \ldots, K$$

which means the equilibrium distribution of wealth satisfies

$$\alpha_k = \frac{V_z y^k}{V_z y} \tag{120.4}$$

where $V \equiv [I - Q]^{-1}$ and z is the row index corresponding to the initial state s_0.

Since $\sum_{k=1}^K V_z y^k = V_z y$, $\sum_{k=1}^K \alpha_k = 1$.

In summary, here is the logical flow of an algorithm to compute a competitive equilibrium:

- compute Q from the aggregate allocation and formula (120.1)
- compute the distribution of wealth α from the formula (120.4)
- Using α assign each consumer k the share α_k of the aggregate endowment at each state
- return to the α-dependent formula (120.2) and compute continuation wealths

- via formula (120.3) equate agent k's portfolio to its continuation wealth state by state

We can also add formulas for optimal value functions in a competitive equilibrium with trades in a complete set of one-period state-contingent Arrow securities.

Call the optimal value functions J^k for consumer k.

For the infinite horizon economy now under study, the formula is

$$J^k = (I - \beta P)^{-1} u(\alpha_k y), \quad u(c) = \frac{c^{1-\gamma}}{1-\gamma}$$

where it is understood that $u(\alpha_k y)$ is a vector.

120.7 Finite Horizon

We now describe a finite-horizon version of the economy that operates for $T+1$ periods $t \in \mathbf{T} = \{0, 1, \dots, T\}$.

Consequently, we'll want $T+1$ counterparts to objects described above, with one important exception: we won't need **borrowing limits**.

- borrowing limits aren't required for a finite horizon economy in which a one-period utility function $u(c)$ satisfies an Inada condition that sets the marginal utility of consumption at zero consumption to zero.
- Nonnegativity of consumption choices at all $t \in \mathbf{T}$ automatically limits borrowing.

120.7.1 Continuation Wealths

We denote a $K \times 1$ vector of state-dependent continuation wealths in Markov state s at time t as

$$\psi_t(s) = \begin{bmatrix} \psi^1(s) \\ \vdots \\ \psi^K(s) \end{bmatrix}, \quad s \in [\bar{s}_1, \dots, \bar{s}_n]$$

and an $n \times 1$ vector of continuation wealths for each individual k as

$$\psi_t^k = \begin{bmatrix} \psi_t^k(\bar{s}_1) \\ \vdots \\ \psi_t^k(\bar{s}_n) \end{bmatrix}, \quad k \in [1, \dots, K]$$

Continuation wealths ψ^k of consumer k satisfy

$$\begin{aligned} \psi_T^k &= [\alpha_k y - y^k] \\ \psi_{T-1}^k &= [I + Q][\alpha_k y - y^k] \\ &\vdots \qquad\qquad \vdots \\ \psi_0^k &= [I + Q + Q^2 + \cdots + Q^T][\alpha_k y - y^k] \end{aligned} \tag{120.5}$$

where

$$y^k = \begin{bmatrix} y^k(\bar{s}_1) \\ \vdots \\ y^k(\bar{s}_n) \end{bmatrix}, \quad y = \begin{bmatrix} y(\bar{s}_1) \\ \vdots \\ y(\bar{s}_n) \end{bmatrix}$$

Note that $\sum_{k=1}^K \psi_t^k = 0_{n \times 1}$ for all $t \in \mathbf{T}$.

Remark: At the initial state $s_0 \in [\bar{s}_1, \dots, \bar{s}_n]$, for all agents $k = 1, \dots, K$, continuation wealth $\psi_0^k(s_0) = 0$. This indicates that the economy begins with all agents being debt-free and financial-asset-free at time 0, state s_0.

Remark: Note that all agents' continuation wealths return to zero when the Markov state returns to whatever value s_0 it had at time 0. This will recur if the Markov chain makes the initial state s_0 recurrent.

With the initial state being a particular state $s_0 \in [\bar{s}_1, \ldots, \bar{s}_n]$, we must have

$$\psi_0^k(s_0) = 0, \quad k = 1, \ldots, K$$

which means the equilibrium distribution of wealth satisfies

$$\alpha_k = \frac{V_z y^k}{V_z y} \tag{120.6}$$

where now in our finite-horizon economy

$$V = [I + Q + Q^2 + \cdots + Q^T] \tag{120.7}$$

and z is the row index corresponding to the initial state s_0.

Since $\sum_{k=1}^K V_z y^k = V_z y$, $\sum_{k=1}^K \alpha_k = 1$.

In summary, here is the logical flow of an algorithm to compute a competitive equilibrium with Arrow securities in our finite-horizon Markov economy:

- compute Q from the aggregate allocation and formula (120.1)
- compute the distribution of wealth α from formulas (120.6) and (120.7)
- using α, assign each consumer k the share α_k of the aggregate endowment at each state
- return to the α-dependent formula (120.5) for continuation wealths and compute continuation wealths
- equate agent k's portfolio to its continuation wealth state by state

While for the infinite horizon economy, the formula for value functions is

$$J^k = (I - \beta P)^{-1} u(\alpha_k y), \quad u(c) = \frac{c^{1-\gamma}}{1-\gamma}$$

for the finite horizon economy the formula is

$$J_0^k = (I + \beta P + \cdots + \beta^T P^T) u(\alpha_k y),$$

where it is understood that $u(\alpha_k y)$ is a vector.

120.8 Python Code

We are ready to dive into some Python code.

As usual, we start with Python imports.

```
import numpy as np
import matplotlib.pyplot as plt
```

```
np.set_printoptions(suppress=True)
```

First, we create a Python class to compute the objects that comprise a competitive equilibrium with sequential trading of one-period Arrow securities.

In addition to infinite-horizon economies, the code is set up to handle finite-horizon economies indexed by horizon T.

We'll study examples of finite horizon economies after we first look at some infinite-horizon economies.

```
class RecurCompetitive:
    """
    A class that represents a recursive competitive economy
    with one-period Arrow securities.
    """

    def __init__(self,
                 s,         # state vector
                 P,         # transition matrix
                 ys,        # endowments ys = [y1, y2, .., yI]
                 γ=0.5,     # risk aversion
                 β=0.98,    # discount rate
                 T=None):  # time horizon, none if infinite

        # preference parameters
        self.γ = γ
        self.β = β

        # variables dependent on state
        self.s = s
        self.P = P
        self.ys = ys
        self.y = np.sum(ys, 1)

        # dimensions
        self.n, self.K = ys.shape

        # compute pricing kernel
        self.Q = self.pricing_kernel()

        # compute price of risk-free one-period bond
        self.PRF = self.price_risk_free_bond()

        # compute risk-free rate
        self.R = self.risk_free_rate()

        # V = [I - Q]^{-1} (infinite case)
        if T is None:
            self.T = None
            self.V = np.empty((1, n, n))
            self.V[0] = np.linalg.inv(np.eye(n) - self.Q)
        # V = [I + Q + Q^2 + ... + Q^T] (finite case)
        else:
            self.T = T
            self.V = np.empty((T+1, n, n))
            self.V[0] = np.eye(n)

            Qt = np.eye(n)
            for t in range(1, T+1):
                Qt = Qt.dot(self.Q)
                self.V[t] = self.V[t-1] + Qt

        # natural debt limit
        self.A = self.V[-1] @ ys

    def u(self, c):
        "The CRRA utility"
```

(continues on next page)

(continued from previous page)

```
        return c ** (1 - self.γ) / (1 - self.γ)

    def u_prime(self, c):
        "The first derivative of CRRA utility"

        return c ** (-self.γ)

    def pricing_kernel(self):
        "Compute the pricing kernel matrix Q"

        c = self.y

        n = self.n
        Q = np.empty((n, n))
        for i in range(n):
            for j in range(n):
                ratio = self.u_prime(c[j]) / self.u_prime(c[i])
                Q[i, j] = self.β * ratio * P[i, j]

        self.Q = Q

        return Q

    def wealth_distribution(self, s0_idx):
        "Solve for wealth distribution α"

        # set initial state
        self.s0_idx = s0_idx

        # simplify notations
        n = self.n
        Q = self.Q
        y, ys = self.y, self.ys

        # row of V corresponding to s0
        Vs0 = self.V[-1, s0_idx, :]
        α = Vs0 @ self.ys / (Vs0 @ self.y)

        self.α = α

        return α

    def continuation_wealths(self):
        "Given α, compute the continuation wealths ψ"

        diff = np.empty((n, K))
        for k in range(K):
            diff[:, k] = self.α[k] * self.y - self.ys[:, k]

        ψ = self.V @ diff
        self.ψ = ψ

        return ψ

    def price_risk_free_bond(self):
        "Give Q, compute price of one-period risk free bond"
```

(continues on next page)

(continued from previous page)

```
        PRF = np.sum(self.Q, axis=1)
        self.PRF = PRF

        return PRF

    def risk_free_rate(self):
        "Given Q, compute one-period gross risk-free interest rate R"

        R = np.sum(self.Q, axis=1)
        R = np.reciprocal(R)
        self.R = R

        return R

    def value_functionss(self):
        "Given α, compute the optimal value functions J in equilibrium"

        n, T = self.n, self.T
        β = self.β
        P = self.P

        # compute (I - βP)^(-1) in infinite case
        if T is None:
            P_seq = np.empty((1, n, n))
            P_seq[0] = np.linalg.inv(np.eye(n) - β * P)
        # and (I + βP + ... + β^T P^T) in finite case
        else:
            P_seq = np.empty((T+1, n, n))
            P_seq[0] = np.eye(n)

            Pt = np.eye(n)
            for t in range(1, T+1):
                Pt = Pt.dot(P)
                P_seq[t] = P_seq[t-1] + Pt * β ** t

        # compute the matrix [u(a_1 y), ..., u(a_K, y)]
        flow = np.empty((n, K))
        for k in range(K):
            flow[:, k] = self.u(self.α[k] * self.y)

        J = P_seq @ flow

        self.J = J

        return J
```

120.9 Examples

We'll use our code to construct equilibrium objects in several example economies.

Our first several examples will be infinite horizon economies.

Our final example will be a finite horizon economy.

120.9.1 Example 1

Please read the preceding class for default parameter values and the following Python code for the fundamentals of the economy.

Here goes.

```
# dimensions
K, n = 2, 2

# states
s = np.array([0, 1])

# transition
P = np.array([[.5, .5], [.5, .5]])

# endowments
ys = np.empty((n, K))
ys[:, 0] = 1 - s        # y1
ys[:, 1] = s            # y2
```

```
ex1 = RecurCompetitive(s, P, ys)
```

```
# endowments
ex1.ys
```

```
array([[1., 0.],
       [0., 1.]])
```

```
# pricing kernal
ex1.Q
```

```
array([[0.49, 0.49],
       [0.49, 0.49]])
```

```
# Risk free rate R
ex1.R
```

```
array([1.02040816, 1.02040816])
```

```
# natural debt limit, A = [A1, A2, ..., AI]
ex1.A
```

```
array([[25.5, 24.5],
       [24.5, 25.5]])
```

```
# when the initial state is state 1
print(f'α = {ex1.wealth_distribution(s0_idx=0)}')
print(f'ψ = \n{ex1.continuation_wealths()}')
print(f'J = \n{ex1.value_functionss()}')
```

```
α = [0.51 0.49]
ψ =
[[[-0.  0.]
  [ 1. -1.]]]
J =
[[[71.41428429 70.        ]
  [71.41428429 70.        ]]]
```

```
# when the initial state is state 2
print(f'α = {ex1.wealth_distribution(s0_idx=1)}')
print(f'ψ = \n{ex1.continuation_wealths()}')
print(f'J = \n{ex1.value_functionss()}')
```

```
α = [0.49 0.51]
ψ =
[[[-1.  1.]
  [ 0. -0.]]]
J =
[[[70.         71.41428429]
  [70.         71.41428429]]]
```

120.9.2 Example 2

```
# dimensions
K, n = 2, 2

# states
s = np.array([1, 2])

# transition
P = np.array([[.5, .5], [.5, .5]])

# endowments
ys = np.empty((n, K))
ys[:, 0] = 1.5          # y1
ys[:, 1] = s            # y2
```

```
ex2 = RecurCompetitive(s, P, ys)
```

```
# endowments

print("ys = \n", ex2.ys)

# pricing kernal
print ("Q = \n", ex2.Q)

# Risk free rate R
print("R = ", ex2.R)
```

```
ys =
 [[1.5 1. ]
 [1.5 2. ]]
Q =
 [[0.49       0.41412558]
 [0.57977582 0.49      ]]
R =  [1.10604104 0.93477529]
```

```
# pricing kernal
ex2.Q
```

```
array([[0.49      , 0.41412558],
       [0.57977582, 0.49      ]])
```

Note that the pricing kernal in example economies 1 and 2 differ.

This comes from differences in the aggregate endowments in state 1 and 2 in example 1.

```
ex2.β * ex2.u_prime(3.5) / ex2.u_prime(2.5) * ex2.P[0,1]
```

```
np.float64(0.4141255848169731)
```

```
ex2.β * ex2.u_prime(2.5) / ex2.u_prime(3.5) * ex2.P[1,0]
```

```
np.float64(0.5797758187437624)
```

```
# Risk free rate R
ex2.R
```

```
array([1.10604104, 0.93477529])
```

```
# natural debt limit, A = [A1, A2, ..., AI]
ex2.A
```

```
array([[69.30941886, 66.91255848],
       [81.73318641, 79.98879094]])
```

```
# when the initial state is state 1
print(f'α = {ex2.wealth_distribution(s0_idx=0)}')
print(f'ψ = \n{ex2.continuation_wealths()}')
print(f'J = \n{ex2.value_functionss()}')
```

```
α = [0.50879763 0.49120237]
ψ =
[[[ 0.          0.        ]
  [ 0.55057195 -0.55057195]]]
J =
[[[122.907875   120.76397493]
  [123.32114686 121.17003803]]]
```

```
# when the initial state is state 1
print(f'α = {ex2.wealth_distribution(s0_idx=1)}')
print(f'ψ = \n{ex2.continuation_wealths()}')
print(f'J = \n{ex2.value_functionss()}')
```

```
α = [0.50539319 0.49460681]
ψ =
[[[-0.46375886  0.46375886]
  [-0.         -0.        ]]]
J =
[[[122.49598809 121.18174895]
  [122.907875   121.58921679]]]
```

120.9.3 Example 3

```
# dimensions
K, n = 2, 2

# states
s = np.array([1, 2])

# transition
λ = 0.9
P = np.array([[1-λ, λ], [0, 1]])

# endowments
ys = np.empty((n, K))
ys[:, 0] = [1, 0]          # y1
ys[:, 1] = [0, 1]          # y2
```

```
ex3 = RecurCompetitive(s, P, ys)
```

```
# endowments

print("ys = ", ex3.ys)

# pricing kernel
print ("Q = ", ex3.Q)

# Risk free rate R
print("R = ", ex3.R)
```

```
ys =  [[1. 0.]
 [0. 1.]]
Q =  [[0.098 0.882]
 [0.    0.98 ]]
R =  [1.02040816 1.02040816]
```

```
# pricing kernel
ex3.Q
```

```
array([[0.098, 0.882],
       [0.   , 0.98 ]])
```

```
# natural debt limit, A = [A1, A2, ..., AI]
ex3.A
```

```
array([[ 1.10864745, 48.89135255],
       [ 0.        , 50.        ]])
```

Note that the natural debt limit for agent 1 in state 2 is 0.

```
# when the initial state is state 1
print(f'α = {ex3.wealth_distribution(s0_idx=0)}')
print(f'ψ = \n{ex3.continuation_wealths()}')
print(f'J = \n{ex3.value_functionss()}')
```

```
α = [0.02217295 0.97782705]
ψ =
[[[ 0.         -0.        ]
  [ 1.10864745 -1.10864745]]]
J =
[[[14.89058394 98.88513796]
  [14.89058394 98.88513796]]]
```

```
# when the initial state is state 1
print(f'α = {ex3.wealth_distribution(s0_idx=1)}')
print(f'ψ = \n{ex3.continuation_wealths()}')
print(f'J = \n{ex3.value_functionss()}')
```

```
α = [0. 1.]
ψ =
[[[-1.10864745  1.10864745]
  [ 0.          0.        ]]]
J =
[[[  0. 100.]
  [  0. 100.]]]
```

For the specification of the Markov chain in example 3, let's take a look at how the equilibrium allocation changes as a function of transition probability λ.

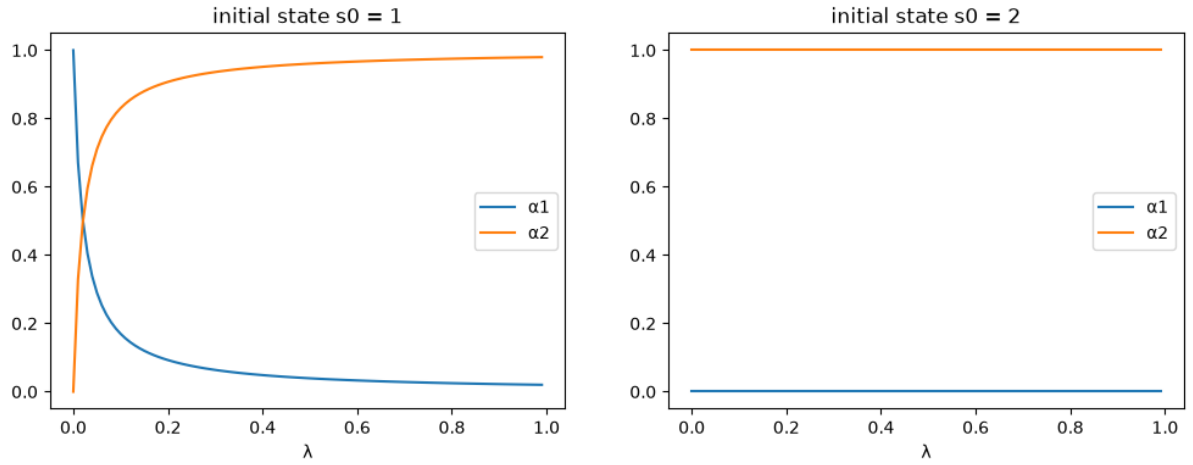
```
λ_seq = np.linspace(0, 0.99, 100)

# prepare containers
as0_seq = np.empty((len(λ_seq), 2))
as1_seq = np.empty((len(λ_seq), 2))

for i, λ in enumerate(λ_seq):
    P = np.array([[1-λ, λ], [0, 1]])
    ex3 = RecurCompetitive(s, P, ys)

    # initial state s0 = 1
    α = ex3.wealth_distribution(s0_idx=0)
    as0_seq[i, :] = α

    # initial state s0 = 2
    α = ex3.wealth_distribution(s0_idx=1)
    as1_seq[i, :] = α
```

```
fig, axs = plt.subplots(1, 2, figsize=(12, 4))

for i, as_seq in enumerate([as0_seq, as1_seq]):
    for j in range(2):
```

(continues on next page)

(continued from previous page)

```
        axs[i].plot(λ_seq, αs_seq[:, j], label=f'α{j+1}')
        axs[i].set_xlabel('λ')
        axs[i].set_title(f'initial state s0 = {s[i]}')
        axs[i].legend()

plt.show()
```



120.9.4 Example 4

```
# dimensions
K, n = 2, 3

# states
s = np.array([1, 2, 3])

# transition
λ = .9
μ = .9
δ = .05

# prosperous, moderate, and recession states
P = np.array([[1-λ, λ, 0], [μ/2, μ, μ/2], [(1-δ)/2, (1-δ)/2, δ]])

# endowments
ys = np.empty((n, K))
ys[:, 0] = [.25, .75, .2]        # y1
ys[:, 1] = [1.25, .25, .2]       # y2
```

```
ex4 = RecurCompetitive(s, P, ys)
```

```
# endowments
print("ys = \n", ex4.ys)

# pricing kernal
print ("Q = \n", ex4.Q)

# Risk free rate R
```

(continues on next page)

(continued from previous page)

```
print("R = ", ex4.R)

# natural debt limit, A = [A1, A2, ..., AI]
print("A = \n", ex4.A)

print('')

for i in range(1, 4):
    # when the initial state is state i
    print(f"when the initial state is state {i}")
    print(f'α = {ex4.wealth_distribution(s0_idx=i-1)}')
    print(f'ψ = \n{ex4.continuation_wealths()}')
    print(f'J = \n{ex4.value_functionss()}\n')
```

```
ys =
 [[0.25 1.25]
 [0.75 0.25]
 [0.2  0.2 ]]
Q =
 [[0.098      1.08022498 0.        ]
 [0.36007499 0.882      0.69728222]
 [0.24038317 0.29440805 0.049     ]]
R =  [0.84873434 0.51563476 1.71294115]
A =
 [[-1.4141307  -0.45854174]
 [-1.4122483  -1.54005386]
 [-0.58434331 -0.3823659 ]]

when the initial state is state 1
α = [0.75514045 0.24485955]
ψ =
[[[ 0.          0.        ]
  [-0.81715447  0.81715447]
  [-0.14565791  0.14565791]]]
J =
[[[-2.65741909 -1.51322919]
  [-5.13103133 -2.92179221]
  [-2.65649938 -1.51270548]]]

when the initial state is state 2
α = [0.47835493 0.52164507]
ψ =
[[[ 0.5183286  -0.5183286 ]
  [ 0.         -0.        ]
  [ 0.12191319 -0.12191319]]]
J =
[[[-2.11505328 -2.20868477]
  [-4.08381377 -4.26460049]
  [-2.11432128 -2.20792037]]]

when the initial state is state 3
α = [0.60446648 0.39553352]
ψ =
[[[ 0.28216299 -0.28216299]
  [-0.37231938  0.37231938]
  [ 0.         -0.        ]]]
```

(continues on next page)

(continued from previous page)

```
J =
[[[-2.37756442 -1.92325926]
  [-4.59067883 -3.71349163]
  [-2.37674158 -1.92259365]]]
```

120.9.5 Finite Horizon Example

We now revisit the economy defined in example 1, but set the time horizon to be $T = 10$.

```
# dimensions
K, n = 2, 2

# states
s = np.array([0, 1])

# transition
P = np.array([[.5, .5], [.5, .5]])

# endowments
ys = np.empty((n, K))
ys[:, 0] = 1 - s        # y1
ys[:, 1] = s            # y2
```

```
ex1_finite = RecurCompetitive(s, P, ys, T=10)
```

```
# (I + Q + Q^2 + ... + Q^T)
ex1_finite.V[-1]
```

```
array([[5.48171623, 4.48171623],
       [4.48171623, 5.48171623]])
```

```
# endowments
ex1_finite.ys
```

```
array([[1., 0.],
       [0., 1.]])
```

```
# pricing kernal
ex1_finite.Q
```

```
array([[0.49, 0.49],
       [0.49, 0.49]])
```

```
# Risk free rate R
ex1_finite.R
```

```
array([1.02040816, 1.02040816])
```

In the finite time horizon case, ψ and $\jmath$ are returned as sequences.

Components are ordered from $t = T$ to $t = 0$.

```
# when the initial state is state 2
print(f'α = {ex1_finite.wealth_distribution(s0_idx=0)}')
print(f'ψ = \n{ex1_finite.continuation_wealths()}\n')
print(f'J = \n{ex1_finite.value_functionss()}')
```

```
α = [0.55018351 0.44981649]
ψ =
[[[-0.44981649  0.44981649]
  [ 0.55018351 -0.55018351]]

 [[-0.40063665  0.40063665]
  [ 0.59936335 -0.59936335]]

 [[-0.35244041  0.35244041]
  [ 0.64755959 -0.64755959]]

 [[-0.30520809  0.30520809]
  [ 0.69479191 -0.69479191]]

 [[-0.25892042  0.25892042]
  [ 0.74107958 -0.74107958]]

 [[-0.21355851  0.21355851]
  [ 0.78644149 -0.78644149]]

 [[-0.16910383  0.16910383]
  [ 0.83089617 -0.83089617]]

 [[-0.12553824  0.12553824]
  [ 0.87446176 -0.87446176]]

 [[-0.08284397  0.08284397]
  [ 0.91715603 -0.91715603]]

 [[-0.04100358  0.04100358]
  [ 0.95899642 -0.95899642]]

 [[-0.         -0.        ]
  [ 1.         -1.        ]]]

J =
[[[ 1.48348712  1.3413672 ]
  [ 1.48348712  1.3413672 ]]

 [[ 2.9373045   2.65590706]
  [ 2.9373045   2.65590706]]

 [[ 4.36204553  3.94415611]
  [ 4.36204553  3.94415611]]

 [[ 5.75829174  5.20664019]
  [ 5.75829174  5.20664019]]

 [[ 7.12661302  6.44387459]
  [ 7.12661302  6.44387459]]

 [[ 8.46756788  7.6563643 ]
```

(continues on next page)

(continued from previous page)

```
  [ 8.46756788  7.6563643 ]]

 [[ 9.78170364  8.84460421]
  [ 9.78170364  8.84460421]]

 [[11.06955669 10.00907933]
  [11.06955669 10.00907933]]

 [[12.33165268 11.15026494]
  [12.33165268 11.15026494]]

 [[13.56850674 12.26862684]
  [13.56850674 12.26862684]]

 [[14.78062373 13.3646215 ]
  [14.78062373 13.3646215 ]]]
```

```
# when the initial state is state 2
print(f'α = {ex1_finite.wealth_distribution(s0_idx=1)}')
print(f'ψ = \n{ex1_finite.continuation_wealths()}\n')
print(f'J = \n{ex1_finite.value_functionss()}')
```

```
α = [0.44981649 0.55018351]
ψ =
[[[-0.55018351  0.55018351]
  [ 0.44981649 -0.44981649]]

 [[-0.59936335  0.59936335]
  [ 0.40063665 -0.40063665]]

 [[-0.64755959  0.64755959]
  [ 0.35244041 -0.35244041]]

 [[-0.69479191  0.69479191]
  [ 0.30520809 -0.30520809]]

 [[-0.74107958  0.74107958]
  [ 0.25892042 -0.25892042]]

 [[-0.78644149  0.78644149]
  [ 0.21355851 -0.21355851]]

 [[-0.83089617  0.83089617]
  [ 0.16910383 -0.16910383]]

 [[-0.87446176  0.87446176]
  [ 0.12553824 -0.12553824]]

 [[-0.91715603  0.91715603]
  [ 0.08284397 -0.08284397]]

 [[-0.95899642  0.95899642]
  [ 0.04100358 -0.04100358]]

 [[-1.          1.        ]
  [-0.         -0.        ]]]
```

(continues on next page)

(continued from previous page)

```
J =
[[[ 1.3413672   1.48348712]
  [ 1.3413672   1.48348712]]

 [[ 2.65590706  2.9373045 ]
  [ 2.65590706  2.9373045 ]]

 [[ 3.94415611  4.36204553]
  [ 3.94415611  4.36204553]]

 [[ 5.20664019  5.75829174]
  [ 5.20664019  5.75829174]]

 [[ 6.44387459  7.12661302]
  [ 6.44387459  7.12661302]]

 [[ 7.6563643   8.46756788]
  [ 7.6563643   8.46756788]]

 [[ 8.84460421  9.78170364]
  [ 8.84460421  9.78170364]]

 [[10.00907933 11.06955669]
  [10.00907933 11.06955669]]

 [[11.15026494 12.33165268]
  [11.15026494 12.33165268]]

 [[12.26862684 13.56850674]
  [12.26862684 13.56850674]]

 [[13.3646215  14.78062373]
  [13.3646215  14.78062373]]]
```

We can check the results with finite horizon converges to the ones with infinite horizon as $T \to \infty$.

```
ex1_large = RecurCompetitive(s, P, ys, T=10000)
ex1_large.wealth_distribution(s0_idx=1)
```

```
array([0.49, 0.51])
```

```
ex1.V, ex1_large.V[-1]
```

```
(array([[[25.5, 24.5],
         [24.5, 25.5]]]),
 array([[25.5, 24.5],
        [24.5, 25.5]]))
```

```
ex1_large.continuation_wealths()
ex1.ψ, ex1_large.ψ[-1]
```

```
(array([[[-1.,  1.],
         [ 0., -0.]]]),
 array([[-1.,  1.],
```

(continues on next page)

(continued from previous page)

```
        [ 0., -0.]]))
```

```
ex1_large.value_functionss()
ex1.J, ex1_large.J[-1]
```

```
(array([[[70.        , 71.41428429],
         [70.        , 71.41428429]]]),
 array([[70.        , 71.41428429],
        [70.        , 71.41428429]]))
```

CHAPTER

ONE

HETEROGENEOUS BELIEFS AND BUBBLES

Contents

- *Heterogeneous Beliefs and Bubbles*
  - *Overview*
  - *Structure of the Model*
  - *Solving the Model*

In addition to what's in Anaconda, this lecture uses following libraries:

```
!pip install quantecon
```

121.1 Overview

This lecture describes a version of a model of Harrison and Kreps [Harrison and Kreps, 1978].

The model determines the price of a dividend-yielding asset that is traded by two types of self-interested investors.

The model features

- heterogeneous beliefs
- incomplete markets
- short sales constraints, and possibly ...
- (leverage) limits on an investor's ability to borrow in order to finance purchases of a risky asset

Let's start with some standard imports:

```
import numpy as np
import quantecon as qe
import scipy.linalg as la
```

121.1.1 References

Prior to reading the following, you might like to review our lectures on

- *Markov chains*
- *Asset pricing with finite state space*

121.1.2 Bubbles

Economists differ in how they define a *bubble*.

The Harrison-Kreps model illustrates the following notion of a bubble that attracts many economists:

> *A component of an asset price can be interpreted as a bubble when all investors agree that the current price of the asset exceeds what they believe the asset's underlying dividend stream justifies*.

121.2 Structure of the Model

The model simplifies things by ignoring alterations in the distribution of wealth among investors who have hard-wired different beliefs about the fundamentals that determine asset payouts.

There is a fixed number A of shares of an asset.

Each share entitles its owner to a stream of dividends $\{d_t\}$ governed by a Markov chain defined on a state space $S \in \{0, 1\}$.

The dividend obeys

$$d_t = \begin{cases} 0 & \text{if } s_t = 0 \\ 1 & \text{if } s_t = 1 \end{cases}$$

An owner of a share at the end of time t and the beginning of time $t+1$ is entitled to the dividend paid at time $t+1$.

Thus, the stock is traded **ex dividend**.

An owner of a share at the beginning of time $t+1$ is also entitled to sell the share to another investor during time $t+1$.

Two types $h = a, b$ of investors differ only in their beliefs about a Markov transition matrix P with typical element

$$P(i,j) = \mathbb{P}\{s_{t+1} = j \mid s_t = i\}$$

Investors of type a believe the transition matrix

$$P_a = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \\ \frac{2}{3} & \frac{1}{3} \end{bmatrix}$$

Investors of type b think the transition matrix is

$$P_b = \begin{bmatrix} \frac{2}{3} & \frac{1}{3} \\ \frac{1}{4} & \frac{3}{4} \end{bmatrix}$$

Thus, in state 0, a type a investor is more optimistic about next period's dividend than is investor b.

But in state 1, a type a investor is more pessimistic about next period's dividend than is investor b.

The stationary (i.e., invariant) distributions of these two matrices can be calculated as follows:

```
qa = np.array([[1/2, 1/2], [2/3, 1/3]])
qb = np.array([[2/3, 1/3], [1/4, 3/4]])
mca = qe.MarkovChain(qa)
mcb = qe.MarkovChain(qb)
mca.stationary_distributions
```

```
array([[0.57142857, 0.42857143]])
```

```
mcb.stationary_distributions
```

```
array([[0.42857143, 0.57142857]])
```

The stationary distribution of P_a is approximately $\pi_a = \begin{bmatrix} .57 & .43 \end{bmatrix}$.

The stationary distribution of P_b is approximately $\pi_b = \begin{bmatrix} .43 & .57 \end{bmatrix}$.

Thus, a type a investor is more pessimistic on average.

121.2.1 Ownership Rights

An owner of the asset at the end of time t is entitled to the dividend at time $t+1$ and also has the right to sell the asset at time $t+1$.

Both types of investors are risk-neutral and both have the same fixed discount factor $\beta \in (0,1)$.

In our numerical example, we'll set $\beta = .75$, just as Harrison and Kreps [Harrison and Kreps, 1978] did.

We'll eventually study the consequences of two alternative assumptions about the number of shares A relative to the resources that our two types of investors can invest in the stock.

1. Both types of investors have enough resources (either wealth or the capacity to borrow) so that they can purchase the entire available stock of the asset[1].
2. No single type of investor has sufficient resources to purchase the entire stock.

Case 1 is the case studied in Harrison and Kreps.

In case 2, both types of investors always hold at least some of the asset.

121.2.2 Short Sales Prohibited

No short sales are allowed.

This matters because it limits how pessimists can express their opinions.

- They **can** express themselves by selling their shares.
- They **cannot** express themsevles more loudly by artificially "manufacturing shares" – that is, they cannot borrow shares from more optimistic investors and then immediately sell them.

[1] By assuming that both types of agents always have "deep enough pockets" to purchase all of the asset, the model takes wealth dynamics off the table. The Harrison-Kreps model generates high trading volume when the state changes either from 0 to 1 or from 1 to 0.

121.2.3 Optimism and Pessimism

The above specifications of the perceived transition matrices P_a and P_b, taken directly from Harrison and Kreps, build in stochastically alternating temporary optimism and pessimism.

Remember that state 1 is the high dividend state.

- In state 0, a type a agent is more optimistic about next period's dividend than a type b agent.
- In state 1, a type b agent is more optimistic about next period's dividend than a type a agaub is.

However, the stationary distributions $\pi_a = \begin{bmatrix} .57 & .43 \end{bmatrix}$ and $\pi_b = \begin{bmatrix} .43 & .57 \end{bmatrix}$ tell us that a type b person is more optimistic about the dividend process in the long run than is a type a person.

121.2.4 Information

Investors know a price function mapping the state s_t at t into the equilibrium price $p(s_t)$ that prevails in that state.

This price function is endogenous and to be determined below.

When investors choose whether to purchase or sell the asset at t, they also know s_t.

121.3 Solving the Model

Now let's turn to solving the model.

We'll determine equilibrium prices under a particular specification of beliefs and constraints on trading selected from one of the specifications described above.

We shall compare equilibrium price functions under the following alternative assumptions about beliefs:

1. There is only one type of agent, either a or b.
2. There are two types of agents differentiated only by their beliefs. Each type of agent has sufficient resources to purchase all of the asset (Harrison and Kreps's setting).
3. There are two types of agents with different beliefs, but because of limited wealth and/or limited leverage, both types of investors hold the asset each period.

121.3.1 Summary Table

The following table gives a summary of the findings obtained in the remainder of the lecture (in an exercise you will be asked to recreate the table and also reinterpret parts of it).

The table reports implications of Harrison and Kreps's specifications of P_a, P_b, β.

s_t	0	1
p_a	1.33	1.22
p_b	1.45	1.91
p_o	1.85	2.08
p_p	1	1
$\hat{p}_a$	1.85	1.69
$\hat{p}_b$	1.69	2.08

Here

- p_a is the equilibrium price function under homogeneous beliefs P_a
- p_b is the equilibrium price function under homogeneous beliefs P_b
- p_o is the equilibrium price function under heterogeneous beliefs with optimistic marginal investors
- p_p is the equilibrium price function under heterogeneous beliefs with pessimistic marginal investors
- $\hat{p}_a$ is the amount type a investors are willing to pay for the asset
- $\hat{p}_b$ is the amount type b investors are willing to pay for the asset

We'll explain these values and how they are calculated one row at a time.

The row corresponding to p_o applies when both types of investor have enough resources to purchase the entire stock of the asset and strict short sales constraints prevail so that temporarily optimistic investors always price the asset.

The row corresponding to p_p would apply if neither type of investor has enough resources to purchase the entire stock of the asset and both types must hold the asset.

The row corresponding to p_p would also apply if both types have enough resources to buy the entire stock of the asset but short sales are also possible so that temporarily pessimistic investors price the asset.

121.3.2 Single Belief Prices

We'll start by pricing the asset under homogeneous beliefs.

(This is the case treated in *the lecture* on asset pricing with finite Markov states)

Suppose that there is only one type of investor, either of type a or b, and that this investor always "prices the asset".

Let $p_h = \begin{bmatrix} p_h(0) \\ p_h(1) \end{bmatrix}$ be the equilibrium price vector when all investors are of type h.

The price today equals the expected discounted value of tomorrow's dividend and tomorrow's price of the asset:

$$p_h(s) = \beta \left(P_h(s,0)(0 + p_h(0)) + P_h(s,1)(1 + p_h(1)) \right), \quad s = 0, 1 \tag{121.1}$$

These equations imply that the equilibrium price vector is

$$\begin{bmatrix} p_h(0) \\ p_h(1) \end{bmatrix} = \beta [I - \beta P_h]^{-1} P_h \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{121.2}$$

The first two rows of the table report $p_a(s)$ and $p_b(s)$.

Here's a function that can be used to compute these values

```
def price_single_beliefs(transition, dividend_payoff, β=.75):
    """
    Function to Solve Single Beliefs
    """
    # First compute inverse piece
    imbq_inv = la.inv(np.eye(transition.shape[0]) - β * transition)

    # Next compute prices
    prices = β * imbq_inv @ transition @ dividend_payoff

    return prices
```

Single Belief Prices as Benchmarks

These equilibrium prices under homogeneous beliefs are important benchmarks for the subsequent analysis.

- $p_h(s)$ tells what a type h investor thinks is the "fundamental value" of the asset.
- Here "fundamental value" means the expected discounted present value of future dividends.

We will compare these fundamental values of the asset with equilibrium values when traders have different beliefs.

121.3.3 Pricing under Heterogeneous Beliefs

There are several cases to consider.

The first is when both types of agents have sufficient wealth to purchase all of the asset themselves.

In this case, the marginal investor who prices the asset is the more optimistic type so that the equilibrium price $\bar{p}$ satisfies Harrison and Kreps's key equation:

$$\bar{p}(s) = \beta \max \left\{ P_a(s,0)\bar{p}(0) + P_a(s,1)(1+\bar{p}(1)),\ P_b(s,0)\bar{p}(0) + P_b(s,1)(1+\bar{p}(1)) \right\} \tag{121.3}$$

for $s = 0, 1$.

In the above equation, the max on the right side is over the two prospective values of next period's payout from owning the asset.

The marginal investor who prices the asset in state s is of type a if

$$P_a(s,0)\bar{p}(0) + P_a(s,1)(1+\bar{p}(1)) > P_b(s,0)\bar{p}(0) + P_b(s,1)(1+\bar{p}(1))$$

The marginal investor is of type b if

$$P_a(s,1)\bar{p}(0) + P_a(s,1)(1+\bar{p}(1)) < P_b(s,1)\bar{p}(0) + P_b(s,1)(1+\bar{p}(1))$$

Thus the marginal investor is the (temporarily) optimistic type.

Equation (121.3) is a functional equation that, like a Bellman equation, can be solved by

- starting with a guess for the price vector $\bar{p}$ and
- iterating to convergence on the operator that maps a guess $\bar{p}^j$ into an updated guess $\bar{p}^{j+1}$ defined by the right side of (121.3), namely

$$\bar{p}^{j+1}(s) = \beta \max \left\{ P_a(s,0)\bar{p}^j(0) + P_a(s,1)(1+\bar{p}^j(1)),\ P_b(s,0)\bar{p}^j(0) + P_b(s,1)(1+\bar{p}^j(1)) \right\} \tag{121.4}$$

for $s = 0, 1$.

The third row of the table labeled p_o reports equilibrium prices that solve the functional equation when $\beta = .75$.

Here the type that is optimistic about s_{t+1} prices the asset in state s_t.

It is instructive to compare these prices with the equilibrium prices for the homogeneous belief economies that solve under beliefs P_a and P_b reported in the rows labeled p_a and p_b, respectively.

Equilibrium prices p_o in the heterogeneous beliefs economy evidently exceed what any prospective investor regards as the fundamental value of the asset in each possible state.

Nevertheless, the economy recurrently visits a state that makes each investor want to purchase the asset for more than he believes its future dividends are worth.

An investor is willing to pay more than what he believes is warranted by fundamental value of the prospective dividend stream because he expects to have the option later to sell the asset to another investor who will value the asset more highly than he will then.

- Investors of type a are willing to pay the following price for the asset

$$\hat{p}_a(s) = \begin{cases} \bar{p}(0) & \text{if } s_t = 0 \\ \beta(P_a(1,0)\bar{p}(0) + P_a(1,1)(1+\bar{p}(1))) & \text{if } s_t = 1 \end{cases}$$

- Investors of type b are willing to pay the following price for the asset

$$\hat{p}_b(s) = \begin{cases} \beta(P_b(0,0)\bar{p}(0) + P_b(0,1)(1+\bar{p}(1))) & \text{if } s_t = 0 \\ \bar{p}(1) & \text{if } s_t = 1 \end{cases}$$

Evidently, $\hat{p}_a(1) < \bar{p}(1)$ and $\hat{p}_b(0) < \bar{p}(0)$.

Investors of type a want to sell the asset in state 1 while investors of type b want to sell it in state 0.

- The asset changes hands whenever the state changes from 0 to 1 or from 1 to 0.
- The valuations $\hat{p}_a(s)$ and $\hat{p}_b(s)$ are displayed in the fourth and fifth rows of the table.
- Even pessimistic investors who don't buy the asset think that it is worth more than they think future dividends are worth.

Here's code to solve for $\bar{p}$, $\hat{p}_a$ and $\hat{p}_b$ using the iterative method described above

```
def price_optimistic_beliefs(transitions, dividend_payoff, β=.75,
                             max_iter=50000, tol=1e-16):
    """
    Function to Solve Optimistic Beliefs
    """
    # We will guess an initial price vector of [0, 0]
    p_new = np.array([[0], [0]])
    p_old = np.array([[10.], [10.]])

    # We know this is a contraction mapping, so we can iterate to conv
    for i in range(max_iter):
        p_old = p_new
        p_new = β * np.max([q @ p_old
                            + q @ dividend_payoff for q in transitions],
                            axis=0)

        # If we succeed in converging, break out of for loop
        if np.max(np.sqrt((p_new - p_old)**2)) < tol:
            break

    ptwiddle = β * np.min([q @ p_old
                           + q @ dividend_payoff for q in transitions],
                           axis=0)

    phat_a = np.array([p_new[0], ptwiddle[1]])
    phat_b = np.array([ptwiddle[0], p_new[1]])

    return p_new, phat_a, phat_b
```

121.3.4 Insufficient Funds

Outcomes differ when the more optimistic type of investor has insufficient wealth — or insufficient ability to borrow enough — to hold the entire stock of the asset.

In this case, the asset price must adjust to attract pessimistic investors.

Instead of equation (121.3), the equilibrium price satisfies

$$\check{p}(s) = \beta \min \{P_a(s,0)\check{p}(0) + P_a(s,1)(1+\check{p}(1)),\ P_b(s,0)\check{p}(0) + P_b(s,1)(1+\check{p}(1))\} \tag{121.5}$$

and the marginal investor who prices the asset is always the one that values it *less* highly than does the other type.

Now the marginal investor is always the (temporarily) pessimistic type.

Notice from the sixth row of that the pessimistic price p_o is lower than the homogeneous belief prices p_a and p_b in both states.

When pessimistic investors price the asset according to (121.5), optimistic investors think that the asset is underpriced.

If they could, optimistic investors would willingly borrow at a one-period risk-free gross interest rate β^{-1} to purchase more of the asset.

Implicit constraints on leverage prohibit them from doing so.

When optimistic investors price the asset as in equation (121.3), pessimistic investors think that the asset is overpriced and would like to sell the asset short.

Constraints on short sales prevent that.

Here's code to solve for $\check{p}$ using iteration

```
def price_pessimistic_beliefs(transitions, dividend_payoff, β=.75,
                              max_iter=50000, tol=1e-16):
    """
    Function to Solve Pessimistic Beliefs
    """
    # We will guess an initial price vector of [0, 0]
    p_new = np.array([[0], [0]])
    p_old = np.array([[10.], [10.]])

    # We know this is a contraction mapping, so we can iterate to conv
    for i in range(max_iter):
        p_old = p_new
        p_new = β * np.min([q @ p_old
                            + q @ dividend_payoff for q in transitions],
                           axis=0)

        # If we succeed in converging, break out of for loop
        if np.max(np.sqrt((p_new - p_old)**2)) < tol:
            break

    return p_new
```

121.3.5 Further Interpretation

Jose Scheinkman [Scheinkman, 2014] interprets the Harrison-Kreps model as a model of a bubble — a situation in which an asset price exceeds what every investor thinks is merited by his or her beliefs about the value of the asset's underlying dividend stream.

Scheinkman stresses these features of the Harrison-Kreps model:

- High volume occurs when the Harrison-Kreps pricing formula (121.3) prevails.
- Type a investors sell the entire stock of the asset to type b investors every time the state switches from $s_t = 0$ to $s_t = 1$.
- Type b investors sell the asset to type a investors every time the state switches from $s_t = 1$ to $s_t = 0$.

Scheinkman takes this as a strength of the model because he observes high volume during *famous bubbles*.

- If the *supply* of the asset is increased sufficiently either physically (more "houses" are built) or artificially (ways are invented to short sell "houses"), bubbles end when the asset supply has grown enough to outstrip optimistic investors' resources for purchasing the asset.
- If optimistic investors finance their purchases by borrowing, tightening leverage constraints can extinguish a bubble.

Scheinkman extracts insights about the effects of financial regulations on bubbles.

He emphasizes how limiting short sales and limiting leverage have opposite effects.

Exercise 121.3.1

This exercise invites you to recreate the summary table using the functions we have built above.

s_t	0	1
p_a	1.33	1.22
p_b	1.45	1.91
p_o	1.85	2.08
p_p	1	1
$\hat{p}_a$	1.85	1.69
$\hat{p}_b$	1.69	2.08

You will want first to define the transition matrices and dividend payoff vector.

In addition, below we'll add an interpretation of the row corresponding to p_o by inventing two additional types of agents, one of whom is **permanently optimistic**, the other who is **permanently pessimistic**.

We construct subjective transition probability matrices for our permanently optimistic and permanently pessimistic investors as follows.

The permanently optimistic investors(i.e., the investor with the most optimistic beliefs in each state) believes the transition matrix

$$P_o = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \\ \frac{1}{4} & \frac{3}{4} \end{bmatrix}$$

The permanently pessimistic investor believes the transition matrix

$$P_p = \begin{bmatrix} \frac{2}{3} & \frac{1}{3} \\ \frac{2}{3} & \frac{1}{3} \end{bmatrix}$$

We'll use these transition matrices when we present our solution of exercise 1 below.

Solution

First, we will obtain equilibrium price vectors with homogeneous beliefs, including when all investors are optimistic or pessimistic.

```
qa = np.array([[1/2, 1/2], [2/3, 1/3]])     # Type a transition matrix
qb = np.array([[2/3, 1/3], [1/4, 3/4]])     # Type b transition matrix
# Optimistic investor transition matrix
qopt = np.array([[1/2, 1/2], [1/4, 3/4]])
# Pessimistic investor transition matrix
qpess = np.array([[2/3, 1/3], [2/3, 1/3]])

dividendreturn = np.array([[0], [1]])

transitions = [qa, qb, qopt, qpess]
labels = ['p_a', 'p_b', 'p_optimistic', 'p_pessimistic']

for transition, label in zip(transitions, labels):
    print(label)
    print("=" * 20)
    s0, s1 = np.round(price_single_beliefs(transition, dividendreturn), 2)
    print(f"State 0: {s0}")
    print(f"State 1: {s1}")
    print("-" * 20)
```

```
p_a
====================
State 0: [1.33]
State 1: [1.22]
--------------------
p_b
====================
State 0: [1.45]
State 1: [1.91]
--------------------
p_optimistic
====================
State 0: [1.85]
State 1: [2.08]
--------------------
p_pessimistic
====================
State 0: [1.]
State 1: [1.]
--------------------
```

We will use the price_optimistic_beliefs function to find the price under heterogeneous beliefs.

```
opt_beliefs = price_optimistic_beliefs([qa, qb], dividendreturn)
labels = ['p_optimistic', 'p_hat_a', 'p_hat_b']

for p, label in zip(opt_beliefs, labels):
    print(label)
    print("=" * 20)
    s0, s1 = np.round(p, 2)
    print(f"State 0: {s0}")
    print(f"State 1: {s1}")
    print("-" * 20)
```

```
p_optimistic
====================
State 0: [1.85]
State 1: [2.08]
--------------------
p_hat_a
====================
State 0: [1.85]
State 1: [1.69]
--------------------
p_hat_b
====================
State 0: [1.69]
State 1: [2.08]
--------------------
```

Notice that the equilibrium price with heterogeneous beliefs is equal to the price under single beliefs with **permanently optimistic** investors - this is due to the marginal investor in the heterogeneous beliefs equilibrium always being the type who is temporarily optimistic.

CHAPTER

TWO

SPECULATIVE BEHAVIOR WITH BAYESIAN LEARNING

Contents

- *Speculative Behavior with Bayesian Learning*
  - *Overview*
  - *Structure of the model*
  - *Information and beliefs*
  - *Source of heterogeneous priors*
  - *Beta priors*
  - *Market prices with learning*
  - *Two Traders*
  - *Concluding remarks*
  - *Exercise*

122.1 Overview

This lecture describes how Morris [1996] extended the Harrison–Kreps model [Harrison and Kreps, 1978] of speculative asset pricing.

Like Harrison and Kreps's model, Morris's model determines the price of a dividend-yielding asset that is traded by risk-neutral investors who have heterogeneous beliefs.

The Harrison-Kreps model assumes that the traders have dogmatic, hard-wired beliefs about the asset's dividend stream.

Morris replaced Harrison and Kreps's traders with hard-wired beliefs about the dividend stream with traders who use Bayes' Law to update their beliefs about prospective dividends as new dividend data arrive.

Note

Morris's traders don't use data on past prices of the asset to update their beliefs about the dividend process.

Key features of the environment in Morris's model include:

- All traders share a set of statistical models for prospective dividends

- A single parameter indexes the set of statistical models
- All traders observe the same dividend history
- All traders use Bayes' Law to update beliefs
- Traders have different initial *prior distributions* over the parameter
- Traders' *posterior distributions* over the parameter eventually merge
- Before their posterior distributions merge, traders disagree about the predictive density over prospective dividends
  - therefore they disagree about the value of the asset

Just as in the hard-wired beliefs model of Harrison and Kreps, those differences of opinion induce investors to engage in *speculative behavior* in the following sense:

- sometimes they are willing to pay more for the asset than what they think is its "fundamental" value, i.e., the expected discounted value of its prospective dividend stream

Prior to reading this lecture, you might want to review the following quantecon lectures:

- *Harrison-Kreps model*
- *Likelihood ratio processes*
- *Bayesian versus frequentist statistics*

Let's start with some standard imports:

```
import numpy as np
import matplotlib.pyplot as plt
```

122.2 Structure of the model

There is a fixed supply of shares of an asset.

Each share entitles its owner to a stream of *binary* i.i.d. dividends $\{d_t\}$ where

$$d_{t+1} \in \{0, 1\}$$

The dividend at time t equals 1 with unknown probability $\theta \in (0, 1)$ and equals 0 with probability $1 - \theta$.

Unlike [Harrison and Kreps, 1978] where traders have hard-wired beliefs about a Markov transition matrix, in Morris's model:

- The true dividend probability θ is unknown
- Traders have *prior beliefs* about θ
- Traders observe dividend realizations and update beliefs via Bayes' Law

There is a finite set $\mathcal{I}$ of *risk-neutral* traders.

All traders have the same discount factor $\beta \in (0, 1)$.

- You can think of β as being related to a net risk-free interest rate r by $\beta = 1/(1 + r)$.

Owning the asset at the end of period t entitles the owner to dividends at time $t + 1, t + 2, \ldots$.

Because the dividend process is i.i.d., trader i thinks that the fundamental value of the asset is the capitalized value of the dividend stream, namely, $\sum_{j=1}^{\infty} \beta^j \hat{\theta}_i = \frac{\hat{\theta}_i}{r}$, where $\hat{\theta}_i$ is the mean of the trader's posterior distribution over θ.

122.2.1 Possible trades

Traders buy and sell the risky asset in competitive markets each period $t = 0, 1, 2, ...$ after dividends are paid.

As in Harrison-Kreps:

- The asset is traded *ex dividend*
- An owner of a share at the end of time t is entitled to the dividend at time $t + 1$
- An owner of a share at the end of period t also has the right to sell the share at time $t + 1$ after having received the dividend at time $t + 1$.

Short sales are prohibited.

This matters because it limits how pessimists can express their opinions:

- They *can* express themselves by selling their shares
- They *cannot* express themselves more emphatically by borrowing shares and immediately selling them

All traders have sufficient wealth to purchase the risky asset.

122.3 Information and beliefs

At time $t \geq 1$, all traders observe $(d_1, d_2, ..., d_t)$.

All traders update their subjective distribution over θ by applying Bayes' rule.

Traders have *heterogeneous priors* over the unknown dividend probability θ.

This heterogeneity in priors produces heterogeneous posterior beliefs.

122.4 Source of heterogeneous priors

Imputing different statistical models to agents inside a model is controversial.

Many game theorists and rational expectations applied economists think it is a bad idea.

While these economists often construct models in which agents have different *information*, they prefer to assume that all of the agents inside their model always share the same statistical model – i.e., the same joint probability distribution over the random process being modeled.

For a statistician or an economic theorist, a statistical model is a joint probability distribution that is characterized by a known parameter vector.

When working with a *set* of statistical models swept out by parameters, say θ in a known set Θ, economic theorists reduce the set of models to a single model by imputing to all agents inside the model the same prior probability distribution over θ.

Note

A set of statistical models that has a particular geometric structure is called a manifold of statistical models. Morris endows traders with a shared manifold of statistical models.

Proceeding in this way adheres to the *Harsanyi Common Priors Doctrine*.

[Harsanyi, 1967], [Harsanyi, 1968], [Harsanyi, 1968] argued that if two rational agents have the same information and the same reasoning capabilities, they will have the same joint probability distribution over outcomes of interest.

Harsanyi interpreted disagreements about prospective outcomes as arising from differences in agents' information sets, not differences in their statistical models.

Evidently, [Harrison and Kreps, 1978] departed from the Harsanyi common statistical model assumption when they hard-wired dogmatic disparate beliefs.

Morris [1996] abandons the Harsanyi doctrine less completely than Harrison and Kreps had.

- Morris does assume that agents share the same set of statistical models, but ...
- Morris assumes that they have different initial prior distributions over the parameter that indexes the models

Morris's agents express their initial ignorance about the parameter differently – they have different priors.

Morris defends his assumption by alluding to the apparent "mispricing" of initial public offerings presented by [Miller, 1977].

Miller described a situation in which agents have access to little or no data about a new enterprise.

Morris wanted his traders to be open to changing their opinions as information about the parameter arrives.

Knowledgeable statisticians have been known to disagree about an appropriate prior.

For example, Morris described *different* respectable ways to express "maximal ignorance" about the parameter of a Bernoulli distribution

- a uniform distribution on $[0, 1]$
- a Jeffreys prior [Jeffreys, 1946] that is invariant to reparameterization; in the present situation, the Jeffreys prior takes the form of a Beta distribution with parameters $.5, .5$

Is one of these priors more "rational" than the other?

Morris thinks not.

122.5 Beta priors

For tractability, assume trader i has a Beta prior over the dividend probability

$$\theta \sim \text{Beta}(a_i, b_i)$$

where $a_i, b_i > 0$ are the prior parameters.

Note

The Beta distribution also appears in these quantecon lectures *Statistical Divergence Measures*, *Likelihood Ratio Processes*, *Job Search VIII: Search with Learning*.

Suppose trader i observes a history of t periods in which a total of s dividends are paid (i.e., s successes with dividend and $t - s$ failures without dividend).

By Bayes' rule, the posterior density over θ is:

$$\pi_i(\theta \mid s, t) = \frac{\theta^s (1-\theta)^{t-s} \pi_i(\theta)}{\int_0^1 \theta^s (1-\theta)^{t-s} \pi_i(\theta) d\theta}$$

where $\pi_i(\theta)$ is trader i's prior density.

Note

The Beta distribution is the conjugate prior for the Binomial likelihood. This means that when the prior is Beta(a_i, b_i) and we observe s successes in t trials, the posterior is Beta$(a_i + s, b_i + t - s)$.

The posterior mean (or expected dividend probability) is:

$$\mu_i(s,t) = \int_0^1 \theta \pi_i(\theta \mid s,t) d\theta = \mathbb{E}[\text{Beta}(a_i + s, b_i + t - s)] = \frac{a_i + s}{a_i + b_i + t}$$

Morris refers to $\mu_i(s,t)$ as trader i's **fundamental valuation** of the asset after history (s,t).

This is the probability trader i assigns to receiving a dividend next period.

It embeds trader i's updated belief about θ.

122.6 Market prices with learning

Fundamental valuations equal expected present values of dividends that our heterogeneous traders attach to the option of holding the asset *forever*.

The equilibrium price process is determined by the condition that the asset is held at time t by the trader who attaches the highest valuation to the asset at time t.

An owner of the asset has the option to sell it after receiving that period's dividend.

Traders take that into account.

That opens the possibility that a trader will be willing to pay more for the asset than that trader's fundamental valuation.

Definition 122.6.1 (Most Optimistic Valuation)

After history (s,t), the *most optimistic fundamental valuation* is:

$$\mu^*(s,t) = \max_{i \in \mathcal{I}} \mu_i(s,t)$$

Definition 122.6.2 (Equilibrium Asset Price)

Write $\tilde{p}(s,t,r)$ for the competitive equilibrium price of the risky asset (in current dollars) after history (s,t) when the interest rate is r.

The equilibrium price satisfies:

$$\tilde{p}(s,t,r) = \frac{1}{1+r}\Big[\mu^*(s,t)\{1 + \tilde{p}(s+1,t+1,r)\} + (1 - \mu^*(s,t))\tilde{p}(s,t+1,r)\Big]$$

The equilibrium price equals the highest expected discounted return among all traders from holding the asset to the next period.

Definition 122.6.3 (Normalized Price)

Define the normalized price as:

$$p(s,t,r) = r\tilde{p}(s,t,r)$$

Since the current "dollar" price of the riskless asset is $1/r$, this represents the price of the risky asset in terms of the riskless asset.

Substituting the preceding formula into the equilibrium condition gives:

$$p(s,t,r) = \frac{r}{1+r}\mu^*(s,t) + \frac{1}{1+r}\Big[\mu^*(s,t)p(s+1,t+1,r) + (1-\mu^*(s,t))p(s,t+1,r)\Big]$$

or equivalently:

$$p(s,t,r) = \mu^*(s,t) + \frac{r}{1+r}\Big[\mu^*(s,t)p(s+1,t+1,r) + (1-\mu^*(s,t))p(s,t+1,r) - \mu^*(s,t)\Big]$$

A price function that satisfies the equilibrium condition can be computed recursively.

Set $p^0(s,t,r) = 0$ for all (s,t,r), and define $p^{n+1}(s,t,r)$ by:

$$p^{n+1}(s,t,r) = \frac{r}{1+r}\mu^*(s,t) + \frac{1}{1+r}\Big[\mu^*(s,t)p^n(s+1,t+1,r) + (1-\mu^*(s,t))p^n(s,t+1,r)\Big]$$

The sequence $\{p^n(s,t,r)\}$ converges to the equilibrium price $p(s,t,r)$.

Definition 122.6.4 (Speculative Premium)

When the identity of the most optimistic trader can switch with future dividend realizations, the market price exceeds *every* trader's fundamental valuation.

In normalized units:

$$p(s,t,r) > \mu_i(s,t) \quad \text{for all } i \in \mathcal{J}$$

Define the **speculative premium** as:

$$p(s,t,r) - \mu^*(s,t) > 0$$

122.7 Two Traders

We now focus on an example with two traders with Beta priors with parameters (a_1, b_1) and (a_2, b_2).

Definition 122.7.1 (Rate Dominance (Beta Priors))

Trader 1 **rate-dominates** trader 2 if:

$$a_1 \geq a_2 \quad \text{and} \quad b_1 \leq b_2$$

Theorem 122.7.1 (Global Optimist (Two Traders))

For two traders with Beta priors:

1. If trader 1 rate-dominates trader 2, then trader 1 is a **global optimist**: $\mu_1(s,t) \geq \mu_2(s,t)$ for all histories (s,t)
2. In this case where $p(s,t,r) = \mu_1(s,t)$ for all (s,t,r), there is *no speculative premium*.

When neither trader rate-dominates the other, the identity of the most optimistic trader can switch as dividends accrue.

Along a history in which perpetual switching occurs, the price of the asset strictly exceeds both traders' fundamental valuations so long as traders continue to disagree:

$$p(s,t,r) > \max\{\mu_1(s,t), \mu_2(s,t)\}$$

Thus, along such a history, there is a persistent speculative premium.

122.7.1 Implementation

For computational tractability, let's work with a finite horizon T and solve by backward induction.

Note

On page 1122, Morris [1996] provides an argument that the limit as $T \to +\infty$ of such finite-horizon economies provides a useful selection algorithm that excludes additional equilibria that involve a Ponzi-scheme price component that Morris dismisses as fragile.

Following *Definition 122.6.2*, we use the discount factor parameterization $\beta = 1/(1+r)$ and compute dollar prices $\tilde{p}(s,t)$ via:

$$\tilde{p}(s,t) = \beta \max_{i \in \{1,2\}} \left[\mu_i(s,t)\{1 + \tilde{p}(s+1,t+1)\} + (1-\mu_i(s,t))\tilde{p}(s,t+1)\right]$$

We set the terminal price $\tilde{p}(s,T)$ equal to the perpetuity value under the most optimistic belief.

```
def posterior_mean(a, b, s, t):
    """
    Compute posterior mean μ_i(s,t) for Beta(a, b) prior.
    """
    return (a + s) / (a + b + t)

def perpetuity_value(a, b, s, t, β=.75):
    """
    Compute perpetuity value (β/(1-β)) * μ_i(s,t).
    """
    return (β / (1 - β)) * posterior_mean(a, b, s, t)

def price_learning_two_agents(prior1, prior2, β=.75, T=200):
    """
    Compute \tilde p(s,t) for two Beta-prior traders via backward induction.
    """
    a1, b1 = prior1
    a2, b2 = prior2
    price_array = np.zeros((T+1, T+1))
```

(continues on next page)

(continued from previous page)

```
    # Terminal condition: set to perpetuity value under max belief
    for s in range(T+1):
        perp1 = perpetuity_value(a1, b1, s, T, β)
        perp2 = perpetuity_value(a2, b2, s, T, β)
        price_array[s, T] = max(perp1, perp2)

    # Backward induction
    for t in range(T-1, -1, -1):
        for s in range(t, -1, -1):
            μ1 = posterior_mean(a1, b1, s, t)
            μ2 = posterior_mean(a2, b2, s, t)

            # One-step continuation values under each trader's beliefs
            cont1 = μ1 * (1.0 + price_array[s+1, t+1]) \
                    + (1.0 - μ1) * price_array[s, t+1]
            cont2 = μ2 * (1.0 + price_array[s+1, t+1]) \
                    + (1.0 - μ2) * price_array[s, t+1]
            price_array[s, t] = β * max(cont1, cont2)

    def μ1_fun(s, t):
        return posterior_mean(a1, b1, s, t)
    def μ2_fun(s, t):
        return posterior_mean(a2, b2, s, t)

    return price_array, μ1_fun, μ2_fun
```

122.7.2 Case A: global optimist (no premium)

Pick priors with rate dominance, e.g., trader 1: Beta$(a_1, b_1) = (2, 1)$ and trader 2: $(a_2, b_2) = (1, 2)$.

Trader 1 is the global optimist, so the normalized price equals trader 1's fundamental valuation: $p(s, t, r) = \mu_1(s, t)$.

```
β = 0.75
price_go, μ1_go, μ2_go = price_learning_two_agents(
        (2,1), (1,2), β=β, T=200)

perpetuity_1 = (β / (1 - β)) * μ1_go(0, 0)
perpetuity_2 = (β / (1 - β)) * μ2_go(0, 0)

print("Price at (0, 0) =", price_go[0,0])
print("Valuation of trader 1 at (0, 0) =", perpetuity_1)
print("Valuation of trader 2 at (0, 0) =", perpetuity_2)
```

```
Price at (0, 0) = 2.0
Valuation of trader 1 at (0, 0) = 2.0
Valuation of trader 2 at (0, 0) = 1.0
```

The price equals trader 1's perpetuity value.

122.7.3 Case B: perpetual switching (positive premium)

Now assume trader 1 has Beta$(1, 1)$, trader 2 has Beta$(1/2, 1/2)$.

These produce crossing posteriors, so there is no global optimist and the price exceeds both fundamentals early on.

```
price_ps, μ1_ps, μ2_ps = price_learning_two_agents(
                                (1,1), (0.5,0.5), β=β, T=200)

price_00 = price_ps[0,0]
μ1_00 = μ1_ps(0,0)
μ2_00 = μ2_ps(0,0)

perpetuity_1 = (β / (1 - β)) * μ1_ps(0, 0)
perpetuity_2 = (β / (1 - β)) * μ2_ps(0, 0)

print("Price at (0, 0) =", np.round(price_00, 6))
print("Valuation of trader 1 at (0, 0) =", perpetuity_1)
print("Valuation of trader 2 at (0, 0) =", perpetuity_2)
```

```
Price at (0, 0) = 1.599322
Valuation of trader 1 at (0, 0) = 1.5
Valuation of trader 2 at (0, 0) = 1.5
```

The resulting premium reflects the option value of reselling to whichever trader becomes temporarily more optimistic as dividends arrive sequentially.

Within this setting, we can reproduce two key figures reported in Morris [1996]

```
def normalized_price_two_agents(prior1, prior2, r, T=250):
    """Return p(s,t,r) = r \tilde p(s,t,r) for two traders."""
    β = 1.0 / (1.0 + r)
    price_array, *_ = price_learning_two_agents(prior1, prior2, β=β, T=T)
    return r * price_array

# Figure I: p*(0,0,r) as a function of r
r_grid = np.linspace(1e-3, 5.0, 200)
priors = ((1,1), (0.5,0.5))
p00 = np.array([normalized_price_two_agents(
                priors[0], priors[1], r, T=300)[0,0]
                for r in r_grid])

fig, ax = plt.subplots()
ax.plot(r_grid, p00, lw=2)
ax.set_xlabel(r'$r$')
ax.set_ylabel(r'$p^*(0,0,r)$')
ax.axhline(0.5, color='C1', linestyle='--')
plt.show()
```

In the first figure, notice that:

- The resale option pushes the normalized price $p^*(0, 0, r)$ above fundamentals (0.5) for any finite r.
- As r increases (β decreases), the option value fades and $p^*(0, 0, r) \to 0.5$.
- At $r = 0.05$ the premium is about 8–9%, consistent with Morris (1996, Section IV).

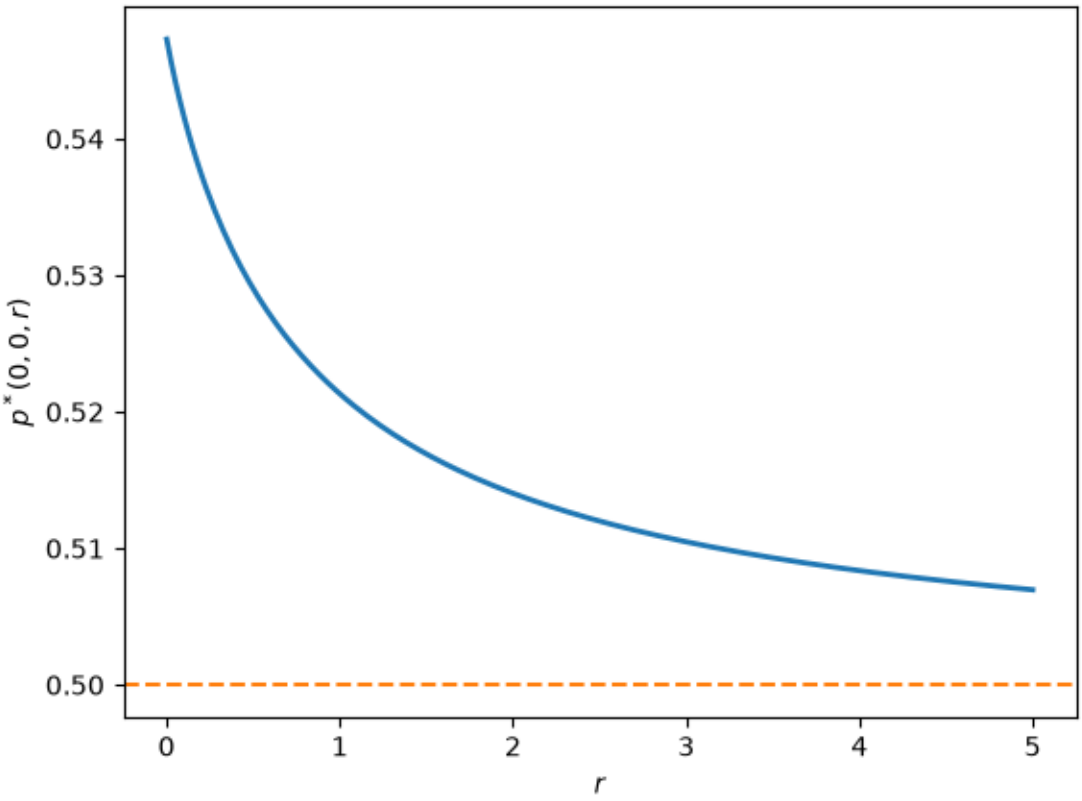


Fig. 122.1: Normalized price against interest rate

```
# Figure II: p*(t/2,t,0.05) as a function of t
r = 0.05
T = 60
p_mat = normalized_price_two_agents(priors[0], priors[1], r, T=T)
t_vals = np.arange(0, 54, 2)
s_vals = t_vals // 2
y = np.array([p_mat[s, t] for s, t in zip(s_vals, t_vals)])

fig, ax = plt.subplots()
ax.plot(t_vals, y, lw=2)
ax.set_xlabel(r'$t$')
ax.set_ylabel(r'$p^*(t/2,t,0.05)$')
ax.axhline(0.5, color='C1', linestyle='--')
plt.show()

p0 = p_mat[0,0]
μ0 = 0.5
print("Initial normalized premium at r=0.05 (%):",
      np.round(100 * (p0 / μ0 - 1.0), 2))
```

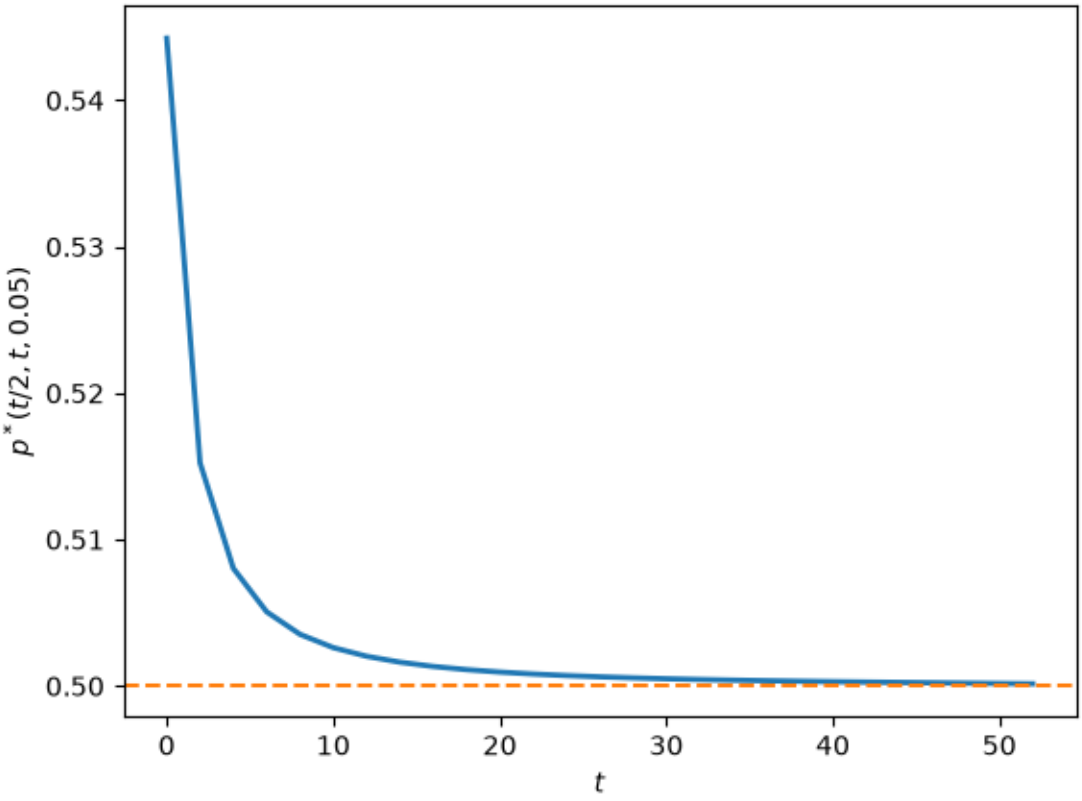


Fig. 122.2: Normalized price against time

```
Initial normalized premium at r=0.05 (%): 8.85
```

In the second figure, notice that:

- Along the symmetric path $s = t/2$, both traders' fundamental valuations equal 0.5 at every t, yet the price starts above 0.5 and declines toward 0.5 as learning reduces disagreement and the resale option loses value.

122.7.4 General N-trader extension

The same recursion extends to any finite set of Beta priors $\{(a_i, b_i)\}_{i=1}^N$ by taking a max over i each period.

```
def price_learning(priors, β=0.75, T=200):
    """
    N-trader version with heterogeneous Beta priors.
    """
    price_array = np.zeros((T+1, T+1))

    def perp_i(i, s, t):
        a, b = priors[i]
        return perpetuity_value(a, b, s, t, β)

    # Terminal condition
    for s in range(T+1):
        price_array[s, T] = max(
            perp_i(i, s, T) for i in range(len(priors)))

    # Backward induction
    for t in range(T-1, -1, -1):
        for s in range(t, -1, -1):
            conts = []
            for (a, b) in priors:
                μ = posterior_mean(a, b, s, t)
                conts.append(μ *
                (1.0 + price_array[s+1, t+1])
                     + (1.0 - μ) * price_array[s, t+1])
            price_array[s, t] = β * max(conts)

    return price_array

β = 0.75
priors = [(1,1), (0.5,0.5), (3,2)]
price_N = price_learning(priors, β=β, T=150)

# Compute valuations for each trader at (0,0)
μ_vals = [posterior_mean(a, b, 0, 0) for a, b in priors]
perp_vals = [(β / (1 - β)) * μ for μ in μ_vals]

print("Three-trader example at (s,t)=(0,0):")
print(f"Price at (0,0) = {np.round(price_N[0,0], 6)}")
print(f"\nTrader valuations:")
for i, (μ, perp) in enumerate(zip(μ_vals, perp_vals), 1):
    print(f"  Trader {i} = {np.round(perp, 6)}")
```

```
Three-trader example at (s,t)=(0,0):
Price at (0,0) = 1.937972

Trader valuations:
  Trader 1 = 1.5
  Trader 2 = 1.5
  Trader 3 = 1.8
```

Note that the asset price is above all traders' valuations.

Morris tells us that no rate dominance exists in this case.

Let's verify this using the code below

```
dominant = None
for i in range(len(priors)):
    is_dom = all(
        priors[i][0] >= priors[j][0] and priors[i][1] <= priors[j][1]
                for j in range(len(priors)) if i != j)
    if is_dom:
        dominant = i
        break

if dominant is not None:
    print(f"\nTrader {dominant+1} is the global optimist (rate-dominant)")
else:
    print(f"\nNo global optimist and speculative premium exists")
```

```
No global optimist and speculative premium exists
```

Indeed, there is no global optimist and a speculative premium exists.

122.8 Concluding remarks

Morris [1996] uses his model to interpret a "hot issue" anomaly described by [Miller, 1977] according to which opening market prices of initial public offerings seem higher than values prices that emerge later.

122.9 Exercise

Exercise 122.9.1

Morris [Morris, 1996] provides a sharp characterization of when speculative bubbles arise.

The key condition is that there is no *global optimist*.

In this exercise, you will verify this condition for the following sets of traders with Beta priors:

1. Trader 1: Beta$(2,1)$, Trader 2: Beta$(1,2)$
2. Trader 1: Beta$(1,1)$, Trader 2: Beta$(1/2,1/2)$
3. Trader 1: Beta$(3,1)$, Trader 2: Beta$(2,1)$, Trader 3: Beta$(1,2)$
4. Trader 1: Beta$(1,1)$, Trader 2: Beta$(1/2,1/2)$, Trader 3: Beta$(3/2,3/2)$

Solution

Here is one solution:

```
def check_rate_dominance(priors):
    """
    Check if any trader rate-dominates all others.
    """
    N = len(priors)

    for i in range(N):
        a_i, b_i = priors[i]
```

```
        is_dominant = True

        for j in range(N):
            if i == j:
                continue
            a_j, b_j = priors[j]

            # Check rate dominance condition
            if not (a_i >= a_j and b_i <= b_j):
                is_dominant = False
                break

        if is_dominant:
            return i

    return None

# Test cases
test_cases = [
    ([(2, 1), (1, 2)], "Global optimist exists"),
    ([(1, 1), (0.5, 0.5)], "Perpetual switching"),
    ([(3, 1), (2, 1), (1, 2)], "Three traders with dominant"),
    ([(1, 1), (0.5, 0.5), (1.5, 1.5)], "Three traders, no dominant")
]

for priors, description in test_cases:
    dominant = check_rate_dominance(priors)

    print(f"\n{description}")
    print(f"Priors: {priors}")
    print("==="*8)
    if dominant is not None:
        print(f"Trader {dominant+1} is the global optimist (rate-dominant)")
    else:
        print(f"No global optimist exists")
    print("==="*8 + "\n")
```

```
Global optimist exists
Priors: [(2, 1), (1, 2)]
================
Trader 1 is the global optimist (rate-dominant)
================


Perpetual switching
Priors: [(1, 1), (0.5, 0.5)]
================
No global optimist exists
================


Three traders with dominant
Priors: [(3, 1), (2, 1), (1, 2)]
================
Trader 1 is the global optimist (rate-dominant)
================
```

```
Three traders, no dominant
Priors: [(1, 1), (0.5, 0.5), (1.5, 1.5)]
================
No global optimist exists
================
```

CHAPTER

THREE

RATIONAL LEARNING AND RATIONAL EXPECTATIONS

Contents

- *Rational Learning and Rational Expectations*
  - *Overview*
  - *The economy*
  - *The rational learning equilibrium*
  - *Computing the equilibrium*
  - *Posterior concentration*
  - *Equilibrium trades and prices*
  - *Convergence of posterior assessments*
  - *Sharpening the convergence result*
  - *Obstacles to convergence*
  - *Learning within versus learning about a rational expectations equilibrium*
  - *Summary*
  - *Exercises*

123.1 Overview

This lecture explores an important question in economic theory: what can agents learn inside a rational expectations equilibrium?

This question is related to, but distinct from, the adaptive-learning question of whether a rational expectations equilibrium can be justified as a dynamic attractor.

The starting point is Bray and Kreps [1987], which gives a rigorous model of Bayesian learning inside a rational expectations equilibrium.

In a rational expectations equilibrium, agents use market prices to make inferences about other agents' private information.

Each agent knows the *statistical relationship* between prices and the underlying payoff-relevant variables and that relationship is *correct* given the equilibrium.

But this raises a question: where does that knowledge come from?

Their **rational learning** approach answers a narrower question.

Agents are uncertain about structural parameters, and the state space is enlarged to include, for each candidate value, the associated equilibrium price and allocation maps.

Agents then use Bayes' rule to update over those possibilities as prices and returns are observed.

This lecture develops that Bayesian formulation through an asset-market model.

The aim is to see what rational learning can explain, and where its limits appear, before turning to the computational illustration.

The discussion also connects to earlier work by Bray [1982], Bray and Savin [1986], and the rational expectations literature of Radner [1979], Grossman [1976], and Jordan [1982].

Let's start with the following imports

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import brentq
```

123.2 The economy

The model is a simple asset market built around the recurring information model of Grossman and Stiglitz [1980], repeated across infinitely many dates.

123.2.1 Agents and assets

Each date is economically disconnected from the others, so agents start each period afresh.

There are two types of agents and two assets:

- A **safe asset** with net return normalized to zero.
- A **risky asset** endowed one unit per agent and traded at date t at spot price p_t.

At each date $t = 0, 1, 2, \ldots$ the risky asset yields a gross return r_t at date $t + 1/2$.

An informed signal s_t satisfies

$$r_t = s_t + \epsilon_t, \qquad s_t \sim N(\mu_s, \tau^2), \qquad \epsilon_t \sim N(0, \sigma^2),$$

where $\{s_t\}$ and $\{\epsilon_t\}$ are IID normal sequences and are mutually independent.

Common knowledge of the prior moments (μ_s, τ^2) is what makes the price observation informative about θ^I, as we will see.

There are two representative agents:

- **Agent I (informed)** observes s_t before trading at date t.
- **Agent U (uninformed)** observes the equilibrium price p_t but not s_t before trading.

Both agents observe the previous return before current trading.

123.2.2 Preferences

Agent $n \in \{I, U\}$ has constant absolute risk tolerance θ^n.

If agent n holds x^n units of the risky asset and y^n units of the safe asset between dates t and $t + 1/2$, period utility is

$$-\exp\left[-\frac{x^n r_t + y^n}{\theta^n}\right].$$

Thus $1/\theta^n$ is the coefficient of absolute risk aversion.

Given the signal s_t, the informed agent's demand is

$$x_t^I = \frac{\theta^I}{\sigma^2}(s_t - p_t). \tag{123.1}$$

Because each of the two agents is endowed with one unit of the risky asset, market clearing is

$$x_t^I + x_t^U = 2.$$

123.2.3 Rational expectations equilibrium

If all agents knew s_t, agent n would demand

$$x_t^n = \frac{\theta^n}{\sigma^2}(s_t - p_t).$$

With N agents and total risky-asset supply N, market clearing gives the **full communication equilibrium price**

$$p_t = s_t - \frac{N\sigma^2}{\sum_{n=1}^N \theta^n}. \tag{123.2}$$

Thus if $\sum_n \theta^n$ is known, the price fully reveals s_t.

Following Radner [1979], this is called a full communication rational expectations equilibrium.

Suppose now that θ^I is unknown to agent U.

Following Bray and Kreps [1987], we treat this uncertainty by *expanding the state space*: we let the unknown parameter θ^I become a coordinate of the state, alongside the per-period shocks (s_t, ϵ_t).

Formally, the state space is $\Omega = \Theta \times \Phi^\infty$, where $\Theta = [a, b]$ supports the unknown θ^I and Φ supports each (s_t, ϵ_t).

Agent U knows σ^2 and θ^U, and starts with a prior density over θ^I on $[a, b]$.

This expansion is what turns the learning problem into Bayesian inference inside a single rational expectations equilibrium on Ω.

At a date when agent U has posterior density f over θ^I, his own trade reveals $x_t^I = 2 - x_t^U$ through market clearing.

Combining this inferred x_t^I with (123.1), each candidate θ^I implies

$$s_t = p_t + \frac{\sigma^2 x_t^I}{\theta^I}. \tag{123.3}$$

After trading, agent U observes r_t.

Bayes' rule then updates the posterior over θ^I using the normal density of the signal implied by (123.3) conditional on the realized return.

This is the main object learned in the two-agent example.

Even in this simple case, the equilibrium can be defined recursively but closed-form prices are unavailable.

123.3 The rational learning equilibrium

The model has two pieces that interact at each date.

The first is the within-period equilibrium given the uninformed agent's current posterior on θ^I.

The second is the Bayesian update of that posterior after the period closes.

123.3.1 Uninformed demand given beliefs

Suppose at date t agent U has posterior density f_t on θ^I, supported on $[a, b]$.

Suppose the equilibrium price is p and the equilibrium informed trade $X^I = 2 - x^U$ has been inferred from market clearing.

Conditional on θ, equation (123.3) pins down the signal as $s_t(\theta) = \sigma^2 X^I/\theta + p$.

Two sources of information about θ are therefore present at the start of date t trading: the carried-over posterior $f_t(\theta)$ and the Gaussian prior $\phi_s(\cdot; \mu_s, \tau^2)$ on s_t that values some implied signals as more plausible than others.

Bayes' rule combines them into the *intra-period* posterior

$$f_t^{(p,X^I)}(\theta) \propto f_t(\theta)\, \phi_s\left(\frac{\sigma^2 X^I}{\theta} + p\,;\, \mu_s, \tau^2\right), \tag{123.4}$$

which is the posterior on θ^I that the agent actually uses to forecast r_t before r_t is observed.

Conditional on a candidate value θ, the excess payoff on one unit of the risky asset is

$$r_t - p = \frac{\sigma^2 X^I}{\theta} + \epsilon_t, \qquad \epsilon_t \sim N(0, \sigma^2).$$

Because CARA preferences have no wealth effects, agent U's problem reduces to

$$\max_{x^U} \mathbb{E}[u^U(x^U, r_t, p)], \qquad u^U(x^U, r_t, p) = -\exp\left(-\frac{x^U(r_t - p)}{\theta^U}\right),$$

where the expectation integrates over $\theta^I \sim f_t^{(p,X^I)}$ and ϵ_t.

Substituting the conditional excess payoff and using the normal moment-generating formula gives

$$\mathbb{E}[u^U] = -\exp\left(\frac{(x^U)^2\sigma^2}{2(\theta^U)^2}\right) \int_a^b f_t(\theta)\, \phi_s\left(\tfrac{\sigma^2 X^I}{\theta} + p; \mu_s, \tau^2\right) \exp\left(-\tfrac{x^U \sigma^2 X^I}{\theta\, \theta^U}\right) d\theta,$$

up to a θ-independent constant absorbed in normalisation.

Define the tilted weight

$$w(\theta;\, p, X^I, x^U) = f_t(\theta)\, \phi_s\left(\tfrac{\sigma^2 X^I}{\theta} + p; \mu_s, \tau^2\right) \exp\left(-\tfrac{x^U \sigma^2 X^I}{\theta\, \theta^U}\right). \tag{123.5}$$

The first-order condition rearranges to

$$\frac{x^U}{\theta^U} = X^I\, \frac{\int_a^b \theta^{-1}\, w(\theta;\, p, X^I, x^U)\, d\theta}{\int_a^b w(\theta;\, p, X^I, x^U)\, d\theta}. \tag{123.6}$$

The right-hand side is X^I multiplied by a tilted expectation of $1/\theta^I$ under the weighting in (123.5).

Equation (123.6) implicitly defines $x^U(p, X^I; f_t)$, the uninformed agent's optimal demand at observed price p, conjectured informed trade X^I, and prior posterior f_t.

Dependence on p enters through the prior weight ϕ_s: at higher prices, candidate values of θ that imply s_t above the prior mean become less plausible, so the agent's demand schedule slopes downward in p as expected.

123.3.2 Market clearing

Equilibrium requires that the informed and uninformed demands sum to the total endowment.

Substituting (123.1) and the implicit function $x^U(p, X^I; f_t)$, the equilibrium (p_t, X_t^I) satisfies the two equations

$$X_t^I = \frac{\theta^I}{\sigma^2}(s_t - p_t), \qquad X_t^I + x^U(p_t, X_t^I; f_t) = 2. \tag{123.7}$$

Eliminating X_t^I between the two leaves a single root-finding problem for p_t.

Combining the two equations, the equilibrium price has the form

$$p_t = s_t - \frac{\sigma^2 X_t^I}{\theta^I}. \tag{123.8}$$

When f_t collapses to a point mass at the true θ^I, equation (123.6) simplifies to $x^U/\theta^U = X^I/\theta^I$, and market clearing gives the full-communication allocation

$$X_t^I = \frac{2\theta^I}{\theta^I + \theta^U}, \qquad x_t^U = \frac{2\theta^U}{\theta^I + \theta^U}. \tag{123.9}$$

This is the full-communication allocation we will use to check the simulation.

123.3.3 Bayesian update

After trading, agent U observes (p_t, x_t^U, r_t).

Market clearing gives $X_t^I = 2 - x_t^U$, and equation (123.3) assigns a candidate $s_t(\theta) = \sigma^2 X_t^I/\theta + p_t$ to each θ.

Since $s_t \sim N(\mu_s, \tau^2)$ independently of $\epsilon_t \sim N(0, \sigma^2)$, the conditional density of s_t given r_t is Gaussian:

$$g(s \mid r) = \phi\left(s;\, \frac{\sigma^2\mu_s + \tau^2 r}{\sigma^2 + \tau^2},\, \frac{\sigma^2\tau^2}{\sigma^2 + \tau^2}\right),$$

where $\phi(\cdot; m, v)$ denotes the Normal density with mean m and variance v.

Bayes' rule then produces the posterior update

$$f_{t+1}(\theta) \propto f_t(\theta)\; g\left(\frac{\sigma^2 X_t^I}{\theta} + p_t \,\middle|\, r_t\right). \tag{123.10}$$

This is the rule we simulate below.

123.4 Computing the equilibrium

We discretise the support $[a, b]$ of θ^I on a fine grid and represent f_t as a vector of density values.

The three computational primitives are:

- `uninformed_demand` solves the FOC in (123.6) for $x^U(p, X^I; f)$ by root-finding.
- `equilibrium_price` solves the market-clearing system (123.7) for p_t.
- `bayes_update` applies (123.10) and renormalises.

```
def uninformed_demand(p, XI, f, θ_grid, θ_U, σ2, μ_s, τ2):
    """
    Solve the FOC for x^U(p, X^I; f), the uninformed
    agent's optimal demand given observed price p, conjectured
    informed trade XI, and carried-over posterior density f.
    """
    with np.errstate(divide='ignore'):
        log_f = np.log(f)
    s_implied = σ2 * XI / θ_grid + p
    log_phi_s = -0.5 * (s_implied - μ_s)**2 / τ2  # prior weight on s_t

    def foc(xU):
        z = xU * σ2 * XI / (θ_grid * θ_U)
        log_w = log_f + log_phi_s - z
        M = log_w.max()
        w = np.exp(log_w - M)
        num = np.sum(w / θ_grid)
        den = np.sum(w)
        return xU / θ_U - XI * num / den

    return brentq(foc, -50.0, 50.0, xtol=1e-10)
```

```
def equilibrium_price(s_t, θ_I_true, f, θ_grid, θ_U, σ2, μ_s, τ2):
    """
    Solve the market-clearing system for the equilibrium
    price p_t given signal s_t, true informed risk tolerance
    θ_I_true, and posterior f.
    """
    def mc_residual(p):
        XI = θ_I_true * (s_t - p) / σ2
        xU = uninformed_demand(p, XI, f, θ_grid, θ_U, σ2, μ_s, τ2)
        return XI + xU - 2.0

    return brentq(mc_residual, s_t - 10.0, s_t, xtol=1e-8)
```

```
def bayes_update(f, θ_grid, p_t, xU_t, r_t, σ2, τ2, μ_s):
    """
    Bayesian update of the posterior on θ^I given the date-t
    observations (p_t, x^U_t, r_t).
    """
    XI = 2.0 - xU_t
    s_mean = (σ2 * μ_s + τ2 * r_t) / (σ2 + τ2)
    s_var = σ2 * τ2 / (σ2 + τ2)
    s_implied = σ2 * XI / θ_grid + p_t

    log_like = -0.5 * (s_implied - s_mean)**2 / s_var
    log_like -= log_like.max()  # log-shift for stability
    f_new = f * np.exp(log_like)
    dθ = θ_grid[1] - θ_grid[0]
    f_new /= np.sum(f_new) * dθ
    return f_new
```

The simulation loop chains (s_t, ϵ_t) shocks through these three functions.

```
def simulate(θ_I_true, θ_U, σ2, μ_s, τ2,
             a, b, n_grid, T, prior=None, seed=42):
    """
```

(continues on next page)

(continued from previous page)

```
    Simulate T periods of the Bray-Kreps rational-learning equilibrium.
    """
    rng = np.random.default_rng(seed)
    θ_grid = np.linspace(a, b, n_grid)
    dθ = θ_grid[1] - θ_grid[0]

    if prior is None:
        f = np.ones(n_grid) / (b - a)
    else:
        f = prior(θ_grid)
        f /= np.sum(f) * dθ

    s_seq = rng.normal(μ_s, np.sqrt(τ2), T)
    eps_seq = rng.normal(0.0, np.sqrt(σ2), T)

    XI_path = np.empty(T)
    p_path = np.empty(T)
    r_path = np.empty(T)
    post_mean = np.empty(T + 1)
    post_var = np.empty(T + 1)
    post_mean[0] = np.sum(θ_grid * f) * dθ
    post_var[0] = np.sum((θ_grid - post_mean[0])**2 * f) * dθ

    snap_times = {0, 5, 20, 50, 100, T}
    snapshots = {0: f.copy()}

    for t in range(T):
        p_t = equilibrium_price(
            s_seq[t], θ_I_true, f, θ_grid, θ_U, σ2, μ_s, τ2
        )
        XI = θ_I_true * (s_seq[t] - p_t) / σ2
        xU = 2.0 - XI
        r_t = s_seq[t] + eps_seq[t]
        f = bayes_update(f, θ_grid, p_t, xU, r_t, σ2, τ2, μ_s)

        XI_path[t] = XI
        p_path[t] = p_t
        r_path[t] = r_t
        post_mean[t + 1] = np.sum(θ_grid * f) * dθ
        post_var[t + 1] = np.sum(
            (θ_grid - post_mean[t + 1])**2 * f
        ) * dθ
        if (t + 1) in snap_times:
            snapshots[t + 1] = f.copy()

    return dict(
        θ_grid=θ_grid,
        snapshots=snapshots,
        XI_path=XI_path,
        p_path=p_path,
        r_path=r_path,
        post_mean=post_mean,
        post_var=post_var,
    )
```

123.5 Posterior concentration

We run the simulation with a uniform prior on $[0.5, 4]$ and true $\theta^I = 2$.

```
params = dict(
    θ_I_true=2.0,
    θ_U=1.0,
    σ2=1.0,
    μ_s=1.0,
    τ2=1.0,
    a=0.5,
    b=4.0,
    n_grid=300,
    T=200,
    seed=42,
)

res = simulate(**params)
```

The first picture shows snapshots of the posterior density at selected dates.

```
fig, ax = plt.subplots(figsize=(10, 5))
for t, ft in sorted(res['snapshots'].items()):
    ax.plot(res['θ_grid'], ft, lw=2, label=f't = {t}')
ax.axvline(params['θ_I_true'], color='black', ls='--', lw=1.5,
           label=r'$\theta^I_{\rm true}$')
ax.set_xlabel(r'$\theta^I$')
ax.set_ylabel('posterior density')
ax.legend()
plt.tight_layout()
plt.show()
```

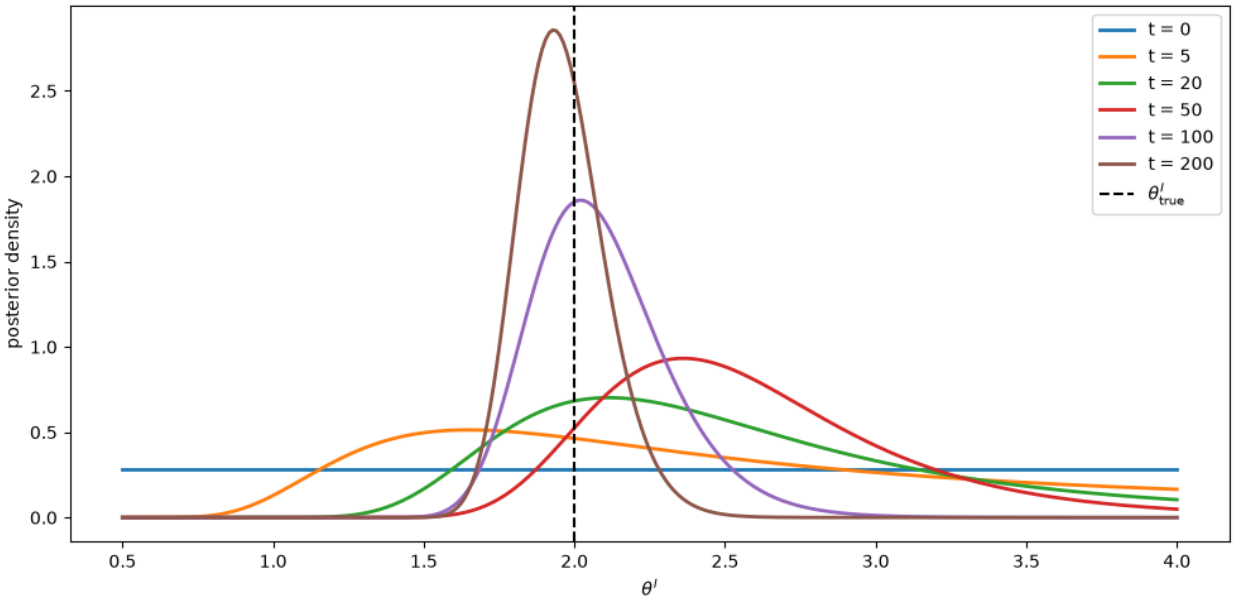


Fig. 123.1: posterior density over θ^I at selected dates

The posterior tightens around $\theta^I_{\text{true}} = 2$ as price and return data accumulate.

The next picture tracks the posterior mean and variance.

```
fig, axes = plt.subplots(1, 2, figsize=(13, 5))

ax = axes[0]
ax.plot(np.arange(params['T'] + 1), res['post_mean'], lw=2)
ax.axhline(params['θ_I_true'], color='red', ls='--', lw=2,
           label=r'$\theta^I_{\rm true}$')
ax.set_xlabel('$t$')
ax.set_ylabel(r'$E_t[\theta^I]$')
ax.legend()

ax = axes[1]
ax.plot(np.arange(params['T'] + 1), res['post_var'], lw=2)
ax.set_xlabel('$t$')
ax.set_ylabel(r'${\rm Var}_t[\theta^I]$')

plt.tight_layout()
plt.show()
```

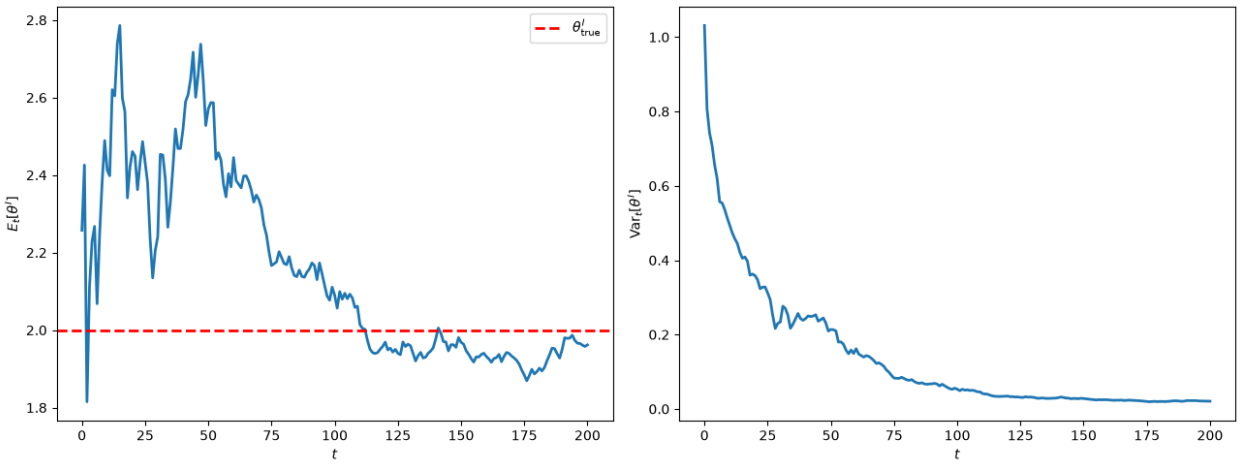


Fig. 123.2: posterior mean and variance over time

The posterior mean converges to the truth and the posterior variance vanishes.

This is the concrete manifestation of weak convergence of posteriors to a point mass at θ^I_{true}, which we describe in general terms below.

123.6 Equilibrium trades and prices

The equilibrium informed trade X_t^I depends on the current signal s_t, on θ_{true}^I, and on the carried-over posterior f_t, all through the market-clearing system (123.7).

As f_t tightens around θ_{true}^I, the average X_t^I approaches the full-communication allocation in (123.9).

```
XI_full = 2 * params['θ_I_true'] / (params['θ_I_true'] + params['θ_U'])
p_mean_full = params['μ_s'] - params['σ2'] * XI_full / params['θ_I_true']

fig, axes = plt.subplots(1, 2, figsize=(13, 5))

ax = axes[0]
ax.plot(res['XI_path'], lw=2, label='$X^I_t$ (learning)')
ax.axhline(XI_full, color='red', ls='--', lw=2,
           label='$X^I$ (full info)')
ax.set_xlabel('$t$')
ax.set_ylabel('$X^I_t$')
ax.legend()

ax = axes[1]
ax.plot(res['p_path'], lw=1.5, alpha=0.7, label='$p_t$ (learning)')
ax.axhline(p_mean_full, color='red', ls='--', lw=2,
           label='$E[p_t]$ (full info)')
ax.set_xlabel('$t$')
ax.set_ylabel('$p_t$')
ax.legend()

plt.tight_layout()
plt.show()
```

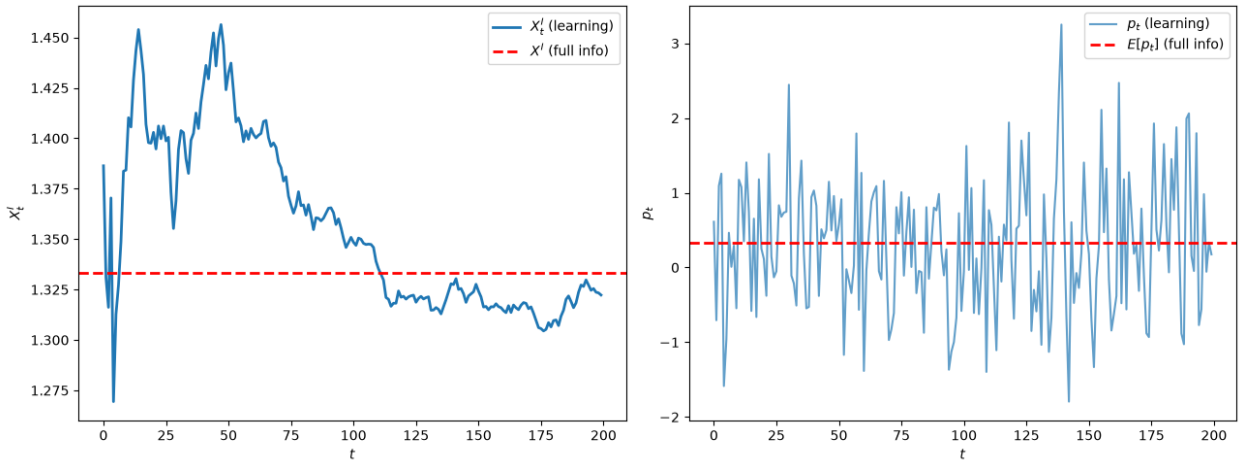


Fig. 123.3: equilibrium trade and prices over time

The left panel shows X_t^I approaching the full-information allocation as beliefs concentrate.

The right panel shows the price path, which fluctuates because p_t inherits the variation in s_t.

The simulation illustrates three features of this equilibrium.

The posterior density on θ^I concentrates around the true value, the posterior variance vanishes, and the equilibrium informed trade X_t^I converges to its full-information benchmark.

The next sections identify the theorems behind these outcomes and the assumptions they require.

We first state the two convergence results of Bray and Kreps [1987] for the abstract model, then specialize to the two-agent example to pin down the hypotheses that drive concentration on the true θ^I, and finally explain when those hypotheses can fail.

123.7 Convergence of posterior assessments

Let $(\Omega, \mathcal{F})$ be a measurable space carrying the equilibrium.

In the two-agent example, $\Omega = \Theta \times \Phi^\infty$, where $\Theta = [a, b] \times \{\theta^U\}$ collects the structural parameters, Φ collects the per-period shocks (s_t, ϵ_t), and $\mathcal{F}$ is the product Borel σ-field.

Agent n enters date 0 with a prior probability measure P^n on $(\Omega, \mathcal{F})$.

Let $G_t^n \subseteq \mathcal{F}$ denote the σ-field generated by agent n's private information through date t, and let

$$H_t^n(p) = G_t^n \vee \sigma(p_0, p_1, \dots, p_t)$$

be the σ-field that adds observation of equilibrium prices through date t.

The tail σ-field is

$$H_\infty^n(p) = \bigvee_{t \geq 0} H_t^n(p).$$

The first result, due to Bray and Kreps [1987], states that the conditional probability of any event converges almost surely.

Proposition 123.7.1

Fix an agent n and an event $A \in \mathcal{F}$.

The process $M_t = \mathbb{E}^n[\mathbf{1}_A \mid H_t^n(p)]$ is a P^n-bounded martingale with respect to $(H_t^n(p))_{t \geq 0}$, and

$$\lim_{t \to \infty} \mathbb{E}^n[\mathbf{1}_A \mid H_t^n(p)] = \mathbb{E}^n[\mathbf{1}_A \mid H_\infty^n(p)], \qquad P^n\text{-a.s.}$$

The proof is the bounded martingale convergence theorem, with $M_t \in [0, 1]$ supplying the uniform integrability needed for the limit identification.

Proposition 123.7.1 is convergence of posterior assessments, not convergence to "correct" beliefs.

Two qualifications are worth stating.

The "a.s." statement is relative to agent n's own prior P^n, so if two priors P^n and $P^{n'}$ are mutually singular, the conclusion need not hold simultaneously on a common P-positive event.

If the priors share a common null collection, simultaneous convergence holds outside a common null set.

The second result in Bray and Kreps [1987] sharpens convergence from individual events to the entire posterior measure on the parameter space, given a topological assumption on Θ.

Assumption 123.7.1

The parameter space Θ is a complete separable metric (Polish) space, and the Borel σ-field on Θ generated by its open sets makes $(\Theta, \mathcal{B}(\Theta))$ a Borel space.

In the two-agent example $\Theta = [a, b]$ trivially satisfies this assumption.

Under *Assumption 123.7.1* one can fix regular versions of the conditional probabilities: maps

$$P_t^n : \Omega \to \mathcal{P}(\Theta), \qquad \omega \mapsto P_t^n(\omega),$$

such that for each measurable $A \subseteq \Theta$, $\omega \mapsto P_t^n(\omega)(A)$ is a version of $\mathbb{E}^n[\mathbf{1}_{A \times \Phi^\infty} \mid H_t^n(p)](\omega)$, and $P_t^n(\omega) \in \mathcal{P}(\Theta)$ is a probability measure P^n-a.s.

The sharpened convergence result says these regular versions converge weakly almost surely.

Proposition 123.7.2

Under *Assumption 123.7.1*, the regular versions P_t^n converge weakly to a regular version P_∞^n, P^n-a.s.

Equivalently, for P^n-a.e. ω and every bounded continuous $f : \Theta \to \mathbb{R}$,

$$\int_\Theta f \, dP_t^n(\omega) \xrightarrow[t \to \infty]{} \int_\Theta f \, dP_\infty^n(\omega).$$

The proof in Bray and Kreps [1987] applies *Proposition 123.7.1* to a countable disjoint partition of Θ by $1/k$-balls, which exists because Θ is Polish, and then invokes the Portmanteau characterisation of weak convergence on bounded continuous functions.

Rational Bayesian learning therefore always produces a limiting posterior measure.

But *Proposition 123.7.2* alone does not pin down what that limit is, and additional structure is needed before the limit assesses the truth correctly.

123.8 Sharpening the convergence result

We now return to the two-agent example and identify hypotheses under which P_∞^U is a point mass at the true θ^I.

Write F_t for the CDF of agent U's posterior on θ^I at date t after observing $(r_{t-1}, p_{t-1}, x_{t-1}^U)$ and all earlier data.

Proposition 123.7.2 yields a random CDF F_∞ such that F_t converges weakly to F_∞, P^U-a.s.

Three ingredients sharpen this to concentration on the truth, corresponding to the three steps in Bray and Kreps [1987].

Assumption 123.8.1

The equilibrium uninformed demand $x^U(p, F)$ is continuous in F with respect to weak convergence, uniformly in p on a P^U-full-measure set of prices.

Assumption 123.8.2

For fixed θ^U and limiting posterior F_∞, the map

$$\theta^I \mapsto \mathcal{L}\{p_\infty(s; F_\infty, \theta^I, \theta^U)\}$$

is injective, where s is drawn from its marginal distribution.

A sufficient condition is strict stochastic monotonicity of this marginal price distribution in θ^I.

In the lecture's CARA-Normal setup, these assumptions are plausible under the maintained uniqueness and regularity of the implicit demand and price equations.

Continuity requires continuous dependence of the FOC root in (123.6) on F, not just pointwise continuity of the integrals.

Identification also uses the full equilibrium map: the equation $p_t = s_t - \sigma^2 X_t^I/\theta^I$ is informative, but X_t^I itself depends on θ^I, F_t, and the equilibrium fixed point.

The IID assumption on $\{s_t\}$, already part of the model, supplies the ergodicity used in step 2 below.

Under these three assumptions and the IID signal sequence, the limiting posterior in the two-agent example concentrates on the truth.

Proposition 123.8.1

Suppose $\theta^I_{\text{true}} \in [a, b]$ and the prior f_0 puts positive density in every neighbourhood of θ^I_{true}.

Under *Assumption 123.7.1*, *Assumption 123.8.1*, and *Assumption 123.8.2*, and given the IID signal sequence $\{s_t\}$, the limiting posterior on θ^I satisfies

$$F_\infty = \delta_{\theta^I_{\text{true}}} \qquad P^U\text{-a.s.}$$

The proof has three steps.

Step 1: price functional convergence.

Assumption 123.8.1 and the weak convergence $F_t \Rightarrow F_\infty$ from *Proposition 123.7.2* imply that equilibrium demands $x^U(p, F_t)$ converge to $x^U(p, F_\infty)$.

Combining with market clearing and the price equation (123.8) gives $p_t - p_\infty(s_t; F_\infty, \theta^I, \theta^U) \to 0$ on a P^U-full-measure set.

Step 2: the limit price distribution is observable.

Since the deviation $p_t - p_\infty(s_t; F_\infty, \theta^I, \theta^U) \to 0$ almost surely and $\{s_t\}$ is IID, the empirical distribution of observed prices has the same limit as the empirical distribution of the limiting price functional.

The latter equals the distribution of $p_\infty(s; F_\infty, \theta^I, \theta^U)$ for $s \sim N(\mu_s, \tau^2)$, and that limit is $H^U_\infty(p)$-measurable as a long-run frequency of an observable sequence.

Step 3: identification.

Assumption 123.8.2 makes the marginal distribution of p_∞ identify θ^I given (F_∞, θ^U).

Combined with step 2, this means θ^I is itself $H^U_\infty(p)$-measurable, so for any subinterval $[c, d] \subseteq [a, b]$ the limiting posterior satisfies $P^U_\infty(\theta^I \in [c, d]) = \mathbf{1}_{\{\theta^I_{\text{true}} \in [c,d]\}}$.

Combining steps 1, 2, and 3 yields $F_\infty = \delta_{\theta^I_{\text{true}}}$.

The numerical simulation above is consistent with this result.

The posterior density on θ^I collapses to a spike at $\theta^I_{\text{true}} = 2$, and the equilibrium informed trade X^I_t converges to the full-information value $2\theta^I_{\text{true}}/(\theta^I_{\text{true}} + \theta^U)$.

Hence, the path connecting *Proposition 123.7.1* (martingale convergence) to *Proposition 123.8.1* (concentration on the truth) depends on three model-specific ingredients: continuity, ergodicity, and identification.

123.9 Obstacles to convergence

It is natural to ask when these ingredients can fail, and what the consequences are for learning.

123.9.1 Obstacle 1: failure of continuity

If *Assumption 123.8.1* fails, step 1 of the proof breaks.

When the equilibrium price functional is discontinuous in F, small changes in beliefs can produce large changes in the information content of prices, and weak convergence of beliefs need not imply convergence of prices.

Bray and Kreps [1987] flag this as the most delicate step in their argument.

Continuity of $x^U(p, F)$ in F is plausible in this lecture's regular CARA-Normal case, but it also requires uniqueness of the FOC solution and continuous dependence of that solution on F.

Verifying the same property in richer market structures often requires non-trivial regularity arguments.

123.9.2 Obstacle 2: failure of identification

If *Assumption 123.8.2* fails, step 3 breaks even when steps 1 and 2 succeed.

Consider a variant with two informed agents and risk tolerances θ^{I1}, θ^{I2} both unknown to the uninformed agent.

With three agents each endowed with one unit of the risky asset, the full-communication formula (123.2) gives

$$p_t = s_t - \frac{3\sigma^2}{\theta^{I1} + \theta^{I2} + \theta^U},$$

which depends on $(\theta^{I1}, \theta^{I2})$ only through the sum $\theta^{I1} + \theta^{I2}$.

Proposition 123.7.2 still applies, but F_∞ is supported on the level set

$$\{(\theta_1, \theta_2) \in [a, b]^2 : \theta_1 + \theta_2 = \theta^{I1}_{\text{true}} + \theta^{I2}_{\text{true}}\},$$

not on the singleton $\{(\theta^{I1}_{\text{true}}, \theta^{I2}_{\text{true}})\}$.

Convergence occurs, but to a manifold of observationally equivalent parameter values rather than to the truth.

123.9.3 Obstacle 3: misspecification

A separate obstacle arises if the true pricing relation lies outside the agent's prior support.

Blume and Easley [1982] give a stylised version of this obstacle, and *Heterogeneous Beliefs and Financial Markets* develops the Blume-Easley heterogeneous-beliefs model in this lecture series.

Each agent entertains competing conditional likelihoods for other agents' information given his own information and the price.

An equilibrium can exist in which agents assign asymptotic probability one to an incorrect model that gives the observed events positive likelihood, while the true stable price relation receives zero posterior probability because that relation was absent from the prior model class.

In strict rational learning the prior already covers the truth: it is supported on Bayesian-consistent models in the expanded state space, and positive weight is placed on the true parameter.

Bayes' rule can only reweight that initial model class, so a pricing relation with zero prior probability stays at zero no matter what data accumulate.

123.10 Learning within versus learning about a rational expectations equilibrium

The framework above points to an important conceptual distinction in Bray and Kreps [1987].

123.10.1 The distinction

Learning *within* a rational expectations equilibrium is the topic of this lecture.

It is Bayesian inference inside a correctly specified model: *Assumption 123.7.1*, *Assumption 123.8.1*, and *Assumption 123.8.2* all hold, and the prior puts positive weight on the truth.

Agent U is uncertain about θ^I, but for every candidate value he already knows the equilibrium price and allocation maps.

The expanded-state-space formulation $\Omega = \Theta \times \Phi^\infty$ embeds a rational expectations equilibrium on the larger space, and inference reduces to conditional probability over Θ.

Learning *about* a rational expectations equilibrium is a fundamentally different exercise.

The agent does not begin with the equilibrium map embedded in his probability model.

Instead he must infer the price-state relation from data generated while his own beliefs and behavior co-evolve with the data.

123.10.2 The trade-off

The two notions sit on opposite sides of a precise trade-off.

A correctly-specified Bayesian learner enjoys the convergence guarantees in *Proposition 123.7.1* and *Proposition 123.7.2*, but only because the equilibrium has been built into the prior from date zero.

An adaptive learner who treats the price-state relation as something to be estimated can hope to discover it from data, but the estimator he uses cannot be derived from Bayes' rule applied to a correctly specified model.

Bayesian rational learning can update among equilibrium maps already included in the agent's prior, but it does not explain how agents come to obtain those maps in the first place.

The literature on learning *about* rational expectations equilibria, beginning with Bray [1982] and Bray and Savin [1986] and extended by Marcet and Sargent [1989], takes the second side of the trade-off and replaces Bayes' rule with **ordinary least squares** or related recursive estimators.

The companion lecture *Least Squares Learning in Self-Referential Models* develops this least-squares-learning framework in self-referential models and traces the resulting dynamics through the associated ordinary differential equation.

Those rules are computationally tractable and converge in important examples, but they are not the Bayesian update implied by the fully specified rational-learning equilibrium prior.

123.11 Summary

This lecture implemented the rational-learning equilibrium of Bray and Kreps [1987].

Posterior assessments converge by bounded martingale convergence (*Proposition 123.7.1*), and posterior measures converge weakly under a Polish-Borel assumption (*Proposition 123.7.2*).

Concentration on the truth additionally requires continuity (*Assumption 123.8.1*), ergodicity, and identification (*Assumption 123.8.2*); each obstacle above is a failure of one of these.

The simulation illustrates both conclusions: the posterior on θ^I collapses toward θ^I_{true} and the equilibrium informed trade approaches its full-information value.

Rational learning describes the limits of Bayesian inference *given* the equilibrium structure; adaptive learning, in *Least Squares Learning in Self-Referential Models*, describes how that structure can be learned in the first place.

123.12 Exercises

Exercise 123.12.1

Off-center prior

The baseline simulation uses a uniform prior on $\theta^I \in [0.5, 4]$.

1. Re-run the simulation with a prior whose mass sits *above* the true value, for example

```
prior = lambda θ: (θ - 0.5)**3 * (4 - θ)
```

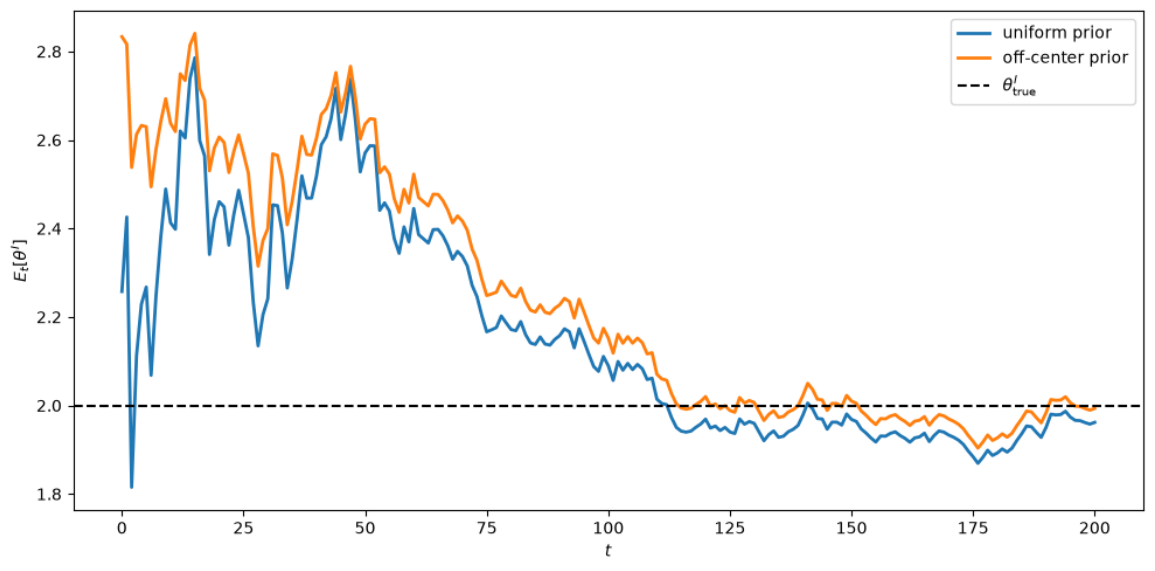
which peaks near $\theta = 3.1$.

2. Plot the posterior mean over time alongside the uniform-prior baseline.
3. Does the posterior eventually concentrate on θ^I_{true}, and how does the speed compare?

Solution

Here is one solution:

```
res_uniform = simulate(**params)

params_biased = dict(params)
params_biased['prior'] = lambda θ: (θ - 0.5)**3 * (4 - θ)
res_biased = simulate(**params_biased)

fig, ax = plt.subplots(figsize=(10, 5))
ax.plot(res_uniform['post_mean'], lw=2, label='uniform prior')
ax.plot(res_biased['post_mean'], lw=2, label='off-center prior')
ax.axhline(params['θ_I_true'], color='black', ls='--',
           label=r'$\theta^I_{\rm true}$')
ax.set_xlabel('$t$')
ax.set_ylabel(r'$E_t[\theta^I]$')
ax.legend()
plt.tight_layout()
plt.show()
```



The off-center prior starts the posterior mean well above $\theta^I_{\text{true}} = 2$, but Bayesian updating drives it down to the truth.

This is the rational-learning convergence result in action: any prior that puts positive density on θ^I_{true} eventually concentrates around it.

Exercise 123.12.2

Speed of learning across θ^I

Information from one period about θ^I comes through the implied signal

$$s_t(\theta) = \frac{\sigma^2 X_t^I}{\theta} + p_t.$$

The sensitivity $|\partial s_t / \partial \theta| = \sigma^2 X_t^I / \theta^2$ depends on the level of θ^I_{true} through X_t^I and θ^{-2}.

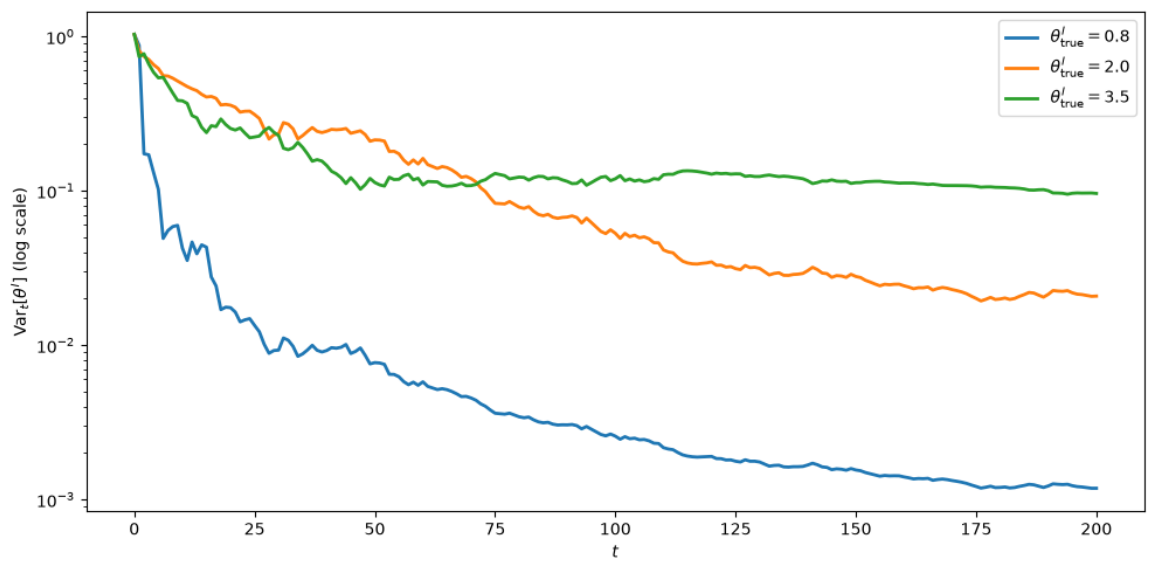
1. Run the simulation for $\theta^I_{\text{true}} \in \{0.8, 2.0, 3.5\}$, holding everything else at the baseline.
2. Plot the posterior variance on a log scale for each case.
3. Which value of θ^I_{true} yields the fastest concentration, and does the result match the sensitivity formula above?

Solution

Here is one solution:

```
fig, ax = plt.subplots(figsize=(10, 5))
for θ_val in [0.8, 2.0, 3.5]:
    params_θ = dict(params)
    params_θ['θ_I_true'] = θ_val
    res_θ = simulate(**params_θ)
    ax.semilogy(res_θ['post_var'], lw=2,
                label=fr'$\theta^I_{{\rm true}} = {θ_val}$')
ax.set_xlabel('$t$')
ax.set_ylabel(r'${\rm Var}_t[\theta^I]$ (log scale)')
ax.legend()
```

```
plt.tight_layout()
plt.show()
```



The smallest θ^I_{true} produces the steepest decline in posterior variance.

The reason is that the sensitivity $\sigma^2 X_t^I / \theta^2$ scales as θ^{-2} for fixed X_t^I, so the same noise level conveys much more information about θ^I when θ^I is small.

Exercise 123.12.3

Effect of return noise

Larger σ^2 widens the conditional density of s_t given r_t, which one might guess slows learning.

But σ^2 also scales the price intercept in (123.8), so price dispersion across candidate θ grows with σ^2.

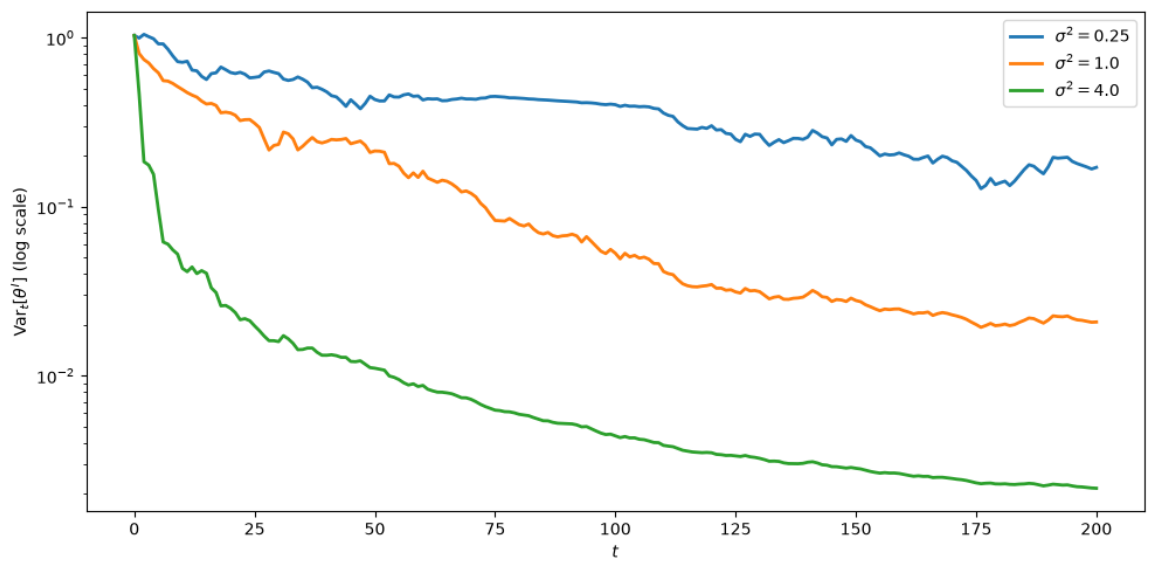
1. Run the simulation with $\sigma^2 \in \{0.25, 1.0, 4.0\}$, keeping $\tau^2 = 1$ fixed.
2. Plot the posterior variance on a log scale for each σ^2.
3. Explain which effect dominates in terms of the signal-to-noise ratio for inferring θ^I from the price.

Solution

Here is one solution:

```
fig, ax = plt.subplots(figsize=(10, 5))
for σ2_val in [0.25, 1.0, 4.0]:
    params_σ = dict(params)
    params_σ['σ2'] = σ2_val
    res_σ = simulate(**params_σ)
    ax.semilogy(res_σ['post_var'], lw=2,
                label=fr'$\sigma^2 = {σ2_val}$')
ax.set_xlabel('$t$')
ax.set_ylabel(r'${\rm Var}_t[\theta^I]$ (log scale)')
ax.legend()
plt.tight_layout()
```

```
plt.show()
```



The posterior variance falls *faster* for larger σ^2.

To see why, write $\nu = \sigma^2$ and hold the realized trade X_t^I fixed.

Two nearby values of θ imply signals separated by approximately

$$\left| \frac{\partial s_t(\theta)}{\partial \theta} \right| = \frac{\nu |X_t^I|}{\theta^2}.$$

The likelihood compares these implied signals using $g(s \mid r_t)$, whose conditional variance is

$$\mathrm{Var}(s_t \mid r_t) = \frac{\nu \tau^2}{\nu + \tau^2}.$$

Thus the local signal-to-noise ratio for distinguishing nearby θ values is proportional to

$$\frac{\nu |X_t^I| / \theta^2}{\sqrt{\nu \tau^2 / (\nu + \tau^2)}} = \frac{|X_t^I|}{\theta^2} \sqrt{\frac{\nu(\nu + \tau^2)}{\tau^2}},$$

This ratio rises with ν when τ^2 is fixed.

The price-revelation effect therefore dominates the extra return noise in this experiment.

CHAPTER

FOUR

LEAST SQUARES LEARNING IN SELF-REFERENTIAL MODELS

Contents

- *Least Squares Learning in Self-Referential Models*
  - *Overview*
  - *The self-referential structure*
  - *Why a differential equation governs the limit*
  - *The governing ODE*
  - *Five examples*
  - *Example 1: Ordinary linear stochastic difference equations*
  - *Example 2: Bray's cobweb model*
  - *Counterexample: Unstable Bray feedback*
  - *Example 3: Bray-Savin supply-shifter model*
  - *Example 4: Hyperinflation / asset prices*
  - *Example 5: Investment under uncertainty*
  - *Connection to rational learning*
  - *Summary*
  - *Exercises*

124.1 Overview

This lecture is a companion to *Rational Learning and Rational Expectations*, which presents the Bray-Kreps perspective on rational learning.

We examine the related but distinct question of whether *least squares* learning converges to a rational expectations equilibrium in self-referential models.

We present the framework of Marcet and Sargent [1989] for studying **least squares learning** in a class of **self-referential** linear stochastic models.

A self-referential model is one where the *actual* law of motion for the economy depends on the *perceived* law of motion of agents within it.

In a rational expectations equilibrium (REE) the two coincide: the perceived and actual laws of motion are the same.

But if agents start away from equilibrium and update their beliefs by running least squares regressions, will they converge to the REE?

Marcet and Sargent [1989] answer this question by using a powerful technique from systems-control engineering: the differential equation approach of Ljung [1977].

The key insight is that the stochastic difference equation of belief evolution can be approximated by a deterministic ordinary differential equation (ODE) in the limit.

Almost-sure convergence can be studied through the stability of that ODE.

Local stability of the small ODE is closely related to local convergence, while global almost-sure convergence also requires boundedness and domain-of-attraction conditions.

The framework unifies and extends earlier work by Bray [1982] and Bray and Savin [1986] and connects naturally to the distinction between learning *within* a rational expectations equilibrium (Bayesian updating inside a correctly specified model) and learning *about* one (adapting an OLS estimator whose data-generating process shifts with beliefs) discussed in Bray and Kreps [1987].

Let's begin with the imports we'll use throughout.

```
import numpy as np
import matplotlib.pyplot as plt
from numpy import linalg as la
from scipy.integrate import solve_ivp
from scipy.optimize import fsolve
```

Before simulating anything, we describe the object being learned.

124.2 The self-referential structure

124.2.1 Perceived and actual laws of motion

At each date t, agents hold a **perceived law of motion** summarized by a parameter matrix β_t.

They believe that the variable z_{1t} they care about evolves according to

$$z_{1t} = \beta_t z_{2,t-1} + \eta_t, \tag{124.1}$$

where z_{2t} is a vector of variables agents use to forecast $z_{1,t+1}$, and η_t is orthogonal to all past z_2's.

Because agents optimize (or behave) on the basis of this belief, their actions feed back into the economy.

To write the actual law of motion for the full state vector $z_t = (z_{1t}, z_{1t}^c)^\top$, partition $A(\beta_t)$ conformably with $(z_{2,t-1}^c, z_{2,t-1})$ as $A(\beta_t) = [A_c(\beta_t)\ A_2(\beta_t)]$.

$$z_t = \begin{bmatrix} 0 & T(\beta_t) \\ A_c(\beta_t) & A_2(\beta_t) \end{bmatrix} \begin{bmatrix} z_{2,t-1}^c \\ z_{2,t-1} \end{bmatrix} + \begin{bmatrix} V(\beta_t) \\ B(\beta_t) \end{bmatrix} u_t, \tag{124.2}$$

where u_t is IID white noise with covariance Σ.

The mapping T is the key object, as it maps the *perceived* coefficient β to the coefficient that *actually* governs z_{1t} in equilibrium.

A **rational expectations equilibrium** is a fixed point $\beta_f = T(\beta_f)$.

124.2.2 The learning scheme

Agents update β_t each period using **recursive least squares** (RLS).

Define R_t as a running estimate of the second-moment matrix $Ez_{2t}z_{2t}^\top$.

Updating equations are

$$\beta_t^\top = \beta_{t-1}^\top + \frac{\alpha_t}{t} R_{t-1}^{-1} \left\{ z_{2,t-2} z_{2,t-2}^\top [T(\beta_{t-1}) - \beta_{t-1}]^\top + z_{2,t-2} u_{t-1}^\top V(\beta_{t-1})^\top \right\}, \tag{124.3}$$

$$R_t = R_{t-1} + \frac{\alpha_t}{t} [z_{2,t-1} z_{2,t-1}^\top - R_{t-1}/\alpha_t], \tag{124.4}$$

where $\{\alpha_t\}$ is a positive, non-decreasing sequence with $\alpha_t \to 1$ as $t \to \infty$.

When $\alpha_t = 1$ for all t, equations (124.3)-(124.4) reduce to **ordinary least squares** updated recursively.

124.2.3 Lagged and contemporaneous data

The recursion above is written with *lagged information*, so the estimate β_t uses observations available through date $t-1$.

Marcet and Sargent [1989] also treat a *contemporaneous-data* version in which agents update using z_{1t} and $z_{2,t-1}$ at date t.

That timing creates simultaneous determination, because z_t depends on the same estimate β_t that is being updated from z_t.

An extra requirement is that the date-t system must have a unique solution (β_t, R_t, z_t) for each history.

If this uniqueness condition holds, and if the regularity and boundedness conditions used below also hold, convergence is still governed by the same full ODE (124.6) and small ODE (124.5).

Note

As Bray and Savin [1986] and Bray and Kreps [1987] emphasize, the RLS algorithm cannot be derived from Bayes' rule applied to a correctly specified model, because during the learning transition the data-generating process is non-stationary — beliefs shift the equilibrium, which shifts the data.

The algorithm is *"irrational"* in the sense that it acts as if the environment were stationary, when it is not.

Thus, any state variable that matters for the actual law of motion must appear among the regressors in the perceived law.

124.3 Why a differential equation governs the limit

The RLS recursion (124.3)-(124.4) is a *stochastic difference equation* with two key features.

First, the *step size* in front of each update is α_t/t, which shrinks to zero as t grows.

Second, the bracketed expression on the right-hand side of (124.3) is, at the long-run average values of the regressors and noise, just the discrepancy $T(\beta_{t-1}) - \beta_{t-1}$ that measures how far perceived beliefs are from the law of motion they actually generate.

Combine these two facts and a clear picture emerges.

The recursion makes only small adjustments to β_t each period and those adjustments average out, by the law of large numbers, to the deterministic drift $T(\beta) - \beta$.

As the gain α_t/t vanishes, the time-rescaled trajectory β_t looks more and more like a solution to the continuous-time differential equation

$$\frac{d\beta}{dt} = T(\beta) - \beta. \tag{124.5}$$

To compare the ODE with RLS after t observations, evaluate the ODE at $\tau_t = \sum_{s=1}^{t} \alpha_s/s$, the total step size accumulated by the learning algorithm.

When $\alpha_t = 1$, this accumulated step size is $\tau_t = \sum_{s=1}^{t} 1/s \approx \log t$.

This idea, due to Ljung [1977], is what lets Marcet and Sargent [1989] reduce the analysis of a noisy adaptive learning rule to the much easier study of a deterministic ODE.

The rest of this section makes the connection precise.

124.4 The governing ODE

124.4.1 Small and full ODEs

The *small ODE* (124.5) keeps only the drift in β, holding the variance estimator R at its long-run value.

Its rest points are precisely the rational expectations equilibria of the model: $\dot\beta = 0$ if and only if $T(\beta) = \beta$.

The *full ODE* tracks both β and R jointly:

$$\frac{d}{dt}\begin{bmatrix}\beta^\top \\ R\end{bmatrix} = \begin{bmatrix} R^{-1} M_{z_2}(\beta)\,[T(\beta) - \beta]^\top \\ M_{z_2}(\beta) - R \end{bmatrix}, \tag{124.6}$$

where $M_{z_2}(\beta) = \mathbb{E}[z_{2t} z_{2t}^\top]$ is computed at the stationary distribution of z_{2t} that prevails when agents believe the perceived law has constant parameter β.

The fixed point of (124.6) is (β_f, R_f) with $R_f = M_{z_2}(\beta_f)$ — the same β_f as the small ODE, paired with the second-moment matrix consistent with it.

124.4.2 Regularity assumptions

Marcet and Sargent [1989] distinguish two groups of assumptions, with quite different roles.

The first five are standard regularity conditions on the operator T, the shocks u_t, and the gain sequence; they are usually easy to check in applications.

Let $D_s \subset \mathbb{R}^{n_1 \times n_2}$ be the set on which $T(\beta)$, $A(\beta)$, $B(\beta)$, $V(\beta)$ are well defined and the eigenvalues of

$$\begin{bmatrix} 0 & T(\beta) \\ A_c(\beta) & A_2(\beta) \end{bmatrix}$$

are less than one in modulus, so that the state process induced by belief β is covariance-stationary.

Assumption 124.4.1 (Unique REE)

The operator T has a unique fixed point $\beta_f = T(\beta_f)$ with $\beta_f \in D_s$.

Assumption 124.4.2 (Smoothness of T, A, B, V)

T is twice differentiable and A, B, V each have one derivative in D_s.

Assumption 124.4.3 (Nonsingular limit covariance)

The covariance matrix $M_{z_2}(\beta_f)$ is nonsingular.

Assumption 124.4.4 (Gain sequence)

For all t, $\alpha_t > 0$; α_t is non-decreasing in t; $\alpha_t \to 1$ as $t \to \infty$; and $\limsup_{t\to\infty} t\,|\alpha_t - \alpha_{t-1}| < \infty$.

Assumption 124.4.5 (Shock moments)

The vector u_t is serially independent, and $E|u_{it}|^p < \infty$ for all $p > 1$ and all $i = 1, \ldots, m$.

These five say: there is a unique target, the model is smooth around it, the regression has a well-defined precision matrix in the limit, the gain shrinks at the right speed, and the shocks have enough moments for laws of large numbers to work.

The remaining two are *boundedness* conditions.

They are needed because Ljung's theorem requires the relevant sample path to return to a bounded region infinitely often.

In this model that includes the regressor process z_{2t} and the covariance estimate R_t.

Assumption 124.4.6 (Boundedness along a subsequence)

There exist a set Ω_0 with $\mathbb{P}(\Omega_0) = 1$, random variables $C_1(\omega)$ and $C_2(\omega)$, and a subsequence $\{t_k(\omega)\}$ such that

$$|z_{2t_k}(\omega)| < C_1(\omega) \quad \text{and} \quad |R_{t_k}(\omega)| < C_2(\omega)$$

for all $\omega \in \Omega_0$ and all $k = 1, 2, \ldots$.

Assumption 124.4.7 (Sample path stays in a workable domain)

Either the compact-state version or the projection-set version holds.

- *Compact-state version.*
  - $D_1 = D_2 = \mathbb{R}^{n_1 \times (n_2)^3}$.
  - There exists a compact $D' \subset D_s$ that contains $\beta_{t_k}(\omega)$ for all k and all $\omega \in \Omega_0$.
  - For every initial condition $(\beta(0), R(0))$ with $\beta(0) \in D'$ and $R(0)$ bounded as in *Assumption 124.4.6*, the β-component of the trajectory of (124.6) remains in a closed subset of D_s.
- *Projection-set version.*
  - D_2 is closed.

- D_1 is open and bounded.
- $\beta \in D_s$ for every $(\beta, R) \in D_1$.
- Trajectories of (124.6) with initial conditions in D_2 never leave a closed subset of D_1.

Assumption 124.4.6 is automatic when the regressors z_{2t} are *exogenous* and ergodic, but it can be delicate when z_{2t} contains endogenous variables.

Assumption 124.4.7 is satisfied in the compact-state version when the model has a natural bounded domain on which T is well-defined; otherwise the projection-set version corresponds to using the projection facility described below.

Let D_A denote the domain of attraction of the unique equilibrium (β_f, R_f) of (124.6).

124.4.3 Convergence of least squares

Proposition 124.4.1 (Least-squares convergence)

Assume *Assumption 124.4.1*, *Assumption 124.4.2*, *Assumption 124.4.3*, *Assumption 124.4.4*, *Assumption 124.4.5*, and *Assumption 124.4.6*.

If either

- the compact-state version of *Assumption 124.4.7* is satisfied and $D' \subset D_A$, or
- the projection-set version of *Assumption 124.4.7* is satisfied and $D_1 \subset D_A$,

then $\beta_t \to \beta_f$ almost surely as $t \to \infty$.

Proposition 124.4.1 reduces the problem of almost-sure convergence of recursive least squares to checking the *stability* of the ODE (124.6) at (β_f, R_f) and guaranteeing the boundedness of the sample path.

124.4.4 Stability governs convergence

Let $\mathcal{M}$ be the Jacobian matrix of $T(\beta) - \beta$ evaluated at the REE β_f:

$$\mathcal{M} = \frac{d \operatorname{col}(T(\beta) - \beta)}{d \operatorname{col}(\beta)^\top}\bigg|_{\beta=\beta_f}, \tag{124.7}$$

and let $h(\beta, R)$ denote the Jacobian of the right-hand side of the full ODE (124.6) after stacking $(\beta^\top, R)$ into a column vector.

Proposition 124.4.2 (Jacobian reduction)

The matrix $h(\beta_f, R_f)$ has $(n_2)^2$ repeated eigenvalues equal to -1; its remaining $n_1 \times n_2$ eigenvalues coincide with the eigenvalues of $\mathcal{M}$.

Consequently:

- If all eigenvalues of $\mathcal{M}$ have *strictly negative real parts*, both (124.5) and (124.6) are locally stable.
- Under the boundedness and domain-of-attraction conditions in *Proposition 124.4.1*, this local stability can be used to obtain almost-sure convergence of β_t to β_f.

- If any eigenvalue of $\mathcal{M}$ has *positive real part*, then the next proposition shows that convergence is impossible.

Proposition 124.4.3 (Necessity)

Assume *Assumption 124.4.1*, *Assumption 124.4.2*, *Assumption 124.4.3*, *Assumption 124.4.4*, and *Assumption 124.4.5*.

1. Let $\hat{\beta} \neq \beta_f$ and suppose $M_{z_2}(\hat{\beta})$ is positive definite and $\hat{\beta} \in \text{int}(D_2)$.

   Then $\mathbb{P}(\beta_t \to \hat{\beta}) = 0$.

2. If $h(\beta_f, R_f)$ has at least one eigenvalue with strictly positive real part, then $\mathbb{P}(\beta_t \to \beta_f) = 0$.

The first part says that recursive least squares cannot settle on a non-equilibrium belief in the interior of the admissible domain.

If learning has a limit, that limit must be a fixed point of T.

The second part says that local instability is not just slow convergence.

If the REE is unstable for the learning ODE, convergence to that REE has probability zero.

The stability condition $\text{Re}(\lambda_i(\mathcal{M})) < 0$ for all i is what is called **E-stability** (See Evans [1985]): the REE is a stable rest point of the expectational dynamics $\dot{\beta} = T(\beta) - \beta$.

E-stability plays the role here that prior support plays in Bayesian learning: it tells us when the learning process can find its way to the equilibrium.

124.4.5 The projection facility

E-stability is the local condition that makes the REE attractive for the learning ODE.

By itself, it is not an almost-sure convergence theorem.

The stochastic recursion must also remain in a bounded region where the ODE approximation is valid.

Conversely, a strictly positive eigenvalue rules out convergence to the REE, while boundary cases with zero real parts are not covered by the theorem.

Ljung's theorem requires the learning process and the relevant regressors to return to bounded regions with probability one.

The **projection facility** is a procedure that enforces boundedness of the estimated coefficients and covariance matrix.

What the projection facility does

The full learning algorithm augments the plain RLS update with a constraint set $D_1 \supset D_2$ in (β, R)-space.

After each unconstrained RLS step produces a candidate $(\tilde{\beta}_t, \tilde{R}_t)$, the projection facility enforces:

$$(\beta_t, R_t) = \begin{cases} (\tilde{\beta}_t,\ \tilde{R}_t) & \text{if } (\tilde{\beta}_t, \tilde{R}_t) \in D_1, \\ \text{some point in } D_2 & \text{otherwise.} \end{cases} \tag{124.8}$$

The set D_1 is chosen so that the model remains well-defined (e.g., R_t stays positive definite; β_t stays in a region where $T(\beta)$ is well-defined and the state process is covariance-stationary).

The set $D_2 \subset D_1$ is a slightly smaller "safe" region to which the algorithm is retracted whenever it threatens to leave D_1.

Without the projection facility, the stochastic path (β_t, R_t) might temporarily wander to regions where the system (124.2) is non-stationary (e.g., an explosive VAR).

The projection facility keeps (β_t, R_t) inside a chosen admissible region.

When regressors include endogenous variables, the separate boundedness condition *Assumption 124.4.6* still has to control z_{2t} along a subsequence.

Formally, Marcet and Sargent [1989] require that trajectories of the full ODE (124.6) originating in D_1 do not leave D_1.

This is often checked by verifying that the full vector field points toward the interior at the boundary ∂D_1.

Corollary 124.4.1

Assume *Assumption 124.4.1*, *Assumption 124.4.2*, *Assumption 124.4.3*, *Assumption 124.4.4*, *Assumption 124.4.5*, and *Assumption 124.4.6*.

Suppose also that $(\beta, R) \in D_1$ implies $\beta \in D_s$, and that D_1 is open and bounded with $D_1 \subset D_A$.

Then for some subsequence $\{t_k(\omega)\}$,

$$\mathbb{P}(\beta_t \to \beta_f) + \mathbb{P}(\beta_{t_k} \to (D_1 \setminus D_2)) = 1.$$

The second event has probability zero whenever the ODE trajectories point inward at ∂D_1, in which case *Corollary 124.4.1* reduces to $\beta_t \to \beta_f$ almost surely.

The exogenous-regressor case

When the regressors z_{2t} are *exogenous*, so that $E(z_{2t} z_{2t}^\top) = M_{z_2}(\beta) \equiv M$ does not depend on β, the verification of the boundary condition becomes routine.

Let $H(\beta)$ be the mean-value slope of T, i.e. the matrix satisfying

$$\operatorname{col}\{T(\beta) - T(\beta_f)\} = H(\beta) \operatorname{col}(\beta - \beta_f). \tag{124.9}$$

Corollary 124.4.2

Consider the algorithm defined by (124.3)-(124.4) with projection rule (124.8).

Choose $0 < K' < K < \infty$ and assume

1. *Assumption 124.4.1*, *Assumption 124.4.2*, *Assumption 124.4.3*, *Assumption 124.4.4*, and *Assumption 124.4.5* hold;
2. z_{2t} is exogenous, so that $E(z_{2t} z_{2t}^\top) = M_{z_2}(\beta) \equiv M$;
3. the small ODE $\dot\beta = T(\beta) - \beta$ is globally stable in $\mathbb{R}^{n_1 \times n_2}$;
4. there exists $\bar\varepsilon > 0$ such that for all $0 < \varepsilon \le \bar\varepsilon$ and all β with $|\beta - \beta_f| = K$, every eigenvalue of $[I(1-\varepsilon) + \varepsilon H(\beta)]^\top [I(1-\varepsilon) + \varepsilon H(\beta)]$ has modulus less than α^2 with $\alpha < 1$.

Take $D_1 = \{(\beta, R) : |\beta - \beta_f| < K\}$ and $D_2 = \{(\beta, R) : |\beta - \beta_f| \le K'\}$, and let the projection rule retract β_t to any value with $|\beta - \beta_f| \le K'$, while leaving $R_t = \tilde R_t$.

Then $\beta_t \to \beta_f$ almost surely.

For the scalar linear examples below, condition (4) reduces to checking that the feedback slope of T is not too strong.

For the first four examples below, T is linear and M_{z_2} is independent of β, so *Corollary 124.4.2* uses D_1 and D_2 only to justify the projection argument and leaves the examples with the scalar ODE stability checks.

D_1 and D_2 are therefore not economic primitives in the examples below.

They are proof devices for the projected learning algorithm: D_1 is the outer admissible region where the model and ODE remain well defined, and D_2 is the smaller region used to reset the estimates if an update tries to leave D_1.

Once the corollary says suitable sets can be chosen, the examples no longer need to display them and can focus on T, β_f, and the local stability slope.

124.5 Five examples

We now work through five examples from Marcet and Sargent [1989].

Each substantive learning example follows the same template.

1. Write down the economic equations that determine the equilibrium.
2. Substitute the perceived law of motion for any expectational variables to obtain the actual law of motion as a function of β.
3. Read off the operator T and the REE $\beta_f = T(\beta_f)$.
4. Check E-stability by computing $\mathcal{M} = dT/d\beta - I$ at β_f.
5. Plot the RLS paths, the small ODE evaluated at cumulative-gain time, and the drift $T(\beta) - \beta$.

In each case, β_t is the law of motion agents currently use when making decisions, while $T(\beta_t)$ is the law of motion their decisions actually produce.

Least-squares learning asks whether repeated observations move β_t toward a fixed point of this map.

The first example is the control case with no self-referential feedback.

For the remaining scalar examples, the simulation figure appears immediately after the model description.

124.6 Example 1: Ordinary linear stochastic difference equations

The first example has no self-referential component.

Let the actual law of motion be fixed, with $T(\beta) = \Gamma$ for a stable matrix Γ and with $V(\beta) = I$.

Economically, this is the control case.

Agents are estimating a stable, exogenous data-generating process.

Their beliefs do not feed back into prices, quantities, or future data.

The REE is $\beta_f = \Gamma$.

Since T is constant, $H(\beta) = 0$ in (124.9).

The small-ODE Jacobian is $\mathcal{M} = -I$, so the small ODE is globally stable.

Corollary 124.4.2 then implies that recursive least squares converges almost surely to the true law of motion.

In this no-feedback benchmark, the Marcet-Sargent result reduces to the standard consistency of least squares for stable linear stochastic difference equations.

124.7 Example 2: Bray's cobweb model

Bray [1982] studied a simple cobweb economy in which the equilibrium price satisfies

$$p_t = a + b\beta_t + \tilde{u}_t, \tag{124.10}$$

where β_t is agents' OLS estimate of the price (their point forecast of p_t), and $\tilde{u}_t$ is IID noise with mean zero and variance σ_u^2.

Here the forecast itself is a state variable for the economy.

If producers expect a high price, their current supply decisions alter the market-clearing price.

The parameter b measures the strength of this expectational feedback.

When $b < 1$, the actual price response is weaker than the forecast error, so least squares has a force pushing beliefs back toward the fixed point.

When $b > 1$, the feedback is too strong and the same learning rule moves beliefs away from the REE.

The mapping T is simply $T(\beta) = a + b\beta$.

The REE is

$$\beta_f = \frac{a}{1-b}, \quad b \neq 1. \tag{124.11}$$

The small ODE is

$$\dot{\beta} = T(\beta) - \beta = a + b\beta - \beta = a - (1-b)\beta, \tag{124.12}$$

which has the unique fixed point $\beta_f = a/(1-b)$.

Its Jacobian is $\mathcal{M} = b - 1 < 0$ when $b < 1$.

Under the boundedness and domain conditions above, RLS then converges almost surely.

When $b > 1$, $\mathcal{M} > 0$ and convergence fails.

The code below sets $a = 1$ and $b = 0.6$.

The rational expectations price forecast is then $\beta_f = 2.5$.

The three panels show the noisy RLS paths, the small-ODE approximation, and the learning drift $T(\beta) - \beta$.

The next cell sets up the simulator and plotting helpers used in the Bray figures.

```
def simulate_rls_scalar(T_map, σ_u, β0, T_periods=500, N_paths=100,
                        a_seq=None, seed=0):
    """
    Simulate the RLS recursion for z1_t = T(β_t) + u_t, z2_t = 1.
    Returns an (N_paths, T_periods) array of belief paths.
    """
    rng = np.random.default_rng(seed)
    if a_seq is None:
        a_seq = np.ones(T_periods)

    β_paths = np.empty((N_paths, T_periods))

    for i in range(N_paths):
        β = β0
        R = 1.0
```

(continues on next page)

(continued from previous page)

```
        for t in range(T_periods):
            α_t = a_seq[t]
            z2 = 1.0
            u_t = rng.normal(0, σ_u)
            z1 = T_map(β) * z2 + u_t

            R_old = max(R, 1e-8)
            β = β + (α_t / (t + 1)) / R_old * z2 * (z1 - β * z2)
            R = R + (α_t / (t + 1)) * (z2**2 - R / α_t)
            R = max(R, 1e-8)

            β_paths[i, t] = β

    return β_paths


def solve_ode_calendar(f_ode, β0, T_periods, a_seq=None):
    """
    Solve dβ/dτ = f_ode(β) and evaluate it at RLS cumulative-gain time.
    """
    if a_seq is None:
        a_seq = np.ones(T_periods)

    periods = np.arange(T_periods + 1)
    gains = a_seq / np.arange(1, T_periods + 1)
    ode_time = np.concatenate(([0.0], np.cumsum(gains)))
    sol = solve_ivp(
        lambda τ, y: [f_ode(y[0])],
        (0.0, ode_time[-1]),
        [β0],
        t_eval=ode_time,
        method='RK45',
        max_step=0.05
    )
    return periods, sol.y[0]


def plot_scalar_drift(ax, β_grid, drift, β_f, color):
    """Plot the one-dimensional learning drift T(β) - β."""
    ax.plot(β_grid, drift, color=color, lw=2)
    ax.axhline(0, color='black', lw=1.5)
    ax.axvline(β_f, color='red', ls='--', lw=2,
               label=f'$\\beta_f = {β_f:.2f}$')
    ax.fill_between(β_grid, drift, 0, where=(drift > 0),
                    color=color, alpha=0.12)
    ax.fill_between(β_grid, drift, 0, where=(drift < 0),
                    color=color, alpha=0.12)

    for β_arrow in np.linspace(β_grid[20], β_grid[-20], 7):
        dβ = np.interp(β_arrow, β_grid, drift)
        if abs(dβ) > 1e-10:
            ax.annotate(
                '', xy=(β_arrow + 0.25 * np.sign(dβ), 0),
                xytext=(β_arrow, 0),
                arrowprops=dict(arrowstyle='->', color=color, lw=1.8)
            )
```

(continues on next page)

(continued from previous page)

```
    ax.set_xlabel('$\\beta$')
    ax.set_ylabel('$T(\\beta) - \\beta$')
    ax.legend(fontsize=9)


T_sim = 400
N_sim = 80
```

```
a_bray, b_bray, σ_bray = 1.0, 0.6, 1.0


def T_bray(β):
    return a_bray + b_bray * β


β_f_bray = a_bray / (1 - b_bray)

β0_bray = 0.0

β_paths_bray = simulate_rls_scalar(
    T_bray, σ_bray, β0_bray,
    T_periods=T_sim, N_paths=N_sim
)


def ode_bray(β):
    return a_bray + b_bray * β - β


t_ode, sol_low = solve_ode_calendar(ode_bray, 0.0, T_sim)
_, sol_high = solve_ode_calendar(ode_bray, 4.5, T_sim)

β_grid_bray = np.linspace(-0.5, 5.0, 300)
drift_bray = np.array([ode_bray(b) for b in β_grid_bray])

fig, axes = plt.subplots(1, 3, figsize=(15, 4.8))

ax = axes[0]
for i in range(min(30, N_sim)):
    ax.plot(β_paths_bray[i], color='steelblue', alpha=0.25, lw=2)
ax.plot(np.mean(β_paths_bray, axis=0), color='navy', lw=2,
        label='cross-path average')
ax.axhline(β_f_bray, color='red', ls='--', lw=2,
           label=f'$\\beta_f = {β_f_bray:.2f}$')
ax.set_xlabel('$t$')
ax.set_ylabel('$\\beta_t$')
ax.legend()

ax = axes[1]
ax.plot(
    t_ode, sol_low, color='steelblue', lw=2,
    label='ODE at $\\tau_t$, $\\beta_0=0$'
)
ax.plot(
    t_ode, sol_high, color='darkorange', lw=2,
    label='ODE at $\\tau_t$, $\\beta_0=4.5$'
```

(continues on next page)

(continued from previous page)

```
)
ax.axhline(β_f_bray, color='red', ls='--', lw=2,
           label=f'$\\beta_f = {β_f_bray:.2f}$')
ax.set_xlabel('$t$')
ax.set_ylabel('$\\beta(\\tau_t)$')
ax.legend()

plot_scalar_drift(
    axes[2], β_grid_bray, drift_bray, β_f_bray, 'steelblue'
)

plt.tight_layout()
plt.show()
print(f"REE: β_f = a/(1-b) = {β_f_bray:.4f}")
print(f"Jacobian M = b - 1 = {b_bray - 1:.4f}  (< 0: E-stable)")
```

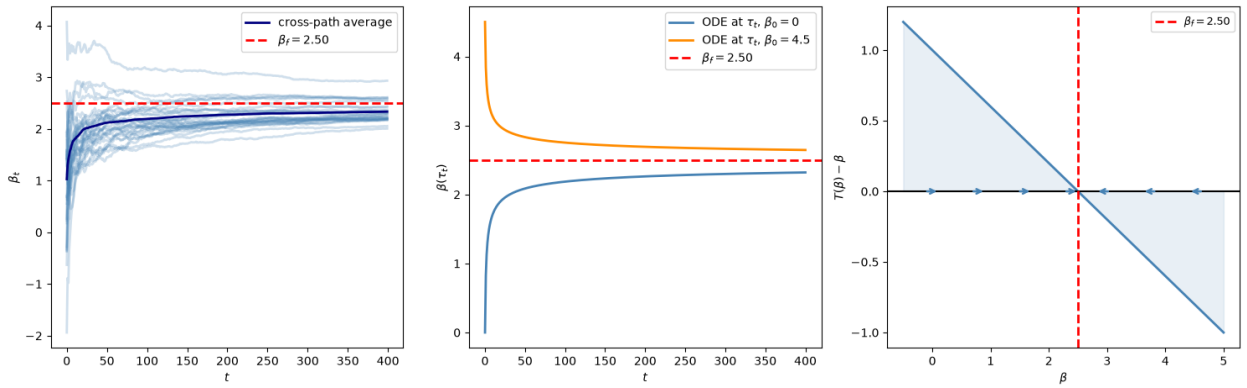


Fig. 124.1: Bray learning dynamics

```
REE: β_f = a/(1-b) = 2.5000
Jacobian M = b - 1 = -0.4000  (< 0: E-stable)
```

The stochastic paths are noisy because prices contain $\tilde{u}_t$.

The average path and the ODE paths remove most of that noise and reveal the mean learning direction.

124.8 Counterexample: Unstable Bray feedback

To see what happens when E-stability is violated, repeat Bray's model with $b > 1$.

It represents an economy where expectations affect actual prices more than one-for-one.

If agents start with a forecast that is too high, the market outcome induced by that forecast is higher still, so least squares raises the forecast.

The same logic works in reverse below the REE.

The REE exists, but it is not learnable by this adaptive rule.

The same three diagnostic panels now show how beliefs are pushed away from the REE rather than toward it, and the drift panel shows that the learning force points away from the fixed point on both sides.

```
b_unstable = 1.4


def T_unstable(β):
    return a_bray + b_unstable * β


β_f_unstable = a_bray / (1 - b_unstable)

β_paths_unstable = simulate_rls_scalar(
    T_unstable, σ_bray, β0=0.0,
    T_periods=200, N_paths=50
)


def ode_unstable(β):
    return T_unstable(β) - β


β_grid = np.linspace(-5, 5, 300)
drift = np.array([ode_unstable(b) for b in β_grid])

t_ode_un, sol_un_low = solve_ode_calendar(ode_unstable, -4.0, 200)
_, sol_un_high = solve_ode_calendar(ode_unstable, 0.0, 200)

fig, axes = plt.subplots(1, 3, figsize=(15, 4.8))

ax = axes[0]
for i in range(min(30, 50)):
    ax.plot(β_paths_unstable[i], color='crimson', alpha=0.3, lw=2)
ax.axhline(β_f_unstable, color='black', ls='--', lw=2,
           label=f'$\\beta_f = {β_f_unstable:.2f}$ (unstable)')
ax.set_xlabel('$t$')
ax.set_ylabel('$\\beta_t$')
ax.legend()

ax = axes[1]
ax.plot(
    t_ode_un, sol_un_low, color='crimson', lw=2,
    label='ODE at $\\tau_t$, $\\beta_0=-4$'
)
ax.plot(
    t_ode_un, sol_un_high, color='steelblue', lw=2,
    label='ODE at $\\tau_t$, $\\beta_0=0$'
)
ax.axhline(β_f_unstable, color='black', ls='--', lw=2,
           label=f'$\\beta_f = {β_f_unstable:.2f}$')
ax.set_xlabel('$t$')
ax.set_ylabel('$\\beta(\\tau_t)$')
ax.legend()

plot_scalar_drift(
    axes[2], β_grid, drift, β_f_unstable, 'crimson'
)

plt.tight_layout()
plt.show()
print(f"Jacobian M = b - 1 = {b_unstable - 1:.2f}  (> 0: NOT E-stable)")
```

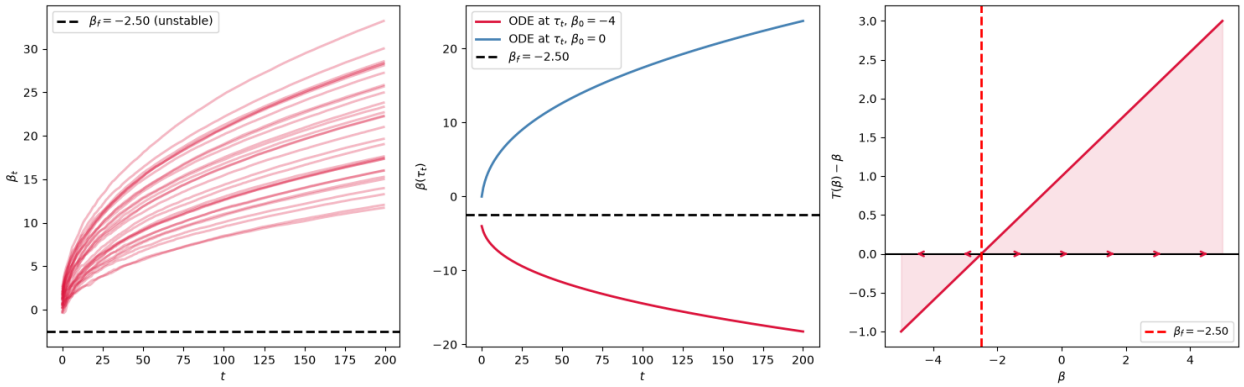


Fig. 124.2: Unstable Bray dynamics

```
Jacobian M = b - 1 = 0.40  (> 0: NOT E-stable)
```

124.9 Example 3: Bray-Savin supply-shifter model

Bray and Savin [1986] studied a model where

$$p_t = x_t^\top (m + a\beta_{t-1}) + \tilde{u}_t, \quad p_t^e = x_t^\top \beta_{t-1}, \tag{124.13}$$

with x_t an exogenous supply-shifter, a a scalar feedback parameter, and agents running an OLS regression of p on x.

This is the same learning problem with one extra economic ingredient: expectations are conditional on an observed shifter.

Agents do not merely forecast the average price.

They estimate how price responds to x_t.

The actual coefficient on x_t is $m + a\beta$, so the perceived slope changes the true slope generated by the market.

If $a < 1$, a mistaken slope feeds back into the actual price equation with attenuation, and the least-squares estimate is pulled toward the REE slope.

The mapping is $T(\beta) = m + a\beta$ (scalar case), giving

$$\dot{\beta} = (a - 1)\beta + m, \quad \beta_f = \frac{m}{1 - a}, \tag{124.14}$$

with Jacobian $\mathcal{M} = a - 1 < 0$ iff $a < 1$.

The simulation below draws the exogenous shifter x_t and lets agents update the slope in the regression of p_t on x_t.

It uses $a = 0.7$.

The drift panel shows that mistaken slopes are pushed back toward the REE slope rather than amplified.

The next cell adapts the RLS simulator to the supply-shifter regression.

```
def simulate_bray_savin(m, a, σ_x, σ_u, β0, T_periods=500,
                        N_paths=100, seed=0):
    """
    Simulate RLS for p_t = x_t (m + a β_t) + u_t.
    Agents regress p_t on x_t.
```

(continues on next page)

(continued from previous page)

```
    """
    rng = np.random.default_rng(seed)
    β_paths = np.empty((N_paths, T_periods))

    for i in range(N_paths):
        β = β0
        R = 1.0

        for t in range(T_periods):
            x_t = rng.normal(0.0, σ_x)
            u_t = rng.normal(0.0, σ_u)
            p_t = x_t * (m + a * β) + u_t
            step = 1.0 / (t + 1)

            R_old = max(R, 1e-8)
            β = β + step / R_old * x_t * (p_t - β * x_t)
            R = R + step * (x_t**2 - R)
            R = max(R, 1e-8)
            β_paths[i, t] = β

    return β_paths
```

```
m_bs, a_bs = 0.5, 0.7
σ_x_bs, σ_u_bs = 1.0, 1.0


def T_bs(β):
    return m_bs + a_bs * β


β_f_bs = m_bs / (1 - a_bs)

β_paths_bs = simulate_bray_savin(
    m_bs, a_bs, σ_x_bs, σ_u_bs, 0.0,
    T_periods=T_sim, N_paths=N_sim
)


def ode_bs(β):
    return T_bs(β) - β


t_ode_bs, sol_bs_low = solve_ode_calendar(ode_bs, 0.0, T_sim)
_, sol_bs_high = solve_ode_calendar(ode_bs, 4.0, T_sim)

β_grid_bs = np.linspace(-0.5, 4.0, 300)
drift_bs = np.array([ode_bs(b) for b in β_grid_bs])

fig, axes = plt.subplots(1, 3, figsize=(15, 4.8))

ax = axes[0]
for i in range(min(30, N_sim)):
    ax.plot(β_paths_bs[i], color='darkorange', alpha=0.25, lw=2)
ax.plot(np.mean(β_paths_bs, axis=0), color='saddlebrown', lw=2,
        label='cross-path average')
ax.axhline(β_f_bs, color='red', ls='--', lw=2,
```

(continues on next page)

(continued from previous page)

```
            label=f'$\\beta_f = {β_f_bs:.2f}$')
ax.set_xlabel('$t$')
ax.set_ylabel('$\\beta_t$')
ax.legend()

ax = axes[1]
ax.plot(
    t_ode_bs, sol_bs_low, color='darkorange', lw=2,
    label='ODE at $\\tau_t$, $\\beta_0=0$'
)
ax.plot(
    t_ode_bs, sol_bs_high, color='steelblue', lw=2,
    label='ODE at $\\tau_t$, $\\beta_0=4$'
)
ax.axhline(β_f_bs, color='red', ls='--', lw=2,
            label=f'$\\beta_f = {β_f_bs:.2f}$')
ax.set_xlabel('$t$')
ax.set_ylabel('$\\beta(\\tau_t)$')
ax.legend()

plot_scalar_drift(
    axes[2], β_grid_bs, drift_bs, β_f_bs, 'darkorange'
)

plt.tight_layout()
plt.show()
print(f"REE: β_f = m/(1-a) = {β_f_bs:.4f}")
print(f"Jacobian M = a - 1 = {a_bs - 1:.4f}  (< 0: E-stable)")
```

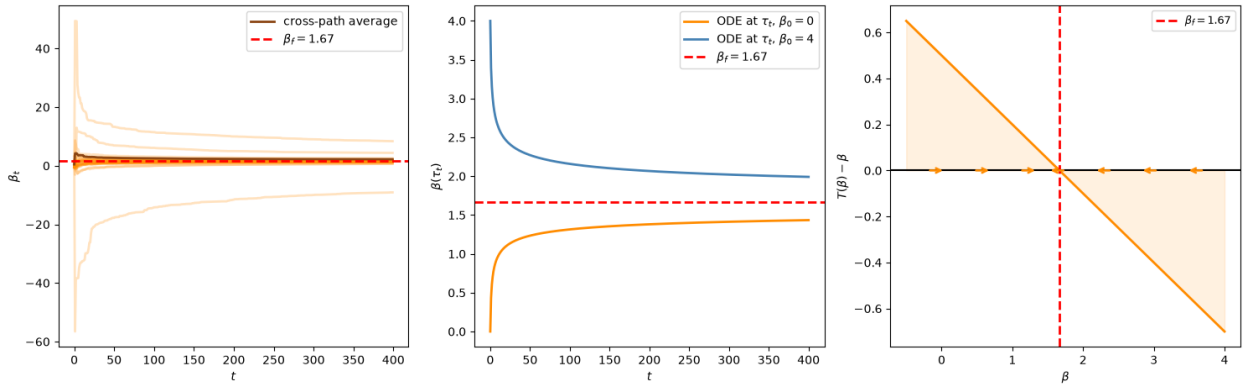


Fig. 124.3: Bray-Savin learning dynamics

```
REE: β_f = m/(1-a) = 1.6667
Jacobian M = a - 1 = -0.3000  (< 0: E-stable)
```

The interpretation is parallel to Bray's cobweb model.

The difference is that agents are now learning how prices co-move with an exogenous variable, not just learning an unconditional mean.

124.10 Example 4: Hyperinflation / asset prices

Consider the present-value asset pricing model

$$y_t = \lambda \mathbb{E}_t y_{t+1} + x_t, \quad x_t = \rho x_{t-1} + \varepsilon_t, \tag{124.15}$$

where $|\lambda| < 1$, $|\rho| < 1$, and agents perceive $y_t = \beta_t x_{t-1} + v_t$.

Under the perceived law, $\mathbb{E}_t y_{t+1} = \beta_t x_t$, so (124.15) becomes

$$y_t = (\lambda\beta_t + 1)x_t = (\lambda\beta_t + 1)\rho x_{t-1} + (\lambda\beta_t + 1)\varepsilon_t.$$

The coefficient on x_{t-1} in the actual law of motion is therefore $(\lambda\beta_t + 1)\rho$, which gives the operator T below.

If y_t is a price level, the equation resembles a Cagan-style hyperinflation model.

If y_t is an asset price and x_t is a dividend or fundamental, it is a present-value relation.

In both cases, beliefs about the future value of y enter the current value of y.

The product $\lambda\rho$ measures the strength of the forward-looking feedback: λ discounts the future and ρ measures how persistent the fundamental is.

When $|\lambda\rho| < 1$, a mistaken perceived coefficient does not amplify without bound.

The mapping is $T(\beta) = (\lambda\beta + 1)\rho$ and the REE is

$$\beta_f = \frac{\rho}{1 - \lambda\rho}. \tag{124.16}$$

The small ODE is

$$\dot{\beta} = (\lambda\rho - 1)\beta + \rho, \tag{124.17}$$

with Jacobian $\mathcal{M} = \lambda\rho - 1 < 0$ for $|\lambda\rho| < 1$, so the small ODE is stable.

The convergence theorem then applies under the exogenous-regressor conditions above.

The simulation uses $\lambda = 0.8$ and $\rho = 0.9$, so $\lambda\rho = 0.72$.

This means the feedback from expectations is strong enough to matter but still less than one.

The ODE panel below uses τ itself on the horizontal axis, where τ is the total step size accumulated by least squares.

The code simulates the autoregressive fundamental x_t and updates an OLS regression of y_t on x_{t-1}.

The next cell simulates the autoregressive fundamental and the matching RLS update.

It keeps the covariance estimate R_t bounded away from zero, mirroring the projection idea above and avoiding misleading early-sample explosions.

```
def simulate_present_value_rls(λ, ρ, σ_ε, β0, T_periods=500,
                               N_paths=100, seed=0):
    """
    Simulate RLS for y_t = (λ β_t + 1) x_t,
    x_t = ρ x_{t-1} + ε_t, with regression of y_t on x_{t-1}.
    """
    rng = np.random.default_rng(seed)
    β_paths = np.empty((N_paths, T_periods))
    σ_x = σ_ε / np.sqrt(1 - ρ**2)
    R_min = 0.05 * σ_x**2
```

(continues on next page)

(continued from previous page)

```
    for i in range(N_paths):
        β = β0
        R = σ_x**2
        x_lag = rng.normal(0.0, σ_x)

        for t in range(T_periods):
            ε_t = rng.normal(0.0, σ_ε)
            x_t = ρ * x_lag + ε_t
            y_t = (λ * β + 1.0) * x_t
            step = 1.0 / (t + 1)

            R_old = max(R, R_min)
            β = β + step / R_old * x_lag * (y_t - β * x_lag)
            R = R + step * (x_lag**2 - R)
            R = max(R, R_min)
            β_paths[i, t] = β
            x_lag = x_t

    return β_paths
```

```
λ, ρ_pv, σ_pv = 0.8, 0.9, 1.0
T_pv_sim = T_sim
N_pv_sim = N_sim


def T_pv(β):
    return (λ * β + 1) * ρ_pv


β_f_pv = ρ_pv / (1 - λ * ρ_pv)

β_paths_pv = simulate_present_value_rls(
    λ, ρ_pv, σ_pv, 0.0,
    T_periods=T_pv_sim, N_paths=N_pv_sim
)


def ode_pv(β):
    return T_pv(β) - β


τ_grid_pv = np.linspace(0.0, 30.0, 300)
sol_pv_low = solve_ivp(
    lambda τ, y: [ode_pv(y[0])],
    (0.0, τ_grid_pv[-1]),
    [0.0],
    t_eval=τ_grid_pv,
    max_step=0.05
).y[0]
sol_pv_high = solve_ivp(
    lambda τ, y: [ode_pv(y[0])],
    (0.0, τ_grid_pv[-1]),
    [10.0],
    t_eval=τ_grid_pv,
    max_step=0.05
).y[0]
```

(continues on next page)

(continued from previous page)

```
β_grid_pv = np.linspace(-1.0, 7.0, 300)
drift_pv = np.array([ode_pv(b) for b in β_grid_pv])

fig, axes = plt.subplots(1, 3, figsize=(15, 4.8))

ax = axes[0]
for i in range(min(30, N_pv_sim)):
    ax.plot(β_paths_pv[i], color='seagreen', alpha=0.25, lw=2)
ax.plot(np.mean(β_paths_pv, axis=0), color='darkgreen', lw=2,
        label='cross-path average')
ax.axhline(β_f_pv, color='red', ls='--', lw=2,
           label=f'$\\beta_f = {β_f_pv:.2f}$')
ax.set_xlabel('$t$')
ax.set_ylabel('$\\beta_t$')
ax.legend()

ax = axes[1]
ax.plot(
    τ_grid_pv, sol_pv_low, color='seagreen', lw=2,
    label='ODE, $\\beta_0=0$'
)
ax.plot(
    τ_grid_pv, sol_pv_high, color='steelblue', lw=2,
    label='ODE, $\\beta_0=10$'
)
ax.axhline(β_f_pv, color='red', ls='--', lw=2,
           label=f'$\\beta_f = {β_f_pv:.2f}$')
ax.set_xlabel('Accumulated step size $\\tau$')
ax.set_ylabel('$\\beta(\\tau)$')
ax.legend()

plot_scalar_drift(
    axes[2], β_grid_pv, drift_pv, β_f_pv, 'seagreen'
)

plt.tight_layout()
plt.show()
print(f"REE: β_f = ρ/(1 - lambda*ρ) = {β_f_pv:.4f}")
print(f"Jacobian M = lambda*ρ - 1 = {λ*ρ_pv - 1:.4f}  (< 0: E-stable)")
```

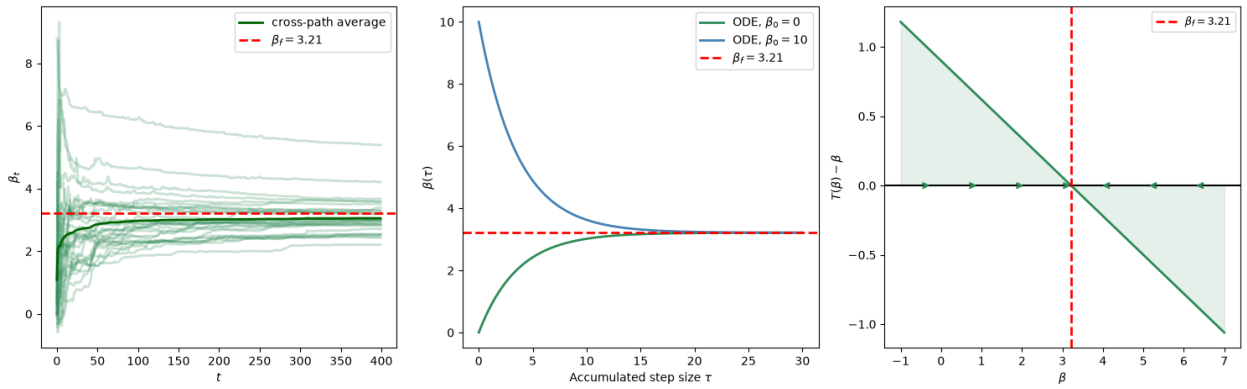


Fig. 124.4: Present-value learning dynamics

```
REE: β_f = ρ/(1 - lambda*ρ) = 3.2143
Jacobian M = lambda*ρ - 1 = -0.2800  (< 0: E-stable)
```

The REE coefficient is larger than ρ because current prices capitalize expected future fundamentals.

Learning converges because each revision in β_t changes expectations, but the discounted persistence term $\lambda\rho$ prevents revisions from feeding back explosively.

124.11 Example 5: Investment under uncertainty

Let's now consider a version of the Lucas-Prescott investment model where agents learn about the aggregate capital stock K_t by regressing on (K_{t-1}, w_{t-1}) where w_t is an exogenous cost shock.

Firms choose investment using a perceived law of motion for aggregate capital.

Those investment choices then determine the actual law of motion for aggregate capital through market clearing.

Learning is therefore about a state variable that agents themselves help create through their investment decisions.

The perceived law of motion is

$$K_t = \beta_1 K_{t-1} + \beta_2 w_{t-1} + \eta_t,$$

while the actual law (from firms' optimal investment decisions and market clearing) is

$$K_t = T_1(\beta) K_{t-1} + T_2(\beta) w_{t-1} + V(\beta) u_t, \tag{124.18}$$

where the nonlinear mappings T_1, T_2 come from solving the firms' linear quadratic control problems.

The two coefficients have different economic roles.

β_1 describes persistence in aggregate capital, while β_2 describes how the exogenous shock w_t affects next period's capital.

Because the optimal investment rule depends on the whole perceived law of motion, the map T is nonlinear.

This is why the larger ODE is harder to analyze here than in the scalar examples with exogenous regressors.

The small ODE decomposes as:

$$\dot{\beta}_1 = T_1(\beta_1) - \beta_1, \quad \dot{\beta}_2 = T_2(\beta_1, \beta_2) - \beta_2, \tag{124.19}$$

and E-stability can be verified analytically for $|\beta_1| < b^{-1/2}$ (where b is the discount factor).

The phase portrait below plots the vector field $T(\beta) - \beta$ over (β_1, β_2) space.

Each arrow shows how the perceived law of motion would be revised if the economy were currently operating under that belief.

The plotted trajectories are deterministic ODE paths, not noisy least-squares sample paths.

```
def T_invest(β, b=0.95, d=1.0, f=1.0, A1=1.0, N=1.0, ρ_w=0.5):
    """Mapping T for the investment model with β = [β1, β2]."""
    b1, b2 = β
    denom1 = 1 - b1 * b + (1 / d) * f**2 * A1 * N
    T1 = (1 - b1 * b) / denom1
    numer2 = 1 - b1 * b + f**2 * A1 * b2 * b * ρ_w
    T2 = (-N / (d * (1 - ρ_w * b))) * (numer2 / denom1) * ρ_w
    return np.array([T1, T2])
```

(continues on next page)

(continued from previous page)

```
def ode_invest(t, β, **kwargs):
    Tb = T_invest(β, **kwargs)
    return Tb - β


params = dict(b=0.95, d=1.0, f=1.0, A1=1.0, N=1.0, ρ_w=0.5)
β1_bound = params["b"] ** (-0.5)
β_f_inv = fsolve(lambda b: T_invest(b, **params) - b, [0.5, 0.1])
print(f"REE: β_f = {β_f_inv}")

eps = 1e-6
J = np.zeros((2, 2))
for j in range(2):
    e = np.zeros(2)
    e[j] = eps
    J[:, j] = (
        T_invest(β_f_inv + e, **params)
        - T_invest(β_f_inv - e, **params)
    ) / (2 * eps)
M = J - np.eye(2)
eigs = la.eigvals(M)
print(f"Jacobian M eigenvalues: {eigs}")
print(f"E-stable: {np.all(eigs.real < 0)}")

fig, ax = plt.subplots(figsize=(8, 6))

b1_grid = np.linspace(-0.1, min(1.0, β1_bound - 1e-3), 20)
b2_grid = np.linspace(-0.8, 0.5, 20)
B1, B2 = np.meshgrid(b1_grid, b2_grid)
U = np.zeros_like(B1)
V_field = np.zeros_like(B2)
for i in range(B1.shape[0]):
    for j in range(B1.shape[1]):
        β_ij = np.array([B1[i, j], B2[i, j]])
        drift = T_invest(β_ij, **params) - β_ij
        U[i, j] = drift[0]
        V_field[i, j] = drift[1]

speed = np.sqrt(U**2 + V_field**2)
speed[speed == 0] = 1e-8
ax.streamplot(b1_grid, b2_grid, U, V_field, color=speed,
              cmap='Blues', density=1.3, linewidth=1)

starts = [(0.1, 0.0), (0.9, 0.4), (0.98, -0.6), (0.3, -0.7)]
colors_traj = ['red', 'darkorange', 'green', 'purple']
for (b10, b20), col in zip(starts, colors_traj):
    sol = solve_ivp(lambda t, β: ode_invest(t, β, **params),
                    [0, 30], [b10, b20],
                    t_eval=np.linspace(0, 30, 300),
                    method='RK45')
    ax.plot(sol.y[0], sol.y[1], color=col, lw=2)
    ax.plot(b10, b20, 'o', color=col, ms=7)

ax.plot(*β_f_inv, 'k*', ms=14, label='REE $\\beta_f$')
ax.set_xlabel('$\\beta_1$', fontsize=12)
ax.set_ylabel('$\\beta_2$', fontsize=12)
```

(continues on next page)

(continued from previous page)

```
ax.legend()
plt.tight_layout()
plt.show()
```

```
REE: β_f = [ 0.38728475 -0.28879438]
Jacobian M eigenvalues: [-1.27718071 -1.35664898]
E-stable: True
```

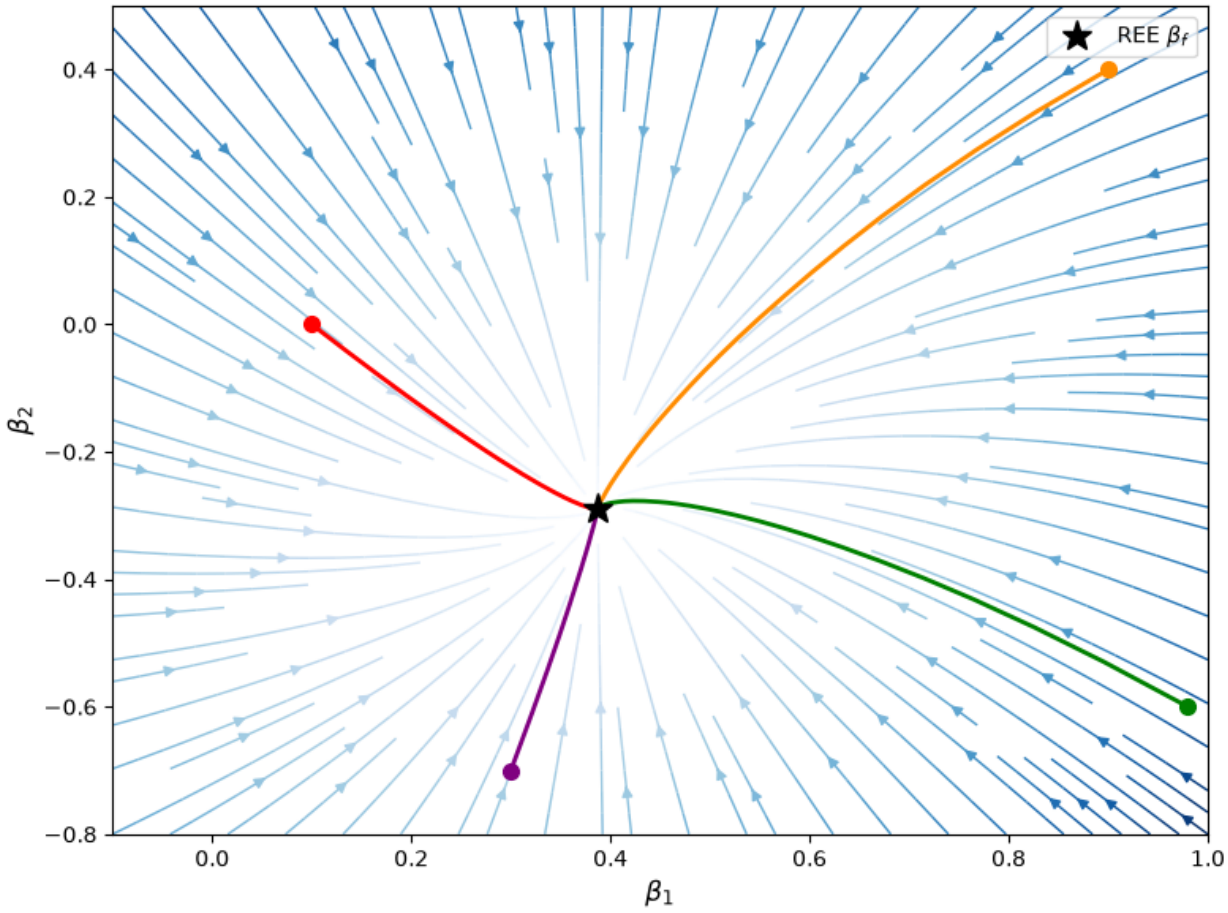


Fig. 124.5: Investment phase portrait

The star marks the REE.

The paths converge because the capital-persistence coefficient β_1 is stabilized first, and then the shock coefficient β_2 adjusts given the limiting value of β_1.

This recursive structure is why the small ODE is tractable even though the full least-squares system has endogenous regressors.

124.12 Connection to rational learning

The framework of Marcet and Sargent [1989] belongs to the program of learning *about* a rational expectations equilibrium, as distinct from learning *within* one — a distinction emphasized by Bray and Kreps [1987].

Learning *within* an REE (the subject of the companion lecture *Rational Learning and Rational Expectations*) refers to Bayesian inference inside a correctly specified model.

In that setting the data-generating process is stationary from the agent's perspective, and Bayes' rule is fully rationalized.

Learning *about* an REE involves an agent who does not know the equilibrium price function.

Because the agent's beliefs shift the equilibrium price, the data the agent uses to update beliefs are themselves generated by a non-stationary process.

As Marcet and Sargent [1989] put it,

> The models do not incorporate fully optimal behavior or rational expectations, because agents operate under the continually falsified assumption that the law of motion is time invariant and known for sure.

Yet it is a compelling rule.

Convergence holds despite the misspecification whenever E-stability combines with the required boundedness and domain-of-attraction conditions.

Unlike Bayesian learning, it does not require strong prior assumptions about the statistical structure of the economy.

124.13 Summary

This lecture studied least-squares learning in the self-referential models of Marcet and Sargent [1989].

The central object is the map T from a perceived law of motion to the actual law of motion generated when agents act on that perception.

A rational expectations equilibrium is a fixed point of this map.

Recursive least squares converges to that fixed point when the associated ODE $\dot{\beta} = T(\beta) - \beta$ is locally stable and the learning process remains in a region where the model is well defined.

It complements rational learning, studied in *Rational Learning and Rational Expectations*, where Bayesian agents learn *within* an equilibrium structure that is already specified.

124.14 Exercises

Exercise 124.14.1

Consider the scalar model with $T(\beta) = a + b\beta$.

1. Derive a formula for the unique REE β_f in terms of a and b.
2. Show that the small ODE $\dot{\beta} = T(\beta) - \beta$ is globally stable if and only if $b < 1$.
3. Simulate $N = 200$ paths of length $T = 500$ for $a = 1$ and each of $b \in \{0.3, 0.7, 0.9, 0.99\}$ (all less than 1).

Plot the cross-path average of β_t for each b value on the same figure and comment on how the rate of convergence changes as $b \to 1$.

Solution

Here is one solution:

Part 1. The REE satisfies $\beta_f = T(\beta_f) = a + b\beta_f$, so

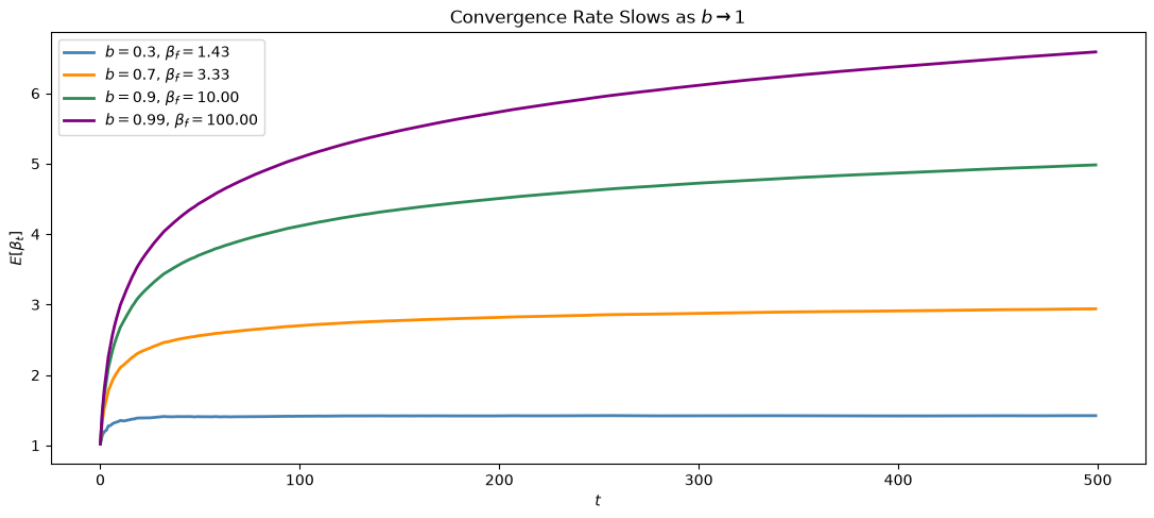
$$\beta_f(1-b) = a \implies \beta_f = \frac{a}{1-b}.$$

Part 2. The small ODE is $\dot{\beta} = a + b\beta - \beta = a - (1-b)\beta$.

This is linear with slope $-(1-b)$, so the unique fixed point $\beta_f = a/(1-b)$ is globally stable iff $1 - b > 0$, i.e., $b < 1$.

Part 3.

```
a_ex, T_ex, N_ex = 1.0, 500, 200
b_values = [0.3, 0.7, 0.9, 0.99]
colors_ex = ['steelblue', 'darkorange', 'seagreen', 'purple']

fig, ax = plt.subplots(figsize=(11, 5))
for b_val, col in zip(b_values, colors_ex):
    def T_fn(β, b_val=b_val):
        return a_ex + b_val * β

    paths = simulate_rls_scalar(T_fn, σ_u=1.0, β0=0.0,
                                T_periods=T_ex, N_paths=N_ex, seed=0)
    bf = a_ex / (1 - b_val)
    ax.plot(np.mean(paths, axis=0), color=col, lw=2,
            label=f'$b={b_val}$, $\\beta_f={bf:.2f}$')

ax.set_xlabel('$t$')
ax.set_ylabel('$E[\\beta_t]$')
ax.set_title('Convergence Rate Slows as $b \\to 1$')
ax.legend()
plt.tight_layout()
plt.show()
```



As $b \to 1$, the Jacobian $\mathcal{M} = b - 1$ approaches zero, so the ODE becomes slow to return to the fixed point.

Convergence still occurs, but it takes longer.

Exercise 124.14.2

Proposition 124.4.3 states that $\mathbb{P}(\beta_t \to \hat{\beta}) = 0$ for any $\hat{\beta} \neq \beta_f$ in the interior.

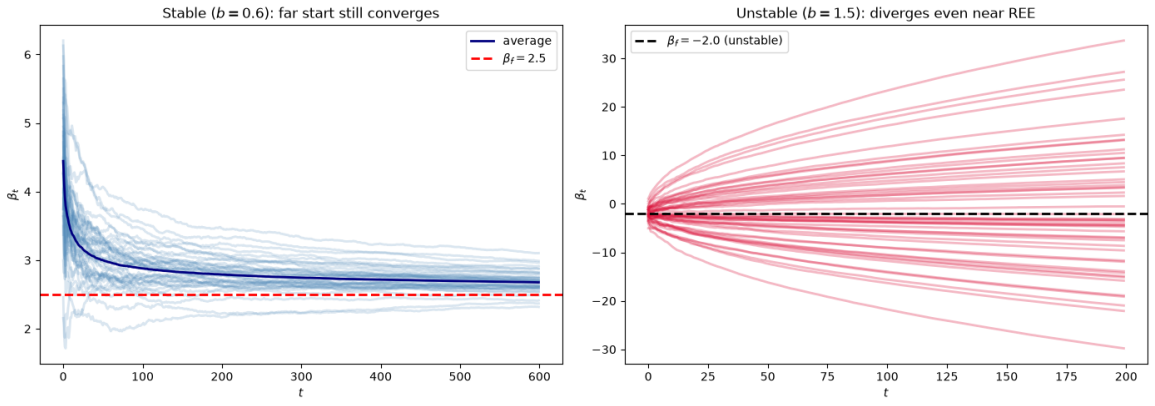
1. Using the Bray model with $a = 1$, $b = 0.6$, simulate 100 paths of length $T = 600$ starting from $\beta_0 = 6$ (far from $\beta_f = 2.5$) and show that paths still converge to β_f.
2. Now consider the *unstable* case $b = 1.5$, simulate 50 paths of length $T = 200$ starting from $\beta_0 = -1.9$ (close to the REE $\beta_f = -2$), and describe what happens.

Solution

Here is one solution:

Parts 1 and 2.

```
fig, axes = plt.subplots(1, 2, figsize=(14, 5))

def T_st(β):
    return 1.0 + 0.6 * β


paths_far = simulate_rls_scalar(T_st, 1.0, β0=6.0,
                                T_periods=600, N_paths=100, seed=1)
ax = axes[0]
for i in range(40):
    ax.plot(paths_far[i], color='steelblue', alpha=0.2, lw=2)
ax.plot(np.mean(paths_far, axis=0), color='navy', lw=2, label='average')
ax.axhline(2.5, color='red', ls='--', lw=2, label='$\\beta_f = 2.5$')
ax.set_title('Stable ($b=0.6$): far start still converges')
ax.set_xlabel('$t$')
ax.set_ylabel('$\\beta_t$')
ax.legend()

def T_un(β):
    return 1.0 + 1.5 * β


β_f_un = 1.0 / (1 - 1.5)
paths_un = simulate_rls_scalar(T_un, 1.0, β0=-1.9,
                               T_periods=200, N_paths=50, seed=2)
ax = axes[1]
for i in range(50):
    ax.plot(paths_un[i], color='crimson', alpha=0.3, lw=2)
ax.axhline(β_f_un, color='black', ls='--', lw=2,
           label=f'$\\beta_f = {β_f_un}$ (unstable)')
ax.set_title('Unstable ($b=1.5$): diverges even near REE')
ax.set_xlabel('$t$')
ax.set_ylabel('$\\beta_t$')
ax.legend()

plt.tight_layout()
plt.show()
```



Geometrically, the slope $dT/d\beta - 1 = b - 1 = 0.5$ is positive at the REE.

The ODE therefore pushes β away from β_f in both directions.

Exercise 124.14.3

The present-value model: effect of λ on E-stability

In the present-value model (124.15), $T(\beta) = (\lambda\beta + 1)\rho$ and the Jacobian is $\mathcal{M} = \lambda\rho - 1$.

1. For $\rho = 0.9$ and each of $\lambda \in \{0.5, 0.8, 0.95, 0.99\}$, compute β_f and $\mathcal{M}$ and determine whether the REE is E-stable.
2. For the E-stable cases, simulate 100 paths of length $T = 400$ and plot representative noisy paths in calendar time, then plot the limiting ODE paths in accumulated-gain time.
3. At $\lambda = 0.99$, $\mathcal{M} = 0.99\rho - 1 < 0$.

   Simulate paths for this near-boundary case and compare the convergence speed with the $\lambda = 0.5$ case, providing an intuitive explanation.

Solution

Here is one solution:

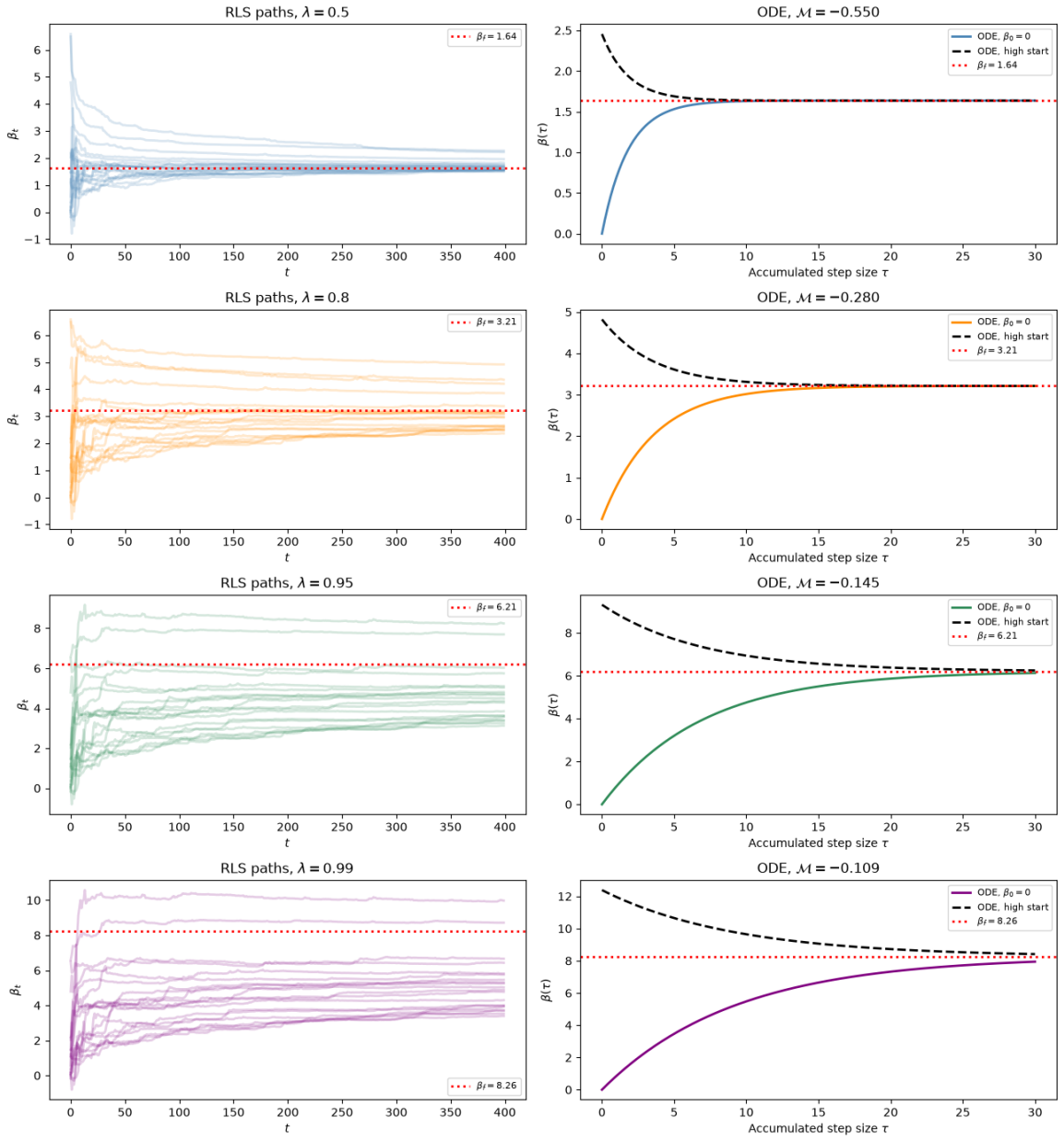
Part 1.

```
ρ_ex = 0.9
λ_values = [0.5, 0.8, 0.95, 0.99]

print(f"{'lambda':>8}  {'β_f':>10}  {'M = λ*ρ-1':>15}  {'E-stable':>10}")
print("-" * 50)
for lv in λ_values:
    bf = ρ_ex / (1 - lv * ρ_ex) if abs(lv * ρ_ex) < 1 else float('inf')
    M_jac = lv * ρ_ex - 1
    estab = "YES" if M_jac < 0 else "NO"
    print(f"{lv:>8.2f}  {bf:>10.4f}  {M_jac:>15.4f}  {estab:>10}")
```

```
  lambda         β_f        M = λ*ρ-1    E-stable
--------------------------------------------------
    0.50      1.6364          -0.5500         YES
    0.80      3.2143          -0.2800         YES
    0.95      6.2069          -0.1450         YES
    0.99      8.2569          -0.1090         YES
```

Parts 2 and 3.

```
fig, axes = plt.subplots(len(λ_values), 2, figsize=(13, 14))
colors_λ = ['steelblue', 'darkorange', 'seagreen', 'purple']

for row, (lv, col) in enumerate(zip(λ_values, colors_λ)):
    def ode_fn(β, λ_val=lv):
        return (λ_val * β + 1) * ρ_ex - β

    bf = ρ_ex / (1 - lv * ρ_ex)

    paths_λ = simulate_present_value_rls(
        lv, ρ_ex, 1.0, β0=0.0,
        T_periods=400, N_paths=100, seed=3
    )

    ax = axes[row, 0]
    for i in range(20):
        ax.plot(paths_λ[i], color=col, alpha=0.2, lw=2)
    ax.axhline(bf, color='red', ls=':', lw=2,
               label=f'$\\beta_f={bf:.2f}$')
    ax.set_title(f'RLS paths, $\\lambda={lv}$')
    ax.set_xlabel('$t$')
    ax.set_ylabel('$\\beta_t$')
    ax.legend(fontsize=8)

    τ_grid = np.linspace(0.0, 30.0, 300)
    sol_low = solve_ivp(
        lambda τ, y: [ode_fn(y[0])],
        (0.0, τ_grid[-1]),
        [0.0],
        t_eval=τ_grid,
        max_step=0.05
    ).y[0]
    sol_high = solve_ivp(
        lambda τ, y: [ode_fn(y[0])],
        (0.0, τ_grid[-1]),
        [1.5 * bf],
        t_eval=τ_grid,
        max_step=0.05
    ).y[0]

    ax = axes[row, 1]
    ax.plot(τ_grid, sol_low, color=col, lw=2,
            label='ODE, $\\beta_0=0$')
    ax.plot(τ_grid, sol_high, color='black', ls='--', lw=2,
            label='ODE, high start')
    ax.axhline(bf, color='red', ls=':', lw=2,
               label=f'$\\beta_f={bf:.2f}$')
    M_jac = lv * ρ_ex - 1
    ax.set_title(f'ODE, $\\mathcal{{M}}={M_jac:.3f}$')
    ax.set_xlabel('Accumulated step size $\\tau$')
    ax.set_ylabel('$\\beta(\\tau)$')
    ax.legend(fontsize=8)

plt.tight_layout()
plt.show()
```



In each row, the left panel shows finite-sample RLS paths in calendar time and the right panel shows deterministic ODE paths in accumulated-gain time.

When $\lambda = 0.99$, $\mathcal{M} = 0.99\rho - 1 \approx -0.109$ is small in absolute value.

The ODE is nearly flat near β_f, so the restoring force is weak and convergence is slow.

When $\lambda = 0.5$, $\mathcal{M} = -0.55$, which gives a stronger restoring force and faster convergence.

CHAPTER

FIVE

AFFINE MODELS OF ASSET PRICES

125.1 Overview

This lecture describes a class of **affine** or **exponential quadratic** models of the stochastic discount factor that have become widely used in empirical finance.

These models are presented in chapter 15 of Ljungqvist and Sargent [2018].

The models discussed here take a different approach from the time-separable CRRA stochastic discount factor of Hansen and Singleton [1983], studied in our *companion lecture*.

The CRRA stochastic discount factor is

$$m_{t+1} = \exp\left(-r_t - \frac{1}{2}\sigma_c^2\gamma^2 - \gamma\sigma_c\varepsilon_{t+1}\right)$$

where $r_t = \rho + \gamma\mu - \frac{1}{2}\sigma_c^2\gamma^2$.

This model asserts that exposure to the random part of aggregate consumption growth, $\sigma_c\varepsilon_{t+1}$, is the *only* priced risk, the sole source of discrepancies among expected returns across assets.

Empirical difficulties with this specification (the equity premium puzzle, the risk-free rate puzzle, and the Hansen-Jagannathan bounds discussed in Doubts or Variability?) motivate the alternative approach described in this lecture.

Put bluntly, the model to be studied in this lecture declares the Lucas asset pricing model's stochastic discount factor to be a failure.

The **affine model** maintains $\mathbb{E}(m_{t+1}R_{j,t+1}) = 1$ but *divorces* the stochastic discount factor from consumption risk, and consequently, from much of macroeconomics too.

Instead, it

- specifies an analytically tractable stochastic process for m_{t+1}, and
- uses overidentifying restrictions from $\mathbb{E}(m_{t+1}R_{j,t+1}) = 1$ applied to N assets to let the data reveal risks and their prices.

Note

Researchers including [Bansal and Yaron, 2004] and [Hansen *et al.*, 2008] have been less willing to give up on consumption-based models of the stochastic discount factor.

Key applications we study include:

1. *Pricing risky assets*: how risk prices and exposures determine excess returns.
2. *Affine term structure models*: bond yields as affine functions of a state vector (Ang and Piazzesi [2003]).

3. *Risk-neutral probabilities*: a change-of-measure representation of the pricing equation.
4. *Distorted beliefs*: reinterpreting risk price estimates when agents hold systematically biased forecasts (Piazzesi *et al.* [2015]); see also Risk Aversion or Mistaken Beliefs?.

We start with the following imports:

```
import numpy as np
import matplotlib.pyplot as plt
from collections import namedtuple
from numpy.linalg import eigvals
```

125.2 The model

125.2.1 State dynamics and short rate

The model has two components.

Component 1 is a vector autoregression that describes the state of the economy and the evolution of the short rate:

$$z_{t+1} = \mu + \phi z_t + C\varepsilon_{t+1} \tag{125.1}$$

$$r_t = \delta_0 + \delta_1^\top z_t \tag{125.2}$$

Here

- ϕ is a stable $m \times m$ matrix,
- C is an $m \times m$ matrix,
- $\varepsilon_{t+1} \sim \mathcal{N}(0, I)$ is an i.i.d. $m \times 1$ random vector,
- z_t is an $m \times 1$ state vector.

Equation (125.2) says that the **short rate** r_t, the net yield on a one-period risk-free claim, is an affine function of the state z_t.

Component 2 is a vector of **risk prices** λ_t and an associated stochastic discount factor m_{t+1}:

$$\lambda_t = \lambda_0 + \lambda_z z_t \tag{125.3}$$

$$\log(m_{t+1}) = -r_t - \frac{1}{2}\lambda_t^\top \lambda_t - \lambda_t^\top \varepsilon_{t+1} \tag{125.4}$$

Here λ_0 is $m \times 1$ and λ_z is $m \times m$.

The entries of λ_t that multiply corresponding entries of the risks ε_{t+1} are called **risk prices** because they determine how exposures to each risk component affect expected returns (as we show below).

Because λ_t is affine in z_t, the stochastic discount factor m_{t+1} is **exponential quadratic** in the state z_t.

We implement the model components as follows.

```
AffineModel = namedtuple('AffineModel',
    ('μ', 'φ', 'C', 'δ_0', 'δ_1', 'λ_0', 'λ_z', 'm', 'φ_rn', 'μ_rn'))

def create_affine_model(μ, φ, C, δ_0, δ_1, λ_0, λ_z):
    """Create an affine term structure model."""
    μ = np.asarray(μ, float)
```

(continues on next page)

(continued from previous page)

```
    Ψ = np.asarray(Ψ, float)
    C = np.asarray(C, float)
    δ_1 = np.asarray(δ_1, float)
    λ_0, λ_z = np.asarray(λ_0, float), np.asarray(λ_z, float)
    return AffineModel(μ=μ, Ψ=Ψ, C=C, δ_0=float(δ_0), δ_1=δ_1,
                       λ_0=λ_0, λ_z=λ_z, m=len(μ),
                       Ψ_rn=Ψ - C @ λ_z, μ_rn=μ - C @ λ_0)

def simulate(model, z0, T, rng=None):
    """Simulate z_{t+1} = μ + Ψ z_t + C ε_{t+1} for T periods."""
    if rng is None:
        rng = np.random.default_rng(42)
    Z = np.zeros((T + 1, model.m))
    Z[0] = z0
    for t in range(T):
        ε = rng.standard_normal(model.m)
        Z[t + 1] = model.μ + model.Ψ @ Z[t] + model.C @ ε
    return Z

def short_rate(model, z):
    """Compute r_t = δ_0 + δ_1^⊤ z_t."""
    return model.δ_0 + model.δ_1 @ z

def risk_prices(model, z):
    """Compute λ_t = λ_0 + λ_z z_t."""
    return model.λ_0 + model.λ_z @ z
```

125.2.2 Properties of the SDF

Since $\lambda_t^\top \varepsilon_{t+1}$ is conditionally normal, it follows that

$$\mathbb{E}_t(m_{t+1}) = \exp(-r_t)$$

and

$$\operatorname{std}_t(m_{t+1}) \approx \|\lambda_t\|.$$

Exercise 125.2.1

Show that the SDF defined in (125.4) satisfies

$$\mathbb{E}_t(m_{t+1}) = \exp(-r_t)$$

and

$$\operatorname{std}_t(m_{t+1}) \approx \|\lambda_t\|$$

where $\|\lambda_t\| = \sqrt{\lambda_t^\top \lambda_t}$ denotes the Euclidean norm of the risk price vector.

For the second result, use the lognormal variance formula and the approximations $\exp(x) \approx 1 + x$ and $\exp(-r_t) \approx 1$ for small x and r_t.

Solution

From (125.4), we have

$$m_{t+1} = \exp\left(-r_t - \frac{1}{2}\lambda_t^\top \lambda_t - \lambda_t^\top \varepsilon_{t+1}\right)$$

Since $-\lambda_t^\top \varepsilon_{t+1} \sim \mathcal{N}(0, \lambda_t^\top \lambda_t)$, we have $\mathbb{E}_t[\exp(-\lambda_t^\top \varepsilon_{t+1})] = \exp\left(\frac{1}{2}\lambda_t^\top \lambda_t\right)$.

Therefore,

$$\mathbb{E}_t(m_{t+1}) = \exp(-r_t - \frac{1}{2}\lambda_t^\top \lambda_t)\mathbb{E}_t[\exp(-\lambda_t^\top \varepsilon_{t+1})] = \exp(-r_t)$$

m_{t+1} is conditionally lognormal with $\log m_{t+1} \sim \mathcal{N}(-r_t - \frac{1}{2}\lambda_t^\top \lambda_t, \lambda_t^\top \lambda_t)$.

By the lognormal variance formula $\mathrm{Var}(\exp(X)) = (\exp(\sigma^2) - 1)\exp(2\mu + \sigma^2)$ for $X \sim \mathcal{N}(\mu, \sigma^2)$, we have

$$\begin{aligned} \mathrm{Var}_t(m_{t+1}) &= (\exp(\lambda_t^\top \lambda_t) - 1)\exp(-2r_t) \\ &\approx \lambda_t^\top \lambda_t \exp(-2r_t) \end{aligned}$$

by the approximation $\exp(x) \approx 1 + x$ for small x.

Hence,

$$\mathrm{std}_t(m_{t+1}) \approx \|\lambda_t\| \exp(-r_t)$$

With $\exp(-r_t) \approx 1$ for small r_t, we obtain

$$\mathrm{std}_t(m_{t+1}) \approx \|\lambda_t\|$$

The first equation confirms that r_t is the net yield on a risk-free one-period bond.

That is why r_t is called **the short rate** in the exponential quadratic literature.

The second equation says that the conditional standard deviation of the SDF is approximately the magnitude of the vector of risk prices, a measure of overall **market price of risk**.

125.3 Pricing risky assets

125.3.1 Lognormal returns

Consider a risky asset j whose gross return has a lognormal conditional distribution:

$$R_{j,t+1} = \exp\left(\nu_t(j) - \frac{1}{2}\alpha_t(j)^\top \alpha_t(j) + \alpha_t(j)^\top \varepsilon_{t+1}\right) \tag{125.5}$$

where the **exposure vector**

$$\alpha_t(j) = \alpha_0(j) + \alpha_z(j)\, z_t \tag{125.6}$$

Here $\alpha_0(j)$ is $m \times 1$ and $\alpha_z(j)$ is $m \times m$.

The components of $\alpha_t(j)$ express the **exposures** of $\log R_{j,t+1}$ to corresponding components of the risk vector ε_{t+1}.

The specification (125.5) implies $\mathbb{E}_t R_{j,t+1} = \exp(\nu_t(j))$, so $\nu_t(j)$ is the log of the expected gross return.

125.3.2 Expected excess returns

Applying the pricing equation $\mathbb{E}_t(m_{t+1}R_{j,t+1}) = 1$ together with the formula for the mean of a lognormal random variable gives

$$\nu_t(j) = r_t + \alpha_t(j)^\top \lambda_t \tag{125.7}$$

Exercise 125.3.1

Using the SDF (125.4) and the return specification (125.5), derive the expected excess return formula (125.7):

$$\nu_t(j) = r_t + \alpha_t(j)^\top \lambda_t$$

Hint: Start by computing $\log(m_{t+1}R_{j,t+1})$, identify its conditional distribution, and apply the pricing condition $\mathbb{E}_t(m_{t+1}R_{j,t+1}) = 1$.

Solution

Combining (125.4) and (125.5), we get

$$\log(m_{t+1}R_{j,t+1}) = -r_t + \nu_t(j) - \frac{1}{2}\lambda_t^\top\lambda_t - \frac{1}{2}\alpha_t(j)^\top\alpha_t(j) + (\alpha_t(j) - \lambda_t)^\top\varepsilon_{t+1}$$

This is conditionally normal with mean $\mu = -r_t + \nu_t(j) - \frac{1}{2}\lambda_t^\top\lambda_t - \frac{1}{2}\alpha_t(j)^\top\alpha_t(j)$ and variance $\sigma^2 = (\alpha_t(j) - \lambda_t)^\top(\alpha_t(j) - \lambda_t)$.

Since $\mathbb{E}_t[\exp(X)] = \exp(\mu + \frac{1}{2}\sigma^2)$ for $X \sim \mathcal{N}(\mu, \sigma^2)$, the pricing condition $\mathbb{E}_t(m_{t+1}R_{j,t+1}) = 1$ requires $\mu + \frac{1}{2}\sigma^2 = 0$.

Expanding $\frac{1}{2}\sigma^2 = \frac{1}{2}\alpha_t(j)^\top\alpha_t(j) - \alpha_t(j)^\top\lambda_t + \frac{1}{2}\lambda_t^\top\lambda_t$ and adding to μ, the $\frac{1}{2}\lambda_t^\top\lambda_t$ and $\frac{1}{2}\alpha_t(j)^\top\alpha_t(j)$ terms cancel, leaving

$$-r_t + \nu_t(j) - \alpha_t(j)^\top\lambda_t = 0$$

which gives (125.7).

This is a central result.

It says:

> The log expected gross return on asset j equals the short rate plus the inner product of the asset's exposure vector $\alpha_t(j)$ with the risk price vector λ_t.

Each component of λ_t prices the corresponding component of ε_{t+1}.

An asset that loads heavily on a risk component with a large risk price earns a correspondingly high expected return.

125.4 Affine term structure of yields

One of the most important applications is the **affine term structure model** studied by Ang and Piazzesi [2003].

125.4.1 Bond prices

Let $p_t(n)$ be the price at time t of a risk-free pure discount bond maturing at $t+n$ (paying one unit of consumption).

The one-period gross return on holding an $(n+1)$-period bond from t to $t+1$ is

$$R_{t+1} = \frac{p_{t+1}(n)}{p_t(n+1)}$$

The pricing equation $\mathbb{E}_t(m_{t+1} R_{t+1}) = 1$ implies

$$p_t(n+1) = \mathbb{E}_t(m_{t+1}\, p_{t+1}(n)) \tag{125.8}$$

with the initial condition

$$p_t(1) = \mathbb{E}_t(m_{t+1}) = \exp(-r_t) = \exp(-\delta_0 - \delta_1^\top z_t).$$

125.4.2 Exponential affine prices

The recursion (125.8) has an **exponential affine** solution:

$$p_t(n) = \exp(\bar{A}_n + \bar{B}_n^\top z_t) \tag{125.9}$$

where the scalar $\bar{A}_n$ and the $m \times 1$ vector $\bar{B}_n$ satisfy the **Riccati difference equations**

$$\bar{A}_{n+1} = \bar{A}_n + \bar{B}_n^\top(\mu - C\lambda_0) + \frac{1}{2}\bar{B}_n^\top CC^\top \bar{B}_n - \delta_0 \tag{125.10}$$

$$\bar{B}_{n+1}^\top = \bar{B}_n^\top(\phi - C\lambda_z) - \delta_1^\top \tag{125.11}$$

with initial conditions $\bar{A}_1 = -\delta_0$ and $\bar{B}_1 = -\delta_1$.

Exercise 125.4.1

Derive the Riccati difference equations (125.10) and (125.11) by substituting the conjectured bond price (125.9) into the pricing recursion (125.8) and matching coefficients.

Hint: Substitute $p_{t+1}(n) = \exp(\bar{A}_n + \bar{B}_n^\top z_{t+1})$ and $\log m_{t+1}$ from (125.4) into (125.8).

Use the state dynamics (125.1) to express z_{t+1} in terms of z_t and ε_{t+1}, then evaluate the conditional expectation using the lognormal moment generating function.

Solution

We want to show that if $p_t(n) = \exp(\bar{A}_n + \bar{B}_n^\top z_t)$, then the recursion $p_t(n+1) = \mathbb{E}_t(m_{t+1}\, p_{t+1}(n))$ yields $p_t(n+1) = \exp(\bar{A}_{n+1} + \bar{B}_{n+1}^\top z_t)$ with $\bar{A}_{n+1}$ and $\bar{B}_{n+1}$ given by (125.10) and (125.11).

From (125.4) and (125.9),

$$\log(m_{t+1}\, p_{t+1}(n)) = -r_t - \frac{1}{2}\lambda_t^\top \lambda_t - \lambda_t^\top \varepsilon_{t+1} + \bar{A}_n + \bar{B}_n^\top z_{t+1}$$

Substituting $z_{t+1} = \mu + \phi z_t + C\varepsilon_{t+1}$ from (125.1) and $r_t = \delta_0 + \delta_1^\top z_t$ from (125.2) gives

$$\log(m_{t+1}\, p_{t+1}(n)) = \bar{A}_n + \bar{B}_n^\top \mu - \delta_0 + (\bar{B}_n^\top \phi - \delta_1^\top) z_t - \frac{1}{2}\lambda_t^\top \lambda_t + (\bar{B}_n^\top C - \lambda_t^\top)\varepsilon_{t+1}$$

Since $\varepsilon_{t+1} \sim \mathcal{N}(0, I)$, and writing the exponent as $a + b^\top \varepsilon_{t+1}$ where $b = C^\top \bar{B}_n - \lambda_t$, we have

$$\mathbb{E}_t[\exp(a + b^\top \varepsilon_{t+1})] = \exp\left(a + \frac{1}{2} b^\top b\right)$$

Computing $\frac{1}{2} b^\top b$:

$$\frac{1}{2}(\bar{B}_n^\top C - \lambda_t^\top)(\bar{B}_n^\top C - \lambda_t^\top)^\top = \frac{1}{2}\bar{B}_n^\top C C^\top \bar{B}_n - \bar{B}_n^\top C \lambda_t + \frac{1}{2}\lambda_t^\top \lambda_t$$

The $\frac{1}{2}\lambda_t^\top \lambda_t$ cancels with the $-\frac{1}{2}\lambda_t^\top \lambda_t$ already in a, and $-\bar{B}_n^\top C \lambda_t = -\bar{B}_n^\top C(\lambda_0 + \lambda_z z_t)$.

$$\log p_t(n+1) = \underbrace{\bar{A}_n + \bar{B}_n^\top(\mu - C\lambda_0) + \frac{1}{2}\bar{B}_n^\top C C^\top \bar{B}_n - \delta_0}_{\bar{A}_{n+1}} + \underbrace{(\bar{B}_n^\top(\phi - C\lambda_z) - \delta_1^\top)}_{\bar{B}_{n+1}^\top} z_t$$

Matching the constant and the coefficient on z_t gives the Riccati equations (125.10) and (125.11).

Setting $n = 0$ with $p_t(1) = \exp(-r_t) = \exp(-\delta_0 - \delta_1^\top z_t)$ gives $\bar{A}_1 = -\delta_0$ and $\bar{B}_1 = -\delta_1$.

125.4.3 Yields

The **yield to maturity** on an n-period bond is the constant rate y at which one would discount the face value to obtain the observed price, i.e., $p_t(n) = e^{-n\,y}$.

Solving for y gives

$$y_t(n) = -\frac{\log p_t(n)}{n}$$

Substituting (125.9) gives

$$y_t(n) = A_n + B_n^\top z_t \tag{125.12}$$

where $A_n = -\bar{A}_n / n$ and $B_n = -\bar{B}_n / n$.

Yields are affine functions of the state vector z_t.

This is the defining property of affine term structure models.

We now implement the bond pricing formulas (125.10), (125.11), and (125.12).

```
def bond_coefficients(model, n_max):
    """Compute (A_bar_n, B_bar_n) for n = 1, ..., n_max."""
    A_bar = np.zeros(n_max + 1)
    B_bar = np.zeros((n_max + 1, model.m))
    A_bar[1], B_bar[1] = -model.δ_0, -model.δ_1
    CC = model.C @ model.C.T
    for n in range(1, n_max):
        Bn = B_bar[n]
        A_bar[n + 1] = (A_bar[n] + Bn @ model.μ_rn
                        + 0.5 * Bn @ CC @ Bn
```

(continues on next page)

(continued from previous page)

```
                        - model.δ_0)
        B_bar[n + 1] = model.ψ_rn.T @ Bn - model.δ_1
    return A_bar, B_bar

def compute_yields(model, z, n_max):
    """Compute yield curve y_t(n) for n = 1, ..., n_max."""
    A_bar, B_bar = bond_coefficients(model, n_max)
    return np.array([(-A_bar[n] - B_bar[n] @ z) / n
                     for n in range(1, n_max + 1)])

def bond_prices(model, z, n_max):
    """Compute bond prices p_t(n) for n = 1, ..., n_max."""
    A_bar, B_bar = bond_coefficients(model, n_max)
    return np.array([np.exp(A_bar[n] + B_bar[n] @ z)
                     for n in range(1, n_max + 1)])
```

125.4.4 A one-factor Gaussian example

To build intuition, we start with a single-factor ($m = 1$) Gaussian model.

With $m = 1$, the state z_t follows an AR(1) process $z_{t+1} = \mu + \phi z_t + C\varepsilon_{t+1}$.

The unconditional standard deviation of z_t is $\sigma_z = C/\sqrt{1-\phi^2}$, which determines the range of short rates the model generates via $r_t = \delta_0 + \delta_1 z_t$.

```
# One-factor Gaussian model (quarterly)
μ      = np.array([0.0])
ψ      = np.array([[0.95]])
C      = np.array([[1.0]])
δ_0    = 0.01                   # 1%/quarter ≈ 4% p.a.
δ_1    = np.array([0.001])
λ_0    = np.array([-0.05])
λ_z    = np.array([[-0.01]])

model_1f = create_affine_model(μ, ψ, C, δ_0, δ_1, λ_0, λ_z)
```

125.4.5 Yield curve shapes

We compute yield curves $y_t(n)$ across a range of short-rate states z_t.

```
n_max_1f = 60
maturities_1f = np.arange(1, n_max_1f + 1)

z_low  = np.array([-5.0])
z_mid  = np.array([0.0])
z_high = np.array([5.0])

fig, ax = plt.subplots(figsize=(9, 5.5))

r_low = short_rate(model_1f, z_low) * 4 * 100
r_mid = short_rate(model_1f, z_mid) * 4 * 100
r_high = short_rate(model_1f, z_high) * 4 * 100

for z, label in [
```

(continues on next page)

(continued from previous page)

```
    (z_low,  f"Low state  ($y_t(1) = ${r_low:.1f}%)"),
    (z_mid,  f"Median state ($y_t(1) = ${r_mid:.1f}%)"),
    (z_high, f"High state ($y_t(1) = ${r_high:.1f}%)"),
]:
    y = compute_yields(model_1f, z, n_max_1f) * 4 * 100
    line, = ax.plot(maturities_1f, y, lw=2.2, label=label)
    ax.plot(1, y[0], 'o', color=line.get_color(), ms=7, zorder=5)

r_bar = short_rate(model_1f, np.array([0.0])) * 4 * 100
ax.axhline(r_bar, color='grey', ls=':', lw=1.2, alpha=0.7,
           label=f"Mean short rate ({r_bar:.1f}%)")

# Long-run yield: B_bar_n converges, so y_inf = lim -A_bar_n / n
ψ_Cλ = (model_1f.ψ_rn)[0, 0]           # ψ - Cλ_z (scalar)
B_inf = -model_1f.δ_1[0] / (1 - ψ_Cλ) # fixed point of B recursion
A_increment = (B_inf * model_1f.μ_rn[0]
               + 0.5 * B_inf**2 * (model_1f.C @ model_1f.C.T)[0, 0]
               - model_1f.δ_0)
y_inf = -A_increment * 4 * 100         # annualised %
ax.axhline(y_inf, color='black', ls='--', lw=1.2, alpha=0.7,
           label=f"Long-run yield ({y_inf:.1f}%)")

ax.set_xlabel("Maturity (quarters)")
ax.set_ylabel("Yield (% per annum)")
ax.legend(fontsize=10, loc='best')
ax.set_xlim(1, n_max_1f)

ax2 = ax.twiny()
ax2.set_xlim(ax.get_xlim())
year_ticks = [4, 20, 40, 60]
ax2.set_xticks(year_ticks)
ax2.set_xticklabels([f"{t/4:.0f}y" for t in year_ticks])
ax2.set_xlabel("Maturity (years)")

plt.tight_layout()
plt.show()
```

When the short rate is low, the yield curve is upward-sloping, while when the short rate is high, it is downward-sloping.

All three curves converge to the same long-run yield y_∞ at long maturities, and the long-run yield lies above the mean short rate δ_0.

Exercise 125.4.2

Show that the long-run yield satisfies

$$y_\infty = \delta_0 - \bar{B}_\infty^\top(\mu - C\lambda_0) - \tfrac{1}{2}\bar{B}_\infty^\top CC^\top \bar{B}_\infty \tag{125.13}$$

where $\bar{B}_\infty = -(I - (\phi - C\lambda_z)^\top)^{-1}\delta_1$ is the fixed point of the recursion (125.11).

Then explain why $y_\infty > \delta_0$ under this parameterization.

Hint: Use (125.12) and the Riccati equations (125.10)–(125.11). For the inequality, consider each subtracted term separately.

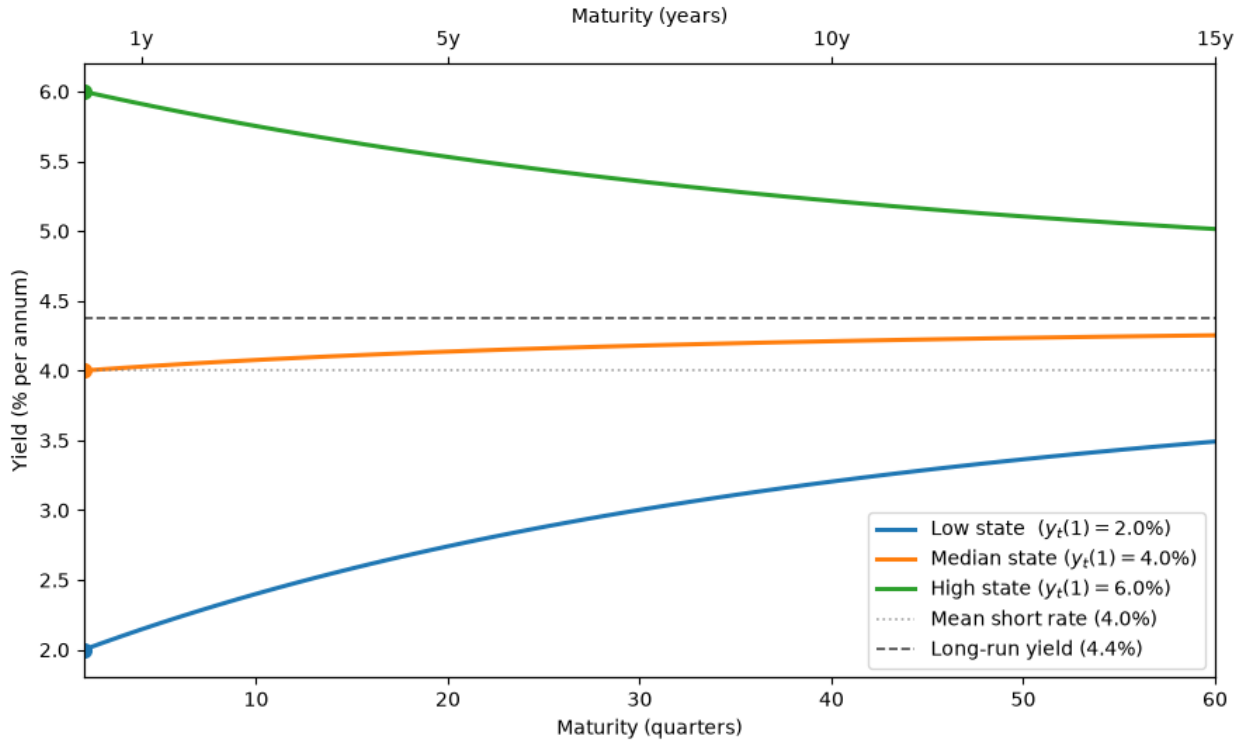


Fig. 125.1: Yield curves under the one-factor affine model

Solution

The recursion (125.11) is a linear difference equation $\bar{B}_{n+1} = (\phi - C\lambda_z)^\top \bar{B}_n - \delta_1$.

When $\phi - C\lambda_z$ has eigenvalues inside the unit circle, $\bar{B}_n$ converges to $\bar{B}_\infty = -(I - (\phi - C\lambda_z)^\top)^{-1}\delta_1$.

Since $\bar{B}_\infty$ is finite, $\bar{B}_n^\top z_t / n \to 0$ in (125.12), so $y_t(n) \to \lim_{n\to\infty} -\bar{A}_n/n$ regardless of z_t.

To find this limit, write $\bar{A}_n = \bar{A}_1 + \sum_{k=1}^{n-1}(\bar{A}_{k+1} - \bar{A}_k)$.

By (125.10), each increment depends on $\bar{B}_k$, which converges to $\bar{B}_\infty$, so the increment converges to $L \equiv \bar{B}_\infty^\top(\mu - C\lambda_0) + \frac{1}{2}\bar{B}_\infty^\top CC^\top \bar{B}_\infty - \delta_0$.

Therefore $\bar{A}_n/n \to L$ and $y_\infty = -L$, giving (125.13).

To see why $y_\infty > \delta_0$, note that the two subtracted terms in (125.13) have opposite signs under this parameterization.

The quadratic term $\frac{1}{2}\bar{B}_\infty^\top CC^\top \bar{B}_\infty = \frac{1}{2}\|C^\top \bar{B}_\infty\|^2 \geq 0$ always.

This is a **convexity effect** from Jensen's inequality that pushes y_∞ below δ_0.

The linear term $\bar{B}_\infty^\top(\mu - C\lambda_0)$ is negative because $\bar{B}_\infty < 0$ (since $\delta_1 > 0$) while $\mu - C\lambda_0 > 0$ (since $\lambda_0 < 0$). Subtracting this negative quantity raises y_∞ above δ_0.

This is a **risk-premium effect**: positive term premiums tilt the average yield curve upward.

Under this parameterization the risk-premium effect dominates the convexity effect, so $y_\infty > \delta_0$.

Let's also simulate the short rate path:

```
T = 200
Z = simulate(model_1f, np.array([0.0]), T)
short_rates = np.array([short_rate(model_1f, Z[t]) * 4 * 100
                        for t in range(T + 1)])
r_bar_pct = short_rate(model_1f, np.array([0.0])) * 4 * 100

fig, ax = plt.subplots(figsize=(10, 4))
quarters = np.arange(T + 1)
line, = ax.plot(quarters, short_rates, lw=1.3)
ax.axhline(r_bar_pct, ls="--", lw=1.3,
           label=f"Unconditional mean ({r_bar_pct:.1f}%)")
ax.fill_between(quarters, short_rates, r_bar_pct,
                alpha=0.08, color=line.get_color())
ax.set_xlabel("Quarter")
ax.set_ylabel("Short rate (% p.a.)")
ax.set_xlim(0, T)
ax.legend(fontsize=11)
plt.tight_layout()
plt.show()
```

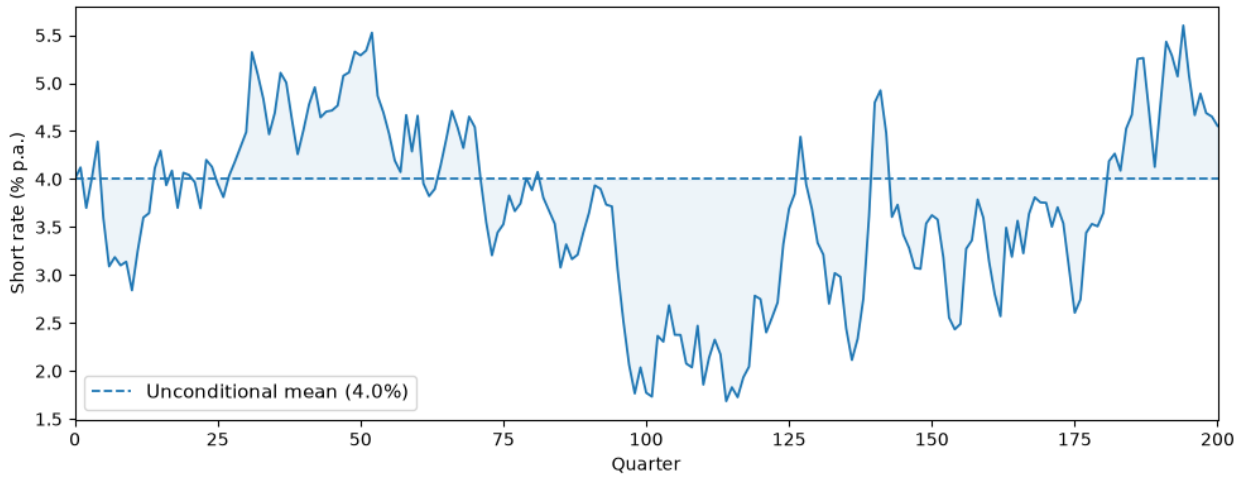


Fig. 125.2: Simulated short rate path

125.4.6 A two-factor model

To match richer yield-curve dynamics, practitioners routinely use $m \geq 2$ factors.

We now introduce a two-factor specification with state $z_t = (z_{1t},\, z_{2t})^\top$, where

$$z_{t+1} = \mu + \phi\, z_t + C\, \varepsilon_{t+1}, \qquad \phi = \begin{pmatrix} 0.97 & -0.03 \\ 0 & 0.90 \end{pmatrix}, \qquad C = I_2$$

The first factor z_{1t} is highly persistent ($\phi_{11} = 0.97$) and drives most of the variation in the short rate through δ_1, so we interpret it as a **level** factor.

The second factor z_{2t} mean-reverts faster ($\phi_{22} = 0.90$) and affects the short rate with a smaller loading, capturing the **slope** of the yield curve.

The off-diagonal entry $\phi_{12} = -0.03$ allows the level factor to respond to the current slope state z_{2t}.

The short rate is $r_t = \delta_0 + \delta_1^\top z_t$ with $\delta_1 = (0.002,\ 0.001)^\top$, so both factors raise the short rate when positive, but the level factor has twice the impact.

Risk prices are $\lambda_t = \lambda_0 + \lambda_z z_t$ with $\lambda_0 = (-0.01,\ -0.005)^\top$ and $\lambda_z = \text{diag}(-0.005,\ -0.003)$.

The negative diagonal entries of λ_z make $\phi - C\lambda_z$ have larger eigenvalues than ϕ, so the state is more persistent under the risk-neutral measure and the yield curve is more sensitive to the current state at long horizons.

```
# Two-factor model: z = [level, slope]
μ_2  = np.array([0.0,  0.0])
φ_2  = np.array([[0.97, -0.03],
                 [0.00,  0.90]])
C_2  = np.eye(2)
δ_0_2 = 0.01
δ_1_2 = np.array([0.002, 0.001])
λ_0_2 = np.array([-0.01, -0.005])
λ_z_2 = np.array([[-0.005, 0.0],
                  [ 0.0, -0.003]])

model_2f = create_affine_model(μ_2, φ_2, C_2, δ_0_2, δ_1_2, λ_0_2, λ_z_2)

print(f"Eigenvalues of φ:       {eigvals(φ_2).real.round(4)}")
print(f"Eigenvalues of φ - Cλ_z: {eigvals(model_2f.φ_rn).real.round(4)}")
```

```
Eigenvalues of φ:       [0.97 0.9 ]
Eigenvalues of φ - Cλ_z: [0.975 0.903]
```

This confirms that the eigenvalues of $\phi - C\lambda_z$ are larger than those of ϕ, so the state is more persistent under the risk-neutral measure.

The following figure shows yield curves across different states of the world, as well as the factor loadings $B_{n,1}$ and $B_{n,2}$ that determine how yields load on the level and slope factors at each maturity

```
n_max_2f = 60
maturities_2f = np.arange(1, n_max_2f + 1)

states = {
    "Normal":              np.array([0.0,   0.0]),
    "Low short rate":      np.array([-4.0,  3.0]),
    "High short rate":     np.array([4.0,  -3.0]),
}

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 5.5))

for label, z in states.items():
    r_now = short_rate(model_2f, z) * 4 * 100
    y = compute_yields(model_2f, z, n_max_2f) * 4 * 100
    line, = ax1.plot(maturities_2f, y, lw=2.2,
                     label=f"{label} ($r_1$ = {r_now:.1f}%)")
    ax1.plot(1, y[0], 'o', color=line.get_color(), ms=7, zorder=5)

ax1.set_xlabel("Maturity (quarters)")
ax1.set_ylabel("Yield (% p.a.)")
ax1.legend(fontsize=10)
ax1.set_xlim(1, n_max_2f)

A_bar, B_bar = bond_coefficients(model_2f, n_max_2f)
```

(continues on next page)

(continued from previous page)

```
ns = np.arange(1, n_max_2f + 1)
B_n = np.array([-B_bar[n] / n for n in ns])

ax2.plot(ns, B_n[:, 0], lw=2.2,
         label=r"Level loading $B_{n,1}$")
ax2.plot(ns, B_n[:, 1], lw=2.2,
         label=r"Slope loading $B_{n,2}$")
ax2.axhline(0, color='black', lw=0.6)
ax2.set_xlabel("Maturity (quarters)")
ax2.set_ylabel(r"Yield loading $B_{n,k}$")
ax2.legend(fontsize=11)
ax2.set_xlim(1, n_max_2f)

for ax in (ax1, ax2):
    ax_top = ax.twiny()
    ax_top.set_xlim(ax.get_xlim())
    year_ticks = [4, 12, 20, 40, 60]
    ax_top.set_xticks(year_ticks)
    ax_top.set_xticklabels([f"{t/4:.0f}y" for t in year_ticks])

plt.tight_layout()
plt.show()
```

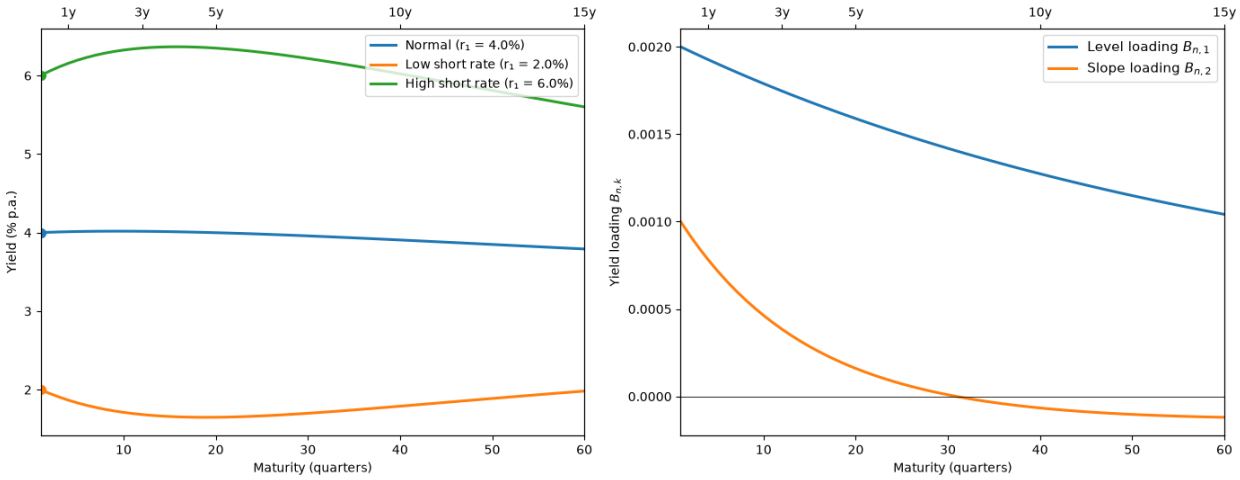


Fig. 125.3: Yield curves and factor loadings under the two-factor model

We can see that the level factor dominates at long maturities.

125.5 Risk premiums

A key object in the affine term structure model is the **term premium**, the expected excess return on a long-term bond relative to rolling over short-term bonds.

For an $(n+1)$-period bond held for one period, the shock loading is $\alpha_n = C^\top \bar{B}_n$, so (125.7) gives

$$\log \mathbb{E}_t R_{t+1}^{(n+1)} - r_t \;=\; \bar{B}_n^\top C \lambda_t$$

The term premium equals the inner product of the bond's shock exposure $\bar{B}_n^\top C$ with the risk price vector λ_t.

Because the term premium equals $\bar{B}_n^\top C \lambda_t$, its sign depends on the *current* risk-price vector λ_t, which is state-dependent whenever $\lambda_z \neq 0$.

To see this more concretely, consider a state where $C\lambda_t$ is negative componentwise (for example, $z_t = 0$ in our calibration below).

When $\delta_1 > 0$, a positive shock ε_{t+1} raises the short rate and lowers long-bond prices, so the bond shock loading $\alpha_n = C^\top \bar{B}_n$ is negative.

A negative $C\lambda_t$ then means the stochastic discount factor m_{t+1} loads positively on ε_{t+1}, i.e. the SDF is high in states where interest rates rise and bond prices fall.

This makes $\mathrm{Cov}_t(m_{t+1}, R_{t+1}^{(n+1)}) < 0$, so long bonds are risky and carry a positive term premium.

Algebraically, $\bar{B}_n < 0$ and $C\lambda_t < 0$ combine to give $\bar{B}_n^\top C \lambda_t > 0$.

In other states, however, λ_t may change sign (e.g. the first component flips in the low-rate regime of our two-state calibration), and long-bond term premiums can become negative at longer maturities.

Exercise 125.5.1

Derive the term premium formula above by computing the one-period holding return on an $(n+1)$-period bond and identifying its shock loading.

Hint: Use $R_{t+1}^{(n+1)} = p_{t+1}(n)/p_t(n+1)$ with $\log p_t(n) = \bar{A}_n + \bar{B}_n^\top z_t$, substitute the state dynamics (125.1), and apply the Riccati equations (125.10)–(125.11) to simplify.

Solution

The one-period holding return on an $(n+1)$-period bond is $R_{t+1}^{(n+1)} = p_{t+1}(n)/p_t(n+1)$, so

$$\log R_{t+1}^{(n+1)} = \bar{A}_n + \bar{B}_n^\top z_{t+1} - \bar{A}_{n+1} - \bar{B}_{n+1}^\top z_t$$

Substituting $z_{t+1} = \mu + \phi z_t + C\varepsilon_{t+1}$ from (125.1):

$$= \underbrace{(\bar{A}_n + \bar{B}_n^\top \mu - \bar{A}_{n+1})}_{\text{constant}} + \underbrace{(\bar{B}_n^\top \phi - \bar{B}_{n+1}^\top)}_{\text{loading on } z_t} z_t + \underbrace{\bar{B}_n^\top C}_{\text{shock loading}} \varepsilon_{t+1}$$

We now use the Riccati equations to simplify each piece.

For the constant piece, (125.10) gives $\bar{A}_{n+1} = \bar{A}_n + \bar{B}_n^\top(\mu - C\lambda_0) + \frac{1}{2}\bar{B}_n^\top CC^\top \bar{B}_n - \delta_0$, so

$$\bar{A}_n + \bar{B}_n^\top \mu - \bar{A}_{n+1} = \bar{B}_n^\top C\lambda_0 - \tfrac{1}{2}\bar{B}_n^\top CC^\top \bar{B}_n + \delta_0$$

For the z_t coefficient, (125.11) gives $\bar{B}_{n+1}^\top = \bar{B}_n^\top(\phi - C\lambda_z) - \delta_1^\top$, so

$$\bar{B}_n^\top \phi - \bar{B}_{n+1}^\top = \bar{B}_n^\top C\lambda_z + \delta_1^\top$$

Combining the pieces:

$$\log R_{t+1}^{(n+1)} = \underbrace{(\delta_0 + \delta_1^\top z_t)}_{r_t} + \bar{B}_n^\top C \underbrace{(\lambda_0 + \lambda_z z_t)}_{\lambda_t} - \tfrac{1}{2}\bar{B}_n^\top C C^\top \bar{B}_n + \bar{B}_n^\top C\,\varepsilon_{t+1}$$

Writing $\alpha_n = C^\top \bar{B}_n$, this takes the generic return form (125.5):

$$\log R_{t+1}^{(n+1)} = \underbrace{(r_t + \alpha_n^\top \lambda_t)}_{\nu_t} - \tfrac{1}{2}\alpha_n^\top \alpha_n + \alpha_n^\top \varepsilon_{t+1}$$

Since $\mathbb{E}_t R_{t+1}^{(n+1)} = \exp(\nu_t)$, we obtain

$$\log \mathbb{E}_t R_{t+1}^{(n+1)} - r_t = \alpha_n^\top \lambda_t = \bar{B}_n^\top C \lambda_t$$

The following figure plots term premiums across maturities for different states of the world, as well as the level and slope factor contributions to the term premium in the normal state

```
def term_premiums(model, z, n_max):
    """Compute term premiums for maturities 1 to n_max."""
    A_bar, B_bar = bond_coefficients(model, n_max + 1)
    λ_t = risk_prices(model, z)
    return np.array([B_bar[n-1] @ model.C @ λ_t
                     for n in range(1, n_max + 1)])

n_max_tp = 60
maturities_tp = np.arange(1, n_max_tp + 1)

z_states_tp = {
    "Low rate ($z_1 < 0$)":  np.array([-3.0, 2.0]),
    "High rate ($z_1 > 0$)": np.array([3.0, -2.0]),
}

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 5.5))

for label, z in z_states_tp.items():
    tp = term_premiums(model_2f, z, n_max_tp) * 4 * 100
    r_now = short_rate(model_2f, z) * 4 * 100
    lam = risk_prices(model_2f, z)
    ax1.plot(maturities_tp, tp, lw=2.2,
             label=(f"{label}\n  r={r_now:.1f}%,"
                    f" λ=[{lam[0]:.3f}, {lam[1]:.3f}]"))

ax1.axhline(0, color="black", lw=0.8, ls="--")
ax1.set_xlabel("Maturity (quarters)")
ax1.set_ylabel("Term premium (% p.a.)")
ax1.legend(fontsize=9)
ax1.set_xlim(1, n_max_tp)

z_decomp = np.array([0.0, 0.0])
A_bar_d, B_bar_d = bond_coefficients(model_2f, n_max_tp + 1)
λ_t = risk_prices(model_2f, z_decomp)
```

(continues on next page)

(continued from previous page)

```
C_lam = model_2f.C @ λ_t

tp_level = np.array([B_bar_d[n-1, 0] * C_lam[0]
                     for n in range(1, n_max_tp + 1)]) * 4 * 100
tp_slope = np.array([B_bar_d[n-1, 1] * C_lam[1]
                     for n in range(1, n_max_tp + 1)]) * 4 * 100
tp_total = tp_level + tp_slope

ax2.plot(maturities_tp, tp_total, 'k-', lw=2.2, label="Total")
ax2.plot(maturities_tp, tp_level, lw=1.8, ls="--",
         label="Level factor")
ax2.plot(maturities_tp, tp_slope, lw=1.8, ls="--",
         label="Slope factor")
ax2.axhline(0, color="black", lw=0.6, ls=":")
ax2.set_xlabel("Maturity (quarters)")
ax2.set_ylabel("Term premium (% p.a.)")
ax2.legend(fontsize=10)
ax2.set_xlim(1, n_max_tp)

for ax in (ax1, ax2):
    ax_top = ax.twiny()
    ax_top.set_xlim(ax.get_xlim())
    year_ticks = [4, 12, 20, 40, 60]
    ax_top.set_xticks(year_ticks)
    ax_top.set_xticklabels([f"{t/4:.0f}y" for t in year_ticks])

plt.tight_layout()
plt.show()
```

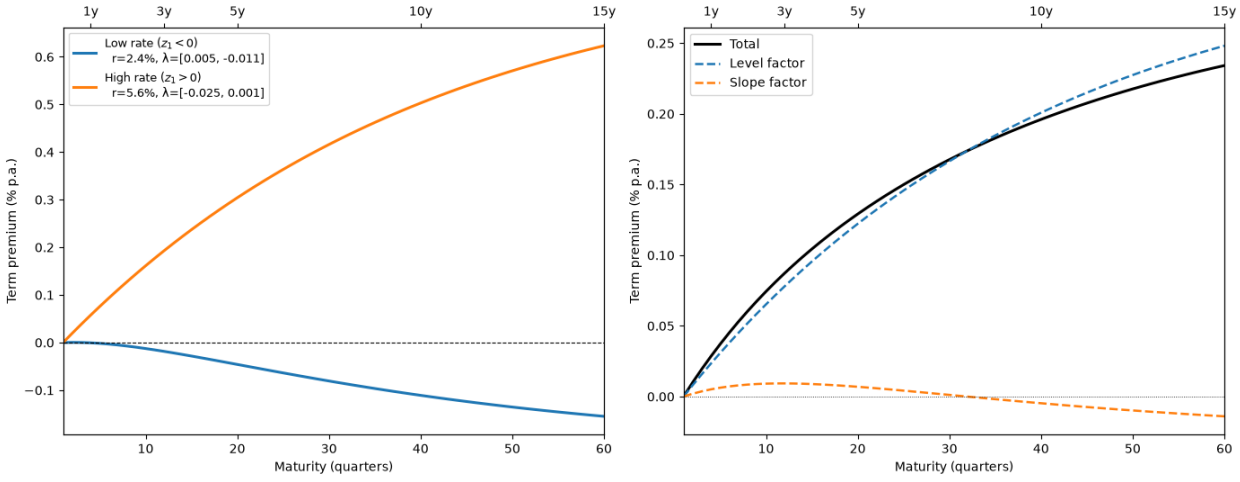


Fig. 125.4: Term premiums and factor decomposition under the two-factor model

We see that the term premium is positive at all maturities in the low-rate state, but becomes negative at longer maturities in the high-rate state.

125.6 Risk-neutral probabilities

We return to the VAR and short-rate equations (125.1)–(125.2), which for convenience we repeat here:

$$z_{t+1} = \mu + \phi z_t + C\varepsilon_{t+1}, \qquad r_t = \delta_0 + \delta_1^\top z_t$$

where $\varepsilon_{t+1} \sim \mathcal{N}(0, I)$.

We suppose that this structure describes the data-generating mechanism.

Finance economists call this the **physical measure** P, to distinguish it from the **risk-neutral measure** Q that we now describe.

Under the physical measure, the conditional distribution of z_{t+1} given z_t is $\mathcal{N}(\mu + \phi z_t,\ CC^\top)$.

125.6.1 Change of measure

With the risk-price vector $\lambda_t = \lambda_0 + \lambda_z z_t$ from (125.3), define the non-negative random variable

$$\frac{\xi_{t+1}^Q}{\xi_t^Q} = \exp\left(-\tfrac{1}{2}\lambda_t^\top \lambda_t - \lambda_t^\top \varepsilon_{t+1}\right) \tag{125.14}$$

This is a log-normal random variable with mean 1, so it is a valid likelihood ratio that can be used to twist the conditional distribution of z_{t+1}.

Multiplying the physical conditional distribution by this likelihood ratio transforms it into the **risk-neutral conditional distribution**

$$z_{t+1} \mid z_t \overset{Q}{\sim} \mathcal{N}(\mu - C\lambda_0 + (\phi - C\lambda_z)z_t,\ CC^\top)$$

In other words, under Q the state follows

$$z_{t+1} = (\mu - C\lambda_0) + (\phi - C\lambda_z)\, z_t + C\varepsilon_{t+1}^Q$$

where $\varepsilon_{t+1}^Q \sim \mathcal{N}(0, I)$ under Q.

The risk-neutral distribution twists the conditional mean from $\mu + \phi z_t$ to $\mu - C\lambda_0 + (\phi - C\lambda_z)z_t$.

The adjustments $-C\lambda_0$ (constant) and $-C\lambda_z$ (state-dependent) encode how the pricing equation $\mathbb{E}_t^P m_{t+1} R_{j,t+1} = 1$ adjusts expected returns for exposure to the risks ε_{t+1}.

125.6.2 Asset pricing in a nutshell

Let $\mathbb{E}^P$ denote an expectation under the physical measure that nature uses to generate the data.

Our key asset pricing equation is $\mathbb{E}_t^P m_{t+1} R_{j,t+1} = 1$ for all returns $R_{j,t+1}$.

Using (125.14), we can express the SDF (125.4) as

$$m_{t+1} = \frac{\xi_{t+1}^Q}{\xi_t^Q}\ \exp(-r_t)$$

Then the condition $\mathbb{E}_t^P\left(\exp(-r_t)\ \frac{\xi_{t+1}^Q}{\xi_t^Q}\ R_{j,t+1}\right) = 1$ is equivalent to

$$\mathbb{E}_t^Q R_{j,t+1} = \exp(r_t) \tag{125.15}$$

Under the risk-neutral measure, expected returns on all assets equal the risk-free return.

125.6.3 Verification via risk-neutral pricing

Bond prices can be computed by discounting at r_t under Q:

$$p_t(n) = \mathbb{E}_t^Q \left[\exp\left(-\sum_{s=0}^{n-1} r_{t+s} \right) \right]$$

We can verify that this agrees with (125.9) by iterating the affine recursion under the risk-neutral VAR.

Below we confirm this numerically

```
def bond_price_mc_Q(model, z0, n, n_sims=50_000, rng=None):
    """Estimate p_t(n) by Monte Carlo under Q."""
    if rng is None:
        rng = np.random.default_rng(0)
    m = len(z0)
    Z = np.tile(z0, (n_sims, 1))
    disc = np.zeros(n_sims)
    for _ in range(n):
        disc += model.δ_0 + Z @ model.δ_1
        ε = rng.standard_normal((n_sims, m))
        Z = model.μ_rn + Z @ model.ψ_rn.T + ε @ model.C.T
    return np.mean(np.exp(-disc))

z_test = np.array([0.01, 0.005])
p_analytic = bond_prices(model_2f, z_test, 40)

rng = np.random.default_rng(0)
maturities_check = [4, 12, 24, 40]
mc_prices = [bond_price_mc_Q(model_2f, z_test, n, n_sims=100_000, rng=rng)
             for n in maturities_check]

header = (f"{'Maturity':>10}  {'Analytic':>12}"
          f"  {'Monte Carlo':>12}  {'Error (bps)':>12}")
print(header)
print("-" * 52)
for n, mc in zip(maturities_check, mc_prices):
    analytic = p_analytic[n - 1]
    error_bp = abs(analytic - mc) / analytic * 10_000
    print(f"{n:>10}  {analytic:>12.6f}  {mc:>12.6f}  {error_bp:>12.2f}")
```

```
  Maturity      Analytic   Monte Carlo   Error (bps)
----------------------------------------------------
         4      0.960592      0.960571          0.23
        12      0.886276      0.886200          0.86
        24      0.787066      0.787580          6.53
        40      0.676314      0.676398          1.24
```

The analytical and Monte Carlo bond prices agree closely, validating the Riccati recursion (125.10)–(125.11).

125.7 Distorted beliefs

Piazzesi *et al.* [2015] assemble survey evidence suggesting that economic experts' forecasts are systematically biased relative to the physical measure.

125.7.1 The subjective measure

Let $\{z_t\}_{t=1}^T$ be a record of observations on the state and let $\{\check{z}_{t+1}\}_{t=1}^T$ be a record of one-period-ahead expert forecasts.

Let $\check{\mu}, \check{\phi}$ be the regression coefficients in

$$\check{z}_{t+1} = \check{\mu} + \check{\phi}\, z_t + e_{t+1}$$

where the residual e_{t+1} has mean zero, is orthogonal to z_t, and satisfies $\mathbb{E}\, e_{t+1} e_{t+1}^\top = CC^\top$.

By comparing estimates of μ, ϕ from (125.1) with estimates of $\check{\mu}, \check{\phi}$ from the experts' forecasts, Piazzesi *et al.* [2015] deduce that the experts' beliefs are systematically distorted.

To organize this evidence, let $\kappa_t = \kappa_0 + \kappa_z z_t$ and define the likelihood ratio

$$\frac{\xi_{t+1}^S}{\xi_t^S} = \exp\left(-\tfrac{1}{2}\kappa_t^\top \kappa_t - \kappa_t^\top \varepsilon_{t+1}\right) \tag{125.16}$$

This is log-normal with mean 1, so it is a valid likelihood ratio.

Multiplying the physical conditional distribution of z_{t+1} by this likelihood ratio transforms it to the experts' **subjective conditional distribution**

$$z_{t+1} \mid z_t \overset{S}{\sim} \mathcal{N}(\mu - C\kappa_0 + (\phi - C\kappa_z)\, z_t,\ CC^\top)$$

In the experts' forecast regression, $\check{\mu}$ estimates $\mu - C\kappa_0$ and $\check{\phi}$ estimates $\phi - C\kappa_z$.

Piazzesi *et al.* [2015] find that the experts behave as if the level and slope of the yield curve are more persistent than under the physical measure: $\check{\phi}$ has larger eigenvalues than ϕ.

125.7.2 Pricing under distorted beliefs

Piazzesi *et al.* [2015] explore the hypothesis that a representative agent with these distorted beliefs prices assets and makes returns satisfy

$$\mathbb{E}_t^S(m_{t+1}^\star\, R_{j,t+1}) = 1$$

where $\mathbb{E}_t^S$ is the conditional expectation under the subjective S measure and $m_{t+1}^\star$ is the SDF of an agent with these beliefs.

In particular, the agent's SDF is

$$m_{t+1}^\star = \exp\left(-r_t^\star - \tfrac{1}{2}\lambda_t^{\star\top}\lambda_t^\star - \lambda_t^{\star\top}\varepsilon_{t+1}\right)$$

where $r_t^\star$ is the short rate and $\lambda_t^\star$ is the agent's vector of risk prices.

Using (125.16) to convert to the physical measure, the subjective pricing equation becomes

$$\mathbb{E}_t^P\left[\exp\left(-r_t^\star - \tfrac{1}{2}\lambda_t^{\star\top}\lambda_t^\star - \lambda_t^{\star\top}\varepsilon_{t+1}\right)\exp\left(-\tfrac{1}{2}\kappa_t^\top\kappa_t - \kappa_t^\top\varepsilon_{t+1}\right) R_{j,t+1}\right] = 1$$

Combining the two exponentials gives

$$\mathbb{E}_t^P\left[\exp\left(-r_t - \tfrac{1}{2}(\lambda_t^\star + \kappa_t)^\top(\lambda_t^\star + \kappa_t) - (\lambda_t^\star + \kappa_t)^\top \varepsilon_{t+1}\right) R_{j,t+1}\right] = 1$$

where $r_t = r_t^\star - \lambda_t^{\star\top}\kappa_t$.

Comparing this with the rational-expectations econometrician's pricing equation

$$\mathbb{E}_t^P\left[\exp\left(-r_t - \tfrac{1}{2}\lambda_t^\top\lambda_t - \lambda_t^\top\varepsilon_{t+1}\right) R_{j,t+1}\right] = 1$$

we see that what the econometrician interprets as λ_t is actually $\lambda_t^\star + \kappa_t$.

Because the econometrician's estimates partly reflect systematic distortions in subjective beliefs, they can overstate the representative agent's true risk prices $\lambda_t^\star$ in this calibration.

Below we construct a numerical example to illustrate this point.

We keep the same physical state dynamics and short-rate specification as above, but choose a separate true risk-price process $(\lambda_t^\star)$ and a distorted-belief econometrician process $(\widehat{\lambda}_t)$ to illustrate the decomposition.

We then set the subjective parameters $\breve{\mu}, \breve{\phi}$ to match the evidence in Piazzesi *et al.* [2015] that experts behave as if the level and slope of the yield curve are more persistent than under the physical measure.

In particular, we use

$$\breve{\phi} = \begin{pmatrix} 0.985 & -0.025 \\ 0.00 & 0.94 \end{pmatrix}$$

```
ψ_P = ψ_2.copy()
μ_P = μ_2.copy()

# Subjective parameters: experts believe factors are MORE persistent
ψ_S = np.array([[0.985, -0.025], [0.00, 0.94]])
μ_S = np.array([0.005, 0.0])

κ_z = np.linalg.solve(C_2, ψ_P - ψ_S)
κ_0 = np.linalg.solve(C_2, μ_P - μ_S)

λ_star_0 = np.array([-0.03, -0.015])
λ_star_z = np.array([[-0.006, 0.0], [0.0, -0.004]])

λ_hat_0 = λ_star_0 + κ_0
λ_hat_z = λ_star_z + κ_z
```

```
model_true = create_affine_model(
    μ_2, ψ_2, C_2, δ_0_2, δ_1_2, λ_star_0, λ_star_z)
model_econ = create_affine_model(
    μ_2, ψ_2, C_2, δ_0_2, δ_1_2, λ_hat_0, λ_hat_z)

z_ref = np.array([0.0, 0.0])
n_max_db = 60
maturities_db = np.arange(1, n_max_db + 1)

tp_true = term_premiums(model_true, z_ref, n_max_db) * 4 * 100
tp_econ = term_premiums(model_econ, z_ref, n_max_db) * 4 * 100

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 5.5))

ax1.plot(maturities_db, tp_true, lw=2.2,
```

(continues on next page)

(continued from previous page)

```
         label=r"True risk prices $\lambda^\star_t$")
line_econ, = ax1.plot(maturities_db, tp_econ, lw=2.2, ls="--",
         label=(r"RE econometrician"
                r" $\hat\lambda_t = \lambda^\star_t + \kappa_t$"))
ax1.fill_between(maturities_db, tp_true, tp_econ,
                 alpha=0.15, color=line_econ.get_color(),
                 label="Belief distortion component")
ax1.axhline(0, color="black", lw=0.8, ls=":")
ax1.set_xlabel("Maturity (quarters)")
ax1.set_ylabel("Term premium (% p.a.)")
ax1.legend(fontsize=9.5)
ax1.set_xlim(1, n_max_db)

mask = np.abs(tp_true) > 1e-8
ratio = np.full_like(tp_true, np.nan)
ratio[mask] = tp_econ[mask] / tp_true[mask]

ax2.plot(maturities_db[mask], ratio[mask], lw=2.2)
ax2.axhline(1, color="black", lw=0.8, ls="--",
            label="No distortion (ratio = 1)")
ax2.set_xlabel("Maturity (quarters)")
ax2.set_ylabel(r"$\hat{tp}\, /\, tp^\star$")
ax2.legend(fontsize=11)
ax2.set_xlim(1, n_max_db)

for ax in (ax1, ax2):
    ax_top = ax.twiny()
    ax_top.set_xlim(ax.get_xlim())
    year_ticks = [4, 12, 20, 40, 60]
    ax_top.set_xticks(year_ticks)
    ax_top.set_xticklabels([f"{t/4:.0f}y" for t in year_ticks])

plt.tight_layout()
plt.show()
```

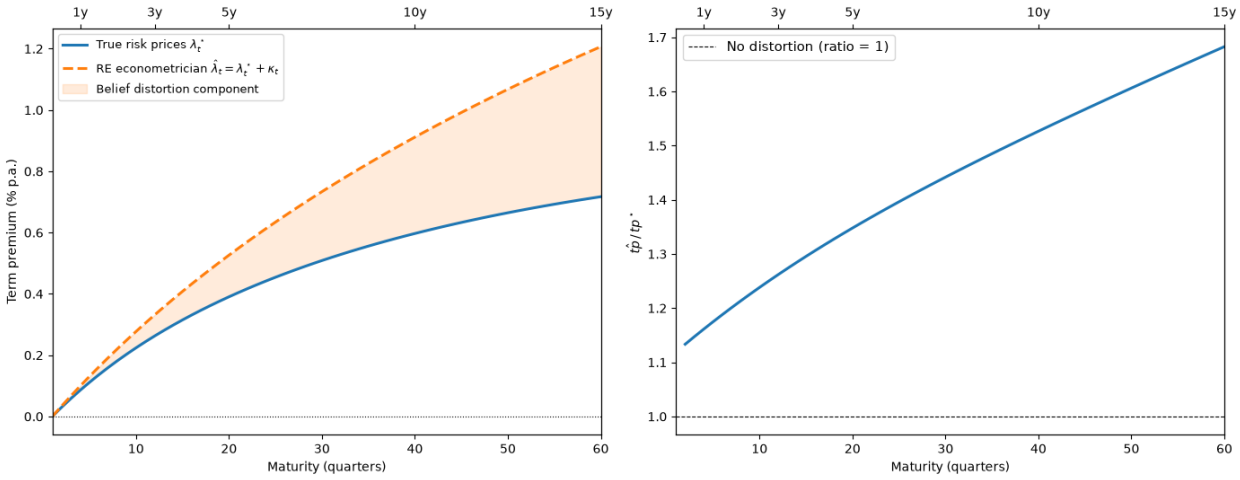


Fig. 125.5: True vs. distorted-belief term premiums and overstatement ratio

When expert beliefs are overly persistent ($\check{\phi}$ has larger eigenvalues than ϕ), the rational-expectations econometrician

attributes too much of the observed risk premium to risk aversion.

Disentangling belief distortions from genuine risk prices requires additional data, for example, the survey forecasts used by Piazzesi *et al.* [2015].

Our Risk Aversion or Mistaken Beliefs? lecture explores this confounding in greater depth.

125.8 Concluding remarks

The affine model of the stochastic discount factor provides a flexible and tractable framework for studying asset prices.

Key features are:

1. **Analytical tractability:** Bond prices are exponential affine in z_t; expected returns decompose cleanly into a short rate plus a risk-price×exposure inner product.
2. **Empirical flexibility:** The free parameters $(\mu, \phi, C, \delta_0, \delta_1, \lambda_0, \lambda_z)$ can be estimated by maximum likelihood (the *Kalman filter* chapter describes the relevant methods) without imposing restrictions from a full general equilibrium model.
3. **Multiple risks:** The vector structure accommodates many sources of risk (monetary policy, real activity, volatility, etc.).
4. **Belief distortions:** The framework naturally accommodates non-rational beliefs via likelihood-ratio twists of the physical measure, as in Piazzesi *et al.* [2015].

The model also connects directly to the Hansen–Jagannathan bounds studied in Doubts or Variability? and to robust control interpretations of the stochastic discount factor described in other chapters of Ljungqvist and Sargent [2018].

For finite-state approaches to asset pricing that complement the continuous-state framework here, see *Asset Pricing: Finite State Models*.

CHAPTER

SIX

THE RECOVERY THEOREM

Contents

- *The Recovery Theorem*
  - *Overview*
  - *Model setup*
  - *The recovery theorem*
  - *Numerical example*
  - *Natural vs. risk-neutral distributions*
  - *Effect of risk aversion*
  - *Recovering the discount rate*
  - *Tail risk: natural vs. risk-neutral probabilities of catastrophe*
  - *From option prices to transition prices*
  - *Testing efficient markets*
  - *Limitations and extensions*
  - *Exercises*

126.1 Overview

Asset prices are forward-looking: they encode investors' expectations about future economic states and their valuations of different risks.

A long-standing question in finance is whether one can *recover* the probability distribution used by investors – their subjective beliefs – from observed asset prices alone.

Option prices reveal **state prices**; once these are normalized by the riskless discount factor, the resulting probabilities are the **risk-neutral probabilities** implied by asset prices after risk adjustments have been folded in.

These are not the **natural probabilities** that investors actually assign to future states of the world.

The two differ because risk-neutral probabilities blend together two distinct objects: the market's true beliefs about the future, and investors' aversion to risk.

The link between them is the **pricing kernel**, which reweights natural probabilities to deliver state prices.

Separating beliefs from risk aversion has traditionally required parametric assumptions about the preferences of a representative investor.

Ross [2015] showed otherwise.

Ross's theorem says that, in a finite-state Markov economy, state prices can be enough.

Suppose the Arrow–Debreu state-price transition matrix is arbitrage-free and irreducible.

If the pricing kernel also satisfies a structural restriction called **transition independence**, then state prices uniquely determine both the natural probability transition matrix and the transition pricing kernel.

No historical return data or assumed utility function is needed if some assumptions about the structure of the pricing kernel hold.

This is the **Recovery Theorem**.

It has several important implications:

- It shows how state-price transition data can identify the market's forward-looking natural distribution when the assumption holds
- It provides tests of the efficient market hypothesis.
- It sheds light on the "dark matter" of finance: the probability of rare catastrophic events embedded in market prices.

This lecture covers

- the Arrow–Debreu framework linking state prices, risk-neutral probabilities, the pricing kernel, and natural probabilities,
- Ross's Recovery Theorem and its proof via the Perron–Frobenius theorem,
- an implementation that recovers the natural distribution from a simulated state-price matrix, and
- how option prices and forward equations can be used to estimate transition state prices,
- comparisons between risk-neutral and recovered natural densities.

Let's import the packages we'll need.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.linalg import eig
from scipy.stats import norm
import matplotlib.cm as cm
```

126.2 Model setup

126.2.1 Arrow–Debreu state prices

Consider a discrete-time, discrete-state economy.

At each date the economy occupies one of m states $\theta_1, \ldots, \theta_m$.

An **Arrow–Debreu security** pays \$1 if the economy is in state θ_j next period and nothing otherwise.

Denote by $p(\theta_i, \theta_j)$ the price today, when the current state is θ_i, of the Arrow–Debreu security paying in state θ_j next period.

Collect these into an $m \times m$ **state price transition matrix**

$$P = [p(\theta_i, \theta_j)]_{i,j=1}^{m}.$$

As in *Competitive Equilibria with Arrow Securities*, the row sums give the state-dependent riskless discount factor: $\sum_j p(\theta_i, \theta_j) = e^{-r(\theta_i)}$.

Here $r(\theta_i)$ is the one-period continuously compounded riskless rate in current state θ_i.

More generally, if an asset pays $g(\theta_j)$ next period, then its price in state θ_i is

$$p_g(\theta_i) = \sum_j p(\theta_i, \theta_j) g(\theta_j).$$

Let

$$b(\theta_i) \equiv \sum_j p(\theta_i, \theta_j) = e^{-r(\theta_i)}$$

be the price of a one-period riskless bond in state θ_i.

Normalizing Arrow prices by this bond price gives the **risk-neutral transition probabilities**

$$q^*(\theta_i, \theta_j) = \frac{p(\theta_i, \theta_j)}{b(\theta_i)} = e^{r(\theta_i)} p(\theta_i, \theta_j).$$

Thus the same asset price can be written as

$$p_g(\theta_i) = b(\theta_i) \sum_j q^*(\theta_i, \theta_j) g(\theta_j) = e^{-r(\theta_i)} E_i^*[g(\theta_{t+1})].$$

Here E_i^* denotes conditional expectation under $q^*(\theta_i, \cdot)$.

The asterisk marks the risk-neutral, or martingale, probability measure.

It is useful to separate this one-period normalization from the dynamic transition structure.

If $Q(\theta_i, \theta_j, T)$ denotes the risk-neutral probability of moving from θ_i to θ_j over T periods, and $0 < t < T$ is an intermediate horizon, then the Markov forward equation is

$$Q(\theta_i, \theta_j, T) = \sum_k Q(\theta_i, \theta_k, t) Q(\theta_k, \theta_j, T - t).$$

In matrix notation, multiperiod risk-neutral transition matrices compose by matrix multiplication.

126.2.2 The pricing kernel

Using the stochastic-discount-factor notation studied in *Asset Pricing: Finite State Models* and the Arrow-security notation used in *Competitive Equilibria with Arrow Securities*, the pricing kernel $\phi(\theta_i, \theta_j)$ relates state prices to natural probabilities via

$$p(\theta_i, \theta_j) = \phi(\theta_i, \theta_j)\, f(\theta_i, \theta_j),$$

where $f(\theta_i, \theta_j)$ is the natural (conditional) probability of transitioning from state θ_i to θ_j.

As in the representative-agent equilibrium calculation in *Competitive Equilibria with Arrow Securities*, the canonical additively separable model with discount factor β gives

$$\phi(\theta_i, \theta_j) = \frac{p(\theta_i, \theta_j)}{f(\theta_i, \theta_j)} = \frac{\beta U'(c(\theta_j))}{U'(c(\theta_i))}. \tag{126.1}$$

This formula has a special structure: the kernel can be written as a ratio of two state-specific terms.

Ross calls this property **transition independence**.

We will say more about it soon.

126.2.3 The identification challenge

Before stating the restriction, it helps to see why one is needed at all.

Given P, any pair (ϕ, f) satisfying $p_{ij} = \phi_{ij} f_{ij}$ for every (i, j) is consistent with observed state prices.

The state-price matrix P supplies m^2 equations.

A natural transition matrix F contributes $m(m-1)$ free entries (rows sum to one), and an arbitrary kernel ϕ contributes another m^2 – a total of $2m^2 - m$ unknowns against only m^2 equations.

The system is under-identified by $m^2 - m$ parameters, so some structural restriction on the kernel is needed to pin down ϕ and f separately.

Transition independence restriction does the job, as we will see in the next section.

126.2.4 Transition independence

Definition (Transition Independence)

A pricing kernel is **transition independent** if there exists a positive function h on the state space and a positive scalar β such that for every transition from state θ_i to θ_j,

$$\phi(\theta_i, \theta_j) = \beta \frac{h(\theta_j)}{h(\theta_i)}.$$

Transition independence says the kernel depends on the *ending* state and normalizes by the *beginning* state.

In the representative-agent complete-markets environment above, it holds under intertemporally additive separable utility (where $h = U'$).

In particular, this holds for (126.1).

Transition independence helps because it ties all m^2 entries of ϕ together: once the m state-specific values are known, the whole kernel is pinned down.

It therefore cuts ϕ from m^2 free entries down to m, so the system becomes exactly identified.

Under transition independence, the state-price equation becomes

$$p(\theta_i, \theta_j) = \beta \frac{h(\theta_j)}{h(\theta_i)} f(\theta_i, \theta_j).$$

In matrix notation, defining the diagonal matrix D with $D_{ii} = h(\theta_i)/\beta$,

$$DP = \beta F D,$$

or equivalently,

$$F = \frac{1}{\beta} D P D^{-1}.$$

126.3 The recovery theorem

126.3.1 Reduction to an eigenvalue problem

Since F is a stochastic matrix, its rows sum to one: $Fe = e$ where e is the vector of ones.

Substituting the expression for F:

$$\frac{1}{\beta} DPD^{-1}e = e \quad \Longrightarrow \quad Pz = \beta z, \quad z \equiv D^{-1}e.$$

This is an **eigenvalue problem** where we seek a positive vector z and scalar β satisfying $Pz = \beta z$.

In principle every eigenvalue-eigenvector pair of P is a formal solution, but only the one with a strictly positive eigenvector is economically valid: $D_{ii} = 1/z_i$ must be positive (so $z_i > 0$), and F must have nonnegative entries.

The Perron–Frobenius theorem guarantees that exactly one such pair exists.

Theorem (Perron–Frobenius)

If A is a nonnegative irreducible matrix, then

1. A has a positive real eigenvalue r equal to its spectral radius (the Perron root).
2. There exists a strictly positive eigenvector $z \gg 0$ with $Az = rz$, unique up to scaling.
3. No other eigenvector is strictly positive.

Other eigenvalues can have the same modulus when the matrix is imprimitive, but the strictly positive eigenvector is unique up to scale.

See Section 1.2.3 of Sargent and Stachurski [2024] for details.

See also the full statement in The Perron-Frobenius Theorem.

Applied to the recovery problem: the Perron root is β (the subjective discount factor) and the Perron vector z determines D via $D_{ii} = 1/z_i$.

126.3.2 Ross's recovery theorem

The three assumptions in the theorem each carry a specific role.

Assuming the Arrow–Debreu state prices are identified, no-arbitrage guarantees that P has nonnegative entries and that the state prices encode a well-defined pricing measure.

Irreducibility ensures the economy is not divided into disconnected sub-economies – without it, the Perron–Frobenius theorem gives multiple candidate eigenvectors and recovery breaks down.

Transition independence is the key economic restriction.

It says the pricing kernel factors as $\beta h(\theta_j)/h(\theta_i)$, so the entire kernel is pinned down by a single vector h (or equivalently z).

With these in mind, the Recovery Theorem follows from the Perron–Frobenius theorem.

Theorem (Recovery Theorem)

Suppose prices provide no arbitrage opportunities, that the state price transition matrix P is irreducible, and that the pricing kernel is transition independent.

Then there exists a positive solution (β, z, F) to the recovery problem in which z is unique up to normalization, and the implied natural probability transition matrix F and transition pricing kernel are unique.

Proof. Because P is nonnegative and irreducible, the Perron–Frobenius theorem gives a unique positive eigenvector $z \gg 0$ with positive eigenvalue $\lambda > 0$ satisfying $Pz = \lambda z$.

Setting

$$\beta = \lambda, \qquad D_{ii} = \frac{1}{z_i},$$

the natural probability transition matrix is uniquely recovered as

$$f_{ij} = \frac{1}{\beta} \frac{z_j}{z_i} p_{ij}.$$

To confirm F is stochastic, note that all entries are nonnegative (since $p_{ij} \geq 0$ and $z_i, z_j > 0$) and

$$\sum_j f_{ij} = \frac{1}{\beta z_i} \sum_j z_j \, p_{ij} = \frac{[Pz]_i}{\beta z_i} = \frac{\beta z_i}{\beta z_i} = 1.$$

Uniqueness follows from the uniqueness of the Perron–Frobenius eigenvector.

126.3.3 Pricing kernel from the eigenvector

The recovered transition-kernel values are

$$\phi(\theta_i, \theta_j) = \beta \frac{z_i}{z_j}, \qquad h(\theta_i) = \frac{\beta}{z_i},$$

where $h(\theta_i) = \beta / z_i$ follows from $D_{ii} = h(\theta_i)/\beta = 1/z_i$.

It is useful to distinguish the **full transition kernel** $\phi_{ij} = \beta z_i / z_j$, which depends on both origin and destination states, from the **relative kernel component** $1/z_j$, which depends only on the destination state.

Ross's Table I reports the destination-state shape $1/z_j$, normalized so that the middle state equals one.

Destination states with high z_j have *low* kernel values: for a fixed origin i, the kernel $\beta z_i / z_j$ is decreasing in z_j.

When h represents marginal utility and states are ordered by consumption or payoff, larger z_j corresponds to lower marginal utility – "good times" that require less insurance and so receive less pricing weight per unit of natural probability.

The same eigenvector argument also yields a useful limiting case.

If the one-period bond price is identical in every current state, then the vector of ones is already the Perron vector, so recovery has no state-dependent change of measure left to perform.

Corollary

If the riskless rate is the same in all states ($Pe = be$ for some scalar b), then the unique natural distribution consistent with recovery is the risk-neutral (martingale) distribution itself: $F = (1/b)P$.

Proof. When $Pe = be$, the vector of ones e is the Perron eigenvector with eigenvalue b.

By the uniqueness part of the Perron–Frobenius theorem, $z = e$ (up to scaling) and $\beta = b$.

Setting $z = e$ gives $D = I$, so

$$F = \frac{1}{\beta} DPD^{-1} = \frac{1}{b} P. \qquad \square$$

126.3.4 Single crossing and the risk premium

Ross also uses the representative-agent formula to compare the natural and risk-neutral densities directly.

For a fixed current state θ_i,

$$\frac{q^*(\theta_i, \theta_j)}{f(\theta_i, \theta_j)} = e^{r(\theta_i)} \phi(\theta_i, \theta_j) = e^{r(\theta_i)} \beta \frac{U'(c(\theta_j))}{U'(c(\theta_i))}. \tag{126.2}$$

If U' is decreasing in consumption, then the ratio in (126.2) is decreasing in next-period consumption $c(\theta_j)$.

Since both $q^*(\theta_i, \cdot)$ and $f(\theta_i, \cdot)$ integrate to one, there is a crossing point v defined by

$$e^{r(\theta_i)} \beta U'(v) = U'(c(\theta_i)).$$

Below v, risk-neutral probability exceeds natural probability; above v, the natural probability exceeds the risk-neutral probability.

Hence the natural consumption distribution first-order stochastically dominates the risk-neutral one.

In a one-period model where terminal consumption is the market payoff, this also gives a positive market risk premium.

Let R denote the market return under the natural law, let R^* denote the same return under the risk-neutral law, and let R_f denote the riskless return in the same one-period units.

The stochastic-dominance result can be represented as

$$R^* \sim R - Z + \epsilon,$$

where $Z \geq 0$ captures the downward shift induced by risk adjustment and ϵ is a residual satisfying $E[\epsilon \mid R - Z] = 0$.

Taking expectations gives

$$E[R] = R_f + E[Z] > R_f.$$

126.4 Numerical example

We now demonstrate the Recovery Theorem numerically.

126.4.1 Building a finite-state example

We build the economy directly on a finite grid of log payoff states $s_1, \dots, s_m$.

On this grid we choose three primitives:

1. a row-stochastic irreducible natural transition matrix F,
2. a subjective discount factor $\beta = e^{-\rho T}$, and
3. a CRRA transition pricing kernel $\phi_{ij} = \beta e^{-\gamma(s_j - s_i)}$.

The state-price matrix is then constructed from

$$p_{ij} = \phi_{ij} f_{ij}.$$

This means the Recovery Theorem assumptions hold by construction: P is nonnegative, F is a Markov transition matrix, and the kernel is transition independent with $z_i \propto e^{\gamma s_i}$.

To keep the example close to Ross's Section IV, we choose F to have lognormal-shaped rows.

The continuous benchmark is a lognormal payoff with CRRA utility:

$$U(S_T) = \frac{S_T^{1-\gamma}}{1-\gamma}, \qquad S_T = S_0 \exp\Big((\mu - \tfrac{1}{2}\sigma^2)T + \sigma\sqrt{T}\xi\Big),$$

where $\xi \sim N(0,1)$, μ is the expected growth-rate parameter, σ is volatility, T is the horizon, γ is the CRRA coefficient, and ρ is the continuously compounded subjective discount rate.

The T-period pricing kernel is

$$\phi_T = e^{-\rho T} \left(\frac{S_T}{S_0}\right)^{-\gamma}.$$

Equivalently, if $s = \log S_0$ and $s_T = \log S_T$, then the state-price density with respect to the future log state s_T is

$$p_T(s, s_T) = e^{-\rho T} e^{-\gamma(s_T - s)} \frac{1}{\sigma\sqrt{T}} n\left(\frac{s_T - s - (\mu - \frac{1}{2}\sigma^2)T}{\sigma\sqrt{T}}\right),$$

where n is the standard normal density.

Thus the natural log return satisfies

$$\log(S_T/S_0) \sim \mathcal{N}\big((\mu - \tfrac{1}{2}\sigma^2)T, \sigma^2 T\big).$$

Following Ross's Table I, we represent the distribution on a finite grid of states.

This example is Ross-inspired rather than an exact reproduction of Ross's Table I.

Ross's Table I uses a fixed future payoff distribution, so its rows of F are identical.

Here the same CRRA/lognormal pricing logic is embedded in a finite Markov transition matrix whose rows shift with the current state.

Ross uses states from -5 to $+5$ standard deviations; we use the same range below.

The truncation is an essential part of the finite-state model: it is what brings the example into the Perron–Frobenius setting.

In the unbounded continuous lognormal growth model, Ross shows that recovery is not unique.

On the finite grid, the natural transition probabilities and state prices are

$$f_{ij} \propto n\left(\frac{s_j - s_i - (\mu - \frac{1}{2}\sigma^2)T}{\sigma\sqrt{T}}\right)\Delta s, \qquad p_{ij} = e^{-\rho T}e^{-\gamma(s_j - s_i)}f_{ij},$$

where $s_i = \ln S_i$, $s_j = \ln S_j$, $n(\cdot)$ is the standard normal density, and the discretized probabilities f_{ij} are normalized row by row.

The next cell constructs this finite grid and builds P.

```
def build_state_price_matrix(μ, σ, γ, ρ, T=1.0, n_states=11, n_σ=5):
    """Build a discretized lognormal/CRRA state-price matrix."""
    states = np.linspace(-n_σ * σ * np.sqrt(T),
                         n_σ * σ * np.sqrt(T),
                         n_states)
    ds = states[1] - states[0]

    m = n_states
    P = np.zeros((m, m))
    F = np.zeros((m, m))

    drift = (μ - 0.5 * σ**2) * T

    # First build a row-stochastic natural transition matrix on the bounded grid
    for i in range(m):
        s_i = states[i]
        for j in range(m):
            s_j = states[j]
            log_return = s_j - s_i
            F[i, j] = norm.pdf(log_return, loc=drift,
                               scale=σ * np.sqrt(T)) * ds

        F[i] = F[i] / F[i].sum()

        # Price each Arrow claim as natural probability times the CRRA kernel
        for j in range(m):
            log_return = states[j] - s_i
            kernel = np.exp(-ρ * T) * np.exp(-γ * log_return)
            P[i, j] = kernel * F[i, j]

    return P, states
```

Now choose a calibration and build the state-price matrix.

```
μ = 0.08    # 8% annual expected return
σ = 0.20    # 20% annual volatility
γ = 3.0     # CRRA coefficient
ρ = 0.02    # 2% annual continuous discount rate
T = 1.0     # one-year horizon

P, states = build_state_price_matrix(μ, σ, γ, ρ, T,
                                     n_states=11, n_σ=5)

print("State-price row sums:")
```

(continues on next page)

(continued from previous page)

```
print(np.round(P.sum(axis=1), 4))
print(f"Middle-state risk-free rate: {-np.log(P[5].sum()):.4f}")
```

```
State-price row sums:
[0.7154 0.9046 0.9713 0.9798 0.9802 0.9802 0.9802 0.9806 0.9892 1.0621
 1.3429]
Middle-state risk-free rate: 0.0200
```

The row sums are the model-implied one-period bond prices in each current state.

They vary near the boundaries because the finite grid truncates and renormalizes the conditional transition probabilities.

126.4.2 Applying the recovery theorem

The Recovery Theorem requires computing the **Perron eigenvector** of P.

```
def recover_natural_distribution(P, tol=1e-10):
    """
    Recover natural probabilities and the relative pricing kernel
    from state prices.
    """

    m = P.shape[0]

    eigenvalues, eigenvectors = eig(P)
    eigenvalues = np.real_if_close(eigenvalues, tol=1000)
    eigenvectors = np.real_if_close(eigenvectors, tol=1000)

    # Ross recovery uses the Perron root and its strictly positive eigenvector
    real_mask = np.isreal(eigenvalues)
    real_eigenvalues = np.asarray(
        eigenvalues[real_mask].real, dtype=float)
    real_eigenvectors = np.asarray(
        eigenvectors[:, real_mask].real, dtype=float)

    order = np.argsort(real_eigenvalues)[::-1]

    for idx in order:
        β_candidate = real_eigenvalues[idx]
        z_candidate = real_eigenvectors[:, idx]

        if np.mean(z_candidate) < 0:
            z_candidate = -z_candidate

        if β_candidate > 0 and np.all(z_candidate > tol):
            β_recovered = β_candidate
            z = z_candidate
            break
    else:
        raise ValueError("No strictly positive real eigenvector found")

    z = z / z[m // 2]

    D = np.diag(1.0 / z)
    D_inv = np.diag(z)
```

(continues on next page)

(continued from previous page)

```
    # Converts state prices into probabilities
    F = (1.0 / β_recovered) * D @ P @ D_inv

    min_entry = F.min()
    row_sum_error = np.max(np.abs(F.sum(axis=1) - 1.0))

    if min_entry < -tol:
        raise ValueError(f"Recovered F has negative entries: min={min_entry}")

    if row_sum_error > 1e-8:
        raise ValueError(
            f"Recovered F row sums are not one: max error={row_sum_error}"
        )

    # The kernel relative to the middle state normalization
    ψ_relative = 1.0 / z

    return F, z, β_recovered, ψ_relative
```

The Perron vector also recovers the shape of the pricing kernel.

Ross's Table I reports this shape with the middle state normalized to one, which is $1/z_j$ under our normalization $z_{\text{mid}} = 1$.

```
F, z, β_rec, ψ_relative = recover_natural_distribution(P)

print("Ross-normalized kernel 1/z (middle state = 1):")
print(np.round(ψ_relative, 4))
```

```
Ross-normalized kernel 1/z (middle state = 1):
[20.0855 11.0232  6.0496  3.3201  1.8221  1.      0.5488  0.3012  0.1653
  0.0907  0.0498]
```

Because we know the data-generating natural transition matrix used to construct P, we can verify that recovery works in this simulation.

```
def true_lognormal_transition_matrix(states, μ, σ, T):
    """
    Construct the bounded-grid natural transition matrix used in the simulation.
    """
    m = len(states)
    ds = states[1] - states[0]
    drift = (μ - 0.5 * σ**2) * T
    F_true = np.zeros((m, m))

    for i in range(m):
        log_returns = states - states[i]
        F_true[i] = norm.pdf(log_returns, loc=drift,
                             scale=σ * np.sqrt(T)) * ds
        F_true[i] = F_true[i] / F_true[i].sum()

    return F_true


F_true = true_lognormal_transition_matrix(states, μ, σ, T)
P_reconstructed = β_rec * (z[:, None] / z[None, :]) * F
```

(continues on next page)

(continued from previous page)

```
print("Recovery numerical checks")
print(f"max |F - true F| = {np.max(np.abs(F - F_true)):.2e}")
print(f"max |P - recovered kernel times F| = "
      f"{np.max(np.abs(P - P_reconstructed)):.2e}")
```

```
Recovery numerical checks
max |F - true F| = 3.46e-14
max |P - recovered kernel times F| = 1.11e-16
```

Indeed, the discrepancies are at the level of numerical roundoff.

126.5 Natural vs. risk-neutral distributions

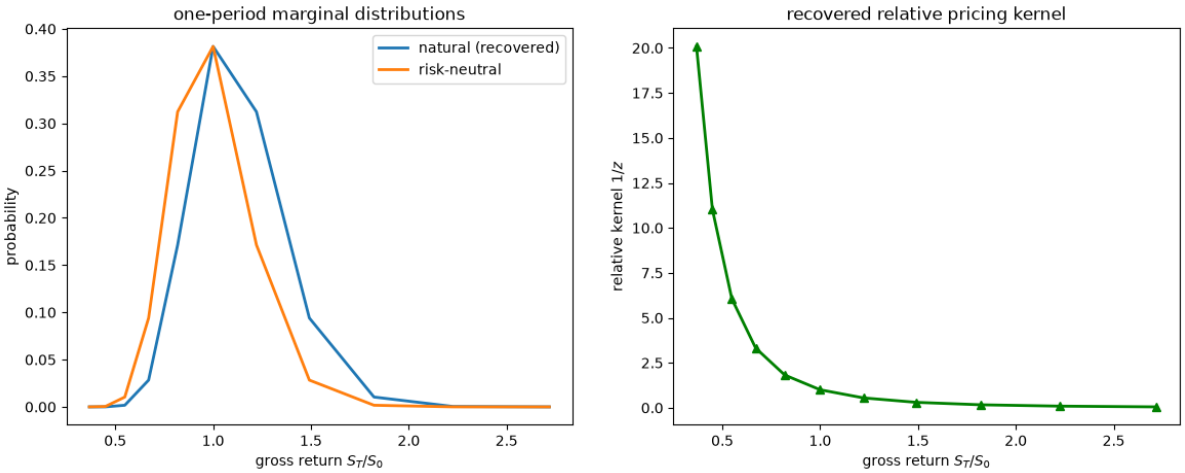
A key insight of Ross [2015] is that the natural distribution can differ systematically from the risk-neutral one.

In this CRRA example, where states are ordered from low to high payoff, the single-crossing argument in *Single crossing and the risk premium* implies that the natural marginal density **first-order stochastically dominates** the risk-neutral density: the CDF of the natural distribution lies *below* that of the risk-neutral distribution.

Because the pricing kernel is declining (investors fear bad outcomes), risk-neutral probabilities overweight bad states and underweight good states relative to the natural measure.

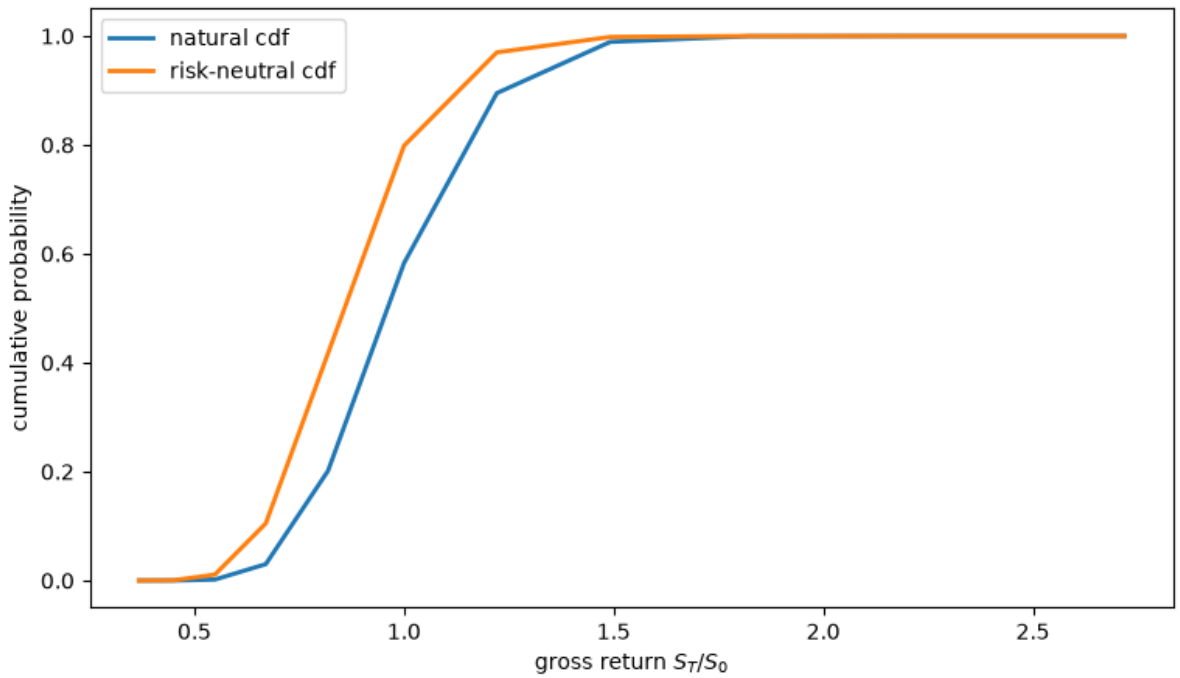
We first plot the natural distribution against the risk-neutral one and the recovered relative pricing kernel

```
mid = len(states) // 2

row_sums = P.sum(axis=1, keepdims=True)

# Normalize Arrow prices by the one-period riskless bond price in each state
Q_rn = P / row_sums

f_nat = F[mid, :]
f_rn = Q_rn[mid, :]

gross_returns = np.exp(states)

fig, axes = plt.subplots(1, 2, figsize=(14, 5))

axes[0].plot(gross_returns, f_nat, label='natural (recovered)', lw=2)
axes[0].plot(gross_returns, f_rn, label='risk-neutral', lw=2)
axes[0].set_xlabel('gross return $S_T / S_0$')
axes[0].set_ylabel('probability')
axes[0].set_title('one-period marginal distributions')
axes[0].legend()

axes[1].plot(gross_returns, ψ_relative, 'g-^', lw=2)
axes[1].set_xlabel('gross return $S_T / S_0$')
axes[1].set_ylabel('relative kernel $1/z$')
axes[1].set_title('recovered relative pricing kernel')
plt.show()
```



The CDF clearly shows the first-order stochastic dominance

```
cdf_nat = np.cumsum(f_nat)
cdf_rn = np.cumsum(f_rn)

fig, ax = plt.subplots(figsize=(9, 5))
ax.plot(gross_returns, cdf_nat, lw=2, label='natural cdf')
ax.plot(gross_returns, cdf_rn, lw=2, label='risk-neutral cdf')
ax.set_xlabel('gross return $S_T / S_0$')
ax.set_ylabel('cumulative probability')
ax.legend()
plt.show()

print(f"Natural CDF <= Risk-neutral CDF at all states: "
      f"{np.all(cdf_nat <= cdf_rn + 1e-10)}")
```



```
Natural CDF <= Risk-neutral CDF at all states: True
```

The gap between the two CDFs is generated by the slope of the pricing kernel.

In the CRRA benchmark, this slope is controlled by the risk-aversion coefficient γ.

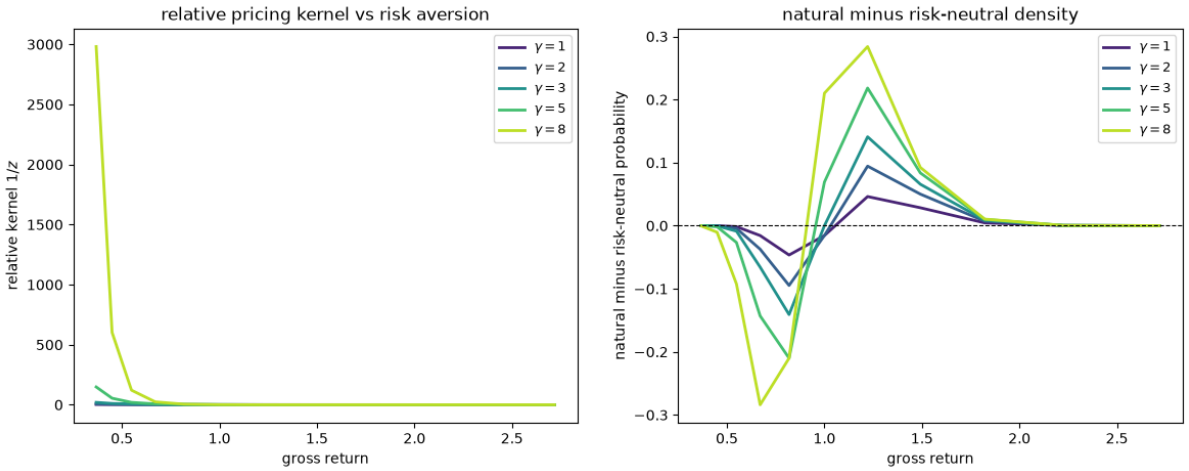
We next vary γ to see how the recovered kernel and the natural/risk-neutral wedge change.

126.6 Effect of risk aversion

The shape of the pricing kernel, and hence the gap between natural and risk-neutral probabilities, depends on the coefficient of risk aversion γ.

We illustrate this by plotting the relative pricing kernel $1/z$ and the gap between the natural and risk-neutral densities for a range of values of γ.

```
γs = [1.0, 2.0, 3.0, 5.0, 8.0]
colors = cm.viridis(np.linspace(0.1, 0.9, len(γs)))

fig, axes = plt.subplots(1, 2, figsize=(14, 5))

for γ_val, color in zip(γs, colors):
    P_g, states_g = build_state_price_matrix(μ, σ, γ_val, ρ, T)
    F_g, z_g, β_g, ψ_relative_g = recover_natural_distribution(P_g)
    mid_g = len(states_g) // 2

    f_nat_g = F_g[mid_g, :]
    row_sum = P_g[mid_g].sum()
    f_rn_g = P_g[mid_g] / row_sum

    gross = np.exp(states_g)

    axes[0].plot(gross, ψ_relative_g, color=color, lw=2,
                 label=f'$\\gamma={γ_val:.0f}$')
    axes[1].plot(gross, f_nat_g - f_rn_g, color=color, lw=2,
                 label=f'$\\gamma={γ_val:.0f}$')

axes[0].set_xlabel('gross return')
axes[0].set_ylabel('relative kernel $1/z$')
axes[0].set_title('relative pricing kernel vs risk aversion')
axes[0].legend(fontsize=9)

axes[1].axhline(0, color='k', lw=0.8, ls='--')
axes[1].set_xlabel('gross return')
axes[1].set_ylabel('natural minus risk-neutral probability')
axes[1].set_title('natural minus risk-neutral density')
axes[1].legend(fontsize=9)

plt.show()
```



Because the states are ordered from low to high payoff, the plots show the single-crossing property discussed in *Single crossing and the risk premium*: for returns below some threshold v, risk-neutral probability exceeds natural probability; above v the natural probability dominates.

A higher γ amplifies this wedge.

126.7 Recovering the discount rate

A useful by-product of the Recovery Theorem is the *recovered subjective discount factor* β, which equals the Perron–Frobenius eigenvalue of P.

If the horizon is T, the corresponding continuously compounded subjective discount rate is

$$\rho = -\frac{\log \beta}{T}.$$

In the numerical examples below, $T = 1$, so this reduces to $\rho = -\log \beta$.

Corollary 1 of Ross [2015] states that β is bounded above by the largest state-dependent one-period discount factor — equivalently, the maximum row sum of P:

$$\beta \leq \max_i \sum_j p(\theta_i, \theta_j).$$

Sweeping the true ρ over a grid and reporting the recovered values alongside the recovery error confirms that the eigenvalue calculation pins down β accurately:

```
true_ρs = np.linspace(0.00, 0.06, 13)
recovered_ρs = np.empty_like(true_ρs)

for k, rho in enumerate(true_ρs):
    P_d, _ = build_state_price_matrix(μ, σ, γ=3.0, ρ=rho, T=1.0)
    _, _, β_d, _ = recover_natural_distribution(P_d)
    recovered_ρs[k] = -np.log(β_d)

print(
    f"max |true ρ - recovered ρ| = {np.max(np.abs(true_ρs - recovered_ρs)):.2e}")
np.column_stack([true_ρs, recovered_ρs])
```

```
max |true ρ - recovered ρ| = 2.11e-15
```

```
array([[ 0.0000000e+00, -4.4408921e-16],
       [ 5.0000000e-03,  5.0000000e-03],
       [ 1.0000000e-02,  1.0000000e-02],
       [ 1.5000000e-02,  1.5000000e-02],
       [ 2.0000000e-02,  2.0000000e-02],
       [ 2.5000000e-02,  2.5000000e-02],
       [ 3.0000000e-02,  3.0000000e-02],
       [ 3.5000000e-02,  3.5000000e-02],
       [ 4.0000000e-02,  4.0000000e-02],
       [ 4.5000000e-02,  4.5000000e-02],
       [ 5.0000000e-02,  5.0000000e-02],
       [ 5.5000000e-02,  5.5000000e-02],
       [ 6.0000000e-02,  6.0000000e-02]])
```

126.8 Tail risk: natural vs. risk-neutral probabilities of catastrophe

One of the most striking applications of the Recovery Theorem is its ability to separate the market's recovered natural probability of catastrophes from the risk premium attached to them.

Barro [2006] and Mehra and Prescott [1985] discuss how rare disasters might explain the equity premium puzzle.

The risk-neutral probability of a large decline is elevated both because (a) the market assigns a high natural probability to such events and (b) the pricing kernel upweights bad outcomes.

Ross's recovery machinery lets us decompose these two forces.

The next cell plots left-tail probabilities under the recovered natural and the risk-neutral measures from the middle state, so the gap between the curves isolates the pricing-kernel contribution to crash probabilities.

```
thresholds = np.linspace(-0.40, 0.10, 200)

def tail_prob(f_dist, states, threshold):
    """Left-tail probability for log returns."""
    return float(np.sum(f_dist[states <= threshold]))

P_base, states_base = build_state_price_matrix(
    μ, σ, γ=3.0, ρ=0.02, T=1.0,
    n_states=41, n_σ=5)
F_base, z_base, β_base, ψ_relative_base = recover_natural_distribution(P_base)

mid_b = len(states_base) // 2
f_nat_base = F_base[mid_b]
f_rn_base = P_base[mid_b] / P_base[mid_b].sum()

prob_nat = [tail_prob(f_nat_base, states_base, t) for t in thresholds]
prob_rn = [tail_prob(f_rn_base, states_base, t) for t in thresholds]

fig, ax = plt.subplots(figsize=(10, 5))
ax.plot(np.exp(thresholds), prob_nat, lw=2, label='natural (recovered)')
ax.plot(np.exp(thresholds), prob_rn, lw=2, label='risk-neutral')
ax.set_xlabel('gross return threshold')
ax.set_ylabel('probability of decline below threshold')
ax.axvline(x=0.75, color='gray', ls=':', lw=1.5, label='25% decline')
```

(continues on next page)

(continued from previous page)

```
ax.axvline(x=0.70, color='silver', ls=':', lw=1.5, label='30% decline')
ax.legend()
plt.show()
```

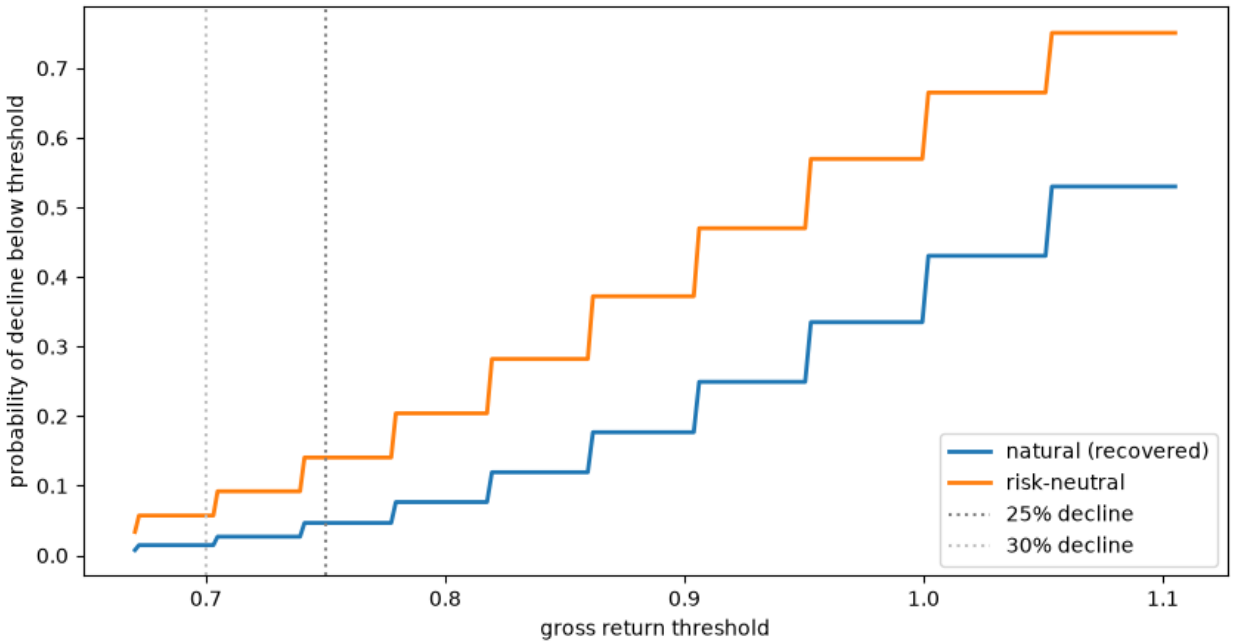


Fig. 126.1: Tail probabilities under the recovered natural and risk-neutral measures

This is a simulation illustrating Ross's decomposition.

The risk-neutral density assigns higher probability to large drops than the recovered natural density.

In this CRRA simulation, increasing risk aversion makes the risk-neutral crash probability rise faster than the recovered natural crash probability.

We will say more in Exercise 126.12.3.

126.9 From option prices to transition prices

The numerical example above starts from a known state-price transition matrix P.

Empirically, Ross starts one step earlier: option prices reveal state-price densities at different maturities from the current state, and the transition matrix must be inferred from those maturity-by-maturity state prices.

Let $C(K,T)$ be the price of a call option with strike K and maturity T.

If $p(S,T)$ is the state-price density for terminal index level S, then

$$C(K,T) = \int_K^\infty (S-K)p(S,T)\,dS.$$

Differentiating twice with respect to the strike gives the Breeden and Litzenberger [1978] formula

$$p(K,T) = \frac{\partial^2 C(K,T)}{\partial K^2}.$$

After discretizing strikes and maturities, let

$$p_t(c) = (p_t(c,1), \ldots, p_t(c,m))$$

be the vector of state prices at horizon t observed from today's state c.

Here c indexes the current state and t counts discrete maturity steps.

The first one-period vector $p_1(c)$ identifies the row of P corresponding to the current state c, supplying m equations.

If the one-period state-price transition matrix P is time homogeneous, these vectors satisfy the forward recursion

$$p_{t+1}(c) = p_t(c)P, \qquad t = 1, \ldots, m-1.$$

Componentwise,

$$p_{t+1}(c,j) = \sum_k p_t(c,k)p(k,j).$$

The remaining $m-1$ forward equations $p_{t+1}(c) = p_t(c)P$, each with m components, supply the remaining $m(m-1)$ equations.

Together these give m^2 equations for the m^2 transition prices $p(k,j)$.

In practice this step is numerically delicate because the second derivative in the option-price formula amplifies measurement error, and because additional shape restrictions such as positivity or unimodality may be needed to obtain a reasonable transition matrix.

126.10 Testing efficient markets

The recovered pricing kernel can also be used to test market efficiency, under the assumptions of the Recovery Theorem.

If a trading strategy has a very high Sharpe ratio, then some pricing kernel must be volatile enough to price that payoff.

The Hansen–Jagannathan bound [Hansen and Jagannathan, 1991] says that, for any excess return with mean μ_{excess} and standard deviation σ_{asset},

$$\frac{|\mu_{\text{excess}}|}{\sigma_{\text{asset}}} \leq e^{rT}\,\sigma(M),$$

where M is the one-period stochastic discount factor and r is the continuously compounded riskless rate over horizon T.

Ross's point is that recovery gives an estimate of the relevant volatility $\sigma(M)$.

Hence it gives an upper bound on the Sharpe ratio of any strategy based on the same stock-market information used in recovery.

If such a strategy has a Sharpe ratio above the bound, then it is too profitable to be consistent with efficiency, under the assumptions of the Recovery Theorem.

The same logic gives a bound on return predictability.

Suppose excess returns are decomposed as

$$x_{t+1} = \mu(I_t) + \epsilon_{t+1},$$

where I_t is the stock-market information set and ϵ_{t+1} is unpredictable from I_t.

Then the R^2 of a forecasting regression based on I_t is bounded above by the variance of the recovered kernel:

$$R^2 \leq e^{2rT}\,\sigma^2(M).$$

Only the component of the kernel projected on this information set is relevant.

Adding unrelated noise to a candidate pricing kernel would raise its variance, but it would not justify stronger return predictability from stock-market information.

126.11 Limitations and extensions

The Recovery Theorem is a remarkable theoretical result, but several caveats apply in practice.

Finite state space:

Ross's theorem is proved for a finite-state irreducible Markov chain; bounded continuous-state recovery requires additional results in *Misspecified Recovery*.

In continuous, unbounded state spaces (e.g., a lognormal diffusion), uniqueness fails because any exponential $e^{\alpha x}$ satisfies the characteristic equation.

To see the issue, consider the continuous lognormal growth state-price density above.

The natural continuous-space analogue of the Perron–Frobenius problem is

$$\int p_T(s,y)v(y)\,dy = \lambda v(s).$$

Here y is a possible future log state, v is a candidate positive eigenfunction, and λ is its eigenvalue.

For every real α, the exponential function $v_\alpha(s) = e^{\alpha s}$ solves this equation with eigenvalue

$$\lambda(\alpha) = \exp\!\left(-\rho T + (\alpha-\gamma)(\mu - \tfrac{1}{2}\sigma^2)T + \tfrac{1}{2}\sigma^2 T(\alpha-\gamma)^2\right).$$

The positive eigenfunction is therefore not unique.

This is why truncation or boundedness assumptions matter: they turn the continuous operator problem back into a Perron–Frobenius problem with a unique positive eigenvector.

Carr and Yu [2012] establish recovery with a bounded diffusion.

Transition independence:

If the kernel is not transition independent, recovery is not guaranteed.

Borovička *et al.* [2016] show that the Ross recovery can confound the long-run risk component of the kernel with the natural probability distribution, yielding an incorrect decomposition.

We discuss this in the sequel lecture *Misspecified Recovery*.

Empirical estimation:

Extracting reliable state prices from observed option prices requires careful interpolation and extrapolation.

The mapping from implied volatilities to state prices via the Breeden and Litzenberger [1978] formula involves second derivatives, which amplify measurement error.

126.12 Exercises

Exercise 126.12.1

The Perron–Frobenius vector and the pricing kernel.

Consider the 3×3 state price matrix

$$P = \begin{pmatrix} 0.5950 & 0.1700 & 0.0272 \\ 0.159375 & 0.5525 & 0.1360 \\ 0.06640625 & 0.31875 & 0.5525 \end{pmatrix}.$$

1. Compute the Perron eigenvalue β and the corresponding eigenvector z of P.
2. Use z to recover the natural probability transition matrix F via
$$f_{ij} = \frac{1}{\beta} \frac{z_j}{z_i} p_{ij}.$$
3. Verify that each row of F sums to one and all entries are positive.
4. For destination state j, the relative kernel component is $1/z_j$; for a transition from state i to state j, the full pricing kernel is $\beta z_i / z_j$. Compute $1/z_j$ for each state.

Does the kernel decrease as we move from state 1 to state 3 (i.e., from bad to good states)?

Solution

Here is one solution:

```
P_ex = np.array([
    [0.5950, 0.1700, 0.0272],
    [0.159375, 0.5525, 0.1360],
    [0.06640625, 0.31875, 0.5525]
])

eigenvalues, eigenvectors = eig(P_ex)
real_mask = np.isreal(eigenvalues)
real_ev = eigenvalues[real_mask].real
real_evec = eigenvectors[:, real_mask].real

idx = np.argmax(real_ev)
β_ex = real_ev[idx]
z_ex = real_evec[:, idx]
if z_ex.min() < 0:
    z_ex = -z_ex
z_ex = z_ex / z_ex[1]

print(f"β = {β_ex:.6f}")
print(f"z = {z_ex}")

D_ex = np.diag(1.0 / z_ex)
D_inv_ex = np.diag(z_ex)
F_ex = (1.0 / β_ex) * D_ex @ P_ex @ D_inv_ex

print("\nRecovered F:")
print(np.round(F_ex, 4))
```

```
print(f"\nRow sums: {np.round(F_ex.sum(axis=1), 8)}")
print(f"Nonnegative: {(F_ex >= -1e-10).all()}")

ψ_relative_ex = 1.0 / z_ex
print(f"\nrelative kernel 1/z = {np.round(ψ_relative_ex, 4)}")
print(f"Decreasing: {ψ_relative_ex[0] > ψ_relative_ex[1] > ψ_relative_ex[2]}")
```

```
β = 0.850000
z = [0.8  1.   1.25]

Recovered F:
[[0.7  0.25 0.05]
 [0.15 0.65 0.2 ]
 [0.05 0.3  0.65]]

Row sums: [1. 1. 1.]
Nonnegative: True

relative kernel 1/z = [1.25 1.   0.8 ]
Decreasing: True
```

Exercise 126.12.2

Stochastic dominance.

Using the recovered F and the normalised risk-neutral matrix $Q = P/\text{row sums}$ from the exercise above:

1. Compute the one-step marginal distributions $f_j = F_{2,j}$ and $q_j = Q_{2,j}$ starting from state 2 (index 1 in Python).
2. Compute the CDFs $\hat{F}_k = \sum_{j \le k} f_j$ and $\hat{Q}_k = \sum_{j \le k} q_j$ for each state.
3. Verify numerically that $\hat{F}_k \le \hat{Q}_k$ for every k, confirming stochastic dominance in this ordered three-state example.

Solution

Here is one solution:

```
eigenvalues, eigenvectors = eig(P_ex)
real_mask = np.isreal(eigenvalues)
real_ev = eigenvalues[real_mask].real
real_evec = eigenvectors[:, real_mask].real
idx = np.argmax(real_ev)
β_ex = real_ev[idx]
z_ex = real_evec[:, idx]
if z_ex.min() < 0:
    z_ex = -z_ex
z_ex = z_ex / z_ex[1]

D_ex = np.diag(1.0 / z_ex)
D_inv_ex = np.diag(z_ex)
F_ex = (1.0 / β_ex) * D_ex @ P_ex @ D_inv_ex

start = 1
```

```
f_marg = F_ex[start]
q_marg = P_ex[start] / P_ex[start].sum()

print("One-step marginals from state 2:")
print(f"natural      = {np.round(f_marg, 4)}")
print(f"risk-neutral = {np.round(q_marg, 4)}")

cdf_nat = np.cumsum(f_marg)
cdf_rn = np.cumsum(q_marg)

print("\nCDFs:")
for k in range(3):
    print(f"state {k+1}: natural = {cdf_nat[k]:.4f}, risk-neutral = {cdf_rn[k]:.4f}
 ↪")

dominates = np.all(cdf_nat <= cdf_rn + 1e-10)
print(f"\nNatural CDF <= risk-neutral CDF: {dominates}")
```

```
One-step marginals from state 2:
natural      = [0.15 0.65 0.2 ]
risk-neutral = [0.188  0.6516 0.1604]

CDFs:
state 1: natural = 0.1500, risk-neutral = 0.1880
state 2: natural = 0.8000, risk-neutral = 0.8396
state 3: natural = 1.0000, risk-neutral = 1.0000

Natural CDF <= risk-neutral CDF: True
```

Exercise 126.12.3

Risk aversion and tail risk.

Write a function `tail_risk_ratio(γ, threshold, μ, σ, ρ, T)` that:

1. Constructs the state price matrix P using `build_state_price_matrix` with the given parameters and `n_states=41`.
2. Applies `recover_natural_distribution` to obtain F.
3. Computes $P(\text{log-return} \leq \text{threshold})$ under both the natural and risk-neutral distributions starting from the middle state.
4. Returns the ratio $p_{\text{risk-neutral}}/p_{\text{natural}}$.

Using this function, plot the ratio as a function of $\gamma \in [1, 10]$ for a 30 percent simple decline, i.e. `threshold = np.log(0.70)`.

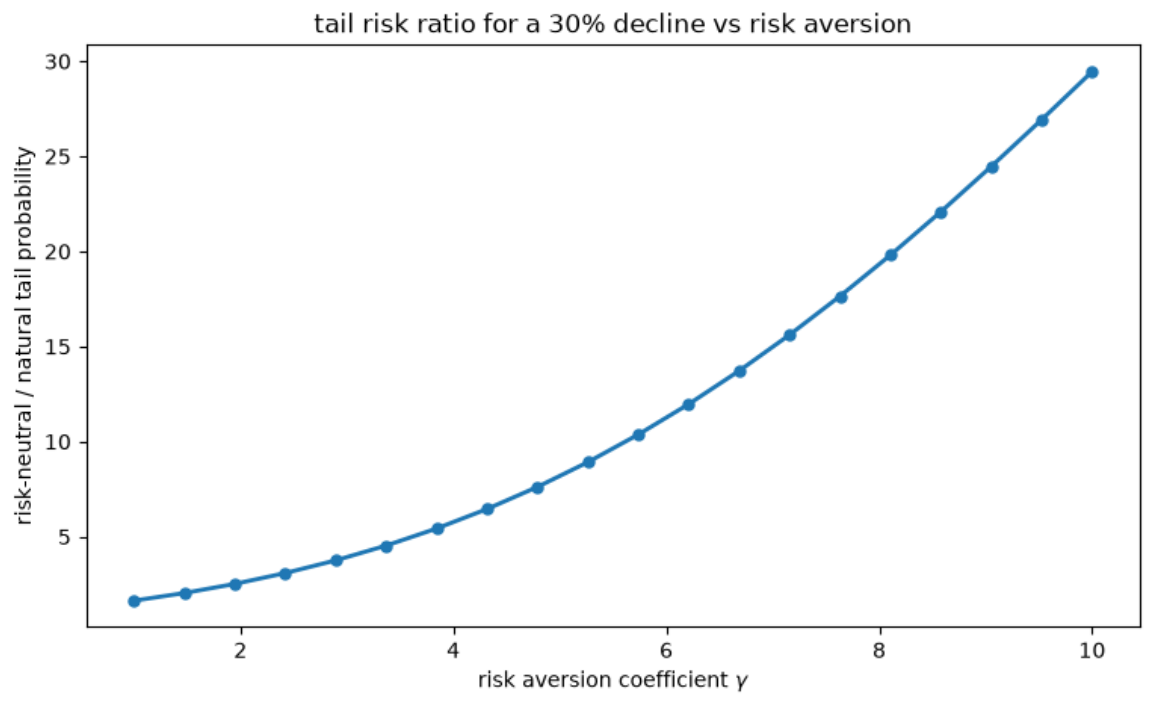
Explain the economic interpretation: why does a higher γ raise the ratio?

Solution

Here is one solution:

```
def tail_risk_ratio(γ, threshold, μ=0.08, σ=0.20, ρ=0.02, T=1.0):
    """Risk-neutral / natural left-tail probability."""
    P_g, states_g = build_state_price_matrix(
```

```
        μ, σ, γ, ρ, T, n_states=41, n_σ=5)

    F_g, _, _, _ = recover_natural_distribution(P_g)

    mid_g = len(states_g) // 2

    f_nat_g = F_g[mid_g]
    f_rn_g = P_g[mid_g] / P_g[mid_g].sum()

    p_nat = float(np.sum(f_nat_g[states_g <= threshold]))
    p_rn = float(np.sum(f_rn_g[states_g <= threshold]))

    if p_nat < 1e-12:
        return np.nan
    return p_rn / p_nat


γs = np.linspace(1.0, 10.0, 20)
threshold_30 = np.log(0.70)
ratios = [tail_risk_ratio(g, threshold_30) for g in γs]

plt.figure(figsize=(9, 5))
plt.plot(γs, ratios, '-o', ms=5, lw=2)
plt.xlabel('risk aversion coefficient $\\gamma$')
plt.ylabel('risk-neutral / natural tail probability')
plt.title('tail risk ratio for a 30% decline vs risk aversion')
plt.show()
```



A higher coefficient of risk aversion γ makes the pricing kernel steeper: the market assigns a larger premium per unit of probability to bad-state payoffs.

Risk-neutral probabilities incorporate this premium, so in this CRRA simulation the risk-neutral crash probability rises faster with γ than the recovered natural crash probability.

Recovery separates the market's estimated natural crash probability from the pricing-kernel premium attached to crash states.

CHAPTER

SEVEN

LONG-TERM RISK: AN OPERATOR APPROACH

Contents

- *Long-term Risk: An Operator Approach*
  - *Overview*
  - *Multiplicative functionals*
  - *Semigroups*
  - *The generator*
  - *Principal eigenfunctions*
  - *A finite-state Markov chain*
  - *The affine diffusion example*
  - *Long-run risk prices*
  - *Assumptions behind the scenes*
  - *Summary*
  - *Exercises*

127.1 Overview

Standard short-horizon asset pricing tells us how equilibrium prices compensate investors for tiny, instantaneous exposures to shocks.

That is, they tell us about the *short end* of the term structure of risk prices.

But many of the most interesting asset pricing questions (e.g., the equity premium puzzle, the slope of the yield curve, the prices of long-dated options) concern the *long end* of the term structure of risk prices.

This lecture studies the long end using the operator approach of Hansen and Scheinkman [2009].

At center stage will be a stochastic discount factor process and a return process that are multiplicative across time in the sense that conditional expectations define a *semigroup* of valuation operators indexed by horizon t.

Long-horizon behaviour of the semigroup is controlled by a single eigenvalue problem on the state space.

By solving that eigenvalue problem and selecting an appropriate eigenfunction, we can factor the multiplicative functional M_t into three economically meaningful pieces: a deterministic exponential trend, a non-negative martingale that changes

probability measure, and a transient state-dependent component.

In particular, the factorization takes the form

$$M_t = \exp(\rho t)\, \widehat{M}_t \, \frac{\phi(X_0)}{\phi(X_t)}, \tag{127.1}$$

where

- ρ is a scalar (the **principal eigenvalue**) giving the asymptotic growth or decay rate,
- ϕ is a strictly positive **principal eigenfunction** on the state space, capturing the long-run dependence on the current state,
- $\widehat{M}$ is a positive martingale used to change probability measure (the **martingale component**), and
- $\phi(X_0)/\phi(X_t)$ is a transient component that washes out as the twisted process settles into its stationary distribution.

In finite-state problems this is exactly the Perron-Frobenius decomposition of a positive matrix; in general state spaces it is a continuous-state counterpart.

We will refer to (127.1) as the **multiplicative factorization** associated with $(\rho, \phi, \widehat{M})$.

Alvarez and Jermann [2005] applied a related permanent-transitory decomposition to stochastic discount factors.

The operator approach links that decomposition to an explicit eigenvalue problem.

See also

This lecture is closely related to the lecture Additive and Multiplicative Functionals, which studies the same kind of permanent-transitory decomposition for additive and multiplicative functionals in a discrete-time linear-Gaussian setting.

Reading these two lectures together is a good way to learn about representations of long-run risks in both continuous and discrete time.

We will build up to (127.1) and use it to compute long-run risk prices in concrete models.

The plan of this lecture is to:

1. Set up positive multiplicative functionals M (discount factors, returns, stochastic growth) and the valuation semigroups they generate.
2. Introduce the **generator** of a semigroup, a local operator whose eigenvalue problem controls long-run behaviour.
3. Find the principal eigenfunction ϕ and derive the factorization.
4. Work two examples in detail: a finite-state Markov chain (where everything reduces to Perron-Frobenius theory) and an affine diffusion model (where the eigenfunction is exponential-affine and we get closed-form formulas).
5. Use the factorization to compute long-run risk prices and compare them to the local risk prices appropriate for short-horizon asset pricing.

A recurring theme is that shocks to persistent state variables drive a wedge between local and long-run risk prices.

Generating that wedge is the mechanism through which long-run risk models like Bansal and Yaron [2004] generate large equity premia.

We start with the following imports

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.linalg import eig, expm
```

127.2 Multiplicative functionals

127.2.1 Setting

Let $\{X_t : t \geq 0\}$ be a continuous-time Markov process with state space $\mathcal{D}_0$, and let $\mathcal{F}_t$ denote the filtration generated by its history.

We will work with a strong Markov process whose sample paths are càdlàg (defined below).

To arrive at the explicit formulas presented later we will specialize to a semimartingale that decomposes into a continuous component X^c and a pure-jump component X^j:

$$X_t = X_t^c + X_t^j.$$

We write the continuous-component dynamics as

$$dX_t^c = \xi(X_{t-})\,dt + \Gamma(X_{t-})\,dB_t,$$

and the pure-jump component as

$$dX_t^j = \int (y - X_{t-})\,\zeta(dy, dt),$$

where ζ is the random counting measure of jumps indexed by the *post-jump state* y, so that $\zeta(B, [0, t])$ counts the number of jumps in $[0, t]$ landing in the Borel set B.

Its compensator is $\eta(dy \mid X_{t-})\,dt$, the rate at which X jumps from X_{t-} into a region dy.

We will use this "y = post-jump state" convention throughout, matching the form $\phi(y) - \phi(x)$ that appears in the generator below.

We also impose two simplifying assumptions:

- **Finite jumps** on finite time intervals: only finitely many jumps occur on any bounded interval; this keeps integrals against the jump measure well-defined and finite.
- **Sufficient rank in** Γ so that the Brownian shocks relevant for pricing can be recovered from the state history; this makes the Markov state X "rich enough" to describe valuation.

These assumptions let us write the generator in closed form and use martingale-based changes of measure freely.

127.2.2 Functionals and càdlàg paths

We need a name for "a process that records something about the history of X".

This includes, for example, a stochastic discount factor or a cumulated return.

Definition 127.2.1 (Functional)

A **functional** is a real-valued adapted process $\{M_t : t \geq 0\}$ constructed from the history of X, so that M_t is $\mathcal{F}_t$-measurable for each t.

We will always work with the **càdlàg** version of a functional, the French acronym for "right-continuous with left limits".

Concretely, for almost every sample path ω,

$$\lim_{s \downarrow t} M_s(\omega) = M_t(\omega) \qquad \text{for all } t \geq 0,$$

and the left limit $M_{t-}(\omega) := \lim_{s \uparrow t} M_s(\omega)$ exists and is finite for all $t > 0$.

Thus, paths can jump, but each jump $\Delta M_t := M_t - M_{t-}$ occurs instantaneously.

At the jump time t, the value is the post-jump value, not the pre-jump value.

Note

Why is the càdlàg property worth insisting on?

Because we will later want to (i) integrate functionals against time, (ii) apply optional stopping arguments, and (iii) take limits like $\lim_{t\to\infty} t^{-1} \log M_t$.

All three operations need the joint measurability in (ω, t) that càdlàg paths give us automatically.

127.2.3 Multiplicativity

The central restriction we impose on the functional M is that it is *multiplicative across time*.

Definition 127.2.2 (Multiplicative Functional)

A functional $\{M_t : t \geq 0\}$ is **multiplicative** if $M_0 = 1$ and

$$M_{t+u} = M_u(\theta_t)\, M_t, \qquad t, u \geq 0, \tag{127.2}$$

where θ_t shifts the underlying Markov path forward by t units.

Why is this a useful condition to require?

Think of $M_t = S_t$, a stochastic discount factor.

The date-0 value of a date-t payoff Π_t is $\mathbb{E}[S_t \Pi_t \mid \mathcal{F}_0]$.

If we instead buy this payoff at intermediate date τ, its date-τ price must be

$$\mathbb{E}\left[\frac{S_t}{S_\tau}\Pi_t \,\middle|\, \mathcal{F}_\tau\right].$$

For the price to depend only on the current Markov state X_τ (and not on the entire history up to τ), the ratio S_t/S_τ must be a function only of the Markov path *after* τ.

Thus, $S_{\tau+u}/S_\tau = S_u(\theta_\tau)$, which is exactly (127.2).

So multiplicativity is the Markov form of the law of one price.

The same structural property is then carried over to stochastic growth and cumulated returns.

127.2.4 Additive functionals

It is often easier to work with the *log* of a strictly positive multiplicative functional.

If $M_t > 0$, we can write $M_t = \exp(A_t)$, and the multiplicative property (127.2) becomes the corresponding additive property of A.

Definition 127.2.3 (Additive Functional)

A functional $\{A_t : t \geq 0\}$ is **additive** if $A_0 = 0$ and

$$A_{t+u} = A_u(\theta_t) + A_t.$$

So exponentials of additive functionals are exactly the strictly positive multiplicative functionals.

In our jump-diffusion setting, a useful parameterization is

$$A_t = \int_0^t \beta(X_s)\, ds + \int_0^t \gamma(X_{s-})^\top dB_s + \sum_{0 \leq s \leq t} \kappa(X_s, X_{s-}), \tag{127.3}$$

where the three functions (β, γ, κ) play the following roles:

- $\beta(x)$ is a **state-dependent drift** rate (e.g., a pure discount factor with short rate $r(x)$ has $\beta(x) = -r(x)$).
- $\gamma(x)$ is a **Brownian loading**, controlling how Brownian shocks feed into A at state x.
- $\kappa(y, x)$ is a **jump amplitude** that fires whenever X jumps from x to y.

For everything to be well defined, we impose the natural integrability conditions

$$\int_0^t |\beta(X_s)|\, ds < \infty, \qquad \int_0^t \|\gamma(X_s)\|^2\, ds < \infty,$$

together with $\kappa(x, x) = 0$ (no jump if the state doesn't change) and

$$\int \exp[\kappa(y, x)]\, \eta(dy \mid x) < \infty.$$

This parameterization is rich enough for everything we do in this lecture, though additive functionals can also include things like occupation times.

127.3 Semigroups

A multiplicative functional M together with the Markov process X defines a **valuation operator** for each horizon t:

$$\mathbb{M}_t \psi(x) = \mathbb{E}\left[M_t \psi(X_t) \mid X_0 = x\right].$$

You should read $\mathbb{M}_t \psi(x)$ as "the date-0 value, starting from state x, of a date-t payoff $\psi(X_t)$", weighted by M_t.

The family of operators $\{\mathbb{M}_t\}_{t \geq 0}$ has a key compositional structure, the *semigroup property*.

Definition 127.3.1 (One-Parameter Semigroup)

A family of linear operators $\{T_t : t \geq 0\}$ is a **one-parameter semigroup** if $T_0 = I$ and $T_{t+s} = T_t T_s$ for all $s, t \geq 0$.

It is **positive** if $T_t \psi \geq 0$ whenever $\psi \geq 0$.

Definition 127.3.2 (Multiplicative Semigroup)

Given a multiplicative functional M, the associated **multiplicative semigroup** is the family of operators

$$\mathbb{M}_t\psi(x) = \mathbb{E}\left[M_t\psi(X_t) \mid X_0 = x\right]. \tag{127.4}$$

The semigroup identity $\mathbb{M}_{t+u} = \mathbb{M}_t\mathbb{M}_u$ follows from iterated expectations and the multiplicative property of M.

Economically, this is the *Markov law of iterated values*: to value a date-$(t+u)$ payoff today, we can either

- discount it directly from $t+u$ back to 0 in one step (apply $\mathbb{M}_{t+u}$), or
- first discount it from $t+u$ back to t (apply $\mathbb{M}_u$), then discount the resulting date-t price back to 0 (apply $\mathbb{M}_t$).

The semigroup identity says these two procedures give the same answer.

This is the operator-level version of the intertemporal consistency that rules out arbitrage across horizons.

Four positive multiplicative functionals will appear often below.

Symbol	Object	Semigroup
S	stochastic discount factor	$\{\mathbb{S}_t\}$
V	cumulated return on an asset	$\{\mathbb{V}_t\}$
G	stochastic growth in cash flows	$\{\mathbb{G}_t\}$
$Q = GS$	valuation of growing cash flows	$\{\mathbb{Q}_t\}$

The first three are primitives.

The fourth combines discounting and growth to value cash flows that grow stochastically over time.

Definition 127.3.3 (Stochastic Discount Factor)

A **stochastic discount factor** S is a positive multiplicative functional for which $\mathbb{E}[S_tZ_t \mid X_0 = x]$ is the date-0 value of an $\mathcal{F}_t$-measurable payoff Z_t.

Definition 127.3.4 (Valuation Functional)

Given a stochastic discount factor S, a **valuation functional** V is a positive multiplicative functional such that $\{V_tS_t : t \geq 0\}$ is a martingale.

Definition 127.3.5 (Stochastic Growth Functional)

A **stochastic growth functional** G is a positive multiplicative functional, interpreted as a multiplicative growth factor applied to a date-0 cash flow.

Consider a cash flow $D_t = D_0\, G_t\, \psi(X_t)$, where $D_0 > 0$ is the date-0 level and ψ is a Borel state-payoff function.

Discounting with S and growing with G, its date-0 value is

$$D_0\, \mathbb{Q}_t\psi(X_0), \qquad \mathbb{Q}_t\psi(x) = \mathbb{E}\left[G_tS_t\psi(X_t) \mid X_0 = x\right].$$

Definition 127.3.6 (Cash-Flow Valuation Semigroup)

The **cash-flow valuation semigroup** is the multiplicative semigroup generated by $Q = GS$.

The long-horizon behaviour of $\mathbb{Q}_t$ is the central object of the lecture: it tells us how current prices value cash-flow growth risk that materializes far in the future.

Note

The factorization $D_t = D_0 G_t \psi(X_t)$ is not unique.

For any positive function φ,

$$D_t = D_0 \left[G_t \frac{\varphi(X_t)}{\varphi(X_0)} \right] \left[\frac{\psi(X_t)\varphi(X_0)}{\varphi(X_t)} \right],$$

so a transient state-dependent factor can be shuffled between G and ψ without changing D_t.

We resolve this indeterminacy by normalizing the growth component so that its permanent part is a martingale: $G_t = \exp(\delta t)\widehat{G}_t$, with $\widehat{G}$ a martingale and δ a constant trend.

The eigenfunction construction below will tell us exactly which martingale to pick.

127.3.1 Local pricing restriction

Before tackling long horizons, it is good to understand valuation at the *short* end.

That is the standard instantaneous risk-return relation.

This will give us a benchmark against which to compare long-run risk prices.

For a textbook discrete-time treatment of the same SDF-based asset-pricing ideas, see Elementary Asset Pricing Theory.

For an estimation perspective on Euler-equation-based asset pricing, see *Estimating Euler Equations by Generalized Method of Moments*.

The key starting point is that a valuation functional V must satisfy the no-arbitrage requirement that VS is a martingale (*Definition 127.3.4*).

We parameterize the stochastic discount factor S and valuation functional V as additive functionals with coefficients $(\beta^s, \gamma^s, \kappa^s)$ and $(\beta^v, \gamma^v, \kappa^v)$ respectively, in the notation of (127.3).

For a generic positive multiplicative functional $M = \exp(A)$ with parameters (β, γ, κ), applying Itô's formula to $\exp(A_t)$ and requiring the drift of dM_t/M_{t-} to vanish gives the **local martingale restriction**:

$$\beta(x) + \frac{\gamma(x)^\top \gamma(x)}{2} + \int \left(\exp[\kappa(y, x)] - 1 \right) \eta(dy \mid x) = 0. \tag{127.5}$$

The three terms correspond to:

- the drift of A itself,
- the Itô correction from the Brownian part (because $M = e^A$ picks up a quadratic-variation contribution), and
- the compensated jumps of M at the multiplier $\exp[\kappa(y, x)]$.

We ask you to verify this in Exercise 127.11.4.

Applying this to VS, whose parameters add: $(\beta^v + \beta^s, \gamma^v + \gamma^s, \kappa^v + \kappa^s)$, gives the **local pricing restriction**

$$\beta^v + \beta^s = -\frac{\|\gamma^v + \gamma^s\|^2}{2} - \int (\exp[\kappa^v(y,\cdot) + \kappa^s(y,\cdot)] - 1)\, \eta(dy \mid \cdot). \tag{127.6}$$

This determines the drift β^v of any candidate valuation functional in terms of its Brownian and jump exposures.

To turn this into an *expected return*, note that the expected net rate of return on V is

$$\epsilon^v = \beta^v + \frac{\|\gamma^v\|^2}{2} + \int (\exp[\kappa^v(y,\cdot)] - 1)\, \eta(dy \mid \cdot).$$

Combining with (127.6) gives

$$\begin{aligned} \epsilon^v = &-\beta^s - \gamma^v \cdot \gamma^s - \frac{\|\gamma^s\|^2}{2} \\ &- \int (\exp[\kappa^v(y,\cdot) + \kappa^s(y,\cdot)] - \exp[\kappa^v(y,\cdot)])\, \eta(dy \mid \cdot). \end{aligned} \tag{127.7}$$

Thus, the Brownian local risk-price vector is $-\gamma^s(x)$, expressed in the same exposure units as $\gamma^v(x)$.

Jump risk is priced through the function κ^s.

This local relation is one end of the term structure of risk prices.

The eigenvalue calculations below describe the other end.

127.4 The generator

So far we have a family of operators $\{\mathbb{M}_t\}_{t\geq 0}$, one for each horizon t.

That is more information than we can use directly.

What we care about is the long-run behaviour: how $\mathbb{M}_t\psi$ grows as $t \to \infty$.

The **generator** $\mathbb{A}$ compresses the entire semigroup into one time-independent operator on the state space.

It records the *instantaneous* rate of change of $M_t\psi(X_t)$, and its eigenvalues drive the long-run growth rate of $\mathbb{M}_t$.

This lets us turn an asymptotic question about a family of operators into a single eigenvalue problem.

127.4.1 Discrete-time intuition

The role of the generator is easiest to see when time is discrete, so we build the picture there once and then carry it over.

Let X_n be Markov and M_n a multiplicative functional, and define the one-period valuation operator

$$K\psi(x) = \mathbb{E}\left[M_1\psi(X_1) \mid X_0 = x\right].$$

Iterating gives the n-period operator K^n.

This parallels the logic of a transition matrix: just as P^n gives n-step probabilities, K^n weights n-period payoffs, with M_1 folded in at each step.

So one local object, K, controls the entire horizon-indexed family.

It also controls long-run growth: if $K\phi = \lambda\phi$, then

$$K^n\phi = \lambda^n\phi,$$

so λ is the long-run geometric growth (or decay) rate and ϕ is the long-run dependence on the current state.

The same operator gives the predictable rate of change of the weighted payoff.

By multiplicativity,

$$\mathbb{E}\left[M_{n+1}\psi(X_{n+1}) - M_n\psi(X_n) \mid \mathcal{F}_n\right] = M_n(K\psi - \psi)(X_n),$$

so subtracting the cumulative predictable change from the total change leaves a martingale:

$$M_n\psi(X_n) - \psi(X_0) - \sum_{j=0}^{n-1} M_j(K\psi - \psi)(X_j).$$

Here $K - I$ is the *local* rate of change of $M_n\psi(X_n)$, and through K^n it also controls long-run growth.

127.4.2 From discrete to continuous time

Continuous time keeps the same logic.

The natural counterpart $K - I$ is the **infinitesimal generator** of the semigroup $\{\mathbb{M}_t\}$, the time derivative at zero:

$$\mathbb{A}\psi(x) := \lim_{h \downarrow 0} \frac{\mathbb{M}_h\psi(x) - \psi(x)}{h},$$

so that for small $h > 0$,

$$\mathbb{M}_h\psi(x) \approx \psi(x) + h\,\mathbb{A}\psi(x).$$

Note

When $M \equiv 1$, the multiplicative semigroup reduces to the standard Markov transition semigroup $\mathbb{M}_t\psi(x) = \mathbb{E}[\psi(X_t) \mid X_0 = x]$, and $\mathbb{A}$ becomes the familiar infinitesimal generator $\mathcal{L}$ of X from textbook stochastic calculus.

For general M, $\mathbb{A}$ is the same kind of object but it carries extra terms that encode the instantaneous "yield" of M: discount rates, Brownian risk prices, jump multipliers.

That is exactly what we will see when we write down its closed form in (127.8) below.

Why is this the right object?

If $\mathbb{A}\phi = \rho\phi$, then the candidate

$$\hat{M}_t := \exp(-\rho t)\, M_t\, \frac{\phi(X_t)}{\phi(X_0)}$$

is automatically a *local* martingale.

When $\hat{M}$ is in fact a martingale (the content of Assumption 6.1 in Hansen and Scheinkman [2009]), taking expectations gives the continuous-time analogue of $K^n\phi = \lambda^n\phi$:

$$\mathbb{M}_t\phi = \exp(\rho t)\,\phi.$$

Without that upgrade we get only the supermartingale inequality $\mathbb{M}_t\phi \le \exp(\rho t)\,\phi$, which we will revisit below.

So the long-run behaviour of $\mathbb{M}_t$ is encoded in an eigenvalue problem for the local operator $\mathbb{A}$, together with the martingale property of $\hat{M}$.

127.4.3 Extended generator

The limit definition above needs a qualification.

To make the limit $h \downarrow 0$ rigorous, the textbook definition requires $(\mathbb{M}_h\psi - \psi)/h$ to converge to $\mathbb{A}\psi$ in a chosen norm, typically uniform convergence over bounded continuous functions, where the semigroup is a contraction.

But the functions we care about most are the principal eigenfunctions ϕ solving $\mathbb{A}\phi = \rho\phi$, and these typically *grow* with the state X (in the affine-Gaussian benchmark, ϕ is exponential-affine in x).

So they do not lie in this space, and the limit need not converge for them.

Hansen and Scheinkman [2009] sidestep this by characterizing $\mathbb{A}$ through a *Doob-Meyer style* semimartingale decomposition of $M_t\psi(X_t)$, a pathwise condition that does not require any norm.

This is the continuous-time counterpart to writing $K - I$ as the predictable rate of change of $M_n\psi(X_n)$ in discrete time.

The resulting **extended generator** admits unbounded ψ, has a strictly larger domain than the textbook generator, and agrees with it where both are defined.

Concretely:

Definition 127.4.1 (Extended Generator)

Fix a Borel function ψ, and look for a second Borel function χ that will play the role of "the instantaneous rate of change of $M_t\psi(X_t)$ at the current state". We ask whether there exists χ such that

$$N_t = M_t\psi(X_t) - \psi(X_0) - \int_0^t M_s\chi(X_s)\,ds$$

is a local martingale.

We then say ψ lies in the domain of the **extended generator** $\mathbb{A}$ of M, and we *define* the operator by

$$\mathbb{A}\psi := \chi.$$

The three terms play the same roles as in discrete time:

- $M_t\psi(X_t) - \psi(X_0)$ is the total change in the weighted payoff over $[0, t]$,
- $\int_0^t M_s\chi(X_s)\,ds$ accumulates the expected rate of change along the path,
- N_t is the residual.

Requiring N_t to be a local martingale pins down $\chi = \mathbb{A}\psi$ as the instantaneous expected rate of change of $M_t\psi(X_t)$ at the current state.

- When $M \equiv 1$, the definition reduces to Dynkin's formula for the standard Markov generator $\mathcal{L}\psi(x) = \lim_{t\downarrow 0} t^{-1}[\mathbb{E}\psi(X_t) - \psi(x)]$.
- When X is a jump diffusion, Itô's formula applied to $M_t\psi(X_t)$ produces the closed-form expression for $\mathbb{A}\psi$ in (127.8) below.

127.4.4 A closed form for jump diffusions

For the jump-diffusion setting introduced above, we can compute $\mathbb{A}$ explicitly by applying Itô's formula to $M_t\phi(X_t)$.

Suppose the continuous part of X satisfies

$$dX_t^c = \xi(X_t)dt + \Gamma(X_t)dB_t$$

with diffusion matrix $\Sigma = \Gamma\Gamma^\top$, and let $\eta(dy \mid x)$ denote the jump compensator.

If $M = \exp(A)$ is parameterized by (β, γ, κ) as in (127.3), then for smooth ϕ,

$$\begin{aligned}\mathbb{A}\phi(x) = &\left[\xi(x) + \Gamma(x)\gamma(x)\right]^\top \frac{\partial\phi(x)}{\partial x} \\ &+ \frac{1}{2}\,\text{trace}\left[\Sigma(x)\frac{\partial^2\phi(x)}{\partial x\partial x^\top}\right] \\ &+ \int [\phi(y) - \phi(x)]\exp[\kappa(y,x)]\eta(dy \mid x) \\ &+ \left[\beta(x) + \frac{\gamma(x)^\top\gamma(x)}{2} + \int \left(\exp[\kappa(y,x)] - 1\right)\eta(dy \mid x)\right]\phi(x).\end{aligned} \tag{127.8}$$

The four terms have transparent interpretations:

1. The first term is the standard Markov drift, modified by $\Gamma\gamma$, a *covariance correction* between the Brownian shocks driving X and those driving M.
2. The second is the standard diffusion (Itô) term.
3. The third integrates ϕ against the jump-compensated transition rates, reweighted by the jump multiplier $\exp[\kappa(y,x)]$.
4. The fourth is a multiplicative *yield-like* term that multiplies $\phi(x)$ itself and combines the drift of M, the Brownian Itô correction, and the compensated jumps.

Note

When $M = S$ is a stochastic discount factor, the local risk prices are spread across the four pieces of the generator rather than concentrated in the level term.

- Brownian factor prices enter through the drift modification $\xi \to \xi + \Gamma\gamma$ in the first line.
- Jump-risk prices enter through the tilted jump measure $\eta \to \exp[\kappa]\,\eta$ in the third line.
- The level term in the fourth line contains the instantaneous risk-free rate component (the *short end* of the term structure) plus the jump-compensator contribution $\int(\exp[\kappa] - 1)\,\eta$.

We ask readers to verify the derivation of (127.8) in Exercise 127.11.5.

We will apply this formula directly in the affine-diffusion example below.

127.5 Principal eigenfunctions

We now arrive at the central technical question of the lecture:

> Which positive payoffs grow at a constant proportional rate under the valuation semigroup?

The answer, when it exists, is a positive eigenfunction of the generator.

Definition 127.5.1 (Eigenfunction of the Extended Generator)

A Borel function ϕ is an **eigenfunction** of $\mathbb{A}$ with eigenvalue ρ if

$$\mathbb{A}\phi = \rho\phi. \tag{127.9}$$

A **principal eigenfunction** is an eigenfunction ϕ that is strictly positive on the state space: $\phi(x) > 0$ for all $x \in \mathcal{D}_0$.

The strict-positivity requirement matters because ϕ will appear in denominators throughout: it has to be safe to divide by it.

127.5.1 From eigenfunction to factorization

Why does an eigenfunction of $\mathbb{A}$ give us the multiplicative factorization (127.1)?

The discrete-time analogy points the way. If $K\phi = \lambda\phi$, then

$$\lambda^{-n} M_n \frac{\phi(X_n)}{\phi(X_0)}$$

is a martingale: the eigenvalue equation absorbs the one-step drift of $M_n\phi(X_n)$ exactly.

In continuous time, λ^n becomes $\exp(\rho t)$, and the candidate martingale is

$$\hat{M}_t = \exp(-\rho t)\, M_t \frac{\phi(X_t)}{\phi(X_0)}. \tag{127.10}$$

Let's check that $\hat{M}$ is indeed a local martingale.

By the definition of $\mathbb{A}$, the semimartingale $Z_t := M_t\phi(X_t)$ admits the decomposition

$$dZ_t = M_t\, \mathbb{A}\phi(X_t)\, dt + dN_t,$$

where N is a local martingale.

The eigenvalue equation $\mathbb{A}\phi = \rho\phi$ replaces the drift by $\rho Z_t\, dt$, and integration by parts gives

$$d(\exp(-\rho t)Z_t) = \exp(-\rho t)\left[dZ_t - \rho Z_t\, dt\right] = \exp(-\rho t)\, dN_t.$$

So $\exp(-\rho t)Z_t$ is a local martingale, and dividing by $\phi(X_0)$ shows the same for $\hat{M}$.

Rearranging (127.10) for M_t recovers the factorization (127.1), with $\hat{M}$ in the role of the martingale component.

Definition 127.5.2 (Martingale Component and Twisted Measure)

When $\hat{M}$ is a martingale (not merely a local martingale), it is the **martingale component** associated with (ρ, ϕ) and defines the **twisted probability measure** by weighting date-t events with $\hat{M}_t$:

$$\widehat{\Pr}(F \mid X_0 = x) = \mathbb{E}[\hat{M}_t \mathbf{1}_F \mid X_0 = x], \qquad F \in \mathcal{F}_t.$$

The verification establishes only that $\hat{M}$ is a *local* martingale, but the definition above (and the change-of-measure interpretation of $\hat{M}$) require it to be a martingale.

We close this gap by adopting Assumption 6.1 of Hansen and Scheinkman [2009]:

> The local martingale $\hat{M}$ defined in (127.10) is a martingale, i.e. $\mathbb{E}[\hat{M}_t \mid X_0 = x] = 1$ for every $t \geq 0$ and x.

We carry this assumption from here on.

Even without closing the gap we still get one-sided control.

Since $\hat{M}$ is nonnegative with $\hat{M}_0 = 1$, it is a supermartingale, so $\mathbb{E}[\hat{M}_t \mid X_0 = x] \leq 1$.

Taking expectations in (127.10),

$$\mathbb{M}_t \phi(x) \leq \exp(\rho t)\, \phi(x),$$

so ρ is at least an upper bound on the long-run growth rate of $\mathbb{M}_t \phi$.

When $\hat{M}$ is in fact a martingale, $\mathbb{E}[\hat{M}_t] = 1$, the inequality becomes an equality, and the local condition $\mathbb{A}\phi = \rho\phi$ lifts to the semigroup eigenvalue equation

$$\mathbb{M}_t \phi = \exp(\rho t)\, \phi, \qquad t \geq 0. \tag{127.11}$$

127.5.2 Stability of the twisted process

We now have a factorization (127.1) for *any* principal eigenfunction.

But for (ρ, ϕ) to describe **long-run** behaviour of $\mathbb{M}_t$ the twisted process must settle into a stationary regime as $t \to \infty$.

If it doesn't, the transient factor $\phi(X_0)/\phi(X_t)$ will not vanish, and we cannot read off the asymptotics from ρ alone.

We need three conditions, each ruling out a specific way in which the twisted process can fail to settle down.

Let $\hat{E}$ and $\widehat{\Pr}$ denote expectation and probability under the twisted measure, and let $\hat{\mathbb{A}}$ be the generator of X under that measure.

Condition 1: a stationary distribution exists.

Definition 127.5.3 (Stationary Distribution of the Twisted Process)

A probability measure $\hat{\varsigma}$ on $\mathcal{D}_0$ is a **stationary distribution** for the twisted process if

$$\int \hat{\mathbb{A}}\psi \, d\hat{\varsigma} = 0$$

for every ψ in the L^∞ domain of $\hat{\mathbb{A}}$.

$\hat{\varsigma}$ is the candidate long-run distribution.

Without it, the twisted process has no steady state, and the long-run limit cannot be expressed as a state-space integral.

Condition 2: every important region is reachable.

Definition 127.5.4 (Irreducible Skeleton)

The discretely sampled skeleton $\{X_{\Delta j} : j = 0, 1, ...\}$ is **irreducible** relative to $\hat{\varsigma}$ if there exists $\Delta > 0$ such that, for every Borel set $\Lambda \subseteq \mathcal{D}_0$ with $\hat{\varsigma}(\Lambda) > 0$,

$$\hat{E}\left[\sum_{j=0}^{\infty} \mathbf{1}_{\{X_{\Delta j} \in \Lambda\}} \middle| X_0 = x\right] > 0 \qquad \text{for all } x \in \mathcal{D}_0.$$

Without it, the long-run distribution could depend on the starting state; different basins of attraction would give different limits.

Condition 3: every important region is visited infinitely often.

Definition 127.5.5 (Harris Recurrence)

The process X is **Harris recurrent** under the twisted measure if, for every Borel set $\Lambda \subseteq \mathcal{D}_0$ with $\hat{\varsigma}(\Lambda) > 0$,

$$\widehat{\Pr}\left\{\int_0^{\infty} \mathbf{1}_{\{X_t \in \Lambda\}}\, dt = \infty \middle| X_0 = x\right\} = 1 \qquad \text{for all } x \in \mathcal{D}_0.$$

Reachability (Condition 2) is not enough.

A set may be reachable but visited with vanishingly small probability, so time averages fail to converge to $\hat{\varsigma}$-averages.

Harris recurrence is the continuous-state counterpart to a "recurrent state" in a finite chain.

Collecting the three conditions:

Definition 127.5.6 (Stochastically Stable Twisted Process)

The $\hat{M}$-twisted Markov process is **stochastically stable** if it has a stationary distribution $\hat{\varsigma}$, the skeleton $\{X_{\Delta j}\}$ is irreducible relative to $\hat{\varsigma}$, and X is Harris recurrent under the twisted measure.

127.5.3 The long-run approximation

Under the martingale condition for $\hat{M}$, strict positivity of M, and stochastic stability, the long-run limit takes a clean form:

$$\lim_{t \to \infty} \exp(-\rho t)\, \mathbb{M}_t \psi = \phi \int \frac{\psi}{\phi}\, d\hat{\varsigma}. \tag{127.12}$$

Read this as follows:

- The factor $\exp(\rho t)$ captures the exponential growth or decay of the semigroup; once we divide it out, what remains has a finite limit.
- The state dependence in that limit is *entirely* captured by $\phi(x)$.
- The scalar $\int (\psi/\phi)\, d\hat{\varsigma}$ is the **long-run intensity** of the payoff ψ, weighted by $1/\phi$ and averaged against the twisted stationary distribution.

The two precise cases of Proposition 7.1 of Hansen and Scheinkman [2009] are:

- **Sampled grid, $\hat{\varsigma}$-almost every starting state.** Fix any $\Delta > 0$ and assume $\int |\psi|/\phi \, d\hat{\varsigma} < \infty$; then the limit holds along $t = \Delta j$ for $\hat{\varsigma}$-almost every $x \in \mathcal{D}_0$.
- **Continuous t, every starting state.** If ψ/ϕ is bounded then the limit holds for every $x \in \mathcal{D}_0$ and every $t \to \infty$, with no continuity assumption on ψ.

Note

There can be more than one positive eigenfunction of $\mathbb{A}$ for which the associated $\hat{M}$ is a martingale, possibly with different eigenvalues.

Stochastic stability selects a particular one.

By Proposition 7.2 of Hansen and Scheinkman [2009], the stable eigenfunction's eigenvalue ρ is the **smallest** among all eigenvalues of $\mathbb{A}$ that admit a strictly positive eigenfunction satisfying the maintained conditions.

Any other positive eigenfunction sharing this ρ is proportional to ϕ $\hat{\varsigma}$-almost surely.

The finite-state section below states the same selection in the more familiar Perron-Frobenius language, calling ρ the eigenvalue with the *largest* real part among all eigenvalues of A.

These two descriptions identify the same eigenvalue because in the irreducible finite-state case only one eigenvalue of A admits a strictly positive eigenvector.

In the affine example below the two sets pull apart, since the quadratic (127.18) has two roots c_f that both give strictly positive exponential-affine eigenfunctions with distinct eigenvalues.

Stochastic stability picks the smaller ρ by rejecting the root that produces an explosive twisted process.

127.6 A finite-state Markov chain

We now apply the framework to a concrete example.

We start with the simplest case: a finite-state Markov chain.

For background on finite Markov chains in discrete time, see *Finite Markov Chains*.

For the asset-pricing applications of finite-state chains that motivate the construction here, see *Asset Pricing: Finite State Models*.

Here, every abstract object collapses to a familiar one:

Abstract object	Finite-state version
Markov process X	continuous-time chain with intensity matrix U
Generator $\mathbb{A}$	a matrix A
Semigroup $\mathbb{M}_t$	matrix exponential $\exp(tA)$
Principal eigenfunction ϕ	Perron right eigenvector
Principal eigenvalue ρ	dominant real eigenvalue of A
Stationary distribution $\hat{\varsigma}$	left eigenvector of twisted generator

So the long-run analysis reduces to Perron-Frobenius theory.

(For an introduction to the Perron-Frobenius theorem, see The Perron-Frobenius Theorem.)

127.6.1 Setup

Let X take values in $\{x_1, \ldots, x_N\}$ and let U be its intensity matrix.

Definition 127.6.1 (Intensity Matrix)

An **intensity matrix** U for a finite-state continuous-time Markov chain satisfies $u_{ij} \geq 0$ for $i \neq j$ and $\sum_j u_{ij} = 0$ for each state i.

Off-diagonal entries u_{ij} are the jump rates from state i to state j; the diagonal entry $u_{ii} = -\sum_{j \neq i} u_{ij}$ is minus the exit rate from state i.

Let the multiplicative functional have

- a **discount or decay rate** r_i in state i (the analogue of the drift $\beta(X_s)$ in the additive parameterization), and
- a **jump multiplier** $\exp[\kappa(x_j, x_i)]$ that fires whenever the state jumps from i to j.

In code below, `κ[j, i]` means $\kappa(x_j, x_i)$.

Then the generator matrix A for the multiplicative semigroup is

$$a_{ij} = \begin{cases} u_{ii} - r_i, & i = j, \\ u_{ij} \exp[\kappa(x_j, x_i)], & i \neq j. \end{cases} \tag{127.13}$$

The off-diagonal entries are the chain's jump rates *weighted* by the jump multipliers; the diagonal entries combine the exit rate with the in-state decay rate.

The semigroup is then just $\mathbb{M}_t = \exp(tA)$.

For an irreducible chain with strictly positive jump multipliers, the principal eigenvalue ρ is the unique real eigenvalue of A with largest real part, and the associated right eigenvector is strictly positive.

The twisted generator under the principal eigenpair (ρ, ϕ) is

$$\hat{A} = D_\phi^{-1} A D_\phi - \rho I,$$

where $D_\phi = \operatorname{diag}(\phi)$.

The row sums of $\hat{A}$ vanish, so $\hat{A}$ is itself a valid intensity matrix.

The stationary distribution $\hat{\varsigma}$ solves $\hat{\varsigma}^\top \hat{A} = 0$.

The helper functions below implement these three calculations.

```
def build_generator(U, r, κ):
    """Generator matrix for a finite-state multiplicative semigroup.

    κ[j, i] is the log jump multiplier for a transition i -> j.
    """
    U = np.asarray(U, dtype=float)
    r = np.asarray(r, dtype=float)
    κ = np.asarray(κ, dtype=float)

    A = U * np.exp(κ.T)
    np.fill_diagonal(A, np.diag(U) - r)
    return A
```

(continues on next page)

(continued from previous page)

```
def principal_eigenpair(A):
    """Perron eigenpair with a positive right eigenvector."""
    vals, vecs = eig(A)
    idx = np.argmax(vals.real)

    ρ = vals[idx].real
    ψ = vecs[:, idx].real

    if ψ.sum() < 0:
        ψ = -ψ

    if np.any(ψ <= 0):
        if np.min(ψ) > -1e-10:
            ψ = np.maximum(ψ, 0)
        else:
            raise ValueError(
                "Dominant eigenvector is not strictly positive.")

    ψ = ψ / ψ.mean()
    return ρ, ψ


def twisted_generator(A, ρ, ψ):
    """Generator under the twisted measure."""
    D = np.diag(ψ)
    D_inv = np.diag(1 / ψ)
    return D_inv @ A @ D - ρ * np.eye(len(ψ))


def stationary_distribution(Q):
    """Stationary distribution of a finite-state generator."""
    vals, vecs = eig(Q.T)
    idx = np.argmin(np.abs(vals))
    π = vecs[:, idx].real

    if π.sum() < 0:
        π = -π

    π = np.maximum(π, 0)
    return π / π.sum()
```

127.6.2 Two states

Consider a boom-recession economy.

State 1 is a *boom* (higher short rate $r_1 = 0.05$, switching to recession at rate $\lambda_1 = 0.30$).

State 2 is a *recession* (lower short rate $r_2 = 0.02$, switching to boom at rate $\lambda_2 = 0.50$).

For now we set the jump multipliers to zero, so the SDF only changes continuously through the in-state decay rates.

```
λ_1 = 0.30
λ_2 = 0.50

U = np.array([[-λ_1,  λ_1],
              [ λ_2, -λ_2]])
```

(continues on next page)

(continued from previous page)

```
r = np.array([0.05, 0.02])
κ = np.zeros((2, 2))

A = build_generator(U, r, κ)
ρ, ψ = principal_eigenpair(A)

print("A =")
print(np.round(A, 4))
print(f"\nρ = {ρ:.6f}")
print(f"ψ = {ψ}")
print(f"long-run zero-coupon yield = {-ρ:.4f}")
```

```
A =
[[-0.35  0.3 ]
 [ 0.5  -0.52]]

ρ = -0.038484
ψ = [0.98116799 1.01883201]
long-run zero-coupon yield = 0.0385
```

Note that $-\rho$ is the asymptotic yield on a zero-coupon bond: from (127.1), the date-0 price of a long zero-coupon bond decays like $\exp(\rho t)$, so its yield is $-\rho$.

Let's verify the semigroup eigenvalue equation $\mathbb{M}_t \phi = \exp(\rho t)\phi$ numerically.

```
for t in [1.0, 5.0, 25.0]:
    lhs = expm(t * A) @ ψ
    rhs = np.exp(ρ * t) * ψ
    err = np.max(np.abs(lhs - rhs))
    print(f"t = {t:4.1f}, error = {err:.2e}")
```

```
t =  1.0, error = 1.11e-16
t =  5.0, error = 1.11e-16
t = 25.0, error = 3.33e-16
```

The error is at machine precision for every t, so the equation holds exactly up to floating-point noise from the eigendecomposition.

Next we compute the twisted generator $\hat{A}$ and the stationary distribution $\hat{\varsigma}$ of the chain under the twisted measure.

This is the candidate long-run distribution that appears in the long-run limit (127.12).

```
A_hat = twisted_generator(A, ρ, ψ)
ς_hat = stationary_distribution(A_hat)

print("twisted generator row sums:")
print(np.round(A_hat.sum(axis=1), 12))

print("\ntwisted stationary distribution:")
print(f"  boom       {ς_hat[0]:.4f}")
print(f"  recession  {ς_hat[1]:.4f}")
```

```
twisted generator row sums:
[-0.  0.]
```

(continues on next page)

(continued from previous page)

```
twisted stationary distribution:
  boom      0.6072
  recession 0.3928
```

This twisted stationary distribution (not the original chain's stationary distribution) is what determines long-horizon valuations.

It differs from the stationary distribution of the original chain because the eigenfunction ϕ reweights states by how persistently they affect the multiplicative functional.

For any payoff function ψ, the long-run limit (127.12) is the vector

$$\phi \sum_i \frac{\psi_i}{\phi_i} \hat{\varsigma}_i.$$

Let's check that the rescaled semigroup converges to this limit as t grows.

```
ψ = np.array([1.0, 2.0])
limit = ψ * np.sum((ψ / ψ) * ς_hat)

for t in [1, 5, 20, 80]:
    approx = np.exp(-ρ * t) * expm(t * A) @ ψ
    print(f"t = {t:2d}, normalized value = {approx}")

print("\nlimit =", limit)
```

```
t =  1, normalized value = [1.19917652 1.68031101]
t =  5, normalized value = [1.35687434 1.42719756]
t = 20, normalized value = [1.36377325 1.41612444]
t = 80, normalized value = [1.3637733  1.41612436]

limit = [1.3637733  1.41612436]
```

The numerical values converge to the limit; the next plot shows the same convergence pictorially.

```
t_grid = np.linspace(0.01, 80, 400)
rescaled = np.array([np.exp(-ρ * t) * expm(t * A) @ ψ for t in t_grid])

fig, ax = plt.subplots()
ax.plot(t_grid, rescaled[:, 0], lw=2, label="starting from boom")
ax.plot(t_grid, rescaled[:, 1], lw=2, label="starting from recession")
ax.axhline(limit[0], color="C0", ls="--", lw=1, alpha=0.7)
ax.axhline(limit[1], color="C1", ls="--", lw=1, alpha=0.7)
ax.set_xlabel("$t$")
ax.set_ylabel(r"$e^{-\rho t}\,(\mathbb{M}_t \psi)(x)$")
ax.legend()
plt.show()
```

The two solid curves trace the rescaled values starting from each state, and the dashed horizontal lines mark the limits predicted by (127.12).

Both curves settle onto their predicted limits, confirming that the long-run behaviour depends on the starting state only through ϕ.

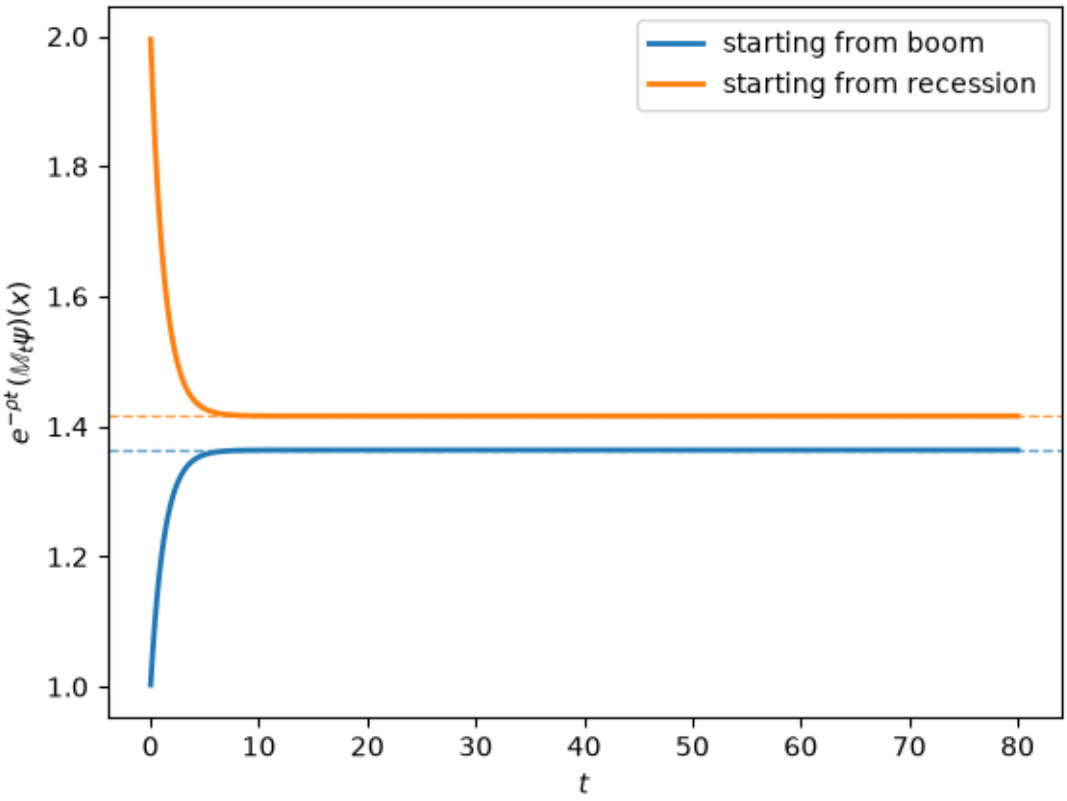


Fig. 127.1: Convergence of the rescaled semigroup to its long-run limit

Note

The asymptotic exponential rate of convergence is governed by the gap between the *real part* of the leading eigenvalue ρ and the largest real part among the remaining eigenvalues of A.

For an irreducible Metzler matrix the leading eigenvalue is real and its real part is strictly larger than the others, so this gap is well defined.

Exercise 127.11.3 works through a three-state example where the gap can be checked directly.

127.6.3 Adding jumps

State transitions in this chain are discontinuous, so it is natural to allow the multiplicative functional to jump at the transition times.

These jumps are the analogue of the κ function in the jump-diffusion parameterization.

A natural example arises with a stochastic discount factor that jumps *down* when the economy moves from recession into boom and *up* on the reverse transition.

The matrix `κ_jump` below encodes this.

We use the convention `κ[j, i]` = log jump multiplier of M for the transition $i \to j$, with state index 0 = boom and state index 1 = recession.

```
# recession (1) -> boom (0): SDF jumps down on good news (exp(-0.20))
# boom (0) -> recession (1): SDF jumps up on bad news (exp(+0.30))
κ_jump = np.array([[ 0.0, -0.20],
                   [ 0.30,  0.0]])

A_jump = build_generator(U, r, κ_jump)
ρ_jump, ψ_jump = principal_eigenpair(A_jump)

print(f"ρ without jumps = {ρ:.6f}")
print(f"ρ with jumps    = {ρ_jump:.6f}")
print("\nψ with jumps:")
print(ψ_jump)
```

```
ρ without jumps = -0.038484
ρ with jumps    = -0.019067

ψ with jumps:
[1.10059123 0.89940877]
```

To see how the long-run rate ρ responds to jump risk, we hold the boom-to-recession multiplier fixed and trace out ρ as the recession-to-boom multiplier varies.

```
κ_grid = np.linspace(-0.5, 0.5, 100)
ρ_grid = np.empty_like(κ_grid)

for n, k in enumerate(κ_grid):
    κ_temp = np.array([[0.0, k],
                       [0.30, 0.0]])
    A_temp = build_generator(U, r, κ_temp)
    ρ_grid[n], _ = principal_eigenpair(A_temp)
```

(continues on next page)

(continued from previous page)

```
fig, ax = plt.subplots()
ax.plot(κ_grid, ρ_grid, lw=2)
ax.axhline(ρ, color="black", ls="--", lw=1)
ax.axvline(0, color="black", ls=":", lw=1)
ax.set_xlabel("jump log multiplier for recession to boom")
ax.set_ylabel("principal eigenvalue")
plt.show()
```

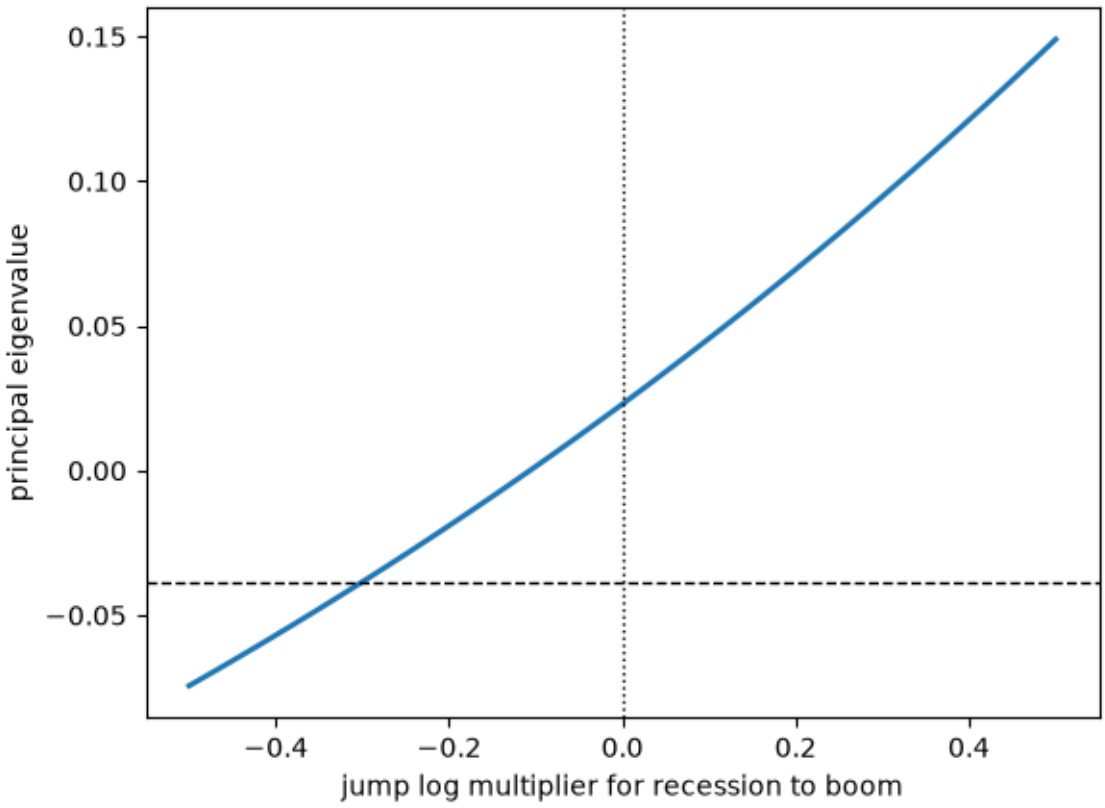


Fig. 127.2: Jumps and the long-run growth rate

The principal eigenvalue is monotonically increasing in the recession-to-boom log multiplier: as that multiplier rises, M jumps less downward (or more upward) on good news, which mechanically pushes ρ up.

The economically sensible SDF region is to the left of zero, where the multiplier is negative.

127.7 The affine diffusion example

We now move to a continuous-state model.

We will use a two-factor affine specification that captures the two main empirical features of asset returns:

- **stochastic volatility**, where the dispersion of shocks is itself a state variable, and
- **predictable growth**, where a small, persistent state variable shifts expected growth rates.

This is the kind of state process used in long-run risk models like Bansal and Yaron [2004].

We work with two independent state components: a Feller square-root process X^f (stochastic volatility) and an Ornstein-Uhlenbeck process X^o (predictable growth):

$$\begin{aligned} dX_t^f &= \xi_f(\bar{x}_f - X_t^f)dt + \sqrt{X_t^f}\sigma_f dB_t^f, \\ dX_t^o &= \xi_o(\bar{x}_o - X_t^o)dt + \sigma_o dB_t^o. \end{aligned} \tag{127.14}$$

The parameters $\xi_f, \xi_o > 0$ are mean-reversion speeds, $\bar{x}_f, \bar{x}_o$ are the unconditional means, and σ_f, σ_o are diffusion coefficients.

The OU process X^o is the continuous-time analogue of the AR(1) process studied in AR(1) Processes, and continuous-time linear asset-pricing models in the same family are developed in *Affine Models of Asset Prices*.

We follow a sign convention with $\sigma_o > 0$ and $\sigma_f < 0$: a positive B^f shock then *reduces* volatility, in line with the empirical "leverage effect."

Now consider a multiplicative functional $M = \exp(A)$ with affine parameters:

$$\begin{aligned} A_t = \bar{\beta}t + \int_0^t \beta_f X_s^f \, ds + \int_0^t \beta_o X_s^o \, ds \\ + \int_0^t \sqrt{X_s^f}\, \gamma_f \, dB_s^f + \int_0^t \gamma_o \, dB_s^o. \end{aligned} \tag{127.15}$$

So the drift of A is affine in the state ($\bar{\beta} + \beta_f X^f + \beta_o X^o$), the Brownian loadings are constant in the B^o direction and proportional to $\sqrt{X^f}$ in the B^f direction.

127.7.1 Why exponential-affine eigenfunctions work

When the state is affine and the drift of A is affine, applying the generator to an exponential-affine function $\phi(x^f, x^o) = \exp(c_f x^f + c_o x^o)$ returns another exponential-affine function.

This closure property turns the eigenvalue equation $\mathbb{A}\phi = \rho\phi$ into a small system of algebraic equations in (c_f, c_o, ρ).

Definition 127.7.1 (Exponential-Affine Eigenfunction)

An eigenfunction ϕ of $\mathbb{A}$ on a state space $\mathcal{D}_0 \subseteq \mathbb{R}^n$ is **exponential-affine** if

$$\phi(x) = \exp(c_0 + c^\top x) \qquad \text{for all } x \in \mathcal{D}_0,$$

for some constant $c_0 \in \mathbb{R}$ and vector $c \in \mathbb{R}^n$.

Substituting $\phi(x^f, x^o) = \exp(c_f x^f + c_o x^o)$ into the generator formula (127.8) and matching coefficients of x^f, x^o, and the constant term gives

$$0 = \beta_f + \frac{\gamma_f^2}{2} + c_f(\gamma_f \sigma_f - \xi_f) + c_f^2 \frac{\sigma_f^2}{2}, \tag{127.16}$$

and

$$c_o = \frac{\beta_o}{\xi_o}. \tag{127.17}$$

The two candidate values for c_f are

$$c_f = \frac{\xi_f - \gamma_f \sigma_f \pm \sqrt{(\xi_f - \gamma_f \sigma_f)^2 - \sigma_f^2 (2\beta_f + \gamma_f^2)}}{\sigma_f^2}. \tag{127.18}$$

The eigenvalue is then determined by matching the constant term:

$$\rho = \bar{\beta} + \frac{\gamma_o^2}{2} + c_f \xi_f \bar{x}_f + c_o(\xi_o \bar{x}_o + \gamma_o \sigma_o) + c_o^2 \frac{\sigma_o^2}{2}. \tag{127.19}$$

Equation (127.18) gives two candidate values of c_f, and we need to know which one is the principal eigenfunction.

This is where stochastic stability (*Definition 127.5.6*) does real work.

Under the twisted measure, the drift of X^f is

$$\xi_f(\bar{x}_f - x^f) + x^f \sigma_f(\gamma_f + c_f \sigma_f),$$

so the mean-reversion coefficient becomes

$$\xi_f - \sigma_f(\gamma_f + c_f \sigma_f).$$

If this is positive, the twisted square-root process stays stationary; if it is negative, the twisted process is explosive and the eigenfunction is not the long-run-relevant one.

So we **pick the root that keeps the twisted process mean-reverting**, exactly the way stochastic stability selects the principal eigenfunction in the abstract theory.

```
def solve_affine_eigenfunction(params):
    """Solve the exponential-affine eigenvalue problem."""
    ξ_f = params["ξ_f"]
    xbar_f = params["xbar_f"]
    σ_f = params["σ_f"]
    ξ_o = params["ξ_o"]
    xbar_o = params["xbar_o"]
    σ_o = params["σ_o"]
    β_bar = params["β_bar"]
    β_f = params["β_f"]
    β_o = params["β_o"]
    γ_f = params["γ_f"]
    γ_o = params["γ_o"]

    co = β_o / ξ_o

    disc = ((ξ_f - γ_f * σ_f) ** 2
            - σ_f ** 2 * (2 * β_f + γ_f ** 2))

    if disc < 0:
        raise ValueError("No real affine eigenfunction for these parameters.")

    root = np.sqrt(disc)
    cf_roots = (
        ((ξ_f - γ_f * σ_f) - root) / σ_f ** 2,
```

(continues on next page)

(continued from previous page)

```
        ((ξ_f - γ_f * σ_f) + root) / σ_f ** 2
    )

    def mean_reversion(cf):
        return ξ_f - σ_f * (γ_f + cf * σ_f)

    for cf in cf_roots:
        mr = mean_reversion(cf)
        if mr > 0:
            break
    else:
        raise ValueError("Neither root gives a stationary twisted process.")

    ρ = (β_bar
         + γ_o ** 2 / 2
         + cf * ξ_f * xbar_f
         + co * (ξ_o * xbar_o + γ_o * σ_o)
         + co ** 2 * σ_o ** 2 / 2)

    return cf, co, ρ, mr
```

127.7.2 A Breeden SDF

To make things concrete, we now plug in a specific stochastic discount factor: the Breeden CRRA consumption-based SDF.

Breeden [1979] derived the continuous-time SDF for an investor with time-separable CRRA preferences over a consumption stream.

We specify log consumption with the same affine state X^o driving expected growth and $\sqrt{X^f}$ driving volatility:

$$dc_t = X_t^o\,dt + \sqrt{X_t^f}\,\vartheta_f\,dB_t^f + \vartheta_o\,dB_t^o.$$

With time-separable CRRA utility (risk aversion a) and subjective discount rate b, the stochastic discount factor is

$$S_t = \exp(-bt - a(c_t - c_0)),$$

i.e. an exponential of $-b$ times time minus a times log consumption growth.

Reading off the additive functional coefficients gives

$$\bar{\beta}^s = -b, \quad \beta_f^s = 0, \quad \beta_o^s = -a, \quad \gamma_f^s = -a\vartheta_f, \quad \gamma_o^s = -a\vartheta_o. \tag{127.20}$$

We will use these parameters in the numerical example below.

127.7.3 Recursive preferences (optional)

A famous limitation of CRRA preferences is that they conflate risk aversion with the elasticity of intertemporal substitution.

Recursive preferences Epstein and Zin [1989], Kreps and Porteus [1978] separate the two, and add a forward-looking continuation-value term to the discount-factor expression.

This is what powers the equity-premium results in Bansal and Yaron [2004].

A QuantEcon lecture that studies long-run dynamics under recursive preferences in a different setting is *Survival and Long-Run Dynamics under Recursive Preferences*.

This section derives the SDF coefficients for the unit-elasticity recursive specification.

You can skip on a first read and come back later — the numerical example uses the simpler Breeden parameters above.

For the unit-elasticity recursive specification, conjecture a continuation value of the form

$$W_t = \frac{1}{1-a}\exp\left[(1-a)(w_f X_t^f + w_o X_t^o + c_t + \bar{w})\right], \qquad a > 1. \tag{127.21}$$

Matching the local mean of this continuation value gives

$$\begin{aligned} 0 &= -\xi_f w_f + \frac{(1-a)\sigma_f^2}{2} w_f^2 + (1-a)\vartheta_f \sigma_f w_f + \frac{(1-a)\vartheta_f^2}{2} - b w_f, \\ 0 &= -\xi_o w_o + 1 - b w_o, \\ b\bar{w} &= \xi_f \bar{x}_f w_f + \xi_o \bar{x}_o w_o + \frac{(1-a)\sigma_o^2}{2} w_o^2 + (1-a)\vartheta_o \sigma_o w_o + \frac{(1-a)\vartheta_o^2}{2}. \end{aligned} \tag{127.22}$$

The relevant solution has $w_o = 1/(\xi_o + b)$, selects the stable root of the quadratic for w_f, and then determines $\bar{w}$ from the constant equation.

The stochastic discount factor is the product $S_t = \exp(A_t^B)\exp(A_t^W)$, where the Breeden-like logarithmic component is

$$A_t^B = -bt - \int_0^t X_s^o ds - \int_0^t \sqrt{X_s^f}\vartheta_f dB_s^f - \int_0^t \vartheta_o dB_s^o$$

The continuation-value martingale component is

$$\begin{aligned} A_t^W = (1-a)\int_0^t \sqrt{X_s^f}(\vartheta_f + w_f\sigma_f) dB_s^f + (1-a)\int_0^t (\vartheta_o + w_o\sigma_o) dB_s^o \\ - \frac{(1-a)^2}{2}\int_0^t X_s^f(\vartheta_f + w_f\sigma_f)^2 ds - \frac{(1-a)^2}{2}(\vartheta_o + w_o\sigma_o)^2 t. \end{aligned} \tag{127.23}$$

Thus the recursive-utility SDF has affine parameters

$$\begin{aligned} \bar{\beta}^s &= -b - \frac{(1-a)^2}{2}(\vartheta_o + w_o\sigma_o)^2, \\ \beta_f^s &= -\frac{(1-a)^2}{2}(\vartheta_f + w_f\sigma_f)^2, \\ \beta_o^s &= -1, \\ \gamma_f^s &= -a\vartheta_f - (a-1)w_f\sigma_f, \\ \gamma_o^s &= -a\vartheta_o - (a-1)w_o\sigma_o. \end{aligned} \tag{127.24}$$

The local Brownian risk prices are therefore $\sqrt{x^f}[a\vartheta_f + (a-1)w_f\sigma_f]$ for B^f exposure and $a\vartheta_o + (a-1)w_o\sigma_o$ for B^o exposure.

The numerical example below uses the simpler Breeden specification, but the same operator calculation applies once the SDF parameters are replaced by (127.24).

127.7.4 Numerical example

Let's set up parameters and solve for the principal eigenpair.

We use parameters in the standard long-run-risk neighbourhood: a mean-reverting volatility factor X^f with mean 0.04, a slower-moving predictable-growth factor X^o with mean 0.02, risk aversion $a = 4$, and a time discount rate $b = 0.03$.

```
params_state = {
    "ξ_f": 0.70,
    "xbar_f": 0.04,
    "σ_f": -0.20,
    "ξ_o": 0.50,
    "xbar_o": 0.02,
    "σ_o": 0.01,
}

a = 4.0
b = 0.03
ϑ_f = 0.06
ϑ_o = 0.02

params_sdf = {
    **params_state,
    "β_bar": -b,
    "β_f": 0.0,
    "β_o": -a,
    "γ_f": -a * ϑ_f,
    "γ_o": -a * ϑ_o,
}

cf_s, co_s, ρ_s, mr_s = solve_affine_eigenfunction(params_sdf)

print("principal eigenfunction ψ(xf, xo) = exp(cf xf + co xo)")
print(f"cf = {cf_s:.6f}")
print(f"co = {co_s:.6f}")
print(f"ρ = {ρ_s:.6f}")
print(f"twisted mean-reversion coefficient for Xf = {mr_s:.6f}")
print(f"long-run zero-coupon yield = {-ρ_s:.4f}")
```

```
principal eigenfunction ψ(xf, xo) = exp(cf xf + co xo)
cf = 0.044232
co = -8.000000
ρ = -0.095962
twisted mean-reversion coefficient for Xf = 0.650231
long-run zero-coupon yield = 0.0960
```

The long-run zero-coupon yield $-\rho_s$ represents the asymptotic decay rate in the SDF expectation $\mathbb{E}[S_t]$.

We can also check that the rejected root for c_f would have produced a non-stationary twisted process, a clear example of stochastic stability selecting one of two algebraically valid eigenfunctions.

```
ξ_f = params_sdf["ξ_f"]
σ_f = params_sdf["σ_f"]
γ_f = params_sdf["γ_f"]
β_f = params_sdf["β_f"]

disc = ((ξ_f - γ_f * σ_f) ** 2
        - σ_f ** 2 * (2 * β_f + γ_f ** 2))
```

(continues on next page)

(continued from previous page)

```
root = np.sqrt(disc)

cf_candidates = np.array([
    ((ξ_f - γ_f * σ_f) - root) / σ_f ** 2,
    ((ξ_f - γ_f * σ_f) + root) / σ_f ** 2
])

for cf in cf_candidates:
    mr = ξ_f - σ_f * (γ_f + cf * σ_f)
    print(f"cf = {cf:8.4f}, twisted mean reversion = {mr:8.4f}")
```

```
cf =   0.0442, twisted mean reversion =   0.6502
cf =  32.5558, twisted mean reversion =  -0.6502
```

127.7.5 The martingale component

Now that we have the eigenpair (ρ, ϕ), we can write down all three pieces of the factorization (127.1) explicitly.

The martingale component $\hat{M}_t = \exp(\hat{A}_t)$ defined in (127.10) has log

$$\begin{aligned}\hat{A}_t = &\int_0^t \sqrt{X_s^f}(\gamma_f + c_f\sigma_f)\, dB_s^f + \int_0^t (\gamma_o + c_o\sigma_o)\, dB_s^o \\ &- \frac{1}{2}\int_0^t X_s^f(\gamma_f + c_f\sigma_f)^2\, ds - \frac{1}{2}\int_0^t (\gamma_o + c_o\sigma_o)^2\, ds.\end{aligned}$$

The first line is the Brownian integral (the "exponential martingale" piece); the second is the Itô correction needed to make it a martingale.

Under the twisted measure induced by $\hat{M}$, the drifts of the state variables shift to

$$\begin{aligned}dX_t^f &: \quad \xi_f(\bar{x}_f - X_t^f) + X_t^f\sigma_f(\gamma_f + c_f\sigma_f), \\ dX_t^o &: \quad \xi_o(\bar{x}_o - X_t^o) + \sigma_o(\gamma_o + c_o\sigma_o).\end{aligned}$$

The drift distortions are exactly the Girsanov shifts induced by the Brownian loadings of $\hat{M}$.

Let's now simulate the state and check the factorization numerically.

The first check is the *algebraic identity*: once we define $\hat{M}$ through (127.10), the equation $M_t = \exp(\rho t)\hat{M}_t\phi(X_0)/\phi(X_t)$ is automatic for any choice of (ρ, ϕ), and the error below is just floating-point round-off.

The second, substantive, check is whether the eigenpair (ρ, ϕ) we solved for really makes $\hat{M}$ a martingale, which we approximate by computing $\mathbb{E}[\hat{M}_t]$ across many simulated paths.

```
def brownian_increments(n, dt, seed=1234):
    """Independent Brownian increments for the simulation."""
    rng = np.random.default_rng(seed)
    draws = rng.normal(scale=np.sqrt(dt), size=(n, 2))
    return draws[:, 0], draws[:, 1]


def simulate_states(params, T=40.0, dt=0.01, seed=1234):
    """Euler simulation of the affine state process."""
    n = int(T / dt)
    t = np.linspace(0, T, n + 1)
```

(continues on next page)

(continued from previous page)

```
    Xf = np.empty(n + 1)
    Xo = np.empty(n + 1)
    dBf, dBo = brownian_increments(n, dt, seed)

    Xf[0] = params["xbar_f"]
    Xo[0] = params["xbar_o"]

    for k, (dBf_k, dBo_k) in enumerate(zip(dBf, dBo)):
        xf = max(Xf[k], 1e-10)
        xo = Xo[k]

        Xf[k + 1] = (xf
                     + params["ξ_f"] * (params["xbar_f"] - xf) * dt
                     + np.sqrt(xf) * params["σ_f"] * dBf_k)
        Xf[k + 1] = max(Xf[k + 1], 1e-10)

        Xo[k + 1] = (xo
                     + params["ξ_o"] * (params["xbar_o"] - xo) * dt
                     + params["σ_o"] * dBo_k)

    return t, Xf, Xo, dBf, dBo


def additive_log_M(params, t, Xf, Xo, dBf, dBo):
    """Additive log functional along a simulated path."""
    dt = t[1] - t[0]
    A = np.zeros_like(t)

    for k, (dBf_k, dBo_k) in enumerate(zip(dBf, dBo)):
        xf = max(Xf[k], 1e-10)
        xo = Xo[k]

        drift = (params["β_bar"]
                 + params["β_f"] * xf
                 + params["β_o"] * xo)

        shock = (np.sqrt(xf) * params["γ_f"] * dBf_k
                 + params["γ_o"] * dBo_k)

        A[k + 1] = A[k] + drift * dt + shock

    return A


t, Xf, Xo, dBf, dBo = simulate_states(params_sdf)
A_log = additive_log_M(params_sdf, t, Xf, Xo, dBf, dBo)

ψ_0 = np.exp(cf_s * Xf[0] + co_s * Xo[0])
ψ_t = np.exp(cf_s * Xf + co_s * Xo)

M = np.exp(A_log)
M_hat = np.exp(-ρ_s * t) * M * ψ_t / ψ_0
transient = ψ_0 / ψ_t

identity_error = np.max(np.abs(M - np.exp(ρ_s * t) * M_hat * transient))
print(f"algebraic identity error = {identity_error:.2e}")
```

```
algebraic identity error = 2.22e-16
```

The error above is up to machine precision, as expected.

Next we estimate $\mathbb{E}[\hat{M}_t \mid X_0 = \bar{x}]$ over a Monte Carlo sample of paths.

If $\hat{M}$ is a martingale, the population mean is exactly 1 at every t, and the sample mean should lie within a few standard errors of 1.

```
def simulate_M_hat(params, ρ, cf, co, n_paths=2000, T=20.0, dt=0.01, seed=2024):
    """Monte Carlo paths of hat M_t along an ensemble of trajectories."""
    rng = np.random.default_rng(seed)
    n = int(T / dt)
    t = np.linspace(0, T, n + 1)
    M_hat_paths = np.empty((n_paths, n + 1))

    for k in range(n_paths):
        seed_k = rng.integers(1, 10**9)
        t_k, Xf_k, Xo_k, dBf_k, dBo_k = simulate_states(
            params, T=T, dt=dt, seed=int(seed_k)
        )
        A_k = additive_log_M(params, t_k, Xf_k, Xo_k, dBf_k, dBo_k)
        ψ_t_k = np.exp(cf * Xf_k + co * Xo_k)
        ψ_0_k = np.exp(cf * Xf_k[0] + co * Xo_k[0])
        M_hat_paths[k] = np.exp(-ρ * t_k) * np.exp(A_k) * ψ_t_k / ψ_0_k

    return t, M_hat_paths


t_mc, M_hat_paths = simulate_M_hat(params_sdf, ρ_s, cf_s, co_s)
M_hat_mean = M_hat_paths.mean(axis=0)
M_hat_se = M_hat_paths.std(axis=0, ddof=1) / np.sqrt(M_hat_paths.shape[0])

print("   t      mean       se      (mean - 1) / se")
for t_check in [1.0, 5.0, 10.0, 20.0]:
    idx = np.argmin(np.abs(t_mc - t_check))
    mean = M_hat_mean[idx]
    se = M_hat_se[idx]
    z = (mean - 1.0) / se
    print(f"{t_mc[idx]:5.2f}   {mean:7.4f}  {se:7.4f}   {z:+6.2f}")
```

```
   t      mean       se      (mean - 1) / se
 1.00    0.9974   0.0038    -0.68
 5.00    0.9928   0.0087    -0.83
10.00    1.0005   0.0129    +0.04
20.00    1.0020   0.0201    +0.10
```

```
fig, axes = plt.subplots(2, 2, figsize=(12, 8))

axes[0, 0].plot(t, Xf)
axes[0, 0].set_ylabel("$X_t^f$")
axes[0, 0].set_xlabel("$t$")

axes[0, 1].plot(t, Xo)
axes[0, 1].set_ylabel("$X_t^o$")
axes[0, 1].set_xlabel("$t$")
```

(continues on next page)

(continued from previous page)

```
axes[1, 0].plot(t, M, label="$M_t$")
axes[1, 0].plot(t, np.exp(ρ_s * t) * M_hat * transient,
                "--", label="factorization")
axes[1, 0].set_ylabel("multiplicative factorization")
axes[1, 0].set_xlabel("$t$")
axes[1, 0].legend()

axes[1, 1].plot(t, np.exp(ρ_s * t), label="$\\exp(\\rho t)$")
axes[1, 1].plot(t, M_hat, label="$\\hat M_t$", alpha=0.8)
axes[1, 1].plot(t, transient, label="$\\phi(X_0)/\\phi(X_t)$", alpha=0.8)
axes[1, 1].set_ylabel("three components")
axes[1, 1].set_xlabel("$t$")
axes[1, 1].legend()

plt.tight_layout()
plt.show()
```

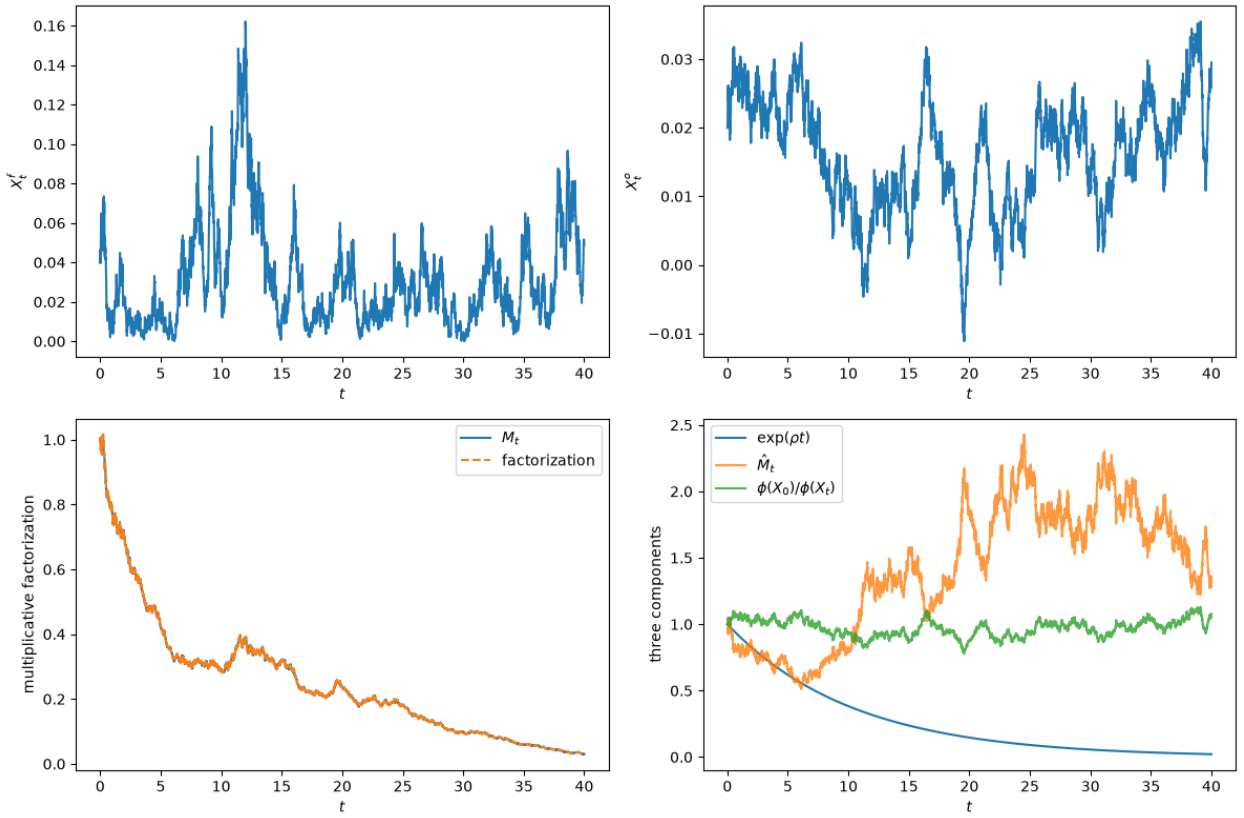


Fig. 127.3: Affine state paths and multiplicative factorization

The figure shows how the three components evolve over time.

127.8 Long-run risk prices

We can now use the factorization to compute long-run analogues of the instantaneous risk prices that arise in standard continuous-time asset pricing.

We can ask:

> If an investor takes on a small exposure to a shock today, how much extra expected return do they need when that return is measured as a long-horizon rate rather than an instantaneous one?

The two answers, local and long-run, need not agree.

A shock that moves a persistent state variable has a small *immediate* effect on the cash flow but a large *cumulative* effect on future expected growth and discounting, which is why the two answers come apart.

So the long-run risk price is the local price *plus a persistence correction.*

The size of the correction depends on the speed of mean reversion.

127.8.1 Defining the prices

Definition 127.8.1 (Local Brownian Risk Price)

The **local Brownian risk price** is the state-dependent vector $-\gamma^s(x)$.

A small Brownian exposure γ_i^v earns a required expected return of $-\gamma_i^v \gamma_i^s$ per unit time, so a unit of exposure γ_i^v is priced at $-\gamma_i^s$.

For the local price we just read off the SDF coefficients.

Definition 127.8.2 (Long-Run Risk Price)

The **long-run risk price** is the marginal change in the long-run required return on a cash flow with respect to a small change in its risk exposure.

When we work with growing cash flows, the long-run required return is $R_\infty = -\rho + \delta$, where ρ is the principal eigenvalue of the GS semigroup and δ is the trend growth rate, held fixed.

Computing the long-run price requires solving the principal eigenvalue problem, which captures how a shock propagates through the persistent state component.

127.8.2 Two frontiers

We will see *two* related ways to vary risk exposure, each leading to a slightly different long-run risk price:

1. **Valuation-functional frontier:**
   - Hold the SDF S fixed and vary the asset's Brownian exposures (γ_f^v, γ_o^v).
   - Use the local pricing restriction to determine the drift β^v, then compute ρ^v for the V-semigroup.
2. **Cash-flow frontier:**
   - Hold the SDF S fixed and vary the cash-flow's growth exposures (γ_f^g, γ_o^g).
   - Set $M = GS$ and compute the principal eigenvalue ρ of the cash-flow valuation semigroup.

In simple log-normal examples, these two frontiers coincide.

But they can differ with stochastic volatility, nonlinear dynamics, or jump risk.

We will work out both types of examples in the affine model below.

127.8.3 Stochastic discount factor decomposition

Before getting into risk prices, a natural benchmark is the case $M = S$: applying the factorization (127.1) directly to the SDF itself.

The factorization becomes

$$S_t = \exp(\rho t)\hat{M}_t \frac{\phi(X_0)}{\phi(X_t)}.$$

This is the **permanent-transitory decomposition** of Alvarez and Jermann [2005], linked now to a concrete eigenfunction construction.

The factor $\exp(\rho t)$ is the deterministic trend in the SDF and the martingale $\hat{M}$ is its random *permanent* component; the state-dependent ratio is *transient* and washes out.

See also

The same spectral decomposition of the pricing operator is the central tool in *The Recovery Theorem*, which uses it to "recover" subjective beliefs from observed prices, and in *Misspecified Recovery*, which examines what goes wrong when the permanent martingale component is mistakenly assumed away.

For a long zero-coupon bond,

$$\exp(-\rho t)\,\mathbb{E}[S_t \mid X_0 = x] \to \phi(x) \int \frac{1}{\phi}\, d\hat{\varsigma},$$

so long-maturity discount bond prices depend on the current state primarily through the eigenfunction ϕ.

127.8.4 Comparison in the affine model

For the affine specification, we can write closed-form expressions for both local and long-run prices of a B^o shock.

The local price is just

$$\text{local price of } B^o = -\gamma_o^s.$$

The long-run price (which we will derive below using the valuation-functional frontier) is

$$\text{long-run price of } B^o = -\gamma_o^s - \frac{\beta_o^s}{\xi_o}\sigma_o. \tag{127.25}$$

The extra term $-(\beta_o^s/\xi_o)\sigma_o$ is the **persistence correction**.

It arises because:

- a B^o shock moves the persistent growth predictor X^o, and
- X^o mean reverts at rate ξ_o, so the cumulative effect of the shock on future SDF growth scales like $1/\xi_o$.

As ξ_o shrinks, persistence grows and the long-run price diverges from the local one, which is the central economic content of long-run risk models.

The local price of B^f exposure is state dependent (it scales with $\sqrt{X_t^f}$), and the long-run price of B^f exposure is nonlinear, since c_f is the root of a quadratic.

```
γ_s_o = params_sdf["γ_o"]
β_s_o = params_sdf["β_o"]
ξ_o = params_sdf["ξ_o"]
σ_o = params_sdf["σ_o"]

local_price_o = -γ_s_o
long_run_price_o = -γ_s_o - (β_s_o / ξ_o) * σ_o

print(f"local price of B^o exposure    = {local_price_o:.4f}")
print(f"long-run price of B^o exposure = {long_run_price_o:.4f}")
```

```
local price of B^o exposure    = 0.0800
long-run price of B^o exposure = 0.1600
```

The next cell illustrates how persistence changes the wedge between local and long-run prices.

```
ξ_o_grid = np.array([0.10, 0.20, 0.50, 1.00, 2.00, 5.00])
local_grid = np.full_like(ξ_o_grid, local_price_o)
long_grid = -γ_s_o - (β_s_o / ξ_o_grid) * σ_o

fig, ax = plt.subplots()
ax.plot(ξ_o_grid, local_grid, "--", lw=2, label="local")
ax.plot(ξ_o_grid, long_grid, "o-", lw=2, label="long-run")
ax.set_xscale("log")
ax.set_xlabel("mean-reversion speed $\\xi_o$")
ax.set_ylabel("risk price")
ax.legend()
plt.show()
```

As we can see, the long-run price of B^o exposure is sensitive to the mean-reversion speed ξ_o, but the local price is a constant.

127.8.5 Changing valuation functionals

Now we work out the long-run risk price formula by varying the asset's exposure, the **valuation-functional frontier** introduced above.

We hold the SDF S fixed and pick Brownian exposures (γ_f^v, γ_o^v) for the asset return, parameterizing the valuation functional as

$$A_t^v = \bar{\beta}^v t + \int_0^t \beta_f^v X_s^f ds + \int_0^t \beta_o^v X_s^o ds + \int_0^t \sqrt{X_s^f} \gamma_f^v dB_s^f + \int_0^t \gamma_o^v dB_s^o.$$

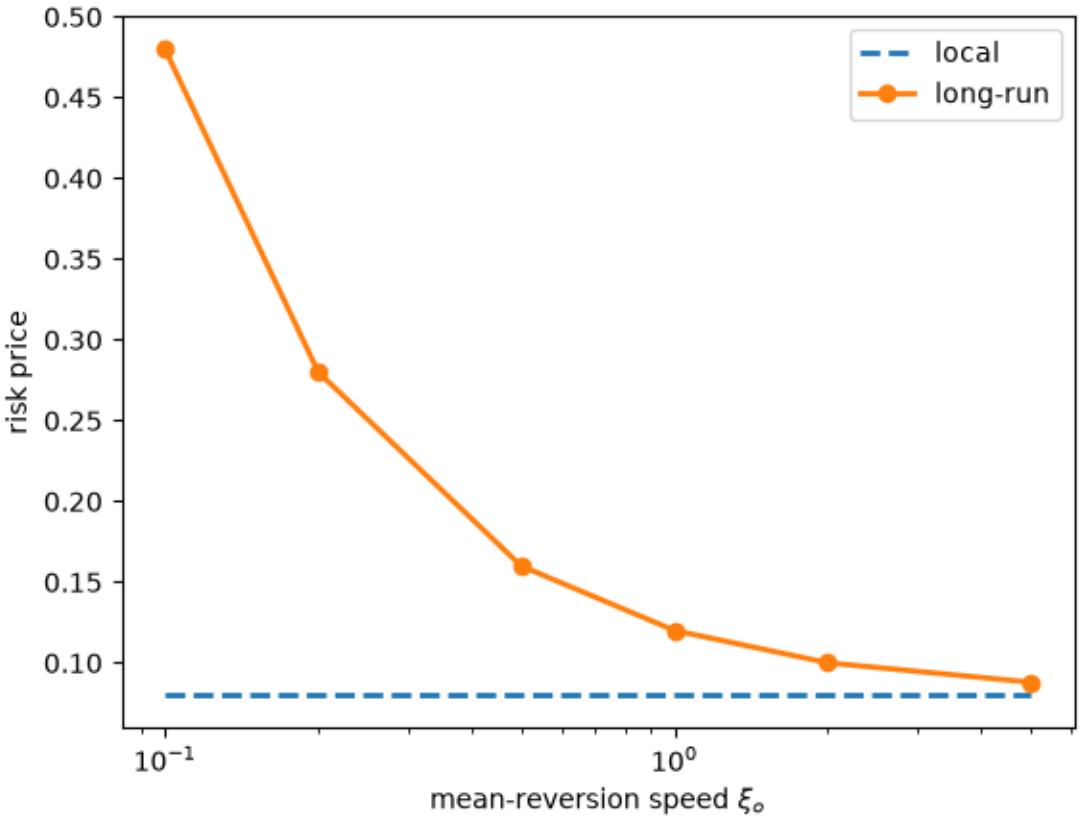


Fig. 127.4: Persistence and long-run risk prices

The martingale restriction on VS determines the drift coefficients from the chosen Brownian exposures (γ_f^v, γ_o^v):

$$\begin{aligned}\bar{\beta}^v &= -\bar{\beta}^s - \frac{1}{2}(\gamma_o^s + \gamma_o^v)^2,\\ \beta_f^v &= -\beta_f^s - \frac{1}{2}(\gamma_f^s + \gamma_f^v)^2,\\ \beta_o^v &= -\beta_o^s.\end{aligned} \tag{127.26}$$

Applying the affine eigenvalue formula to $M = V$ gives

$$\begin{aligned}\rho^v &= \bar{\beta}^v + \frac{(\gamma_o^v)^2}{2} + c_f^v \xi_f \bar{x}_f + c_o^v(\xi_o \bar{x}_o + \gamma_o^v \sigma_o) + (c_o^v)^2 \frac{\sigma_o^2}{2}\\ &= -\bar{\beta}^s - \frac{(\gamma_o^s)^2}{2} - \gamma_o^s \gamma_o^v + c_f^v \xi_f \bar{x}_f + c_o^v(\xi_o \bar{x}_o + \gamma_o^v \sigma_o) + (c_o^v)^2 \frac{\sigma_o^2}{2},\end{aligned} \tag{127.27}$$

where $c_o^v = \beta_o^v/\xi_o = -\beta_o^s/\xi_o$ and c_f^v solves the same quadratic equation as in (127.16) with (β_f, γ_f) replaced by (β_f^v, γ_f^v).

Holding γ_f^v fixed, differentiating (127.27) with respect to γ_o^v gives the long-run valuation-functional price of B^o exposure:

$$\frac{\partial \rho^v}{\partial \gamma_o^v} = -\gamma_o^s + c_o^v \sigma_o = -\gamma_o^s - \frac{\beta_o^s}{\xi_o}\sigma_o. \tag{127.28}$$

This matches the formula (127.25) we previewed above, namely the local price plus the persistence correction $-(\beta_o^s/\xi_o)\sigma_o$.

Let's verify the formula numerically by finite-differencing the eigenvalue computation.

```
def valuation_params_from_exposure(γ_v_o, γ_v_f=0.0):
    """Affine parameters for a valuation functional V."""
    p = dict(params_sdf)
    p.update({
        "β_bar": (-params_sdf["β_bar"]
                  - 0.5 * (params_sdf["γ_o"] + γ_v_o) ** 2),
        "β_f": (-params_sdf["β_f"]
                 - 0.5 * (params_sdf["γ_f"] + γ_v_f) ** 2),
        "β_o": -params_sdf["β_o"],
        "γ_f": γ_v_f,
        "γ_o": γ_v_o,
    })
    return p


def valuation_eigenvalue_for_exposure(γ_v_o, γ_v_f=0.0):
    """Principal eigenvalue for the valuation functional frontier."""
    p = valuation_params_from_exposure(γ_v_o, γ_v_f)
    _, _, ρ, _ = solve_affine_eigenfunction(p)
    return ρ
```

```
γ_v_o_grid = np.linspace(-0.5, 0.5, 101)
ρ_v_grid = np.array([
    valuation_eigenvalue_for_exposure(g) for g in γ_v_o_grid
])

fig, ax = plt.subplots()
ax.plot(γ_v_o_grid, ρ_v_grid, lw=2)
ax.set_xlabel("valuation exposure $\\gamma_o^v$")
ax.set_ylabel("principal eigenvalue $\\rho^v$")
plt.show()
```

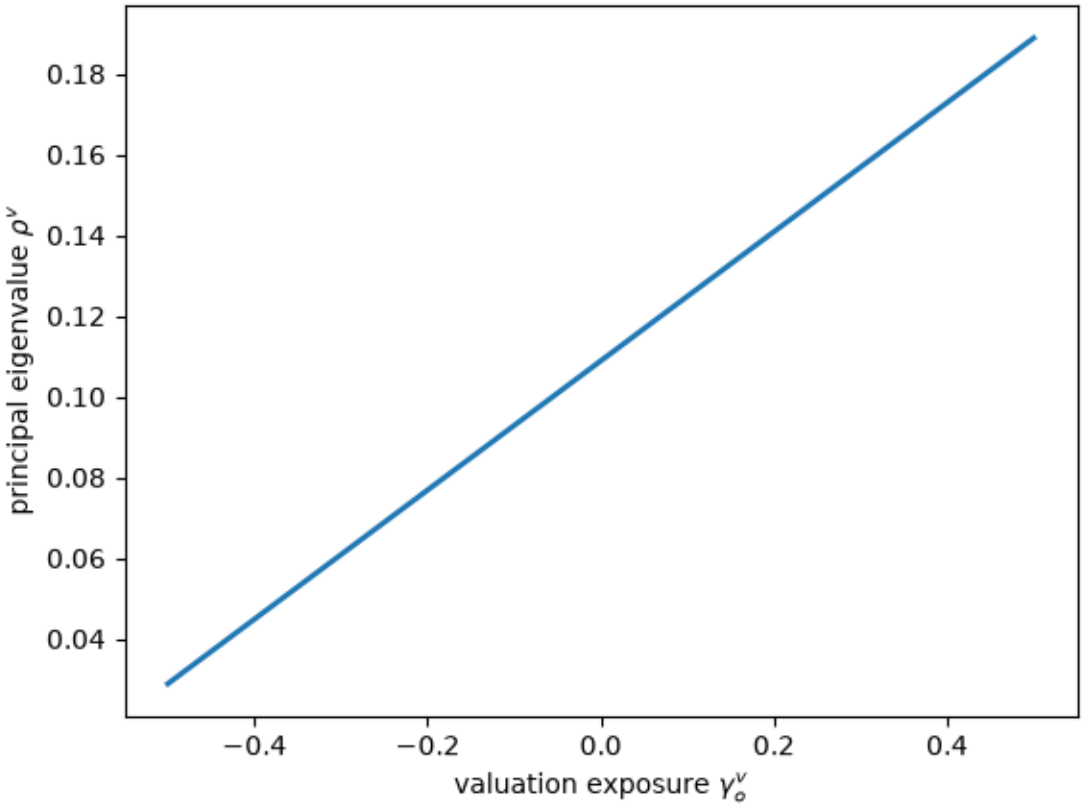


Fig. 127.5: Changing valuation functionals

```
valuation_slope = (
    valuation_eigenvalue_for_exposure(0.001)
    - valuation_eigenvalue_for_exposure(-0.001)
) / 0.002

print(f"finite-difference slope = {valuation_slope:.6f}")
print(f"formula                 = {long_run_price_o:.6f}")
```

```
finite-difference slope = 0.160000
formula                 = 0.160000
```

127.8.6 Changing cash-flow risk

The second long-run frontier varies the cash-flow's exposure to risk rather than the return.

We pick a growth functional $G_t = \exp(A_t^g)$ with the affine parameterization

$$\begin{aligned} A_t^g = \delta t + \int_0^t \sqrt{X_s^f}\gamma_f^g dB_s^f + \int_0^t \gamma_o^g dB_s^o \\ - \frac{1}{2}\int_0^t \left[X_s^f(\gamma_f^g)^2 + (\gamma_o^g)^2\right] ds. \end{aligned} \tag{127.29}$$

The last line is the Itô compensator that makes $\exp(A_t^g - \delta t) = \hat{G}_t$ a *local* martingale, with δ the constant trend growth rate.

Stochastic stability of the growth-twisted process needs three conditions.

The **Feller-type nonattainment** inequality

$$2(\xi_f + \sigma_f\gamma_f^g)\bar{x}_f \geq \sigma_f^2$$

keeps the twisted X^f from hitting zero.

Mean reversion of the twisted X^f is picked by the same root-selection argument we used for the SDF in (127.18).

$\hat{G}$ itself must be a martingale, the Assumption-6.1 analogue for the growth twist.

The Feller inequality is necessary but not sufficient on its own.

> **Note**
>
> This Feller restriction is a concrete instance of a general point we flagged earlier: changing growth risk can violate stability and invalidate the long-run approximation, so the choice of (γ_f^g, γ_o^g) isn't free.

To price the cash flow $D_t = D_0 G_t \psi(X_t)$, use the semigroup generated by $M = GS$.

The combined affine parameters are

$$\begin{aligned}
\bar{\beta} &= \bar{\beta}^s + \delta - \frac{(\gamma_o^g)^2}{2}, \\
\beta_f &= \beta_f^s - \frac{(\gamma_f^g)^2}{2}, \\
\beta_o &= \beta_o^s, \\
\gamma_f &= \gamma_f^s + \gamma_f^g, \\
\gamma_o &= \gamma_o^s + \gamma_o^g.
\end{aligned}$$

Let ρ be the principal eigenvalue of this GS semigroup.

Then $-\rho$ is the long-run decay rate in value, and

$$R_\infty = -\rho + \delta$$

is the asymptotic required return net of the cash-flow growth rate.

```
def required_return_for_growth_exposure(γ_g_o, γ_g_f=0.0, δ=0.02):
    """Long-run required return for a cash-flow growth exposure."""
    p = dict(params_sdf)
    p.update({
        "β_bar": params_sdf["β_bar"] + δ - 0.5 * γ_g_o ** 2,
        "β_f": params_sdf["β_f"] - 0.5 * γ_g_f ** 2,
        "β_o": params_sdf["β_o"],
        "γ_f": params_sdf["γ_f"] + γ_g_f,
        "γ_o": params_sdf["γ_o"] + γ_g_o,
    })

    _, _, ρ, _ = solve_affine_eigenfunction(p)
    return -ρ + δ
```

```
γ_g_o_grid = np.linspace(-0.5, 0.5, 101)
required_returns = np.array([
    required_return_for_growth_exposure(g) for g in γ_g_o_grid
])

local_line = (required_return_for_growth_exposure(0.0)
              + local_price_o * γ_g_o_grid)

fig, ax = plt.subplots()
ax.plot(γ_g_o_grid, required_returns, lw=2,
        label="long-run required return")
ax.plot(γ_g_o_grid, local_line, "--", lw=2,
        label="local slope")
ax.set_xlabel("cash-flow exposure $\\gamma_o^g$")
ax.set_ylabel("rate of return")
ax.legend()
plt.show()
```

The slope of the long-run line is the risk price in (127.25), and the dashed line shares the same value at $\gamma_o^g = 0$ but with slope equal to the local Brownian risk price $-\gamma_o^s$.

The gap between the two slopes is the *persistence correction*; the dashed line is a slope comparator, not the actual Breeden local expected-return frontier (which is state dependent in X^f and X^o).

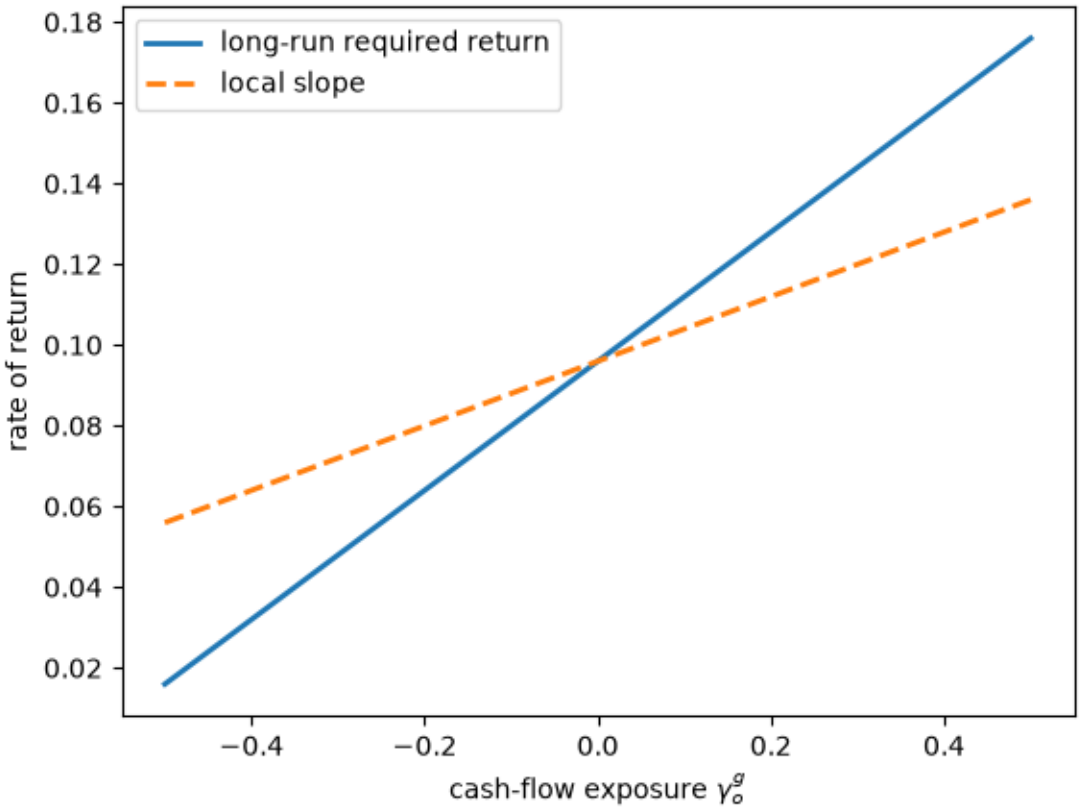


Fig. 127.6: Local and long-run pricing of persistent growth risk

```
finite_difference = (
    required_return_for_growth_exposure(0.001)
    - required_return_for_growth_exposure(-0.001)
) / 0.002

print(f"finite-difference slope = {finite_difference:.6f}")
print(f"formula                 = {long_run_price_o:.6f}")
```

```
finite-difference slope = 0.160000
formula                 = 0.160000
```

127.9 Assumptions behind the scenes

The examples above make the eigenfunction calculation look routine.

For finite-state chains and the affine model, it really is; Perron-Frobenius theory and closed-form algebra handle every requirement.

But in a general state space, three things can go wrong, each corresponding to one of the assumptions we have been carrying.

The rest of this section examines each in turn.

127.9.1 Issue 1: $\hat{M}$ might fail to be a martingale

A positive eigenfunction ϕ gives us a candidate martingale $\hat{M}$ from (127.10), but $\hat{M}$ is only a nonnegative local martingale, hence a supermartingale.

A supermartingale is not enough to define a probability measure: we need $\mathbb{E}[\hat{M}_t] = 1$, i.e. a genuine martingale, the content of Assumption 6.1 in Hansen and Scheinkman [2009].

A standard way to verify this is a two-sided **Girsanov construction**: write the drift and jump distortion induced by $\hat{M}$, check that the distorted Markov process is well-behaved, and verify that the reverse density (the inverse of $\hat{M}$) is locally integrable.

127.9.2 Issue 2: the twisted process might fail to be stable

Even with $\hat{M}$ a martingale, the long-run limit (127.12) requires that the twisted process actually settles into a steady state.

This is where stochastic stability (our trio of stationary distribution, irreducibility of the skeleton, and Harris recurrence) does real work.

The affine example illustrates this concretely, since we *rejected* one of the two algebraically valid eigenfunctions because it implied an explosive twisted square-root process.

127.9.3 Issue 3: a principal eigenfunction might not exist

In a general state space, even *existence* of a strictly positive eigenfunction is not automatic.

A standard sufficient condition starts with a **Lyapunov-type drift bound**: there is a function $V \geq 1$ on the state space and a constant a_0 such that

$$\frac{\mathbb{A}V}{V} \leq a_0.$$

Roughly: V doesn't grow too fast under the semigroup.

With this in hand, for any $\alpha > a_0$ define the **resolvent operator**

$$F_\alpha \psi(x) = \int_0^\infty \exp(-\alpha t)\, \mathbb{E}\left[M_t \, \frac{V(X_t)}{V(x)} \, \psi(X_t) \,\middle|\, X_0 = x \right] dt. \tag{127.30}$$

F_α is the Laplace transform of the semigroup generated by the *rescaled* multiplicative functional $M_t V(X_t)/V(X_0)$.

The existence proof then proceeds in three steps:

1. **Irreducibility for the resolvent.** There exists a reference measure ν such that $F_\alpha \mathbf{1}_\Lambda(x) > 0$ for every x whenever $\nu(\Lambda) > 0$, so the resolvent doesn't "miss" any region of state space.
2. **Nummelin minorization.** Irreducibility yields a *bounded nonnegative function* s on the state space, with $\int s \, d\nu > 0$, such that for every nonnegative ψ,

   $$F_\alpha \psi(x) \;\geq\; s(x) \int \psi \, d\nu.$$

   The function s (often called the *minorization function*) measures how strongly the resolvent dominates a fixed reference measure ν.
3. **Eigenfunction extraction.** The minorization, combined with additional boundedness or strengthened drift assumptions, identifies a critical spectral value for F_α and an associated positive eigenfunction.
   - Inverting the resolvent transform produces a positive eigenfunction for the original semigroup.

These steps are all nontrivial and are out of scope for this lecture.

The details are in Section 9 of Hansen and Scheinkman [2009].

We can summarize the chain of conditions as:

Want	Need
A factorization (127.1)	A positive eigenfunction ϕ
$\widehat{M}$ to define a probability measure	$\widehat{M}$ is a martingale (Assumption 6.1)
The long-run limit (127.12)	Stochastic stability of the twisted process
A unique principal eigenfunction	Stability selects among positive eigenfunctions

In the finite-state case, all four follow from one Perron-Frobenius calculation; in the affine model, they reduce to picking the right root of a quadratic. In general, each must be checked separately.

The full theory in Hansen and Scheinkman [2009] also delivers stronger L^p approximation results and Lyapunov criteria for stochastic stability, which we don't reproduce here.

127.10 Summary

The Hansen-Scheinkman approach studies long-run risk by studying positive eigenfunctions of valuation semigroups.

The main steps are:

1. Model discounting, growth, or cumulated returns by a positive multiplicative functional M.
2. Build the semigroup $\mathbb{M}_t\psi(x) = \mathbb{E}[M_t\psi(X_t) \mid X_0 = x]$.
3. When $M = VS$ is the product of a valuation functional and an SDF, impose the local pricing restriction that VS is a martingale; for cash-flow valuation semigroups $\mathbb{Q}_t = GS$, the pricing restriction is on S alone, and G enters only as a growth twist.
4. Solve the principal eigenvalue problem $\mathbb{A}\phi = \rho\phi$.
5. Use the factorization $M_t = \exp(\rho t)\hat{M}_t\phi(X_0)/\phi(X_t)$.
6. Under the twisted probability measure induced by $\hat{M}$, use stability to obtain long-run approximations of the form (127.12).

In finite-state problems, this is Perron-Frobenius theory.

In affine diffusion problems, exponential-affine eigenfunctions often produce closed-form formulas.

The long-run risk prices that emerge can differ sharply from local risk prices when shocks move persistent state variables.

This persistence effect is the economic channel emphasized in long-run risk asset-pricing models and in the empirical work of Hansen *et al.* [2008].

127.11 Exercises

Exercise 127.11.1

Consider a two-state Markov chain with intensity matrix

$$U = \begin{bmatrix} -\lambda & \lambda \\ \mu & -\mu \end{bmatrix}.$$

Let the multiplicative functional have decay rate $r_1 > 0$ in state 1, decay rate $r_2 = 0$ in state 2, and no jumps.

1. Write down the generator matrix A.
2. Find the principal eigenvalue ρ in terms of λ, μ, and r_1.
3. Verify numerically with $\lambda = 0.4$, $\mu = 0.6$, and $r_1 = 0.05$.
4. Show that $-r_1 < \rho < 0$.

Solution

Here is one solution:

1. The generator is

$$A = \begin{bmatrix} -\lambda - r_1 & \lambda \\ \mu & -\mu \end{bmatrix}.$$

2. The characteristic equation is

$$\rho^2 + (\lambda + \mu + r_1)\rho + \mu r_1 = 0.$$

Hence the principal eigenvalue is the larger root

$$\rho = \frac{-(\lambda + \mu + r_1) + \sqrt{(\lambda + \mu + r_1)^2 - 4\mu r_1}}{2}.$$

3. Numerical verification:

```
λ, μ, r1 = 0.4, 0.6, 0.05

disc = (λ + μ + r1) ** 2 - 4 * μ * r1
ρ_formula = (-(λ + μ + r1) + np.sqrt(disc)) / 2

U_ex = np.array([[-λ, λ],
                 [μ, -μ]])
r_ex = np.array([r1, 0.0])
κ_ex = np.zeros((2, 2))

A_ex = build_generator(U_ex, r_ex, κ_ex)
ρ_numeric, ψ_numeric = principal_eigenpair(A_ex)

print(f"formula  ρ = {ρ_formula:.8f}")
print(f"numeric  ρ = {ρ_numeric:.8f}")
print(f"difference   = {abs(ρ_formula-ρ_numeric):.2e}")
```

```
formula  ρ = -0.02939431
numeric  ρ = -0.02939431
difference   = 0.00e+00
```

4. Let

$$q(x) = x^2 + (\lambda + \mu + r_1)x + \mu r_1.$$

Then $q(0) = \mu r_1 > 0$ and

$$q(-r_1) = -\lambda r_1 < 0.$$

Since the parabola opens upward, one root lies in $(-r_1, 0)$.

The principal eigenvalue is the larger root, so $-r_1 < \rho < 0$.

Exercise 127.11.2

In the affine model, compute the local and long-run prices of exposure to B^o for

$$\xi_o \in \{0.1, 0.2, 0.5, 1, 2, 5\}.$$

Use the formulas

$$\text{local price} = -\gamma_o^s$$

and

$$\text{long-run price} = -\gamma_o^s - \frac{\beta_o^s}{\xi_o}\sigma_o.$$

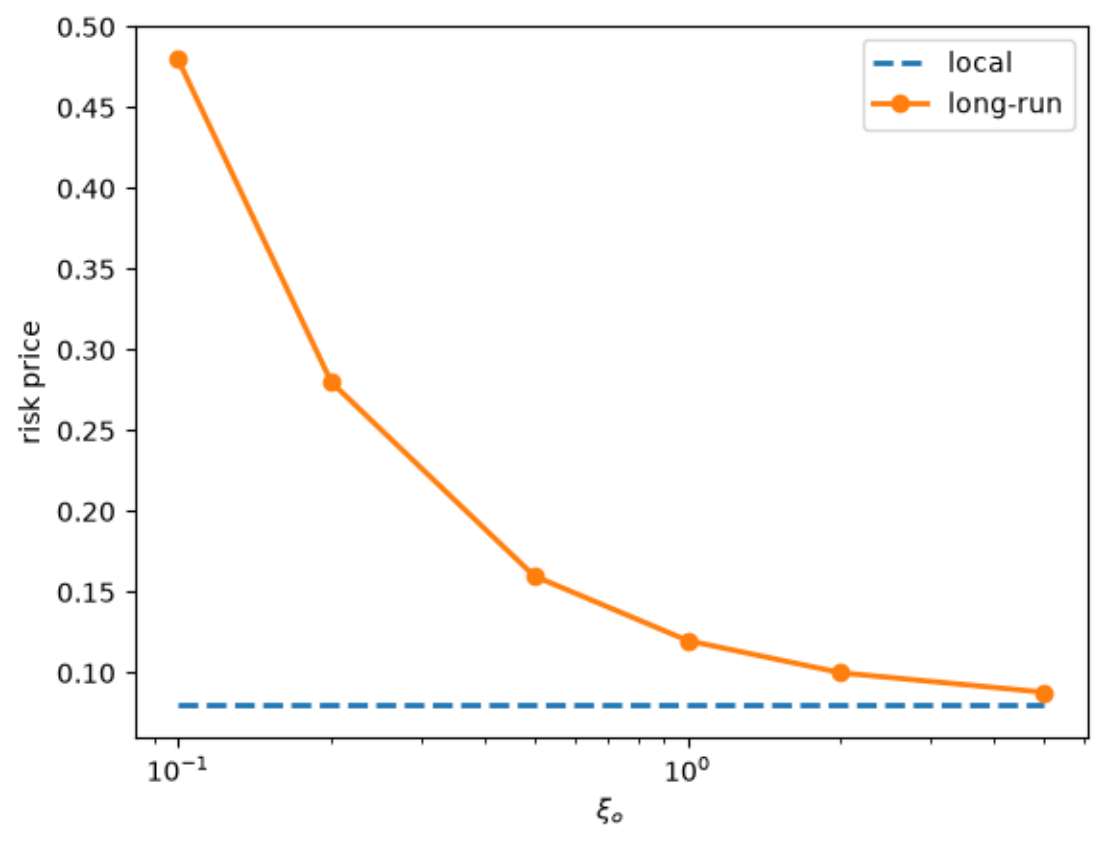
Explain why the two prices converge as $\xi_o \to \infty$.

Solution

Here is one solution:

```
ξ_vals = np.array([0.1, 0.2, 0.5, 1.0, 2.0, 5.0])
local_vals = np.full_like(ξ_vals, -params_sdf["γ_o"])
long_vals = (-params_sdf["γ_o"]
             - (params_sdf["β_o"] / ξ_vals) * params_sdf["σ_o"])

for ξ, lp, lrp in zip(ξ_vals, local_vals, long_vals):
    print(f"ξ_o = {ξ:3.1f}: local = {lp:.4f}, long-run = {lrp:.4f}")

fig, ax = plt.subplots()
ax.plot(ξ_vals, local_vals, "--", lw=2, label="local")
ax.plot(ξ_vals, long_vals, "o-", lw=2, label="long-run")
ax.set_xscale("log")
ax.set_xlabel("$\\xi_o$")
ax.set_ylabel("risk price")
ax.legend()
plt.show()
```

```
ξ_o = 0.1: local = 0.0800, long-run = 0.4800
ξ_o = 0.2: local = 0.0800, long-run = 0.2800
ξ_o = 0.5: local = 0.0800, long-run = 0.1600
ξ_o = 1.0: local = 0.0800, long-run = 0.1200
ξ_o = 2.0: local = 0.0800, long-run = 0.1000
ξ_o = 5.0: local = 0.0800, long-run = 0.0880
```



As ξ_o increases, X^o mean reverts faster.

A shock to B^o then has a shorter-lived effect on future expected growth.

The persistence term $(\beta_o^s/\xi_o)\sigma_o$ converges to zero, so the long-run price converges to the local price.

Exercise 127.11.3

Consider a three-state chain with states {expansion, normal, contraction}, intensity matrix

$$U = \begin{bmatrix} -0.40 & 0.30 & 0.10 \\ 0.20 & -0.50 & 0.30 \\ 0.10 & 0.20 & -0.30 \end{bmatrix},$$

decay-rate vector $r = (0.06, 0.04, 0.01)$, and no jumps in the multiplicative functional. Let $\psi = (3, 1, 2)$.

1. Compute the principal eigenpair (ρ, ϕ) and twisted stationary distribution $\hat{\varsigma}$, and report the theoretical limit

$$\phi \sum_i \frac{\psi_i}{\phi_i} \hat{\varsigma}_i.$$

2. Plot

$$\max_i \left| \exp(-\rho t)(\mathbb{M}_t \psi)_i - \phi_i \sum_j \frac{\psi_j}{\phi_j} \hat{\varsigma}_j \right|$$

on a logarithmic scale.

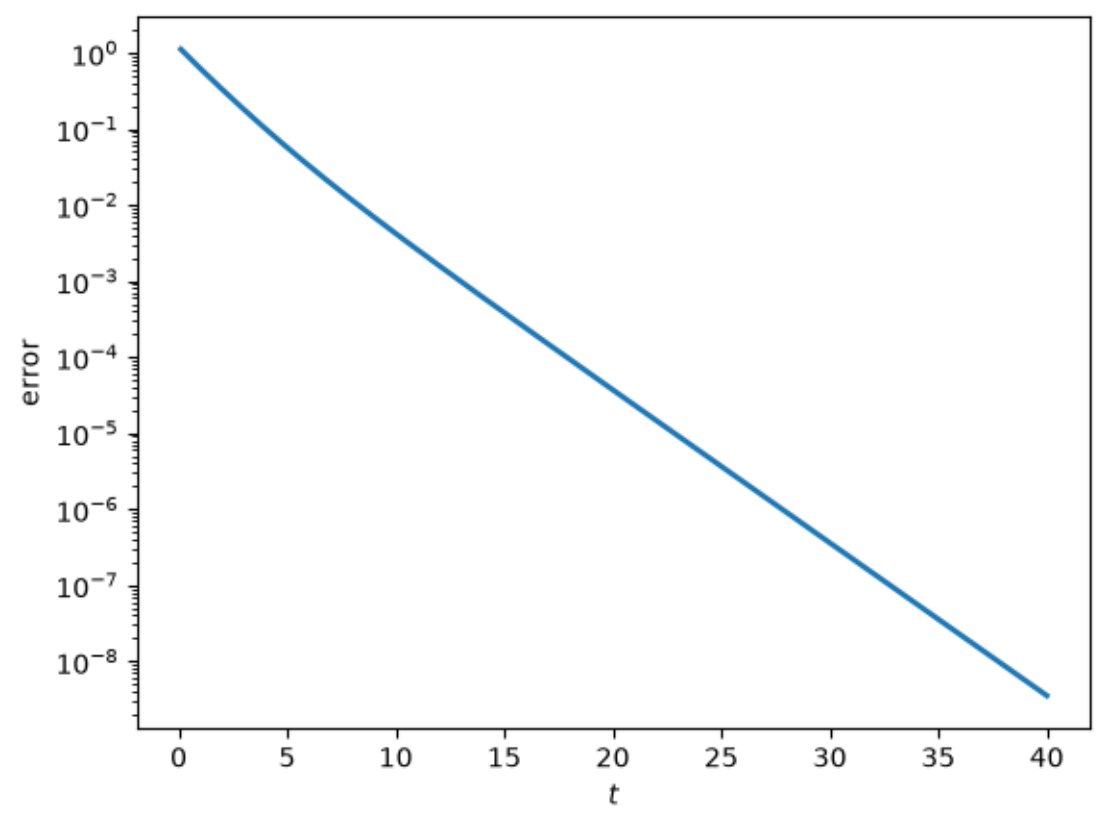
3. Compare the convergence rate to the spectral gap between the largest and second-largest real parts of the eigenvalues of A.

Solution

Here is one solution:

```
state_names = ["expansion", "normal", "contraction"]

U3 = np.array([[-0.40,  0.30,  0.10],
               [ 0.20, -0.50,  0.30],
               [ 0.10,  0.20, -0.30]])

r3 = np.array([0.06, 0.04, 0.01])
κ3 = np.zeros((3, 3))

A3 = build_generator(U3, r3, κ3)
ρ3, ψ3 = principal_eigenpair(A3)
A3_hat = twisted_generator(A3, ρ3, ψ3)
ς3 = stationary_distribution(A3_hat)

ψ = np.array([3.0, 1.0, 2.0])
limit = ψ3 * np.sum((ψ / ψ3) * ς3)

print("limit:")
for name, value in zip(state_names, limit):
    print(f"  {name:11s} {value:.6f}")

t_vals = np.linspace(0.1, 40, 300)
errors = np.array([
    np.max(np.abs(np.exp(-ρ3 * t_val) * expm(t_val * A3) @ ψ - limit))
    for t_val in t_vals
])

eigenvalues = eig(A3, right=False)
real_parts = np.sort(eigenvalues.real)[::-1]
gap = real_parts[0] - real_parts[1]

fig, ax = plt.subplots()
ax.semilogy(t_vals, errors, lw=2)
ax.set_xlabel("$t$")
ax.set_ylabel("error")
plt.show()

print(f"spectral gap = {gap:.6f}")
```

```
limit:
  expansion   1.808490
  normal      1.906080
  contraction 2.021897
spectral gap = 0.462679
```



The normalized semigroup converges at an exponential rate governed by the separation between the dominant eigenvalue and the remaining eigenvalues.

In this finite-state example, that separation is the spectral gap computed above.

Exercise 127.11.4

Derive the local martingale restriction (127.5) from Itô's formula.

Let $M = \exp(A)$ for the additive functional A in (127.3), with parameters (β, γ, κ).

1. Decompose $A_t = A_t^c + A_t^j$ into its continuous and pure-jump parts and write down dA_t^c and the jump magnitudes ΔA_t.
2. Apply Itô's formula for semimartingales to $f(a) = e^a$ to show that

$$dM_t = M_{t-}\, dA_t^c + \tfrac{1}{2}\, M_{t-}\, d\langle A^c, A^c\rangle_t + M_{t-}\big(\exp[\Delta A_t] - 1\big) \quad \text{at jumps.}$$

3. Use $d\langle A^c, A^c\rangle_t = \gamma^\top \gamma\, dt$ and rewrite the jump term as an integral against the random counting measure ζ.
4. Split ζ into its compensator $\eta(dy \mid X_{t-})\, dt$ and the compensated martingale measure $\tilde{\zeta} = \zeta - \eta(dy \mid X_{t-})\, dt$.
5. Collect drift (predictable) and martingale terms and conclude that M is a local martingale iff the drift vanishes at every state, which gives (127.5).

Solution

Here is one solution.

1. From the parameterization (127.3),

$$dA_t^c = \beta(X_t)\,dt + \gamma(X_{t-})^\top\,dB_t, \qquad \Delta A_t = \kappa(X_t, X_{t-}) \text{ at a jump time.}$$

2. For $f(a) = e^a$ we have $f'(a) = f''(a) = e^a$, so $f'(A_{t-}) = f''(A_{t-}) = M_{t-}$.

The (generalized) Itô's formula for a semimartingale gives

$$dM_t = f'(A_{t-})\,dA_t^c + \tfrac{1}{2}\,f''(A_{t-})\,d\langle A^c, A^c\rangle_t + [f(A_t) - f(A_{t-})].$$

Since $A_t = A_{t-} + \Delta A_t$ at a jump, $f(A_t) - f(A_{t-}) = M_{t-}(\exp[\Delta A_t] - 1)$, which is the stated expression.

3. Substituting $d\langle A^c, A^c\rangle_t = \gamma(X_{t-})^\top\gamma(X_{t-})\,dt$ and rewriting the jump contribution as an integral against the random counting measure ζ of (X, A) gives

$$\begin{aligned}\frac{dM_t}{M_{t-}} &= \beta(X_t)\,dt + \gamma(X_{t-})^\top dB_t + \tfrac{1}{2}\,\gamma(X_{t-})^\top\gamma(X_{t-})\,dt \\ &\quad + \int (\exp[\kappa(y, X_{t-})] - 1)\,\zeta(dy, dt).\end{aligned}$$

4. Writing $\zeta = \tilde{\zeta} + \eta(dy \mid X_{t-})\,dt$ separates the jump integral into a martingale and a predictable drift contribution:

$$\int (\exp[\kappa(y, X_{t-})] - 1)\zeta(dy, dt) = \int (\exp[\kappa(y, X_{t-})] - 1)\tilde{\zeta}(dy, dt) + \int (\exp[\kappa(y, X_{t-})] - 1)\eta(dy \mid X_{t-})\,dt.$$

5. Collecting drift and martingale terms,

$$\begin{aligned}\frac{dM_t}{M_{t-}} &= \left[\beta(X_t) + \tfrac{1}{2}\gamma^\top\gamma + \int (\exp[\kappa(y, X_{t-})] - 1)\,\eta(dy \mid X_{t-})\right]dt \\ &\quad + \gamma^\top dB_t + \int (\exp[\kappa(y, X_{t-})] - 1)\,\tilde{\zeta}(dy, dt).\end{aligned}$$

The Brownian and compensated-jump terms are local martingales, so M is a local martingale iff the bracketed drift vanishes for every state x, giving (127.5).

Exercise 127.11.5

Derive the closed-form expression for the extended generator in (127.8).

Take the jump-diffusion setting in which the Markov state satisfies

$$dX_t^c = \xi(X_t)dt + \Gamma(X_t)dB_t$$

between jumps, with $\Sigma = \Gamma\Gamma^\top$ and jump compensator $\eta(dy \mid x)$.

Let $M = \exp(A)$ for the additive functional A in (127.3) and let ϕ be a smooth, strictly positive function.

Recall from *Definition 127.4.1* that $\mathbb{A}\phi = \chi$ if

$$N_t = M_t\phi(X_t) - \phi(X_0) - \int_0^t M_s\chi(X_s)\,ds$$

is a local martingale. The task therefore has two pieces: identify the predictable drift of $M_t\phi(X_t)$ to read off a candidate χ, and verify that the residual N_t really is a local martingale.

(1) Apply Itô's formula to $Y_t = \exp(A_t)\phi(X_t)$ between jumps and show that the continuous part of dY_t has drift

$$M_t\left[(\xi+\Gamma\gamma)^\top\frac{\partial\phi}{\partial x}+\frac{1}{2}\operatorname{trace}\left(\Sigma\frac{\partial^2\phi}{\partial x\partial x^\top}\right)+\left(\beta+\frac{\gamma^\top\gamma}{2}\right)\phi\right]dt.$$

(2) Show that at a jump time t with $X_{t-}=x$ and $X_t=y$,

$$\Delta Y_t = M_{t-}\big[\exp[\kappa(y,x)]\phi(y)-\phi(x)\big],$$

and conclude that the predictable jump compensator contributes drift

$$M_{t-}\int\big\{\exp[\kappa(y,x)]\phi(y)-\phi(x)\big\}\eta(dy\mid x)\,dt.$$

(3) Decompose

$$\exp[\kappa(y,x)]\phi(y)-\phi(x)=\exp[\kappa(y,x)][\phi(y)-\phi(x)]+\big[\exp[\kappa(y,x)]-1\big]\phi(x),$$

combine with part (1) to obtain the full predictable drift coefficient $\chi(x)$, and check that it matches the closed form (127.8).

(4) Identify the continuous Itô integrand from part (1) as a local martingale N^c_t and the compensated jump sum from part (2) as a local martingale N^j_t.

Show that the semimartingale decomposition of $Y_t = M_t\phi(X_t)$ implies

$$M_t\phi(X_t)-\phi(X_0)-\int_0^t M_s\,\chi(X_s)\,ds = N^c_t+N^j_t,$$

which is a local martingale. Conclude via *Definition 127.4.1* that $\mathbb{A}\phi=\chi$.

Solution

Here is one solution:

(1) Set $g(a,x)=e^a\phi(x)$, so that $Y_t=g(A_t,X_t)$.

Between jumps, the continuous parts of A and X are

$$dA^c_t=\beta(X_t)\,dt+\gamma(X_t)^\top dB_t,\qquad dX^c_t=\xi(X_t)\,dt+\Gamma(X_t)\,dB_t,$$

with quadratic covariations

$$d\langle A^c,A^c\rangle_t=\gamma^\top\gamma\,dt,\qquad d\langle X^c,X^c\rangle_t=\Sigma\,dt,\qquad d\langle A^c,X^c\rangle_t=\Gamma\gamma\,dt.$$

The partial derivatives of g satisfy

$$\partial_a g=g,\quad \partial_{aa}g=g,\quad \partial_x g=e^a\frac{\partial\phi}{\partial x},\quad \partial_{ax}g=e^a\frac{\partial\phi}{\partial x},\quad \partial_{xx}g=e^a\frac{\partial^2\phi}{\partial x\partial x^\top}.$$

Itô's formula yields a continuous local martingale N^c_t (the dB_t part) plus the drift

$$M_t\Big[\beta\phi+\tfrac{1}{2}\gamma^\top\gamma\,\phi+\xi^\top\tfrac{\partial\phi}{\partial x}+\tfrac{1}{2}\operatorname{trace}(\Sigma\tfrac{\partial^2\phi}{\partial x\partial x^\top})+(\Gamma\gamma)^\top\tfrac{\partial\phi}{\partial x}\Big]\,dt.$$

Grouping the gradient terms gives the expression in the question.

(2) At a jump time t, $\Delta A_t = \kappa(X_t, X_{t-}) = \kappa(y, x)$, so $M_t = M_{t-} \exp[\kappa(y, x)]$ and

$$\Delta Y_t = M_t \phi(X_t) - M_{t-} \phi(X_{t-}) = M_{t-} \big[\exp[\kappa(y, x)] \phi(y) - \phi(x) \big].$$

The sum of jumps minus its predictable compensator is a local martingale N_t^j, and the compensator itself contributes the predictable drift quoted in the question.

(3) Adding the jump drift from (2) to the continuous drift from (1), the predictable drift of $Y_t = M_t \phi(X_t)$ at state x is $M_t \, \chi(x) \, dt$ with

$$\begin{aligned} \chi(x) = (\xi + \Gamma \gamma)^\top \frac{\partial \phi}{\partial x} + \frac{1}{2} \operatorname{trace}\left(\Sigma \frac{\partial^2 \phi}{\partial x \partial x^\top} \right) + \left(\beta + \frac{\gamma^\top \gamma}{2} \right) \phi(x) \\ + \int \exp[\kappa(y, x)] [\phi(y) - \phi(x)] \, \eta(dy \mid x) + \phi(x) \int \big[\exp[\kappa(y, x)] - 1 \big] \, \eta(dy \mid x). \end{aligned}$$

Collecting the terms multiplying $\phi(x)$ recovers (127.8).

(4) Putting the two martingale pieces together and integrating the drift, steps (1) and (2) give the semimartingale decomposition

$$M_t \phi(X_t) = \phi(X_0) + \int_0^t M_s \, \chi(X_s) \, ds + N_t^c + N_t^j.$$

Rearranging,

$$N_t := M_t \phi(X_t) - \phi(X_0) - \int_0^t M_s \, \chi(X_s) \, ds = N_t^c + N_t^j,$$

which is a local martingale because it is a sum of local martingales.

This is exactly the property required by *Definition 127.4.1*, so $\chi = \mathbb{A}\phi$.

CHAPTER

EIGHT

MISSPECIFIED RECOVERY

Contents

- *Misspecified Recovery*
  - *Overview*
  - *Three transition matrices*
  - *Martingale Component*
  - *From matrices to the general framework*
  - *When the recovery fails*
  - *Additional state vector*
  - *Measuring the martingale component*
  - *Lessons*
  - *Exercises*

128.1 Overview

The lecture *The Recovery Theorem* studies the case in which recovery is valid.

There, **transition independence** lets us use Arrow prices to separate investors' beliefs from the pricing kernel.

This lecture asks what the same Perron–Frobenius approach delivers when that restriction is not imposed.

We will keep three probability measures separate.

The first is the correctly specified probability measure, which governs the Markov state in the model.

The second is the one-period risk-neutral probability measure, which comes from normalizing one-period Arrow prices by bond prices.

The third is the probability measure recovered by Perron–Frobenius Theory, also called the long-term risk-neutral measure.

The central question is whether the recovered probability measure equals the correctly specified probability measure.

Borovička *et al.* [2016] show that, in general, the answer is no.

The paper studies the ratio of the recovered probability measure to the correctly specified probability measure.

The reason is that the stochastic discount factor can contain a martingale component that changes the probability measure.

If that martingale component is identically one, Ross recovery returns the correctly specified transition probabilities.

If it is not identically one, the recovered probability measure absorbs long-term risk adjustments, because martingale increments compound along histories.

In the examples below, the recovered probability measure assigns more probability to adverse long-run-risk states than the correctly specified probability measure.

We will:

- use results from *The Recovery Theorem* without re-proving it,
- study misspecification through the martingale component,
- show why recursive utility and permanent shocks make the recovered probability measure differ from the correctly specified probability measure,
- measure the difference in a long-run risk model.

We will use the following imports.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import linalg
from scipy.integrate import solve_ivp
from scipy.stats import gaussian_kde
```

The next cell contains code inherited from the previous lecture.

It row-normalizes Arrow prices, finds the Perron–Frobenius eigenvalue and positive right eigenvector, and computes stationary distributions.

128.2 Three transition matrices

Let $\mathbf{P} = [p_{ij}]$ denote the correctly specified transition matrix and $\mathbf{Q} = [q_{ij}]$ the Arrow price matrix.

Here "correctly specified" means that $\mathbf{P}$ is the transition matrix that governs the Markov state in the model.

The one-period stochastic discount factor (SDF) satisfies

$$q_{ij} = s_{ij} p_{ij}. \tag{128.1}$$

We will compare $\mathbf{P}$ with two probability matrices constructed from $\mathbf{Q}$.

The first one is the **one-period risk-neutral matrix**.

It divides each row of $\mathbf{Q}$ by the price of a one-period discount bond in the current state:

$$\bar{p}_{ij} = \frac{q_{ij}}{\sum_k q_{ik}}.$$

This matrix absorbs one-period risk adjustments into transition probabilities.

The second one is the transition matrix associated with the **long-term risk-neutral probability**.

It starts from the Perron–Frobenius eigenvalue and positive right eigenvector of $\mathbf{Q}$.

Let $(\exp(\hat{\eta}), \hat{e})$ solve

$$\mathbf{Q}\hat{e} = \exp(\hat{\eta})\hat{e}. \tag{128.2}$$

Then define

$$\hat{p}_{ij} = \exp(-\hat{\eta}) q_{ij} \frac{\hat{e}_j}{\hat{e}_i}. \tag{128.3}$$

The factor $\hat{e}_j / \hat{e}_i$ is chosen to cancel any SDF component of the form $\exp(\hat{\eta}) \hat{e}_i / \hat{e}_j$.

The result is a stochastic matrix $\hat{\mathbf{P}}$.

This construction assumes that $\mathbf{Q}$ has a unique positive right eigenvector up to scale.

For a finite irreducible nonnegative matrix, the Perron root has a strictly positive right eigenvector unique up to scale.

For long-horizon dominance and convergence, one typically imposes a stronger condition such as primitivity or aperiodicity; the paper uses a positivity condition on $\sum_{t=0}^{\infty} \lambda^t \mathbf{Q}^t$.

In general state spaces this guarantee does not carry over: multiple positive eigenfunctions may exist, and an additional selection condition is needed to pin down the long-term risk-neutral measure.

The general framework in the next section makes that selection condition explicit.

Following Borovička *et al.* [2016], $\hat{\mathbf{P}}$ is called a **long-term risk-neutral** transition matrix.

The name means that the Perron–Frobenius eigenvalue and eigenvector isolate the part of pricing that dominates long-maturity Arrow claims.

It is not the same transition matrix as the one-period risk-neutral matrix $\bar{\mathbf{P}}$.

In *The Recovery Theorem*, transition independence restricts the SDF to

$$s_{ij} = \exp(-\delta) \frac{m_j}{m_i}$$

for a positive vector m and scalar δ, which pins down the split between s_{ij} and p_{ij}.

Here we drop that restriction.

The question is whether the transition matrix associated with the long-term risk-neutral probability, $\hat{\mathbf{P}}$, still equals the correctly specified matrix $\mathbf{P}$.

128.2.1 Degenerate Martingale Component

We start with a three-state economy: recession, normal, and expansion.

The correctly specified transition matrix is deliberately simple.

For trend-stationary consumption and power utility, the SDF is

$$s_{ij} = A \left(\frac{c_j}{c_i} \right)^{-\gamma}.$$

This is a case where Ross recovery should return the correctly specified transition matrix.

```
P_true = np.array([
    [0.70, 0.25, 0.05],
    [0.15, 0.65, 0.20],
    [0.05, 0.30, 0.65],
])

c_levels = np.array([0.997, 1.000, 1.003])
state_names = ['recession', 'normal', 'expansion']
```

(continues on next page)

(continued from previous page)

```
δ = -np.log(0.99)    # monthly subjective discount rate
γ_power = 5.0        # risk aversion
g_c = 0.002          # monthly trend growth

# Price Arrow claims as actual probabilities times the power-utility SDF
S_power = (
    np.exp(-δ - γ_power * g_c)
    * (c_levels[None, :] / c_levels[:, None])**(-γ_power)
)
Q_power = S_power * P_true
```

We now compute the one-period risk-neutral matrix and the transition matrix associated with the long-term risk-neutral probability from the same Arrow price matrix.

```
P_bar, q_bonds = risk_neutral_probs(Q_power)
η_hat, exp_η, e_hat, P_hat = perron_frobenius(Q_power)
π_true = stationary_dist(P_true)
π_bar = stationary_dist(P_bar)
π_hat = stationary_dist(P_hat)
```

These two matrices should not be expected to agree.

The row-normalized matrix $\bar{\mathbf{P}}$ is a short-horizon risk-neutral change of measure: it folds the one-period SDF into transition probabilities, so it generally differs from the correctly specified matrix $\mathbf{P}$.

The logic comes from the Perron–Frobenius construction in *The Recovery Theorem*.

In the transition-independent case, the pricing kernel has the form $s_{ij} = \exp(\hat{\eta})\hat{e}_i/\hat{e}_j$.

Substituting this into the Perron–Frobenius transition formula gives

$$\hat{p}_{ij} = \exp(-\hat{\eta})q_{ij}\frac{\hat{e}_j}{\hat{e}_i} = \exp(-\hat{\eta})\left(\exp(\hat{\eta})\frac{\hat{e}_i}{\hat{e}_j}p_{ij}\right)\frac{\hat{e}_j}{\hat{e}_i} = p_{ij}.$$

Thus the transition matrix $\hat{\mathbf{P}}$ associated with the long-term risk-neutral probability cancels the transition-independent part of the SDF.

In this power-utility benchmark, the whole SDF has exactly that form, so the remaining martingale increment should be one and $\hat{\mathbf{P}}$ should coincide with $\mathbf{P}$.

The next calculation checks this by comparing the Perron–Frobenius eigenfunction with c_i^γ and then computing the ratio $\hat{\mathbf{P}}/\mathbf{P}$.

Define the one-period martingale increment

$$\hat{h}_{ij} = \frac{\hat{p}_{ij}}{p_{ij}} = \exp(-\hat{\eta})s_{ij}\frac{\hat{e}_j}{\hat{e}_i}.$$

When $\hat{h}_{ij} = 1$ for every transition, $\hat{\mathbf{P}}$ and $\mathbf{P}$ are the same.

The next section explains why this ratio is the one-period martingale increment.

In the power-utility example, write

$$A = \exp(-\delta - \gamma g_c), \qquad s_{ij} = A\left(\frac{c_j}{c_i}\right)^{-\gamma}.$$

Taking $\hat{e}_i = c_i^\gamma$, up to scale, gives

$$[\mathbf{Q}\hat{e}]_i = \sum_j A\left(\frac{c_j}{c_i}\right)^{-\gamma} p_{ij}c_j^\gamma = Ac_i^\gamma = A\hat{e}_i,$$

so $\exp(\hat{\eta}) = A$.

Consequently,

$$\hat{h}_{ij} = A^{-1} A \left(\frac{c_j}{c_i}\right)^{-\gamma} \frac{c_j^\gamma}{c_i^\gamma} = 1.$$

```
H_power = np.divide(P_hat, P_true, out=np.ones_like(P_true), where=P_true > 0)
e_theory = c_levels**y_power

print("Perron-Frobenius eigenfunction: numerical vs c^gamma")
for name, e_num, e_th in zip(state_names, e_hat / e_hat[1],
                             e_theory / e_theory[1]):
    print(f"{name:9s}: {e_num:.6f}  {e_th:.6f}")

print("\nmartingale increment h_hat = P_hat / P")
print(np.round(H_power, 6))

print("\nconditional means under P")
print(np.round((P_true * H_power).sum(axis=1), 6))

print(f"\nmax |h_hat - 1| = "
      f"{np.max(np.abs(H_power[P_true > 0] - 1)):.2e}")
```

```
Perron-Frobenius eigenfunction: numerical vs c^gamma
recession: 0.985090  0.985090
normal   : 1.000000  1.000000
expansion: 1.015090  1.015090

martingale increment h_hat = P_hat / P
[[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]

conditional means under P
[1. 1. 1.]

max |h_hat - 1| = 1.33e-15
```

The output separates a short-horizon risk adjustment from the Perron–Frobenius approach.

The one-period risk-neutral matrix $\bar{\mathbf{P}}$ is close to, but not the same as, the correctly specified matrix $\mathbf{P}$.

It changes the transition probabilities because one-period Arrow prices include one-period risk adjustments.

By contrast, the long-term risk-neutral matrix $\hat{\mathbf{P}}$ is exactly the same as $\mathbf{P}$ in this example.

The calculation confirms why: the martingale increment $\hat{h}_{ij}$ is one for every transition.

This is the condition under which Ross recovery returns the correctly specified transition matrix.

In this example, that cancellation exhausts the SDF, so the martingale component is degenerate.

128.3 Martingale Component

Let $(\hat{\eta}, \hat{e})$ be the Perron–Frobenius eigenvalue exponent and positive right eigenvector of $\mathbf{Q}$:

$$\mathbf{Q}\hat{e} = \exp(\hat{\eta})\hat{e}.$$

The associated long-term risk-neutral transition matrix is

$$\hat{p}_{ij} = \exp(-\hat{\eta}) q_{ij} \frac{\hat{e}_j}{\hat{e}_i}.$$

To see whether recovery has changed the probability measure, compare each recovered transition probability with the corresponding correctly specified transition probability.

For feasible transitions with $p_{ij} > 0$, define the one-period martingale increment

$$\hat{h}_{ij} = \frac{\hat{p}_{ij}}{p_{ij}}. \tag{128.4}$$

If $\hat{h}_{ij} > 1$, the recovered probability measure assigns more probability to transition (i, j) than the correctly specified probability measure.

If $\hat{h}_{ij} < 1$, it assigns less probability to that transition.

For a fixed current state i, the numbers $\hat{h}_{ij}$ average to one under the correctly specified transition probabilities:

$$\sum_j \hat{h}_{ij} p_{ij} = 1.$$

Thus $\hat{h}_{ij}$ is a one-period martingale increment.

Multiplying these increments along a history of states gives the ratio of the recovered probability measure to the correctly specified probability measure for the whole history.

That ratio process is a martingale, which is why the last term in (128.5) is called a martingale component.

Using (128.1), (128.3), and (128.4), the one-period SDF can be written as

$$s_{ij} = \exp(\hat{\eta}) \frac{\hat{e}_i}{\hat{e}_j} \hat{h}_{ij}. \tag{128.5}$$

The Perron–Frobenius approach therefore separates the SDF into:

Part	Role
$\exp(\hat{\eta})$	deterministic long-run discounting
$\hat{e}_i / \hat{e}_j$	state-dependent long-run term
$\hat{h}_{ij}$	martingale increment that changes probabilities

If $\hat{h}_{ij} = 1$ for every feasible transition, then the transition matrix associated with the recovered probability measure and the correctly specified transition matrix are the same.

This is the condition under which Ross recovery returns the correctly specified transition matrix.

Proposition 128.3.1 (Finite-state martingale component)

Under the finite-state assumptions used in this lecture, for a Markov model with correctly specified transition matrix **P** and Arrow matrix **Q**, the probability measure recovered by Perron–Frobenius Theory returns the correctly specified transition matrix if and only if $\hat{h}_{ij} = 1$ for every transition with $p_{ij} > 0$.

Equivalently, recovery returns the correctly specified transition matrix if and only if the SDF in (128.5) has no non-constant martingale component:

$$s_{ij} = \exp(\hat{\eta}) \frac{\hat{e}_i}{\hat{e}_j}.$$

Proof. Using $q_{ij} = s_{ij} p_{ij}$,

$$\hat{h}_{ij} = \frac{\hat{p}_{ij}}{p_{ij}} = \exp(-\hat{\eta}) s_{ij} \frac{\hat{e}_j}{\hat{e}_i}.$$

Thus $\hat{\mathbf{P}} = \mathbf{P}$ if and only if $\hat{h}_{ij} = 1$ on every feasible transition.

This condition is the same as saying that the SDF can be written as (128.5) with no extra martingale increment.

This finite-state implication is a special case of the paper's general identification result.

If a pair (S, P) explains asset prices and H is a positive multiplicative martingale, then the same asset prices are also explained by the changed probability measure P^H together with the adjusted stochastic discount factor

$$S_t^H = S_t \frac{H_0}{H_t}.$$

More generally, any strictly positive martingale can change probability measures, but multiplicativity preserves the Markov structure used here.

Thus Arrow prices alone cannot usually distinguish a change in beliefs from a change in the SDF.

Ross recovery becomes an identification result only after imposing a restriction such as

$$S_t = \exp(-\delta t) \frac{m(X_t)}{m(X_0)},$$

which rules out a nontrivial martingale component.

The power-utility example above illustrates the proposition.

In that benchmark, the martingale increment $\hat{h}_{ij}$ is identically one.

128.4 From matrices to the general framework

The finite-state calculation has three objects:

1. the correctly specified transition probabilities p_{ij},
2. the SDF increments s_{ij},
3. the Arrow prices $q_{ij} = s_{ij} p_{ij}$.

It also has one diagnostic object:

$$\hat{h}_{ij} = \frac{\hat{p}_{ij}}{p_{ij}} = \exp(-\hat{\eta}) s_{ij} \frac{\hat{e}_j}{\hat{e}_i}.$$

The numbers $\hat{h}_{ij}$ are one-period martingale increments.

They change the probability of a one-period transition from p_{ij} to $\hat{p}_{ij} = \hat{h}_{ij} p_{ij}$.

The general framework in Borovička *et al.* [2016] does the same thing without assuming that states are finite.

The transition matrix becomes a Markov probability measure, the Arrow price matrix becomes a family of pricing operators, and the one-period ratios $\hat{h}_{ij}$ become increments of a positive multiplicative martingale.

The point of this section is to build that dictionary.

128.4.1 Probability space and state

Start with a probability space $(\Omega, \mathcal{F}, P)$.

Here P is the correctly specified probability measure.

In the rational-expectations interpretation of the paper, this is the actual, or original, probability measure governing the state.

In the finite-state section, P was represented by the transition matrix $\mathbf{P} = [p_{ij}]$.

The index set is either discrete time, $\mathbb{T} = \{0, 1, 2, \ldots\}$, or continuous time, $\mathbb{T} = \mathbb{R}_+$.

The main state process is $X = \{X_t : t \in \mathbb{T}\}$, which is stationary and Markov under P.

A second process $W = \{W_t : t \in \mathbb{T}\}$ records shocks that drive X and other economic quantities.

In discrete time, the shock increment between dates t and $t+1$ is

$$\Delta W_{t+1} = W_{t+1} - W_t.$$

The known function ϕ_x maps today's state and the next shock increment into tomorrow's state.

The discrete-time state evolution is

$$X_{t+1} = \phi_x(X_t, \Delta W_{t+1}).$$

The conditional law generated by this equation is the general-state replacement for the finite-matrix row indexed by the current state x.

Assumption 128.4.1 (Markov state and shock increments)

The process X is ergodic under P.

The conditional distribution of ΔW_{t+1} given X_t is time invariant and independent of past shock histories conditioned on X_t.

The filtration $\{\mathcal{F}_t\}$ is generated by the initial condition X_0 and by the history of shocks through date t.

128.4.2 Information and Y

The Markov state and the information that reveals shocks need not coincide.

The state X_t is observed at date t.

The next shock ΔW_{t+1} need not be directly observed from the pair (X_t, X_{t+1}).

If (X_t, X_{t+1}) does reveal ΔW_{t+1}, then X alone carries the relevant shock information.

If it does not, introduce an auxiliary process $Y = \{Y_t\}$ with stationary increments.

The known function ϕ_y maps today's state and the next shock increment into the increment of Y.

The discrete-time evolution for the auxiliary increment is

$$Y_{t+1} - Y_t = \phi_y(X_t, \Delta W_{t+1}).$$

The pair $(X_{t+1}, Y_{t+1} - Y_t)$ is then rich enough, together with X_t, to recover the shock increment ΔW_{t+1}.

This device lets the model handle shocks or growth components that affect payoffs and SDFs but are not fully summarized by the next Markov state alone.

Write the enlarged process as $Z = (X, Y)$.

The process Z is Markov with a triangular structure: the conditional distribution of $(X_{t+1}, Y_{t+1} - Y_t)$ depends on the past only through X_t.

Histories of Z, together with X_0, generate the same information as the shock history.

This is why the next Perron–Frobenius problem can first be posed with eigenfunctions of X alone.

The section *Additional state vector* later returns to what changes when the eigenfunction is allowed to depend on Y as well.

128.4.3 Multiplicative functionals

The general framework needs a way to describe objects that compound over time.

This is the role of a positive multiplicative functional $M = \{M_t\}$.

Its log increment is a function of today's state and the next shock increment:

$$\log \frac{M_{t+1}}{M_t} = \kappa(X_t, \Delta W_{t+1}).$$

Equivalently,

$$\frac{M_{t+1}}{M_t} = \exp\{\kappa(X_t, \Delta W_{t+1})\}.$$

Thus M_t / M_0 is a product of positive one-period increments.

This is the Condition-1 version of a multiplicative functional used in the paper.

The formal definition is slightly broader, but this form covers the models studied below.

Under *Assumption 128.4.1*, the logarithm of M has stationary increments.

Products and reciprocals of positive multiplicative functionals are again positive multiplicative functionals.

Exponential functions of linear combinations of the components of Y are examples.

Stochastic discount factors, stochastic growth factors, and positive multiplicative martingales are all modeled this way.

In the finite-state model, the SDF increment s_{ij} is one example of a multiplicative-functional increment.

The ratio h_{ij} that changes probabilities is another.

128.4.4 Stochastic discount factors and pricing operators

A stochastic discount factor $S = \{S_t\}$ is a positive multiplicative functional with $S_0 = 1$ and finite first moments conditional on X_0.

Let Φ_t be a bounded payoff measurable with respect to the date-t information.

The date-τ price of Φ_t is

$$\Pi_{\tau,t}(\Phi_t) = E\left[\frac{S_t}{S_\tau}\Phi_t \mid \mathcal{F}_\tau\right].$$

The ratio S_t/S_τ is the stochastic discount factor from date t back to date τ.

If the payoff is a bounded function $f(X_t)$ of the future Markov state, this pricing formula defines a horizon-t operator Q_t by

$$[Q_t f](x) = E[S_t f(X_t) \mid X_0 = x].$$

This operator is the general-state analogue of multiplying a payoff vector by the Arrow-price matrix $\mathbf{Q}$.

To see the connection, suppose again that the state space is finite and $t = 1$.

Then

$$[Q_1 f]_i = \sum_j s_{ij} p_{ij} f_j = \sum_j q_{ij} f_j.$$

Thus Q_1 is exactly the matrix $\mathbf{Q}$.

In discrete time, the multiplicative property of S implies that Q_t is obtained by applying the one-period operator Q_1 repeatedly.

In continuous time, the family $\{Q_t : t \geq 0\}$ is a semigroup of pricing operators.

128.4.5 Martingales and equivalent probability measures

Different stochastic discount factor / probability pairs can produce the same pricing operators, and that flexibility is the source of the identification problem.

For the Markov setting used here, the probability changes of interest are generated by positive multiplicative martingales.

At the level of probability changes, let $H = \{H_t\}$ be a strictly positive martingale with $E[H_0] = 1$ under P.

For an event A observable by date τ, the changed probability measure P^H is defined by

$$P^H(A) = E[1_A H_\tau].$$

The law of iterated expectations makes this definition consistent across dates.

When H is also a multiplicative functional, making it a multiplicative martingale, the change of probability preserves the Markov structure of Z.

The SDF that represents the same prices under P^H is

$$S_t^H = S_t \frac{H_0}{H_t}.$$

Thus the same pricing operators can be represented by the pair (S, P) or by the pair (S^H, P^H).

In the finite-state model, this is just

$$p_{ij}^H = h_{ij} p_{ij}, \qquad s_{ij}^H = \frac{s_{ij}}{h_{ij}},$$

so that

$$s_{ij}^H p_{ij}^H = s_{ij} p_{ij} = q_{ij}.$$

The same Arrow prices can therefore be explained by changing the probability measure and offsetting that change in the SDF.

This is also the precise sense in which Arrow prices alone do not identify beliefs.

128.4.6 What Perron–Frobenius recovers

Now return to the Perron–Frobenius step.

The finite-state equation was (128.2).

The general-state replacement is an eigenfunction problem for the pricing operators: find a scalar $\hat{\eta}$ and a positive function $\hat{e}$ such that, for every horizon t,

$$[Q_t \hat{e}](x) = \exp(\hat{\eta} t) \hat{e}(x). \tag{128.6}$$

The positive function $\hat{e}$ is the general-state counterpart of the Perron–Frobenius eigenvector.

The scalar $\hat{\eta}$ is the log eigenvalue.

In finite states, $\hat{e}$ is just a positive vector with one entry for each state.

In general state spaces, $\hat{e}(x)$ is a positive function of the current state.

Its job is to record the state-dependent part of long-horizon valuation.

Equation (128.6) says that a future payoff equal to $\hat{e}(X_t)$ has date-0 price $\exp(\hat{\eta} t)\hat{e}(X_0)$.

Thus $\hat{\eta}$ gives the common growth or discount rate, while $\hat{e}$ gives the state-dependent scaling.

Since eigenfunctions are defined only up to scale, uniqueness always means uniqueness up to multiplication by a positive constant.

In general state spaces, existence of a positive eigenfunction is also a substantive condition.

The eigenfunction equation implies the conditional moment restriction

$$E[S_t \hat{e}(X_t) \mid \mathcal{F}_\tau] = \exp((t-\tau)\hat{\eta}) S_\tau \hat{e}(X_\tau), \qquad t \geq \tau.$$

Use this restriction to define

$$\frac{\hat{H}_t}{\hat{H}_0} = \exp(-\hat{\eta} t) S_t \frac{\hat{e}(X_t)}{\hat{e}(X_0)}. \tag{128.7}$$

This process is a martingale because, for $t \geq \tau$,

$$\begin{aligned} E\left[\frac{\hat{H}_t}{\hat{H}_0} \mid \mathcal{F}_\tau\right] &= \frac{\exp(-\hat{\eta}t)}{\hat{e}(X_0)} E[S_t \hat{e}(X_t) \mid \mathcal{F}_\tau] \\ &= \frac{\exp(-\hat{\eta}t)}{\hat{e}(X_0)} \exp((t-\tau)\hat{\eta}) S_\tau \hat{e}(X_\tau) \\ &= \exp(-\hat{\eta}\tau) S_\tau \frac{\hat{e}(X_\tau)}{\hat{e}(X_0)} = \frac{\hat{H}_\tau}{\hat{H}_0}. \end{aligned}$$

The process is positive because S and $\hat{e}$ are positive.

Its one-period increment is

$$\frac{\hat{H}_{t+1}}{\hat{H}_t} = \exp(-\hat{\eta}) \frac{S_{t+1}}{S_t} \frac{\hat{e}(X_{t+1})}{\hat{e}(X_t)}. \tag{128.8}$$

In the finite-state model, when $X_t = i$ and $X_{t+1} = j$, (128.8) becomes

$$\frac{\hat{H}_{t+1}}{\hat{H}_t} = \exp(-\hat{\eta}) s_{ij} \frac{\hat{e}_j}{\hat{e}_i} = \hat{h}_{ij} = \frac{\hat{p}_{ij}}{p_{ij}}.$$

This is the same three-component decomposition as (128.5).

In finite states, (128.5) is equivalent to

$$\frac{S_{t+1}}{S_t} = \exp(\hat{\eta}) \frac{\hat{e}(X_t)}{\hat{e}(X_{t+1})} \frac{\hat{H}_{t+1}}{\hat{H}_t}.$$

The only change is notation: $\hat{h}_{ij}$ is the one-period density ratio in a finite Markov chain, while $\hat{H}_{t+1}/\hat{H}_t$ is the corresponding one-period density ratio in the general Markov setting.

Conditional on X_0, the likelihood ratio for histories through date t is $\hat{H}_t/\hat{H}_0$.

For the unconditional measure on $\mathcal{F}_t$, the Radon–Nikodym density is $\hat{H}_t$, where $\hat{H}_0$ adjusts the initial distribution.

If $\hat{H}_{t+1}/\hat{H}_t$ is not identically one, the recovered probability measure differs from the correctly specified probability measure.

We show the restriction that rules out this difference in the next section.

128.4.7 Selection and recovery

In finite irreducible matrix problems, Perron–Frobenius theory gives a unique positive eigenvector up to scale, so the recovered transition matrix is pinned down by $\mathbf{Q}$.

In general state spaces, a positive eigenfunction need not exist, and multiple positive eigenfunctions may solve the same pricing operator problem when one does.

The paper therefore imposes a selection condition on the probability measure induced by the candidate eigenfunction.

Assumption 128.4.2 (Ergodicity of the recovered measure)

The process X is stationary and ergodic under $P^{\hat{H}}$, the probability measure induced by the multiplicative martingale $\hat{H}$ defined in the previous section.

Proposition 128.4.1 (Uniqueness of the Perron–Frobenius solution)

There is at most one solution $(\hat{e}, \hat{\eta})$ to the Perron–Frobenius problem such that X is stationary and ergodic under the induced probability measure $P^{\hat{H}}$.

This selected solution, when it exists, identifies the long-term risk-neutral measure.

It does not by itself identify subjective beliefs.

To make recovery identify beliefs, an additional restriction is needed on the SDF.

The restriction used by Ross recovery is the paper's Condition 4:

Assumption 128.4.3

Let

$$S_t = \exp(-\delta t)\frac{m(X_t)}{m(X_0)}$$

for some positive function m and real number δ.

This is the transition-independence restriction from *The Recovery Theorem*, imposed on the SDF representation whose probability measure one wants to recover.

Under this restriction, setting $\hat{e} = 1/m$ and $\hat{\eta} = -\delta$ gives

$$\frac{\hat{H}_t}{\hat{H}_0} = \exp(\delta t)\left[\exp(-\delta t)\frac{m(X_t)}{m(X_0)}\right]\frac{1/m(X_t)}{1/m(X_0)} = 1.$$

Thus the martingale component is identically one after normalization.

This is the general-state version of the finite-state condition $\hat{h}_{ij} = 1$ for every feasible transition.

If this martingale is not identically one, the recovered probability measure absorbs it and generally differs from the correctly specified probability measure.

We will see a few important examples of this in the next section.

The section *Additional state vector* returns to what happens when the eigenfunction is allowed to depend on the auxiliary process Y.

128.4.8 Continuous-time version

Let's briefly introduce the model in continuous time before discussing examples where recovery fails.

We introduce the diffusion notation because the long-run risk example below is written in continuous time.

The objects are the same as before:

- X is the Markov state,
- Y records additional growing or shock-revealing components,
- M is a positive multiplicative functional, such as an SDF, a cash-flow growth process, or a martingale used to change probabilities.

In the continuous-time version, W is a Brownian motion.

The state, auxiliary process, and multiplicative functional satisfy

$$\begin{aligned} dX_t &= \mu_x(X_t)dt + \sigma_x(X_t)dW_t, \\ dY_t &= \mu_y(X_t)dt + \sigma_y(X_t)dW_t, \\ d\log M_t &= \beta(X_t)dt + \alpha(X_t)\cdot dW_t. \end{aligned}$$

Here μ_x and μ_y are drift functions, while σ_x and σ_y are shock-exposure matrices.

The function β is the drift of $\log M$, and α is the Brownian shock exposure of $\log M$.

The invertibility assumption on the stacked shock-exposure matrix is the continuous-time counterpart of the discrete-time condition that $(X_{t+1}, Y_{t+1} - Y_t)$ reveals the shock increment.

It lets the history of $Z = (X, Y)$ reveal the Brownian information.

For M to be a local martingale, its drift must satisfy

$$\beta(x) = -\frac{1}{2}\alpha(x)\cdot\alpha(x).$$

This follows from Ito's formula:

$$\frac{dM_t}{M_t} = \left(\beta(X_t) + \frac{1}{2}\alpha(X_t)\cdot\alpha(X_t)\right)dt + \alpha(X_t)\cdot dW_t.$$

A local martingale has zero drift in dM_t/M_t, which gives the displayed restriction.

Additional integrability conditions then ensure that this local martingale is a true martingale.

Under the probability measure induced by a martingale H with this exposure α, $\widetilde{W}_t = W_t - \int_0^t \alpha(X_s)ds$ is Brownian.

With this sign convention, the drift of X changes from μ_x to $\mu_x + \sigma_x\alpha$, and the drift of Y changes from μ_y to $\mu_y + \sigma_y\alpha$.

This is the continuous-time analogue of replacing p_{ij} by $h_{ij}p_{ij}$ in the finite-state model.

The Markov and triangular structure of Z is preserved, which is why the same Perron–Frobenius decomposition can be applied.

128.5 When the recovery fails

Now let's discuss a few examples where the recovered probability measure differs from the correctly specified probability measure.

128.5.1 Recursive utility

We now use the martingale component to see when the recovered probability measure differs from the correctly specified probability measure.

In the previous example, all risk adjustment in the SDF could be written as a ratio of a function of today's state to a function of tomorrow's state.

The Perron–Frobenius transition formula cancels exactly that kind of term.

Recursive utility adds a continuation-value term.

The key point is that this term behaves like the martingale increment defined above.

For the unit-EIS Epstein–Zin case in Borovička *et al.* [2016], with $C_t = \exp(g_c t) c(X_t)$, write the translated continuation value as $V_t = g_c t + v(X_t)$, and define

$$v_i^* = \exp((1-\gamma) v_i).$$

The SDF is

$$s_{ij} = \exp(-\delta - g_c) \frac{c_i}{c_j} \frac{v_j^*}{\sum_k p_{ik} v_k^*}.$$

In this unit-EIS example, the Perron–Frobenius eigenfunction is $\hat{e}_j = c_j$ and $\hat{\eta} = -(\delta + g_c)$.

Applying the Perron–Frobenius transition formula therefore leaves

$$\hat{p}_{ij} = p_{ij} \frac{v_j^*}{\sum_k p_{ik} v_k^*}.$$

The denominator is the conditional expectation of v_j^* given current state i.

Therefore the last fraction has conditional mean one under $\mathbf{P}$.

It is therefore a martingale increment.

When v^* is not constant, that ratio varies across next-period states.

That variation is why the probability measure recovered by Perron–Frobenius Theory no longer gives the correctly specified transition matrix.

The next cell solves the finite-state continuation-value equation and builds the SDF.

```
def solve_ez_unit_eis(P, c, δ, γ, g_c, tol=1e-12, max_iter=10_000):
    """Finite-state unit-EIS Epstein-Zin continuation values and SDF."""
    β = np.exp(-δ)
    log_c = np.log(c)
    n = len(c)
    flow = (1 - β) * log_c + β * g_c

    if abs(γ - 1) < 1e-10:
        v = linalg.solve(np.eye(n) - β * P, flow)
        v_star = np.ones(n)
        Pv_star = np.ones(n)
    else:
        v = log_c.copy()
        for _ in range(max_iter):
            v_star = np.exp((1 - γ) * v)
            Pv_star = P @ v_star
            v_new = flow + β / (1 - γ) * np.log(Pv_star)
            if np.max(np.abs(v_new - v)) < tol:
                v = v_new
                break
            v = v_new
        else:
            raise ValueError("Epstein-Zin fixed point did not converge.")

        v_star = np.exp((1 - γ) * v)
        Pv_star = P @ v_star

    S = (
        np.exp(-δ - g_c)
        * (c[:, None] / c[None, :])
```

(continues on next page)

(continued from previous page)

```
        * (v_star[None, :] / Pv_star[:, None])
    )

    return v, v_star, S
```

At log utility, v^* is constant and the martingale increment is one.

As risk aversion rises, continuation values matter more.

The recovered probability measure then differs more from the correctly specified probability measure.

To make the mechanism visible in a small three-state example, the figure below uses the more dispersed consumption vector

$$c = (0.85, 1.00, 1.15).$$

The heatmap reports percentage deviations of the martingale increment from one: $100(\hat{h}_{ij} - 1)$.

Positive entries are transitions that receive more probability under the recovered probability measure than under the correctly specified probability measure.

The right panel reports the increase in the recovered recession probability, measured in percentage points.

```
c_recursive = np.array([0.85, 1.00, 1.15])
γ_demo = 10.0
_, _, S_demo = solve_ez_unit_eis(P_true, c_recursive, δ, γ_demo, g_c)
Q_demo = S_demo * P_true
H_demo, _, _, P_hat_demo = martingale_increment(Q_demo, P_true)
H_dev = 100 * (H_demo - 1)

γ_grid = np.linspace(1, 15, 80)
rec_prob = []
for γ in γ_grid:
    _, _, S_g = solve_ez_unit_eis(P_true, c_recursive, δ, γ, g_c)
    Q_g = S_g * P_true
    _, _, _, P_hat_g = martingale_increment(Q_g, P_true)
    rec_prob.append(stationary_dist(P_hat_g)[0])
rec_prob = np.array(rec_prob)
rec_prob_gain = 100 * (rec_prob - π_true[0])

fig, axes = plt.subplots(1, 2, figsize=(12, 4.5))

bound = np.max(np.abs(H_dev))
im = axes[0].imshow(H_dev, cmap='RdBu_r', vmin=-bound, vmax=bound)
axes[0].set_xticks(range(3))
axes[0].set_yticks(range(3))
axes[0].set_xticklabels(state_names, rotation=20)
axes[0].set_yticklabels(state_names)
axes[0].set_xlabel('next state')
axes[0].set_ylabel(r'current state')
axes[0].set_title(r'martingale increment, $\gamma=10$')

for i in range(3):
    for j in range(3):
        axes[0].text(j, i, f"{H_dev[i, j]:.1f}",
                     ha='center', va='center', fontsize=9)
plt.colorbar(im, ax=axes[0], fraction=0.046, pad=0.04,
             label=r'$100(\hat h_{ij}-1)$')
```

(continues on next page)

(continued from previous page)

```
axes[1].plot(y_grid, rec_prob_gain, lw=2.5)
axes[1].axhline(0, ls='--', lw=1.5, color='0.5')
axes[1].set_xlabel(r"risk aversion $\gamma$")
axes[1].set_ylabel('increase in recession probability\n(percentage points)')
axes[1].set_title('recovered recession probability')
axes[1].set_ylim(0, rec_prob_gain.max() * 1.08)

plt.tight_layout()
plt.show()
```

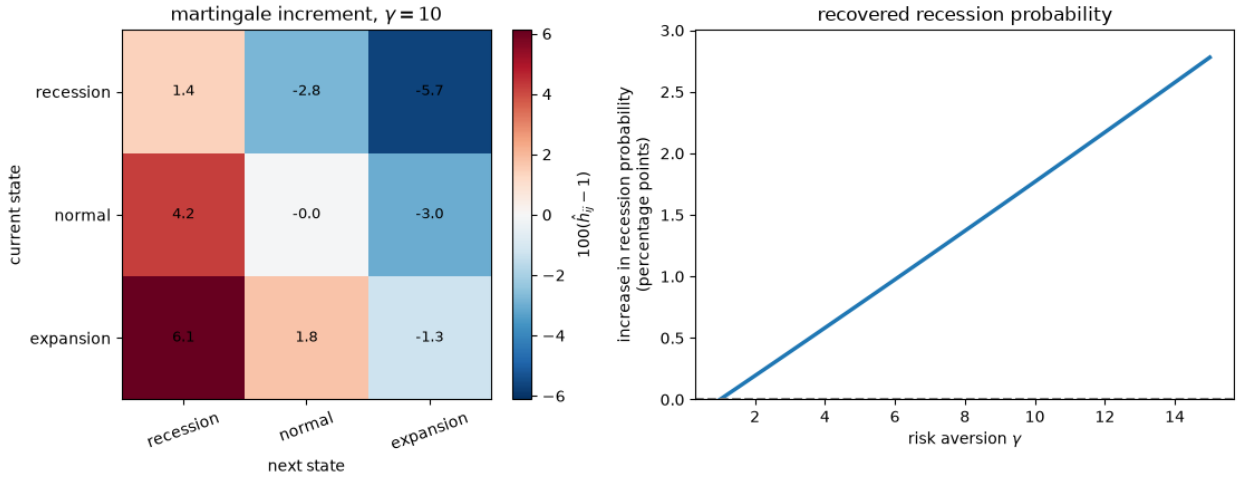


Fig. 128.1: Recursive utility generates a nonconstant martingale increment.

Recursive utility makes the recovered probability measure assign more probability to recession transitions.

At $\gamma = 10$, transitions into recession receive more probability under the recovered probability measure, while transitions into expansion receive less.

As risk aversion rises, the stationary recession probability under the recovered probability measure moves further above its correctly specified value.

Thus, as the continuation-value term creates a nonconstant $\hat{h}_{ij}$, the transition matrix associated with the long-term risk-neutral probability no longer equals the correctly specified transition matrix.

128.5.2 Permanent Shocks

Recursive utility gives one nonconstant martingale component.

Permanent shocks provide another.

Suppose consumption has a permanent shock,

$$\log C_{t+1} - \log C_t = g + x(X_{t+1}) - x(X_t) + \sigma\varepsilon_{t+1},$$

where ε_{t+1} is independent over time.

With power utility, the SDF contains

$$\exp(-\delta - \gamma g)\exp\{-\gamma[x(X_{t+1}) - x(X_t)]\}\exp(-\gamma\sigma\varepsilon_{t+1}).$$

The middle term depends only on the current and next Markov states.

It is a ratio of state functions, so the Perron–Frobenius transition formula can cancel it.

The permanent shock term depends on the new shock ε_{t+1}.

Because that shock is not summarized by the finite Markov state in this construction, there is no state function whose ratio can cancel it.

After dividing by its conditional mean, the shock term becomes a martingale increment:

$$\frac{\exp(-\gamma\sigma\varepsilon_{t+1})}{E[\exp(-\gamma\sigma\varepsilon_{t+1})]}.$$

Thus permanent consumption shocks can make the recovered probability measure differ from investors' beliefs, even under ordinary power utility.

This statement is relative to the Markov state used in the recovery procedure.

Enlarging the state or information structure to account for the shock can accommodate it, but doing so leads to the identification problem discussed in *Additional state vector*.

128.5.3 Long-run risk

We now move from small finite-state examples to a standard continuous-time macro-finance model.

The model is the Bansal–Yaron long-run risk model, using the calibration reported by Borovička *et al.* [2016].

The point is to compare the recovered probability measure with the correctly specified probability measure in a standard macro-finance model.

The construction has the same structure as before.

We first write the correctly specified state dynamics, then compute the probability measure implied by the Perron–Frobenius approach.

The state vector $X_t = (X_{1t}, X_{2t})'$ follows

$$\begin{aligned} dX_{1t} &= [\mu_{11}(X_{1t} - \iota_1) + \mu_{12}(X_{2t} - \iota_2)]dt + \sqrt{X_{2t}}\sigma_1 dW_t, \\ dX_{2t} &= \mu_{22}(X_{2t} - \iota_2)dt + \sqrt{X_{2t}}\sigma_2 dW_t. \end{aligned}$$

Here X_1 is predictable consumption growth and X_2 is stochastic volatility.

The representative agent has Epstein–Zin utility with unit elasticity of intertemporal substitution.

The continuation value introduces the continuous-time analogue of the martingale component above.

We denote that process by H^*, and the SDF satisfies

$$d\log S_t = -\delta dt - d\log C_t + d\log H_t^*.$$

Here H^* is the continuation-value martingale entering the Epstein–Zin SDF.

The multiplicative martingale $\widehat{H}$ associated with the Perron–Frobenius problem is obtained only after also incorporating the Perron–Frobenius eigenfunction.

In models with martingale components in consumption growth, H^* and $\widehat{H}$ need not coincide.

The next cell sets the calibration.

```
lrr_params = dict(
    δ=0.002,
    γ=10.0,
    μ11=-0.021,
    μ12=0.0,
    μ22=-0.013,
    ι1=0.0,
    ι2=1.0,
    σ1=np.array([0.0, 0.00034, 0.0]),
    σ2=np.array([0.0, 0.0, -0.038]),
    β_c0=0.0015,
    β_c1=1.0,
    β_c2=0.0,
    α_c=np.array([0.0078, 0.0, 0.0]),
)
```

The next code block computes how the different probability measures change the drift of the state vector.

The first quantity is the continuation value.

In this affine model, the translated continuation value is linear in the state:

$$v(x) = v_0 + v_1 x_1 + v_2 x_2.$$

This is why we call v_1 and v_2 slopes.

They are the derivatives of the continuation value with respect to predictable growth and volatility.

These slopes enter the continuation-value martingale H^*.

In the code, this martingale has shock exposure

$$\alpha_{H^*} = (1-\gamma)(\alpha_c + \sigma_1 v_1 + \sigma_2 v_2).$$

Since the SDF is $d\log S_t = -\delta dt - d\log C_t + d\log H_t^*$, its shock exposure is

$$\alpha_S = -\alpha_c + \alpha_{H^*}.$$

This vector α_S drives the one-period risk-neutral change of measure.

The second quantity is the Perron–Frobenius eigenfunction.

It is exponential-affine:

$$\widehat{e}(x) = \exp(e_0 + e_1 x_1 + e_2 x_2).$$

Thus e_1 and e_2 are slopes of the log eigenfunction.

Because X_1 and X_2 have shock loadings σ_1 and σ_2, the Perron–Frobenius eigenfunction contributes the additional shock exposure

$$\sigma_1 e_1 + \sigma_2 e_2.$$

Therefore the one-period risk-neutral dynamics use only α_S, while the dynamics under the long-term risk-neutral measure use

$$\alpha_S + \sigma_1 e_1 + \sigma_2 e_2.$$

The functions below follow this order: compute (v_1, v_2), compute α_S and (e_1, e_2), and then translate these shock exposures into drifts for X.

```
def solve_value_function(p):
    """Slopes of the affine continuation value."""
    δ, γ = p["δ"], p["γ"]
    μ11, μ12, μ22 = p["μ11"], p["μ12"], p["μ22"]
    σ1, σ2 = p["σ1"], p["σ2"]
    β_c1, β_c2 = p["β_c1"], p["β_c2"]
    α_c = p["α_c"]

    # v1 is the coefficient on predictable growth in v(x).
    v1 = β_c1 / (δ - μ11)

    # v2 is the coefficient on volatility.
    # In the affine model it is the stable root of a scalar quadratic.
    A_vec = α_c + σ1 * v1
    B_vec = σ2

    a = 0.5 * (1 - γ) * np.dot(B_vec, B_vec)
    b = (μ22 - δ) + (1 - γ) * np.dot(A_vec, B_vec)
    c = β_c2 + μ12 * v1 + 0.5 * (1 - γ) * np.dot(A_vec, A_vec)

    disc = b**2 - 4 * a * c
    if disc < 0:
        raise ValueError("Value function does not exist for these parameters.")

    v2 = (-b - np.sqrt(disc)) / (2 * a)
    return v1, v2


def solve_pf_lrr(p, v1, v2):
    """Perron-Frobenius eigenfunction slopes and the SDF diffusion loading."""
    δ, γ = p["δ"], p["γ"]
    μ11, μ12, μ22 = p["μ11"], p["μ12"], p["μ22"]
    ι1, ι2 = p["ι1"], p["ι2"]
    σ1, σ2 = p["σ1"], p["σ2"]
    α_c = p["α_c"]
    β_c0, β_c1, β_c2 = p["β_c0"], p["β_c1"], p["β_c2"]

    # Continuation-value martingale exposure and SDF exposure.
    α_h_star = (1 - γ) * (α_c + σ1 * v1 + σ2 * v2)
    α_s = -α_c + α_h_star

    # Drift coefficients of log S before the Perron-Frobenius decomposition.
    β_s11 = -β_c1
    β_s12 = -β_c2 - 0.5 * np.dot(α_h_star, α_h_star)
    β_s0 = -δ - β_c0 - 0.5 * ι2 * np.dot(α_h_star, α_h_star)

    # e1 and e2 are coefficients in log e(x) = e0 + e1 x1 + e2 x2.
    e1 = -β_s11 / μ11

    # e2 solves the remaining quadratic from the Perron-Frobenius eigenvalue equation.
    const = (β_s12 + 0.5 * np.dot(α_s, α_s)
             + e1 * (μ12 + np.dot(σ1, α_s))
             + 0.5 * e1**2 * np.dot(σ1, σ1))
    lin = μ22 + np.dot(σ2, α_s) + e1 * np.dot(σ1, σ2)
    quad = 0.5 * np.dot(σ2, σ2)

    disc = lin**2 - 4 * quad * const
```

(continues on next page)

(continued from previous page)

```
    roots = [(-lin - np.sqrt(disc)) / (2 * quad),
             (-lin + np.sqrt(disc)) / (2 * quad)]

    candidates = []
    for e2 in roots:
        eta = (β_s0 - β_s11 * ι1 - β_s12 * ι2
               - e1 * (μ11 * ι1 + μ12 * ι2) - e2 * μ22 * ι2)
        candidates.append((eta, e2))

    # Choose the solution that gives the smaller eigenvalue exponent.
    eta, e2 = min(candidates)
    return e1, e2, eta, α_s


def recovered_lrr_dynamics(p, e1, e2, α_s):
    """State dynamics under the long-term risk-neutral measure."""
    μ11, μ12, μ22 = p["μ11"], p["μ12"], p["μ22"]
    ι1, ι2 = p["ι1"], p["ι2"]
    σ1, σ2 = p["σ1"], p["σ2"]

    # The long-term risk-neutral measure uses the SDF exposure plus the eigenfunction␣
 ↪exposure.
    α_h = α_s + σ1 * e1 + σ2 * e2

    # A diffusion change of measure shifts each drift by sigma_i dot alpha_h.
    μ_hat_11 = μ11
    μ_hat_12 = μ12 + np.dot(σ1, α_h)
    μ_hat_22 = μ22 + np.dot(σ2, α_h)

    # Rewrite the shifted drift in mean-reversion form.
    ι_hat_2 = (μ22 / μ_hat_22) * ι2
    ι_hat_1 = ι1 + (μ12 * ι2 - μ_hat_12 * ι_hat_2) / μ11

    return dict(
        μ11=μ_hat_11,
        μ12=μ_hat_12,
        μ22=μ_hat_22,
        ι1=ι_hat_1,
        ι2=ι_hat_2,
        σ1=σ1,
        σ2=σ2,
        α_h=α_h,
    )


def risk_neutral_lrr_dynamics(p, α_s):
    """State dynamics under the one-period risk-neutral measure."""
    μ11, μ12, μ22 = p["μ11"], p["μ12"], p["μ22"]
    ι1, ι2 = p["ι1"], p["ι2"]
    σ1, σ2 = p["σ1"], p["σ2"]

    # The one-period risk-neutral measure uses only the SDF exposure.
    μ_bar_11 = μ11
    μ_bar_12 = μ12 + np.dot(σ1, α_s)
    μ_bar_22 = μ22 + np.dot(σ2, α_s)

    # Rewrite the shifted drift in mean-reversion form.
```

(continues on next page)

(continued from previous page)

```
    ι_bar_2 = (μ22 / μ_bar_22) * ι2
    ι_bar_1 = ι1 + (μ12 * ι2 - μ_bar_12 * ι_bar_2) / μ11

    return dict(
        μ11=μ_bar_11,
        μ12=μ_bar_12,
        μ22=μ_bar_22,
        ι1=ι_bar_1,
        ι2=ι_bar_2,
        σ1=σ1,
        σ2=σ2,
    )
```

For the calibration used here, the recovered probability measure changes the long-run state distribution.

It lowers the mean of expected growth and raises the mean of volatility.

```
v1, v2 = solve_value_function(lrr_params)
e1, e2, η_lrr, α_s = solve_pf_lrr(lrr_params, v1, v2)
dyn_hat = recovered_lrr_dynamics(lrr_params, e1, e2, α_s)
dyn_bar = risk_neutral_lrr_dynamics(lrr_params, α_s)

print(f"value slopes:       v1 = {v1:.4f}, v2 = {v2:.4f}")
print(f"eigenfunction coefficients: e1 = {e1:.4f}, e2 = {e2:.4f}")
print(f"log eigenvalue:     eta = {η_lrr:.6f}  "
      f"(annualized {12 * η_lrr:.4f})")
print()
print("Long-run means under three measures")
print("measure        iota_1     iota_2     mu_12      mu_22")
print("---------   --------   --------   --------   --------")
print(f"actual      {lrr_params['ι1']:8.5f}   {lrr_params['ι2']:8.5f}"
      f"   {lrr_params['μ12']:8.5f}   {lrr_params['μ22']:8.5f}")
print(f"one-period  {dyn_bar['ι1']:8.5f}   {dyn_bar['ι2']:8.5f}"
      f"   {dyn_bar['μ12']:8.5f}   {dyn_bar['μ22']:8.5f}")
print(f"long-term   {dyn_hat['ι1']:8.5f}   {dyn_hat['ι2']:8.5f}"
      f"   {dyn_hat['μ12']:8.5f}   {dyn_hat['μ22']:8.5f}")
```

```
value slopes:       v1 = 43.4783, v2 = -0.0871
eigenfunction coefficients: e1 = -47.6190, e2 = 0.2449
log eigenvalue:     eta = -0.000316  (annualized -0.0038)

Long-run means under three measures
measure        iota_1     iota_2     mu_12      mu_22
---------   --------   --------   --------   --------
actual       0.00000    1.00000    0.00000   -0.01300
one-period  -0.00236    1.09537   -0.00005   -0.01187
long-term   -0.00273    1.12901   -0.00005   -0.01151
```

These numbers show the mechanism clearly.

The positive value slope v_1 says that the continuation value is very sensitive to predictable consumption growth.

The volatility slope v_2 is negative in this calibration, so higher volatility lowers continuation value.

The eigenfunction coefficient e_1 has the opposite sign: the long-term change of measure loads negatively on predictable growth.

Thus the recovered probability measure assigns more probability to histories with lower expected growth.

The positive e_2 has the opposite implication for volatility, assigning more probability to higher-volatility states.

The table translates those coefficients into state dynamics.

Relative to the correctly specified probability measure, both risk-neutral measures lower the long-run mean of predictable growth and raise the long-run mean of volatility.

The long-term risk-neutral measure moves further in that direction than the one-period risk-neutral measure: ι_1 falls from 0 to about -0.0027, while ι_2 rises from 1 to about 1.13.

The small negative log eigenvalue means that $\exp(\eta)$ is slightly below one; with the usual yield sign convention, $-\eta$ is the corresponding long-run discount rate.

Stationary Densities

The coefficient table gives one summary of the difference between probability measures.

A stationary-density plot gives another.

It shows not only that the means of X_1 and X_2 move, but also which combinations of growth and volatility become more likely.

This matters because treating the recovered probability measure as beliefs changes the whole forecast distribution, not just a pair of long-run averages.

Under the recovered probability measure, probability mass shifts toward adverse long-run-risk states.

These are states with lower predictable growth X_1 and higher volatility X_2.

The dashed contour adds the one-period risk-neutral probability measure.

In this calibration, the one-period risk-neutral and long-term risk-neutral stationary distributions are close to each other, and both are far from the correctly specified distribution.

Thus the martingale component accounts for much of the risk adjustment in the state dynamics.

The paper's Figure 1 reports model-implied stationary densities; the simulation below is a numerical approximation to those densities.

The plot below simulates the state process under each probability measure and estimates the stationary joint density of (X_2, X_1).

The horizontal line marks $X_1 = 0$ and the vertical line marks the correctly specified mean of volatility, $X_2 = \iota_2$.

```
def simulate_lrr(dyn, T=180_000, seed=123):
    """
    Euler simulation of the LRR state process under one probability measure.
    """
    rng = np.random.default_rng(seed)
    X1 = np.zeros(T)
    X2 = np.full(T, dyn["ι2"])

    # Euler step with monthly time increment
    for t in range(1, T):
        X2_prev = max(X2[t-1], 1e-9)
        dW = rng.standard_normal(3)
        sqrt_X2 = np.sqrt(X2_prev)

        X1[t] = (
            X1[t-1]
            + dyn["μ11"] * (X1[t-1] - dyn["ι1"])
            + dyn["μ12"] * (X2_prev - dyn["ι2"])
```

(continues on next page)

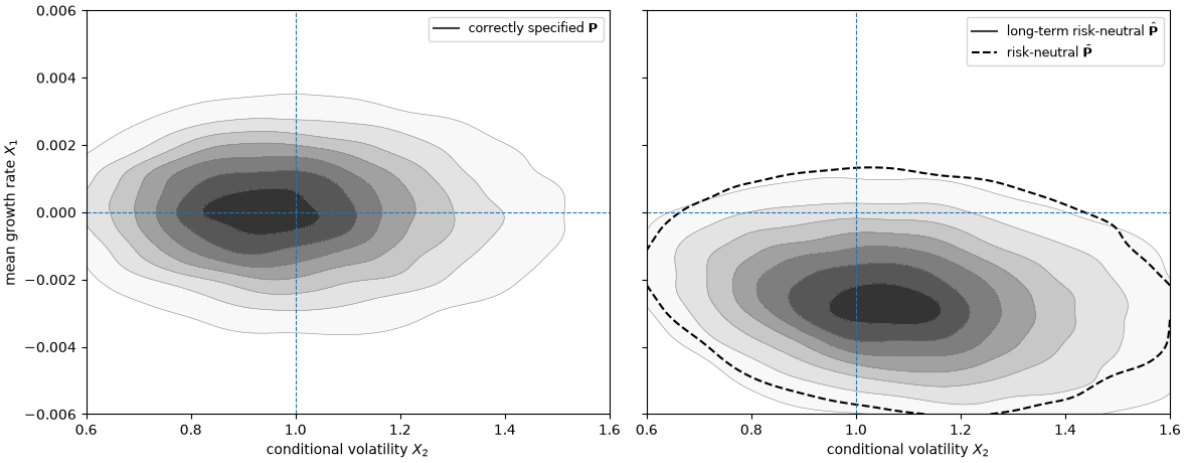
(continued from previous page)

```
            + sqrt_X2 * np.dot(dyn["σ1"], dW)
        )
        X2[t] = max(
            X2_prev
            + dyn["μ22"] * (X2_prev - dyn["ι2"])
            + sqrt_X2 * np.dot(dyn["σ2"], dW),
            1e-9,
        )

    burn = T // 5
    return X1[burn:], X2[burn:]


def kde2d_contour(ax, X1, X2, label, levels=7, fill=True,
                  linestyle='solid', outer_only=False):
    """Estimate the stationary density and draw its contours."""
    m = min(25_000, len(X1))
    idx = np.linspace(0, len(X1) - 1, m, dtype=int)
    x1 = X1[idx]
    x2 = X2[idx]

    kde = gaussian_kde(np.vstack([x2, x1]))
    x2_grid = np.linspace(0.6, 1.6, 140)
    x1_grid = np.linspace(-0.006, 0.006, 140)
    X2g, X1g = np.meshgrid(x2_grid, x1_grid)
    Z = kde(np.vstack([X2g.ravel(), X1g.ravel()])).reshape(X2g.shape)

    contour_levels = np.linspace(0.12 * Z.max(), 0.9 * Z.max(), levels)
    if outer_only:
        contour_levels = contour_levels[:1]

    if fill:
        fill_levels = np.r_[contour_levels, Z.max()]
        ax.contourf(X2g, X1g, Z, levels=fill_levels, cmap='Greys',
                    alpha=0.85)
        ax.contour(X2g, X1g, Z, levels=contour_levels, colors='0.55',
                   linewidths=0.4)
        ax.plot([], [], color='0.25', lw=1.5, label=label)
    else:
        ax.contour(X2g, X1g, Z, levels=contour_levels, colors='black',
                   linewidths=1.5, linestyles=linestyle)
        ax.plot([], [], color='black', lw=1.5, ls=linestyle, label=label)


dyn_true = dict(
    μ11=lrr_params["μ11"],
    μ12=lrr_params["μ12"],
    μ22=lrr_params["μ22"],
    ι1=lrr_params["ι1"],
    ι2=lrr_params["ι2"],
    σ1=lrr_params["σ1"],
    σ2=lrr_params["σ2"],
)

X1_P, X2_P = simulate_lrr(dyn_true, seed=1)
X1_H, X2_H = simulate_lrr(dyn_hat, seed=2)
X1_B, X2_B = simulate_lrr(dyn_bar, seed=3)
```

(continues on next page)

(continued from previous page)

```
fig, axes = plt.subplots(1, 2, figsize=(12, 4.8), sharex=True, sharey=True)
kde2d_contour(axes[0], X1_P, X2_P, label=r'correctly specified $\mathbf{P}$')
kde2d_contour(axes[1], X1_H, X2_H,
              label=r'long-term risk-neutral $\hat{\mathbf{P}}$')
kde2d_contour(axes[1], X1_B, X2_B,
              label=r'risk-neutral $\bar{\mathbf{P}}$',
              fill=False, linestyle='--', outer_only=True)

for ax in axes:
    ax.axhline(0, lw=0.8, ls='--')
    ax.axvline(lrr_params["ι2"], lw=0.8, ls='--')
    ax.set_xlim(0.6, 1.6)
    ax.set_ylim(-0.006, 0.006)
    ax.set_xlabel(r"conditional volatility $X_2$")
    ax.legend(fontsize=9)

axes[0].set_ylabel(r"mean growth rate $X_1$")
plt.tight_layout()
plt.show()
```



The movement below the horizontal line means lower expected growth, while movement to the right of the vertical line means higher volatility.

Yield implications

The difference between probability measures matters for asset-pricing interpretation because yields mix two quantities: a payoff forecast and an asset price.

The recovered probability measure is called long-term risk-neutral because it absorbs the martingale component that prices long-horizon risk.

For stochastically growing cash flows, the paper's long-horizon result is that risk premia relative to maturity-matched bonds vanish under the recovered probability measure, subject to the stability and moment conditions used for the limit.

Under the correctly specified probability measure, those same long-term risk premia need not vanish.

For a cash flow G_t, write expectations under the correctly specified probability measure as E_P and expectations under the recovered probability measure as $E_{\hat{P}}$.

The yield computed under the correctly specified probability measure is

$$y_t^P[G](x) = \frac{1}{t}\log E_P[G_t \mid X_0 = x] - \frac{1}{t}\log E_P[S_t G_t \mid X_0 = x].$$

The first term is the payoff forecast.

The second term is the asset price, written using the original SDF representation.

Arrow prices determine the second term.

The question here is what happens to the first term if an analyst treats the recovered probability measure $\hat{\mathbf{P}}$ as investors' beliefs.

In that comparison, prices are held fixed and only the forecast term is recomputed:

$$y_t^{\hat{P}}[G](x) = \frac{1}{t}\log E_{\hat{P}}[G_t \mid X_0 = x] - \frac{1}{t}\log E_P[S_t G_t \mid X_0 = x].$$

For an aggregate-consumption payoff, the answer is substantial.

The recovered probability measure assigns more probability to low-growth, high-volatility states, so it forecasts lower future consumption.

Holding prices fixed, that lower forecast translates into lower consumption yields.

The zero-coupon bond is the comparison case.

Its payoff is one, so the forecast term is always $\log E[1] = 0$.

Changing beliefs therefore does not move the bond-yield panel.

The same solution to the Perron–Frobenius problem also appears in long-bond and forward-measure limits.

The limiting one-period return on a very long bond is

$$R^{\infty}_{t,t+1} = \exp(-\hat{\eta})\frac{\hat{e}(X_{t+1})}{\hat{e}(X_t)}.$$

The martingale increment satisfies

$$\frac{\hat{H}_{t+1}}{\hat{H}_t} = \frac{S_{t+1}}{S_t} R^{\infty}_{t,t+1}.$$

Thus the limiting one-period transition from forward measures coincides with the transition associated with the long-term risk-neutral probability.

The calculation below uses the affine formulas implied by the long-run risk model.

If a multiplicative functional M has log drift affine in X and diffusion proportional to $\sqrt{X_2}$, then

$$E[M_t \mid X_0 = x] = \exp\{\theta_0(t) + \theta_1(t)x_1 + \theta_2(t)x_2\},$$

where the coefficients solve Riccati equations.

The code below computes these affine expectations under the correctly specified measure, recomputes only the consumption forecast under the recovered probability measure, and keeps asset prices fixed.

It then plots median and interquartile yield bands across the same simulated initial states.

```
def affine_expectation_coeffs(dyn, β0, β1, β2, α, horizons):
    """Riccati coefficients for log E[M_t | X_0=x]."""
    μ11, μ12, μ22 = dyn["μ11"], dyn["μ12"], dyn["μ22"]
    ι1, ι2 = dyn["ι1"], dyn["ι2"]
```

(continues on next page)

(continued from previous page)

```
    σ1, σ2 = dyn["σ1"], dyn["σ2"]

    def ode(_, θ):
        θ0, θ1, θ2 = θ
        θ0_dot = (β0 - β1 * ι1 - β2 * ι2
                  - θ1 * (μ11 * ι1 + μ12 * ι2)
                  - θ2 * μ22 * ι2)
        θ1_dot = β1 + μ11 * θ1
        θ2_dot = (β2 + μ12 * θ1 + μ22 * θ2
                  + 0.5 * np.dot(α, α)
                  + θ1 * np.dot(σ1, α)
                  + θ2 * np.dot(σ2, α)
                  + 0.5 * θ1**2 * np.dot(σ1, σ1)
                  + θ1 * θ2 * np.dot(σ1, σ2)
                  + 0.5 * θ2**2 * np.dot(σ2, σ2))
        return [θ0_dot, θ1_dot, θ2_dot]

    sol = solve_ivp(ode, (0, horizons[-1]), np.zeros(3),
                    t_eval=horizons, rtol=1e-8, atol=1e-10)
    if not sol.success:
        raise ValueError("Riccati equation failed to solve")
    return sol.y.T


def log_expectation(θ, X1, X2):
    """Evaluate log E[M_t | X_0=x] on simulated states."""
    return θ[:, 0, None] + θ[:, 1, None] * X1[None, :] + θ[:, 2, None] * X2[None, :]


def yield_quantiles(log_num, log_den, horizons):
    """Quartiles of annualized yields across initial states."""
    yields = 12 * (log_num - log_den) / horizons[:, None]
    return np.quantile(yields, [0.25, 0.5, 0.75], axis=1)


def transform_functional(β0, β1, β2, α, dyn_old, dyn_new, α_h):
    """Rewrite a multiplicative functional after changing probabilities."""
    # The drift changes because the martingale component changes the
    # Brownian shock exposure used to forecast the cash flow.
    β_level = β0 - β1 * dyn_old["ι1"] - β2 * dyn_old["ι2"]
    β2_new = β2 + np.dot(α, α_h)
    β0_new = β_level + β1 * dyn_new["ι1"] + β2_new * dyn_new["ι2"]
    return β0_new, β1, β2_new, α


def sdf_coefficients(p, v1, v2):
    """SDF coefficients used in the affine expectation calculation."""
    δ, γ = p["δ"], p["γ"]
    α_c, σ1, σ2 = p["α_c"], p["σ1"], p["σ2"]

    α_h_star = (1 - γ) * (α_c + σ1 * v1 + σ2 * v2)
    α_s = -α_c + α_h_star

    β_s1 = -p["β_c1"]
    β_s2 = -p["β_c2"] - 0.5 * np.dot(α_h_star, α_h_star)
    β_s0 = -δ - p["β_c0"] - 0.5 * p["ι2"] * np.dot(α_h_star, α_h_star)
```

(continues on next page)

(continued from previous page)

```
    return β_s0, β_s1, β_s2, α_s


quarters = np.arange(1, 101)
horizons = 3 * quarters

β_c0, β_c1, β_c2 = (lrr_params["β_c0"],
                    lrr_params["β_c1"],
                    lrr_params["β_c2"])
α_c = lrr_params["α_c"]

β_s0, β_s1, β_s2, α_s = sdf_coefficients(lrr_params, v1, v2)

# Numerators and denominators for yields under the correctly specified measure
θ_C_P = affine_expectation_coeffs(dyn_true, β_c0, β_c1, β_c2, α_c, horizons)
θ_S_P = affine_expectation_coeffs(dyn_true, β_s0, β_s1, β_s2, α_s, horizons)
θ_SC_P = affine_expectation_coeffs(
    dyn_true, β_s0 + β_c0, β_s1 + β_c1, β_s2 + β_c2,
    α_s + α_c, horizons
)

# Numerator for the aggregate-consumption payoff under the recovered probability␣
↪measure
β_Ch0, β_Ch1, β_Ch2, α_Ch = transform_functional(
    β_c0, β_c1, β_c2, α_c, dyn_true, dyn_hat, dyn_hat["α_h"]
)
θ_C_H = affine_expectation_coeffs(dyn_hat, β_Ch0, β_Ch1, β_Ch2,
                                  α_Ch, horizons)

log_C_P = log_expectation(θ_C_P, X1_P, X2_P)
log_C_H = log_expectation(θ_C_H, X1_P, X2_P)
log_S_P = log_expectation(θ_S_P, X1_P, X2_P)
log_SC_P = log_expectation(θ_SC_P, X1_P, X2_P)

qC_P = yield_quantiles(log_C_P, log_SC_P, horizons)
qC_H = yield_quantiles(log_C_H, log_SC_P, horizons)
qB_P = yield_quantiles(np.zeros_like(log_S_P), log_S_P, horizons)
# A zero-coupon payoff has the same numerator, log E[1] = 0, under either belief.
qB_H = qB_P.copy()

fig, axes = plt.subplots(1, 2, figsize=(12, 4.8), sharex=True)

def plot_yield_band(ax, x, q, color, label, linestyle='solid',
                    alpha=0.35):
    """Plot quartile band and quartile lines."""
    ax.fill_between(x, q[0], q[2], color=color, alpha=alpha, linewidth=0)
    ax.plot(x, q[1], color=color, lw=2.4, ls=linestyle, label=label)
    ax.plot(x, q[0], color=color, lw=1.3, ls=linestyle)
    ax.plot(x, q[2], color=color, lw=1.3, ls=linestyle)


plot_yield_band(axes[0], quarters, qC_P, color='0.2',
                label='correctly specified measure', alpha=0.45)
plot_yield_band(axes[0], quarters, qC_H, color='0.65',
                label='recovered probability measure', linestyle='--', alpha=0.35)
plot_yield_band(axes[1], quarters, qB_P, color='0.2',
                label='correctly specified measure', alpha=0.45)
```

(continues on next page)

(continued from previous page)

```
plot_yield_band(axes[1], quarters, qB_H, color='0.65',
                label='recovered probability measure', linestyle='--', alpha=0.25)

axes[0].set_xlabel('maturity (quarters)')
axes[0].set_ylabel('consumption yield to maturity')
axes[1].set_xlabel('maturity (quarters)')
axes[1].set_ylabel('bond yield to maturity')

axes[0].legend(fontsize=9)

plt.tight_layout()
plt.show()
```

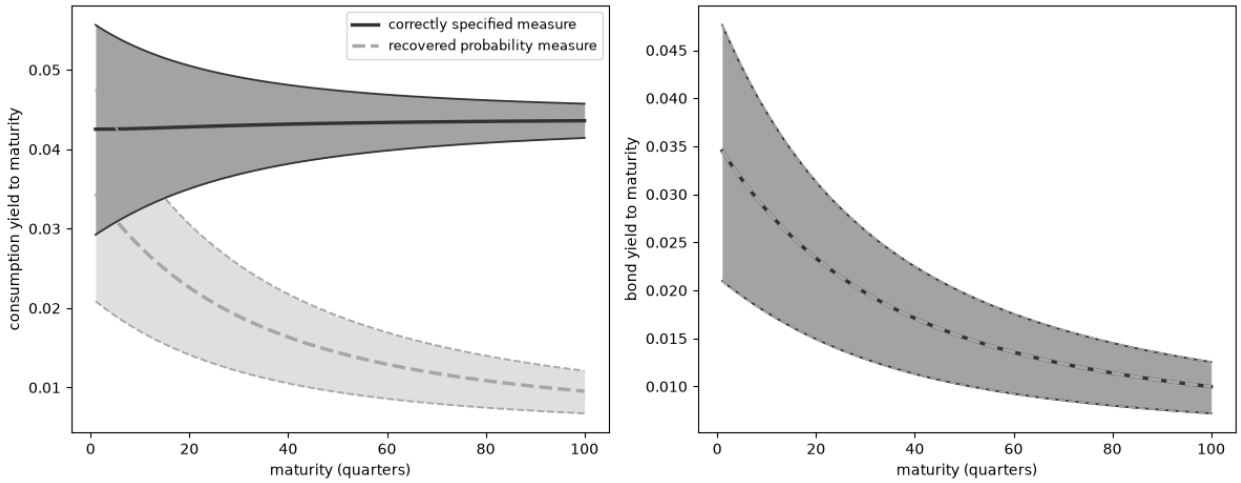


Fig. 128.2: Yield implications of using the recovered probability measure as beliefs. Dashed consumption-yield bands use payoff forecasts under the recovered probability measure with prices fixed; bond yields are unchanged because the zero-coupon payoff has no forecast term.

The left panel is the key one: treating the recovered probability measure as beliefs assigns more probability to low-growth, high-volatility states, so the implied forecast for consumption is lower and consumption yields fall when prices are held fixed.

The bond panel verifies the zero-coupon comparison.

Since $\log E[1] = 0$ under any measure, the solid and dashed bond-yield bands coincide.

128.6 Additional state vector

Borovička *et al.* [2016] then asks whether enlarging the state vector changes the recovery problem.

So far, the Perron–Frobenius eigenfunction has depended only on the Markov state X_t.

But many models also contain a growing component Y_t, such as log consumption, with increments driven by the same shock increments.

Here ΔW_{t+1} denotes the shock increment between dates t and $t+1$.

The map ϕ_x sends today's state and the next shock increment into tomorrow's state.

The map ϕ_y sends today's state and the next shock increment into the increment of Y.

$$X_{t+1} = \phi_x(X_t, \Delta W_{t+1}), \qquad Y_{t+1} - Y_t = \phi_y(X_t, \Delta W_{t+1}).$$

Let ε denote an eigenfunction candidate that is allowed to depend on both the stationary state X_t and the growing component Y_t.

Let ζ be a vector of loadings on Y, and let e_ζ be a positive function of X.

Then a natural candidate is

$$\varepsilon(x, y) = \exp(\zeta \cdot y) e_\zeta(x).$$

This form is natural because Y enters through increments.

Along a path,

$$\exp(\zeta \cdot Y_{t+1}) = \exp(\zeta \cdot Y_t) \exp\{\zeta \cdot (Y_{t+1} - Y_t)\}.$$

Since $Y_{t+1} - Y_t$ is a function of $(X_t, \Delta W_{t+1})$, the ratio $\exp(\zeta \cdot Y_{t+1}) / \exp(\zeta \cdot Y_t)$ is a one-period positive multiplicative functional increment.

For a fixed ζ, this factor tilts the one-period pricing operator by $\exp\{\zeta \cdot (Y_{t+1} - Y_t)\}$.

The x-dependent term is therefore not simply the earlier eigenfunction reused.

For each choice of ζ, the remaining x-dependent part solves a different Perron–Frobenius problem:

$$E\left[\frac{S_{t+1}}{S_t} \exp\{\zeta \cdot (Y_{t+1} - Y_t)\} e_\zeta(X_{t+1}) \mid X_t = x\right] = \exp(\eta_\zeta) e_\zeta(x).$$

Changing ζ changes how much long-run growth risk is loaded into the eigenfunction.

Thus adding Y_t can make the subjective probability measure one possible solution, but it also creates a family of possible solutions.

The extra state variable therefore does not remove the identification problem; it usually makes the selection problem more explicit.

The paper also points out a related practical issue.

Highly persistent stationary processes can be hard to distinguish from processes with stationary increments.

A stationary approximation may have a unique solution to the Perron–Frobenius problem for each finite persistence level, but as persistence becomes extreme, the limiting problem can have many approximate solutions.

Numerically, this means the solution to the Perron–Frobenius problem can be sensitive exactly in the cases where a stationary model is being used to approximate stochastic growth.

There is, however, a structured way forward.

If the analyst supplies a reference multiplicative functional Y^r that is known to have the same martingale component as the SDF, then one can restrict the enlarged eigenfunction to the form

$$(Y^r)^{-1} e(x).$$

This restriction chooses which long-run martingale component is allowed into the eigenfunction.

With this extra structure, Arrow prices can again reveal subjective probabilities.

But the key input is external: the long-run martingale component has been supplied by the analyst, not recovered from Arrow prices alone.

128.7 Measuring the martingale component

The paper also asks how large the martingale component is in asset-market data.

Under rational expectations, this measures how important long-term risk adjustments are for valuation.

Under a subjective-beliefs interpretation, it measures the discrepancy between subjective beliefs and the correctly specified probability measure only after imposing that the subjective SDF itself has no martingale component.

With that extra restriction in place, a small martingale component would make the recovered probability measure close to beliefs, while a large one would make long-term risk adjustments more important for the recovered probability measure.

One family of measures applies a convex function to the martingale increment $\hat{H}_{t+1}/\hat{H}_t$.

For example, conditional relative entropy uses

$$E\left[\frac{\hat{H}_{t+1}}{\hat{H}_t}\log\frac{\hat{H}_{t+1}}{\hat{H}_t}\mid X_t = x\right].$$

This expression is zero only when the martingale increment is identically one.

With incomplete asset-market data, the full martingale increment is not observed.

The paper therefore uses pricing restrictions and long-bond return approximations to derive lower bounds on such discrepancy measures.

These bounds are a way to test whether the martingale component is economically small without requiring a full set of Arrow prices.

128.8 Lessons

The Perron–Frobenius approach remains useful under misspecification, but it no longer solves the belief-recovery problem by itself.

It delivers a probability measure that may include long-horizon risk premia.

That measure equals investors' beliefs only when the martingale component is identically one.

Recursive utility, permanent shocks, and long-run risk models give this martingale an economically important role, so it should not be overlooked when assessing the implications of transition independence for belief recovery.

128.9 Exercises

Exercise 128.9.1

A two-state martingale component.

Let

$$\mathbf{P} = \begin{pmatrix} 0.8 & 0.2 \\ 0.4 & 0.6 \end{pmatrix}, \qquad \mathbf{Q} = \begin{pmatrix} 0.72 & 0.15 \\ 0.36 & 0.42 \end{pmatrix}.$$

1. Compute the one-period risk-neutral transition matrix $\bar{\mathbf{P}}$.
2. Compute the transition matrix $\hat{\mathbf{P}}$ associated with the recovered probability measure.

3. Compute $\hat{h}_{ij} = \hat{p}_{ij} / p_{ij}$ and decide whether recovery returns the correctly specified transition matrix.

Solution

Here is one solution:

```
P2 = np.array([[0.8, 0.2],
               [0.4, 0.6]])
Q2 = np.array([[0.72, 0.15],
               [0.36, 0.42]])

Pbar2, qb2 = risk_neutral_probs(Q2)
H2, eta2, e2, Phat2 = martingale_increment(Q2, P2)

print("One-period risk-neutral transition matrix P_bar")
print(np.round(Pbar2, 4))
print("\nTransition matrix P_hat associated with the recovered probability measure
 ↪")
print(np.round(Phat2, 4))
print("\nMartingale increment h_hat")
print(np.round(H2, 4))
print("\nRecovery returns P:", np.allclose(H2[P2 > 0], 1))
```

```
One-period risk-neutral transition matrix P_bar
[[0.8276 0.1724]
 [0.4615 0.5385]]

Transition matrix P_hat associated with the recovered probability measure
[[0.8505 0.1495]
 [0.5039 0.4961]]

Martingale increment h_hat
[[1.0631 0.7476]
 [1.2597 0.8269]]

Recovery returns P: False
```

Exercise 128.9.2

Power utility benchmark.

For trend-stationary consumption and power utility,

$$s_{ij} = A \left(\frac{c_j}{c_i} \right)^{-\gamma}.$$

Show that $\hat{e}_i = c_i^\gamma$ is the Perron–Frobenius eigenvector and that $\hat{\mathbf{P}} = \mathbf{P}$.

Then verify the result numerically using the three-state baseline in the lecture.

Solution

The analytical check is:

$$[\mathbf{Q}\hat{e}]_i = \sum_j A \left(\frac{c_j}{c_i}\right)^{-\gamma} p_{ij} c_j^{\gamma} = A c_i^{\gamma} = A\hat{e}_i.$$

Thus $\exp(\hat{\eta}) = A$ and

$$\hat{p}_{ij} = \frac{1}{A} q_{ij} \frac{\hat{e}_j}{\hat{e}_i} = p_{ij}.$$

Below is the numerical check.

```
H_power, _, e_power, P_hat_power = martingale_increment(Q_power, P_true)
e_theory = c_levels**γ_power
e_theory = e_theory / e_theory.sum()

print("Perron-Frobenius eigenvector")
print(np.round(e_power, 6))
print("\nNormalized c^gamma")
print(np.round(e_theory, 6))
print("\nmax |P_hat - P|:",
      np.max(np.abs(P_hat_power - P_true)))
print("max |h_hat - 1|:",
      np.max(np.abs(H_power[P_true > 0] - 1)))
```

```
Perron-Frobenius eigenvector
[0.328344 0.333313 0.338343]

Normalized c^gamma
[0.328344 0.333313 0.338343]

max |P_hat - P|: 3.3306690738754696e-16
max |h_hat - 1|: 1.3322676295501878e-15
```

Exercise 128.9.3

Recursive utility and risk aversion.

Using the finite-state Epstein–Zin example with $c = (0.85, 1.00, 1.15)$, compute the stationary distribution of $\hat{\mathbf{P}}$ for $\gamma \in \{1, 5, 10, 15\}$.

Which state receives the largest increase in stationary probability as γ rises?

Solution

Here is one solution:

```
for γ in [1, 5, 10, 15]:
    _, _, S_g = solve_ez_unit_eis(P_true, c_recursive, δ, γ, g_c)
    Q_g = S_g * P_true
    _, _, _, P_hat_g = martingale_increment(Q_g, P_true)
    π_g = stationary_dist(P_hat_g)
    print(f"gamma={γ:2.0f}: {np.round(π_g, 4)}")

print("\nCorrectly specified:", np.round(π_true, 4))
```

```
gamma= 1: [0.2688 0.4409 0.2903]
gamma= 5: [0.2766 0.4399 0.2835]
gamma=10: [0.2865 0.4385 0.275 ]
gamma=15: [0.2966 0.4368 0.2666]

Correctly specified: [0.2688 0.4409 0.2903]
```

The recession state receives the largest increase.

Part XVII

Data and Empirics

CHAPTER

NINE

PANDAS FOR PANEL DATA

Contents

- *Pandas for Panel Data*
  - *Overview*
  - *Slicing and Reshaping Data*
  - *Merging Dataframes and Filling NaNs*
  - *Grouping and Summarizing Data*
  - *Final Remarks*
  - *Exercises*

129.1 Overview

In an earlier lecture on pandas, we looked at working with simple data sets.

Econometricians often need to work with more complex data sets, such as panels.

Common tasks include

- Importing data, cleaning it and reshaping it across several axes.
- Selecting a time series or cross-section from a panel.
- Grouping and summarizing data.

`pandas` (derived from 'panel' and 'data') contains powerful and easy-to-use tools for solving exactly these kinds of problems.

In what follows, we will use a panel data set of real minimum wages from the OECD to create:

- summary statistics over multiple dimensions of our data
- a time series of the average minimum wage of countries in the dataset
- kernel density estimates of wages by continent

We will begin by reading in our long format panel data from a CSV file and reshaping the resulting `DataFrame` with `pivot_table` to build a `MultiIndex`.

Additional detail will be added to our `DataFrame` using pandas' `merge` function, and data will be summarized with the `groupby` function.

129.2 Slicing and Reshaping Data

We will read in a dataset from the OECD of real minimum wages in 32 countries and assign it to `realwage`.

The dataset can be accessed with the following link:

```
url1 = 'https://github.com/QuantEcon/data-lectures/raw/main/lectures/realwage.csv'
```

```
import pandas as pd

# Display 6 columns for viewing purposes
pd.set_option('display.max_columns', 6)

# Reduce decimal points to 2
pd.options.display.float_format = '{:,.2f}'.format

realwage = pd.read_csv(url1)
```

Let's have a look at what we've got to work with

```
realwage.head()  # Show first 5 rows
```

```
   Unnamed: 0        Time  Country                                    Series  \
0           0  2006-01-01  Ireland  In 2015 constant prices at 2015 USD PPPs
1           1  2007-01-01  Ireland  In 2015 constant prices at 2015 USD PPPs
2           2  2008-01-01  Ireland  In 2015 constant prices at 2015 USD PPPs
3           3  2009-01-01  Ireland  In 2015 constant prices at 2015 USD PPPs
4           4  2010-01-01  Ireland  In 2015 constant prices at 2015 USD PPPs

  Pay period     value
0     Annual 17,132.44
1     Annual 18,100.92
2     Annual 17,747.41
3     Annual 18,580.14
4     Annual 18,755.83
```

The data is currently in long format, which is difficult to analyze when there are several dimensions to the data.

We will use `pivot_table` to create a wide format panel, with a `MultiIndex` to handle higher dimensional data.

`pivot_table` arguments should specify the data (values), the index, and the columns we want in our resulting dataframe.

By passing a list in columns, we can create a `MultiIndex` in our column axis

```
realwage = realwage.pivot_table(values='value',
                                index='Time',
                                columns=['Country', 'Series', 'Pay period'])
realwage.head()
```

```
Country                                    Australia         \
Series     In 2015 constant prices at 2015 USD PPPs
Pay period                                    Annual Hourly
```

(continues on next page)

(continued from previous page)

```
Time
2006-01-01                                  20,410.65  10.33
2007-01-01                                  21,087.57  10.67
2008-01-01                                  20,718.24  10.48
2009-01-01                                  20,984.77  10.62
2010-01-01                                  20,879.33  10.57

Country                                                                 ...  \
Series     In 2015 constant prices at 2015 USD exchange rates           ...
Pay period                                                   Annual     ...
Time                                                                    ...
2006-01-01                                                23,826.64     ...
2007-01-01                                                24,616.84     ...
2008-01-01                                                24,185.70     ...
2009-01-01                                                24,496.84     ...
2010-01-01                                                24,373.76     ...

Country                                 United States  \
Series     In 2015 constant prices at 2015 USD PPPs
Pay period                                     Hourly
Time
2006-01-01                                       6.05
2007-01-01                                       6.24
2008-01-01                                       6.78
2009-01-01                                       7.58
2010-01-01                                       7.88

Country
Series     In 2015 constant prices at 2015 USD exchange rates
Pay period                                                   Annual Hourly
Time
2006-01-01                                                12,594.40   6.05
2007-01-01                                                12,974.40   6.24
2008-01-01                                                14,097.56   6.78
2009-01-01                                                15,756.42   7.58
2010-01-01                                                16,391.31   7.88

[5 rows x 128 columns]
```

To more easily filter our time series data, later on, we will convert the index into a `DateTimeIndex`

```
realwage.index = pd.to_datetime(realwage.index)
type(realwage.index)
```

```
pandas.DatetimeIndex
```

The columns contain multiple levels of indexing, known as a `MultiIndex`, with levels being ordered hierarchically (Country > Series > Pay period).

A `MultiIndex` is the simplest and most flexible way to manage panel data in pandas

```
type(realwage.columns)
```

```
pandas.MultiIndex
```

```
realwage.columns.names
```

```
FrozenList(['Country', 'Series', 'Pay period'])
```

Like before, we can select the country (the top level of our `MultiIndex`)

```
realwage['United States'].head()
```

```
Series     In 2015 constant prices at 2015 USD PPPs         \
Pay period                                    Annual Hourly
Time
2006-01-01                                 12,594.40   6.05
2007-01-01                                 12,974.40   6.24
2008-01-01                                 14,097.56   6.78
2009-01-01                                 15,756.42   7.58
2010-01-01                                 16,391.31   7.88

Series     In 2015 constant prices at 2015 USD exchange rates
Pay period                                              Annual Hourly
Time
2006-01-01                                           12,594.40   6.05
2007-01-01                                           12,974.40   6.24
2008-01-01                                           14,097.56   6.78
2009-01-01                                           15,756.42   7.58
2010-01-01                                           16,391.31   7.88
```

Stacking and unstacking levels of the `MultiIndex` will be used throughout this lecture to reshape our dataframe into a format we need.

`.stack()` rotates the lowest level of the column `MultiIndex` to the row index (`.unstack()` works in the opposite direction - try it out)

```
realwage.stack(future_stack=True).head()
```

```
Country                                                Australia  \
Series                 In 2015 constant prices at 2015 USD PPPs
Time       Pay period
2006-01-01 Annual                                      20,410.65
           Hourly                                          10.33
2007-01-01 Annual                                      21,087.57
           Hourly                                          10.67
2008-01-01 Annual                                      20,718.24

Country                                                                    \
Series                 In 2015 constant prices at 2015 USD exchange rates
Time       Pay period
2006-01-01 Annual                                                 23,826.64
           Hourly                                                     12.06
2007-01-01 Annual                                                 24,616.84
           Hourly                                                     12.46
2008-01-01 Annual                                                 24,185.70

Country                                                  Belgium  ...  \
Series                 In 2015 constant prices at 2015 USD PPPs  ...
Time       Pay period                                            ...
2006-01-01 Annual                                      21,042.28  ...
```

(continues on next page)

(continued from previous page)

```
           Hourly                                          10.09  ...
2007-01-01 Annual                                      21,310.05  ...
           Hourly                                          10.22  ...
2008-01-01 Annual                                      21,416.96  ...

Country                                                  United Kingdom  \
Series                 In 2015 constant prices at 2015 USD exchange rates
Time       Pay period
2006-01-01 Annual                                                20,376.32
           Hourly                                                     9.81
2007-01-01 Annual                                                20,954.13
           Hourly                                                    10.07
2008-01-01 Annual                                                20,902.87

Country                                           United States  \
Series                 In 2015 constant prices at 2015 USD PPPs
Time       Pay period
2006-01-01 Annual                                      12,594.40
           Hourly                                           6.05
2007-01-01 Annual                                      12,974.40
           Hourly                                           6.24
2008-01-01 Annual                                      14,097.56

Country
Series                 In 2015 constant prices at 2015 USD exchange rates
Time       Pay period
2006-01-01 Annual                                                12,594.40
           Hourly                                                     6.05
2007-01-01 Annual                                                12,974.40
           Hourly                                                     6.24
2008-01-01 Annual                                                14,097.56

[5 rows x 64 columns]
```

We can also pass in an argument to select the level we would like to stack

```
realwage.stack(level='Country', future_stack=True).head()
```

```
Series                In 2015 constant prices at 2015 USD PPPs         \
Pay period                                              Annual Hourly
Time       Country
2006-01-01 Australia                                 20,410.65  10.33
           Belgium                                   21,042.28  10.09
           Brazil                                     3,310.51   1.41
           Canada                                    13,649.69   6.56
           Chile                                      5,201.65   2.22

Series                In 2015 constant prices at 2015 USD exchange rates
Pay period                                                      Annual Hourly
Time       Country
2006-01-01 Australia                                          23,826.64  12.06
           Belgium                                            20,228.74   9.70
           Brazil                                              2,032.87   0.87
           Canada                                             14,335.12   6.89
           Chile                                               3,333.76   1.42
```

Using a `DatetimeIndex` makes it easy to select a particular time period.

Selecting one year and stacking the two lower levels of the `MultiIndex` creates a cross-section of our panel data

```
realwage.loc['2015'].stack(level=(1, 2), future_stack=True).transpose().head()
```

```
Time                                       2015-01-01         \
Series     In 2015 constant prices at 2015 USD PPPs
Pay period                                    Annual Hourly
Country
Australia                                 21,715.53  10.99
Belgium                                   21,588.12  10.35
Brazil                                     4,628.63   2.00
Canada                                    16,536.83   7.95
Chile                                      6,633.56   2.80

Time
Series     In 2015 constant prices at 2015 USD exchange rates
Pay period                                                Annual Hourly
Country
Australia                                                 25,349.90  12.83
Belgium                                                   20,753.48   9.95
Brazil                                                     2,842.28   1.21
Canada                                                    17,367.24   8.35
Chile                                                      4,251.49   1.81
```

For the rest of lecture, we will work with a dataframe of the hourly real minimum wages across countries and time, measured in 2015 US dollars.

To create our filtered dataframe (`realwage_f`), we can use the `xs` method to select values at lower levels in the multiindex, while keeping the higher levels (countries in this case)

```
realwage_f = realwage.xs(('Hourly', 'In 2015 constant prices at 2015 USD exchange␣
↪rates'),
                         level=('Pay period', 'Series'), axis=1)
realwage_f.head()
```

```
Country     Australia  Belgium  Brazil  ...  Turkey  United Kingdom  \
Time                                    ...
2006-01-01      12.06     9.70    0.87  ...    2.27            9.81
2007-01-01      12.46     9.82    0.92  ...    2.26           10.07
2008-01-01      12.24     9.87    0.96  ...    2.22           10.04
2009-01-01      12.40    10.21    1.03  ...    2.28           10.15
2010-01-01      12.34    10.05    1.08  ...    2.30            9.96

Country     United States
Time
2006-01-01           6.05
2007-01-01           6.24
2008-01-01           6.78
2009-01-01           7.58
2010-01-01           7.88

[5 rows x 32 columns]
```

129.3 Merging Dataframes and Filling NaNs

Similar to relational databases like SQL, pandas has built in methods to merge datasets together.

Using country information from WorldData.info, we'll add the continent of each country to `realwage_f` with the `merge` function.

The dataset can be accessed with the following link:

```
url2 = 'https://github.com/QuantEcon/data-lectures/raw/main/lectures/countries.csv'
```

```
worlddata = pd.read_csv(url2, sep=';')
worlddata.head()
```

```
    Country (en) Country (de)        Country (local)  ... Deathrate  \
0    Afghanistan  Afghanistan  Afganistan/Afqanestan  ...     13.70
1          Egypt      Ägypten                  Misr  ...      4.70
2  Åland Islands  Ålandinseln                 Åland  ...      0.00
3        Albania     Albanien              Shqipëria  ...      6.70
4        Algeria     Algerien     Al-Jaza'ir/Algérie  ...      4.30

  Life expectancy                                                Url
0           51.30  https://www.laenderdaten.info/Asien/Afghanista...
1           72.70  https://www.laenderdaten.info/Afrika/Aegypten/...
2            0.00  https://www.laenderdaten.info/Europa/Aland/ind...
3           78.30  https://www.laenderdaten.info/Europa/Albanien/...
4           76.80  https://www.laenderdaten.info/Afrika/Algerien/...

[5 rows x 17 columns]
```

First, we'll select just the country and continent variables from `worlddata` and rename the column to 'Country'

```
worlddata = worlddata[['Country (en)', 'Continent']]
worlddata = worlddata.rename(columns={'Country (en)': 'Country'})
worlddata.head()
```

```
         Country Continent
0    Afghanistan      Asia
1          Egypt    Africa
2  Åland Islands    Europe
3        Albania    Europe
4        Algeria    Africa
```

We want to merge our new dataframe, `worlddata`, with `realwage_f`.

The pandas `merge` function allows dataframes to be joined together by rows.

Our dataframes will be merged using country names, requiring us to use the transpose of `realwage_f` so that rows correspond to country names in both dataframes

```
realwage_f.transpose().head()
```

```
Time       2006-01-01  2007-01-01  2008-01-01  ...  2014-01-01  2015-01-01  \
Country                                        ...
Australia       12.06       12.46       12.24  ...       12.67       12.83
Belgium          9.70        9.82        9.87  ...       10.01        9.95
Brazil           0.87        0.92        0.96  ...        1.21        1.21
```

(continues on next page)

(continued from previous page)

```
Canada            6.89        6.96        7.24  ...        8.22        8.35
Chile             1.42        1.45        1.44  ...        1.76        1.81

Time       2016-01-01
Country
Australia       12.98
Belgium          9.76
Brazil           1.24
Canada           8.48
Chile            1.91

[5 rows x 11 columns]
```

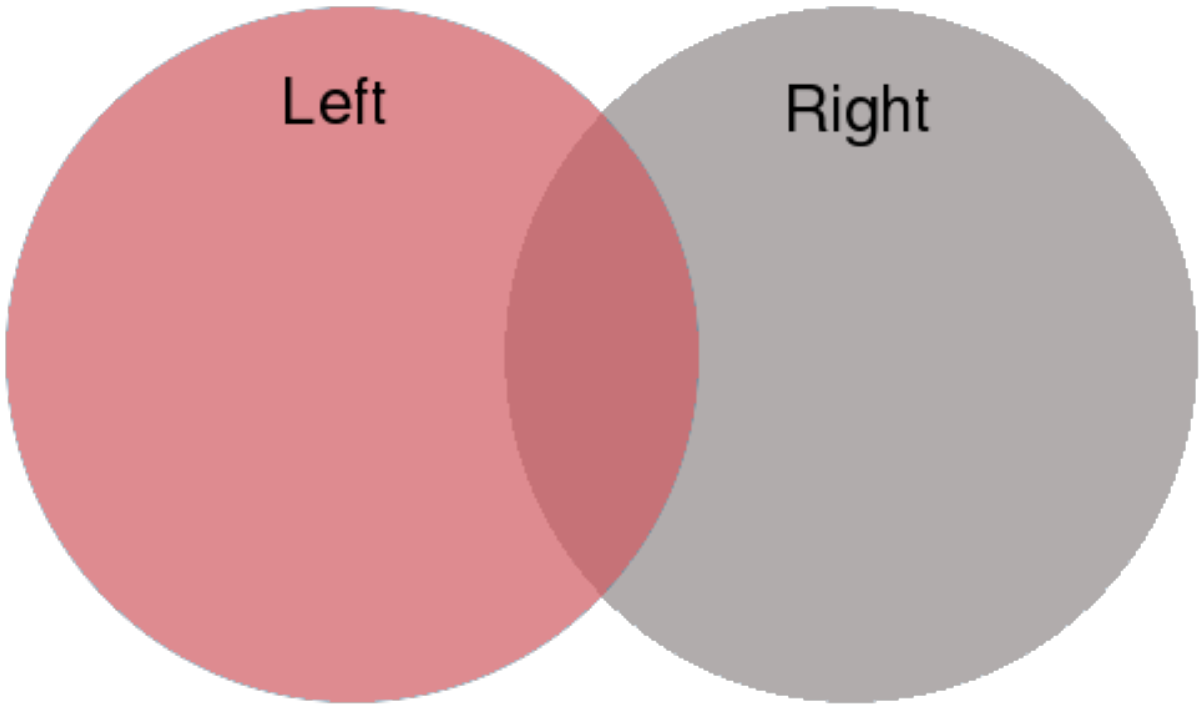
We can use either left, right, inner, or outer join to merge our datasets:

- left join includes only countries from the left dataset
- right join includes only countries from the right dataset
- outer join includes countries that are in either the left and right datasets
- inner join includes only countries common to both the left and right datasets

By default, `merge` will use an inner join.

Here we will pass `how='left'` to keep all countries in `realwage_f`, but discard countries in `worlddata` that do not have a corresponding data entry `realwage_f`.

This is illustrated by the red shading in the following diagram



We will also need to specify where the country name is located in each dataframe, which will be the `key` that is used to

merge the dataframes 'on'.

Our 'left' dataframe (`realwage_f.transpose()`) contains countries in the index, so we set `left_index=True`.

Our 'right' dataframe (`worlddata`) contains countries in the 'Country' column, so we set `right_on='Country'`

```
merged = pd.merge(realwage_f.transpose(), worlddata,
                  how='left', left_index=True, right_on='Country')
merged.head()
```

```
        2006-01-01 00:00:00  2007-01-01 00:00:00  2008-01-01 00:00:00  ...  \
17.00                 12.06                12.46                12.24  ...
23.00                  9.70                 9.82                 9.87  ...
32.00                  0.87                 0.92                 0.96  ...
100.00                 6.89                 6.96                 7.24  ...
38.00                  1.42                 1.45                 1.44  ...

        2016-01-01 00:00:00    Country      Continent
17.00                 12.98  Australia      Australia
23.00                  9.76    Belgium         Europe
32.00                  1.24     Brazil  South America
100.00                 8.48     Canada  North America
38.00                  1.91      Chile  South America

[5 rows x 13 columns]
```

Countries that appeared in `realwage_f` but not in `worlddata` will have `NaN` in the Continent column.

To check whether this has occurred, we can use `.isnull()` on the continent column and filter the merged dataframe

```
merged[merged['Continent'].isnull()]
```

```
     2006-01-01 00:00:00  2007-01-01 00:00:00  2008-01-01 00:00:00  ...  \
NaN                 3.42                 3.74                 3.87  ...
NaN                 0.23                 0.45                 0.39  ...
NaN                 1.50                 1.64                 1.71  ...

     2016-01-01 00:00:00             Country  Continent
NaN                 5.28               Korea        NaN
NaN                 0.55  Russian Federation        NaN
NaN                 2.08     Slovak Republic        NaN

[3 rows x 13 columns]
```

We have three missing values!

One option to deal with NaN values is to create a dictionary containing these countries and their respective continents.

`.map()` will match countries in `merged['Country']` with their continent from the dictionary.

Notice how countries not in our dictionary are mapped with `NaN`

```
missing_continents = {'Korea': 'Asia',
                      'Russian Federation': 'Europe',
                      'Slovak Republic': 'Europe'}

merged['Country'].map(missing_continents)
```

```
17.00         NaN
23.00         NaN
32.00         NaN
100.00        NaN
38.00         NaN
108.00        NaN
41.00         NaN
225.00        NaN
53.00         NaN
58.00         NaN
45.00         NaN
68.00         NaN
233.00        NaN
86.00         NaN
88.00         NaN
91.00         NaN
NaN          Asia
117.00        NaN
122.00        NaN
123.00        NaN
138.00        NaN
153.00        NaN
151.00        NaN
174.00        NaN
175.00        NaN
NaN        Europe
NaN        Europe
198.00        NaN
200.00        NaN
227.00        NaN
241.00        NaN
240.00        NaN
Name: Country, dtype: str
```

We don't want to overwrite the entire series with this mapping.

`.fillna()` only fills in `NaN` values in `merged['Continent']` with the mapping, while leaving other values in the column unchanged

```
merged['Continent'] = merged['Continent'].fillna(merged['Country'].map(missing_
↪continents))

# Check for whether continents were correctly mapped

merged[merged['Country'] == 'Korea']
```

```
     2006-01-01 00:00:00  2007-01-01 00:00:00  2008-01-01 00:00:00  ...  \
NaN                 3.42                 3.74                 3.87  ...

     2016-01-01 00:00:00  Country  Continent
NaN                 5.28    Korea       Asia

[1 rows x 13 columns]
```

We will also combine the Americas into a single continent - this will make our visualization nicer later on.

To do this, we will use `.replace()` and loop through a list of the continent values we want to replace

```
replace = ['Central America', 'North America', 'South America']

for country in replace:
    merged.Continent = merged.Continent.replace(to_replace=country,
                                value='America')
```

Now that we have all the data we want in a single `DataFrame`, we will reshape it back into panel form with a `MultiIndex`.

We should also ensure to sort the index using `.sort_index()` so that we can efficiently filter our dataframe later on.

By default, levels will be sorted top-down

```
merged = merged.set_index(['Continent', 'Country']).sort_index()
merged.head()
```

```
                     2006-01-01 00:00:00  2007-01-01 00:00:00  \
Continent Country
America   Brazil                    0.87                 0.92
          Canada                    6.89                 6.96
          Chile                     1.42                 1.45
          Colombia                  1.01                 1.02
          Costa Rica                 NaN                  NaN

                     2008-01-01 00:00:00  ...  2014-01-01 00:00:00  \
Continent Country                         ...
America   Brazil                    0.96  ...                 1.21
          Canada                    7.24  ...                 8.22
          Chile                     1.44  ...                 1.76
          Colombia                  1.01  ...                 1.13
          Costa Rica                 NaN  ...                 2.41

                     2015-01-01 00:00:00  2016-01-01 00:00:00
Continent Country
America   Brazil                    1.21                 1.24
          Canada                    8.35                 8.48
          Chile                     1.81                 1.91
          Colombia                  1.13                 1.12
          Costa Rica                2.56                 2.63

[5 rows x 11 columns]
```

While merging, we lost our `DatetimeIndex`, as we merged columns that were not in datetime format

```
merged.columns
```

```
Index([2006-01-01 00:00:00, 2007-01-01 00:00:00, 2008-01-01 00:00:00,
       2009-01-01 00:00:00, 2010-01-01 00:00:00, 2011-01-01 00:00:00,
       2012-01-01 00:00:00, 2013-01-01 00:00:00, 2014-01-01 00:00:00,
       2015-01-01 00:00:00, 2016-01-01 00:00:00],
      dtype='object')
```

Now that we have set the merged columns as the index, we can recreate a `DatetimeIndex` using `.to_datetime()`

```
merged.columns = pd.to_datetime(merged.columns)
merged.columns = merged.columns.rename('Time')
merged.columns
```

```
DatetimeIndex(['2006-01-01', '2007-01-01', '2008-01-01', '2009-01-01',
               '2010-01-01', '2011-01-01', '2012-01-01', '2013-01-01',
               '2014-01-01', '2015-01-01', '2016-01-01'],
              dtype='datetime64[us]', name='Time', freq=None)
```

The `DatetimeIndex` tends to work more smoothly in the row axis, so we will go ahead and transpose `merged`

```
merged = merged.transpose()
merged.head()
```

```
Continent  America                ...   Europe
Country     Brazil Canada Chile   ... Slovenia Spain United Kingdom
Time                              ...
2006-01-01    0.87   6.89  1.42   ...     3.92  3.99           9.81
2007-01-01    0.92   6.96  1.45   ...     3.88  4.10          10.07
2008-01-01    0.96   7.24  1.44   ...     3.96  4.14          10.04
2009-01-01    1.03   7.67  1.52   ...     4.08  4.32          10.15
2010-01-01    1.08   7.94  1.56   ...     4.81  4.30           9.96

[5 rows x 32 columns]
```

129.4 Grouping and Summarizing Data

Grouping and summarizing data can be particularly useful for understanding large panel datasets.

A simple way to summarize data is to call an aggregation method on the dataframe, such as `.mean()` or `.max()`.

For example, we can calculate the average real minimum wage for each country over the period 2006 to 2016 (the default is to aggregate over rows)

```
merged.mean().head(10)
```

```
Continent  Country
America    Brazil          1.09
           Canada          7.82
           Chile           1.62
           Colombia        1.07
           Costa Rica      2.53
           Mexico          0.53
           United States   7.15
Asia       Israel          5.95
           Japan           6.18
           Korea           4.22
dtype: float64
```

Using this series, we can plot the average real minimum wage over the past decade for each country in our data set

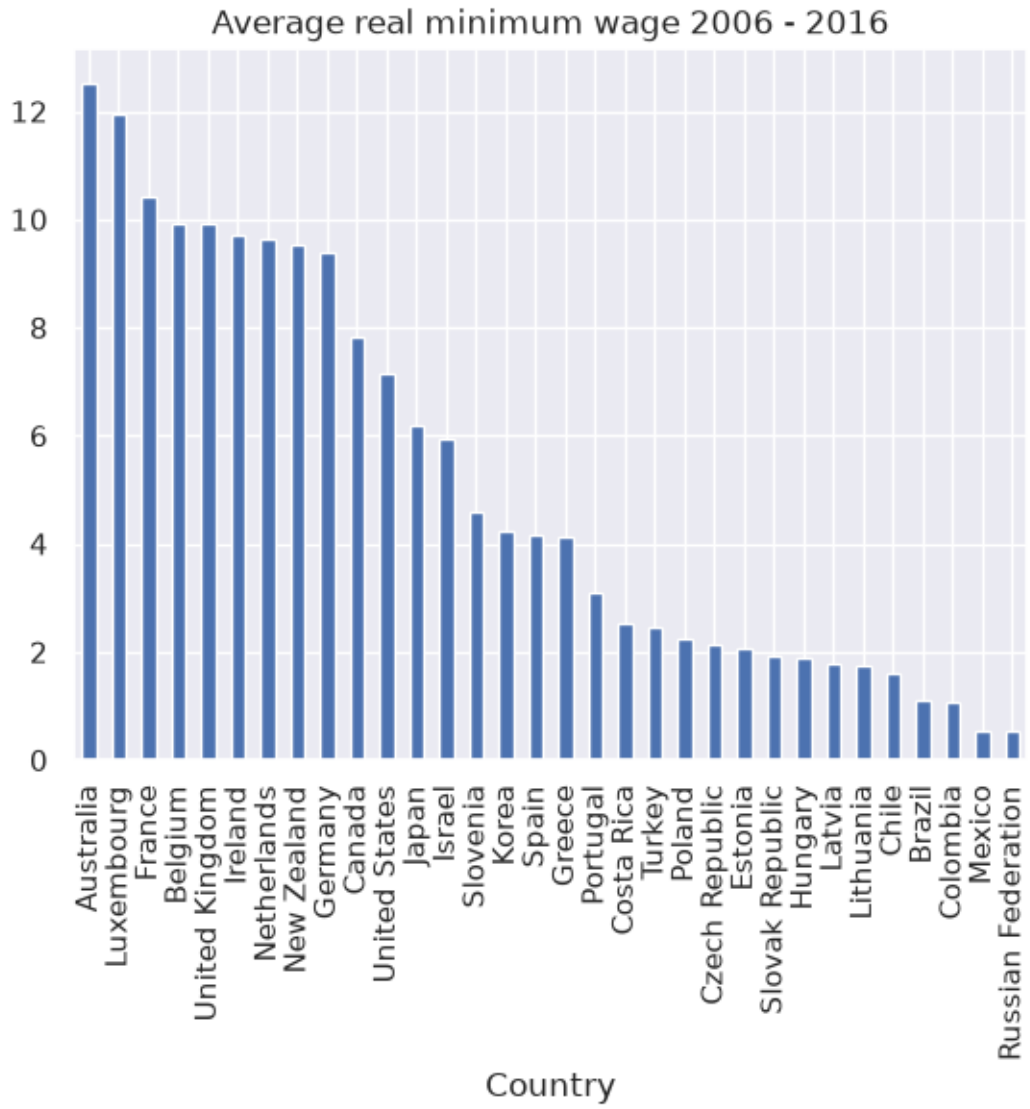
```
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_theme()
```

```
merged.mean().sort_values(ascending=False).plot(kind='bar',
                                                title="Average real minimum wage 2006␣
 ↪- 2016")
```

(continues on next page)

(continued from previous page)

```
# Set country labels
country_labels = merged.mean().sort_values(ascending=False).index.get_level_values(
↪'Country').tolist()
plt.xticks(range(0, len(country_labels)), country_labels)
plt.xlabel('Country')

plt.show()
```



Passing in `axis=1` to `.mean()` will aggregate over columns (giving the average minimum wage for all countries over time)

```
merged.mean(axis=1).head()
```

```
Time
2006-01-01    4.69
2007-01-01    4.84
```

(continues on next page)

(continued from previous page)

```
2008-01-01    4.90
2009-01-01    5.08
2010-01-01    5.11
dtype: float64
```

We can plot this time series as a line graph

```
merged.mean(axis=1).plot()
plt.title('Average real minimum wage 2006 - 2016')
plt.ylabel('2015 USD')
plt.xlabel('Year')
plt.show()
```

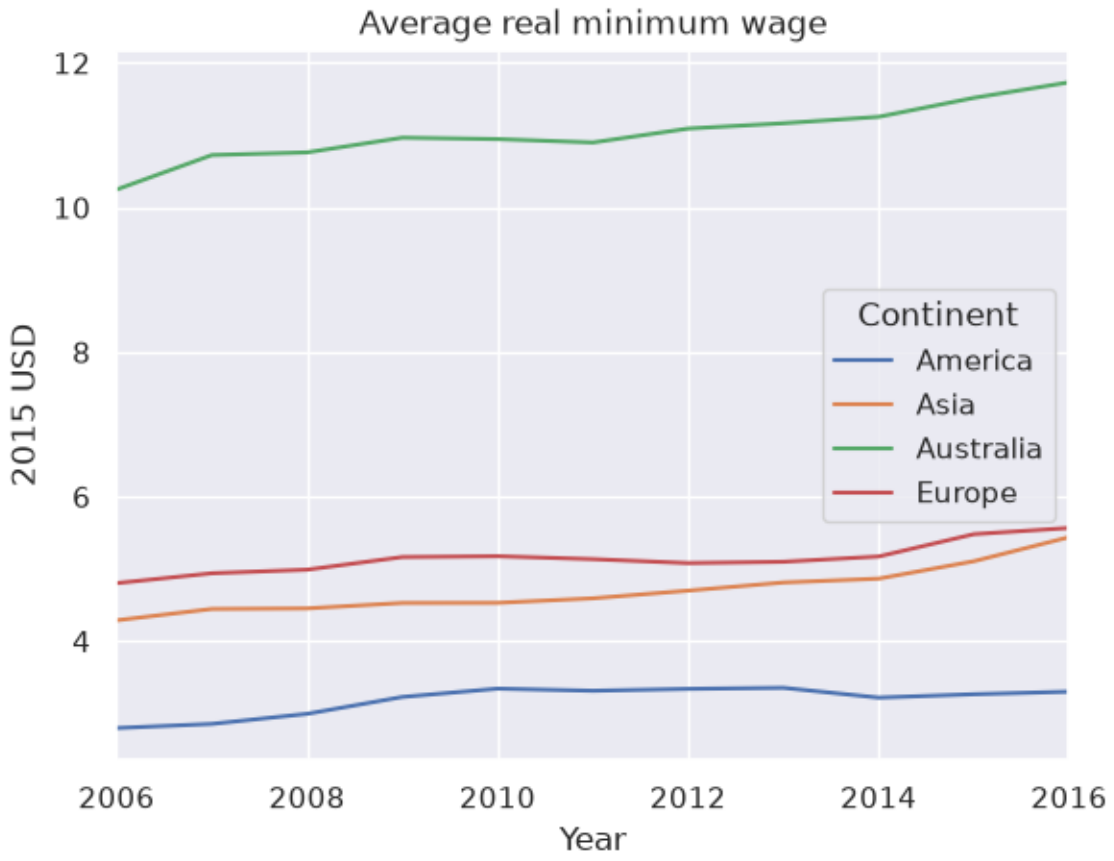
We can also specify a level of the `MultiIndex` (in the column axis) to aggregate over

```
merged.T.groupby(level='Continent').mean().T.head()
```

```
Continent   America  Asia  Australia  Europe
Time
2006-01-01     2.80  4.29      10.25    4.80
2007-01-01     2.85  4.44      10.73    4.94
2008-01-01     2.99  4.45      10.76    4.99
2009-01-01     3.23  4.53      10.97    5.16
2010-01-01     3.34  4.53      10.95    5.17
```

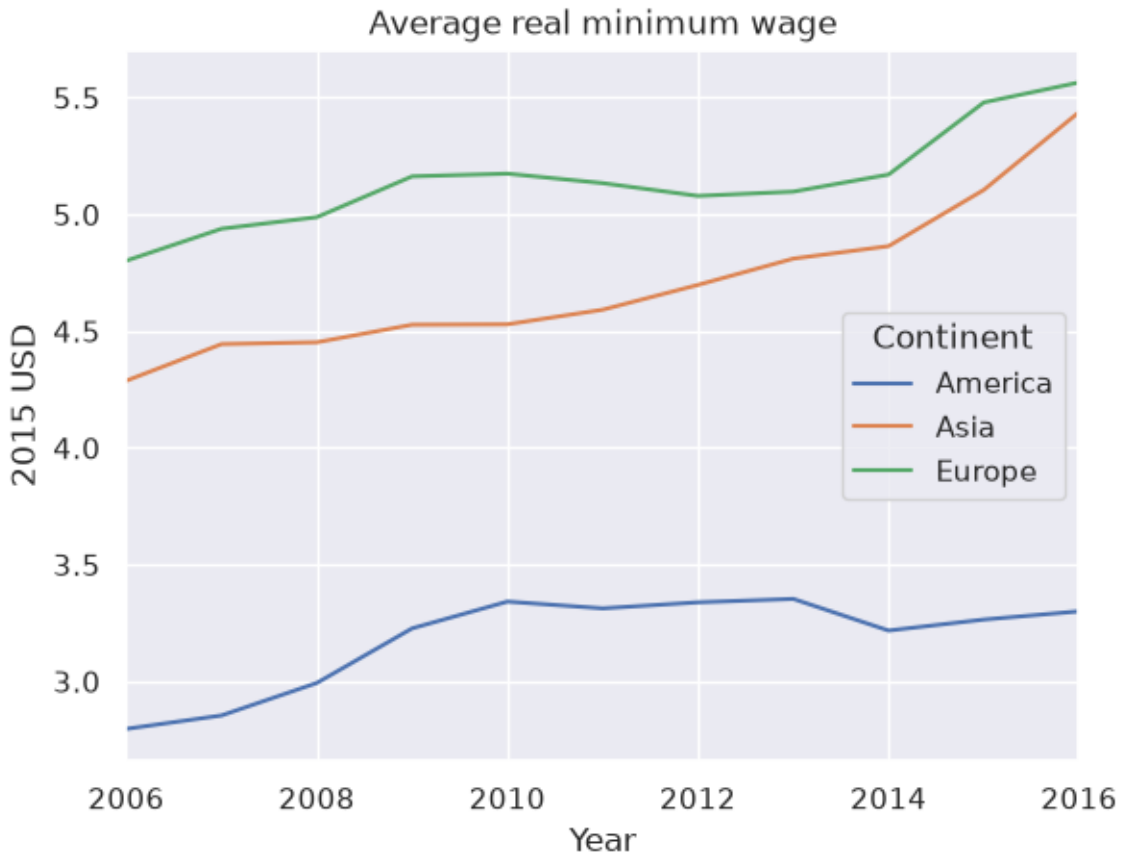
We can plot the average minimum wages in each continent as a time series

```
merged.T.groupby(level='Continent').mean().T.plot()
plt.title('Average real minimum wage')
plt.ylabel('2015 USD')
plt.xlabel('Year')
plt.show()
```



We will drop Australia as a continent for plotting purposes

```
merged = merged.drop('Australia', level='Continent', axis=1)
merged.T.groupby(level='Continent').mean().T.plot()
plt.title('Average real minimum wage')
plt.ylabel('2015 USD')
plt.xlabel('Year')
plt.show()
```



`.describe()` is useful for quickly retrieving a number of common summary statistics

```
merged.stack(future_stack=True).describe()
```

```
Continent  America  Asia  Europe
count        69.00 44.00  200.00
mean          3.19  4.70    5.15
std           3.02  1.56    3.82
min           0.52  2.22    0.23
25%           1.03  3.37    2.02
50%           1.44  5.48    3.54
75%           6.96  5.95    9.70
max           8.48  6.65   12.39
```

This is a simplified way to use `groupby`.

Using `groupby` generally follows a 'split-apply-combine' process:

- split: data is grouped based on one or more keys
- apply: a function is called on each group independently
- combine: the results of the function calls are combined into a new data structure

The `groupby` method achieves the first step of this process, creating a new `DataFrameGroupBy` object with data split into groups.

Let's split `merged` by continent again, this time using the `groupby` function, and name the resulting object `grouped`

```
grouped = merged.T.groupby(level='Continent')
grouped
```

```
<pandas.api.typing.DataFrameGroupBy object at 0x7c443b77fbf0>
```

Calling an aggregation method on the object applies the function to each group, the results of which are combined in a new data structure.

For example, we can return the number of countries in our dataset for each continent using `.size()`.

In this case, our new data structure is a `Series`

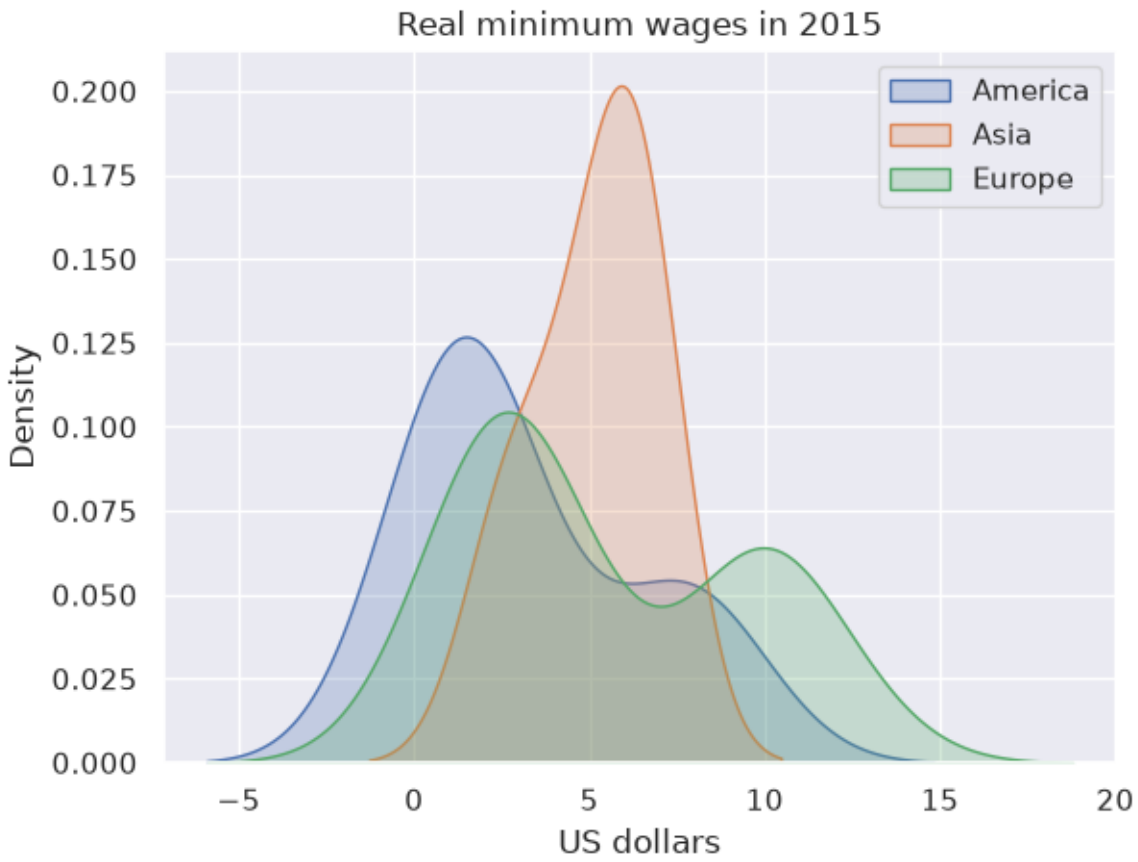
```
grouped.size()
```

```
Continent
America     7
Asia        4
Europe     19
dtype: int64
```

Calling `.get_group()` to return just the countries in a single group, we can create a kernel density estimate of the distribution of real minimum wages in 2016 for each continent.

`grouped.groups.keys()` will return the keys from the `groupby` object

```
continents = grouped.groups.keys()

for continent in continents:
    sns.kdeplot(grouped.get_group(continent).T.loc['2015'].unstack(), label=continent,
 ↪ fill=True)

plt.title('Real minimum wages in 2015')
plt.xlabel('US dollars')
plt.legend()
plt.show()
```



129.5 Final Remarks

This lecture has provided an introduction to some of pandas' more advanced features, including multiindices, merging, grouping and plotting.

Other tools that may be useful in panel data analysis include xarray, a python package that extends pandas to N-dimensional data structures.

129.6 Exercises

Exercise 129.6.1

In these exercises, you'll work with a dataset of employment rates in Europe by age and sex from Eurostat.

The dataset can be accessed with the following link:

```
url3 = 'https://github.com/QuantEcon/data-lectures/raw/main/lectures/employ.csv'
```

Reading in the CSV file returns a panel dataset in long format. Use `.pivot_table()` to construct a wide format dataframe with a `MultiIndex` in the columns.

Start off by exploring the dataframe and the variables available in the `MultiIndex` levels.

Write a program that quickly returns all values in the `MultiIndex`.

Solution

```
employ = pd.read_csv(url3)
employ = employ.pivot_table(values='Value',
                            index=['DATE'],
                            columns=['UNIT','AGE', 'SEX', 'INDIC_EM', 'GEO'])
employ.index = pd.to_datetime(employ.index) # ensure that dates are datetime format
employ.head()
```

```
UNIT       Percentage of total population                   ...  \
AGE                    From 15 to 24 years                  ...
SEX                                Females                  ...
INDIC_EM                 Active population                  ...
GEO                              Austria Belgium Bulgaria   ...
DATE                                                        ...
2007-01-01                         56.00   31.60    26.00   ...
2008-01-01                         56.20   30.80    26.10   ...
2009-01-01                         56.20   29.90    24.80   ...
2010-01-01                         54.00   29.80    26.60   ...
2011-01-01                         54.80   29.80    24.80   ...

UNIT                                          Thousand persons           \
AGE                                        From 55 to 64 years
SEX                                                      Total
INDIC_EM   Total employment (resident population concept - LFS)
GEO                                                Switzerland   Turkey
DATE
2007-01-01                                                 NaN   1,282.00
2008-01-01                                                 NaN   1,354.00
2009-01-01                                                 NaN   1,449.00
2010-01-01                                              640.00   1,583.00
2011-01-01                                              661.00   1,760.00

UNIT
AGE
SEX
INDIC_EM
GEO        United Kingdom
DATE
2007-01-01       4,131.00
2008-01-01       4,204.00
2009-01-01       4,193.00
2010-01-01       4,186.00
2011-01-01       4,164.00

[5 rows x 1440 columns]
```

This is a large dataset so it is useful to explore the levels and variables available

```
employ.columns.names
```

```
FrozenList(['UNIT', 'AGE', 'SEX', 'INDIC_EM', 'GEO'])
```

Variables within levels can be quickly retrieved with a loop

```
for name in employ.columns.names:
    print(name, employ.columns.get_level_values(name).unique())
```

```
UNIT Index(['Percentage of total population', 'Thousand persons'], dtype='str',␣
 ↪name='UNIT')
AGE Index(['From 15 to 24 years', 'From 25 to 54 years', 'From 55 to 64 years'],
 ↪ dtype='str', name='AGE')
SEX Index(['Females', 'Males', 'Total'], dtype='str', name='SEX')
INDIC_EM Index(['Active population', 'Total employment (resident population␣
 ↪concept - LFS)'], dtype='str', name='INDIC_EM')
GEO Index(['Austria', 'Belgium', 'Bulgaria', 'Croatia', 'Cyprus', 'Czech␣
 ↪Republic',
       'Denmark', 'Estonia', 'Euro area (17 countries)',
       'Euro area (18 countries)', 'Euro area (19 countries)',
       'European Union (15 countries)', 'European Union (27 countries)',
       'European Union (28 countries)', 'Finland',
       'Former Yugoslav Republic of Macedonia, the', 'France',
       'France (metropolitan)',
       'Germany (until 1990 former territory of the FRG)', 'Greece', 'Hungary',
       'Iceland', 'Ireland', 'Italy', 'Latvia', 'Lithuania', 'Luxembourg',
       'Malta', 'Netherlands', 'Norway', 'Poland', 'Portugal', 'Romania',
       'Slovakia', 'Slovenia', 'Spain', 'Sweden', 'Switzerland', 'Turkey',
       'United Kingdom'],
      dtype='str', name='GEO')
```

Exercise 129.6.2

Filter the above dataframe to only include employment as a percentage of 'active population'.

Create a grouped boxplot using `seaborn` of employment rates in 2015 by age group and sex.

Hint

`GEO` includes both areas and countries.

Solution

To easily filter by country, swap `GEO` to the top level and sort the `MultiIndex`

```
employ.columns = employ.columns.swaplevel(0,-1)
employ = employ.sort_index(axis=1)
```

We need to get rid of a few items in `GEO` which are not countries.

A fast way to get rid of the EU areas is to use a list comprehension to find the level values in `GEO` that begin with 'Euro'

```
geo_list = employ.columns.get_level_values('GEO').unique().tolist()
countries = [x for x in geo_list if not x.startswith('Euro')]
employ = employ[countries]
employ.columns.get_level_values('GEO').unique()
```

```
Index(['Austria', 'Belgium', 'Bulgaria', 'Croatia', 'Cyprus', 'Czech Republic',
       'Denmark', 'Estonia', 'Finland',
       'Former Yugoslav Republic of Macedonia, the', 'France',
       'France (metropolitan)',
       'Germany (until 1990 former territory of the FRG)', 'Greece', 'Hungary',
       'Iceland', 'Ireland', 'Italy', 'Latvia', 'Lithuania', 'Luxembourg',
       'Malta', 'Netherlands', 'Norway', 'Poland', 'Portugal', 'Romania',
       'Slovakia', 'Slovenia', 'Spain', 'Sweden', 'Switzerland', 'Turkey',
       'United Kingdom'],
      dtype='str', name='GEO')
```

Select only percentage employed in the active population from the dataframe

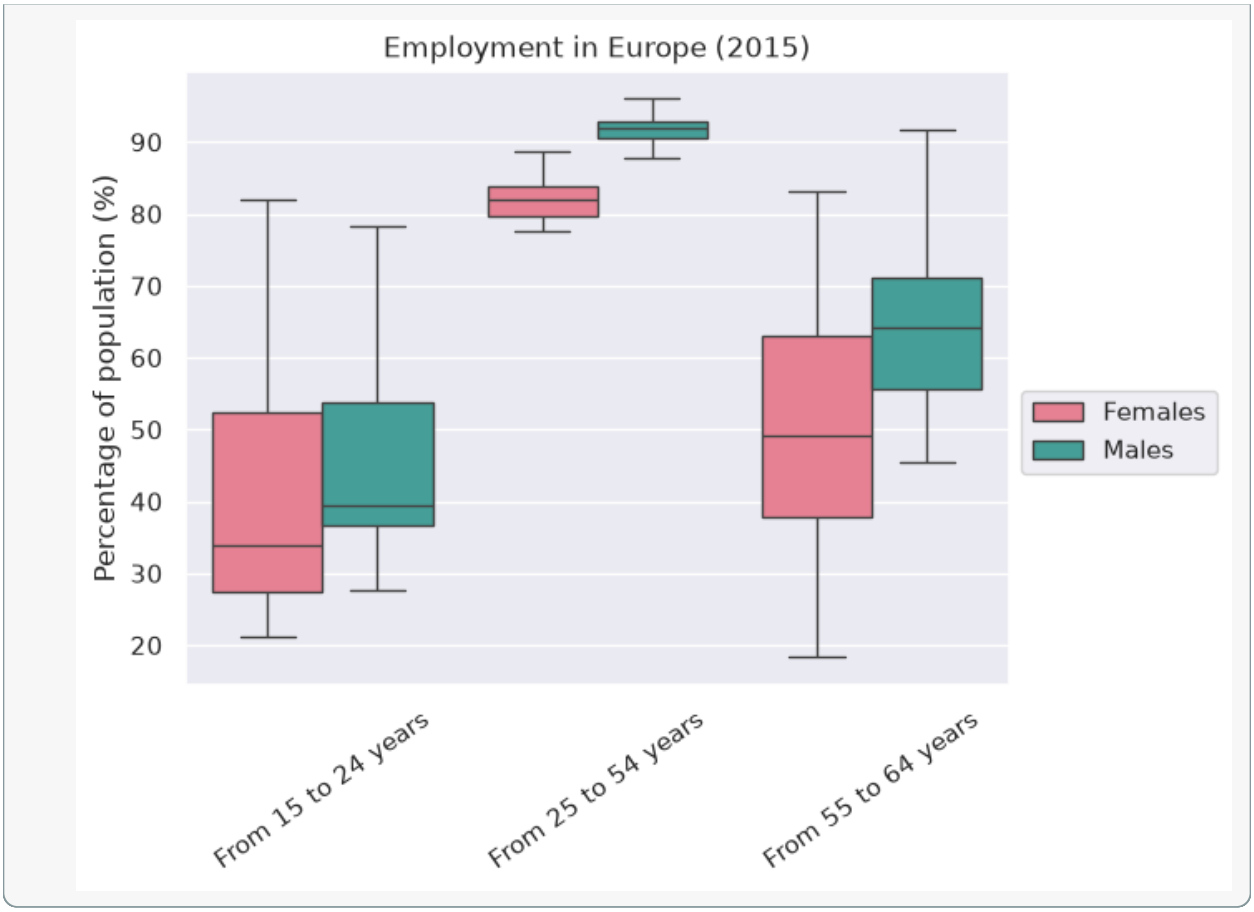
```
employ_f = employ.xs(('Percentage of total population', 'Active population'),
                     level=('UNIT', 'INDIC_EM'),
                     axis=1)
employ_f.head()
```

```
GEO                    Austria                ...      United Kingdom        \
AGE         From 15 to 24 years                ... From 55 to 64 years
SEX                     Females Males Total   ...              Females Males
DATE                                          ...
2007-01-01                56.00 62.90 59.40   ...                49.90 68.90
2008-01-01                56.20 62.90 59.50   ...                50.20 69.80
2009-01-01                56.20 62.90 59.50   ...                50.60 70.30
2010-01-01                54.00 62.60 58.30   ...                51.10 69.20
2011-01-01                54.80 63.60 59.20   ...                51.30 68.40

GEO
AGE
SEX         Total
DATE
2007-01-01 59.30
2008-01-01 59.80
2009-01-01 60.30
2010-01-01 60.00
2011-01-01 59.70

[5 rows x 306 columns]
```

Drop the 'Total' value before creating the grouped boxplot

```
employ_f = employ_f.drop('Total', level='SEX', axis=1)
```

```
box = employ_f.loc['2015'].unstack().reset_index()
sns.boxplot(x="AGE", y=0, hue="SEX", data=box, palette=("husl"), showfliers=False)
plt.xlabel('')
plt.xticks(rotation=35)
plt.ylabel('Percentage of population (%)')
plt.title('Employment in Europe (2015)')
plt.legend(bbox_to_anchor=(1,0.5))
plt.show()
```


Employment in Europe (2015)
Percentage of population (%)
90
80
70
60
50
40
30
20
From 15 to 24 years
From 25 to 54 years
From 55 to 64 years
Females
Males

LINEAR REGRESSION IN PYTHON

Contents

- *Linear Regression in Python*
  - *Overview*
  - *Simple Linear Regression*
  - *Extending the Linear Regression Model*
  - *Endogeneity*
  - *Summary*
  - *Exercises*

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install linearmodels
```

130.1 Overview

Linear regression is a standard tool for analyzing the relationship between two or more variables.

In this lecture, we'll use the Python package `statsmodels` to estimate, interpret, and visualize linear regression models.

Along the way, we'll discuss a variety of topics, including

- simple and multivariate linear regression
- visualization
- endogeneity and omitted variable bias
- two-stage least squares

As an example, we will replicate results from Acemoglu, Johnson and Robinson's seminal paper [Acemoglu *et al.*, 2001].

- You can download a copy here.

In the paper, the authors emphasize the importance of institutions in economic development.

The main contribution is the use of settler mortality rates as a source of *exogenous* variation in institutional differences.

Such variation is needed to determine whether it is institutions that give rise to greater economic growth, rather than the other way around.

Let's start with some imports:

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import statsmodels.api as sm
from statsmodels.iolib.summary2 import summary_col
from linearmodels.iv import IV2SLS
import seaborn as sns
sns.set_theme()
```

130.1.1 Prerequisites

This lecture assumes you are familiar with basic econometrics.

For an introductory text covering these topics, see, for example, [Wooldridge, 2015].

130.2 Simple Linear Regression

[Acemoglu *et al.*, 2001] wish to determine whether or not differences in institutions can help to explain observed economic outcomes.

How do we measure *institutional differences* and *economic outcomes*?

In this paper,

- economic outcomes are proxied by log GDP per capita in 1995, adjusted for exchange rates.
- institutional differences are proxied by an index of protection against expropriation on average over 1985-95, constructed by the Political Risk Services Group.

These variables and other data used in the paper are available for download on Daron Acemoglu's webpage.

We will use pandas' `.read_stata()` function to read in data contained in the `.dta` files to dataframes

```
df1 = pd.read_stata('https://github.com/QuantEcon/lecture-python.myst/raw/refs/heads/
↪main/lectures/_static/lecture_specific/ols/maketable1.dta')
df1.head()
```

```
  shortnam    euro1900  excolony    avexpr  logpgp95  cons1  cons90  democ00a  \
0      AFG    0.000000       1.0       NaN       NaN    1.0     2.0       1.0
1      AGO    8.000000       1.0  5.363636  7.770645    3.0     3.0       0.0
2      ARE    0.000000       1.0  7.181818  9.804219    NaN     NaN       NaN
3      ARG   60.000004       1.0  6.386364  9.133459    1.0     6.0       3.0
4      ARM    0.000000       0.0       NaN  7.682482    NaN     NaN       NaN

   cons00a    extmort4    logem4  loghjypl  baseco
0      1.0   93.699997  4.540098       NaN     NaN
1      1.0  280.000000  5.634789 -3.411248     1.0
2      NaN         NaN       NaN       NaN     NaN
3      3.0   68.900002  4.232656 -0.872274     1.0
4      NaN         NaN       NaN       NaN     NaN
```

Let's use a scatterplot to see whether any obvious relationship exists between GDP per capita and the protection against expropriation index

```
df1.plot(x='avexpr', y='logpgp95', kind='scatter')
plt.show()
```

The plot shows a fairly strong positive relationship between protection against expropriation and log GDP per capita.

Specifically, if higher protection against expropriation is a measure of institutional quality, then better institutions appear to be positively correlated with better economic outcomes (higher GDP per capita).

Given the plot, choosing a linear model to describe this relationship seems like a reasonable assumption.

We can write our model as

$$logpgp95_i = \beta_0 + \beta_1 avexpr_i + u_i$$

where:

- β_0 is the intercept of the linear trend line on the y-axis
- β_1 is the slope of the linear trend line, representing the *marginal effect* of protection against risk on log GDP per capita
- u_i is a random error term (deviations of observations from the linear trend due to factors not included in the model)

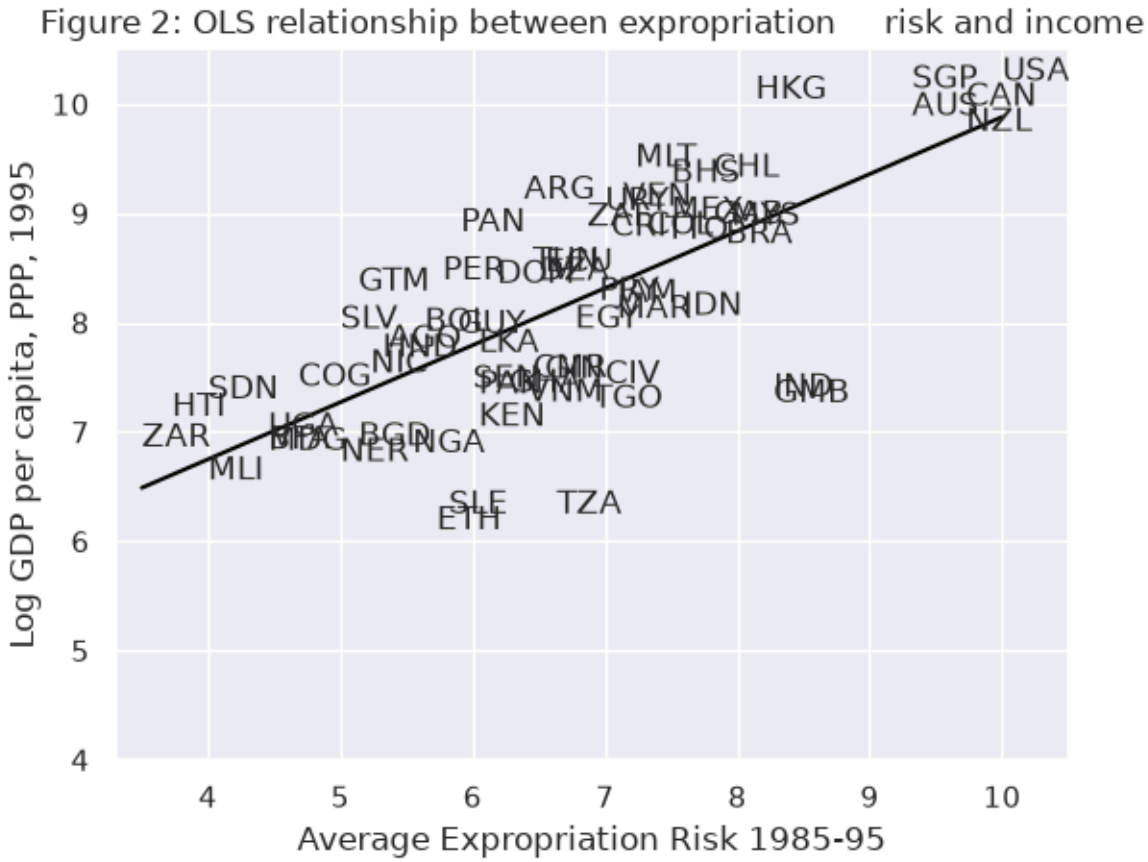
Visually, this linear model involves choosing a straight line that best fits the data, as in the following plot (Figure 2 in [Acemoglu *et al.*, 2001])

```
# Dropping NA's is required to use numpy's polyfit
df1_subset = df1.dropna(subset=['logpgp95', 'avexpr'])

# Use only 'base sample' for plotting purposes
df1_subset = df1_subset[df1_subset['baseco'] == 1]

X = df1_subset['avexpr']
y = df1_subset['logpgp95']
```

(continues on next page)

(continued from previous page)

```
labels = df1_subset['shortnam']

# Replace markers with country labels
fig, ax = plt.subplots()
ax.scatter(X, y, marker='')

for i, label in enumerate(labels):
    ax.annotate(label, (X.iloc[i], y.iloc[i]))

# Fit a linear trend line
ax.plot(np.unique(X),
         np.poly1d(np.polyfit(X, y, 1))(np.unique(X)),
         color='black')

ax.set_xlim([3.3,10.5])
ax.set_ylim([4,10.5])
ax.set_xlabel('Average Expropriation Risk 1985-95')
ax.set_ylabel('Log GDP per capita, PPP, 1995')
ax.set_title('Figure 2: OLS relationship between expropriation \
    risk and income')
plt.show()
```



The most common technique to estimate the parameters (β's) of the linear model is Ordinary Least Squares (OLS).

As the name implies, an OLS model is solved by finding the parameters that minimize **the sum of squared residuals**,

i.e.

$$\min_{\hat{\beta}} \sum_{i=1}^{N} \hat{u}_i^2$$

where $\hat{u}_i$ is the difference between the observation and the predicted value of the dependent variable.

To estimate the constant term β_0, we need to add a column of 1's to our dataset (consider the equation if β_0 was replaced with $\beta_0 x_i$ and $x_i = 1$)

```
df1['const'] = 1
```

Now we can construct our model in `statsmodels` using the OLS function.

We will use `pandas` dataframes with `statsmodels`, however standard arrays can also be used as arguments

```
reg1 = sm.OLS(endog=df1['logpgp95'], exog=df1[['const', 'avexpr']], \
    missing='drop')
type(reg1)
```

```
statsmodels.regression.linear_model.OLS
```

So far we have simply constructed our model.

We need to use `.fit()` to obtain parameter estimates $\hat{\beta}_0$ and $\hat{\beta}_1$

```
results = reg1.fit()
type(results)
```

```
statsmodels.regression.linear_model.RegressionResultsWrapper
```

We now have the fitted regression model stored in `results`.

To view the OLS regression results, we can call the `.summary()` method.

Note that an observation was mistakenly dropped from the results in the original paper (see the note located in `maketable2.do` from Acemoglu's webpage), and thus the coefficients differ slightly.

```
print(results.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                logpgp95   R-squared:                       0.611
Model:                            OLS   Adj. R-squared:                  0.608
Method:                 Least Squares   F-statistic:                     171.4
Date:                Mon, 20 Jul 2026   Prob (F-statistic):           4.16e-24
Time:                        05:10:48   Log-Likelihood:                -119.71
No. Observations:                 111   AIC:                             243.4
Df Residuals:                     109   BIC:                             248.8
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const          4.6261      0.301     15.391      0.000       4.030       5.222
avexpr         0.5319      0.041     13.093      0.000       0.451       0.612
==============================================================================
Omnibus:                        9.251   Durbin-Watson:                   1.689
```

(continues on next page)

(continued from previous page)

```
Prob(Omnibus):                  0.010   Jarque-Bera (JB):                 9.170
Skew:                          -0.680   Prob(JB):                        0.0102
Kurtosis:                       3.362   Cond. No.                          33.2
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly␣
↪specified.
```

From our results, we see that

- The intercept $\hat{\beta}_0 = 4.63$.
- The slope $\hat{\beta}_1 = 0.53$.
- The positive $\hat{\beta}_1$ parameter estimate implies that. institutional quality has a positive effect on economic outcomes, as we saw in the figure.
- The p-value of 0.000 for $\hat{\beta}_1$ implies that the effect of institutions on GDP is statistically significant (using p < 0.05 as a rejection rule).
- The R-squared value of 0.611 indicates that around 61% of variation in log GDP per capita is explained by protection against expropriation.

Using our parameter estimates, we can now write our estimated relationship as

$$\widehat{logpgp95}_i = 4.63 + 0.53 \; avexpr_i$$

This equation describes the line that best fits our data, as shown in Figure 2.

We can use this equation to predict the level of log GDP per capita for a value of the index of expropriation protection.

For example, for a country with an index value of 7.07 (the average for the dataset), we find that their predicted level of log GDP per capita in 1995 is 8.38.

```
mean_expr = np.mean(df1_subset['avexpr'])
mean_expr
```

```
np.float32(6.515625)
```

```
predicted_logpdp95 = 4.63 + 0.53 * 7.07
predicted_logpdp95
```

```
8.3771
```

An easier (and more accurate) way to obtain this result is to use `.predict()` and set $constant = 1$ and $avexpr_i = mean_expr$

```
results.predict(exog=[1, mean_expr])
```

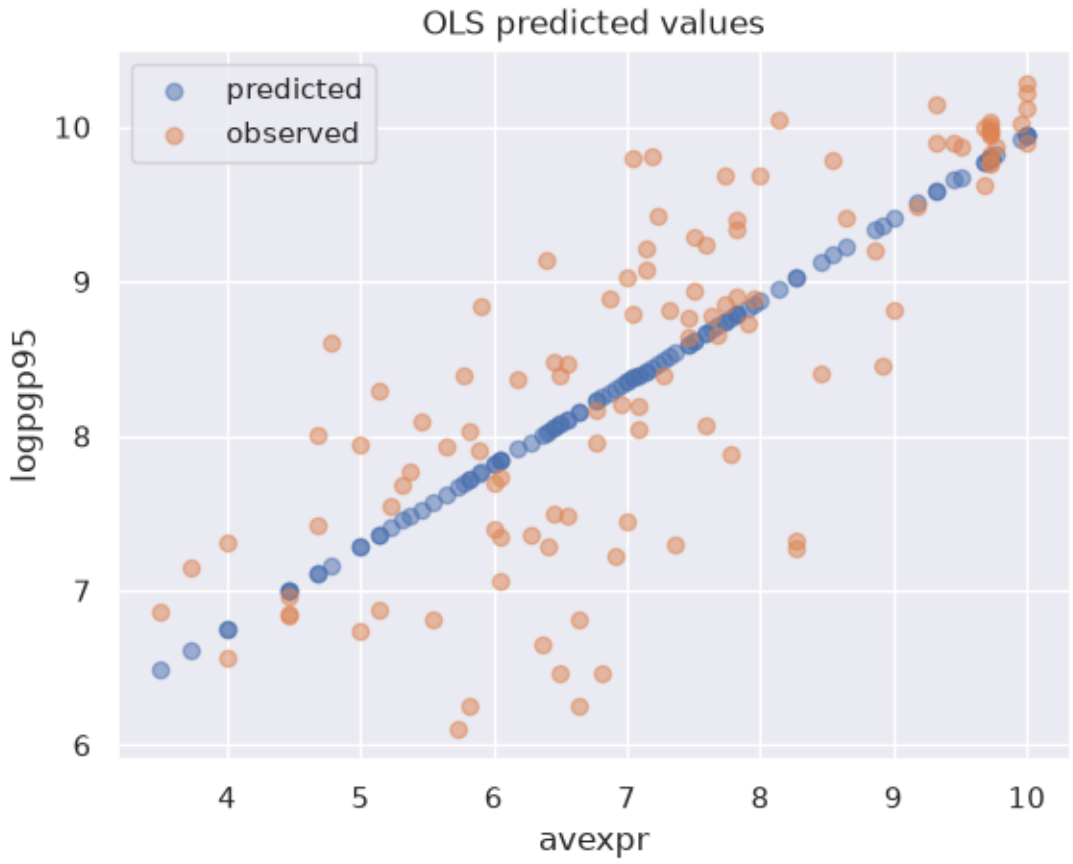
```
array([8.09156367])
```

We can obtain an array of predicted $logpgp95_i$ for every value of $avexpr_i$ in our dataset by calling `.predict()` on our results.

Plotting the predicted values against $avexpr_i$ shows that the predicted values lie along the linear line that we fitted above.

The observed values of $logpgp95_i$ are also plotted for comparison purposes

```
# Drop missing observations from whole sample

df1_plot = df1.dropna(subset=['logpgp95', 'avexpr'])

# Plot predicted values

fix, ax = plt.subplots()
ax.scatter(df1_plot['avexpr'], results.predict(), alpha=0.5,
        label='predicted')

# Plot observed values

ax.scatter(df1_plot['avexpr'], df1_plot['logpgp95'], alpha=0.5,
        label='observed')

ax.legend()
ax.set_title('OLS predicted values')
ax.set_xlabel('avexpr')
ax.set_ylabel('logpgp95')
plt.show()
```



130.3 Extending the Linear Regression Model

So far we have only accounted for institutions affecting economic performance - almost certainly there are numerous other factors affecting GDP that are not included in our model.

Leaving out variables that affect $logpgp95_i$ will result in **omitted variable bias**, yielding biased and inconsistent parameter estimates.

We can extend our bivariate regression model to a **multivariate regression model** by adding in other factors that may affect $logpgp95_i$.

[Acemoglu *et al.*, 2001] consider other factors such as:

- the effect of climate on economic outcomes; latitude is used to proxy this
- differences that affect both economic performance and institutions, eg. cultural, historical, etc.; controlled for with the use of continent dummies

Let's estimate some of the extended models considered in the paper (Table 2) using data from `maketable2.dta`

```
df2 = pd.read_stata('https://github.com/QuantEcon/lecture-python.myst/raw/refs/heads/
↪main/lectures/_static/lecture_specific/ols/maketable2.dta')

# Add constant term to dataset
df2['const'] = 1

# Create lists of variables to be used in each regression
X1 = ['const', 'avexpr']
X2 = ['const', 'avexpr', 'lat_abst']
X3 = ['const', 'avexpr', 'lat_abst', 'asia', 'africa', 'other']

# Estimate an OLS regression for each set of variables
reg1 = sm.OLS(df2['logpgp95'], df2[X1], missing='drop').fit()
reg2 = sm.OLS(df2['logpgp95'], df2[X2], missing='drop').fit()
reg3 = sm.OLS(df2['logpgp95'], df2[X3], missing='drop').fit()
```

Now that we have fitted our model, we will use `summary_col` to display the results in a single table (model numbers correspond to those in the paper)

```
info_dict={'R-squared' : lambda x: f"{x.rsquared:.2f}",
           'No. observations' : lambda x: f"{int(x.nobs):d}"}

results_table = summary_col(results=[reg1,reg2,reg3],
                            float_format='%0.2f',
                            stars = True,
                            model_names=['Model 1',
                                         'Model 3',
                                         'Model 4'],
                            info_dict=info_dict,
                            regressor_order=['const',
                                             'avexpr',
                                             'lat_abst',
                                             'asia',
                                             'africa'])

results_table.add_title('Table 2 - OLS Regressions')

print(results_table)
```

```
        Table 2 - OLS Regressions
=========================================
                 Model 1 Model 3 Model 4
-----------------------------------------
const            4.63*** 4.87*** 5.85***
                 (0.30)  (0.33)  (0.34)
avexpr           0.53*** 0.46*** 0.39***
                 (0.04)  (0.06)  (0.05)
lat_abst                 0.87*   0.33
                         (0.49)  (0.45)
asia                             -0.15
                                 (0.15)
africa                           -0.92***
                                 (0.17)
other                            0.30
                                 (0.37)
R-squared        0.61    0.62    0.72
R-squared Adj.   0.61    0.62    0.70
No. observations 111     111     111
R-squared        0.61    0.62    0.72
=========================================
Standard errors in parentheses.
* p<.1, ** p<.05, ***p<.01
```

130.4 Endogeneity

As [Acemoglu *et al.*, 2001] discuss, the OLS models likely suffer from **endogeneity** issues, resulting in biased and inconsistent model estimates.

Namely, there is likely a two-way relationship between institutions and economic outcomes:

- richer countries may be able to afford or prefer better institutions
- variables that affect income may also be correlated with institutional differences
- the construction of the index may be biased; analysts may be biased towards seeing countries with higher income having better institutions

To deal with endogeneity, we can use **two-stage least squares (2SLS) regression**, which is an extension of OLS regression.

This method requires replacing the endogenous variable $avexpr_i$ with a variable that is:

1. correlated with $avexpr_i$
2. not correlated with the error term (ie. it should not directly affect the dependent variable, otherwise it would be correlated with u_i due to omitted variable bias)

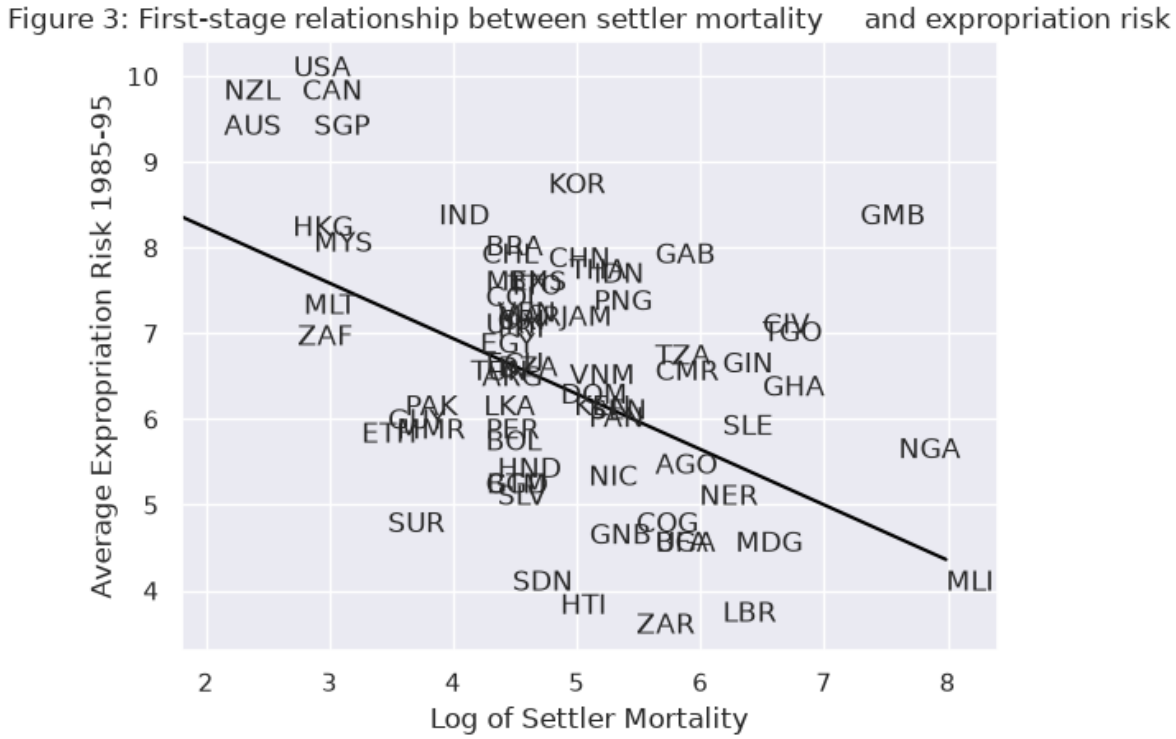
The new set of regressors is called an **instrument**, which aims to remove endogeneity in our proxy of institutional differences.

The main contribution of [Acemoglu *et al.*, 2001] is the use of settler mortality rates to instrument for institutional differences.

They hypothesize that higher mortality rates of colonizers led to the establishment of institutions that were more extractive in nature (less protection against expropriation), and these institutions still persist today.

Using a scatterplot (Figure 3 in [Acemoglu *et al.*, 2001]), we can see protection against expropriation is negatively correlated with settler mortality rates, coinciding with the authors' hypothesis and satisfying the first condition of a valid instrument.

```
# Dropping NA's is required to use numpy's polyfit
df1_subset2 = df1.dropna(subset=['logem4', 'avexpr'])

X = df1_subset2['logem4']
y = df1_subset2['avexpr']
labels = df1_subset2['shortnam']

# Replace markers with country labels
fig, ax = plt.subplots()
ax.scatter(X, y, marker='')

for i, label in enumerate(labels):
    ax.annotate(label, (X.iloc[i], y.iloc[i]))

# Fit a linear trend line
ax.plot(np.unique(X),
         np.poly1d(np.polyfit(X, y, 1))(np.unique(X)),
         color='black')

ax.set_xlim([1.8,8.4])
ax.set_ylim([3.3,10.4])
ax.set_xlabel('Log of Settler Mortality')
ax.set_ylabel('Average Expropriation Risk 1985-95')
ax.set_title('Figure 3: First-stage relationship between settler mortality \
    and expropriation risk')
plt.show()
```



The second condition may not be satisfied if settler mortality rates in the 17th to 19th centuries have a direct effect on

current GDP (in addition to their indirect effect through institutions).

For example, settler mortality rates may be related to the current disease environment in a country, which could affect current economic performance.

[Acemoglu *et al.*, 2001] argue this is unlikely because:

- The majority of settler deaths were due to malaria and yellow fever and had a limited effect on local people.
- The disease burden on local people in Africa or India, for example, did not appear to be higher than average, supported by relatively high population densities in these areas before colonization.

As we appear to have a valid instrument, we can use 2SLS regression to obtain consistent and unbiased parameter estimates.

First stage

The first stage involves regressing the endogenous variable ($avexpr_i$) on the instrument.

The instrument is the set of all exogenous variables in our model (and not just the variable we have replaced).

Using model 1 as an example, our instrument is simply a constant and settler mortality rates $logem4_i$.

Therefore, we will estimate the first-stage regression as

$$avexpr_i = \delta_0 + \delta_1 logem4_i + v_i$$

The data we need to estimate this equation is located in `maketable4.dta` (only complete data, indicated by `baseco = 1`, is used for estimation)

```
# Import and select the data
df4 = pd.read_stata('https://github.com/QuantEcon/lecture-python.myst/raw/refs/heads/
↪main/lectures/_static/lecture_specific/ols/maketable4.dta')
df4 = df4[df4['baseco'] == 1]

# Add a constant variable
df4['const'] = 1

# Fit the first stage regression and print summary
results_fs = sm.OLS(df4['avexpr'],
                    df4[['const', 'logem4']],
                    missing='drop').fit()
print(results_fs.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                  avexpr   R-squared:                       0.270
Model:                            OLS   Adj. R-squared:                  0.258
Method:                 Least Squares   F-statistic:                     22.95
Date:                Mon, 20 Jul 2026   Prob (F-statistic):           1.08e-05
Time:                        05:10:49   Log-Likelihood:                -104.83
No. Observations:                  64   AIC:                             213.7
Df Residuals:                      62   BIC:                             218.0
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const          9.3414      0.611     15.296      0.000       8.121      10.562
logem4        -0.6068      0.127     -4.790      0.000      -0.860      -0.354
==============================================================================
```

(continues on next page)

(continued from previous page)

```
Omnibus:                        0.035   Durbin-Watson:                   2.003
Prob(Omnibus):                  0.983   Jarque-Bera (JB):                0.172
Skew:                           0.045   Prob(JB):                        0.918
Kurtosis:                       2.763   Cond. No.                         19.4
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly␣
↪specified.
```

Second stage

We need to retrieve the predicted values of $avexpr_i$ using `.predict()`.

We then replace the endogenous variable $avexpr_i$ with the predicted values $\widehat{avexpr}_i$ in the original linear model.

Our second stage regression is thus

$$logpgp95_i = \beta_0 + \beta_1 \widehat{avexpr}_i + u_i$$

```
df4['predicted_avexpr'] = results_fs.predict()

results_ss = sm.OLS(df4['logpgp95'],
                    df4[['const', 'predicted_avexpr']]).fit()
print(results_ss.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                logpgp95   R-squared:                       0.477
Model:                            OLS   Adj. R-squared:                  0.469
Method:                 Least Squares   F-statistic:                     56.60
Date:                Mon, 20 Jul 2026   Prob (F-statistic):           2.66e-10
Time:                        05:10:49   Log-Likelihood:                -72.268
No. Observations:                  64   AIC:                             148.5
Df Residuals:                      62   BIC:                             152.9
Df Model:                           1
Covariance Type:            nonrobust
================================================================================
                     coef    std err          t      P>|t|      [0.025      0.
↪975]
--------------------------------------------------------------------------------
↪-
const                1.9097      0.823      2.320      0.024       0.264       3.
↪555
predicted_avexpr     0.9443      0.126      7.523      0.000       0.693       1.
↪195
==============================================================================
Omnibus:                       10.547   Durbin-Watson:                   2.137
Prob(Omnibus):                  0.005   Jarque-Bera (JB):               11.010
Skew:                          -0.790   Prob(JB):                      0.00407
Kurtosis:                       4.277   Cond. No.                         58.1
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly␣
↪specified.
```

The second-stage regression results give us an unbiased and consistent estimate of the effect of institutions on economic outcomes.

The result suggests a stronger positive relationship than what the OLS results indicated.

Note that while our parameter estimates are correct, our standard errors are not and for this reason, computing 2SLS 'manually' (in stages with OLS) is not recommended.

We can correctly estimate a 2SLS regression in one step using the linearmodels package, an extension of `statsmodels`

Note that when using `IV2SLS`, the exogenous and instrument variables are split up in the function arguments (whereas before the instrument included exogenous variables)

```
iv = IV2SLS(dependent=df4['logpgp95'],
            exog=df4['const'],
            endog=df4['avexpr'],
            instruments=df4['logem4']).fit(cov_type='unadjusted')

print(iv.summary)
```

```
                          IV-2SLS Estimation Summary
==============================================================================
Dep. Variable:               logpgp95   R-squared:                      0.1870
Estimator:                    IV-2SLS   Adj. R-squared:                 0.1739
No. Observations:                  64   F-statistic:                    37.568
Date:                Mon, Jul 20 2026   P-value (F-stat)                0.0000
Time:                        05:10:49   Distribution:                  chi2(1)
Cov. Estimator:            unadjusted

                             Parameter Estimates
==============================================================================
            Parameter  Std. Err.     T-stat    P-value    Lower CI    Upper CI
------------------------------------------------------------------------------
const          1.9097     1.0106     1.8897     0.0588     -0.0710      3.8903
avexpr         0.9443     0.1541     6.1293     0.0000      0.6423      1.2462
==============================================================================

Endogenous: avexpr
Instruments: logem4
Unadjusted Covariance (Homoskedastic)
Debiased: False
```

Given that we now have consistent and unbiased estimates, we can infer from the model we have estimated that institutional differences (stemming from institutions set up during colonization) can help to explain differences in income levels across countries today.

[Acemoglu *et al.*, 2001] use a marginal effect of 0.94 to calculate that the difference in the index between Chile and Nigeria (ie. institutional quality) implies up to a 7-fold difference in income, emphasizing the significance of institutions in economic development.

130.5 Summary

We have demonstrated basic OLS and 2SLS regression in `statsmodels` and `linearmodels`.

If you are familiar with R, you may want to use the formula interface to `statsmodels`, or consider using r2py to call R from within Python.

130.6 Exercises

Exercise 130.6.1

In the lecture, we think the original model suffers from endogeneity bias due to the likely effect income has on institutional development.

Although endogeneity is often best identified by thinking about the data and model, we can formally test for endogeneity using the **Hausman test**.

We want to test for correlation between the endogenous variable, $avexpr_i$, and the errors, u_i

$$H_0 : Cov(avexpr_i, u_i) = 0 \quad (no\ endogeneity)$$
$$H_1 : Cov(avexpr_i, u_i) \neq 0 \quad (endogeneity)$$

This test is running in two stages.

First, we regress $avexpr_i$ on the instrument, $logem4_i$

$$avexpr_i = \pi_0 + \pi_1 logem4_i + \upsilon_i$$

Second, we retrieve the residuals $\hat{\upsilon}_i$ and include them in the original equation

$$logpgp95_i = \beta_0 + \beta_1 avexpr_i + \alpha\hat{\upsilon}_i + u_i$$

If α is statistically significant (with a p-value < 0.05), then we reject the null hypothesis and conclude that $avexpr_i$ is endogenous.

Using the above information, estimate a Hausman test and interpret your results.

Solution

```
# Load in data
df4 = pd.read_stata('https://github.com/QuantEcon/lecture-python.myst/raw/refs/
↪heads/main/lectures/_static/lecture_specific/ols/maketable4.dta')

# Add a constant term
df4['const'] = 1

# Estimate the first stage regression
reg1 = sm.OLS(endog=df4['avexpr'],
              exog=df4[['const', 'logem4']],
              missing='drop').fit()

# Retrieve the residuals
df4['resid'] = reg1.resid
```

```
# Estimate the second stage residuals
reg2 = sm.OLS(endog=df4['logpgp95'],
              exog=df4[['const', 'avexpr', 'resid']],
              missing='drop').fit()

print(reg2.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                logpgp95   R-squared:                       0.689
Model:                            OLS   Adj. R-squared:                  0.679
Method:                 Least Squares   F-statistic:                     74.05
Date:                Mon, 20 Jul 2026   Prob (F-statistic):           1.07e-17
Time:                        05:10:49   Log-Likelihood:                -62.031
No. Observations:                  70   AIC:                             130.1
Df Residuals:                      67   BIC:                             136.8
Df Model:                           2
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const          2.4782      0.547      4.530      0.000       1.386       3.570
avexpr         0.8564      0.082     10.406      0.000       0.692       1.021
resid         -0.4951      0.099     -5.017      0.000      -0.692      -0.298
==============================================================================
Omnibus:                       17.597   Durbin-Watson:                   2.086
Prob(Omnibus):                  0.000   Jarque-Bera (JB):               23.194
Skew:                          -1.054   Prob(JB):                     9.19e-06
Kurtosis:                       4.873   Cond. No.                         53.8
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is␣
↪correctly specified.
```

The output shows that the coefficient on the residuals is statistically significant, indicating $avexpr_i$ is endogenous.

Exercise 130.6.2

The OLS parameter β can also be estimated using matrix algebra and `numpy` (you may need to review the numpy lecture to complete this exercise).

The linear equation we want to estimate is (written in matrix form)

$$y = X\beta + u$$

To solve for the unknown parameter β, we want to minimize the sum of squared residuals

$$\min_{\hat{\beta}} \hat{u}'\hat{u}$$

Rearranging the first equation and substituting into the second equation, we can write

$$\min_{\hat{\beta}} (Y - X\hat{\beta})'(Y - X\hat{\beta})$$

Solving this optimization problem gives the solution for the $\hat{\beta}$ coefficients

$$\hat{\beta} = (X'X)^{-1}X'y$$

Using the above information, compute $\hat{\beta}$ from model 1 using `numpy` - your results should be the same as those in the `statsmodels` output from earlier in the lecture.

Solution

```
# Load in data
df1 = pd.read_stata('https://github.com/QuantEcon/lecture-python.myst/raw/refs/
↪heads/main/lectures/_static/lecture_specific/ols/maketable1.dta')
df1 = df1.dropna(subset=['logpgp95', 'avexpr'])

# Add a constant term
df1['const'] = 1

# Define the X and y variables
y = np.asarray(df1['logpgp95'])
X = np.asarray(df1[['const', 'avexpr']])

# Compute β_hat
β_hat = np.linalg.solve(X.T @ X, X.T @ y)

# Print out the results from the 2 x 1 vector β_hat
print(f'β_0 = {β_hat[0]:.2}')
print(f'β_1 = {β_hat[1]:.2}')
```

```
β_0 = 4.6
β_1 = 0.53
```

It is also possible to use `np.linalg.inv(X.T @ X) @ X.T @ y` to solve for β, however `.solve()` is preferred as it involves fewer computations.

CHAPTER

ONE

MAXIMUM LIKELIHOOD ESTIMATION

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

Contents

- *Maximum Likelihood Estimation*
  - *Overview*
  - *Set up and assumptions*
  - *Conditional distributions*
  - *Maximum likelihood estimation*
  - *MLE with numerical methods*
  - *Maximum likelihood estimation with* `statsmodels`
  - *Summary*
  - *Exercises*

131.1 Overview

In *Linear Regression in Python*, we estimated the relationship between dependent and explanatory variables using linear regression.

But what if a linear relationship is not an appropriate assumption for our model?

One widely used alternative is maximum likelihood estimation, which involves specifying a class of distributions, indexed by unknown parameters, and then using the data to pin down these parameter values.

The benefit relative to linear regression is that it allows more flexibility in the probabilistic relationships between variables.

Here we illustrate maximum likelihood by replicating Daniel Treisman's (2016) paper, Russia's Billionaires, which connects the number of billionaires in a country to its economic characteristics.

The paper concludes that Russia has a higher number of billionaires than economic factors such as market size and tax rate predict.

We'll require the following imports:

```
import numpy as np
import jax.numpy as jnp
import jax
import pandas as pd
from typing import NamedTuple

from jax.scipy.special import factorial, gammaln
from jax.scipy.stats import norm

from statsmodels.api import Poisson
from statsmodels.iolib.summary2 import summary_col

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

131.1.1 Prerequisites

We assume familiarity with basic probability and multivariate calculus.

131.2 Set up and assumptions

Let's consider the steps we need to go through in maximum likelihood estimation and how they pertain to this study.

131.2.1 Flow of ideas

The first step with maximum likelihood estimation is to choose the probability distribution believed to be generating the data.

More precisely, we need to make an assumption as to which *parametric class* of distributions is generating the data.

- e.g., the class of all normal distributions, or the class of all gamma distributions.

Each such class is a family of distributions indexed by a finite number of parameters.

- e.g., the class of normal distributions is a family of distributions indexed by its mean $\mu \in (-\infty, \infty)$ and standard deviation $\sigma \in (0, \infty)$.

We'll let the data pick out a particular element of the class by pinning down the parameters.

The parameter estimates so produced will be called **maximum likelihood estimates**.

131.2.2 Counting billionaires

Treisman [Treisman, 2016] is interested in estimating the number of billionaires in different countries.

The number of billionaires is integer-valued.

Hence we consider distributions that take values only in the nonnegative integers.

(This is one reason least squares regression is not the best tool for the present problem, since the dependent variable in linear regression is not restricted to integer values.)

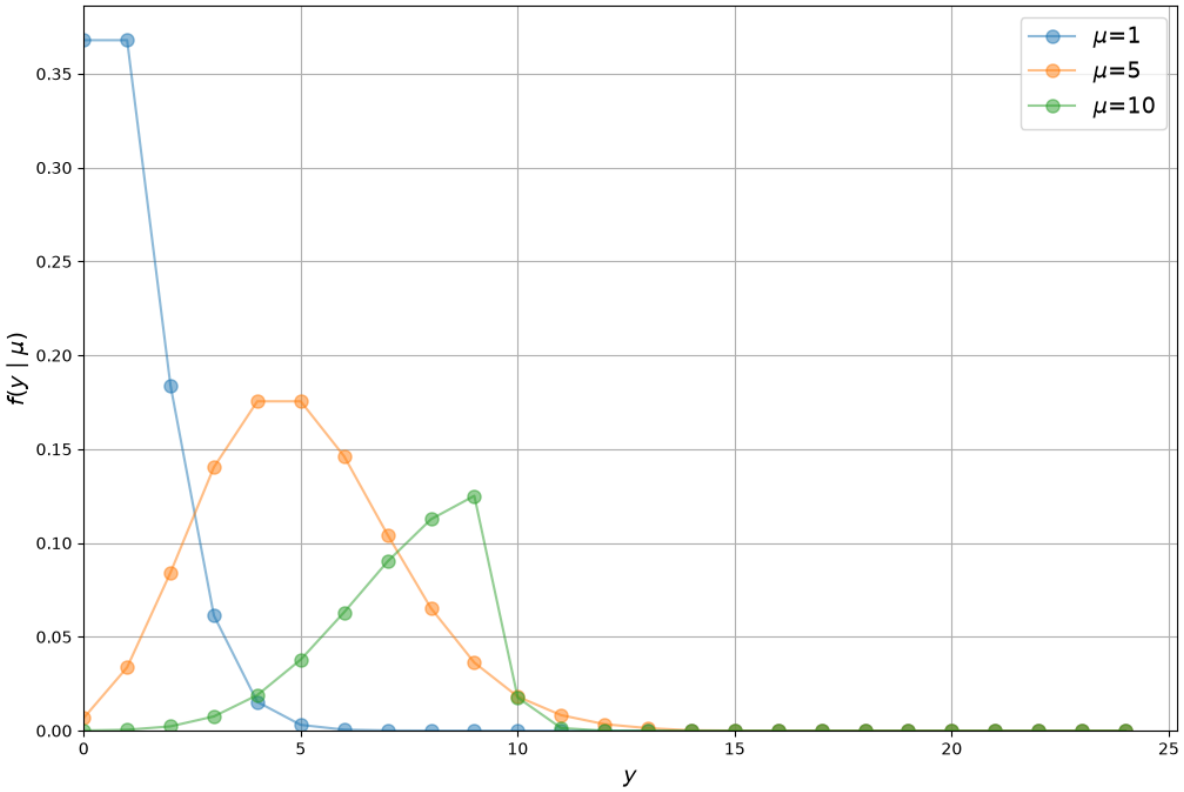
One integer distribution is the Poisson distribution, the probability mass function (pmf) of which is

$$f(y) = \frac{\mu^y}{y!} e^{-\mu}, \qquad y = 0, 1, 2, \ldots, \infty$$

We can plot the Poisson distribution over y for different values of μ as follows

```
@jax.jit
def poisson_pmf(y, μ):
    return μ**y / factorial(y) * jnp.exp(-μ)
```

```
y_values = range(0, 25)

fig, ax = plt.subplots(figsize=(12, 8))

for μ in [1, 5, 10]:
    distribution = []
    for y_i in y_values:
        distribution.append(poisson_pmf(y_i, μ))
    ax.plot(
        y_values,
        distribution,
        label=rf"$\mu$={μ}",
        alpha=0.5,
        marker="o",
        markersize=8,
    )

ax.grid()
ax.set_xlabel(r"$y$", fontsize=14)
ax.set_ylabel(r"$f(y \mid \mu)$", fontsize=14)
ax.axis(xmin=0, ymin=0)
ax.legend(fontsize=14)

plt.show()
```



Notice that the Poisson distribution begins to resemble a normal distribution as the mean of y increases.

Let's have a look at the distribution of the data we'll be working with in this lecture.

Treisman's main source of data is *Forbes*' annual rankings of billionaires and their estimated net worth.

The dataset `mle/fp.dta` can be downloaded from here or its AER page.

```
# Load in data and view
df = pd.read_stata(
    "https://github.com/QuantEcon/lecture-python.myst/raw/refs/heads/main/lectures/_
↪static/lecture_specific/mle/fp.dta"
)
df.head()
```

```
         country  ccode    year    cyear  numbil  numbil0  numbilall  netw  \
0  United States    2.0  1990.0  21990.0     NaN      NaN        NaN   NaN
1  United States    2.0  1991.0  21991.0     NaN      NaN        NaN   NaN
2  United States    2.0  1992.0  21992.0     NaN      NaN        NaN   NaN
3  United States    2.0  1993.0  21993.0     NaN      NaN        NaN   NaN
4  United States    2.0  1994.0  21994.0     NaN      NaN        NaN   NaN

   netw0  netwall  ...  gattwto08     mcapbdol    mcapbdol08  lnmcap08  \
0    NaN      NaN  ...        0.0  3060.000000  11737.599609  9.370638
1    NaN      NaN  ...        0.0  4090.000000  11737.599609  9.370638
2    NaN      NaN  ...        0.0  4490.000000  11737.599609  9.370638
3    NaN      NaN  ...        0.0  5136.198730  11737.599609  9.370638
4    NaN      NaN  ...        0.0  5067.016113  11737.599609  9.370638

   topintaxnew   topint08     rintr  noyrs  roflaw  nrrents
0    39.799999  39.799999  4.988405   20.0    1.61      NaN
```

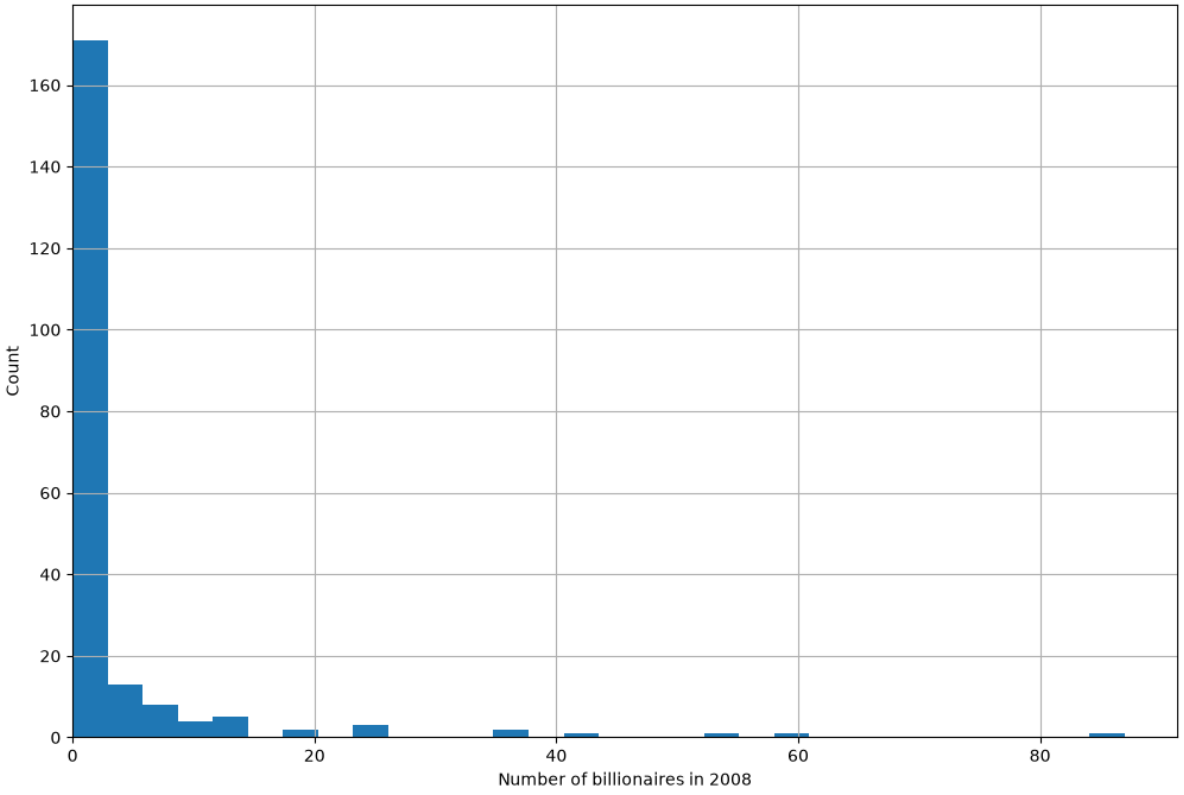
(continues on next page)

(continued from previous page)

```
1    39.799999  39.799999  4.988405   20.0    1.61      NaN
2    39.799999  39.799999  4.988405   20.0    1.61      NaN
3    39.799999  39.799999  4.988405   20.0    1.61      NaN
4    39.799999  39.799999  4.988405   20.0    1.61      NaN

[5 rows x 36 columns]
```

Using a histogram, we can view the distribution of the number of billionaires per country, `numbil0`, in 2008 (the United States is dropped for plotting purposes)

```
numbil0_2008 = df[
    (df["year"] == 2008) & (df["country"] != "United States")
].loc[:, "numbil0"]

plt.subplots(figsize=(12, 8))
plt.hist(numbil0_2008, bins=30)
plt.xlim(left=0)
plt.grid()
plt.xlabel("Number of billionaires in 2008")
plt.ylabel("Count")
plt.show()
```



From the histogram, it appears that the Poisson assumption is not unreasonable (albeit with a very low μ and some outliers).

131.3 Conditional distributions

In Treisman's paper, the dependent variable — the number of billionaires y_i in country i — is modeled as a function of GDP per capita, population size, and years membership in GATT and WTO.

Hence, the distribution of y_i needs to be conditioned on the vector of explanatory variables $\mathbf{x}_i$.

The standard formulation — the so-called **Poisson regression** model — is as follows:

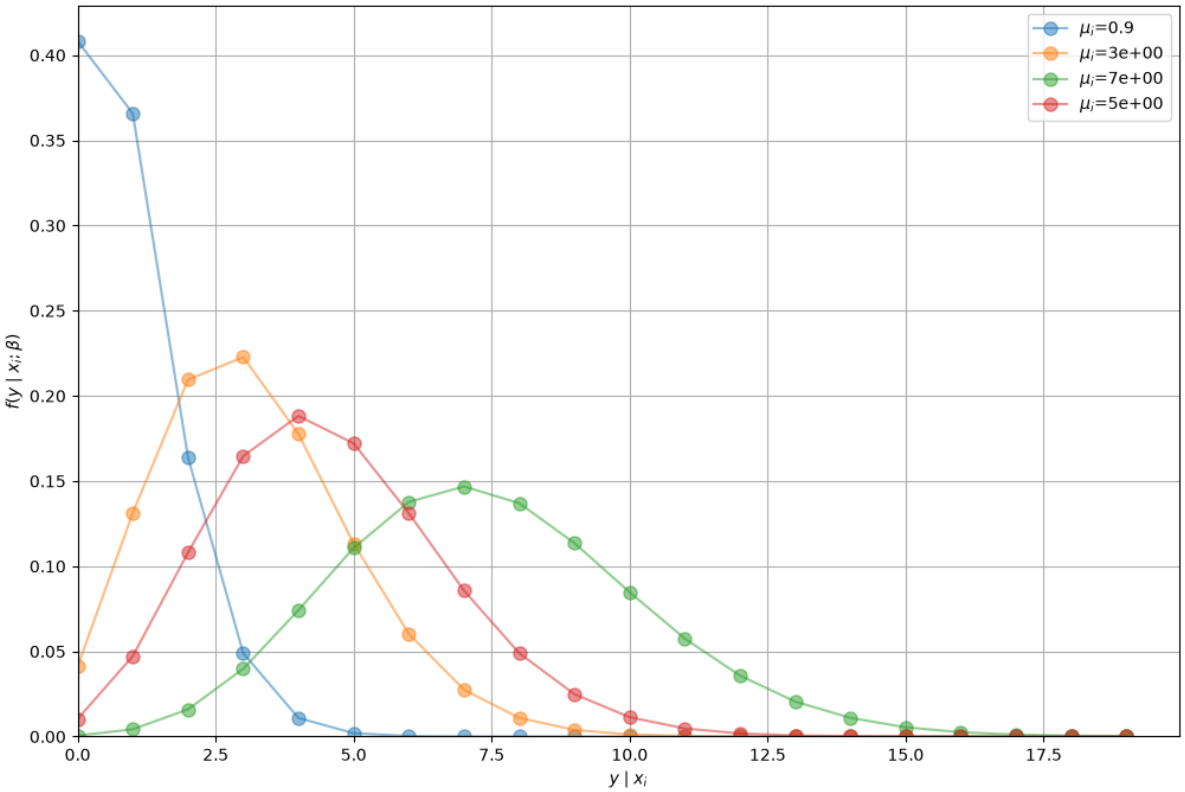
$$f(y_i \mid \mathbf{x}_i) = \frac{\mu_i^{y_i}}{y_i!} e^{-\mu_i}; \qquad y_i = 0, 1, 2, \ldots, \infty. \tag{131.1}$$

$$\text{where } \mu_i = \exp(\mathbf{x}_i'\beta) = \exp(\beta_0 + \beta_1 x_{i1} + \ldots + \beta_k x_{ik})$$

To illustrate the idea that the distribution of y_i depends on $\mathbf{x}_i$ let's run a simple simulation.

We use our `poisson_pmf` function from above and arbitrary values for β and $\mathbf{x}_i$

```
y_values = range(0, 20)

# Define a parameter vector with estimates
β = jnp.array([0.26, 0.18, 0.25, -0.1, -0.22])

# Create some observations X
datasets = [
    jnp.array([0, 1, 1, 1, 2]),
    jnp.array([2, 3, 2, 4, 0]),
    jnp.array([3, 4, 5, 3, 2]),
    jnp.array([6, 5, 4, 4, 7]),
]


fig, ax = plt.subplots(figsize=(12, 8))

for X in datasets:
    μ = jnp.exp(X @ β)
    distribution = []
    for y_i in y_values:
        distribution.append(poisson_pmf(y_i, μ))
    ax.plot(
        y_values,
        distribution,
        label=rf"$\mu_i$={μ:.1}",
        marker="o",
        markersize=8,
        alpha=0.5,
    )

ax.grid()
ax.legend()
ax.set_xlabel(r"$y \mid x_i$")
ax.set_ylabel(r"$f(y \mid x_i; \beta )$")
ax.axis(xmin=0, ymin=0)
plt.show()
```



We can see that the distribution of y_i is conditional on $\mathbf{x}_i$ (μ_i is no longer constant).

131.4 Maximum likelihood estimation

In our model for number of billionaires, the conditional distribution contains 4 ($k = 4$) parameters that we need to estimate.

We will label our entire parameter vector as β where

$$\beta = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \beta_3 \end{bmatrix}$$

To estimate the model using MLE, we want to maximize the likelihood that our estimate $\hat{\beta}$ is the true parameter β.

Intuitively, we want to find the $\hat{\beta}$ that best fits our data.

First, we need to construct the likelihood function $\mathcal{L}(\beta)$, which is similar to a joint probability density function.

Assume we have some data $y_i = \{y_1, y_2\}$ and $y_i \sim f(y_i)$.

If y_1 and y_2 are independent, the joint pmf of these data is $f(y_1, y_2) = f(y_1) \cdot f(y_2)$.

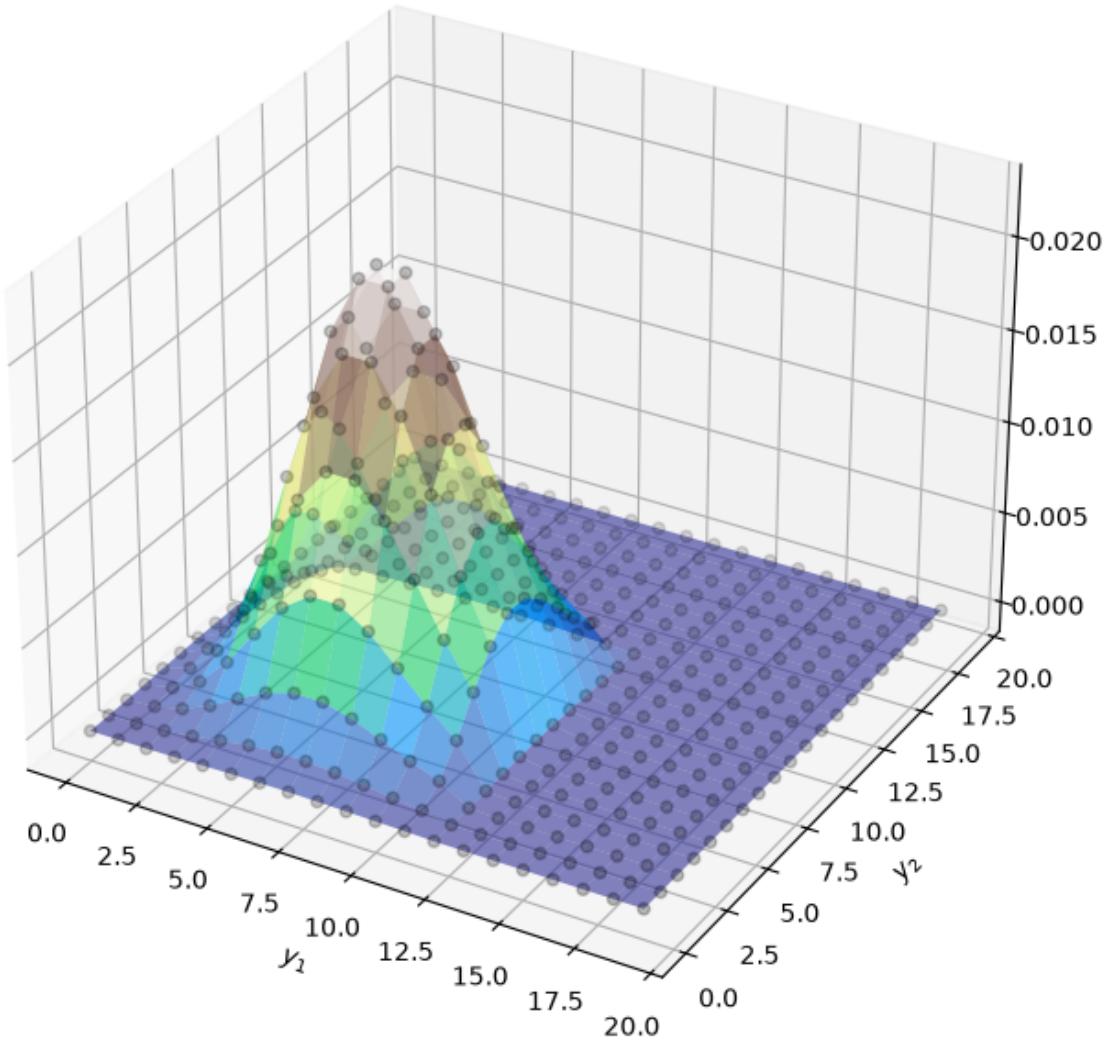
If y_i follows a Poisson distribution with $\lambda = 7$, we can visualize the joint pmf like so

```
def plot_joint_poisson(μ=7, y_n=20):
    yi_values = jnp.arange(0, y_n, 1)
```

(continues on next page)

(continued from previous page)

```
    # Create coordinate points of X and Y
    X, Y = jnp.meshgrid(yi_values, yi_values)

    # Multiply distributions together
    Z = poisson_pmf(X, μ) * poisson_pmf(Y, μ)

    fig = plt.figure(figsize=(12, 8))
    ax = fig.add_subplot(111, projection="3d")
    ax.plot_surface(X, Y, Z.T, cmap="terrain", alpha=0.6)
    ax.scatter(X, Y, Z.T, color="black", alpha=0.5, linewidths=1)
    ax.set(xlabel=r"$y_1$", ylabel=r"$y_2$")
    ax.set_zlabel(r"$f(y_1, y_2)$", labelpad=10)
    plt.show()


plot_joint_poisson(μ=7, y_n=20)
```



Similarly, the joint pmf of our data (which is distributed as a conditional Poisson distribution) can be written as

$$f(y_1, y_2, \ldots, y_n \mid \mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n; \beta) = \prod_{i=1}^{n} \frac{\mu_i^{y_i}}{y_i!} e^{-\mu_i}$$

y_i is conditional on both the values of $\mathbf{x}_i$ and the parameters β.

The likelihood function is the same as the joint pmf, but treats the parameter β as a random variable and takes the observations $(y_i, \mathbf{x}_i)$ as given

$$\begin{aligned}\mathcal{L}(\beta \mid y_1, y_2, \ldots, y_n \; ; \; \mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n) &= \prod_{i=1}^{n} \frac{\mu_i^{y_i}}{y_i!} e^{-\mu_i} \\ &= f(y_1, y_2, \ldots, y_n \mid \mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n; \beta)\end{aligned}$$

Now that we have our likelihood function, we want to find the $\widehat{\beta}$ that yields the maximum likelihood value

$$\max_{\beta} \mathcal{L}(\beta)$$

In doing so it is generally easier to maximize the log-likelihood (consider differentiating $f(x) = x \exp(x)$ vs. $f(x) = \log(x) + x$).

Given that taking a logarithm is a monotone increasing transformation, a maximizer of the likelihood function will also be a maximizer of the log-likelihood function.

In our case the log-likelihood is

$$\begin{aligned}\log \mathcal{L}(\beta) &= \log\Big(f(y_1; \beta) \cdot f(y_2; \beta) \cdot \ldots \cdot f(y_n; \beta)\Big) \\ &= \sum_{i=1}^{n} \log f(y_i; \beta) \\ &= \sum_{i=1}^{n} \log\Big(\frac{\mu_i^{y_i}}{y_i!} e^{-\mu_i}\Big) \\ &= \sum_{i=1}^{n} y_i \log \mu_i - \sum_{i=1}^{n} \mu_i - \sum_{i=1}^{n} \log y_i!\end{aligned}$$

The MLE of the Poisson for $\widehat{\beta}$ can be obtained by solving

$$\max_{\beta}\Big(\sum_{i=1}^{n} y_i \log \mu_i - \sum_{i=1}^{n} \mu_i - \sum_{i=1}^{n} \log y_i!\Big)$$

However, no analytical solution exists to the above problem – to find the MLE we need to use numerical methods.

131.5 MLE with numerical methods

Many distributions do not have nice, analytical solutions and therefore require numerical methods to solve for parameter estimates.

One such numerical method is the Newton-Raphson algorithm.

Our goal is to find the maximum likelihood estimate $\widehat{\beta}$.

At $\widehat{\beta}$, the first derivative of the log-likelihood function will be equal to 0.

Let's illustrate this by supposing

$$\log \mathcal{L}(\beta) = -(\beta - 10)^2 - 10$$

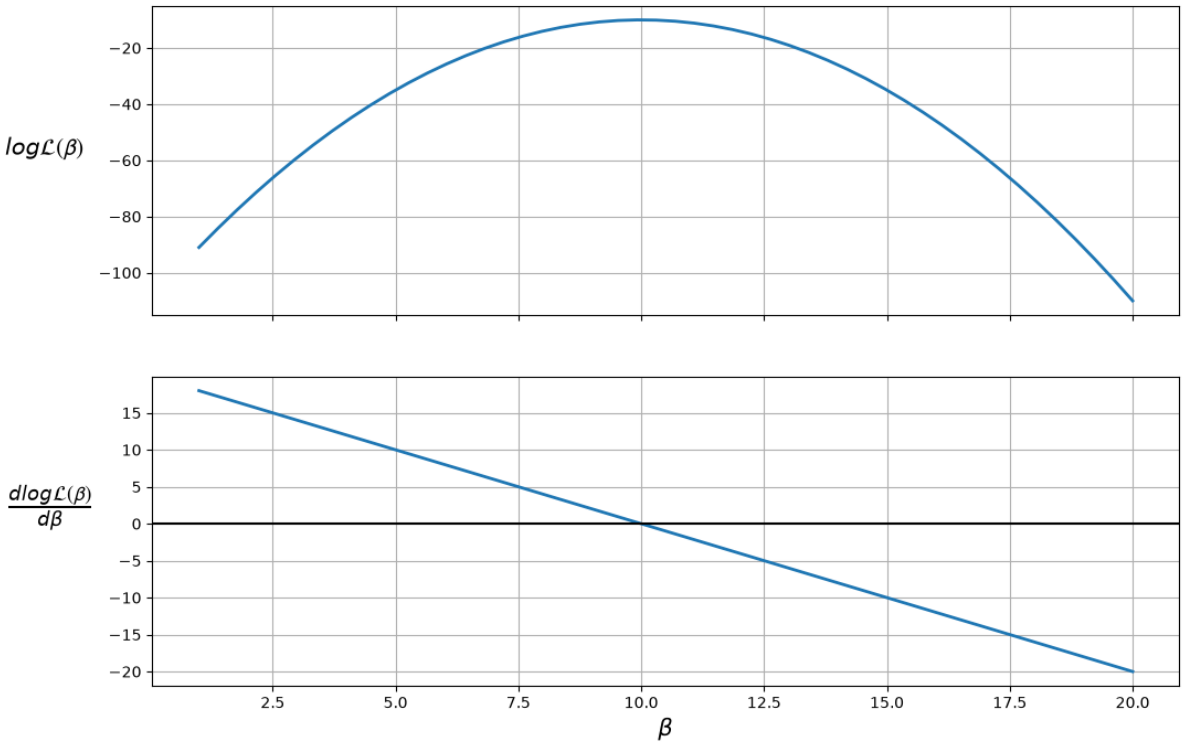
```
@jax.jit
def logL(β):
    return -((β - 10) ** 2) - 10
```

To find the value of the gradient of the above function, we can use jax.grad which auto-differentiates the given function.

We further use jax.vmap which vectorizes the given function i.e. the function acting upon scalar inputs can now be used with vector inputs.

```
dlogL = jax.vmap(jax.grad(logL))
```

```
β = jnp.linspace(1, 20)

fig, (ax1, ax2) = plt.subplots(2, sharex=True, figsize=(12, 8))

ax1.plot(β, logL(β), lw=2)
ax2.plot(β, dlogL(β), lw=2)

ax1.set_ylabel(
    r"$log \mathcal{L(\beta)}$", rotation=0, labelpad=35, fontsize=15
)
ax2.set_ylabel(
    r"$\frac{dlog \mathcal{L(\beta)}}{d \beta}$ ",
    rotation=0,
    labelpad=35,
    fontsize=19,
)
ax2.set_xlabel(r"$\beta$", fontsize=15)
ax1.grid(), ax2.grid()
plt.axhline(c="black")
plt.show()
```



The plot shows that the maximum likelihood value (the top plot) occurs when $\frac{d \log \mathcal{L}(\beta)}{d\beta} = 0$ (the bottom plot).

Therefore, the likelihood is maximized when $\beta = 10$.

We can also ensure that this value is a *maximum* (as opposed to a minimum) by checking that the second derivative (slope of the bottom plot) is negative.

The Newton-Raphson algorithm finds a point where the first derivative is 0.

To use the algorithm, we take an initial guess at the maximum value, β_0 (the OLS parameter estimates might be a reasonable guess), then

1. Use the updating rule to iterate the algorithm

$$\beta_{(k+1)} = \beta_{(k)} - H^{-1}(\beta_{(k)})G(\beta_{(k)})$$

   where:

$$G(\beta_{(k)}) = \frac{d \log \mathcal{L}(\beta_{(k)})}{d\beta_{(k)}}$$
$$H(\beta_{(k)}) = \frac{d^2 \log \mathcal{L}(\beta_{(k)})}{d\beta_{(k)} d\beta'_{(k)}}$$

2. Check whether $\beta_{(k+1)} - \beta_{(k)} < tol$
   - If true, then stop iterating and set $\widehat{\beta} = \beta_{(k+1)}$
   - If false, then update $\beta_{(k+1)}$

As can be seen from the updating equation, $\beta_{(k+1)} = \beta_{(k)}$ only when $G(\beta_{(k)}) = 0$ i.e. where the first derivative is equal to 0.

(In practice, we stop iterating when the difference is below a small tolerance threshold.)

Let's have a go at implementing the Newton-Raphson algorithm.

First, we'll create a class called `PoissonRegression` so we can easily recompute the values of the log likelihood, gradient and Hessian for every iteration

```
class PoissonRegression(NamedTuple):
    X: jnp.ndarray
    y: jnp.ndarray
```

Now we can define the log likelihood function in Python

```
@jax.jit
def logL(β, model):
    y = model.y
    μ = jnp.exp(model.X @ β)
    return jnp.sum(model.y * jnp.log(μ) - μ - jnp.log(factorial(y)))
```

To find the gradient of the `poisson_logL`, we again use jax.grad.

According to the documentation,

- `jax.jacfwd` uses forward-mode automatic differentiation, which is more efficient for "tall" Jacobian matrices, while
- `jax.jacrev` uses reverse-mode, which is more efficient for "wide" Jacobian matrices.

(The documentation also states that when matrices that are near-square, `jax.jacfwd` probably has an edge over `jax.jacrev`.)

Therefore, to find the Hessian, we can directly use `jax.jacfwd`.

```
G_logL = jax.grad(logL)
H_logL = jax.jacfwd(G_logL)
```

Our function `newton_raphson` will take a `PoissonRegression` object that has an initial guess of the parameter vector β_0.

The algorithm will update the parameter vector according to the updating rule, and recalculate the gradient and Hessian matrices at the new parameter estimates.

Iteration will end when either:

- The difference between the parameter and the updated parameter is below a tolerance level.
- The maximum number of iterations has been achieved (meaning convergence is not achieved).

So we can get an idea of what's going on while the algorithm is running, an option `display=True` is added to print out values at each iteration.

```
def newton_raphson(model, β, tol=1e-3, max_iter=100, display=True):

    i = 0
    error = 100  # Initial error value

    # Print header of output
    if display:
        header = f'{"Iteration_k":<13}{"Log-likelihood":<16}{"θ":<60}'
        print(header)
        print("-" * len(header))

    # While loop runs while any value in error is greater
    # than the tolerance until max iterations are reached
    while jnp.any(error > tol) and i < max_iter:
        H, G = jnp.squeeze(H_logL(β, model)), G_logL(β, model)
        β_new = β - (jnp.dot(jnp.linalg.inv(H), G))
        error = jnp.abs(β_new - β)
        β = β_new

        if display:
            β_list = [f"{t:.3}" for t in list(β.flatten())]
            update = f"{i:<13}{logL(β, model):<16.8}{β_list}"
            print(update)

        i += 1

    print(f"Number of iterations: {i}")
    print(f"β_hat = {β.flatten()}")

    return β
```

Let's try out our algorithm with a small dataset of 5 observations and 3 variables in $\mathbf{X}$.

```
X = jnp.array([[1, 2, 5], [1, 1, 3], [1, 4, 2], [1, 5, 2], [1, 3, 1]])

y = jnp.array([1, 0, 1, 1, 0])
```

(continues on next page)

(continued from previous page)

```
# Take a guess at initial βs
init_β = jnp.array([0.1, 0.1, 0.1])

# Create an object with Poisson model values
poi = PoissonRegression(X=X, y=y)

# Use newton_raphson to find the MLE
β_hat = newton_raphson(poi, init_β, display=True)
```

```
Iteration_k  Log-likelihood  θ
--------------------------------------------------------------------------------
 ↪------
0             -4.3447633      ['-1.49', '0.265', '0.244']
1             -3.5742409      ['-3.38', '0.528', '0.474']
2             -3.3999527      ['-5.06', '0.782', '0.702']
3             -3.3788645      ['-5.92', '0.909', '0.82']
4             -3.3783555      ['-6.07', '0.933', '0.843']
5             -3.3783557      ['-6.08', '0.933', '0.843']
6             -3.3783557      ['-6.08', '0.933', '0.843']
Number of iterations: 7
β_hat = [-6.078486   0.9334028  0.8432968]
```

As this was a simple model with few observations, the algorithm achieved convergence in only 7 iterations.

You can see that with each iteration, the log-likelihood value increased.

Remember, our objective was to maximize the log-likelihood function, which the algorithm has worked to achieve.

Also, note that the increase in $\log \mathcal{L}(\beta_{(k)})$ becomes smaller with each iteration.

This is because the gradient is approaching 0 as we reach the maximum, and therefore the numerator in our updating equation is becoming smaller.

The gradient vector should be close to 0 at $\hat{\beta}$

```
G_logL(β_hat, poi)
```

```
Array([ 7.4505806e-09, -2.9802322e-07,  3.7252903e-08], dtype=float32)
```

The iterative process can be visualized in the following diagram, where the maximum is found at $\beta = 10$

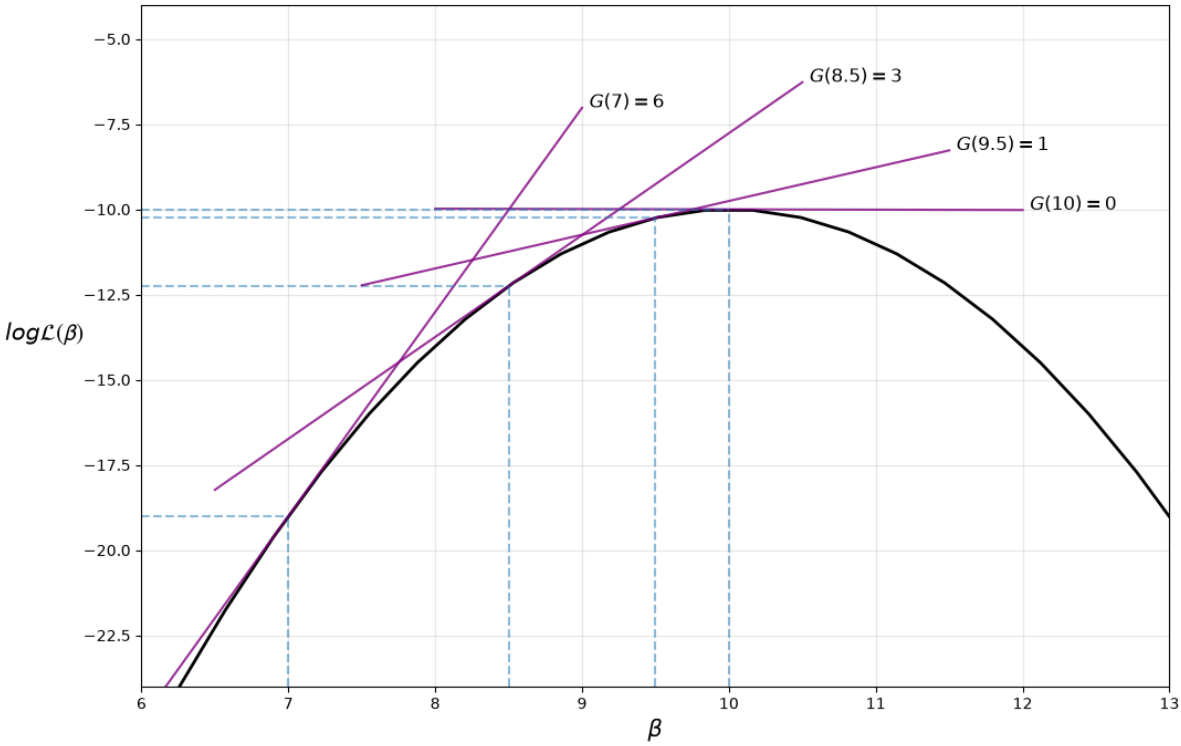
```
@jax.jit
def logL(x):
    return -((x - 10) ** 2) - 10


@jax.jit
def find_tangent(β, a=0.01):
    y1 = logL(β)
    y2 = logL(β + a)
    x = jnp.array([[β, 1], [β + a, 1]])
    m, c = jnp.linalg.lstsq(x, jnp.array([y1, y2]), rcond=None)[0]
    return m, c
```

```
β = jnp.linspace(2, 18)
fig, ax = plt.subplots(figsize=(12, 8))
ax.plot(β, logL(β), lw=2, c="black")
```

(continues on next page)

(continued from previous page)

```
for β in [7, 8.5, 9.5, 10]:
    β_line = jnp.linspace(β - 2, β + 2)
    m, c = find_tangent(β)
    y = m * β_line + c
    ax.plot(β_line, y, "-", c="purple", alpha=0.8)
    ax.text(β + 2.05, y[-1], rf"$G({β}) = {abs(m):.0f}$", fontsize=12)
    ax.vlines(β, -24, logL(β), linestyles="--", alpha=0.5)
    ax.hlines(logL(β), 6, β, linestyles="--", alpha=0.5)

ax.set(ylim=(-24, -4), xlim=(6, 13))
ax.set_xlabel(r"$\beta$", fontsize=15)
ax.set_ylabel(
    r"$log \mathcal{L(\beta)}$", rotation=0, labelpad=25, fontsize=15
)
ax.grid(alpha=0.3)
plt.show()
```



Note that our implementation of the Newton-Raphson algorithm is rather basic — for more robust implementations see, for example, scipy.optimize.

131.6 Maximum likelihood estimation with `statsmodels`

Now that we know what's going on under the hood, we can apply MLE to an interesting application.

We'll use the Poisson regression model in `statsmodels` to obtain a richer output with standard errors, test values, and more.

`statsmodels` uses the same algorithm as above to find the maximum likelihood estimates.

Before we begin, let's re-estimate our simple model with `statsmodels` to confirm we obtain the same coefficients and log-likelihood value.

Now, as `statsmodels` accepts only NumPy arrays, we can use `np.array` method to convert them to NumPy arrays.

```
X = jnp.array([[1, 2, 5], [1, 1, 3], [1, 4, 2], [1, 5, 2], [1, 3, 1]])

y = jnp.array([1, 0, 1, 1, 0])

y_numpy = np.array(y)
X_numpy = np.array(X)
stats_poisson = Poisson(y_numpy, X_numpy).fit()
print(stats_poisson.summary())
```

```
Optimization terminated successfully.
         Current function value: 0.675671
         Iterations 7
                          Poisson Regression Results
==============================================================================
Dep. Variable:                      y   No. Observations:                    5
Model:                        Poisson   Df Residuals:                        2
Method:                           MLE   Df Model:                            2
Date:                Mon, 20 Jul 2026   Pseudo R-squ.:                  0.2546
Time:                        05:07:38   Log-Likelihood:                -3.3784
converged:                       True   LL-Null:                       -4.5325
Covariance Type:            nonrobust   LLR p-value:                    0.3153
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const         -6.0785      5.279     -1.151      0.250     -16.425       4.268
x1             0.9334      0.829      1.126      0.260      -0.691       2.558
x2             0.8433      0.798      1.057      0.291      -0.720       2.407
==============================================================================
```

Now let's replicate results from Daniel Treisman's paper, Russia's Billionaires, mentioned earlier in the lecture.

Treisman starts by estimating equation (131.1), where:

- y_i is $number\ of\ billionaires_i$
- x_{i1} is $\log GDP\ per\ capita_i$
- x_{i2} is $\log population_i$
- x_{i3} is $years\ in\ GATT_i$ – years membership in GATT and WTO (to proxy access to international markets)

The paper only considers the year 2008 for estimation.

We will set up our variables for estimation like so (you should have the data assigned to `df` from earlier in the lecture)

```
# Keep only year 2008
df = df[df["year"] == 2008]

# Add a constant
df["const"] = 1

# Variable sets
reg1 = ["const", "lngdppc", "lnpop", "gattwto08"]
reg2 = [
    "const",
    "lngdppc",
    "lnpop",
    "gattwto08",
    "lnmcap08",
    "rintr",
    "topint08",
]
reg3 = [
    "const",
    "lngdppc",
    "lnpop",
    "gattwto08",
    "lnmcap08",
    "rintr",
    "topint08",
    "nrrents",
    "roflaw",
]
```

Then we can use the `Poisson` function from `statsmodels` to fit the model.

We'll use robust standard errors as in the author's paper

```
# Specify model
poisson_reg = Poisson(df[["numbil0"]], df[reg1], missing="drop").fit(
    cov_type="HC0"
)
print(poisson_reg.summary())
```

```
Optimization terminated successfully.
         Current function value: 2.226090
         Iterations 9
                          Poisson Regression Results
==============================================================================
Dep. Variable:                numbil0   No. Observations:                  197
Model:                        Poisson   Df Residuals:                      193
Method:                           MLE   Df Model:                            3
Date:                Mon, 20 Jul 2026   Pseudo R-squ.:                  0.8574
Time:                        05:07:38   Log-Likelihood:                -438.54
converged:                       True   LL-Null:                       -3074.7
Covariance Type:                  HC0   LLR p-value:                     0.000
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const        -29.0495      2.578    -11.268      0.000     -34.103     -23.997
lngdppc        1.0839      0.138      7.834      0.000       0.813       1.355
lnpop          1.1714      0.097     12.024      0.000       0.980       1.362
```

(continues on next page)

(continued from previous page)

```
gattwto08      0.0060      0.007      0.868      0.386      -0.008       0.019
==============================================================================
```

Success! The algorithm was able to achieve convergence in 9 iterations.

Our output indicates that GDP per capita, population, and years of membership in the General Agreement on Tariffs and Trade (GATT) are positively related to the number of billionaires a country has, as expected.

Let's also estimate the author's more full-featured models and display them in a single table

```
regs = [reg1, reg2, reg3]
reg_names = ["Model 1", "Model 2", "Model 3"]
info_dict = {
    "Pseudo R-squared": lambda x: f"{x.prsquared:.2f}",
    "No. observations": lambda x: f"{int(x.nobs):d}",
}
regressor_order = [
    "const",
    "lngdppc",
    "lnpop",
    "gattwto08",
    "lnmcap08",
    "rintr",
    "topint08",
    "nrrents",
    "roflaw",
]
results = []

for reg in regs:
    result = Poisson(df[["numbil0"]], df[reg], missing="drop").fit(
        cov_type="HC0", maxiter=100, disp=0
    )
    results.append(result)

results_table = summary_col(
    results=results,
    float_format="%0.3f",
    stars=True,
    model_names=reg_names,
    info_dict=info_dict,
    regressor_order=regressor_order,
)
results_table.add_title(
    "Table 1 - Explaining the Number of Billionaires \
                        in 2008"
)
print(results_table)
```

```
Table 1 - Explaining the Number of Billionaires                        in 2008
================================================
                  Model 1    Model 2    Model 3
------------------------------------------------
const            -29.050*** -19.444*** -20.858***
                 (2.578)    (4.820)    (4.255)
lngdppc          1.084***   0.717***   0.737***
                 (0.138)    (0.244)    (0.233)
```

(continues on next page)

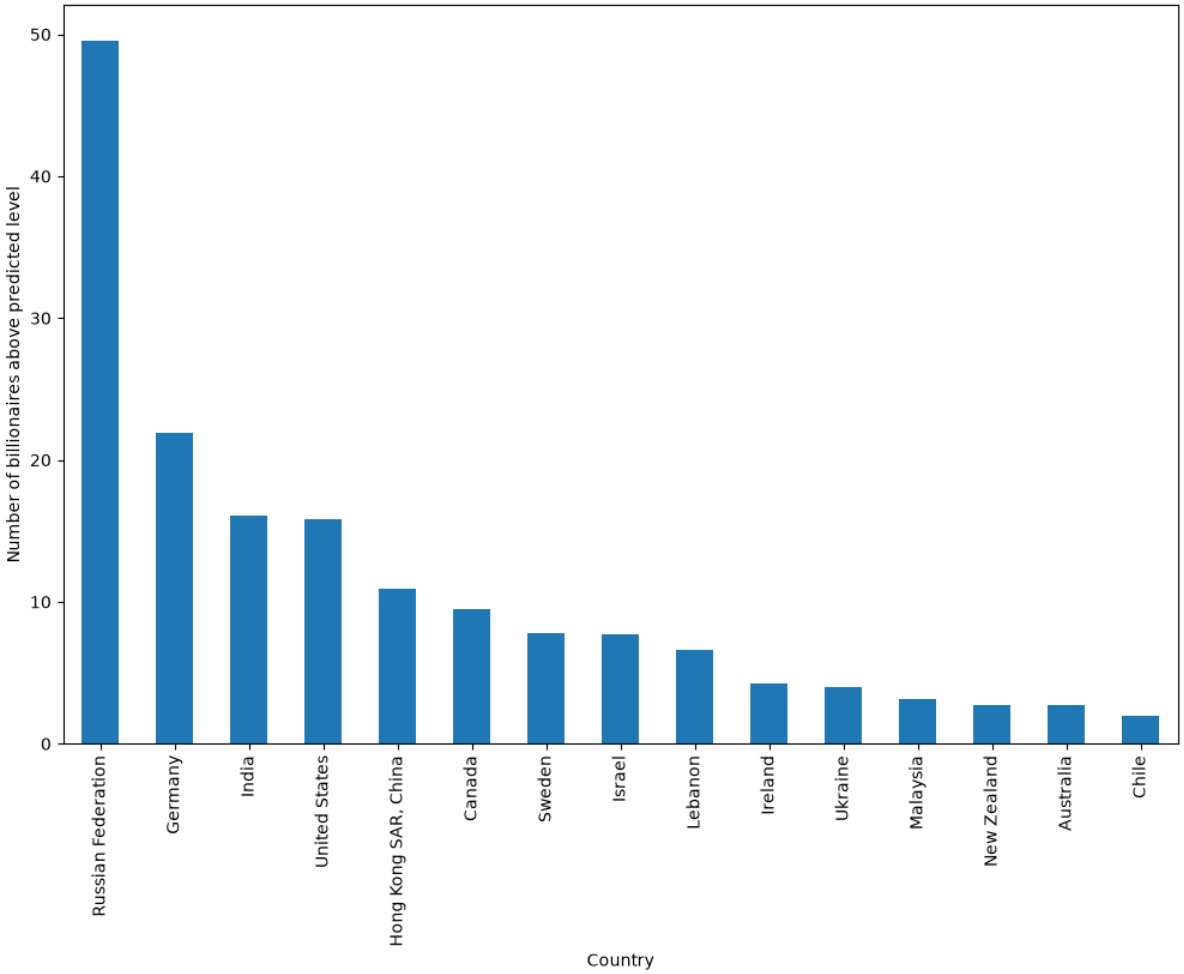
(continued from previous page)

```
lnpop             1.171***   0.806***   0.929***
                  (0.097)    (0.213)    (0.195)
gattwto08         0.006      0.007      0.004
                  (0.007)    (0.006)    (0.006)
lnmcap08                     0.399**    0.286*
                             (0.172)    (0.167)
rintr                        -0.010     -0.009
                             (0.010)    (0.010)
topint08                     -0.051***  -0.058***
                             (0.011)    (0.012)
nrrents                                 -0.005
                                        (0.010)
roflaw                                  0.203
                                        (0.372)
No. observations 197         131        131
Pseudo R-squared 0.86        0.90       0.90
================================================
Standard errors in parentheses.
* p<.1, ** p<.05, ***p<.01
```

The output suggests that the frequency of billionaires is positively correlated with GDP per capita, population size, stock market capitalization, and negatively correlated with top marginal income tax rate.

To analyze our results by country, we can plot the difference between the predicted and actual values, then sort from highest to lowest and plot the first 15

```
data = [
    "const",
    "lngdppc",
    "lnpop",
    "gattwto08",
    "lnmcap08",
    "rintr",
    "topint08",
    "nrrents",
    "roflaw",
    "numbil0",
    "country",
]
results_df = df[data].dropna()

# Use last model (model 3)
results_df["prediction"] = results[-1].predict()

# Calculate difference
results_df["difference"] = results_df["numbil0"] - results_df["prediction"]

# Sort in descending order
results_df.sort_values("difference", ascending=False, inplace=True)

# Plot the first 15 data points
results_df[:15].plot(
    "country", "difference", kind="bar", figsize=(12, 8), legend=False
)
plt.ylabel("Number of billionaires above predicted level")
plt.xlabel("Country")
plt.show()
```



As we can see, Russia has by far the highest number of billionaires in excess of what is predicted by the model (around 50 more than expected).

Treisman uses this empirical result to discuss possible reasons for Russia's excess of billionaires, including the origination of wealth in Russia, the political climate, and the history of privatization in the years after the USSR.

131.7 Summary

In this lecture, we used Maximum Likelihood Estimation to estimate the parameters of a Poisson model.

`statsmodels` contains other built-in likelihood models such as Probit and Logit.

For further flexibility, `statsmodels` provides a way to specify the distribution manually using the `GenericLikelihoodModel` class - an example notebook can be found here.

131.8 Exercises

Exercise 131.8.1

Suppose we wanted to estimate the probability of an event y_i occurring, given some observations.

We could use a probit regression model, where the pmf of y_i is

$$f(y_i; \beta) = \mu_i^{y_i}(1-\mu_i)^{1-y_i}, \quad y_i = 0, 1$$
$$\text{where} \quad \mu_i = \Phi(\mathbf{x}_i'\beta)$$

Φ represents the **cumulative normal distribution** and constrains the predicted y_i to be between 0 and 1 (as required for a probability).

β is a vector of coefficients.

Following the example in the lecture, write a class to represent the Probit model.

To begin, find the log-likelihood function and derive the gradient and Hessian.

The `jax.scipy.stats` module `norm` contains the functions needed to compute the cdf and pdf of the normal distribution.

Solution

The log-likelihood can be written as

$$\log \mathcal{L} = \sum_{i=1}^{n} \left[y_i \log \Phi(\mathbf{x}_i'\beta) + (1-y_i) \log(1 - \Phi(\mathbf{x}_i'\beta)) \right]$$

Using the **fundamental theorem of calculus**, the derivative of a cumulative probability distribution is its marginal distribution

$$\frac{\partial}{\partial s}\Phi(s) = \phi(s)$$

where ϕ is the marginal normal distribution.

The gradient vector of the Probit model is

$$\frac{\partial \log \mathcal{L}}{\partial \beta} = \sum_{i=1}^{n} \left[y_i \frac{\phi(\mathbf{x}_i'\beta)}{\Phi(\mathbf{x}_i'\beta)} - (1-y_i)\frac{\phi(\mathbf{x}_i'\beta)}{1-\Phi(\mathbf{x}_i'\beta)} \right] \mathbf{x}_i$$

The Hessian of the Probit model is

$$\frac{\partial^2 \log \mathcal{L}}{\partial \beta \partial \beta'} = -\sum_{i=1}^{n} \phi(\mathbf{x}_i'\beta) \left[y_i \frac{\phi(\mathbf{x}_i'\beta) + \mathbf{x}_i'\beta\Phi(\mathbf{x}_i'\beta)}{[\Phi(\mathbf{x}_i'\beta)]^2} + (1-y_i)\frac{\phi(\mathbf{x}_i'\beta) - \mathbf{x}_i'\beta(1-\Phi(\mathbf{x}_i'\beta))}{[1-\Phi(\mathbf{x}_i'\beta)]^2} \right] \mathbf{x}_i\mathbf{x}_i'$$

Using these results, we can write a class for the Probit model as follows

```
class ProbitRegression(NamedTuple):
    X: jnp.ndarray
    y: jnp.ndarray
```

```
@jax.jit
def logL(β, model):
    y = model.y
    μ = norm.cdf(model.X @ β.T)
    return y @ jnp.log(μ) + (1 - y) @ jnp.log(1 - μ)
```

```
G_logL = jax.grad(logL)
H_logL = jax.jacfwd(G_logL)
```

Exercise 131.8.2

Use the following dataset and initial values of β to estimate the MLE with the Newton-Raphson algorithm developed earlier in the lecture

$$\mathbf{X} = \begin{bmatrix} 1 & 2 & 4 \\ 1 & 1 & 1 \\ 1 & 4 & 3 \\ 1 & 5 & 6 \\ 1 & 3 & 5 \end{bmatrix} \quad y = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \quad \beta_{(0)} = \begin{bmatrix} 0.1 \\ 0.1 \\ 0.1 \end{bmatrix}$$

Verify your results with `statsmodels` - you can import the Probit function with the following import statement

```
from statsmodels.discrete.discrete_model import Probit
```

Note that the simple Newton-Raphson algorithm developed in this lecture is very sensitive to initial values, and therefore you may fail to achieve convergence with different starting values.

Solution

Here is one solution

```
X = jnp.array([[1, 2, 4], [1, 1, 1], [1, 4, 3], [1, 5, 6], [1, 3, 5]])

y = jnp.array([1, 0, 1, 1, 0])

# Take a guess at initial βs
β = jnp.array([0.1, 0.1, 0.1])

# Create a model of Probit regression
prob = ProbitRegression(y=y, X=X)

# Run Newton-Raphson algorithm
newton_raphson(prob, β)
```

```
Iteration_k  Log-likelihood  θ
--------------------------------------------------------------------------------
 ↪---------
0             -2.3796887      ['-1.34', '0.775', '-0.157']
1             -2.3687525      ['-1.53', '0.775', '-0.0981']
2             -2.3687296      ['-1.55', '0.778', '-0.0971']
3             -2.3687291      ['-1.55', '0.778', '-0.0971']
Number of iterations: 4
β_hat = [-1.5462587   0.77778953 -0.09709755]
```

```
Array([-1.5462587 ,  0.77778953, -0.09709755], dtype=float32)
```

```
# Use statsmodels to verify results
y_numpy = np.array(y)
X_numpy = np.array(X)
print(Probit(y_numpy, X_numpy).fit().summary())
```

```
Optimization terminated successfully.
         Current function value: 0.473746
         Iterations 6
                          Probit Regression Results
==============================================================================
Dep. Variable:                      y   No. Observations:                    5
Model:                         Probit   Df Residuals:                        2
Method:                           MLE   Df Model:                            2
Date:                Mon, 20 Jul 2026   Pseudo R-squ.:                  0.2961
Time:                        05:07:39   Log-Likelihood:                -2.3687
converged:                       True   LL-Null:                       -3.3651
Covariance Type:            nonrobust   LLR p-value:                    0.3692
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const         -1.5463      1.866     -0.829      0.407      -5.204       2.111
x1             0.7778      0.788      0.986      0.324      -0.768       2.323
x2            -0.0971      0.590     -0.165      0.869      -1.254       1.060
==============================================================================
```

CHAPTER

TWO

A LINEAR MODEL OF UNEMPLOYMENT

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

In addition to what's in Anaconda, this lecture needs the following libraries.

We first install `numpyro` and `jax`:

```
!pip install numpyro jax
```

We also install `pandas_datareader`, which we use to download data from FRED:

```
!pip install pandas_datareader
```

132.1 Overview

This lecture applies Bayesian estimation to a linear AR(1) model of the US unemployment rate.

We met the machinery in *Posterior Distributions for AR(1) Parameters*, but there the data were simulated and the focus was theoretical.

Here we work carefully through real data, organizing the lecture around a question that divided macroeconomists in the 1980s and 1990s: is unemployment a *random walk*?

In the process, we will observe an asymmetry the model cannot capture.

This will motivate a sequel lecture, titled *Unemployment Dynamics with Asymmetric Shocks*.

As in *Posterior Distributions for AR(1) Parameters* and *Non-Conjugate Priors*, we estimate by sampling posteriors with the NUTS sampler in NumPyro.

(See the *introduction to NUTS* in *Non-Conjugate Priors* for a brief account of how it works.)

Let's start with some imports.

```
import numpy as np
import matplotlib.pyplot as plt
import datetime as dt
from pandas_datareader import data as web

import jax.numpy as jnp
from jax import random
import numpyro
import numpyro.distributions as dist
from numpyro.infer import MCMC, NUTS
```

132.2 The natural rate versus hysteresis

In the early 1980s, Nelson and Plosser [1982] launched the unit-root literature.

They examined 14 macroeconomic time series and found that they could reject the random walk for only one of them.

Roughly speaking, this meant that, for most macroeconomic series, the effects of shocks looked permanent rather than transitory.

For unemployment, that view found expression in the **hysteresis hypothesis** of [Blanchard and Summers, 1986].

This hypothesis states that a shock to unemployment can be more or less permanent, because spells of joblessness erode skills and attachment to the labor force.

In contrast, the **natural rate hypothesis** of [Friedman, 1968] holds that unemployment fluctuates around a stable equilibrium rate — and hence shocks are transitory rather than permanent.

The debate matters because the two views imply different policies.

In particular, if shocks are permanent, a deep recession leaves a lasting scar, encouraging remedial action.

Here we reexamine the question with Bayesian estimation.

Note

There is an irony in the history described above.

While Nelson and Plosser [1982] fueled the unit root debate, and hence the hysteresis hypothesis, the one macroeconomic series they rejected the unit root for was the unemployment rate.

132.3 The data

We use the US civilian unemployment rate, series `UNRATE` from FRED, monthly and seasonally adjusted.

```
start, end = dt.datetime(1948, 1, 1), dt.datetime(2024, 12, 31)
unrate = web.DataReader("UNRATE", "fred", start, end)["UNRATE"]
```

The COVID-19 spike of 2020 is an extreme outlier driven by events our models know nothing about, so we drop it.

```
pre_covid = unrate[unrate.index < "2020-01-01"]
u_monthly = pre_covid.to_numpy()
print(f"{len(u_monthly)} monthly observations")
```

```
864 monthly observations
```

Here is the monthly series.

```
fig, ax = plt.subplots()
ax.plot(pre_covid.index, u_monthly, lw=2)
ax.set_xlabel('year')
ax.set_ylabel('unemployment rate (%)')
plt.show()
```

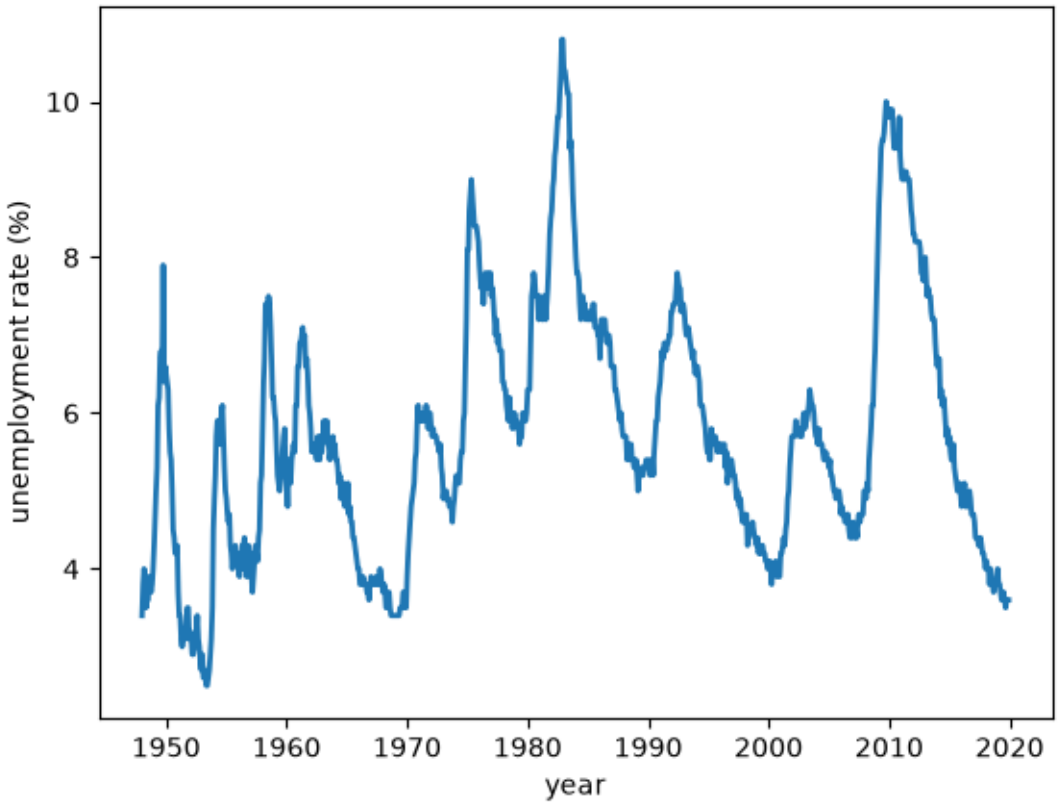


Fig. 132.1: US monthly unemployment rate, 1948–2019

As Fig. 132.1 shows, the rate rises sharply in recessions and drifts down in recoveries, but it always stays inside a band — roughly 3% to 11% over the whole post-war period.

(Keep that band in mind, since it connects back to the unit root debate, as we discuss below.)

132.4 A linear model of unemployment

Let's now set up and estimate our model, using monthly data.

132.4.1 The model

We let unemployment be pulled back toward a normal level $\bar{u}$:

$$u_{t+1} = \bar{u} + \phi\,(u_t - \bar{u}) + \varepsilon_{t+1}, \qquad \varepsilon_{t+1} \sim N(0, \sigma^2), \tag{132.1}$$

with $0 \leq \phi < 1$.

This is a linear AR(1) model, written so that $\bar{u}$ is the level the series reverts to and ϕ measures persistence.

The closer ϕ is to one, the more slowly the series returns to $\bar{u}$ after a shock; the random walk is the limiting case $\phi = 1$.

132.4.2 Priors

We treat $\bar{u}$, ϕ and σ as unknown and place weakly informative priors on them.

We give ϕ a uniform prior on $[0, 1)$.

The upper endpoint is excluded deliberately, since doing so confines us to the *stationary* region.

This is necessary, as we'll see below.

At the same time, the prior still lets ϕ approach one as closely as the data demand.

We center $\bar{u}$ on a plausible natural rate with a fairly wide normal prior, and give the shock scale σ a half-normal prior.

We write the model as a NumPyro function: each `numpyro.sample` introduces a random variable, and the keyword `obs=` ties the last one to the data, supplying the likelihood.

```
def linear_model(u):
    ubar = numpyro.sample("ubar",  dist.Normal(5.5, 2.0))      # Natural rate prior
    ψ    = numpyro.sample("phi",   dist.Uniform(0.0, 1.0))     # Persistence prior
    σ    = numpyro.sample("sigma", dist.HalfNormal(1.0))       # Volatility prior
    μ = ubar + ψ * (u[:-1] - ubar)
    numpyro.sample("u_obs", dist.Normal(μ, σ), obs=u[1:])
```

The vector `μ` holds the conditional means $\bar{u} + \phi(u_t - \bar{u})$, and `obs=u[1:]` says that each next value is drawn from $N(\mu_t, \sigma^2)$.

This one statement encodes the whole likelihood. (See *Non-Conjugate Priors* for more on writing NumPyro models.)

132.4.3 Estimation

We sample the posterior with NUTS, running four chains so that we can check convergence.

We use `chain_method="vectorized"`, which evaluates all chains together on a single device, so the same code runs unchanged on a CPU or a GPU.

```
def run_nuts(model, data, seed=0, num_warmup=1000, num_samples=2000, num_chains=4):
    "Sample a NumPyro model with the NUTS sampler."
    mcmc = MCMC(NUTS(model),
                num_warmup=num_warmup, num_samples=num_samples,
```

(continues on next page)

(continued from previous page)

```
                num_chains=num_chains, chain_method="vectorized",
                progress_bar=False)
    mcmc.run(random.PRNGKey(seed), jnp.asarray(data))
    return mcmc
```

We fit the model to the monthly data.

```
mcmc_monthly = run_nuts(linear_model, u_monthly)
```

Now we inspect the output.

```
mcmc_monthly.print_summary()
```

```
                mean       std    median      5.0%     95.0%     n_eff     r_hat
       phi       0.99      0.00      0.99      0.99      1.00   3607.29      1.00
     sigma       0.21      0.01      0.21      0.20      0.22   4086.14      1.00
      ubar       5.66      1.17      5.69      3.72      7.51   3250.21      1.00

Number of divergences: 0
```

Each row summarizes the posterior for one parameter.

The `mean`, `median`, and `5.0%`/`95.0%` columns give the posterior mean, median, and a 90% credible interval; `std` is the posterior standard deviation.

The last two columns are convergence diagnostics: `n_eff` is the effective number of independent draws, and `r_hat` compares variation within and across chains — a value very close to 1.0 means the chains agree and the sampler has converged.

Here `r_hat` is essentially one and `n_eff` is large, so we can trust the draws.

The number that matters for us is the posterior for ϕ: its mass crowds right up against one.

In particular, the mean and median are very close to one, while the standard deviation is very small.

In other words, at a monthly frequency, US unemployment is *almost* a random walk.

This is the hysteresis boundary.

Thus, we find that the natural-rate and hysteresis views are nearly indistinguishable in the monthly data — the estimate is consistent with both.

This is why the unit-root tests of the era struggled to settle the debate [Røed, 1997].

132.5 A random walk would wander off

While the estimation above seems to give reasonable support to the unit root hypothesis, there is a good reason to view it as false.

To see the argument, suppose that unemployment really is a pure random walk, with $\phi = 1$:

$$u_{t+1} = u_t + \varepsilon_{t+1}, \qquad \varepsilon_{t+1} \sim N(0, \sigma^2).$$

Then $u_t = u_0 + \sum_{s=1}^{t} \varepsilon_s$, so its variance grows without bound: $\text{Var}(u_t) = t\sigma^2$.

The distribution spreads out forever, and eventually probability mass leaves *every* bounded interval.

We can see this by simulating many random-walk paths, using the observed one-month changes to set the shock size, and watching them fan out.

```
rng = np.random.default_rng(0)
T = len(u_monthly)
σ_rw = np.diff(u_monthly).std()
paths = u_monthly[0] + np.cumsum(rng.normal(0, σ_rw, size=(400, T)), axis=1)

u_min, u_max = u_monthly.min(), u_monthly.max()

fig, ax = plt.subplots()
ax.plot(paths[:60].T, color='C0', lw=0.5, alpha=0.3)
ax.axhspan(u_min, u_max, color='C1', alpha=0.15, label='observed range')
ax.axhline(u_min, color='C1', ls='--', lw=1.5)
ax.axhline(u_max, color='C1', ls='--', lw=1.5)
ax.set_xlabel('months since 1948')
ax.set_ylabel('unemployment rate (%)')
ax.legend()
plt.show()
```

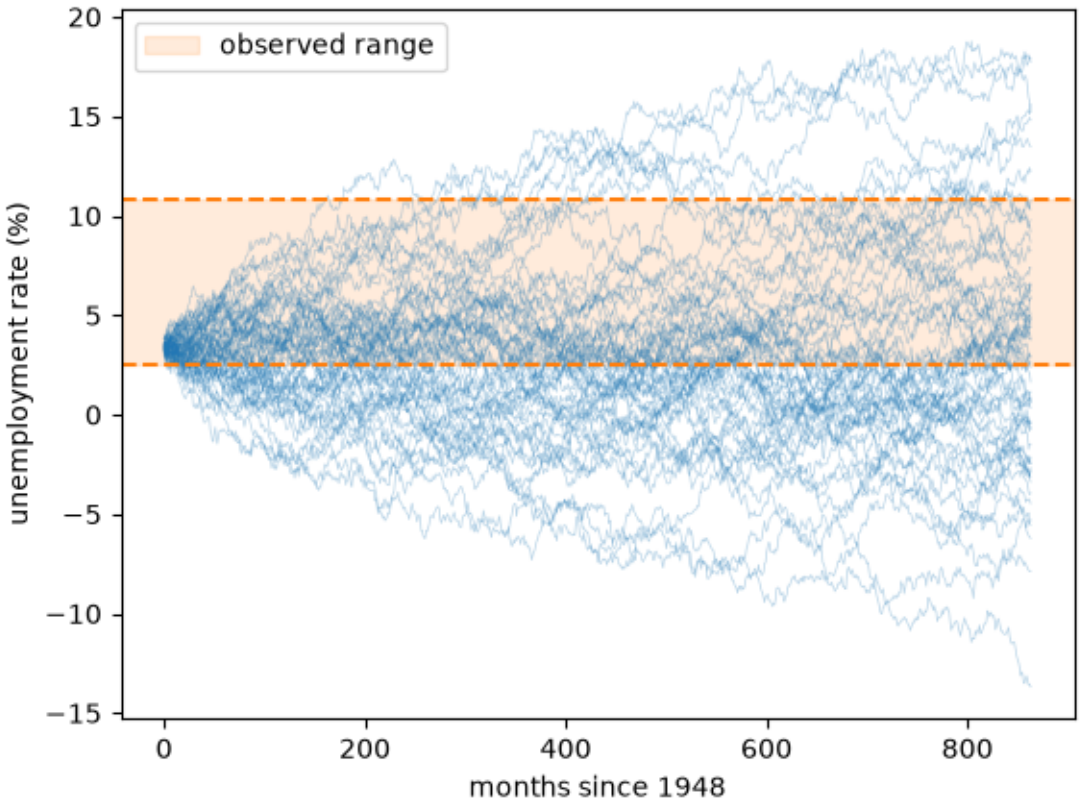


Fig. 132.2: Random walks leave the observed range

In Fig. 132.2 the dashed lines mark the lowest and highest unemployment rates ever seen in the data, and the shaded region is the band between them.

The simulated paths fan out like $\sqrt{t}$ and quickly spread far beyond this band, including into negative rates.

A random walk has no anchor, but unemployment clearly does — it has stayed inside a narrow band for seventy years.

So we can already rule out an exact random walk.

This will become even clearer when we examine annual data.

132.6 Monthly versus annual

The monthly data leave ϕ pinned against one.

Does a different frequency allow us to see something more?

We form an annual series from end-of-year values and fit the same model to it.

```
u_annual = pre_covid.resample("YE").last().to_numpy()
print(f"{len(u_annual)} annual observations")
```

```
72 annual observations
```

We fit the same model to this shorter series.

```
mcmc_annual = run_nuts(linear_model, u_annual)
```

Now we compare the posterior for ϕ across the two frequencies.

```
φ_m = np.asarray(mcmc_monthly.get_samples()["phi"])
φ_a = np.asarray(mcmc_annual.get_samples()["phi"])

fig, ax = plt.subplots()
ax.hist(φ_m, bins=50, density=True, alpha=0.6, label='monthly')
ax.hist(φ_a, bins=50, density=True, alpha=0.6, label='annual')
ax.set_xlabel('$\\phi$')
ax.legend()
plt.show()
```

Fig. 132.3 illustrates our main finding: the annual posterior for ϕ sits well below one, with clear reversion, while the monthly posterior pushes up against the boundary.

This is not a contradiction.

If the monthly persistence is ϕ, then for end-of-year values the persistence is about ϕ^{12}, and raising a number near one to the twelfth power pulls it appreciably below one — in line with our annual estimate.

132.7 What the model misses

Looking again at Fig. 132.1, we notice that unemployment jumps up quickly in recessions and drifts down slowly in recoveries.

This is an *asymmetry* that our current model cannot replicate.

To see why, we look closely at the one part of the model that is random — the shocks.

We proceed in three steps: state what the model assumes about the shocks, recover them from the data, and compare the two.

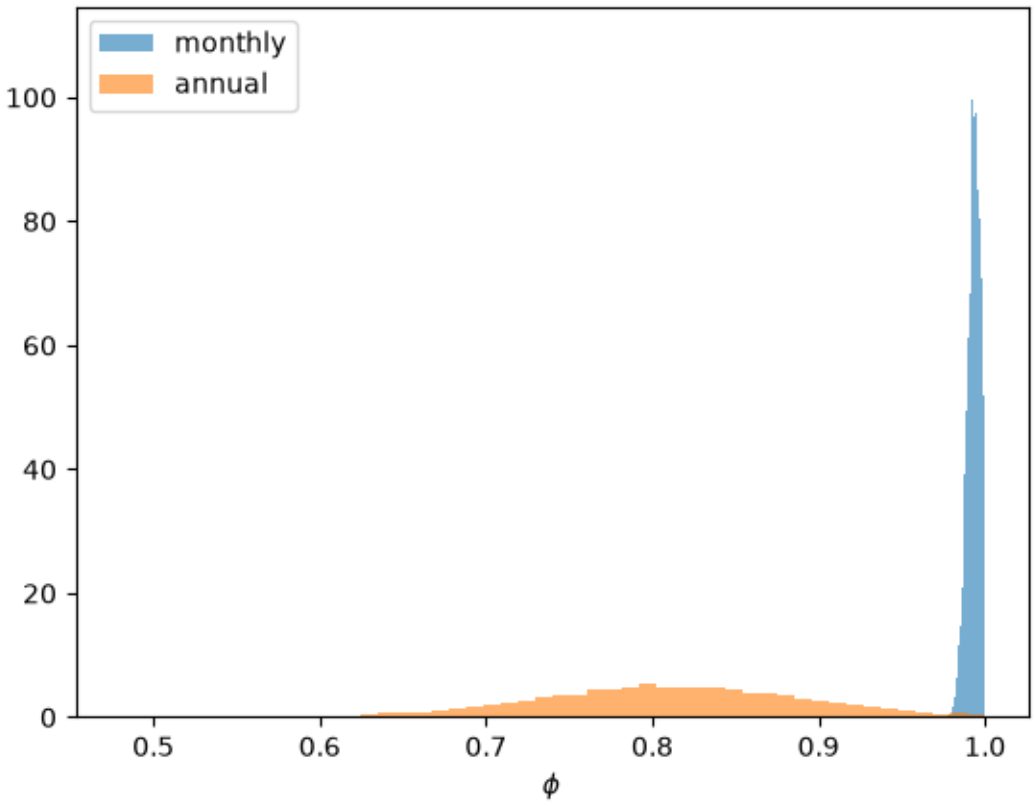


Fig. 132.3: Posterior for the persistence parameter

132.7.1 What the model assumes about the shocks

Given last period's rate u_t and the parameters, the next rate is a deterministic conditional mean plus a shock:

$$u_{t+1} = \underbrace{\bar{u} + \phi\,(u_t - \bar{u})}_{\text{conditional mean}} + \varepsilon_{t+1}, \qquad \varepsilon_{t+1} \sim N(0, \sigma^2).$$

Rearranging, the shock is just the gap between what happened and what the model expected:

$$\varepsilon_{t+1} = u_{t+1} - (\bar{u} + \phi\,(u_t - \bar{u})).$$

The model makes a strong and testable claim about these shocks: they are independent draws from a *symmetric* normal distribution.

If that claim holds, the shocks we recover from the data should look like a bell curve.

If it fails, the way it fails will tell us what the model is missing.

132.7.2 Recovering the shocks

We cannot read the shocks off directly, because we do not know the parameters $\bar{u}$ and ϕ.

So we estimate them, plugging a single representative value in for each.

We use the posterior medians — a reasonable choice for a quick diagnostic.

```
med = {k: np.median(np.asarray(mcmc_monthly.get_samples()[k]))
       for k in ("ubar", "phi", "sigma")}
resid = u_monthly[1:] - (med["ubar"] + med["phi"] * (u_monthly[:-1] - med["ubar"]))
```

The `resid` array holds our estimated shocks, one for each month-to-month transition — the model's **residuals**.

The slicing is what lines up each month with the one before it.

Each entry of `resid` is $u_{t+1} - (\hat{\bar{u}} + \hat{\phi}\,(u_t - \hat{\bar{u}}))$, the model's one-step-ahead forecast error, with hats denoting the median estimates.

(Because our estimate $\hat{\phi}$ is so close to one, this is nearly just the monthly change $u_{t+1} - u_t$.)

132.7.3 Comparing with the Gaussian

Now we ask whether these residuals look Gaussian.

We overlay a normal density whose standard deviation we set equal to that of the residuals themselves.

This is deliberate: the residuals already have mean near zero, so matching the variance makes the mean and the spread agree.

Anything left over is then a difference in *shape* — which is what we want to isolate.

One way to measure the shape is the **skewness**, the third standardized moment:

$$\text{skew} = \frac{\frac{1}{n}\sum_i (\varepsilon_i - \bar{\varepsilon})^3}{\left(\frac{1}{n}\sum_i (\varepsilon_i - \bar{\varepsilon})^2\right)^{3/2}}.$$

This measure is zero for any symmetric distribution and positive when the right tail is the longer one.

We now plot the residuals and compute their skewness:

```
def skewness(x):
    x = x - x.mean()
    return (x**3).mean() / x.std()**3

fig, ax = plt.subplots()
ax.hist(resid, bins=60, density=True, alpha=0.6, label='residuals')
grid = np.linspace(resid.min(), resid.max(), 200)
gauss = np.exp(-grid**2 / (2 * resid.std()**2)) / (resid.std() * np.sqrt(2 * np.pi))
ax.plot(grid, gauss, 'C1', lw=2, label='symmetric Gaussian')
ax.set_xlabel('one-month change not explained by the model')
ax.legend()
plt.show()

print(f"residual skewness = {skewness(resid):.2f}")
```

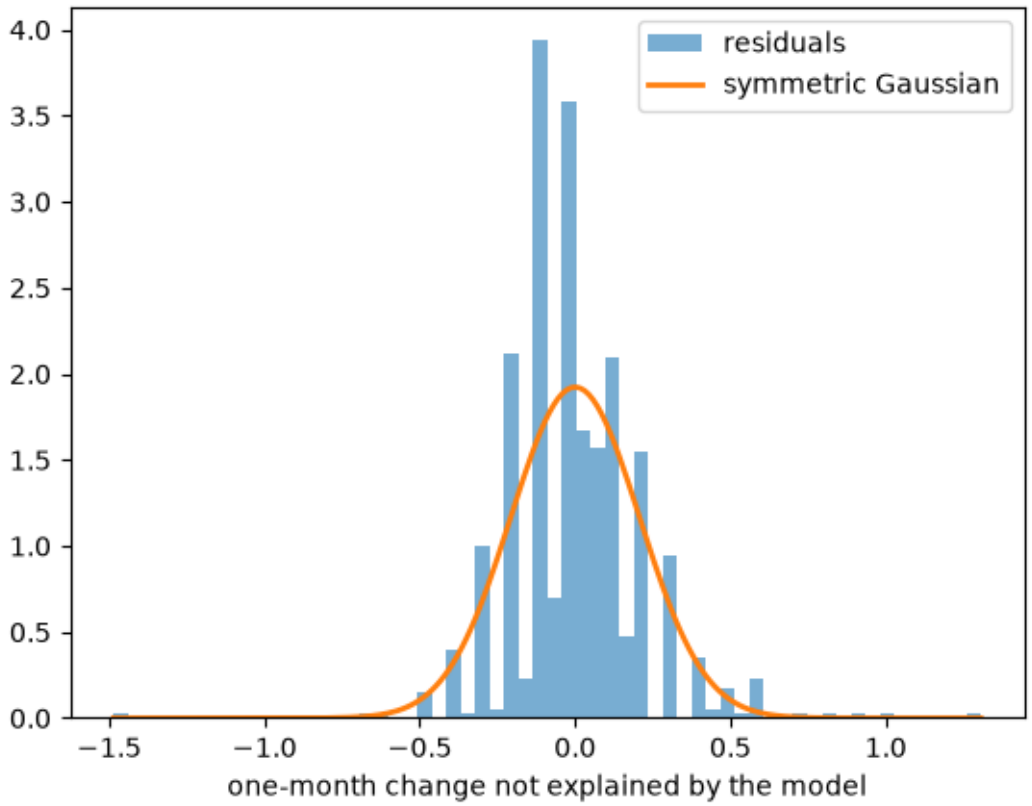


Fig. 132.4: Model residuals are right-skewed

```
residual skewness = 0.39
```

The residuals in Fig. 132.4 depart from the Gaussian in two ways.

They are *heavy-tailed* and *sharply peaked*: more tiny changes, and more large ones, than a bell curve allows.

Moreover, they are *right-skewed*, with the largest surprises on the upside (recessions).

The symmetric Gaussian (orange) can match neither feature: it treats upward and downward shocks as equally likely and has no room for the occasional very large jump.

So our model is bound to misread the data, seeing the rare jump up and the long gentle slide down as the *same* kind of

shock.

132.7.4 A note on the plug-in

A Bayesian purist would object that there is no single residual series here.

Every posterior draw of $(\bar{u}, \phi)$ implies its own, and we simply chose the medians.

That is fair, and the fully Bayesian version of this check — simulating whole datasets from the posterior and comparing a summary statistic — is what we do in *Unemployment Dynamics with Asymmetric Shocks*.

This plug-in check is a quick preview of that fuller test.

Capturing the asymmetry is the task of that next lecture, where we model the shocks themselves rather than the reversion curve.

132.8 Exercises

The lecture *Forecasting an AR(1) Process* forecasts an AR(1) process by simulating future paths, considering both a predictive distribution that **conditions on** fixed parameter values and one that **integrates over** their posterior uncertainty.

The following exercises apply these ideas to our fitted unemployment model.

Exercise 132.8.1

Using the fitted **annual** model, forecast unemployment over the next $H = 15$ years, starting from the last observed value, in two ways:

1. **plug-in**: hold the parameters at their posterior medians and simulate many future paths;
2. **extended**: for each future path, draw a fresh $(\bar{u}, \phi, \sigma)$ from the posterior.

Plot the 90% predictive band for each on the same axes and compare.

Which band is wider, and why?

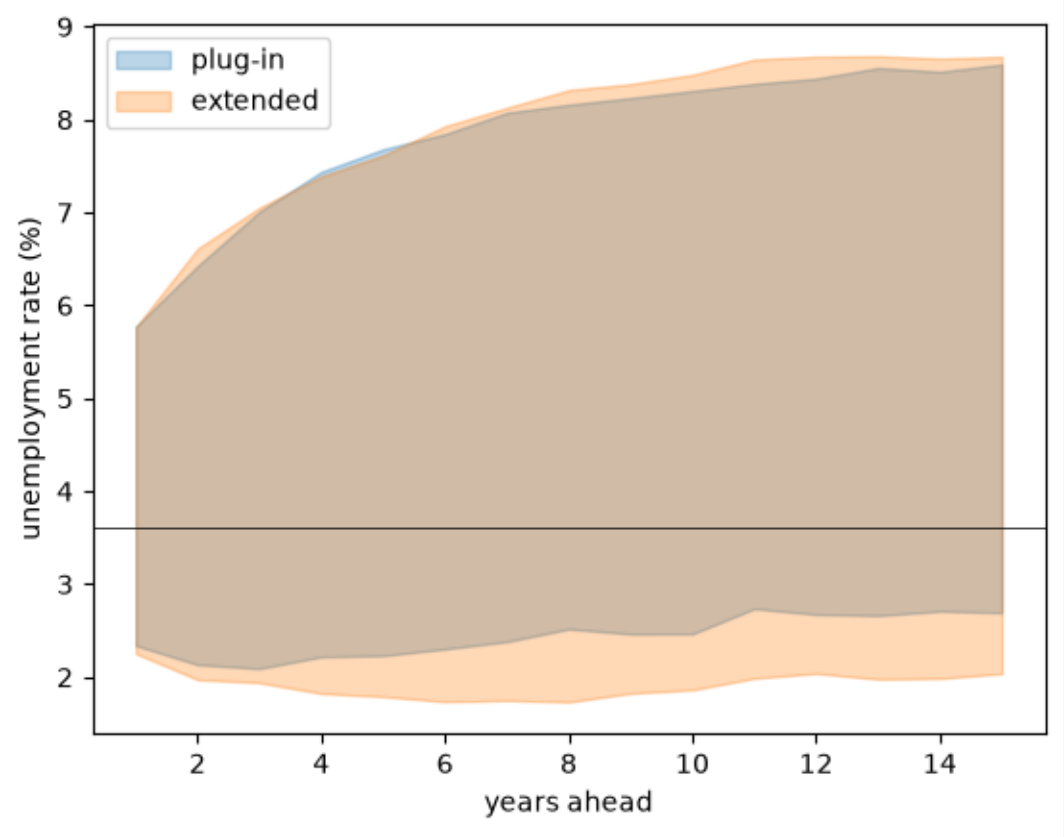
Solution

We use the annual posterior draws and simulate forward from the last observation.

We work with the annual model because its persistence ϕ is far less certain than at the monthly frequency, so parameter uncertainty has more to say.

```
post = mcmc_annual.get_samples()
ubar_s = np.asarray(post["ubar"])
ψ_s = np.asarray(post["phi"])
σ_s = np.asarray(post["sigma"])

def sim_future(u_last, ubar, ψ, σ, H, rng):
    "Simulate H steps of the linear model forward from u_last."
    u = np.empty(H)
    prev = u_last
    for h in range(H):
        prev = ubar + ψ * (prev - ubar) + rng.normal(0, σ)
        u[h] = prev
    return u
```

```
H, N = 15, 2000
u_last = u_annual[-1]
rng = np.random.default_rng(0)

# plug-in: parameters fixed at posterior medians
ub0, ψ0, σ0 = np.median(ubar_s), np.median(ψ_s), np.median(σ_s)
plug = np.array([sim_future(u_last, ub0, ψ0, σ0, H, rng) for _ in range(N)])

# extended: a fresh posterior draw for each path
idx = rng.integers(0, len(ψ_s), N)
ext = np.array([sim_future(u_last, ubar_s[i], ψ_s[i], σ_s[i], H, rng)
                for i in idx])

fig, ax = plt.subplots()
horizon = np.arange(1, H + 1)
for data, c, lab in [(plug, 'C0', 'plug-in'), (ext, 'C1', 'extended')]:
    lo, hi = np.percentile(data, [5, 95], axis=0)
    ax.fill_between(horizon, lo, hi, alpha=0.3, color=c, label=lab)
ax.axhline(u_last, color='k', lw=0.5)
ax.set_xlabel('years ahead')
ax.set_ylabel('unemployment rate (%)')
ax.legend()
plt.show()
```



Both forecasts drift up from the low starting value toward the reversion level $\bar{u}$, with bands that widen as we look further ahead.

The extended band is clearly wider, because it adds uncertainty about the parameters on top of the uncertainty from

future shocks — and here the persistence ϕ and the level $\bar{u}$ are both genuinely uncertain.

Exercise 132.8.2

Following Wecker (see *Forecasting an AR(1) Process*), we can also form a predictive distribution for a **path statistic** — a nonlinear function of the whole future path.

Take the **maximum unemployment rate over the next** $H = 8$ **years** as the statistic: a simple measure of how bad the next several years might get.

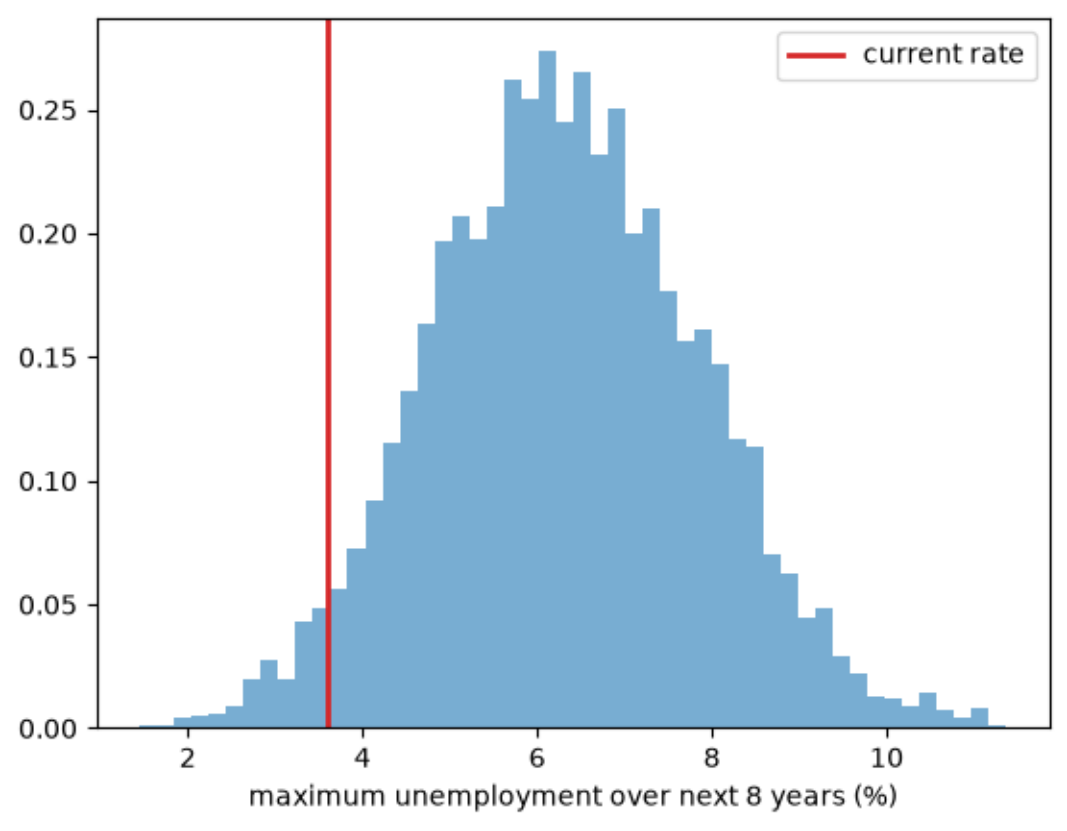
Using the extended simulation (a posterior draw per path), compute this maximum for each path and plot its predictive distribution.

What is the posterior predictive probability that unemployment reaches at least 7% — recession territory — at some point over the next eight years?

Solution

We reuse `sim_future` and the posterior draws from the previous exercise.

```
H2, M = 8, 5000
idx = rng.integers(0, len(ψ_s), M)
peak = np.array([sim_future(u_last, ubar_s[i], ψ_s[i], σ_s[i], H2, rng).max()
                 for i in idx])

fig, ax = plt.subplots()
ax.hist(peak, bins=50, density=True, alpha=0.6)
ax.axvline(u_last, color='C3', lw=2, label='current rate')
ax.set_xlabel('maximum unemployment over next 8 years (%)')
ax.legend()
plt.show()

prob = (peak > 7.0).mean()
print(f"P(unemployment reaches 7% within 8 years) = {prob:.2f}")
```




```
P(unemployment reaches 7% within 8 years) = 0.32
```

The predictive distribution summarizes the plausible “worst case” over the next several years, integrating over both shocks and parameter uncertainty.

Starting from a cyclical low, the model sees a substantial chance of a return to recession-level unemployment within the horizon.

CHAPTER

THREE

UNEMPLOYMENT DYNAMICS WITH ASYMMETRIC SHOCKS

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

This lecture is a sequel to *A Linear Model of Unemployment*.

There we fit a linear AR(1) model to the US unemployment rate and examined persistence.

One thing we found but didn't address is that the model's residuals were both *heavy-tailed and right-skewed*.

Unemployment jumps up sharply in recessions and drifts down slowly in recoveries.

A model with symmetric Gaussian shocks cannot reproduce these features.

In this lecture we allow the innovations to be right-skewed and, to some degree, heavy-tailed.

We also use this lecture to develop a very useful Bayesian technique: *model comparison* by leave-one-out cross-validation.

We use this to decide whether one model really predicts better than another.

Our plan is to

1. build a model with a linear mean but asymmetric, occasionally-large shocks,
2. estimate it on the annual data, and
3. compare it to the original Gaussian model with cross-validation.

In addition to what's in Anaconda, this lecture needs the following libraries.

```
!pip install numpyro jax pandas_datareader arviz
```

We'll use the following imports.

```
import numpy as np
import matplotlib.pyplot as plt
import datetime as dt
from pandas_datareader import data as web

import jax.numpy as jnp
```

(continues on next page)

(continued from previous page)

```
from jax import random
import numpyro
import numpyro.distributions as dist
from numpyro.infer import MCMC, NUTS, log_likelihood
```

133.1 The data

We use the same series as in *A Linear Model of Unemployment* — the US unemployment rate (`UNRATE` from FRED), excluding the COVID-19 spike — and work at an annual frequency.

```
start, end = dt.datetime(1948, 1, 1), dt.datetime(2024, 12, 31)
unrate = web.DataReader("UNRATE", "fred", start, end)["UNRATE"]
pre_covid = unrate[unrate.index < "2020-01-01"]
u_annual = pre_covid.resample("YE").last().to_numpy()
years = pre_covid.resample("YE").last().index.year
```

The shape we want to capture is plain in the annual series.

```
fig, ax = plt.subplots()
ax.plot(years, u_annual, lw=2, marker='o', ms=3)
ax.set_xlabel('year')
ax.set_ylabel('unemployment rate (%)')
plt.show()
```

In Fig. 133.1 each recession is a sharp rise followed by a long, gradual decline — fast up, slow down.

133.2 A model with asymmetric shocks

We keep the linear structure from *A Linear Model of Unemployment*:

$$u_{t+1} = \bar{u} + \rho\,(u_t - \bar{u}) + \eta_{t+1}, \qquad 0 \le \rho < 1, \tag{133.1}$$

Here, however, the innovation η_{t+1} is drawn from a *mixture of two normals*:

$$\eta_{t+1} \sim \begin{cases} N(0, \sigma_s^2) & \text{with probability } 1-p \quad \text{(a quiet year)}, \\ N(\mu_J, \sigma_J^2) & \text{with probability } p \quad \text{(a recession jump)}, \end{cases} \qquad \mu_J > 0.$$

The model has six parameters:

symbol	name	role
$\bar{u}$	floor	the low level the series reverts to
ρ	persistence	speed of reversion (as in the linear lecture)
p	jump probability	how often a large shock arrives
μ_J	jump size	the average upward kick of a recession
σ_s, σ_J	shock spreads	the quiet and jumpy volatilities

The mechanism leads to a sawtooth dynamic for the unemployment rate.

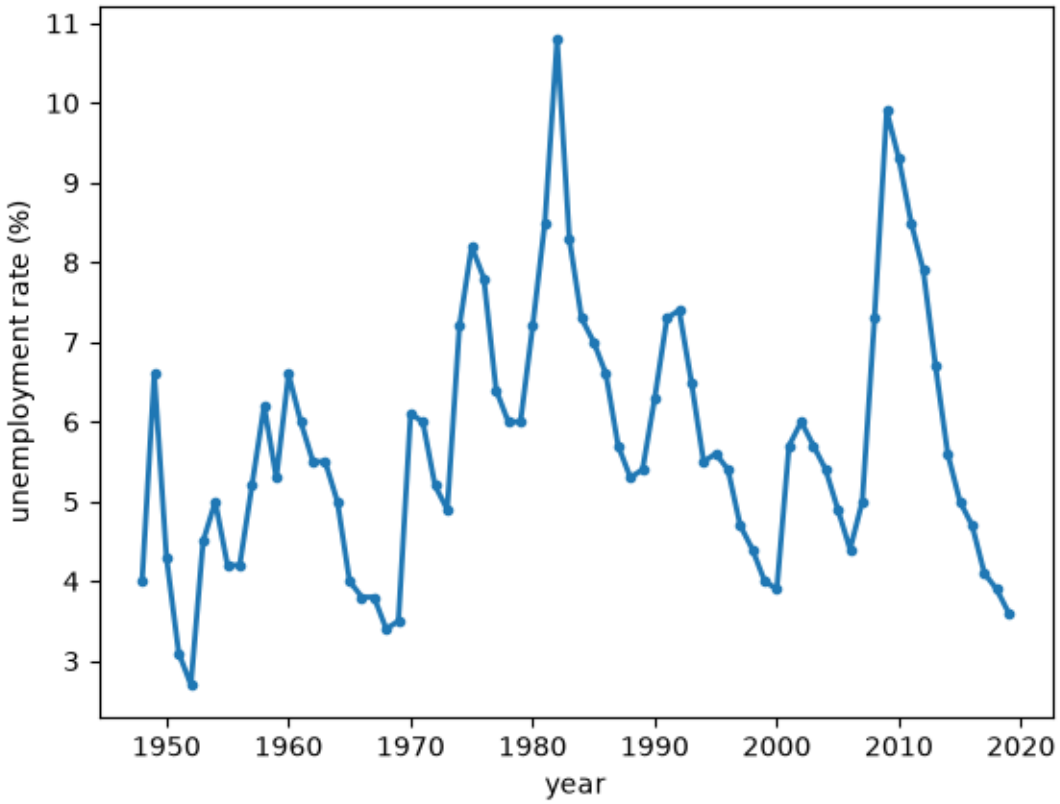


Fig. 133.1: Annual US unemployment

Most years are quiet, with small noise.

Occasionally a positive jump pushes unemployment up by a large amount.

Then, in most cases, the linear reversion pulls it slowly back down toward $\bar{u}$.

Thus, a spike arrives in one step, while the recovery takes time — the asymmetric dynamics we observe in the data.

This is, in spirit, Milton Friedman's "plucking model": a floor near full employment, from which bad shocks pluck the series upward.

The next cell shows the density of the shock and its two components.

```
def normal_pdf(x, m, s):
    return np.exp(-(x - m)**2 / (2 * s**2)) / (s * np.sqrt(2 * np.pi))

η = np.linspace(-3, 6, 400)
p_, μJ_, σs_, σJ_ = 0.25, 2.0, 0.4, 1.2
quiet = (1 - p_) * normal_pdf(η, 0.0, σs_)
jump  = p_ * normal_pdf(η, μJ_, σJ_)

fig, ax = plt.subplots()
ax.plot(η, quiet + jump, lw=2, label='shock density')
ax.fill_between(η, quiet, alpha=0.3, label='quiet years')
ax.fill_between(η, jump, alpha=0.3, label='recession jumps')
ax.set_xlabel('innovation $\\eta$')
ax.legend()
plt.show()
```

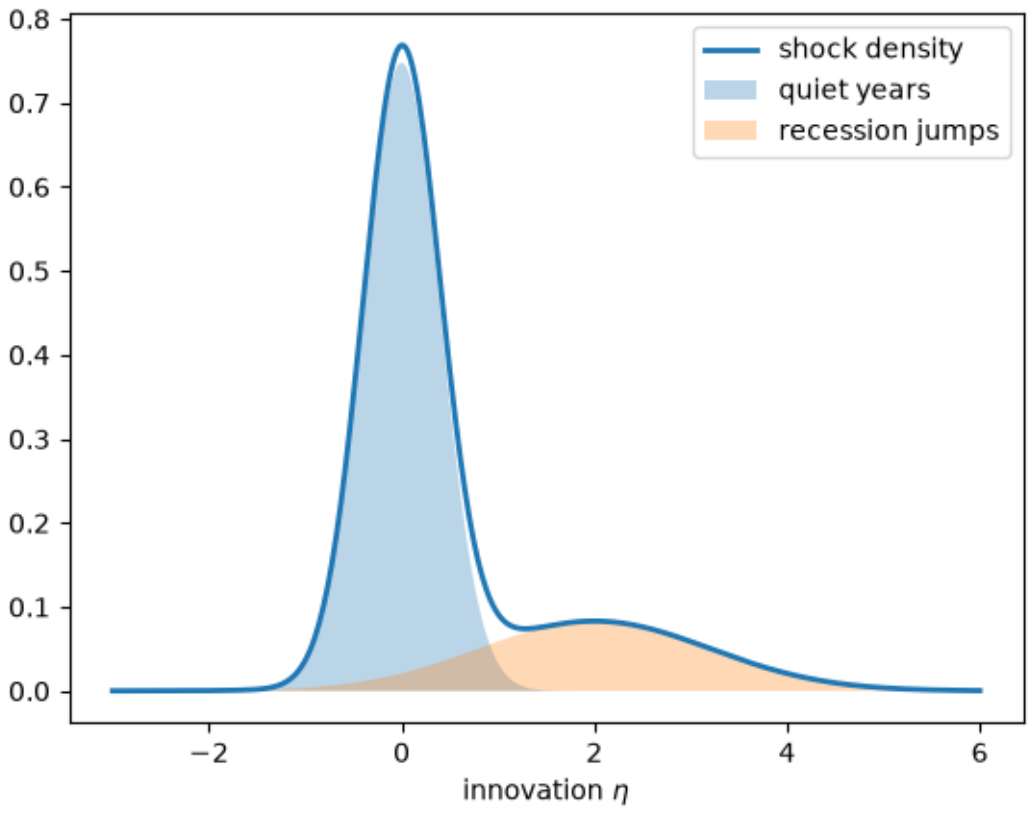


Fig. 133.2: Shock distribution as a mixture

Fig. 133.2 shows a tall, narrow peak at zero for ordinary years, plus a low, broad bump on the positive side for the recession jumps.

133.3 Bayesian estimation

We place weakly informative priors on the six parameters.

The mixture is written with NumPyro's `MixtureSameFamily`, which sums over the two components analytically.

```
def jump_model(u):
    # Set the priors
    ubar = numpyro.sample("ubar",    dist.Normal(4.5, 1.5))
    ρ    = numpyro.sample("rho",     dist.Uniform(0.0, 1.0))
    p    = numpyro.sample("p",       dist.Beta(2.0, 8.0))
    μ_J  = numpyro.sample("mu_J",    dist.HalfNormal(2.0))
    σ_s  = numpyro.sample("sigma_s", dist.HalfNormal(0.5))
    σ_J  = numpyro.sample("sigma_J", dist.HalfNormal(1.5))
    # Build the model
    n = u.shape[0] - 1
    base   = ubar + ρ * (u[:-1] - ubar)
    locs   = jnp.stack([base, base + μ_J], axis=-1)
    scales = jnp.stack([jnp.broadcast_to(σ_s, (n,)),
                        jnp.broadcast_to(σ_J, (n,))], axis=-1)
    probs  = jnp.broadcast_to(jnp.stack([1 - p, p]), (n, 2))
    mix = dist.MixtureSameFamily(dist.Categorical(probs=probs),
                                 dist.Normal(locs, scales))
    numpyro.sample("u_obs", mix, obs=u[1:])
```

We sample with NUTS, four chains, vectorized so the code runs on a CPU or a GPU.

```
def run_nuts(model, data, seed=0, num_warmup=2000, num_samples=4000, num_chains=4):
    "Sample a NumPyro model with the NUTS sampler."
    mcmc = MCMC(NUTS(model),
                num_warmup=num_warmup, num_samples=num_samples,
                num_chains=num_chains, chain_method="vectorized",
                progress_bar=False)
    mcmc.run(random.PRNGKey(seed), jnp.asarray(data))
    return mcmc
```

We fit the model to the annual data.

```
mcmc_jump = run_nuts(jump_model, u_annual)
```

Let us look at the posterior.

```
mcmc_jump.print_summary()
```

```
                mean       std    median      5.0%     95.0%     n_eff     r_hat
      mu_J      1.26      0.43      1.20      0.57      2.00   3583.57      1.00
         p      0.35      0.09      0.35      0.19      0.49   4178.84      1.00
       rho      0.83      0.04      0.84      0.76      0.90   5632.75      1.00
   sigma_J      1.28      0.26      1.29      0.83      1.73   3970.18      1.00
   sigma_s      0.39      0.09      0.38      0.25      0.54   3715.71      1.00
      ubar      3.03      0.74      3.09      1.81      4.19   5156.56      1.00
```

(continues on next page)

(continued from previous page)

```
Number of divergences: 0
```

The `r_hat` values are essentially one, so the chains have converged.

Notice the distinct jump component: the normal year spread σ_s is small and the jump spread σ_J is large, with a positive mean μ_J.

Let's do a quick check by overlaying the fitted shock density on the model's actual residuals, as we did in *A Linear Model of Unemployment*.

```
m = {k: np.median(np.asarray(mcmc_jump.get_samples()[k]))
     for k in ("ubar", "rho", "p", "mu_J", "sigma_s", "sigma_J")}
resid = u_annual[1:] - (m["ubar"] + m["rho"] * (u_annual[:-1] - m["ubar"]))

η = np.linspace(resid.min() - 0.5, resid.max() + 0.5, 400)
quiet = (1 - m["p"]) * normal_pdf(η, 0.0, m["sigma_s"])
jump  = m["p"] * normal_pdf(η, m["mu_J"], m["sigma_J"])

fig, ax = plt.subplots()
ax.hist(resid, bins=25, density=True, alpha=0.5, label='residuals')
ax.plot(η, quiet + jump, lw=2, label='fitted density')
ax.fill_between(η, jump, alpha=0.3, color='C2', label='jump component')
ax.set_xlabel('innovation $\\eta$')
ax.legend()
plt.show()
```

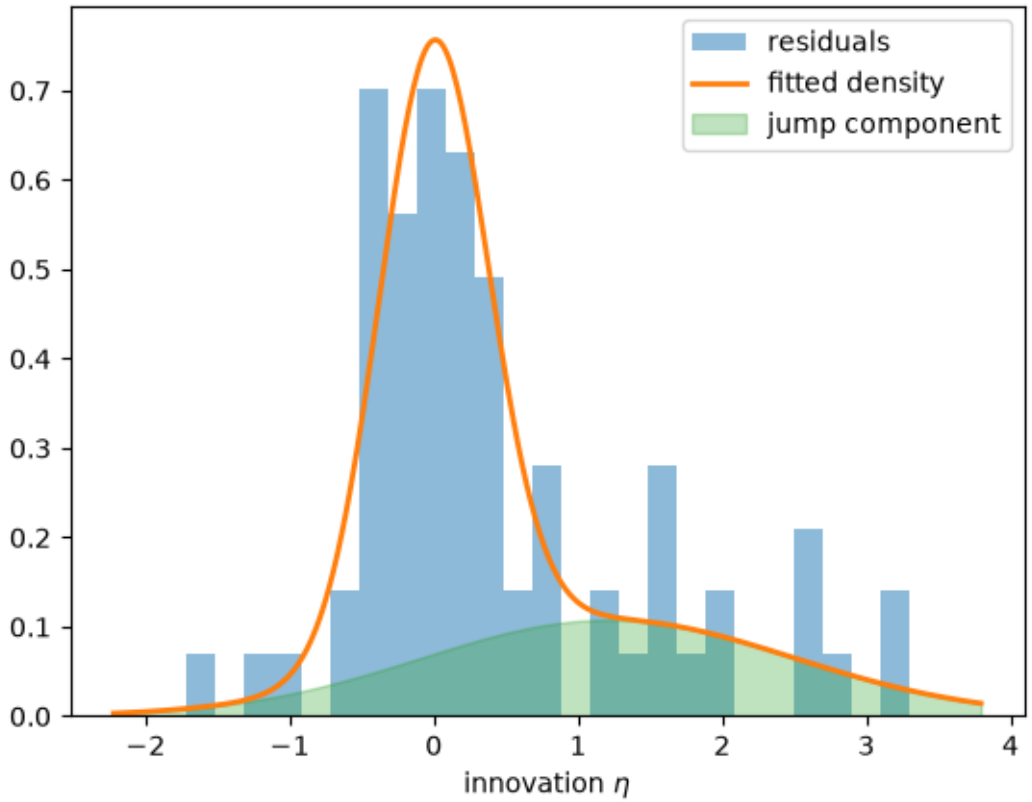


Fig. 133.3: Fitted shock density against the residuals

The fit looks quite good (although we are still struggling to capture some of the large shocks in the right tail).

133.4 Comparing models with cross-validation

The jump model looks reasonable, but is it actually better than the linear one?

To answer this question, we will use a form of *Bayesian model comparison*.

We build it up in stages: the guiding principle, the leave-one-out estimate, and how the computation is actually done.

133.4.1 The guiding principle: out-of-sample predictions

It is tempting to compare two models by asking which one fits the observed data better.

But this isn't the right criterion: more complex models can always be tuned to better match the data they are fitting.

The right question is instead: which model better predicts data it has not seen?

In other words, the criterion we care about is out-of-sample predictive accuracy.

We do not have new data for out-of-sample testing, so we manufacture some using *cross-validation*.

133.4.2 Leave-one-out cross-validation

The idea is to leave out one observation, fit the model to the rest, and then score the held-out point using a predictive distribution constructed from the rest of the data set.

The predictive distribution for data point u_i given the rest of the data u_{-i} is

$$p(u_i \mid u_{-i}) = \int p(u_i \mid \theta)\, p(\theta \mid u_{-i})\, d\theta.$$

Here $p(\theta \mid u_{-i})$ is the posterior obtained from the data in u_{-i}.

We score each held-out point by the *log* of this density.

This rewards a model for putting high probability on what actually happened.

Adding these scores over all data points gives the **expected log predictive density** (elpd), here in its leave-one-out (LOO) form,

$$\text{elpd}_{\text{loo}} = \sum_{i=1}^{n} \log p(u_i \mid u_{-i}).$$

133.4.3 Computational methods

The definition of elpd suggests that we need to refit the model n times, once for each omitted point.

This would be very time-consuming.

Here we discuss a method that removes all but the first fit, using the full posterior — which we have already sampled.

Regarding notation: let $u = (u_1, \ldots, u_n)$, and write u_{-i} for the sample with u_i deleted.

The method depends on the following independence-type assumption:

Assumption 133.4.1

Given θ, the observations are conditionally independent:

$$p(u \mid \theta) = \prod_{j=1}^{n} p(u_j \mid \theta).$$

For our autoregressive data this assumption is false, since each u_i is dependent on earlier observations through the transition dynamics.

We adopt it nonetheless, since

- it's what makes the leave-one-out computation tractable and
- it's a very standard procedure that needs to be understood.

We discuss the consequences in the section on time-series structure below.

Conditional on our assumption, the proposition that follows is exact.

Proposition 133.4.1

Under *Assumption 133.4.1*, we have

$$\frac{1}{p(u_i \mid u_{-i})} = \int \frac{1}{p(u_i \mid \theta)} p(\theta \mid u)\, d\theta \qquad (i = 1, \dots, n).$$

Proof. By Bayes' rule, $p(\theta \mid u) = p(u \mid \theta)\, p(\theta)/p(u)$ and $p(\theta \mid u_{-i}) = p(u_{-i} \mid \theta)\, p(\theta)/p(u_{-i})$.

By *Assumption 133.4.1*, $p(u \mid \theta) = p(u_i \mid \theta)\, p(u_{-i} \mid \theta)$.

Dividing the two posteriors and substituting this factorization,

$$\frac{p(\theta \mid u)}{p(\theta \mid u_{-i})} = \frac{p(u \mid \theta)}{p(u_{-i} \mid \theta)} \cdot \frac{p(u_{-i})}{p(u)} = p(u_i \mid \theta) \cdot \frac{p(u_{-i})}{p(u)},$$

so, rearranging,

$$\frac{p(\theta \mid u)}{p(u_i \mid \theta)} = \frac{p(u_{-i})}{p(u)} p(\theta \mid u_{-i}).$$

Now integrate both sides over θ.

On the right, $\int p(\theta \mid u_{-i})\, d\theta = 1$, leaving the constant factor $p(u_{-i})/p(u)$, so

$$\int \frac{1}{p(u_i \mid \theta)} p(\theta \mid u)\, d\theta = \frac{p(u_{-i})}{p(u)} = \frac{1}{p(u_i \mid u_{-i})},$$

the last equality by $p(u) = p(u_i \mid u_{-i})\, p(u_{-i})$.

The leave-i-out posterior has vanished; only the full posterior remains, and of it we have S draws $\theta^1, \dots, \theta^S$.

Replacing the integral in *Proposition 133.4.1* by the average over these draws gives

$$\frac{1}{p(u_i \mid u_{-i})} \;\approx\; \frac{1}{S}\sum_{s=1}^{S}\frac{1}{p(u_i \mid \theta^s)}, \qquad \text{so} \qquad p(u_i \mid u_{-i}) \;\approx\; \frac{S}{\sum_{s=1}^{S} 1/p(u_i \mid \theta^s)},$$

the harmonic mean of the per-draw likelihoods of u_i.

The integrand has finite mean, equal to $1/p(u_i \mid u_{-i})$ itself, so the law of large numbers applies.

The average therefore converges to that mean as $S \to \infty$, and the approximation becomes exact, with probability one, in the limit.

Its one ingredient is the likelihood of each observation under each draw — the **pointwise log-likelihood** — which is why we have been careful to keep it.

For numerical stability we compute on the log scale, where the harmonic mean becomes

$$\log p(u_i \mid u_{-i}) \;\approx\; \log S - \log \sum_{s=1}^{S} e^{-\ell_i^s}, \qquad \ell_i^s = \log p(u_i \mid \theta^s),$$

which is the `jnp.log(S) - logsumexp(-ll)` we use below.

To compare against the linear model, we first fit it on the same data.

```
def linear_model(u):
    ubar = numpyro.sample("ubar",  dist.Normal(5.5, 2.0))
    ρ    = numpyro.sample("rho",   dist.Uniform(0.0, 1.0))
    σ    = numpyro.sample("sigma", dist.HalfNormal(1.0))
    numpyro.sample("u_obs", dist.Normal(ubar + ρ * (u[:-1] - ubar), σ), obs=u[1:])
```

We sample it on the same annual data.

```
mcmc_lin = run_nuts(linear_model, u_annual)
```

Now we can do the whole leave-one-out calculation by hand in a few lines.

```
from jax.scipy.special import logsumexp

def pointwise_loo(mcmc, model):
    "Leave-one-out log predictive density for each observation."
    ll = log_likelihood(model, mcmc.get_samples(),
                        u=jnp.asarray(u_annual))["u_obs"]  # (draws, obs)
    S = ll.shape[0]
    # log of the harmonic mean of the per-draw likelihoods, per observation
    return jnp.log(S) - logsumexp(-ll, axis=0)

elpd_jump = pointwise_loo(mcmc_jump, jump_model)
elpd_lin  = pointwise_loo(mcmc_lin,  linear_model)

print(f"jump   elpd_loo = {elpd_jump.sum():.1f}")
print(f"linear elpd_loo = {elpd_lin.sum():.1f}")
```

```
jump   elpd_loo = -93.9
linear elpd_loo = -105.7
```

The jump model scores higher (closer to zero), so on out-of-sample prediction it wins.

To help understand whether the gap is real, we compute its standard error:

```
diff = elpd_jump - elpd_lin
n = diff.size
print(f"elpd difference = {diff.sum():.1f}")
print(f"standard error  = {jnp.sqrt(n) * diff.std():.1f}")
```

```
elpd difference = 11.8
standard error  = 5.7
```

The jump model is ahead by about twelve points, against a standard error near six — a bit over two standard errors, so a real improvement.

133.4.4 ArviZ implementation

In practice, most people do this calculation with a library like ArviZ.

We hand it the same pointwise log-likelihoods, packaged in its data format.

```
import arviz as az
import xarray as xr

def to_arviz(mcmc, model, u):
    "Package a NumPyro fit for ArviZ, with pointwise log-likelihoods."
    idata = az.from_numpyro(mcmc)
    ll = np.asarray(log_likelihood(model, mcmc.get_samples(),
                                   u=jnp.asarray(u))["u_obs"])
    ll = ll.reshape(mcmc.num_chains, -1, ll.shape[-1])
    idata["log_likelihood"] = xr.Dataset(
        {"u_obs": (("chain", "draw", "obs"), ll)})
    return idata

az.compare({
    "linear": to_arviz(mcmc_lin,  linear_model, u_annual),
    "jump":   to_arviz(mcmc_jump, jump_model,   u_annual),
})
```

```
        rank  elpd_diff  dse  p_worse diag_diff diag_elpd    p   elpd   se  \
jump       0        0.0  0.0      NaN                      6.5  -90.0  9.2
linear     1      -10.0  5.7     0.98   N < 100            3.6 -110.0  7.3

        weight
jump      0.87
linear    0.13
```

Here `az.compare` confirms the hand calculation and ranks the jump model first (its table rounds the scores for display).

133.4.5 Accommodating the time series structure

Leave-one-out drops one transition at a time, but it does not respect the order of time.

When it scores the step from u_t to u_{t+1}, the model doing the scoring was fit on data lying on *both* sides of that step — including the neighboring values u_t and u_{t+1} themselves.

So the held-out point was never truly unseen.

The formally correct measure is **leave-future-out** cross-validation.

It only ever predicts forward: fit the model on $u_1, \ldots, u_t$, score its one-step-ahead forecast of u_{t+1}, then expand the window by one step and repeat.

Now the model is genuinely blind to the future it is asked to predict, exactly as in real forecasting.

The price is computational.

Leave-one-out reused a single fit through its importance-sampling shortcut, but leave-future-out has no such trick.

For our short annual series leave-future-out cross-validation is still feasible but for a long series it can become prohibitive.

(We ran it without including it here and the conclusion holds: under leave-future-out the jump model still beats the linear one, by an even larger margin.)

133.5 Conclusion

In *A Linear Model of Unemployment* we applied a linear AR(1) model with Gaussian shocks to unemployment and found high levels of persistence in monthly data.

We also argued that the model is overly simplistic.

Here we found that the feature the linear model most conspicuously misses — the asymmetry of recessions — can be addressed by modeling the distribution of the shocks.

A linear reversion with large, one-sided innovations matches the right-skewed, heavy-tailed shocks that the Gaussian model could not.

Cross-validation prefers it.

An even better model than the one considered above would let the economy switch between persistent expansion and recession regimes.

This is the Markov-switching approach, which we leave to further reading.

133.6 Exercises

Exercise 133.6.1

The jump component here is symmetric within itself, $N(\mu_J, \sigma_J^2)$.

Replace the two-component mixture with a single **skew-normal** (or Student-t with a skew) innovation, refit, and compare to the mixture model with `az.compare`.

Does the simpler skewed shock do as well as the mixture?

Part XVIII

Auctions

CHAPTER

FOUR

FIRST-PRICE AND SECOND-PRICE AUCTIONS

This lecture is designed to set the stage for a subsequent lecture about Multiple Good Allocation Mechanisms

In that lecture, a planner or auctioneer simultaneously allocates several goods to set of people.

In the present lecture, a single good is allocated to one person within a set of people.

Here we'll learn about and simulate two classic auctions :

- a First-Price Sealed-Bid Auction (FPSB)
- a Second-Price Sealed-Bid Auction (SPSB) created by William Vickrey [Vickrey, 1961]

We'll also learn about and apply a

- Revenue Equivalent Theorem

We recommend watching this video about second price auctions by Anders Munk-Nielsen:

https://youtu.be/qwWk_Bqtue8

and

https://youtu.be/eYTGQCGpmXI

Anders Munk-Nielsen put his code on GitHub.

Much of our Python code below is based on his.

134.1 First-price sealed-bid auction (FPSB)

Protocols:

- A single good is auctioned.
- Prospective buyers simultaneously submit sealed bids.
- Each bidder knows only his/her own bid.
- The good is allocated to the person who submits the highest bid.
- The winning bidder pays price she has bid.

Detailed Setting:

There are $n > 2$ prospective buyers named $i = 1, 2, \ldots, n$.

Buyer i attaches value v_i to the good being sold.

Buyer i wants to maximize the expected value of her **surplus** defined as $v_i - p$, where p is the price that she pays, conditional on her winning the auction.

Evidently,

- If i bids exactly v_i, she pays what she thinks it is worth and gathers no surplus value.
- Buyer i will never want to bid more than v_i.
- If buyer i bids $b < v_i$ and wins the auction, then she gathers surplus value $v_i - b > 0$.
- If buyer i bids $b < v_i$ and someone else bids more than b, buyer i loses the auction and gets no surplus value.
- To proceed, buyer i wants to know the probability that she wins the auction as a function of her bid v_i
  - this requires that she know a probability distribution of bids v_j made by prospective buyers $j \neq i$
- Given her idea about that probability distribution, buyer i wants to set a bid that maximizes the mathematical expectation of her surplus value.

Bids are sealed, so no bidder knows bids submitted by other prospective buyers.

This means that bidders are in effect participating in a game in which players do not know **payoffs** of other players.

This is a **Bayesian game**, a Nash equilibrium of which is called a **Bayesian Nash equilibrium**.

To complete the specification of the situation, we'll assume that prospective buyers' valuations are independently and identically distributed according to a probability distribution that is known by all bidders.

Bidder optimally chooses to bid less than v_i.

134.1.1 Characterization of FPSB auction

A FPSB auction has a unique symmetric Bayesian Nash Equilibrium.

The optimal bid of buyer i is

$$\mathbf{E}[y_i | y_i < v_i] \tag{134.1}$$

where v_i is the valuation of bidder i and y_i is the maximum valuation of all other bidders:

$$y_i = \max_{j \neq i} v_j \tag{134.2}$$

A proof for this assertion is available at the Wikipedia page about Vickrey auctions

134.2 Second-price sealed-bid auction (SPSB)

Protocols: In a second-price sealed-bid (SPSB) auction, the winner pays the second-highest bid.

134.3 Characterization of SPSB auction

In a SPSB auction bidders optimally choose to bid their values.

Formally, a dominant strategy profile in a SPSB auction with a single, indivisible item has each bidder bidding its value.

A proof is provided at the Wikipedia page about Vickrey auctions

134.4 Uniform distribution of private values

We assume valuation v_i of bidder i is distributed $v_i \overset{\text{i.i.d.}}{\sim} U(0,1)$.

Under this assumption, we can analytically compute probability distributions of prices bid in both FPSB and SPSB.

We'll simulate outcomes and, by using a law of large numbers, verify that the simulated outcomes agree with analytical ones.

We can use our simulation to illustrate a **Revenue Equivalence Theorem** that asserts that on average first-price and second-price sealed bid auctions provide a seller the same revenue.

To read about the revenue equivalence theorem, see this Wikepedia page

134.5 Setup

There are n bidders.

Each bidder knows that there are $n-1$ other bidders.

134.6 First price sealed bid auction

An optimal bid for bidder i in a **FPSB** is described by equations (134.1) and (134.2).

When bids are i.i.d. draws from a uniform distribution, the CDF of y_i is

$$\begin{aligned}\tilde{F}_{n-1}(y) = \mathbf{P}(y_i \leq y) &= \mathbf{P}(\max_{j \neq i} v_j \leq y) \\ &= \prod_{j \neq i} \mathbf{P}(v_j \leq y) \\ &= y^{n-1}\end{aligned}$$

and the PDF of y_i is $\tilde{f}_{n-1}(y) = (n-1)y^{n-2}$.

Then bidder i's optimal bid in a **FPSB** auction is:

$$\begin{aligned}\mathbf{E}(y_i | y_i < v_i) &= \frac{\int_0^{v_i} y_i \tilde{f}_{n-1}(y_i) dy_i}{\int_0^{v_i} \tilde{f}_{n-1}(y_i) dy_i} \\ &= \frac{\int_0^{v_i} (n-1) y_i^{n-1} dy_i}{\int_0^{v_i} (n-1) y_i^{n-2} dy_i} \\ &= \frac{n-1}{n} y_i \Big|_0^{v_i} \\ &= \frac{n-1}{n} v_i\end{aligned}$$

134.7 Second price sealed bid auction

In a **SPSB**, it is optimal for bidder i to bid v_i.

134.8 Python code

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import scipy.stats as stats
import scipy.interpolate as interp

# for plots
plt.rcParams.update({"text.usetex": True, 'font.size': 14})
colors = plt.rcParams['axes.prop_cycle'].by_key()['color']

# ensure the notebook generates the same randomness
np.random.seed(1337)
```

We repeat an auction with 5 bidders for 100,000 times.

The valuations of each bidder is distributed $U(0, 1)$.

```
N = 5
R = 100_000

v = np.random.uniform(0, 1, (N, R))

# BNE in first-price sealed bid

b_star = lambda vi, N: ((N-1)/N) * vi
b = b_star(v,N)
```

We compute and sort bid price distributions that emerge under both FPSB and SPSB.

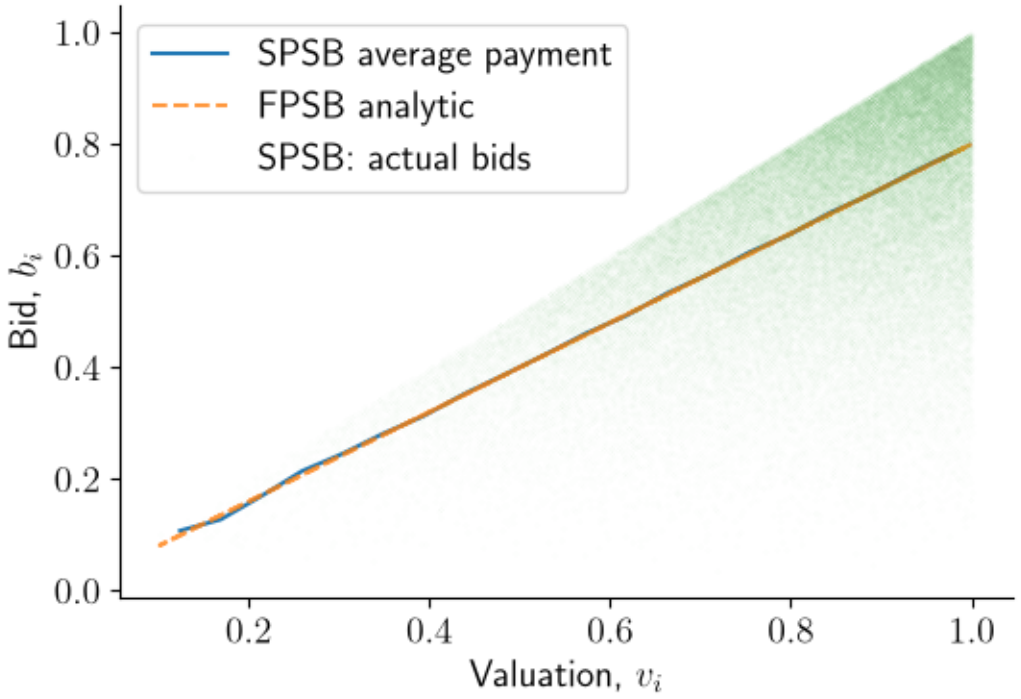
```
# Bidders' values are sorted in ascending order in each auction.
# We record the order because we want to apply it to bid price and their id.
idx = np.argsort(v, axis=0)

# same as np.sort(v, axis=0), except now we retain the idx
v = np.take_along_axis(v, idx, axis=0)
b = np.take_along_axis(b, idx, axis=0)

# the id for the bidders is created.
ii = np.repeat(np.arange(1, N+1)[:, None], R, axis=1)
# the id is sorted according to bid price as well.
ii = np.take_along_axis(ii, idx, axis=0)

# In FPSB and SPSB, winners are those with highest values.
winning_player = ii[-1, :]

# highest bid
winner_pays_fpsb = b[-1, :]
# 2nd-highest valuation
winner_pays_spsb = v[-2, :]
```

Let's now plot the *winning* bids $b_{(n)}$ (i.e. the payment) against valuations, $v_{(n)}$ for both FPSB and SPSB.

Note that

- FPSB: There is a unique bid corresponding to each valuation
- SPSB: Because it equals the valuation of a second-highest bidder, what a winner pays varies even holding fixed the winner's valuation. So here there is a frequency distribution of payments for each valuation.

```
# We intend to compute average payments of different groups of bidders
binned = stats.binned_statistic(v[-1, :], v[-2, :], statistic='mean', bins=20)
xx = binned.bin_edges
xx = [(xx[ii]+xx[ii+1])/2 for ii in range(len(xx)-1)]
yy = binned.statistic

fig, ax = plt.subplots(figsize=(6, 4))

ax.plot(xx, yy, label='SPSB average payment')
ax.plot(v[-1, :], b[-1, :], '--', alpha=0.8, label='FPSB analytic')
ax.plot(v[-1, :], v[-2, :], 'o', alpha=0.05,
                markersize=0.1, label='SPSB: actual bids')

ax.legend(loc='best')
ax.set_xlabel('Valuation, $v_i$')
ax.set_ylabel('Bid, $b_i$')
sns.despine()
```



134.9 Revenue equivalence theorem

We now compare FPSB and a SPSB auctions from the point of view of the revenues that a seller can expect to acquire.

Expected Revenue FPSB:

The winner with valuation y pays $\frac{n-1}{n} * y$, where n is the number of bidders.

Above we computed that the CDF is $F_n(y) = y^n$ and the PDF is $f_n = ny^{n-1}$.

Consequently, expected revenue is

$$\mathbf{R} = \int_0^1 \frac{n-1}{n} v_i \times n v_i^{n-1} dv_i = \frac{n-1}{n+1}$$

Expected Revenue SPSB:

The expected revenue equals n $\times$ expected payment of a bidder.

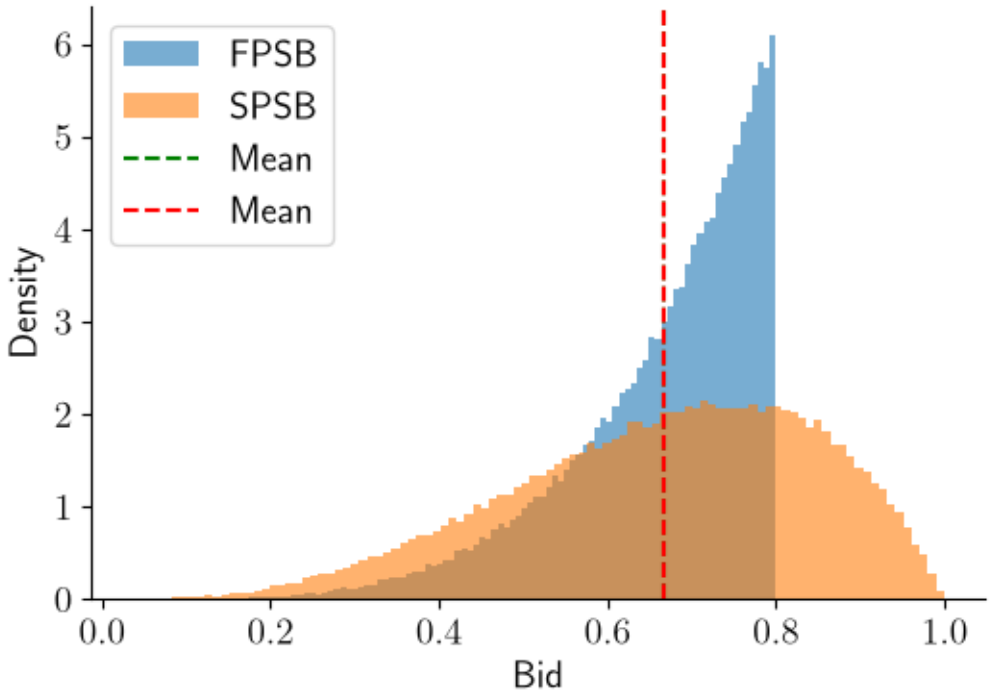
Computing this we get

$$\begin{aligned}
\mathbf{TR} &= n\mathbf{E}_{\mathbf{v_i}}\left[\mathbf{E}_{\mathbf{y_i}}[y_i|y_i < v_i]\mathbf{P}(y_i < v_i) + 0 \times \mathbf{P}(y_i > v_i)\right] \\
&= n\mathbf{E}_{\mathbf{v_i}}\left[\mathbf{E}_{\mathbf{y_i}}[y_i|y_i < v_i]\tilde{F}_{n-1}(v_i)\right] \\
&= n\mathbf{E}_{\mathbf{v_i}}[\frac{n-1}{n} \times v_i \times v_i^{n-1}] \\
&= (n-1)\mathbf{E}_{\mathbf{v_i}}[v_i^n] \\
&= \frac{n-1}{n+1}
\end{aligned}$$

Thus, while probability distributions of winning bids typically differ across the two types of auction, we deduce that expected payments are identical in FPSB and SPSB.

```
fig, ax = plt.subplots(figsize=(6, 4))

for payment, label in zip([winner_pays_fpsb, winner_pays_spsb], ['FPSB', 'SPSB']):
    print('The average payment of %s: %.4f. Std.: %.4f. Median: %.4f' % (
        label, payment.mean(), payment.std(), np.median(payment)))
    ax.hist(payment, density=True, alpha=0.6, label=label, bins=100)

ax.axvline(winner_pays_fpsb.mean(), ls='--', c='g', label='Mean')
ax.axvline(winner_pays_spsb.mean(), ls='--', c='r', label='Mean')

ax.legend(loc='best')
ax.set_xlabel('Bid')
ax.set_ylabel('Density')
sns.despine()
```

```
The average payment of FPSB: 0.6665. Std.: 0.1129. Median: 0.6967
The average payment of SPSB: 0.6667. Std.: 0.1782. Median: 0.6862
```



Summary of FPSB and SPSB results with uniform distribution on $[0, 1]$

Auction: Sealed-Bid	First-Price	Second-Price
Winner	Agent with highest bid	Agent with highest bid
Winner pays	Winner's bid	Second-highest bid
Loser pays	0	0
Dominant strategy	No dominant strategy	Bidding truthfully is dominant strategy
Bayesian Nash equilibrium	Bidder i bids $\frac{n-1}{n}v_i$	Bidder i truthfully bids v_i
Auctioneer's revenue	$\frac{n-1}{n+1}$	$\frac{n-1}{n+1}$

Detour: Computing a Bayesian Nash Equibrium for FPSB

The Revenue Equivalence Theorem lets us find an optimal bidding strategy for a FPSB auction from outcomes of a SPSB auction.

Let $b(v_i)$ be the optimal bid in a FPSB auction.

The revenue equivalence theorem tells us that a bidder agent with value v_i on average receives the same **payment** in the two types of auction.

Consequently,

$$b(v_i)\mathbf{P}(y_i < v_i) + 0 * \mathbf{P}(y_i \geq v_i) = \mathbf{E}_{y_i}[y_i | y_i < v_i]\mathbf{P}(y_i < v_i) + 0 * \mathbf{P}(y_i \geq v_i)$$

It follows that an optimal bidding strategy in a FPSB auction is $b(v_i) = \mathbf{E}_{y_i}[y_i | y_i < v_i]$.

134.10 Calculation of bid price in FPSB

In equations (134.1) and (134.2), we displayed formulas for optimal bids in a symmetric Bayesian Nash Equilibrium of a FPSB auction.

$$\mathbf{E}[y_i | y_i < v_i]$$

where

- $v_i =$ value of bidder i
- $y_i =:$ maximum value of all bidders except i, i.e., $y_i = \max_{j \neq i} v_j$

Above, we computed an optimal bid price in a FPSB auction analytically for a case in which private values are uniformly distributed.

For most probability distributions of private values, analytical solutions aren't easy to compute.

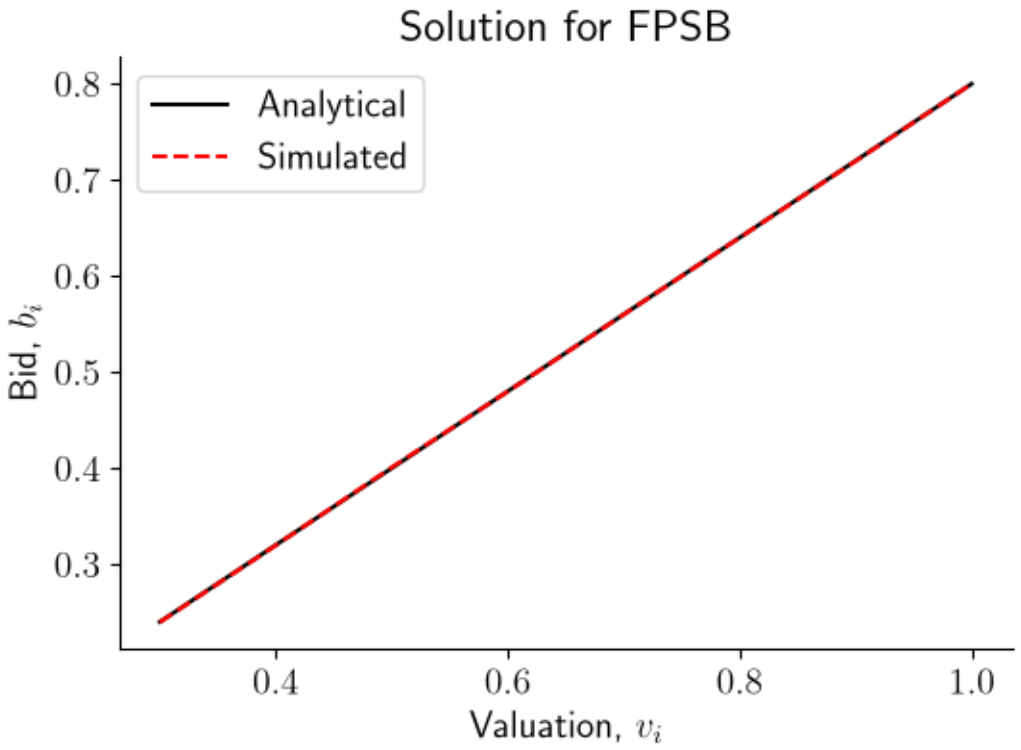
Instead, we can compute bid prices in FPSB auctions numerically as functions of the distribution of private values.

```
def evaluate_largest(v_hat, array, order=1):
    """
    A method to estimate the largest (or certain-order largest) value of the other␣
↪biders,
    conditional on player 1 wins the auction.

    Parameters:
    ----------
    v_hat : float, the value of player 1. The biggest value in the auction that␣
↪player 1 wins.

    array: 2 dimensional array of bidders' values in shape of (N,R),
           where N: number of players, R: number of auctions

    order: int. The order of largest number among bidders who lose.
                e.g. the order for largest number beside winner is 1.
                     the order for second-largest number beside winner is 2.

    """
    N, R = array.shape

    # drop the first row because we assume first row is the winner's bid
    array_residual = array[1:, :].copy()

    winning_auctions_mask = (array_residual < v_hat).all(axis=0)

    num_winning_auctions = np.sum(winning_auctions_mask)

    if num_winning_auctions == 0:
        return np.nan

    array_conditional = array_residual[:, winning_auctions_mask]

    array_conditional_sorted = np.sort(array_conditional, axis=0)

    order_largest_bids = array_conditional_sorted[-order, :]

    return np.mean(order_largest_bids)
```

We can check the accuracy of our `evaluate_largest` method by comparing it with an analytical solution.

We find that the `evaluate_largest` method functions well

```
v_grid = np.linspace(0.3, 1, 8)
bid_analytical = b_star(v_grid, N)

# Redraw valuations
v = np.random.uniform(0, 1, (N, R))
bid_simulated = [evaluate_largest(ii, v) for ii in v_grid]

fig, ax = plt.subplots(figsize=(6, 4))

ax.plot(v_grid, bid_analytical, '-', color='k', label='Analytical')
ax.plot(v_grid, bid_simulated, '--', color='r', label='Simulated')

ax.legend(loc='best')
ax.set_xlabel('Valuation, $v_i$')
ax.set_ylabel('Bid, $b_i$')
ax.set_title('Solution for FPSB')
sns.despine()
```

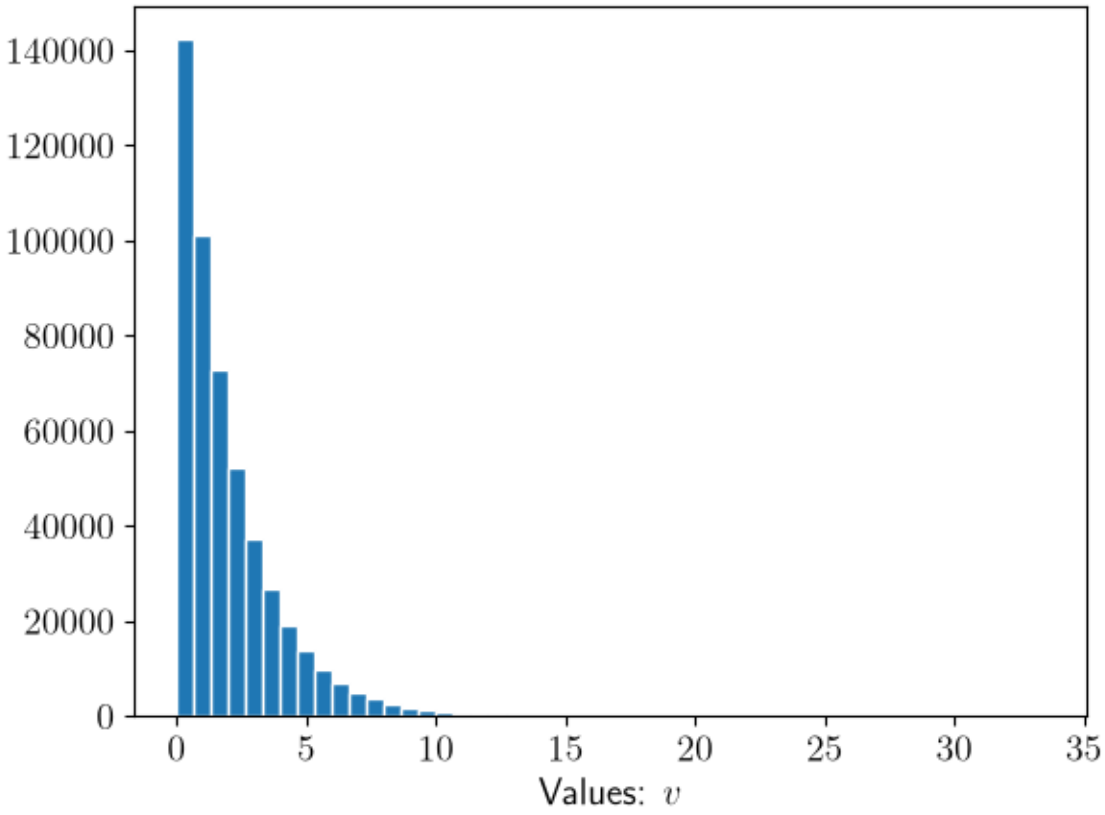


134.11 χ^2 Distribution

Let's try an example in which the distribution of private values is a χ^2 distribution.

We'll start by taking a look at a χ^2 distribution with the help of the following Python code:

```
np.random.seed(1337)
v = np.random.chisquare(df=2, size=(N * R,))

plt.hist(v, bins=50, edgecolor='w')
plt.xlabel('Values: $v$')
plt.show()
```



Now we'll get Python to construct a bid price function

```
np.random.seed(1337)
v = np.random.chisquare(df=2, size=(N, R))

# we compute the quantile of v as our grid
pct_quantile = np.linspace(0, 100, 101)[1:-1]
v_grid = np.percentile(v.flatten(), q=pct_quantile)

# nan values are returned for some low quantiles due to lack of observations
EV = [evaluate_largest(ii, v) for ii in v_grid]
```

```
# we insert 0 into our grid and bid price function as a complement
EV = np.insert(EV, 0, 0)
```

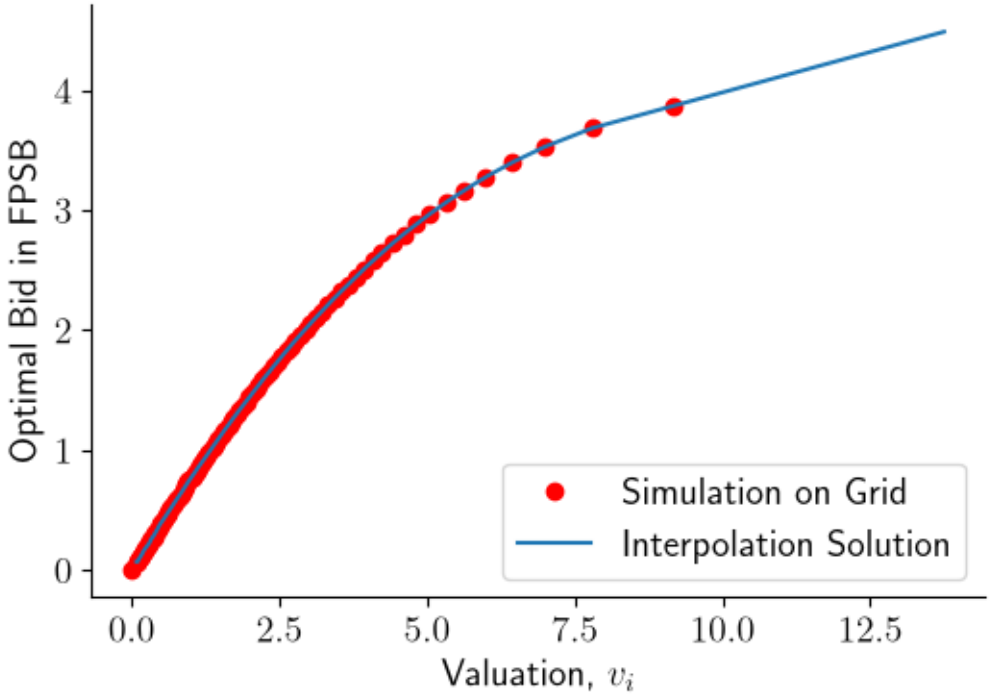
(continues on next page)

(continued from previous page)

```
v_grid = np.insert(v_grid, 0, 0)

b_star_num = interp.interp1d(v_grid, EV, fill_value="extrapolate")
```

We check our bid price function by computing and visualizing the result.

```
pct_quantile_fine = np.linspace(0, 100, 1001)[1:-1]
v_grid_fine = np.percentile(v.flatten(), q=pct_quantile_fine)

fig, ax = plt.subplots(figsize=(6, 4))

ax.plot(v_grid, EV, 'or', label='Simulation on Grid')
ax.plot(v_grid_fine, b_star_num(v_grid_fine),
                '-', label='Interpolation Solution')

ax.legend(loc='best')
ax.set_xlabel('Valuation, $v_i$')
ax.set_ylabel('Optimal Bid in FPSB')
sns.despine()
```



Now we can use Python to compute the probability distribution of the price paid by the winning bidder

```
b = b_star_num(v)

idx = np.argsort(v, axis=0)
# same as np.sort(v, axis=0), except now we retain the idx
v = np.take_along_axis(v, idx, axis=0)
b = np.take_along_axis(b, idx, axis=0)

ii = np.repeat(np.arange(1, N + 1)[:, None], R, axis=1)
ii = np.take_along_axis(ii, idx, axis=0)
```

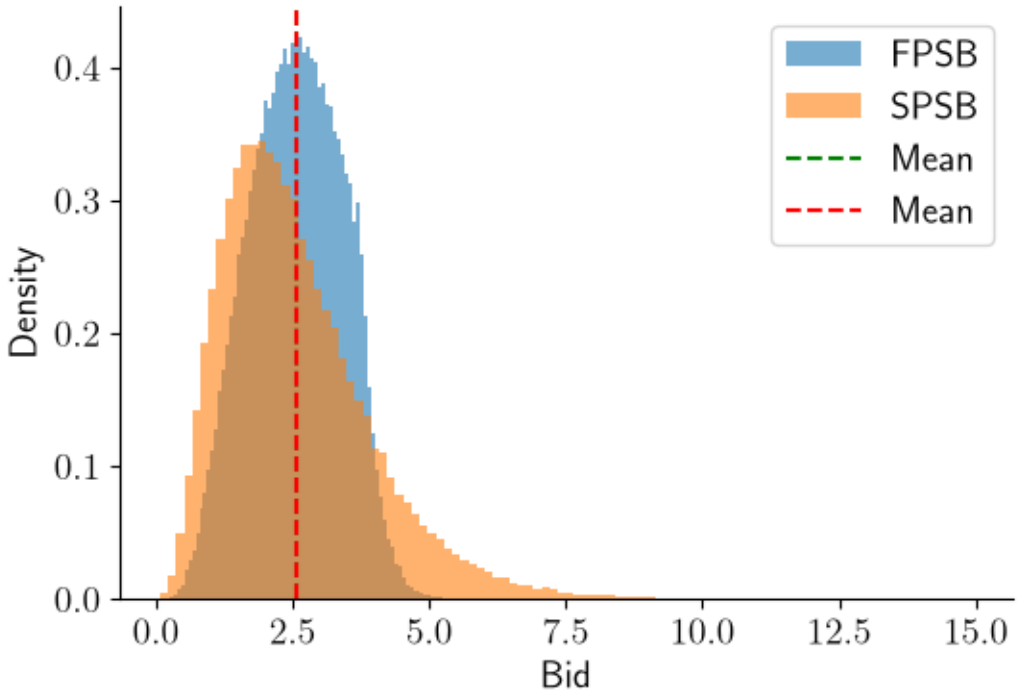
(continues on next page)

(continued from previous page)

```
winning_player = ii[-1, :]

# highest bid
winner_pays_fpsb = b[-1, :]
# 2nd-highest valuation
winner_pays_spsb = v[-2, :]
```

```
fig, ax = plt.subplots(figsize=(6, 4))

for payment, label in zip([winner_pays_fpsb, winner_pays_spsb],
                          ['FPSB', 'SPSB']):
    print('The average payment of %s: %.4f. Std.: %.4f. Median: %.4f' % (
        label, payment.mean(), payment.std(), np.median(payment)))
    ax.hist(payment, density=True, alpha=0.6, label=label, bins=100)

ax.axvline(winner_pays_fpsb.mean(), ls='--', c='g', label='Mean')
ax.axvline(winner_pays_spsb.mean(), ls='--', c='r', label='Mean')

ax.legend(loc='best')
ax.set_xlabel('Bid')
ax.set_ylabel('Density')
sns.despine()
```

```
The average payment of FPSB: 2.5693. Std.: 0.8383. Median: 2.5829
The average payment of SPSB: 2.5661. Std.: 1.3580. Median: 2.3180
```



134.12 Code summary

We assemble the functions that we have used into a Python class

```
class bid_price_solution:

    def __init__(self, array):
        """
        A class that can plot the value distribution of bidders,
        compute the optimal bid price for bidders in FPSB
        and plot the distribution of winner's payment in both FPSB and SPSB

        Parameters:
        ----------

        array: 2 dimensional array of bidders' values in shape of (N, R),
               where N: number of players, R: number of auctions

        """
        self.value_mat = array.copy()

        return None

    def plot_value_distribution(self):
        plt.hist(self.value_mat.flatten(), bins=50, edgecolor='w')
        plt.xlabel('Values: $v$')
        plt.show()

        return None

    def evaluate_largest(self, v_hat, order=1):
        N, R = self.value_mat.shape

        # drop the first row because we assume first row is the winner's bid
        array_residual = self.value_mat[1:, :].copy()

        winning_auctions_mask = (array_residual < v_hat).all(axis=0)

        num_winning_auctions = np.sum(winning_auctions_mask)

        if num_winning_auctions == 0:
            return np.nan

        array_conditional = array_residual[:, winning_auctions_mask]
        array_conditional_sorted = np.sort(array_conditional, axis=0)
        order_largest_bids = array_conditional_sorted[-order, :]

        return np.mean(order_largest_bids)

    def compute_optimal_bid_FPSB(self):
        # we compute the quantile of v as our grid
        pct_quantile = np.linspace(0, 100, 101)[1:-1]
        v_grid = np.percentile(self.value_mat.flatten(), q=pct_quantile)

        # nan values are returned for some low quantiles due to lack of observations
        EV = [self.evaluate_largest(ii) for ii in v_grid]
```

(continues on next page)

(continued from previous page)

```
        # we insert 0 into our grid and bid price function as a complement
        EV = np.insert(EV, 0, 0)
        v_grid = np.insert(v_grid, 0, 0)

        self.b_star_num = interp.interp1d(v_grid, EV,
                                          fill_value="extrapolate")

        pct_quantile_fine = np.linspace(0, 100, 1001)[1:-1]
        v_grid_fine = np.percentile(self.value_mat.flatten(),
                                    q=pct_quantile_fine)

        fig, ax = plt.subplots(figsize=(6, 4))

        ax.plot(v_grid, EV, 'or', label='Simulation on Grid')
        ax.plot(v_grid_fine, self.b_star_num(v_grid_fine),
                             '-', label='Interpolation Solution')

        ax.legend(loc='best')
        ax.set_xlabel('Valuation, $v_i$')
        ax.set_ylabel('Optimal Bid in FPSB')
        sns.despine()

        return None

    def plot_winner_payment_distribution(self):
        self.b = self.b_star_num(self.value_mat)

        idx = np.argsort(self.value_mat, axis=0)
        # same as np.sort(v, axis=0), except now we retain the idx
        self.v = np.take_along_axis(self.value_mat, idx, axis=0)
        self.b = np.take_along_axis(self.b, idx, axis=0)

        N, R = self.value_mat.shape
        self.ii = np.repeat(np.arange(1, N + 1)[:, None], R, axis=1)
        self.ii = np.take_along_axis(self.ii, idx, axis=0)

        winning_player = self.ii[-1, :]

        # highest bid
        winner_pays_fpsb = self.b[-1, :]
        # 2nd-highest valuation
        winner_pays_spsb = self.v[-2, :]

        fig, ax = plt.subplots(figsize=(6, 4))

        for payment, label in zip([winner_pays_fpsb, winner_pays_spsb],
                                  ['FPSB', 'SPSB']):
            print('The average payment of %s: %.4f. Std.: %.4f. Median: %.4f' %
                  (label, payment.mean(), payment.std(), np.median(payment)))
            ax.hist(payment, density=True, alpha=0.6, label=label, bins=100)

        ax.axvline(winner_pays_fpsb.mean(), ls='--', c='g', label='Mean')
        ax.axvline(winner_pays_spsb.mean(), ls='--', c='r', label='Mean')

        ax.legend(loc='best')
        ax.set_xlabel('Bid')
        ax.set_ylabel('Density')
```

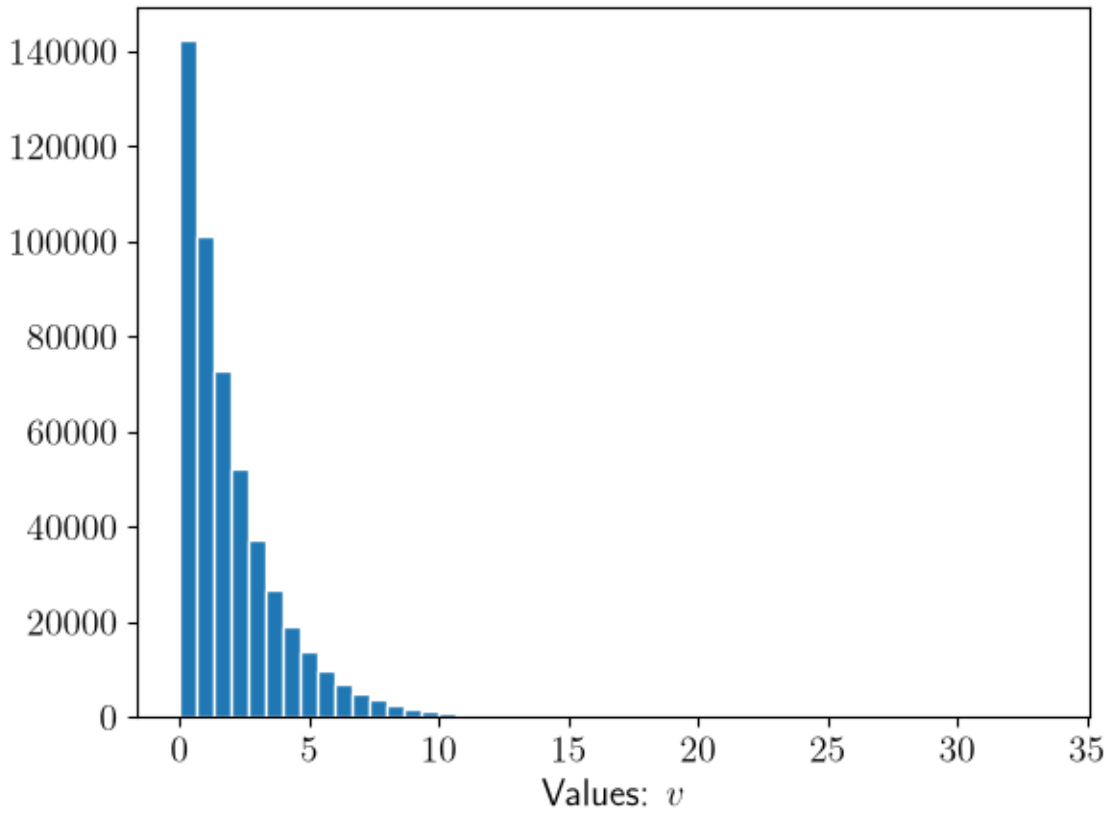
(continues on next page)

(continued from previous page)

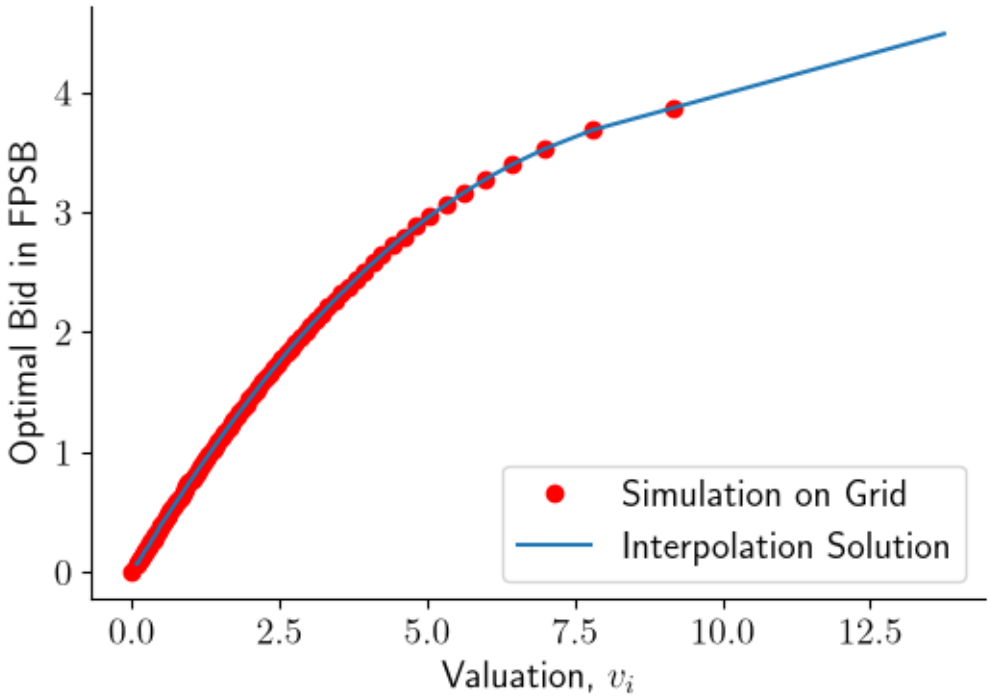
```
        sns.despine()

        return None
```

```
np.random.seed(1337)
v = np.random.chisquare(df=2, size=(N, R))

chi_squ_case = bid_price_solution(v)
```

```
chi_squ_case.plot_value_distribution()
```

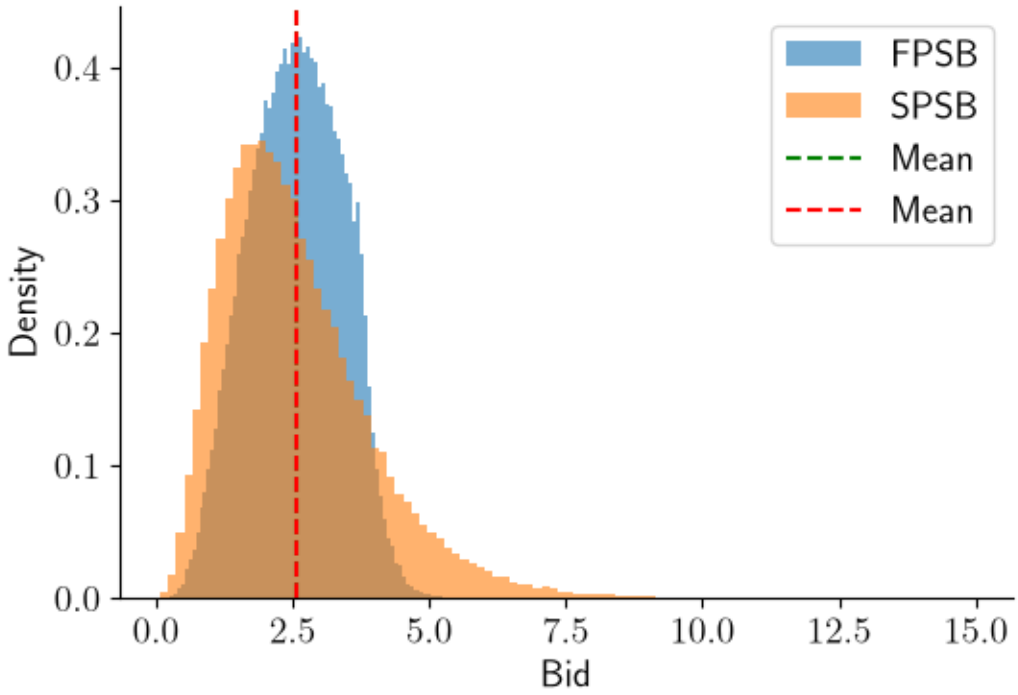



```
chi_squ_case.compute_optimal_bid_FPSB()
```




```
chi_squ_case.plot_winner_payment_distribution()
```

```
The average payment of FPSB: 2.5693. Std.: 0.8383. Median: 2.5829
The average payment of SPSB: 2.5661. Std.: 1.3580. Median: 2.3180
```



134.13 References

1. Wikipedia for FPSB: https://en.wikipedia.org/wiki/First-price_sealed-bid_auction
2. Wikipedia for SPSB: https://en.wikipedia.org/wiki/Vickrey_auction
3. Chandra Chekuri's lecture note for algorithmic game theory: https://chekuri.cs.illinois.edu/teaching/spring2008/Lectures/scribed/Notes20.pdf
4. Tim Salmon. ECO 4400 Supplemental Handout: All About Auctions: https://s2.smu.edu/tsalmon/auctions.pdf
5. Auction Theory- Revenue Equivalence Theorem: https://michaellevet.wordpress.com/2015/07/06/auction-theory-revenue-equivalence-theorem/
6. Order Statistics: https://online.stat.psu.edu/stat415/book/export/html/834

CHAPTER

FIVE

MULTIPLE GOOD ALLOCATION MECHANISMS

```
!pip install "prettytable<3.18"
```

135.1 Overview

This lecture describes two mechanisms for allocating n private goods ("houses") to m people ("buyers").

We assume that $m > n$ so that there are more potential buyers than there are houses.

Prospective buyers regard the houses as **substitutes**.

Buyer j attaches value v_{ij} to house i.

These values are **private**

- v_{ij} is known only to person j unless person j chooses to tell someone.

We require that a mechanism allocate **at most** one house to one prospective buyer.

We describe two distinct mechanisms

- A multiple rounds, ascending bid auction
- A special case of a Groves-Clarke [Groves, 1973], [Clarke, 1971] mechanism with a benevolent social planner

Note

In 1994, the multiple rounds, ascending bid auction was actually used by Stanford University to sell leases to 9 lots on the Stanford campus to eligible faculty members.

We begin with overviews of the two mechanisms.

135.2 Ascending Bids Auction for Multiple Goods

An auction is administered by an **auctioneer**

The auctioneer has an $n \times 1$ vector r of reservation prices on the n houses.

The auctioneer sells house i only if the final price bid for it exceeds r_i

The auctioneer allocates all n houses **simultaneously**

The auctioneer does not know bidders' private values v_{ij}

There are multiple **rounds**

- during each round, active participants can submit bids on any of the n houses
- each bidder can bid on only one house during one round
- a person who was high bidder on a particular house in one round is understood to submit that same bid for the same house in the next round
- between rounds, a bidder who was not a high bidder can change the house on which he/she chooses to bid
- the auction ends when the price of no house changes from one round to the next
- all n houses are allocated after the final round
- house i is retained by the auctioneer if not prospective buyer offers more that r_i for the house

In this auction, person j never tells anyone else his/her private values v_{ij}

135.3 A Benevolent Planner

This mechanism is designed so that all prospective buyers voluntarily choose to reveal their private values to a **social planner** who uses them to construct a socially optimal allocation.

Among all feasible allocations, a **socially optimal allocation** maximizes the sum of private values across all prospective buyers.

The planner tells everyone in advance how he/she will allocate houses based on the matrix of values that prospective buyers report.

The mechanism provide every prospective buyer an incentive to reveal his vector of private values to the planner.

After the planner receives everyone's vector of private values, the planner deploys a **sequential** algorithm to determine an **allocation** of houses and a set of **fees** that he charges awardees for the negative **externality** that their presence impose on other prospective buyers.

135.4 Equivalence of Allocations

Remarkably, these two mechanisms can produce virtually identical allocations.

We construct Python code for both mechanism.

We also work out some examples by hand or almost by hand.

Next, let's dive down into the details.

135.5 Ascending Bid Auction

135.5.1 Basic Setting

We start with a more detailed description of the setting.

- A seller owns n houses that he wants to sell for the maximum possible amounts to a set of m prospective eligible buyers.
- The seller wants to sell at most one house to each potential buyer.

- There are m potential eligible buyers, identified by $j = [1, 2, \ldots, m]$
  - Each potential buyer is permitted to buy at most one house.
  - Buyer j would be willing to pay at most v_{ij} for house i.
  - Buyer j knows $v_{ij}, i = 1, \ldots, n$, but no one else does.
  - If buyer j pays p_i for house i, he enjoys surplus value $v_{ij} - p_i$.
  - Each buyer j wants to choose the i that maximizes his/her surplus value $v_{ij} - p_i$.
  - The seller wants to maximize $\sum_i p_i$.

The seller conducts a **simultaneous, multiple goods, ascending bid auction**.

Outcomes of the auction are

- An $n \times 1$ vector p of sales prices $p = [p_1, \ldots, p_n]$ for the n houses.
- An $n \times m$ matrix Q of 0's and 1's, where $Q_{ij} = 1$ if and only if person j bought house i.
- An $n \times m$ matrix S of surplus values consisting of all zeros unless person j bought house i, in which case $S_{ij} = v_{ij} - p_i$

We describe rules for the auction it terms of **pseudo code**.

The pseudo code will provide a road map for writing Python code to implement the auction.

135.6 Pseudocode

Here is a quick sketch of a possible simple structure for our Python code

Inputs:

- n, m.
- an $n \times m$ non-negative matrix v of private values
- an $n \times 1$ vector r of seller-specified reservation prices
- the seller will not accept a price less than r_i for house i
- we are free to think of these reservation prices as private values of a fictitious $m+1$ th buyer who does not actually participate in the auction
- initial bids can be thought of starting at r
- a scalar ϵ of seller-specified minimum price-bid increments

For each round of the auction, new bids on a house must be at least the prevailing highest bid so far **plus** ϵ

Auction Protocols

- the auction consists of a finite number of **rounds**
- in each round, a prospective buyer can bid on one and only one house
- after each round, a house is temporarily awarded to the person who made the highest bid for that house
  - temporarily winning bids on each house are announced
  - this sets the stage to move on to the next round
- a new round is held

  - bids for temporary winners from the previous round are again attached to the houses on which they bid; temporary winners of the last round leave their bids from the previous round unchanged
  - all other active prospective buyers must submit a new bid on some house
  - new bids on a house must be at least equal to the prevailing temporary price that won the last round **plus** ϵ
  - if a person does not submit a new bid and was also not a temporary winner from the previous round, that person must drop out of the auction permanently
  - for each house, the highest bid, whether it is a new bid or was the temporary winner from the previous round, is announced, with the person who made that new (temporarily) winning bid being (temporarily) awarded the house to start the next round
- rounds continue until no price on **any** house changes from the previous round
- houses are sold to the winning bidders from the final round at the prices that they bid

Outputs:

- an $n \times 1$ vector p of sales prices
- an $n \times m$ matrix S of surplus values consisting of all zeros unless person j bought house i, in which case $S_{ij} = v_{ij} - p_i$
- an $n \times (m+1)$ matrix Q of 0's and 1's that tells which buyer bought which house. (The last column accounts for unsold houses.)

Proposed buyer strategy:

In this pseudo code and the actual Python code below, we'll assume that all buyers choose to use the following strategy

- The strategy is optimal for each buyer

Each buyer $j = 1, \ldots, m$ uses the same strategy.

The strategy has the form:

- Let $\check{p}^t$ be the $n \times 1$ vector of prevailing highest-bid prices at the beginning of round t
- Let $\epsilon > 0$ be the minimum bid increment specified by the seller
- For each prospective buyer j, compute the index of the best house to bid on during round t, namely $\hat{i}_t = \text{argmax}_i\{[v_{ij} - \check{p}_i^t - \epsilon]\}$
- If $\max_i\{[v_{ij} - \check{p}_i^t - \epsilon]\} \leq 0$, person j permanently drops out of the auction at round t
- If $v_{\hat{i}_t,j} - \check{p}_i^t - \epsilon > 0$, person j bids $\check{p}_i^t + \epsilon$ on house j

Resolving ambiguities: The protocols we have described so far leave open two possible sources of ambiguity.

(1) **The optimal bid choice for buyers in each round.** It is possible that a buyer has the same surplus value for multiple houses. The argmax function in Python always returns the first argmax element. We instead prefer to randomize among such winner. For that reason, we write our own argmax function below.

(2) **Seller's choice of winner if same price bid cast by several buyers.** To resolve this ambiguity, we use the np.random.choice function below.

Given the randomness in outcomes, it is possible that different allocations of houses could emerge from the same inputs.

However, this will happen only when the bid price increment ϵ is nonnegligible.

```
import numpy as np
import prettytable as pt

np.random.seed(100)
```

```
np.set_printoptions(precision=3, suppress=True)
```

135.7 An Example

Before building a Python class, let's step by step solve things almost "by hand" to grasp how the auction proceeds.

A step-by-step procedure also helps reduce bugs, especially when the value matrix is peculiar (e.g. the differences between values are negligible, a column containing identical values or multiple buyers have the same valuation etc.).

Fortunately, our auction behaves well and robustly with various peculiar matrices.

We provide some examples later in this lecture.

```
v = np.array([[8, 5, 9, 4],
              [4, 11, 7, 4],
              [9, 7, 6, 4]])
n, m = v.shape
r = np.array([2, 1, 0])
ϵ = 1
p = r.copy()
buyer_list = np.arange(m)
house_list = np.arange(n)
```

```
v
```

```
array([[ 8,  5,  9,  4],
       [ 4, 11,  7,  4],
       [ 9,  7,  6,  4]])
```

Remember that column indexes j indicate buyers and row indexes i indicate houses.

The above value matrix v is peculiar in the sense that Buyer 3 (indexed from 0) puts the same value 4 on every house being sold.

Maybe buyer 3 is a bureaucrat who purchases these house simply by following instructions from his superior.

```
r
```

```
array([2, 1, 0])
```

```
def find_argmax_with_randomness(v):
    """
    We build our own verion of argmax function such that the argmax index will be␣
↪returned randomly
    when there are multiple maximum values. This function is similiar to np.argmax(v,
↪axis=0)

    Parameters:
    ----------
    v: 2 dimensional np.array

    """

    n, m = v.shape
    index_array = np.arange(n)
```

(continues on next page)

(continued from previous page)

```
    result=[]

    for ii in range(m):
        max_value = v[:,ii].max()
        result.append(np.random.choice(index_array[v[:,ii] == max_value]))

    return np.array(result)
```

```
def present_dict(dt):
    """
    A function that present the information in table.

    Parameters:
    ----------
    dt: dictionary.

    """

    ymtb = pt.PrettyTable()
    ymtb.field_names = ['House Number', *dt.keys()]
    ymtb.add_row(['Buyer', *dt.values()])
    print(ymtb)
```

Check Kick Off Condition

```
def check_kick_off_condition(v, r, ϵ):
    """
    A function that checks whether the auction could be initiated given the␣
 ↪reservation price and value matrix.
    To avoid the situation that the reservation prices are so high that no one would␣
 ↪even bid in the first round.

    Parameters:
    ----------
    v : value matrix of the shape (n,m).

    r: the reservation price

    ϵ: the minimun price increment in each round

    """

    # we convert the price vector to a matrix in the same shape as value matrix to␣
 ↪facilitate subtraction
    p_start = (ϵ+r)[:,None] @ np.ones(m)[None,:]

    surplus_value = v - p_start
    buyer_decision = (surplus_value > 0).any(axis = 0)
    return buyer_decision.any()
```

```
check_kick_off_condition(v, r, ϵ)
```

```
np.True_
```

135.7.1 round 1

submit bid

```
def submit_initial_bid(p_initial, ϵ, v):
    """
    A function that describes the bid information in the first round.

    Parameters:
    ----------
    p_initial: the price (or the reservation prices) at the beginning of auction.

    v: the value matrix

    ϵ: the minimun price increment in each round

    Returns:
    ----------
    p: price array after this round of bidding

    bid_info: a dictionary that contains bidding information (house number as keys␣
 ↪and buyer as values).

    """

    p = p_initial.copy()
    p_start_mat = (ϵ + p)[:,None] @ np.ones(m)[None,:]
    surplus_value = v - p_start_mat

    # we only care about active buyers who have positve surplus values
    active_buyer_diagnosis = (surplus_value > 0).any(axis = 0)
    active_buyer_list = buyer_list[active_buyer_diagnosis]
    active_buyer_surplus_value = surplus_value[:,active_buyer_diagnosis]
    active_buyer_choice = find_argmax_with_randomness(active_buyer_surplus_value)
    # choice means the favourite houses given the current price and ϵ

    # we only retain the unique house index because prices increase once at one round
    house_bid =  list(set(active_buyer_choice))
    p[house_bid] += ϵ

    bid_info = {}
    for house_num in house_bid:
        bid_info[house_num] = active_buyer_list[active_buyer_choice == house_num]

    return p, bid_info
```

```
p, bid_info = submit_initial_bid(p, ϵ, v)
```

```
p
```

```
array([3, 2, 1])
```

```
present_dict(bid_info)
```

```
+--------------+-----+-----+-------+
| House Number |  0  |  1  |   2   |
```

(continues on next page)

(continued from previous page)

```
+--------------+-----+-----+-------+
|    Buyer     | [2] | [1] | [0 3] |
+--------------+-----+-----+-------+
```

check terminal condition

Notice that two buyers bid for house 2 (indexed from 0).

Because the auction protocol does not specify a selection rule in this case, we simply select a winner **randomly**.

This is reasonable because the seller can't distinguish these buyers and doesn't know the valuation of each buyer.

It is both convenient and practical for him to just pick a winner randomly.

There is a 50% probability that Buyer 3 is chosen as the winner for house 2, although he values it less than buyer 0.

In this case, buyer 0 has to bid one more time with a higher price, which crowds out Buyer 3.

Therefore, final price could be 3 or 4, depending on the winner in the last round.

```
def check_terminal_condition(bid_info, p, v):
    """
    A function that checks whether the auction ends.

    Recall that the auction ends when either losers have non-positive surplus values␣
↪for each house
    or there is no loser (every buyer gets a house).

    Parameters:
    ----------
    bid_info: a dictionary that contains bidding information of house numbers (as␣
↪keys) and buyers (as values).

    p: np.array. price array of houses

    v: value matrix

    Returns:
    ----------
    allocation: a dictionary that descirbe how the houses bid are assigned.

    winner_list: a list of winners

    loser_list: a list of losers

    """

    # there may be several buyers bidding one house, we choose a winner randomly
    winner_list=[np.random.choice(bid_info[ii]) for ii in bid_info.keys()]

    allocation = {house_num:winner for house_num,winner in zip(bid_info.keys(),winner_
↪list)}

    loser_set = set(buyer_list).difference(set(winner_list))
    loser_list = list(loser_set)
    loser_num = len(loser_list)

    if loser_num == 0:
        print('The auction ends because every buyer gets one house.')
```

(continues on next page)

(continued from previous page)

```
        return allocation,winner_list,loser_list

    p_mat = (ϵ + p)[:,None] @ np.ones(loser_num)[None,:]
    loser_surplus_value = v[:,loser_list] - p_mat
    loser_decision = (loser_surplus_value > 0).any(axis = 0)

    print(~(loser_decision.any()))

    return allocation,winner_list,loser_list
```

```
allocation,winner_list,loser_list = check_terminal_condition(bid_info, p, v)
```

```
False
```

```
present_dict(allocation)
```

```
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Buyer     | 2 | 1 | 0 |
+--------------+---+---+---+
```

```
winner_list
```

```
[np.int64(2), np.int64(1), np.int64(0)]
```

```
loser_list
```

```
[np.int64(3)]
```

135.7.2 round 2

From the second round on, the auction proceeds differently from the first round.

Now only active losers (those who have positive surplus values) have an incentive to submit bids to displace temporary winners from the previous round.

```
def submit_bid(loser_list, p, ϵ, v, bid_info):
    """
    A function that executes the bid operation after the first round.
    After the first round, only active losers would cast a new bid with price as old␣
↪price + increment.
    By such bid, winners of last round are replaced by the active losers.

    Parameters:
    ----------
    loser_list: a list that includes the indexes of losers

    p: np.array. price array of houses

    ϵ: minimum increment of bid price
```

(continues on next page)

(continued from previous page)

```
    v: value matrix

    bid_info: a dictionary that contains bidding information of house numbers (as␣
↪keys) and buyers (as values).

    Returns:
    ----------
    p_end: a price array after this round of bidding

    bid_info: a dictionary that contains updated bidding information.

    """

    p_end=p.copy()

    loser_num = len(loser_list)
    p_mat = (ϵ + p_end)[:,None] @ np.ones(loser_num)[None,:]
    loser_surplus_value = v[:,loser_list] - p_mat
    loser_decision = (loser_surplus_value > 0).any(axis = 0)

    active_loser_list = np.array(loser_list)[loser_decision]
    active_loser_surplus_value = loser_surplus_value[:,loser_decision]
    active_loser_choice = find_argmax_with_randomness(active_loser_surplus_value)

    # we retain the unique house index and increasing the corresponding bid price
    house_bid = list(set(active_loser_choice))
    p_end[house_bid] += ϵ

    # we record the bidding information from active losers
    bid_info_active_loser = {}
    for house_num in house_bid:
        bid_info_active_loser[house_num] = active_loser_list[active_loser_choice ==␣
↪house_num]

    # we update the bidding information according to the bidding from actice losers
    for house_num in bid_info_active_loser.keys():
        bid_info[house_num] = bid_info_active_loser[house_num]

    return p_end,bid_info
```

```
p,bid_info = submit_bid(loser_list, p, ϵ, v, bid_info)
```

```
p
```

```
array([3, 2, 2])
```

```
present_dict(bid_info)
```

```
+--------------+-----+-----+-----+
| House Number |  0  |  1  |  2  |
+--------------+-----+-----+-----+
|    Buyer     | [2] | [1] | [3] |
+--------------+-----+-----+-----+
```

```
allocation,winner_list,loser_list = check_terminal_condition(bid_info, p, v)
```

```
False
```

```
present_dict(allocation)
```

```
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Buyer     | 2 | 1 | 3 |
+--------------+---+---+---+
```

135.7.3 round 3

```
p,bid_info = submit_bid(loser_list, p, ϵ, v, bid_info)
```

```
p
```

```
array([3, 2, 3])
```

```
present_dict(bid_info)
```

```
+--------------+-----+-----+-----+
| House Number |  0  |  1  |  2  |
+--------------+-----+-----+-----+
|    Buyer     | [2] | [1] | [0] |
+--------------+-----+-----+-----+
```

```
allocation,winner_list,loser_list = check_terminal_condition(bid_info, p, v)
```

```
False
```

```
present_dict(allocation)
```

```
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Buyer     | 2 | 1 | 0 |
+--------------+---+---+---+
```

135.7.4 round 4

```
p,bid_info = submit_bid(loser_list, p, ϵ, v, bid_info)
```

```
p
```

```
array([3, 3, 3])
```

```
present_dict(bid_info)
```

```
+--------------+-----+-----+-----+
| House Number |  0  |  1  |  2  |
+--------------+-----+-----+-----+
|    Buyer     | [2] | [3] | [0] |
+--------------+-----+-----+-----+
```

Notice that Buyer 3 now switches to bid for house 1 having recongized that house 2 is no longer his best option.

```
allocation,winner_list,loser_list = check_terminal_condition(bid_info, p, v)
```

```
False
```

```
present_dict(allocation)
```

```
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Buyer     | 2 | 3 | 0 |
+--------------+---+---+---+
```

135.7.5 round 5

```
p,bid_info = submit_bid(loser_list, p, ϵ, v, bid_info)
```

```
p
```

```
array([3, 4, 3])
```

```
present_dict(bid_info)
```

```
+--------------+-----+-----+-----+
| House Number |  0  |  1  |  2  |
+--------------+-----+-----+-----+
|    Buyer     | [2] | [1] | [0] |
+--------------+-----+-----+-----+
```

Now Buyer 1 bids for house 1 again with price at 4, which crowds out Buyer 3, marking the end of the auction.

```
allocation,winner_list,loser_list = check_terminal_condition(bid_info, p, v)
```

```
True
```

```
present_dict(allocation)
```

```
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Buyer     | 2 | 1 | 0 |
+--------------+---+---+---+
```

```
# as for the houses unsold

house_unsold_list = list(set(house_list).difference(set(allocation.keys())))
house_unsold_list
```

```
[]
```

```
total_revenue = p[list(allocation.keys())].sum()
total_revenue
```

```
np.int64(10)
```

135.8 A Python Class

Above we simulated an ascending bid auction step by step.

When defining functions, we repeatedly computed some intermediate objects because our Python function loses track of variables once the function is executed.

That of course led to redundancy in our code

It is much more efficient to collect all of the aforementioned code into a class that records information about all rounds.

```
class ascending_bid_auction:

    def __init__(self, v, r, ϵ):
        """
        A class that simulates an ascending bid auction for houses.

        Given buyers' value matrix, sellers' reservation prices and minimum increment␣
↪of bid prices,
        this class can execute an ascending bid auction and present information round␣
↪by round until the end.

        Parameters:
        ----------
        v: 2 dimensional value matrix

        r: np.array of reservation prices

        ϵ: minimum increment of bid price

        """

        self.v = v.copy()
        self.n,self.m = self.v.shape
        self.r = r
        self.ϵ = ϵ
        self.p = r.copy()
        self.buyer_list = np.arange(self.m)
        self.house_list = np.arange(self.n)
        self.bid_info_history = []
        self.allocation_history = []
        self.winner_history = []
```

(continues on next page)

(continued from previous page)

```
        self.loser_history = []


    def find_argmax_with_randomness(self, v):
        n,m = v.shape
        index_array = np.arange(n)
        result=[]

        for ii in range(m):
            max_value = v[:,ii].max()
            result.append(np.random.choice(index_array[v[:,ii] == max_value]))

        return np.array(result)


    def check_kick_off_condition(self):
        # we convert the price vector to a matrix in the same shape as value matrix␣
↪to facilitate subtraction
        p_start = (self.ϵ + self.r)[:,None] @ np.ones(self.m)[None,:]
        self.surplus_value = self.v - p_start
        buyer_decision = (self.surplus_value > 0).any(axis = 0)
        return buyer_decision.any()


    def submit_initial_bid(self):
        # we intend to find the optimal choice of each buyer
        p_start_mat = (self.ϵ + self.p)[:,None] @ np.ones(self.m)[None,:]
        self.surplus_value = self.v - p_start_mat

        # we only care about active buyers who have positve surplus values
        active_buyer_diagnosis = (self.surplus_value > 0).any(axis = 0)
        active_buyer_list = self.buyer_list[active_buyer_diagnosis]
        active_buyer_surplus_value = self.surplus_value[:,active_buyer_diagnosis]
        active_buyer_choice = self.find_argmax_with_randomness(active_buyer_surplus_
↪value)

        # we only retain the unique house index because prices increase once at one␣
↪round
        house_bid =  list(set(active_buyer_choice))
        self.p[house_bid] += self.ϵ

        bid_info = {}
        for house_num in house_bid:
            bid_info[house_num] = active_buyer_list[active_buyer_choice == house_num]
        self.bid_info_history.append(bid_info)

        print('The bid information is')
        ymtb = pt.PrettyTable()
        ymtb.field_names = ['House Number', *bid_info.keys()]
        ymtb.add_row(['Buyer', *bid_info.values()])
        print(ymtb)

        print('The bid prices for houses are')
        ymtb = pt.PrettyTable()
        ymtb.field_names = ['House Number', *self.house_list]
        ymtb.add_row(['Price', *self.p])
        print(ymtb)
```

(continues on next page)

(continued from previous page)

```
        self.winner_list=[np.random.choice(bid_info[ii]) for ii in bid_info.keys()]
        self.winner_history.append(self.winner_list)

        self.allocation = {house_num:[winner] for house_num,winner in zip(bid_info.
↪keys(),self.winner_list)}
        self.allocation_history.append(self.allocation)

        loser_set = set(self.buyer_list).difference(set(self.winner_list))
        self.loser_list = list(loser_set)
        self.loser_history.append(self.loser_list)

        print('The winners are')
        print(self.winner_list)

        print('The losers are')
        print(self.loser_list)
        print('\n')


    def check_terminal_condition(self):
        loser_num = len(self.loser_list)

        if loser_num == 0:
            print('The auction ends because every buyer gets one house.')
            print('\n')
            return True

        p_mat = (self.ϵ + self.p)[:,None] @ np.ones(loser_num)[None,:]
        self.loser_surplus_value = self.v[:,self.loser_list] - p_mat
        self.loser_decision = (self.loser_surplus_value > 0).any(axis = 0)

        return ~(self.loser_decision.any())


    def submit_bid(self):
        bid_info = self.allocation_history[-1].copy()  # we only record the bid info␣
↪of winner

        loser_num = len(self.loser_list)
        p_mat = (self.ϵ + self.p)[:,None] @ np.ones(loser_num)[None,:]
        self.loser_surplus_value = self.v[:,self.loser_list] - p_mat
        self.loser_decision = (self.loser_surplus_value > 0).any(axis = 0)

        active_loser_list = np.array(self.loser_list)[self.loser_decision]
        active_loser_surplus_value = self.loser_surplus_value[:,self.loser_decision]
        active_loser_choice = self.find_argmax_with_randomness(active_loser_surplus_
↪value)

        # we retain the unique house index and increasing the corresponding bid price
        house_bid = list(set(active_loser_choice))
        self.p[house_bid] += self.ϵ

        # we record the bidding information from active losers
        bid_info_active_loser = {}
        for house_num in house_bid:
            bid_info_active_loser[house_num] = active_loser_list[active_loser_choice␣
```

(continues on next page)

(continued from previous page)

```
↪== house_num]

        # we update the bidding information according to the bidding from actice␣
↪losers
        for house_num in bid_info_active_loser.keys():
            bid_info[house_num] = bid_info_active_loser[house_num]
        self.bid_info_history.append(bid_info)

        print('The bid information is')
        ymtb = pt.PrettyTable()
        ymtb.field_names = ['House Number', *bid_info.keys()]
        ymtb.add_row(['Buyer', *bid_info.values()])
        print(ymtb)

        print('The bid prices for houses are')
        ymtb = pt.PrettyTable()
        ymtb.field_names = ['House Number', *self.house_list]
        ymtb.add_row(['Price', *self.p])
        print(ymtb)

        self.winner_list=[np.random.choice(bid_info[ii]) for ii in bid_info.keys()]
        self.winner_history.append(self.winner_list)

        self.allocation = {house_num:[winner] for house_num,winner in zip(bid_info.
↪keys(),self.winner_list)}
        self.allocation_history.append(self.allocation)

        loser_set = set(self.buyer_list).difference(set(self.winner_list))
        self.loser_list = list(loser_set)
        self.loser_history.append(self.loser_list)

        print('The winners are')
        print(self.winner_list)

        print('The losers are')
        print(self.loser_list)
        print('\n')


    def start_auction(self):
        print('The Ascending Bid Auction for Houses')
        print('\n')

        print('Basic Information: %d houses, %d buyers'%(self.n, self.m))

        print('The valuation matrix is as follows')
        ymtb = pt.PrettyTable()
        ymtb.field_names = ['Buyer Number', *(np.arange(self.m))]
        for ii in range(self.n):
            ymtb.add_row(['House %d'%(ii), *self.v[ii,:]])
        print(ymtb)

        print('The reservation prices for houses are')
        ymtb = pt.PrettyTable()
        ymtb.field_names = ['House Number', *self.house_list]
        ymtb.add_row(['Price', *self.r])
        print(ymtb)
```

(continues on next page)

(continued from previous page)

```
        print('The minimum increment of bid price is %.2f' % self.ϵ)
        print('\n')

        ctr = 1
        if self.check_kick_off_condition():
            print('Auction starts successfully')
            print('\n')
            print('Round %d'% ctr)

            self.submit_initial_bid()

            while True:
                if self.check_terminal_condition():
                    print('Auction ends')
                    print('\n')

                    print('The final result is as follows')
                    print('\n')
                    print('The allocation plan is')
                    ymtb = pt.PrettyTable()
                    ymtb.field_names = ['House Number', *self.allocation.keys()]
                    ymtb.add_row(['Buyer', *self.allocation.values()])
                    print(ymtb)

                    print('The bid prices for houses are')
                    ymtb = pt.PrettyTable()
                    ymtb.field_names = ['House Number', *self.house_list]
                    ymtb.add_row(['Price', *self.p])
                    print(ymtb)

                    print('The winners are')
                    print(self.winner_list)

                    print('The losers are')
                    print(self.loser_list)

                    self.house_unsold_list = list(set(self.house_list).
↪difference(set(self.allocation.keys())))
                    print('The houses unsold are')
                    print(self.house_unsold_list)

                    self.total_revenue = self.p[list(self.allocation.keys())].sum()
                    print('The total revenue is %.2f' % self.total_revenue)

                    break

                ctr += 1
                print('Round %d'% ctr)
                self.submit_bid()

            # we compute the surplus matrix S and the quantity matrix X as required␣
↪in 1.1
            self.S = np.zeros((self.n, self.m))
            for ii,jj in zip(self.allocation.keys(),self.allocation.values()):
                self.S[ii,jj] = self.v[ii,jj] - self.p[ii]

            self.Q = np.zeros((self.n, self.m + 1))  # the last column records the␣
```

(continues on next page)

(continued from previous page)

```
↪houses unsold
            for ii,jj in zip(self.allocation.keys(),self.allocation.values()):
                self.Q[ii,jj] = 1
            for ii in self.house_unsold_list:
                self.Q[ii,-1] = 1

            # we sort the allocation result by the house number
            house_sold_list = list(self.allocation.keys())
            house_sold_list.sort()

            dict_temp = {}
            for ii in house_sold_list:
                dict_temp[ii] = self.allocation[ii]
            self.allocation = dict_temp

        else:
            print('The auction can not start because of high reservation prices')
```

Let's use our class to conduct the auction described in one of the above examples.

```
v = np.array([[8,5,9,4],[4,11,7,4],[9,7,6,4]])
r = np.array([2,1,0])
ϵ = 1

auction_1 = ascending_bid_auction(v, r, ϵ)

auction_1.start_auction()
```

```
The Ascending Bid Auction for Houses


Basic Information: 3 houses, 4 buyers
The valuation matrix is as follows
+--------------+---+----+---+---+
| Buyer Number | 0 | 1  | 2 | 3 |
+--------------+---+----+---+---+
|   House 0    | 8 | 5  | 9 | 4 |
|   House 1    | 4 | 11 | 7 | 4 |
|   House 2    | 9 | 7  | 6 | 4 |
+--------------+---+----+---+---+
The reservation prices for houses are
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Price     | 2 | 1 | 0 |
+--------------+---+---+---+
The minimum increment of bid price is 1.00


Auction starts successfully


Round 1
The bid information is
+--------------+-----+-----+-------+
| House Number |  0  |  1  |   2   |
```

(continues on next page)

(continued from previous page)

```
+--------------+-----+-----+-------+
|    Buyer     | [2] | [1] | [0 3] |
+--------------+-----+-----+-------+
The bid prices for houses are
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Price     | 3 | 2 | 1 |
+--------------+---+---+---+
The winners are
[np.int64(2), np.int64(1), np.int64(0)]
The losers are
[np.int64(3)]


Round 2
The bid information is
+--------------+--------------+---------------+-----+
| House Number |       0      |       1       |  2  |
+--------------+--------------+---------------+-----+
|    Buyer     | [np.int64(2)] | [np.int64(1)] | [3] |
+--------------+--------------+---------------+-----+
The bid prices for houses are
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Price     | 3 | 2 | 2 |
+--------------+---+---+---+
The winners are
[np.int64(2), np.int64(1), np.int64(3)]
The losers are
[np.int64(0)]


Round 3
The bid information is
+--------------+--------------+---------------+-----+
| House Number |       0      |       1       |  2  |
+--------------+--------------+---------------+-----+
|    Buyer     | [np.int64(2)] | [np.int64(1)] | [0] |
+--------------+--------------+---------------+-----+
The bid prices for houses are
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Price     | 3 | 2 | 3 |
+--------------+---+---+---+
The winners are
[np.int64(2), np.int64(1), np.int64(0)]
The losers are
[np.int64(3)]


Round 4
The bid information is
+--------------+--------------+-----+---------------+
| House Number |       0      |  1  |       2       |
```

(continues on next page)

(continued from previous page)

```
+--------------+---------------+-----+---------------+
|    Buyer     | [np.int64(2)] | [3] | [np.int64(0)] |
+--------------+---------------+-----+---------------+
The bid prices for houses are
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Price     | 3 | 3 | 3 |
+--------------+---+---+---+
The winners are
[np.int64(2), np.int64(3), np.int64(0)]
The losers are
[np.int64(1)]


Round 5
The bid information is
+--------------+---------------+-----+---------------+
| House Number |       0       |  1  |       2       |
+--------------+---------------+-----+---------------+
|    Buyer     | [np.int64(2)] | [1] | [np.int64(0)] |
+--------------+---------------+-----+---------------+
The bid prices for houses are
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Price     | 3 | 4 | 3 |
+--------------+---+---+---+
The winners are
[np.int64(2), np.int64(1), np.int64(0)]
The losers are
[np.int64(3)]


Auction ends


The final result is as follows


The allocation plan is
+--------------+---------------+---------------+---------------+
| House Number |       0       |       1       |       2       |
+--------------+---------------+---------------+---------------+
|    Buyer     | [np.int64(2)] | [np.int64(1)] | [np.int64(0)] |
+--------------+---------------+---------------+---------------+
The bid prices for houses are
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Price     | 3 | 4 | 3 |
+--------------+---+---+---+
The winners are
[np.int64(2), np.int64(1), np.int64(0)]
The losers are
[np.int64(3)]
The houses unsold are
```

(continues on next page)

(continued from previous page)

```
[]
The total revenue is 10.00
```

```
# the surplus matrix S

auction_1.S
```

```
array([[0., 0., 6., 0.],
       [0., 7., 0., 0.],
       [6., 0., 0., 0.]])
```

```
# the quantity matrix X

auction_1.Q
```

```
array([[0., 0., 1., 0., 0.],
       [0., 1., 0., 0., 0.],
       [1., 0., 0., 0., 0.]])
```

135.9 Robustness Checks

Let's do some stress testing of our code by applying it to auctions characterized by different matrices of private values.

1. number of houses = number of buyers

```
v2 = np.array([[8,5,9],[4,11,7],[9,7,6]])

auction_2 = ascending_bid_auction(v2, r, ϵ)

auction_2.start_auction()
```

```
The Ascending Bid Auction for Houses


Basic Information: 3 houses, 3 buyers
The valuation matrix is as follows
+--------------+---+----+---+
| Buyer Number | 0 | 1  | 2 |
+--------------+---+----+---+
|   House 0    | 8 | 5  | 9 |
|   House 1    | 4 | 11 | 7 |
|   House 2    | 9 | 7  | 6 |
+--------------+---+----+---+
The reservation prices for houses are
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Price     | 2 | 1 | 0 |
+--------------+---+---+---+
The minimum increment of bid price is 1.00
```

(continues on next page)

(continued from previous page)

```
Auction starts successfully


Round 1
The bid information is
+--------------+-----+-----+-----+
| House Number |  0  |  1  |  2  |
+--------------+-----+-----+-----+
|    Buyer     | [2] | [1] | [0] |
+--------------+-----+-----+-----+
The bid prices for houses are
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Price     | 3 | 2 | 1 |
+--------------+---+---+---+
The winners are
[np.int64(2), np.int64(1), np.int64(0)]
The losers are
[]


The auction ends because every buyer gets one house.


Auction ends


The final result is as follows


The allocation plan is
+--------------+---------------+---------------+---------------+
| House Number |       0       |       1       |       2       |
+--------------+---------------+---------------+---------------+
|    Buyer     | [np.int64(2)] | [np.int64(1)] | [np.int64(0)] |
+--------------+---------------+---------------+---------------+
The bid prices for houses are
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Price     | 3 | 2 | 1 |
+--------------+---+---+---+
The winners are
[np.int64(2), np.int64(1), np.int64(0)]
The losers are
[]
The houses unsold are
[]
The total revenue is 6.00
```

2. multilple excess buyers

```
v3 = np.array([[8,5,9,4,3],[4,11,7,4,6],[9,7,6,4,2]])

auction_3 = ascending_bid_auction(v3, r, ϵ)
```

(continues on next page)

(continued from previous page)

```
auction_3.start_auction()
```

```
The Ascending Bid Auction for Houses


Basic Information: 3 houses, 5 buyers
The valuation matrix is as follows
+--------------+---+----+---+---+---+
| Buyer Number | 0 | 1  | 2 | 3 | 4 |
+--------------+---+----+---+---+---+
|   House 0    | 8 | 5  | 9 | 4 | 3 |
|   House 1    | 4 | 11 | 7 | 4 | 6 |
|   House 2    | 9 | 7  | 6 | 4 | 2 |
+--------------+---+----+---+---+---+
The reservation prices for houses are
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Price     | 2 | 1 | 0 |
+--------------+---+---+---+
The minimum increment of bid price is 1.00


Auction starts successfully


Round 1
The bid information is
+--------------+-----+-------+-------+
| House Number |  0  |   1   |   2   |
+--------------+-----+-------+-------+
|    Buyer     | [2] | [1 4] | [0 3] |
+--------------+-----+-------+-------+
The bid prices for houses are
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Price     | 3 | 2 | 1 |
+--------------+---+---+---+
The winners are
[np.int64(2), np.int64(4), np.int64(3)]
The losers are
[np.int64(0), np.int64(1)]


Round 2
The bid information is
+--------------+---------------+-----+-----+
| House Number |       0       |  1  |  2  |
+--------------+---------------+-----+-----+
|    Buyer     | [np.int64(2)] | [1] | [0] |
+--------------+---------------+-----+-----+
The bid prices for houses are
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
```

(continues on next page)

(continued from previous page)

```
|    Price     | 3 | 3 | 2 |
+--------------+---+---+---+
The winners are
[np.int64(2), np.int64(1), np.int64(0)]
The losers are
[np.int64(3), np.int64(4)]


Round 3
The bid information is
+--------------+---------------+-----+-----+
| House Number |       0       |  1  |  2  |
+--------------+---------------+-----+-----+
|    Buyer     | [np.int64(2)] | [4] | [3] |
+--------------+---------------+-----+-----+
The bid prices for houses are
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Price     | 3 | 4 | 3 |
+--------------+---+---+---+
The winners are
[np.int64(2), np.int64(4), np.int64(3)]
The losers are
[np.int64(0), np.int64(1)]


Round 4
The bid information is
+--------------+---------------+-----+-----+
| House Number |       0       |  1  |  2  |
+--------------+---------------+-----+-----+
|    Buyer     | [np.int64(2)] | [1] | [0] |
+--------------+---------------+-----+-----+
The bid prices for houses are
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Price     | 3 | 5 | 4 |
+--------------+---+---+---+
The winners are
[np.int64(2), np.int64(1), np.int64(0)]
The losers are
[np.int64(3), np.int64(4)]


Auction ends


The final result is as follows


The allocation plan is
+--------------+---------------+---------------+---------------+
| House Number |       0       |       1       |       2       |
+--------------+---------------+---------------+---------------+
|    Buyer     | [np.int64(2)] | [np.int64(1)] | [np.int64(0)] |
```

(continues on next page)

(continued from previous page)

```
+--------------+--------------+---------------+--------------+
The bid prices for houses are
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Price     | 3 | 5 | 4 |
+--------------+---+---+---+
The winners are
[np.int64(2), np.int64(1), np.int64(0)]
The losers are
[np.int64(3), np.int64(4)]
The houses unsold are
[]
The total revenue is 12.00
```

3. more houses than buyers

```
v4 = np.array([[8,5,4],[4,11,7],[9,7,9],[6,4,5],[2,2,2]])
r2 = np.array([2,1,0,1,1])

auction_4 = ascending_bid_auction(v4, r2, ϵ)

auction_4.start_auction()
```

```
The Ascending Bid Auction for Houses


Basic Information: 5 houses, 3 buyers
The valuation matrix is as follows
+--------------+---+----+---+
| Buyer Number | 0 | 1  | 2 |
+--------------+---+----+---+
|   House 0    | 8 | 5  | 4 |
|   House 1    | 4 | 11 | 7 |
|   House 2    | 9 | 7  | 9 |
|   House 3    | 6 | 4  | 5 |
|   House 4    | 2 | 2  | 2 |
+--------------+---+----+---+
The reservation prices for houses are
+--------------+---+---+---+---+---+
| House Number | 0 | 1 | 2 | 3 | 4 |
+--------------+---+---+---+---+---+
|    Price     | 2 | 1 | 0 | 1 | 1 |
+--------------+---+---+---+---+---+
The minimum increment of bid price is 1.00


Auction starts successfully


Round 1
The bid information is
+--------------+-----+-------+
| House Number |  1  |   2   |
+--------------+-----+-------+
|    Buyer     | [1] | [0 2] |
```

(continues on next page)

(continued from previous page)

```
+--------------+-----+-------+
The bid prices for houses are
+--------------+---+---+---+---+---+
| House Number | 0 | 1 | 2 | 3 | 4 |
+--------------+---+---+---+---+---+
|    Price     | 2 | 2 | 1 | 1 | 1 |
+--------------+---+---+---+---+---+
The winners are
[np.int64(1), np.int64(2)]
The losers are
[np.int64(0)]


Round 2
The bid information is
+--------------+---------------+-----+
| House Number |       1       |  2  |
+--------------+---------------+-----+
|    Buyer     | [np.int64(1)] | [0] |
+--------------+---------------+-----+
The bid prices for houses are
+--------------+---+---+---+---+---+
| House Number | 0 | 1 | 2 | 3 | 4 |
+--------------+---+---+---+---+---+
|    Price     | 2 | 2 | 2 | 1 | 1 |
+--------------+---+---+---+---+---+
The winners are
[np.int64(1), np.int64(0)]
The losers are
[np.int64(2)]


Round 3
The bid information is
+--------------+---------------+-----+
| House Number |       1       |  2  |
+--------------+---------------+-----+
|    Buyer     | [np.int64(1)] | [2] |
+--------------+---------------+-----+
The bid prices for houses are
+--------------+---+---+---+---+---+
| House Number | 0 | 1 | 2 | 3 | 4 |
+--------------+---+---+---+---+---+
|    Price     | 2 | 2 | 3 | 1 | 1 |
+--------------+---+---+---+---+---+
The winners are
[np.int64(1), np.int64(2)]
The losers are
[np.int64(0)]


Round 4
The bid information is
+--------------+---------------+-----+
| House Number |       1       |  2  |
+--------------+---------------+-----+
|    Buyer     | [np.int64(1)] | [0] |
```

(continues on next page)

(continued from previous page)

```
+--------------+--------------+-----+
The bid prices for houses are
+--------------+---+---+---+---+---+
| House Number | 0 | 1 | 2 | 3 | 4 |
+--------------+---+---+---+---+---+
|    Price     | 2 | 2 | 4 | 1 | 1 |
+--------------+---+---+---+---+---+
The winners are
[np.int64(1), np.int64(0)]
The losers are
[np.int64(2)]


Round 5
The bid information is
+--------------+-----+--------------+
| House Number |  1  |       2      |
+--------------+-----+--------------+
|    Buyer     | [2] | [np.int64(0)] |
+--------------+-----+--------------+
The bid prices for houses are
+--------------+---+---+---+---+---+
| House Number | 0 | 1 | 2 | 3 | 4 |
+--------------+---+---+---+---+---+
|    Price     | 2 | 3 | 4 | 1 | 1 |
+--------------+---+---+---+---+---+
The winners are
[np.int64(2), np.int64(0)]
The losers are
[np.int64(1)]


Round 6
The bid information is
+--------------+-----+--------------+
| House Number |  1  |       2      |
+--------------+-----+--------------+
|    Buyer     | [1] | [np.int64(0)] |
+--------------+-----+--------------+
The bid prices for houses are
+--------------+---+---+---+---+---+
| House Number | 0 | 1 | 2 | 3 | 4 |
+--------------+---+---+---+---+---+
|    Price     | 2 | 4 | 4 | 1 | 1 |
+--------------+---+---+---+---+---+
The winners are
[np.int64(1), np.int64(0)]
The losers are
[np.int64(2)]


Round 7
The bid information is
+--------------+--------------+-----+
| House Number |       1      |  2  |
+--------------+--------------+-----+
|    Buyer     | [np.int64(1)] | [2] |
```

(continues on next page)

(continued from previous page)

```
+--------------+--------------+-----+
The bid prices for houses are
+--------------+---+---+---+---+---+
| House Number | 0 | 1 | 2 | 3 | 4 |
+--------------+---+---+---+---+---+
|    Price     | 2 | 4 | 5 | 1 | 1 |
+--------------+---+---+---+---+---+
The winners are
[np.int64(1), np.int64(2)]
The losers are
[np.int64(0)]


Round 8
The bid information is
+--------------+--------------+---------------+-----+
| House Number |       1      |       2       |  0  |
+--------------+--------------+---------------+-----+
|    Buyer     | [np.int64(1)] | [np.int64(2)] | [0] |
+--------------+--------------+---------------+-----+
The bid prices for houses are
+--------------+---+---+---+---+---+
| House Number | 0 | 1 | 2 | 3 | 4 |
+--------------+---+---+---+---+---+
|    Price     | 3 | 4 | 5 | 1 | 1 |
+--------------+---+---+---+---+---+
The winners are
[np.int64(1), np.int64(2), np.int64(0)]
The losers are
[]


The auction ends because every buyer gets one house.


Auction ends


The final result is as follows


The allocation plan is
+--------------+--------------+---------------+---------------+
| House Number |       1      |       2       |       0       |
+--------------+--------------+---------------+---------------+
|    Buyer     | [np.int64(1)] | [np.int64(2)] | [np.int64(0)] |
+--------------+--------------+---------------+---------------+
The bid prices for houses are
+--------------+---+---+---+---+---+
| House Number | 0 | 1 | 2 | 3 | 4 |
+--------------+---+---+---+---+---+
|    Price     | 3 | 4 | 5 | 1 | 1 |
+--------------+---+---+---+---+---+
The winners are
[np.int64(1), np.int64(2), np.int64(0)]
The losers are
[]
```

(continues on next page)

(continued from previous page)

```
The houses unsold are
[np.int64(3), np.int64(4)]
The total revenue is 12.00
```

4. some houses have extremely high reservation prices

```
v5 = np.array([[8,5,4],[4,11,7],[9,7,9],[6,4,5],[2,2,2]])
r3 = np.array([10,1,0,1,1])

auction_5 = ascending_bid_auction(v5, r3, ϵ)

auction_5.start_auction()
```

```
The Ascending Bid Auction for Houses


Basic Information: 5 houses, 3 buyers
The valuation matrix is as follows
+--------------+---+----+---+
| Buyer Number | 0 | 1  | 2 |
+--------------+---+----+---+
|   House 0    | 8 | 5  | 4 |
|   House 1    | 4 | 11 | 7 |
|   House 2    | 9 | 7  | 9 |
|   House 3    | 6 | 4  | 5 |
|   House 4    | 2 | 2  | 2 |
+--------------+---+----+---+
The reservation prices for houses are
+--------------+----+---+---+---+---+
| House Number | 0  | 1 | 2 | 3 | 4 |
+--------------+----+---+---+---+---+
|    Price     | 10 | 1 | 0 | 1 | 1 |
+--------------+----+---+---+---+---+
The minimum increment of bid price is 1.00


Auction starts successfully


Round 1
The bid information is
+--------------+-----+-------+
| House Number |  1  |   2   |
+--------------+-----+-------+
|    Buyer     | [1] | [0 2] |
+--------------+-----+-------+
The bid prices for houses are
+--------------+----+---+---+---+---+
| House Number | 0  | 1 | 2 | 3 | 4 |
+--------------+----+---+---+---+---+
|    Price     | 10 | 2 | 1 | 1 | 1 |
+--------------+----+---+---+---+---+
The winners are
[np.int64(1), np.int64(0)]
The losers are
[np.int64(2)]
```

(continues on next page)

(continued from previous page)

```
Round 2
The bid information is
+--------------+---------------+-----+
| House Number |       1       |  2  |
+--------------+---------------+-----+
|    Buyer     | [np.int64(1)] | [2] |
+--------------+---------------+-----+
The bid prices for houses are
+--------------+----+---+---+---+---+
| House Number | 0  | 1 | 2 | 3 | 4 |
+--------------+----+---+---+---+---+
|    Price     | 10 | 2 | 2 | 1 | 1 |
+--------------+----+---+---+---+---+
The winners are
[np.int64(1), np.int64(2)]
The losers are
[np.int64(0)]


Round 3
The bid information is
+--------------+---------------+-----+
| House Number |       1       |  2  |
+--------------+---------------+-----+
|    Buyer     | [np.int64(1)] | [0] |
+--------------+---------------+-----+
The bid prices for houses are
+--------------+----+---+---+---+---+
| House Number | 0  | 1 | 2 | 3 | 4 |
+--------------+----+---+---+---+---+
|    Price     | 10 | 2 | 3 | 1 | 1 |
+--------------+----+---+---+---+---+
The winners are
[np.int64(1), np.int64(0)]
The losers are
[np.int64(2)]


Round 4
The bid information is
+--------------+---------------+-----+
| House Number |       1       |  2  |
+--------------+---------------+-----+
|    Buyer     | [np.int64(1)] | [2] |
+--------------+---------------+-----+
The bid prices for houses are
+--------------+----+---+---+---+---+
| House Number | 0  | 1 | 2 | 3 | 4 |
+--------------+----+---+---+---+---+
|    Price     | 10 | 2 | 4 | 1 | 1 |
+--------------+----+---+---+---+---+
The winners are
[np.int64(1), np.int64(2)]
The losers are
[np.int64(0)]
```

(continues on next page)

(continued from previous page)

```
Round 5
The bid information is
+--------------+---------------+-----+
| House Number |       1       |  2  |
+--------------+---------------+-----+
|    Buyer     | [np.int64(1)] | [0] |
+--------------+---------------+-----+
The bid prices for houses are
+--------------+----+---+---+---+---+
| House Number | 0  | 1 | 2 | 3 | 4 |
+--------------+----+---+---+---+---+
|    Price     | 10 | 2 | 5 | 1 | 1 |
+--------------+----+---+---+---+---+
The winners are
[np.int64(1), np.int64(0)]
The losers are
[np.int64(2)]


Round 6
The bid information is
+--------------+-----+---------------+
| House Number |  1  |       2       |
+--------------+-----+---------------+
|    Buyer     | [2] | [np.int64(0)] |
+--------------+-----+---------------+
The bid prices for houses are
+--------------+----+---+---+---+---+
| House Number | 0  | 1 | 2 | 3 | 4 |
+--------------+----+---+---+---+---+
|    Price     | 10 | 3 | 5 | 1 | 1 |
+--------------+----+---+---+---+---+
The winners are
[np.int64(2), np.int64(0)]
The losers are
[np.int64(1)]


Round 7
The bid information is
+--------------+-----+---------------+
| House Number |  1  |       2       |
+--------------+-----+---------------+
|    Buyer     | [1] | [np.int64(0)] |
+--------------+-----+---------------+
The bid prices for houses are
+--------------+----+---+---+---+---+
| House Number | 0  | 1 | 2 | 3 | 4 |
+--------------+----+---+---+---+---+
|    Price     | 10 | 4 | 5 | 1 | 1 |
+--------------+----+---+---+---+---+
The winners are
[np.int64(1), np.int64(0)]
The losers are
[np.int64(2)]
```

(continues on next page)

(continued from previous page)

```
Round 8
The bid information is
+--------------+---------------+-----+
| House Number |       1       |  2  |
+--------------+---------------+-----+
|    Buyer     | [np.int64(1)] | [2] |
+--------------+---------------+-----+
The bid prices for houses are
+--------------+----+---+---+---+---+
| House Number | 0  | 1 | 2 | 3 | 4 |
+--------------+----+---+---+---+---+
|    Price     | 10 | 4 | 6 | 1 | 1 |
+--------------+----+---+---+---+---+
The winners are
[np.int64(1), np.int64(2)]
The losers are
[np.int64(0)]


Round 9
The bid information is
+--------------+---------------+---------------+-----+
| House Number |       1       |       2       |  3  |
+--------------+---------------+---------------+-----+
|    Buyer     | [np.int64(1)] | [np.int64(2)] | [0] |
+--------------+---------------+---------------+-----+
The bid prices for houses are
+--------------+----+---+---+---+---+
| House Number | 0  | 1 | 2 | 3 | 4 |
+--------------+----+---+---+---+---+
|    Price     | 10 | 4 | 6 | 2 | 1 |
+--------------+----+---+---+---+---+
The winners are
[np.int64(1), np.int64(2), np.int64(0)]
The losers are
[]


The auction ends because every buyer gets one house.


Auction ends


The final result is as follows


The allocation plan is
+--------------+---------------+---------------+---------------+
| House Number |       1       |       2       |       3       |
+--------------+---------------+---------------+---------------+
|    Buyer     | [np.int64(1)] | [np.int64(2)] | [np.int64(0)] |
+--------------+---------------+---------------+---------------+
The bid prices for houses are
+--------------+----+---+---+---+---+
```

(continues on next page)

(continued from previous page)

```
| House Number | 0  | 1 | 2 | 3 | 4 |
+--------------+----+---+---+---+---+
|    Price     | 10 | 4 | 6 | 2 | 1 |
+--------------+----+---+---+---+---+
The winners are
[np.int64(1), np.int64(2), np.int64(0)]
The losers are
[]
The houses unsold are
[np.int64(0), np.int64(4)]
The total revenue is 12.00
```

5. reservation prices are so high that the auction can't start

```
r4 = np.array([15,15,15])

auction_6 = ascending_bid_auction(v, r4, ϵ)

auction_6.start_auction()
```

```
The Ascending Bid Auction for Houses


Basic Information: 3 houses, 4 buyers
The valuation matrix is as follows
+--------------+---+----+---+---+
| Buyer Number | 0 | 1  | 2 | 3 |
+--------------+---+----+---+---+
|   House 0    | 8 | 5  | 9 | 4 |
|   House 1    | 4 | 11 | 7 | 4 |
|   House 2    | 9 | 7  | 6 | 4 |
+--------------+---+----+---+---+
The reservation prices for houses are
+--------------+----+----+----+
| House Number | 0  | 1  | 2  |
+--------------+----+----+----+
|    Price     | 15 | 15 | 15 |
+--------------+----+----+----+
The minimum increment of bid price is 1.00


The auction can not start because of high reservation prices
```

135.10 A Groves-Clarke Mechanism

We now decribe an alternative way for society to allocate n houses to m possible buyers in a way that maximizes total value across all potential buyers.

We continue to assume that each buyer can purchase at most one house.

The mechanism is a very special case of a Groves-Clarke mechanism [Groves, 1973], [Clarke, 1971].

Its special structure substantially simplifies writing Python code to find an optimal allocation.

Our mechanims works like this.

- The values V_{ij} are private information to person j
- The mechanism makes each person j willing to tell a social planner his private values $V_{i,j}$ for all $i = 1, \ldots, n$.
- The social planner asks all potential bidders to tell the planner their private values V_{ij}
- The social planner tells no one these, but uses them to allocate houses and set prices
- The mechanism is designed in a way that makes all prospective buyers want to tell the planner their private values
  - truth telling is a dominant strategy for each potential buyer
- The planner finds a house, bidder pair with highest private value by computing $(\tilde{i}, \tilde{j}) = \text{argmax}(V_{ij})$
- The planner assigns house $\tilde{i}$ to buyer $\tilde{j}$
- The planner charges buyer $\tilde{j}$ the price $\max_{-\tilde{j}} V_{\tilde{i},j}$, where $-\tilde{j}$ means all j's except $\tilde{j}$.
- The planner creates a matrix of private values for the remaining houses $-\tilde{i}$ by deleting row (i.e., house) $\tilde{i}$ and column (i.e., buyer) $\tilde{j}$ from V.
  - (But in doing this, the planner keeps track of the real names of the bidders and the houses).
- The planner returns to the original step and repeat it.
- The planner iterates until all n houses are allocated and the charges for all n houses are set.

135.11 An Example Solved by Hand

Let's see how our Groves-Clarke algorithm would work for the following simple matrix V matrix of private values

$$V = \begin{bmatrix} 10 & 9 & 8 & 7 & 6 \\ 9 & 9 & 7 & 6 & 6 \\ 8 & 6 & 6 & 9 & 4 \\ 7 & 5 & 6 & 4 & 9 \end{bmatrix}$$

Remark: In the first step, when the highest private value corresponds to more than one house, bidder pairs, we choose the pair with the highest sale price. If a highest sale price corresponds to two or more pairs with highest private values, we randomly choose one.

```
np.random.seed(666)

V_orig = np.array([[10, 9, 8, 7, 6],  # record the origianl values
                   [9, 9, 7, 6, 6],
                   [8, 6, 6, 9, 4],
                   [7, 5, 6, 4, 9]])
V = np.copy(V_orig)  # used iteratively
n, m = V.shape
p = np.zeros(n) # prices of houses
Q = np.zeros((n, m)) # keep record the status of houses and buyers
```

First assignment

First, we find house, bidder pair with highest private value.

```
i, j = np.where(V==np.max(V))
i, j
```

```
(array([0]), array([0]))
```

So, house 0 will be sold to buyer 0 at a price of 9. We then update the sale price of house 0 and the status matrix Q.

```
p[i] = np.max(np.delete(V[i, :], j))
Q[i, j] = 1
p, Q
```

```
(array([9., 0., 0., 0.]),
 array([[1., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.]]))
```

Then we remove row 0 and column 0 from V. To keep the real number of houses and buyers, we set this row and this column to -1, which will have the same result as removing them since $V \geq 0$.

```
V[i, :] = -1
V[:, j] = -1
V
```

```
array([[-1, -1, -1, -1, -1],
       [-1,  9,  7,  6,  6],
       [-1,  6,  6,  9,  4],
       [-1,  5,  6,  4,  9]])
```

Second assignment

We find house, bidder pair with the highest private value again.

```
i, j = np.where(V==np.max(V))
i, j
```

```
(array([1, 2, 3]), array([1, 3, 4]))
```

In this special example, there are three pairs (1, 1), (2, 3) and (3, 4) with the highest private value. To solve this problem, we choose the one with highest sale price.

```
p_candidate = np.zeros(len(i))
for k in range(len(i)):
    p_candidate[k] = np.max(np.delete(V[i[k], :], j[k]))
k, = np.where(p_candidate==np.max(p_candidate))
i, j = i[k], j[k]
i, j
```

```
(array([1]), array([1]))
```

So, house 1 will be sold to buyer 1 at a price of 7. We update matrices.

```
p[i] = np.max(np.delete(V[i, :], j))
Q[i, j] = 1
V[i, :] = -1
V[:, j] = -1
p, Q, V
```

```
(array([9., 7., 0., 0.]),
 array([[1., 0., 0., 0., 0.],
        [0., 1., 0., 0., 0.],
```

(continues on next page)

(continued from previous page)

```
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.]]),
 array([[-1, -1, -1, -1, -1],
        [-1, -1, -1, -1, -1],
        [-1, -1,  6,  9,  4],
        [-1, -1,  6,  4,  9]]))
```

Third assignment

```
i, j = np.where(V==np.max(V))
i, j
```

```
(array([2, 3]), array([3, 4]))
```

In this special example, there are two pairs (2, 3) and (3, 4) with the highest private value.

To resolve the assignment, we choose the one with highest sale price.

```
p_candidate = np.zeros(len(i))
for k in range(len(i)):
    p_candidate[k] = np.max(np.delete(V[i[k], :], j[k]))
k, = np.where(p_candidate==np.max(p_candidate))
i, j = i[k], j[k]
i, j
```

```
(array([2, 3]), array([3, 4]))
```

The two pairs even have the same sale price.

We randomly choose one pair.

```
k = np.random.choice(len(i))
i, j = i[k], j[k]
i, j
```

```
(np.int64(2), np.int64(3))
```

Finally, house 2 will be sold to buyer 3.

We update matrices accordingly.

```
p[i] = np.max(np.delete(V[i, :], j))
Q[i, j] = 1
V[i, :] = -1
V[:, j] = -1
p, Q, V
```

```
(array([9., 7., 6., 0.]),
 array([[1., 0., 0., 0., 0.],
        [0., 1., 0., 0., 0.],
        [0., 0., 0., 1., 0.],
        [0., 0., 0., 0., 0.]]),
 array([[-1, -1, -1, -1, -1],
        [-1, -1, -1, -1, -1],
        [-1, -1, -1, -1, -1],
        [-1, -1,  6, -1,  9]]))
```

Fourth assignment

```
i, j = np.where(V==np.max(V))
i, j
```

```
(array([3]), array([4]))
```

House 3 will be sold to buyer 4.

The final outcome follows.

```
p[i] = np.max(np.delete(V[i, :], j))
Q[i, j] = 1
V[i, :] = -1
V[:, j] = -1
S = V_orig*Q - np.diag(p)@Q
p, Q, V, S
```

```
(array([9., 7., 6., 6.]),
 array([[1., 0., 0., 0., 0.],
        [0., 1., 0., 0., 0.],
        [0., 0., 0., 1., 0.],
        [0., 0., 0., 0., 1.]]),
 array([[-1, -1, -1, -1, -1],
        [-1, -1, -1, -1, -1],
        [-1, -1, -1, -1, -1],
        [-1, -1, -1, -1, -1]]),
 array([[1., 0., 0., 0., 0.],
        [0., 2., 0., 0., 0.],
        [0., 0., 0., 3., 0.],
        [0., 0., 0., 0., 3.]]))
```

135.12 Another Python Class

It is efficient to assemble our calculations in a single Python Class.

```
class GC_Mechanism:

    def __init__(self, V):
        """
        Implementation of the special Groves Clarke Mechanism for house auction.

        Parameters:
        ----------
        V: 2 dimensional private value matrix

        """

        self.V_orig = V.copy()
        self.V = V.copy()
        self.n, self.m = self.V.shape
        self.p = np.zeros(self.n)
        self.Q = np.zeros((self.n, self.m))
        self.S = np.copy(self.Q)
```

(continues on next page)

(continued from previous page)

```
    def find_argmax(self):
        """
        Find the house-buyer pair with the highest value.
        When the highest private value corresponds to more than one house, bidder␣
↪pairs,
        we choose the pair with the highest sale price.
        Moreoever, if the highest sale price corresponds to two or more pairs with␣
↪highest private value,
        We randomly choose one.

        Parameters:
        ----------
        V: 2 dimensional private value matrix with -1 indicating revomed rows and␣
↪columns

        Returns:
        ----------
        i: the index of the sold house

        j: the index of the buyer

        """
        i, j = np.where(self.V==np.max(self.V))

        if (len(i)>1):
            p_candidate = np.zeros(len(i))
            for k in range(len(i)):
                p_candidate[k] = np.max(np.delete(self.V[i[k], :], j[k]))
            k, = np.where(p_candidate==np.max(p_candidate))
            i, j = i[k], j[k]

            if (len(i)>1):
                k = np.random.choice(len(i))
                k = np.array([k])
                i, j = i[k], j[k]
        return i, j

    def update_status(self, i, j):
        self.p[i] = np.max(np.delete(self.V[i, :], j))
        self.Q[i, j] = 1
        self.V[i, :] = -1
        self.V[:, j] = -1

    def calculate_surplus(self):
        self.S = self.V_orig*self.Q - np.diag(self.p)@self.Q

    def start(self):
        while (np.max(self.V)>=0):
            i, j = self.find_argmax()
            self.update_status(i, j)
            print("House %i is sold to buyer %i at price %i"%(i[0], j[0], self.
↪p[i[0]]))
            print("\n")
        self.calculate_surplus()
        print("Prices of house:\n", self.p)
        print("\n")
        print("The status matrix:\n", self.Q)
```

(continues on next page)

(continued from previous page)

```
        print("\n")
        print("The surplus matrix:\n", self.S)
```

```
np.random.seed(666)

V_orig = np.array([[10, 9, 8, 7, 6],
                   [9, 9, 7, 6, 6],
                   [8, 6, 6, 9, 4],
                   [7, 5, 6, 4, 9]])
gc_mechanism = GC_Mechanism(V_orig)
gc_mechanism.start()
```

```
House 0 is sold to buyer 0 at price 9


House 1 is sold to buyer 1 at price 7


House 2 is sold to buyer 3 at price 6


House 3 is sold to buyer 4 at price 6


Prices of house:
 [9. 7. 6. 6.]


The status matrix:
 [[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 1.]]


The surplus matrix:
 [[1. 0. 0. 0. 0.]
 [0. 2. 0. 0. 0.]
 [0. 0. 0. 3. 0.]
 [0. 0. 0. 0. 3.]]
```

135.12.1 Elaborations

Here we use some additional notation designed to conform with standard notation in parts of the VCG literature.

We want to verify that our pseudo code is indeed a **pivot mechanism**, also called a **VCG** (Vickrey-Clarke-Groves) mechanism.

- The mechanism is named after [Groves, 1973], [Clarke, 1971], and [Vickrey, 1961].

To prepare for verifying this, we add some notation.

Let X be the set of feasible allocations of houses under the protocols above (i.e., at most one house to each person).

Let $X(v)$ be the allocation that the mechanism chooses for matrix v of private values.

The mechanism maps a matrix v of private values into an $x \in X$.

Let $v_j(x)$ be the value that person j attaches to allocation $x \in X$.

Let $\check{t}_j(v)$ the payment that the mechanism charges person j.

The VCG mechanism chooses the allocation

$$X(v) = \text{argmax}_{x \in X} \sum_{j=1}^{m} v_j(x) \tag{135.1}$$

and charges person j a "social cost"

$$\check{t}_j(v) = \max_{x \in X} \sum_{k \neq j} v_k(x) - \sum_{k \neq j} v_k(X(v)) \tag{135.2}$$

In our setting, equation (135.1) says that the VCG allocation allocates houses to maximize the total value of the successful prospective buyers.

In our setting, equation (135.2) says that the mechanism charges people for the externality that their presence in society imposes on other prospective buyers.

Thus, notice that according to equation (135.2):

- unsuccessful prospective buyers pay 0 because removing them from "society" would not affect the allocation chosen by the mechanim
- successful prospective buyers pay the difference between the total value society could achieve without them present and the total value that others present in society do achieve under the mechanism.

The generalized second-price auction described in our pseudo code above does indeed satisfy (1). We want to compute $\check{t}_j$ for $j = 1, \ldots, m$ and compare with p_j from the second price auction.

135.12.2 Social Cost

Using the GC_Mechanism class, we can calculate the social cost of each buyer.

Let's see a simpler example with private value matrix

$$V = \begin{bmatrix} 10 & 9 & 8 & 7 & 6 \\ 9 & 8 & 7 & 6 & 6 \\ 8 & 7 & 6 & 5 & 4 \end{bmatrix}$$

To begin with, we implement the GC mechanism and see the outcome.

```
np.random.seed(666)

V_orig = np.array([[10, 9, 8, 7, 6],
                   [9, 8, 7, 6, 6],
                   [8, 7, 6, 5, 4]])
gc_mechanism = GC_Mechanism(V_orig)
gc_mechanism.start()
```

```
House 0 is sold to buyer 0 at price 9


House 1 is sold to buyer 1 at price 7


House 2 is sold to buyer 2 at price 5
```

(continues on next page)

(continued from previous page)

```
Prices of house:
 [9. 7. 5.]


The status matrix:
 [[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]]


The surplus matrix:
 [[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]]
```

We exclude buyer 0 and calculate the allocation.

```
V_exc_0 = np.copy(V_orig)
V_exc_0[:, 0] = -1
V_exc_0
gc_mechanism_exc_0 = GC_Mechanism(V_exc_0)
gc_mechanism_exc_0.start()
```

```
House 0 is sold to buyer 1 at price 8


House 1 is sold to buyer 2 at price 6


House 2 is sold to buyer 3 at price 4


Prices of house:
 [8. 6. 4.]


The status matrix:
 [[0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0.]]


The surplus matrix:
 [[-0.  1.  0.  0.  0.]
 [-0.  0.  1.  0.  0.]
 [-0.  0.  0.  1.  0.]]
```

Calculate the social cost of buyer 0.

```
print("The social cost of buyer 0:",
     np.sum(gc_mechanism_exc_0.Q*gc_mechanism_exc_0.V_orig)-np.sum(np.delete(gc_
 ↪mechanism.Q*gc_mechanism.V_orig, 0, axis=1)))
```

```
The social cost of buyer 0: 7.0
```

Repeat this process for buyer 1 and buyer 2

```
V_exc_1 = np.copy(V_orig)
V_exc_1[:, 1] = -1
V_exc_1
gc_mechanism_exc_1 = GC_Mechanism(V_exc_1)
gc_mechanism_exc_1.start()

print("\nThe social cost of buyer 1:",
     np.sum(gc_mechanism_exc_1.Q*gc_mechanism_exc_1.V_orig)-np.sum(np.delete(gc_
 ↪mechanism.Q*gc_mechanism.V_orig, 1, axis=1)))
```

```
House 0 is sold to buyer 0 at price 8


House 1 is sold to buyer 2 at price 6


House 2 is sold to buyer 3 at price 4


Prices of house:
 [8. 6. 4.]


The status matrix:
 [[1. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0.]]


The surplus matrix:
 [[ 2. -0.  0.  0.  0.]
 [ 0. -0.  1.  0.  0.]
 [ 0. -0.  0.  1.  0.]]

The social cost of buyer 1: 6.0
```

```
V_exc_2 = np.copy(V_orig)
V_exc_2[:, 2] = -1
V_exc_2
gc_mechanism_exc_2 = GC_Mechanism(V_exc_2)
gc_mechanism_exc_2.start()

print("\nThe social cost of buyer 2:",
     np.sum(gc_mechanism_exc_2.Q*gc_mechanism_exc_2.V_orig)-np.sum(np.delete(gc_
 ↪mechanism.Q*gc_mechanism.V_orig, 2, axis=1)))
```

```
House 0 is sold to buyer 0 at price 9


House 1 is sold to buyer 1 at price 6
```

(continues on next page)

(continued from previous page)

```
House 2 is sold to buyer 3 at price 4


Prices of house:
 [9. 6. 4.]


The status matrix:
 [[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 0. 1. 0.]]


The surplus matrix:
 [[ 1.  0. -0.  0.  0.]
 [ 0.  2. -0.  0.  0.]
 [ 0.  0. -0.  1.  0.]]

The social cost of buyer 2: 5.0
```

Part XIX

Other

CHAPTER

SIX

TROUBLESHOOTING

Contents

- *Troubleshooting*
  - *Fixing Your Local Environment*
  - *Reporting an Issue*

This page is for readers experiencing errors when running the code from the lectures.

136.1 Fixing Your Local Environment

The basic assumption of the lectures is that code in a lecture should execute whenever

1. it is executed in a Jupyter notebook and
2. the notebook is running on a machine with the latest version of Anaconda Python.

You have installed Anaconda, haven't you, following the instructions in this lecture?

Assuming that you have, the most common source of problems for our readers is that their Anaconda distribution is not up to date.

Here's a useful article on how to update Anaconda.

Another option is to simply remove Anaconda and reinstall.

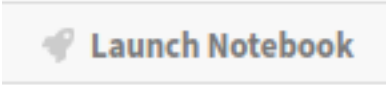
You also need to keep the external code libraries, such as QuantEcon.py up to date.

For this task you can either

- use `pip install --upgrade quantecon` on the command line, or
- execute `!pip install --upgrade quantecon` within a Jupyter notebook.

If your local environment is still not working you can do two things.

First, you can use a remote machine instead, by clicking on the Launch Notebook icon available for each lecture



Second, you can report an issue, so we can try to fix your local set up.

We like getting feedback on the lectures so please don't hesitate to get in touch.

136.2 Reporting an Issue

One way to give feedback is to raise an issue through our issue tracker.

Please be as specific as possible. Tell us where the problem is and as much detail about your local set up as you can provide.

Finally, you can provide direct feedback to contact@quantecon.org

CHAPTER

SEVEN

REFERENCES

CHAPTER

EIGHT

EXECUTION STATISTICS

This table contains the latest execution statistics.

Document	Modified	Method	Run Time (s)	Status
affine_risk_prices	2026-07-20 04:24	cache	7.72	✓
aiyagari	2026-07-20 04:24	cache	17.9	✓
aiyagari_egm	2026-07-20 04:25	cache	17.38	✓
ak2	2026-07-20 04:25	cache	12.51	✓
ak_aiyagari	2026-07-20 04:25	cache	24.46	✓
ar1_bayes	2026-07-20 04:30	cache	259.89	✓
ar1_turningpts	2026-07-20 04:34	cache	254.14	✓
back_prop	2026-07-20 04:35	cache	98.47	✓
bayes_nonconj	2026-07-20 04:36	cache	49.47	✓
blackwell_kihlstrom	2026-07-20 04:36	cache	6.38	✓
bounded_rationality	2026-07-26 05:58	cache	34.42	✓
career	2026-07-20 04:37	cache	9.79	✓
cass_fiscal	2026-07-20 04:37	cache	17.77	✓
cass_fiscal_2	2026-07-20 04:37	cache	5.63	✓
cass_koopmans_1	2026-07-20 04:37	cache	13.76	✓
cass_koopmans_2	2026-07-20 04:37	cache	6.63	✓
chow_business_cycles	2026-07-20 04:37	cache	9.19	✓
cross_product_trick	2026-07-20 04:37	cache	1.14	✓
divergence_measures	2026-07-20 04:38	cache	12.85	✓
eig_circulant	2026-07-20 04:38	cache	2.85	✓
endogenous_lake	2026-07-20 04:46	cache	495.33	✓
exchange_rate_learning	2026-07-26 05:59	cache	62.0	✓
exchangeable	2026-07-20 04:46	cache	7.81	✓
finite_markov	2026-07-20 04:46	cache	7.59	✓
ge_arrow	2026-07-20 04:46	cache	1.93	✓
genetic_classifier	2026-07-26 06:00	cache	52.74	✓
hansen_singleton_1982	2026-07-20 04:47	cache	13.72	✓
hansen_singleton_1983	2026-07-20 04:47	cache	24.45	✓
harrison_kreps	2026-07-20 04:47	cache	3.49	✓
hoist_failure	2026-07-20 04:48	cache	49.36	✓
house_auction	2026-07-20 04:48	cache	2.87	✓
ifp_advanced	2026-07-20 04:48	cache	25.62	✓
ifp_discrete	2026-07-20 04:49	cache	12.54	✓
ifp_egm	2026-07-20 04:49	cache	10.07	✓
ifp_egm_transient_shocks	2026-07-20 04:49	cache	17.05	✓
ifp_opi	2026-07-20 04:49	cache	8.5	✓

continues on next page

Table 138.1 – continued from previous page

Document	Modified	Method	Run Time (s)	Status
imp_sample	2026-07-20 04:53	cache	247.97	✅
information_market_equilibrium	2026-07-20 04:54	cache	62.25	✅
intro	2026-07-20 04:54	cache	0.94	✅
inventory_dynamics	2026-07-26 06:00	cache	14.89	✅
inventory_q	2026-07-20 04:55	cache	6.49	✅
jv	2026-07-20 04:55	cache	10.83	✅
kalman	2026-07-20 04:55	cache	7.76	✅
kalman_2	2026-07-20 04:55	cache	32.37	✅
kalman_filter_var	2026-07-20 04:56	cache	8.58	✅
kesten_processes	2026-07-20 04:56	cache	9.74	✅
lagrangian_lqdp	2026-07-20 04:56	cache	25.61	✅
lake_model	2026-07-20 04:56	cache	10.36	✅
learning_approximation	2026-07-26 06:00	cache	1.45	❌ (CellExecutionError)
likelihood_bayes	2026-07-20 04:57	cache	44.58	✅
likelihood_ratio_process	2026-07-20 04:57	cache	22.71	✅
likelihood_ratio_process_2	2026-07-20 04:58	cache	30.95	✅
likelihood_var	2026-07-20 04:58	cache	21.56	✅
linear_algebra	2026-07-20 04:58	cache	2.48	✅
linear_models	2026-07-20 04:59	cache	7.81	✅
lln_clt	2026-07-20 04:59	cache	10.55	✅
long_run_risk_operator	2026-07-20 04:59	cache	18.51	✅
lq_bewley_complete_markets	2026-07-20 04:59	cache	4.44	✅
lq_inventories	2026-07-20 04:59	cache	13.01	✅
lq_permanent_income	2026-07-20 04:59	cache	3.86	✅
lq_robust_smoothing	2026-07-20 04:59	cache	4.14	✅
lqcontrol	2026-07-26 06:00	cache	7.0	✅
ls_learning	2026-07-20 05:00	cache	9.96	✅
marimon_mcgrattan_sargent	2026-07-26 06:07	cache	406.72	✅
markov_asset	2026-07-20 05:00	cache	5.72	✅
markov_perf	2026-07-20 05:00	cache	4.42	✅
mccall_fitted_vfi	2026-07-20 05:01	cache	45.36	✅
mccall_model	2026-07-20 05:01	cache	13.16	✅
mccall_model_with_sep_markov	2026-07-20 05:01	cache	26.13	✅
mccall_model_with_separation	2026-07-20 05:02	cache	15.32	✅
mccall_persist_trans	2026-07-20 05:02	cache	11.2	✅
mccall_q	2026-07-20 05:02	cache	15.78	✅
measurement_models	2026-07-20 05:02	cache	3.54	✅
merging_of_opinions	2026-07-20 05:05	cache	165.4	✅
misspecified_recovery	2026-07-20 05:05	cache	28.22	✅
mix_model	2026-07-20 05:07	cache	99.27	✅
mle	2026-07-20 05:07	cache	10.95	✅
morris_learn	2026-07-20 05:07	cache	15.88	✅
multi_hyper	2026-07-20 05:08	cache	29.75	✅
multivariate_normal	2026-07-20 05:08	cache	5.66	✅
navy_captain	2026-07-20 05:09	cache	27.26	✅
newton_method	2026-07-20 05:09	cache	52.63	✅
odu	2026-07-20 05:10	cache	48.99	✅
olg_adaptive_money	2026-07-26 06:08	cache	41.39	✅
ols	2026-07-20 05:10	cache	7.51	✅
opt_transport	2026-07-20 05:11	cache	13.69	✅

continues on next page

Table 138.1 – continued from previous page

Document	Modified	Method	Run Time (s)	Status
organization_capital	2026-07-20 05:11	cache	5.45	✅
os	2026-07-20 05:11	cache	2.0	✅
os_egm	2026-07-20 05:11	cache	3.77	✅
os_egm_jax	2026-07-20 05:11	cache	6.93	✅
os_numerical	2026-07-20 05:11	cache	14.29	✅
os_stochastic	2026-07-20 05:12	cache	58.21	✅
os_time_iter	2026-07-20 05:12	cache	5.06	✅
pandas_panel	2026-07-20 05:12	cache	5.37	✅
perm_income	2026-07-26 06:08	cache	3.87	✅
perm_income_cons	2026-07-20 05:12	cache	7.38	✅
phillips_adaptive	2026-07-26 06:08	cache	4.54	✅
phillips_credibility	2026-07-26 06:08	cache	2.25	✅
phillips_credible_policies	2026-07-26 06:08	cache	3.93	✅
phillips_drifts_volatilities	2026-07-26 06:29	cache	1283.08	✅
phillips_escaping_nash	2026-07-26 06:29	cache	3.36	✅
phillips_learning	2026-07-26 06:30	cache	27.54	✅
phillips_lost_conquest	2026-07-26 06:30	cache	6.58	✅
phillips_misspecified	2026-07-26 06:30	cache	2.57	✅
phillips_priors	2026-07-26 06:30	cache	8.05	✅
phillips_self_confirming	2026-07-26 06:30	cache	5.29	✅
phillips_two_stories	2026-07-26 06:30	cache	5.95	✅
prob_matrix	2026-07-20 05:13	cache	10.66	✅
prob_meaning	2026-07-20 05:13	cache	38.56	✅
prospects_bounded_rationality	2026-07-26 06:30	cache	4.57	✅
qr_decomp	2026-07-20 05:13	cache	1.37	✅
rand_resp	2026-07-20 05:13	cache	2.45	✅
rational_expectations	2026-07-20 05:13	cache	3.56	✅
rational_learning_re	2026-07-20 05:14	cache	5.49	✅
re_with_feedback	2026-07-20 05:14	cache	10.11	✅
robust_permanent_income	2026-07-20 05:14	cache	6.59	✅
ross_recovery	2026-07-20 05:14	cache	6.11	✅
rs_inventory_q	2026-07-20 05:14	cache	7.7	✅
samuelson	2026-07-20 05:14	cache	10.63	✅
sir_model	2026-07-20 05:14	cache	3.11	✅
stats_examples	2026-07-20 05:14	cache	4.23	✅
status	2026-07-20 05:14	cache	8.3	✅
survival_recursive_preferences	2026-07-20 05:15	cache	5.36	✅
svd_intro	2026-07-20 05:15	cache	1.4	✅
theil_1	2026-07-20 05:15	cache	4.28	✅
theil_2	2026-07-26 06:31	cache	5.28	✅
troubleshooting	2026-07-20 04:54	cache	0.94	✅
two_auctions	2026-07-20 05:15	cache	20.8	✅
two_computation	2026-07-20 05:16	cache	47.02	✅
uncertainty_traps	2026-07-20 05:16	cache	2.6	✅
unemployment_linear	2026-07-20 05:16	cache	31.18	✅
unemployment_shocks	2026-07-20 05:17	cache	55.06	✅
util_rand_resp	2026-07-20 05:17	cache	2.9	✅
var_dmd	2026-07-20 04:54	cache	0.94	✅
von_neumann_model	2026-07-20 05:17	cache	2.99	✅
wald_friedman	2026-07-20 05:18	cache	20.09	✅

continues on next page

Table 138.1 – continued from previous page

Document	Modified	Method	Run Time (s)	Status
wald_friedman_2	2026-07-20 05:18	cache	12.87	✅
wealth_dynamics	2026-07-26 06:31	cache	32.62	✅
zreferences	2026-07-20 04:54	cache	0.94	✅

These lectures are built on `linux` instances through `github actions`.

These lectures are using the following python version

```
!python --version
```

```
Python 3.13.14
```

and the following package versions

```
!conda list
```

This lecture series has access to the following GPU

```
!nvidia-smi
```

```
Mon Jul 20 05:14:56 2026
+-----------------------------------------------------------------------------------------
 ↪-------+
| NVIDIA-SMI 580.105.08             Driver Version: 580.105.08     CUDA Version:␣
 ↪13.0     |
+-----------------------------------------+------------------------+----------------
 ↪-------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile␣
 ↪Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util ␣
 ↪Compute M. |
|                                         |                        |              ␣
 ↪MIG M. |
|=========================================+========================+======================|
|   0  Tesla T4                       On  |   00000000:00:1E.0 Off |                ␣
 ↪      0 |
| N/A   31C    P8             13W /   70W |       0MiB /  15360MiB |      0%      ␣
 ↪Default |
|                                         |                        |              ␣
 ↪   N/A |
+-----------------------------------------+------------------------+----------------
 ↪-------+

+-----------------------------------------------------------------------------------------
 ↪-------+
| Processes:                                                                            ␣
 ↪       |
|  GPU   GI   CI              PID   Type   Process name                            GPU␣
 ↪Memory |
|        ID   ID                                                                     ␣
 ↪Usage      |
|=========================================================================================|
|  No running processes found                                                           ␣
 ↪       |
```

(continues on next page)

(continued from previous page)

```
+---------------------------------------------------------------------------------
 ↪--------+
```

You can check the backend used by JAX using:

```
import jax
# Check if JAX is using GPU
print(f"JAX backend: {jax.devices()[0].platform}")
```

```
JAX backend: gpu
```

BIBLIOGRAPHY

[Imrohorouglu93] Selahattin İmrohoroğlu. Testing for sunspot equilibria in the German hyperinflation. *Journal of Economic Dynamics and Control*, 17(1-2):289–317, 1993. doi:10.1016/S0165-1889(06)80013-0.

[Abr88] Dilip Abreu. On the theory of infinitely repeated games with discounting. *Econometrica*, 56:383–396, 1988.

[APS90] Dilip Abreu, David Pearce, and Ennio Stacchetti. Toward a theory of discounted repeated games with imperfect monitoring. *Econometrica*, 58(5):1041–1063, September 1990.

[ACY16] Daron Acemoglu, Victor Chernozhukov, and Muhamet Yildiz. Fragility of asymptotic agreement under Bayesian learning. *Theoretical Economics*, 11(1):187–225, 2016. doi:10.3982/TE436.

[AJR01] Daron Acemoglu, Simon Johnson, and James A Robinson. The colonial origins of comparative development: an empirical investigation. *The American Economic Review*, 91(5):1369–1401, 2001.

[Aiy94] S Rao Aiyagari. Uninsured Idiosyncratic Risk and Aggregate Saving. *The Quarterly Journal of Economics*, 109(3):659–684, 1994.

[Alc50] Armen A Alchian. Uncertainty, evolution, and economic theory. *Journal of political economy*, 58(3):211–221, 1950.

[AS66] S. M. Ali and S. D. Silvey. A general class of coefficients of divergence of one distribution from another. *Journal of the Royal Statistical Society, Series B*, 28(1):131–142, 1966.

[AJ05] Fernando Alvarez and Urban J. Jermann. Using asset prices to measure the persistence of the marginal utility of wealth. *Econometrica*, 73(6):1977–2016, 2005. doi:10.1111/j.1468-0262.2005.00643.x.

[AN00] Shun-ichi Amari and Hiroshi Nagaoka. *Methods of Information Geometry*. Volume 191 of Translations of Mathematical Monographs. American Mathematical Society and Oxford University Press, Providence, RI, 2000.

[AM05] D. B. O. Anderson and J. B. Moore. *Optimal Filtering*. Dover Publications, 2005.

[AHMS96] E. W. Anderson, L. P. Hansen, E. R. McGrattan, and T. J. Sargent. Mechanics of Forming and Estimating Dynamic Linear Economies. In *Handbook of Computational Economics*. Elsevier, vol 1 edition, 1996.

[AHS03] Evan W. Anderson, Lars Peter Hansen, and Thomas J. Sargent. A Quartet of Semigroups for Model Specification, Robustness, Prices of Risk, and Model Detection. *Journal of the European Economic Association*, 1(1):68–123, March 2003. URL: https://ideas.repec.org/a/tpr/jeurec/v1y2003i1p68-123.html, doi:.

[And76] Harald Anderson. Estimation of a proportion through randomized response. *International Statistical Review/Revue Internationale de Statistique*, pages 213–217, 1976.

[AP03] Andrew Ang and Monika Piazzesi. A no-arbitrage vector autoregression of term structure dynamics with macroeconomic and latent variables. *Journal of Monetary Economics*, 50(4):745–787, 2003.

[Apo90] George Apostolakis. The concept of probability in safety assessments of technological systems. *Science*, 250(4986):1359–1364, 1990.

[Ari96] Jasmina Arifovic. The behavior of the exchange rate in the genetic algorithm and experimental economies. *Journal of Political Economy*, 104(3):510–541, 1996. doi:10.1086/262032.

[AK87] Alan J Auerbach and Laurence J Kotlikoff. *Dynamic fiscal policy*. Cambridge University Press, Cambridge, 1987.

[Aum76] Robert J. Aumann. Agreeing to disagree. *Annals of Statistics*, 4(6):1236–1239, 1976.

[Axe87] Robert Axelrod. The evolution of strategies in the iterated prisoner's dilemma. In Lawrence Davis, editor, *Genetic Algorithms and Simulated Annealing*, pages 32–41. Morgan Kaufmann, Los Altos, CA, 1987.

[Axt01] Robert L Axtell. Zipf distribution of us firm sizes. *science*, 293(5536):1818–1820, 2001.

[BY04] Ravi Bansal and Amir Yaron. Risks for the long run: a potential resolution of asset pricing puzzles. *Journal of Finance*, 59(4):1481–1509, 2004. doi:10.1111/j.1540-6261.2004.00670.x.

[BHS09] Francisco Barillas, Lars Peter Hansen, and Thomas J. Sargent. Doubts or variability? *Journal of Economic Theory*, 144(6):2388–2418, November 2009. URL: https://ideas.repec.org/a/eee/jetheo/v144y2009i6p2388-2418.html, doi:.

[Bar79] Robert J Barro. On the Determination of the Public Debt. *Journal of Political Economy*, 87(5):940–971, 1979.

[Bar06] Robert J Barro. Rare disasters and asset markets in the twentieth century. *The Quarterly Journal of Economics*, 121(3):823–866, 2006.

[BG83] Robert J. Barro and David B. Gordon. Rules, discretion and reputation in a model of monetary policy. *Journal of Monetary Economics*, 12(1):101–121, 1983.

[BK99] Marianne Baxter and Robert G. King. Measuring business cycles: approximate band-pass filters for economic time series. *Review of Economics and Statistics*, 81(4):575–593, 1999.

[BacsarB08] Tamer Başar and Pierre Bernhard. *H-infinity optimal control and related minimax design problems: a dynamic game approach*. Springer Science & Business Media, 2008.

[Bec75] Gary S. Becker. *Human Capital: A Theoretical and Empirical Analysis, with Special Reference to Education*. National Bureau of Economic Research, New York, 2nd edition, 1975.

[BB18] Jess Benhabib and Alberto Bisin. Skewed wealth distributions: theory and empirics. *Journal of Economic Literature*, 56(4):1261–91, 2018.

[BBZ15] Jess Benhabib, Alberto Bisin, and Shenghao Zhu. The wealth distribution in bewley economies with capital income risk. *Journal of Economic Theory*, 159:489–515, 2015.

[BMetivierP90] Albert Benveniste, Michel Métivier, and Pierre Priouret. *Adaptive Algorithms and Stochastic Approximations*. Springer-Verlag, Berlin, 1990.

[BS79] L M Benveniste and J A Scheinkman. On the Differentiability of the Value Function in Dynamic Models of Economics. *Econometrica*, 47(3):727–732, 1979.

[Ber22] Ben S. Bernanke. *21st Century Monetary Policy: The Federal Reserve from the Great Inflation to COVID-19*. W. W. Norton and Company, New York, 2022.

[BM98] Ben S. Bernanke and Ilian Mihov. Measuring monetary policy. *Quarterly Journal of Economics*, 113(3):869–902, 1998.

[Ber76] Dimitri P. Bertsekas. *Dynamic Programming and Stochastic Control*. Academic Press, New York, 1976.

[Bew77] Truman Bewley. The permanent income hypothesis: a theoretical formulation. *Journal of Economic Theory*, 16(2):252–292, 1977.

[Bew86] Truman F Bewley. Stationary monetary equilibrium with a continuum of independently fluctuating consumers. In Werner Hildenbran and Andreu Mas-Colell, editors, *Contributions to Mathematical Economics in Honor of Gerard Debreu*, pages 27–102. North-Holland, Amsterdam, 1986.

[Bis06] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[Bla51] David Blackwell. Comparison of experiments. In Jerzy Neyman, editor, *Proceedings of the Second Berkeley Symposium on Mathematical Statistics and Probability*, 93–102. Berkeley, CA, 1951. University of California Press.

[Bla53] David Blackwell. Equivalent comparisons of experiments. *Annals of Mathematical Statistics*, 24(2):265–272, 1953. doi:10.1214/aoms/1177729032.

[BD62] David Blackwell and Lester E. Dubins. Merging of opinions with increasing information. *Annals of Mathematical Statistics*, 33(3):882–886, 1962.

[BS86a] Olivier J. Blanchard and Lawrence H. Summers. Hysteresis and the European unemployment problem. In Stanley Fischer, editor, *NBER Macroeconomics Annual 1986, Volume 1*, pages 15–78. MIT Press, 1986.

[BK80] Olivier Jean Blanchard and Charles M Kahn. The Solution of Linear Difference Models under Rational Expectations. *Econometrica*, 48(5):1305–1311, July 1980.

[BE92] Lawrence Blume and David Easley. Evolution and market behavior. *Journal of Economic Theory*, 58(1):9–40, 1992.

[BE06] Lawrence Blume and David Easley. If you're so smart, why aren't you rich? Belief selection in complete and incomplete markets. *Econometrica*, 74(4):929–966, 2006.

[BCE+18] Lawrence E Blume, Timothy Cogley, David A Easley, Thomas J Sargent, and Viktor Tsyrennikov. A case for incomplete markets. *Journal of Economic Theory*, 178:191–221, 2018.

[BE82] Lawrence E. Blume and David Easley. Learning to be rational. *Journal of Economic Theory*, 26(2):340–351, 1982. doi:10.1016/0022-0531(82)90008-4.

[BSS49] H. F. Bohnenblust, Lloyd S. Shapley, and Seymour Sherman. Reconnaissance in game theory. Technical Report RM-208, The RAND Corporation, Santa Monica, CA, 1949. Cited for the economic criterion for comparing experiments.

[Bor20] Jaroslav Borovička. Survival and long-run dynamics with heterogeneous beliefs under recursive preferences. *Journal of Political Economy*, 128(1):206–251, 2020. doi:10.1086/704072.

[BHS16] Jaroslav Borovička, Lars Peter Hansen, and José A. Scheinkman. Misspecified recovery. *Journal of Finance*, 71(6):2493–2544, 2016. doi:10.1111/jofi.12404.

[Bra82] Margaret M. Bray. Learning, estimation, and the stability of rational expectations. *Journal of Economic Theory*, 26(2):318–339, 1982. doi:10.1016/0022-0531(82)90007-2.

[BK87] Margaret M. Bray and David M. Kreps. Rational learning and rational expectations. In George R. Feiwel, editor, *Arrow and the Ascent of Modern Economic Theory*, pages 597–625. Palgrave Macmillan, London, 1987. doi:10.1007/978-1-349-07239-2_19.

[BS86b] Margaret M. Bray and N. E. Savin. Rational expectations equilibria, learning and model specification. *Econometrica*, 54(5):1129–1160, 1986. doi:10.2307/1912325.

[Bre79] Douglas T. Breeden. An intertemporal asset pricing model with stochastic consumption and investment opportunities. *Journal of Financial Economics*, 7(3):265–296, 1979. doi:10.1016/0304-405X(79)90016-3.

[BL78] Douglas T. Breeden and Robert H. Litzenberger. Prices of state-contingent claims implicit in option prices. *Journal of Business*, 51(4):621–651, 1978. doi:10.1086/296025.

[Bro82] William A Brock. Asset prices in a production economy. In John J. McCall, editor, *The Economics of Information and Uncertainty*, pages 1–46. University of Chicago Press, Chicago, 1982.

[Bro74] William A. Brock. Money and growth: the case of long run perfect foresight. *International Economic Review*, 15(3):750–777, 1974. doi:10.2307/2525739.

[BH97] William A. Brock and Cars H. Hommes. A rational route to randomness. *Econometrica*, 65(5):1059–1095, 1997.

[BF90] Michael Bruno and Stanley Fischer. Seigniorage, operating rules, and the high inflation trap. *Quarterly Journal of Economics*, 105(2):353–374, 1990. doi:10.2307/2937791.

[BK19] Steven L. Brunton and J. Nathan Kutz. *Data-Driven Science and Engineering: Machine Learning, Dynamical Systems, and Control*. Cambridge University Press, 2019.

[BK22] Steven L. Brunton and J. Nathan Kutz. *Data-Driven Science and Engineering, Second Edition*. Cambridge University Press, New York, 2022.

[Bul94] James Bullard. Learning equilibria. *Journal of Economic Theory*, 64(2):468–485, 1994.

[BDM+16] Dariusz Buraczewski, Ewa Damek, Thomas Mikosch, and others. *Stochastic models with power-law tails*. Springer, 2016.

[Bur23] Jennifer Burns. *Milton Friedman: The Last Conservative*. Farrar, Straus, and Giroux, New York, 2023.

[Cag56] Philip Cagan. The monetary dynamics of hyperinflation. In Milton Friedman, editor, *Studies in the Quantity Theory of Money*, pages 25–117. University of Chicago Press, Chicago, 1956.

[Cam87] John Y. Campbell. Does Saving Anticipate Declining Labor Income? An Alternative Test of the Permanent Income Hypothesis. *Econometrica*, 55(6):1249–1273, 1987.

[CS88] John Y. Campbell and Robert J. Shiller. The Dividend-Price Ratio and Expectations of Future Dividends and Discount Factors. *Review of Financial Studies*, 1(3):195–228, 1988.

[Cap85] Andrew S Caplin. The variability of aggregate demand with (s, s) inventory policies. *Econometrica*, pages 1395–1409, 1985.

[CE09] Giacomo Carboni and Martin Ellison. The great inflation and the Greenbook. *Journal of Monetary Economics*, 56(6):831–841, 2009.

[CY12] Peter Carr and Jiming Yu. Risk, return, and Ross recovery. *Journal of Derivatives*, 20(1):38–59, 2012. doi:10.3905/jod.2012.20.1.038.

[Car01] Christopher D Carroll. A Theory of the Consumption Function, with and without Liquidity Constraints. *Journal of Economic Perspectives*, 15(3):23–45, 2001.

[Car06] Christopher D Carroll. The method of endogenous gridpoints for solving dynamic stochastic optimization problems. *Economics Letters*, 91(3):312–320, 2006.

[CK94] C. K. Carter and R. Kohn. On Gibbs sampling for state space models. *Biometrika*, 81(3):541–553, 1994.

[Cas65] David Cass. Optimum growth in an aggregative model of capital accumulation. *Review of Economic Studies*, 32(3):233–240, 1965.

[CM88] A. Chaudhuri and R. Mukerjee. *Randomized Response: Theory and Techniques*. Marcel Dekker, New York, 1988.

[CW98] Xiaohong Chen and Halbert White. Nonparametric adaptive learning with feedback. *Journal of Economic Theory*, 82(1):190–222, 1998. doi:10.1006/jeth.1998.2426.

[CWS02] In-Koo Cho, Noah Williams, and Thomas J. Sargent. Escaping Nash inflation. *Review of Economic Studies*, 69(1):1–40, 2002.

[Cho68] Gregory C. Chow. The acceleration principle and the nature of business cycles. *The Quarterly Journal of Economics*, 82(3):403–418, aug 1968.

[CL69] Gregory C. Chow and Richard E. Levitan. Nature of business cycles implicit in a linear economic model. *The Quarterly Journal of Economics*, 83(3):504–517, aug 1969.

[Chu90] Heetaik Chung. *Did Policy Makers Really Believe in the Phillips Curve? An Econometric Test*. PhD thesis, University of Minnesota, 1990.

[Cla71] Edward H. Clarke. Multipart pricing of public goods. *Public Choice*, 11:17–33, 1971.

[Coa37] Ronald H. Coase. The nature of the firm. *Economica*, 4(16):386–405, 1937.

[CS01] Timothy Cogley and Thomas J. Sargent. Evolving post-world war II U.S. inflation dynamics. *NBER Macroeconomics Annual*, 16:331–373, 2001.

[CS05a] Timothy Cogley and Thomas J. Sargent. Drifts and volatilities: monetary policies and outcomes in the post WWII US. *Review of Economic Dynamics*, 8(2):262–302, 2005.

[CS05b] Timothy Cogley and Thomas J. Sargent. The conquest of US inflation: learning and robustness to model uncertainty. *Review of Economic Dynamics*, 8(2):528–563, 2005.

[Col90] Wilbur John Coleman. Solving the Stochastic Growth Model by Policy-Function Iteration. *Journal of Business & Economic Statistics*, 8(1):27–29, 1990.

[CP73] Thomas F. Cooley and Edward C. Prescott. An adaptive regression model. *International Economic Review*, 14(2):364–371, 1973.

[Csiszar63] Imre Csiszár. Eine informationstheoretische Ungleichung und ihre Anwendung auf den Beweis der Ergodizität von Markoffschen Ketten. *Magyar Tud. Akad. Mat. Kutató Int. Közl.*, 8:85–108, 1963.

[DFH06] Steven J. Davis, R. Jason Faberman, and John Haltiwanger. The flow approach to labor markets: new data sources and micro-macro links. *Journal of Economic Perspectives*, 20(3):3–26, 2006.

[dF37] Bruno de Finetti. La prevision: ses lois logiques, ses sources subjectives. *Annales de l'Institut Henri Poincaré*, 7:1 – 68, 1937. English translation in Kyburg and Smokler (eds.), \it Studies in Subjective Probability, Wiley, New York, 1964.

[DLSSW91] J. Bradford De Long, Andrei Shleifer, Lawrence H. Summers, and Robert J. Waldmann. The survival of noise traders in financial markets. *Journal of Business*, 64(1):1–19, 1991.

[Dea91] Angus Deaton. Saving and Liquidity Constraints. *Econometrica*, 59(5):1221–1248, 1991.

[DP94] Angus Deaton and Christina Paxson. Intertemporal Choice and Inequality. *Journal of Political Economy*, 102(3):437–467, 1994.

[DeL97] J. Bradford DeLong. America's peacetime inflation: the 1970s. In Christina D. Romer and David H. Romer, editors, *Reducing Inflation: Motivation and Strategy*, pages 247–280. University of Chicago Press, 1997.

[DH10] Wouter J Den Haan. Comparison of solutions to the incomplete markets model with aggregate uncertainty. *Journal of Economic Dynamics and Control*, 34(1):4–27, 2010.

[DF86] Persi Diaconis and David Freedman. On the consistency of Bayes estimates. *The Annals of Statistics*, 14(1):1–26, 1986. doi:10.1214/aos/1176349830.

[Dia65] Peter A Diamond. National debt in a neoclassical growth model. *The American Economic Review*, 55(5):1126–1150, 1965.

[Doo53] Joseph L. Doob. *Stochastic Processes*. Wiley, New York, 1953.

[DS10] Ulrich Doraszelski and Mark Satterthwaite. Computable markov-perfect industry dynamics. *The RAND Journal of Economics*, 41(2):215–243, 2010.

[DSS58] Robert Dorfman, Paul A. Samuelson, and Robert M. Solow. *Linear Programming and Economic Analysis: Revised Edition*. McGraw Hill, New York, 1958.

[DLP13] Y E Du, Ehud Lehrer, and A D Y Pauzner. Competitive economy as a ranking device over networks. submitted, 2013.

[Dud02] R M Dudley. *Real Analysis and Probability*. Cambridge Studies in Advanced Mathematics. Cambridge University Press, 2002.

[DE92] Darrell Duffie and Larry G. Epstein. Stochastic differential utility. *Econometrica*, 60(2):353–394, 1992.

[DUW00] Bernard Dumas, Raman Uppal, and Tan Wang. Efficient intertemporal allocations with recursive utility. *Journal of Economic Theory*, 93(2):154–183, 2000.

[DRS89] Timothy Dunne, Mark J Roberts, and Larry Samuelson. The growth and failure of us manufacturing plants. *The Quarterly Journal of Economics*, 104(4):671–698, 1989.

[DE97] Paul Dupuis and Richard S. Ellis. *A Weak Convergence Approach to the Theory of Large Deviations*. John Wiley and Sons, New York, 1997.

[DK87] Paul Dupuis and Harold J. Kushner. Asymptotic behavior of constrained stochastic approximations via the theory of large deviations. *Probability Theory and Related Fields*, 75(2):223–244, 1987.

[DK89] Paul Dupuis and Harold J. Kushner. Stochastic approximation and large deviations: upper bounds and w.p.1 convergence. *SIAM Journal on Control and Optimization*, 27(5):1108–1135, 1989.

[ESAW18] Ashraf Ben El-Shanawany, Keith H Ardron, and Simon P Walker. Lognormal approximations of fault tree uncertainty distributions. *Risk Analysis*, 38(8):1576–1584, 2018.

[EY07] Martin Ellison and Tony Yates. Escaping volatile inflation. *Journal of Money, Credit and Banking*, 39(4):981–993, 2007.

[EG87] Robert F Engle and Clive W J Granger. Co-integration and Error Correction: Representation, Estimation, and Testing. *Econometrica*, 55(2):251–276, 1987.

[EW94] Larry G. Epstein and Tan Wang. Intertemporal Asset Pricing under Knightian Uncertainty. *Econometrica*, 62(2):283–322, 1994.

[EZ89] Larry G. Epstein and Stanley E. Zin. Substitution, risk aversion, and the temporal behavior of consumption and asset returns: a theoretical framework. *Econometrica*, 57(4):937–969, 1989. doi:10.2307/1913778.

[EP95] Richard Ericson and Ariel Pakes. Markov-perfect industry dynamics: a framework for empirical work. *The Review of Economic Studies*, 62(1):53–82, 1995.

[Eva87] David S Evans. The relationship between firm growth, size, and age: estimates for 100 manufacturing industries. *The Journal of Industrial Economics*, pages 567–581, 1987.

[EH01] G W Evans and S Honkapohja. *Learning and Expectations in Macroeconomics*. Frontiers of Economic Research. Princeton University Press, 2001.

[EH05] George W Evans and Seppo Honkapohja. An interview with thomas j. sargent. *Macroeconomic Dynamics*, 9(4):561–583, 2005.

[Eva85] George W. Evans. Expectational stability and the multiple equilibria problem in linear rational expectations models. *Quarterly Journal of Economics*, 100(4):1217–1233, 1985. doi:10.2307/1885681.

[Fab82] Joseph F Faber. Life tables for the United States: 1900–2050. Actuarial Study Actuarial Study No. 87, Social Security Administration, Office of the Actuary, 1982.

[FSTD15] Pablo Fajgelbaum, Edouard Schaal, and Mathieu Taschereau-Dumouchel. Uncertainty traps. Technical Report, National Bureau of Economic Research, 2015.

[FR24] Lasse Fischer and Aaditya Ramdas. Improving the (approximate) sequential probability ratio test by avoiding overshoot. *arXiv preprint arXiv:2410.16076*, 2024.

[FPS77] Michael A Fligner, George E Policello, and Jagbir Singh. A comparison of two randomized response survey methods with consideration for the level of respondent protection. *Communications in Statistics-Theory and Methods*, 6(15):1511–1524, 1977.

[FW98] Mark I. Freidlin and Alexander D. Wentzell. *Random Perturbations of Dynamical Systems*. Springer-Verlag, New York, 2nd edition, 1998.

[Fri56] M. Friedman. *A Theory of the Consumption Function*. Princeton University Press, 1956.

[Fri53] Milton Friedman. *Essays in positive economics*. University of Chicago press, 1953.

[Fri57] Milton Friedman. *A Theory of the Consumption Function*. Princeton University Press, Princeton, New Jersey, 1957.

[Fri68] Milton Friedman. The role of monetary policy. *American Economic Review*, 58(1):1–17, 1968.

[FF98] Milton Friedman and Rose D Friedman. *Two Lucky People*. University of Chicago Press, 1998.

[Fri33] Ragnar Frisch. Propagation problems and impulse problems in dynamic economics. In *Economic Essays in Honour of Gustav Cassel*, pages 171–203. Allen and Unwin, 1933.

[Gab16] Xavier Gabaix. Power laws in economics: an introduction. *Journal of Economic Perspectives*, 30(1):185–206, 2016.

[Gal89] David Gale. *The theory of linear economic models*. University of Chicago press, 1989.

[Gal16] Alfred Galichon. *Optimal Transport Methods in Economics*. Princeton University Press, Princeton, New Jersey, 2016.

[Gib31] Robert Gibrat. *Les inégalités économiques: Applications d'une loi nouvelle, la loi de l'effet proportionnel*. PhD thesis, Recueil Sirey, 1931.

[GS89] Itzhak Gilboa and David Schmeidler. Maxmin Expected Utility with Non-Unique Prior. *Journal of Mathematical Economics*, 18(2):141–153, apr 1989.

[Gir60] Igor V. Girsanov. On transforming a certain class of stochastic processes by absolutely continuous substitution of measures. *Theory of Probability and Its Applications*, 5(3):285–301, 1960.

[Gol89] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, Reading, MA, 1989.

[Gor95] Geoffrey J Gordon. Stable function approximation in dynamic programming. In *Machine Learning Proceedings 1995*, pages 261–268. Elsevier, 1995.

[Gra66] C. W. J. Granger. The typical spectral shape of an economic variable. *Econometrica*, 34(1):150–161, 1966.

[GAESH69] Bernard G Greenberg, Abdel-Latif A Abul-Ela, Walt R Simmons, and Daniel G Horvitz. The unrelated question randomized response model: theoretical framework. *Journal of the American Statistical Association*, 64(326):520–539, 1969.

[GKAH77] Bernard G Greenberg, Roy R Kuebler, James R Abernathy, and Daniel G Horvitz. Respondent hazards in the unrelated question randomized response model. *Journal of Statistical Planning and Inference*, 1(1):53–60, 1977.

[GS93] Moses A. Greenfield and Thomas J. Sargent. A probabilistic analysis of a catastrophic transuranic waste hoist accident at the wipp. Environmental Evaluation Group, Albuquerque, New Mexico, June 1993. URL: http://www.tomsargent.com/research/EEG-53.pdf.

[Gro76] Sanford J. Grossman. On the efficiency of competitive stock markets where trades have diverse information. *Journal of Finance*, 31(2):573–585, 1976.

[GS80] Sanford J. Grossman and Joseph E. Stiglitz. On the impossibility of informationally efficient markets. *American Economic Review*, 70(3):393–408, 1980.

[Gro73] T. Groves. Incentives in teams. *Econometrica*, 41:617–631, 1973.

[Hal87] Bronwyn H Hall. The relationship between firm size and firm growth in the us manufacturing sector. *The Journal of Industrial Economics*, pages 583–606, 1987.

[Hal71] Robert E Hall. The dynamic effects of fiscal policy in an economy with foresight. *The Review of Economic Studies*, 38(2):229–244, 1971.

[Hal78] Robert E Hall. Stochastic Implications of the Life Cycle-Permanent Income Hypothesis: Theory and Evidence. *Journal of Political Economy*, 86(6):971–987, 1978.

[HM82] Robert E. Hall and Frederic S. Mishkin. The Sensitivity of Consumption to Transitory Income: Estimates from Panel Data on Households. *Econometrica*, 50(2):461–481, 1982.

[HTW67] Michael J Hamburger, Gerald L Thompson, and Roman L Weil. Computation of expansion rates for the generalized von neumann model of an expanding economy. *Econometrica, Journal of the Econometric Society*, pages 542–547, 1967.

[Ham05] James D Hamilton. What's real about the business cycle? *Federal Reserve Bank of St. Louis Review*, pages 435–452, 2005.

[Han93] Gary D Hansen. The cyclical and secular behaviour of the labour input: comparing efficiency units and hours worked. *Journal of Applied Econometrics*, 8(1):71–80, 1993.

[HS08] L P Hansen and T J Sargent. *Robustness*. Princeton University Press, 2008.

[HS13] L P Hansen and T J Sargent. *Recursive Models of Dynamic Linear Economies*. The Gorman Lectures in Economics. Princeton University Press, 2013.

[Han82] Lars Peter Hansen. Large sample properties of generalized method of moments estimators. *Econometrica*, 50(4):1029–1054, 1982.

[Han14] Lars Peter Hansen. Nobel lecture: uncertainty outside and inside economic models. *Journal of Political Economy*, 122(5):945–987, 2014.

[HHL08] Lars Peter Hansen, John C. Heaton, and Nan Li. Consumption strikes back? measuring long-run risk. *Journal of Political Economy*, 116(2):260–302, 2008. doi:10.1086/588200.

[HJ91] Lars Peter Hansen and Ravi Jagannathan. Implications of security market data for models of dynamic economies. *Journal of Political Economy*, 99(2):225–262, 1991. doi:10.1086/261750.

[HR87] Lars Peter Hansen and Scott F Richard. The role of conditioning information in deducing testable restrictions implied by dynamc asset pricing models. *Econometrica*, 55(3):587–613, May 1987.

[HS95] Lars Peter Hansen and Thomas J Sargent. Discounted linear exponential quadratic gaussian control. *IEEE Transactions on Automatic control*, 40(5):968–971, 1995.

[HS04] Lars Peter Hansen and Thomas J Sargent. Certainty equivalence and model uncertainty. In *Conference on Models and Monetary Policy: Research in the Tradition of Dale Henderson, Richard Porter, and Peter Tinsley (http://www. federalreserve. gov/events/conferences/mmp2004/pdf/hansensargent. pdf)*. 2004.

[HS93] Lars Peter Hansen and Thomas J. Sargent. Seasonality and approximation errors in rational expectations models. *Journal of Econometrics*, 55(1–2):21–55, 1993.

[HS01] Lars Peter Hansen and Thomas J. Sargent. Robust control and model uncertainty. *American Economic Review*, 91(2):60–66, 2001.

[HST99] Lars Peter Hansen, Thomas J. Sargent, and Thomas D. Tallarini. Robust Permanent Income and Pricing. *Review of Economic Studies*, 66(4):873–907, 1999. URL: https://ideas.repec.org/a/oup/restud/v66y1999i4p873-907..html, doi:.

[HS09] Lars Peter Hansen and José A. Scheinkman. Long-term risk: an operator approach. *Econometrica*, 77(1):177–234, 2009. doi:10.3982/ECTA6761.

[HS82] Lars Peter Hansen and Kenneth J Singleton. Generalized instrumental variables estimation of nonlinear rational expectations models. *Econometrica: Journal of the Econometric Society*, pages 1269–1286, 1982.

[HS83] Lars Peter Hansen and Kenneth J Singleton. Stochastic consumption, risk aversion, and the temporal behavior of asset returns. *Journal of political economy*, 91(2):249–265, 1983.

[HK78] J. Michael Harrison and David M. Kreps. Speculative investor behavior in a stock market with heterogeneous expectations. *The Quarterly Journal of Economics*, 92(2):323–336, 1978.

[HK79] J. Michael Harrison and David M. Kreps. Martingales and arbitrage in multiperiod securities markets. *Journal of Economic Theory*, 20(3):381–408, June 1979.

[Har67] John C. Harsanyi. Games with incomplete information played by “Bayesian” players, I–III part I. The basic model. *Management Science*, 14(3):159–182, 1967.

[Har68a] John C. Harsanyi. Games with incomplete information played by "Bayesian" players, I–III part III. The basic probability distribution of the game. *Management Science*, 14(7):486–502, 1968.

[Har68b] John C. Harsanyi. Games with incomplete information played by "Bayesian" players, I–III part II. Bayesian equilibrium points. *Management Science*, 14(5):320–334, 1968.

[HL96] John Heaton and Deborah J Lucas. Evaluating the effects of incomplete markets on risk sharing and asset pricing. *Journal of Political Economy*, pages 443–487, 1996.

[HLL96] O Hernandez-Lerma and J B Lasserre. *Discrete-Time Markov Control Processes: Basic Optimality Criteria*. Number Vol 1 in Applications of Mathematics Stochastic Modelling and Applied Probability. Springer, 1996.

[Hol75] John H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, 1975.

[HHNT86] John H. Holland, Keith J. Holyoak, Richard E. Nisbett, and Paul R. Thagard. *Induction: Processes of Inference, Learning, and Discovery*. MIT Press, Cambridge, MA, 1986.

[HMMS60] Charles Holt, Franco Modigliani, John F. Muth, and Herbert Simon. *Planning Production, Inventories, and Work Force*. Prentice-Hall International Series in Management, New Jersey, 1960.

[Hop92] Hugo A Hopenhayn. Entry, exit, and firm dynamics in long run equilibrium. *Econometrica: Journal of the Econometric Society*, pages 1127–1150, 1992.

[HIS97] He Huang, Selahattin Imrohoroglu, and Thomas J Sargent. Two computations to fund social security. *Macroeconomic Dynamics*, 1(1):7–44, 1997.

[Hug93] Mark Huggett. The risk-free rate in heterogeneous-agent incomplete-insurance economies. *Journal of Economic Dynamics and Control*, 17(5-6):953–969, 1993.

[Hur50] Leonid Hurwicz. Least squares bias in time series. *Statistical inference in dynamic economic models*, 10:365–383, 1950.

[Hur62] Leonid Hurwicz. On the structural form of interdependent systems. In *Logic, Methodology and Philosophy of Science*, pages 232–239. Stanford University Press, Stanford, CA, 1962.

[HP62] Stephen Hymer and Peter Pashigian. Firm size and rate of growth. *Journal of Political Economy*, 70(6):556–569, 1962.

[Haggstrom02] Olle Häggström. *Finite Markov chains and algorithmic applications*. Volume 52. Cambridge University Press, 2002.

[Int02] Michael D Intriligator. *Mathematical optimization and economic theory*. SIAM, 2002.

[Janich94] K Jänich. *Linear Algebra*. Springer Undergraduate Texts in Mathematics and Technology. Springer, 1994.

[JK99] Matthew O. Jackson and Ehud Kalai. Reputation versus social learning. *Journal of Economic Theory*, 88(1):40–59, 1999. doi:10.1006/jeth.1999.2538.

[JKS99] Matthew O. Jackson, Ehud Kalai, and Rann Smorodinsky. Bayesian representation of stochastic processes under learning: de Finetti revisited. *Econometrica*, 67(4):875–893, 1999. doi:10.1111/1468-0262.00055.

[Jac73] D. H. Jacobson. Optimal stochastic linear systems with exponential performance criteria and their relation to differential games. *IEEE Transactions on Automatic Control*, 18(2):124–131, 1973.

[JPR94] Eric Jacquier, Nicholas G. Polson, and Peter E. Rossi. Bayesian analysis of stochastic volatility models. *Journal of Business and Economic Statistics*, 12(4):371–389, 1994.

[Jef46] Harold Jeffreys. An invariant form for the prior probability in estimation problems. *Proceedings of the Royal Society of London. Series A. Mathematical and Physical Sciences*, 186(1007):453–461, 1946.

[Jor82] James S. Jordan. The generic existence of rational expectations equilibrium in the higher dimensional case. *Journal of Economic Theory*, 26(2):224–243, 1982. doi:10.1016/0022-0531(82)90002-3.

[Jov79] Boyan Jovanovic. Firm-specific capital and turnover. *Journal of Political Economy*, 87(6):1246–1260, 1979.

[Jud90] K L Judd. Cournot versus bertrand: a dynamic resolution. Technical Report, Hoover Institution, Stanford University, 1990.

[Kak48] Shizuo Kakutani. On equivalence of infinite product measures. *Annals of Mathematics*, 49(1):214–224, 1948.

[KL93a] Ehud Kalai and Ehud Lehrer. Rational learning leads to Nash equilibrium. *Econometrica*, 61(5):1019–1045, 1993. doi:10.2307/2951492.

[KL93b] Ehud Kalai and Ehud Lehrer. Subjective equilibrium in repeated games. *Econometrica*, 61(5):1231–1240, 1993. doi:10.2307/2951500.

[KL94] Ehud Kalai and Ehud Lehrer. Weak and strong merging of opinions. *Journal of Mathematical Economics*, 23(1):73–86, 1994. doi:10.1016/0304-4068(94)90037-X.

[KLS99] Ehud Kalai, Ehud Lehrer, and Rann Smorodinsky. Calibrated forecasting and merging. *Games and Economic Behavior*, 29(1–2):151–169, 1999. doi:10.1006/game.1998.0608.

[KMR93] Michihiro Kandori, George J. Mailath, and Rafael Rob. Learning, mutation, and long run equilibria in games. *Econometrica*, 61(1):29–56, 1993.

[KW81] John Kareken and Neil Wallace. On the indeterminacy of equilibrium exchange rates. *Quarterly Journal of Economics*, 96(2):207–222, 1981. doi:10.2307/1882388.

[KT81] Samuel Karlin and Howard M. Taylor. *A Second Course in Stochastic Processes*. Academic Press, 1981.

[Kas04] Kenneth Kasa. Learning, large deviations, and recurrent currency crises. *International Economic Review*, 45(1):141–173, 2004.

[KMT56] John G Kemeny, Oskar Morgenstern, and Gerald L Thompson. A generalization of the von neumann model of an expanding economy. *Econometrica, Journal of the Econometric Society*, pages 115–135, 1956.

[KW52] Jack Kiefer and Jacob Wolfowitz. Stochastic estimation of the maximum of a regression function. *Annals of Mathematical Statistics*, 23(3):462–466, 1952.

[Kih74] Richard E. Kihlstrom. A general theory of demand for information about product quality. *Journal of Economic Theory*, 8(4):413–439, 1974. doi:10.1016/0022-0531(74)90019-2.

[Kih84] Richard E. Kihlstrom. A Bayesian exposition of Blackwell's theorem on the comparison of experiments. In Marcel Boyer and Richard E. Kihlstrom, editors, *Bayesian Models in Economic Theory*, volume 5 of Studies in Bayesian Econometrics, pages 13–31. North-Holland, Amsterdam, 1984.

[KM75] Richard E. Kihlstrom and Leonard J. Mirman. Information and market equilibrium. *The Bell Journal of Economics*, 6(1):357–376, 1975.

[KN99] Chang-Jin Kim and Charles R. Nelson. Has the U.S. economy become more stable? a Bayesian approach based on a Markov-switching model of the business cycle. *Review of Economics and Statistics*, 81(4):608–616, 1999.

[KW94] Robert G. King and Mark W. Watson. The post-war U.S. phillips curve: a revisionist econometric history. *Carnegie-Rochester Conference Series on Public Policy*, 41:157–219, 1994.

[KW89] Nobuhiro Kiyotaki and Randall Wright. On money as a medium of exchange. *Journal of Political Economy*, 97(4):927–954, 1989. doi:10.1086/261634.

[Koo65] Tjalling C. Koopmans. On the concept of optimal economic growth. In Tjalling C. Koopmans, editor, *The Economic Approach to Development Planning*, pages 225–287. Rand McNally, Chicago, 1965.

[Kre88] David M. Kreps. *Notes on the Theory of Choice*. Westview Press, Boulder, Colorado, 1988.

[Kre98] David M. Kreps. Anticipated utility and dynamic choice. In Donald P. Jacobs, Ehud Kalai, and Morton I. Kamien, editors, *Frontiers of Research in Economic Theory: The Nancy L. Schwartz Memorial Lectures, 1983–1997*, pages 242–274. Cambridge University Press, Cambridge, 1998.

[KP78] David M. Kreps and Evan L. Porteus. Temporal resolution of uncertainty and dynamic choice theory. *Econometrica*, 46(1):185–200, 1978. doi:10.2307/1913656.

[Kuh13] Moritz Kuhn. Recursive Equilibria In An Aiyagari-Style Economy With Permanent Income Shocks. *International Economic Review*, 54:807–835, 2013.

[KL51] Solomon Kullback and Richard A Leibler. On information and sufficiency. *The Annals of Mathematical Statistics*, 22(1):79–86, 1951.

[KC78] Harold J. Kushner and Dean S. Clark. *Stochastic Approximation Methods for Constrained and Unconstrained Systems*. Springer-Verlag, New York, 1978.

[KY03] Harold J. Kushner and G. George Yin. *Stochastic Approximation and Recursive Algorithms and Applications*. Springer-Verlag, New York, 2nd edition, 2003.

[KBBP16] J. Nathan Kutz, Steven L. Brunton, Bingni W. Brunton, and Joshua L. Proctor. *Dynamic Mode Decomposition: Data-Driven Modeling of Complex Systems*. SIAM, 2016.

[KP77] Finn E. Kydland and Edward C. Prescott. Rules rather than discretion: the inconsistency of optimal plans. *Journal of Political Economy*, 85(3):473–492, 1977.

[Lan75] Jan Lanke. On the choice of the unrelated question in simmons' version of randomized response. *Journal of the American Statistical Association*, 70(349):80–83, 1975.

[Lan76] Jan Lanke. On the degree of protection in randomized interviews. *International Statistical Review/Revue Internationale de Statistique*, pages 197–203, 1976.

[LS96a] Ehud Lehrer and Rann Smorodinsky. Compatible measures and merging. *Mathematics of Operations Research*, 21(3):697–706, 1996. doi:10.1287/moor.21.3.697.

[LS96b] Ehud Lehrer and Rann Smorodinsky. Merging and learning. In Thomas S. Ferguson, Lloyd S. Shapley, and James B. MacQueen, editors, *Statistics, Probability and Game Theory: Papers in Honor of David Blackwell*, volume 30 of IMS Lecture Notes – Monograph Series, pages 147–168. Institute of Mathematical Statistics, Hayward, CA, 1996. doi:10.1214/lnms/1215453571.

[LL01] Martin Lettau and Sydney Ludvigson. Consumption, Aggregate Wealth, and Expected Stock Returns. *Journal of Finance*, 56(3):815–849, 06 2001.

[LL04] Martin Lettau and Sydney C. Ludvigson. Understanding Trend and Cycle in Asset Values: Reevaluating the Wealth Effect on Consumption. *American Economic Review*, 94(1):276–299, March 2004.

[LM80] David Levhari and Leonard J Mirman. The great fish war: an example using a dynamic cournot-nash solution. *The Bell Journal of Economics*, pages 322–334, 1980.

[LW76] Frederick W Leysieffer and Stanley L Warner. Respondent jeopardy and optimal designs in randomized response models. *Journal of the American Statistical Association*, 71(355):649–656, 1976.

[Lie12] Friedrich Liese. phi-divergences, sufficiency, Bayes sufficiency, and deficiency. *Kybernetika*, 48(4):690–713, 2012.

[Lju77] Lennart Ljung. Analysis of recursive stochastic algorithms. *IEEE Transactions on Automatic Control*, 22(4):551–575, 1977. doi:10.1109/TAC.1977.1101561.

[LS18] L Ljungqvist and T J Sargent. *Recursive Macroeconomic Theory*. MIT Press, 4 edition, 2018.

[Lju93] Lars Ljungqvist. A unified approach to measures of privacy in randomized response models: a utilitarian perspective. *Journal of the American Statistical Association*, 88(421):97–103, 1993.

[Luc72] Jr. Lucas, Robert E. Econometric testing of the natural rate hypothesis. In Otto Eckstein, editor, *The Econometrics of Price Determination*, pages 50–59. Board of Governors of the Federal Reserve System, Washington, D.C., 1972.

[LP71] Jr. Lucas, Robert E. and Edward C. Prescott. Investment under uncertainty. *Econometrica*, 39(5):659–681, 1971.

[LS81] Robert E Lucas and Thomas J Sargent. *Rational expectations and econometric practice*. U of Minnesota Press, Minneapolis, Minnesota, 1981.

[Luc78] Robert E Lucas, Jr. Asset prices in an exchange economy. *Econometrica*, 46(6):1429–1445, 1978.

[Luc76] Robert E Jr Lucas. Econometric policy evaluation: a critique. In *Carnegie-Rochester conference series on public policy*, volume 1, 19–46. North-Holland, 1976.

[LJ67] Robert E Lucas Jr. Adjustment costs and the theory of supply. *Journal of political economy*, 75(4, Part 1):321–334, 1967.

[MST20] Qingyin Ma, John Stachurski, and Alexis Akira Toda. The income fluctuation problem and the evolution of wealth. *Journal of Economic Theory*, 187:105003, 2020.

[MaC82] Thomas E. MaCurdy. The use of time series processes to model the error structure of earnings in a longitudinal data analysis. *Journal of Econometrics*, 18(1):83–114, 1982.

[Man62] Edwin Mansfield. Entry, Gibrat's law, innovation, and the growth of firms. *American Economic Review*, 52(5):1023–1051, 1962.

[MS89a] Albert Marcet and Thomas J Sargent. Convergence of Least-Squares Learning in Environments with Hidden State Variables and Private Information. *Journal of Political Economy*, 97(6):1306–1322, 1989.

[MS89b] Albert Marcet and Thomas J. Sargent. Convergence of least squares learning mechanisms in self-referential linear stochastic models. *Journal of Economic Theory*, 48(2):337–368, 1989. doi:10.1016/0022-0531(89)90032-X.

[MS89c] Albert Marcet and Thomas J. Sargent. Least squares learning and the dynamics of hyperinflation. In William A. Barnett, John Geweke, and Karl Shell, editors, *Economic Complexity: Chaos, Sunspots, Bubbles, and Nonlinearity*, pages 119–137. Cambridge University Press, Cambridge, 1989.

[MMS90] Ramon Marimon, Ellen McGrattan, and Thomas J. Sargent. Money as a medium of exchange in an economy with artificially intelligent agents. *Journal of Economic Dynamics and Control*, 14(2):329–373, 1990. doi:10.1016/0165-1889(90)90025-C.

[MS93] Ramon Marimon and Shyam Sunder. Indeterminacy of equilibria in a hyperinflationary world: experimental evidence. *Econometrica*, 61(5):1073–1107, 1993. doi:10.2307/2951494.

[MCWG95] A Mas-Colell, M D Whinston, and J R Green. *Microeconomic Theory*. Volume 1. Oxford University Press, 1995.

[McC70] J J McCall. Economics of Information and Job Search. *The Quarterly Journal of Economics*, 84(1):113–126, 1970.

[MPQ00] Margaret M. McConnell and Gabriel Perez-Quiros. Output fluctuations in the United States: what has changed since the early 1980's? *American Economic Review*, 90(5):1464–1476, 2000.

[MT19] Michael McLeay and Silvana Tenreyro. Optimal inflation and the identification of the phillips curve. In Martin Eichenbaum, Erik Hurst, and Jonathan A. Parker, editors, *NBER Macroeconomics Annual 2018, Volume 33*, pages 199–255. University of Chicago Press, Chicago, 2019.

[MP04] Costas Meghir and Luigi Pistaferri. Income variance dynamics and heterogeneity. *Econometrica*, 72(1):1–32, 2004.

[MP85] Rajnish Mehra and Edward C Prescott. The equity premium: A puzzle. *Journal of Monetary Economics*, 15(2):145–161, 1985.

[MT98] Enrique G Mendoza and Linda L Tesar. The international ramifications of tax reforms: supply-side economics in a global economy. *American Economic Review*, pages 226–245, 1998.

[Mil77] Edward M Miller. Risk, uncertainty, and divergence of opinion. *The Journal of finance*, 32(4):1151–1168, 1977.

[MS99] Ronald I. Miller and Chris William Sanchirico. The role of absolute continuity in "Merging of Opinions" and "Rational Learning". *Games and Economic Behavior*, 29(1–2):170–190, 1999. doi:10.1006/game.1999.0752.

[MP69] Marvin Minsky and Seymour Papert. *Perceptrons: An Introduction to Computational Geometry*. MIT Press, Cambridge, MA, 1969.

[MB54] F. Modigliani and R. Brumberg. Utility analysis and the consumption function: An interpretation of cross-section data. In K.K Kurihara, editor, *Post-Keynesian Economics*. 1954.

[Mor63] Tetsuzo Morimoto. Markov Processes and the H-Theorem. *Journal of the Physical Society of Japan*, 18(3):328–331, 1963. doi:10.1143/JPSJ.18.328.

[Mor96] Stephen Morris. Speculative investor behavior and learning. *The Quarterly Journal of Economics*, 111(4):1111–1133, 1996.

[MAM10] Iain Murray, Ryan P. Adams, and David J. C. MacKay. Elliptical slice sampling. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of Proceedings of Machine Learning Research, 541–548. 2010.

[Mut60] John F Muth. Optimal properties of exponentially weighted forecasts. *Journal of the american statistical association*, 55(290):299–306, 1960.

[Mut61] John F. Muth. Rational expectations and the theory of price movements. *Econometrica*, 29(3):315–335, 1961.

[Nea99] Derek Neal. The Complexity of Job Mobility among Young Men. *Journal of Labor Economics*, 17(2):237–261, 1999.

[NP82] Charles R. Nelson and Charles I. Plosser. Trends and random walks in macroeconomic time series: some evidence and implications. *Journal of Monetary Economics*, 10(2):139–162, 1982.

[NP33] J. Neyman and E. S. Pearson. On the problem of the most efficient tests of statistical hypotheses. *Philosophical Transactions of the Royal Society of London*, 231(694–706):289–337, 1933.

[Nov72] Alexander A. Novikov. On an identity for stochastic integrals. *Theory of Probability and Its Applications*, 17(4):717–720, 1972.

[Nya94] Yaw Nyarko. Bayesian learning leads to correlated equilibria in normal form games. *Economic Theory*, 4(6):821–841, 1994. doi:10.1007/BF01213814.

[OW69] Guy H. Orcutt and Herbert S. Winokur. First order autoregression: inference, estimation, and prediction. *Econometrica*, 37(1):1–14, 1969.

[Orp01] Athanasios Orphanides. Monetary policy rules based on real-time data. *American Economic Review*, 91(4):964–985, 2001.

[Par99] Jonathan A Parker. The Reaction of Household Consumption to Predictable Changes in Social Security Taxes. *American Economic Review*, 89(4):959–973, 1999.

[Per96] Lawrence Perko. *Differential Equations and Dynamical Systems*. Springer-Verlag, New York, 2nd edition, 1996.

[Phe67] Edmund S. Phelps. Phillips curves, expectations of inflation and optimal unemployment over time. *Economica*, 34(135):254–281, 1967.

[PT77] Edmund S. Phelps and John B. Taylor. Stabilizing powers of monetary policy under rational expectations. *Journal of Political Economy*, 85(1):163–190, 1977.

[PSS15] Monika Piazzesi, Juliana Salomao, and Martin Schneider. Trend and Cycle in Bond Premia. Working Paper, Stanford University, 2015.

[PANS14] Luciano Pomatto, Nabil I. Al-Najjar, and Alvaro Sandroni. Merging and testing opinions. *The Annals of Statistics*, 42(3):1003–1028, 2014. doi:10.1214/14-AOS1212.

[PM80] Edward C Prescott and Rajnish Mehra. Recursive competitive equilibrium: the case of homogeneous households. *Econometrica*, 48(6):1365–1379, 1980.

[PV80] Edward C. Prescott and Michael Visscher. Organization capital. *Journal of Political Economy*, 88(3):446–461, 1980.

[Pri06] Giorgio E. Primiceri. Why inflation rose and fell: policymakers' beliefs and U.S. postwar stabilization policy. *Quarterly Journal of Economics*, 121(3):867–901, 2006.

[PalS13] Jenő Pál and John Stachurski. Fitted value function iteration with probability one contractions. *Journal of Economic Dynamics and Control*, 37(1):251–264, 2013.

[Rab02] Guillaume Rabault. When do borrowing constraints bind? Some new results on the income fluctuation problem. *Journal of Economic Dynamics and Control*, 26(2):217–245, 2002.

[Rad79] Roy Radner. Rational expectations equilibrium: generic existence and the information revealed by prices. *Econometrica*, 47(3):655–678, 1979. doi:10.2307/1910413.

[Ref96] Kevin L Reffett. Production-based asset pricing in monetary economies with transactions costs. *Economica*, pages 427–443, 1996.

[Rei09] Michael Reiter. Solving heterogeneous-agent models by projection and perturbation. *Journal of Economic Dynamics and Control*, 33(3):649–665, 2009.

[RM51] Herbert Robbins and Sutton Monro. A stochastic approximation method. *Annals of Mathematical Statistics*, 22(3):400–407, 1951.

[Rog85] Kenneth Rogoff. The optimal degree of commitment to an intermediate monetary target. *Quarterly Journal of Economics*, 100(4):1169–1189, 1985.

[Ros15] Stephen A. Ross. The recovery theorem. *Journal of Finance*, 70(2):615–648, 2015. doi:10.1111/jofi.12092.

[Rya12] Stephen P Ryan. The costs of environmental regulation in a concentrated industry. *Econometrica*, 80(3):1019–1061, 2012.

[Red97] Knut Røed. Hysteresis in unemployment. *Journal of Economic Surveys*, 11(4):389–418, 1997.

[Sam39] Paul A. Samuelson. Interactions between the multiplier analysis and the principle of acceleration. *The Review of Economics and Statistics*, 21(2):75–78, 1939.

[Sam58] Paul A. Samuelson. An exact consumption-loan model of interest with or without the social contrivance of money. *Journal of Political Economy*, 66(6):467–482, 1958. doi:10.1086/258100.

[SS60] Paul A. Samuelson and Robert M. Solow. Analytical aspects of anti-inflation policy. *American Economic Review*, 50(2):177–194, 1960.

[San98] Alvaro Sandroni. Necessary and sufficient conditions for convergence to Nash equilibrium: the almost absolute continuity hypothesis. *Games and Economic Behavior*, 22(1):121–147, 1998. doi:10.1006/game.1997.0572.

[San00] Alvaro Sandroni. Do markets favor agents able to make accurate predictions? *Econometrica*, 68(6):1303–1341, 2000.

[Sar77] Thomas J Sargent. The Demand for Money During Hyperinflations under Rational Expectations: I. *International Economic Review*, 18(1):59–82, February 1977.

[Sar81] Thomas J Sargent. Interpreting economic time series. *Journal of political Economy*, 89(2):213–248, 1981.

[Sar87] Thomas J Sargent. *Macroeconomic Theory*. Academic Press, New York, 2nd edition, 1987.

[Sar89] Thomas J Sargent. Two models of measurements and the investment accelerator. *Journal of Political Economy*, 97(2):251–287, 1989.

[Sar93] Thomas J. Sargent. *Bounded Rationality in Macroeconomics*. The Arne Ryde Memorial Lectures. Oxford University Press, Oxford, 1993.

[Sar99] Thomas J. Sargent. *The Conquest of American Inflation*. Princeton University Press, Princeton, New Jersey, 1999.

[Sar08] Thomas J. Sargent. Evolution and intelligent design. *American Economic Review*, 98(1):5–37, 2008.

[SS24] Thomas J. Sargent and John Stachurski. *Economic Networks: Theory and Computation*. Structural Analysis in the Social Sciences. Cambridge University Press, 2024.

[SS25] Thomas J. Sargent and John Stachurski. *Dynamic Programming: Finite States*. Cambridge University Press, 2025.

[SV95] Thomas J. Sargent and François R. Velde. Macroeconomic features of the french revolution. *Journal of Political Economy*, 103(3):474–518, 1995.

[SW05] Thomas J. Sargent and Noah Williams. Impacts of priors on convergence and escapes from Nash inflation. *Review of Economic Dynamics*, 8(2):360–391, 2005.

[SW25] Thomas J. Sargent and Noah Williams. Rationalizing Fed interest rate decisions in the 2020s. *Journal of Political Economy: Macroeconomics*, 2025. Forthcoming.

[SWZ06] Thomas J. Sargent, Noah Williams, and Tao Zha. Shocks and government beliefs: the rise and fall of american inflation. *American Economic Review*, 96(4):1193–1224, 2006.

[SE77] Jack Schechtman and Vera L S Escudero. Some results on an income fluctuation problem. *Journal of Economic Theory*, 16(2):151–166, 1977.

[Sch14] Jose A. Scheinkman. *Speculation, Trading, and Bubbles*. Columbia University Press, New York, 2014.

[Sch10] Peter J Schmid. Dynamic mode decomposition of numerical and experimental data. *Journal of fluid mechanics*, 656:5–28, 2010.

[Sha48] Claude E Shannon. A mathematical theory of communication. *The Bell system technical journal*, 27(3):379–423, 1948.

[SZT17] Ravid Shwartz-Ziv and Naftali Tishby. Opening the Black Box of Deep Neural Networks via Information. *arXiv preprint arXiv:1703.00810*, 2017.

[Sim56] Herbert A Simon. Dynamic programming under uncertainty with a quadratic criterion function. *Econometrica, Journal of the Econometric Society*, pages 74–81, 1956.

[Sim80] Christopher A. Sims. Macroeconomics and reality. *Econometrica*, 48(1):1–48, 1980.

[Sim88] Christopher A. Sims. Projecting policy effects with statistical models. *Revista de Análisis Económico*, 3(1):3–20, 1988.

[Sim01] Christopher A. Sims. Comment on Sargent and Cogley's `evolving post world war II US inflation dynamics'. *NBER Macroeconomics Annual*, 16:373–379, 2001.

[SZ06] Christopher A. Sims and Tao Zha. Were there regime switches in U.S. monetary policy? *American Economic Review*, 96(1):54–81, 2006.

[Slu37] Eugen Slutzky. The summation of random causes as the source of cyclic processes. *Econometrica*, 5(2):105–146, 1937. doi:10.2307/1907241.

[Sol68] Robert M. Solow. Recent controversy on the theory of inflation: an eclectic view. In Stephen W. Rousseaus, editor, *Proceedings of a Symposium on Inflation: Its Causes, Consequences, and Control*. New York University, New York, 1968.

[Sta08] John Stachurski. Continuous state dynamic programming via nonexpansive approximation. *Computational Economics*, 31(2):141–160, 2008.

[ST19] John Stachurski and Alexis Akira Toda. An impossibility theorem for wealth in heterogeneous-agent models with limited heterogeneity. *Journal of Economic Theory*, 182:1–24, 2019.

[Sti58] George J. Stigler. The economies of scale. *Journal of Law and Economics*, 1:54–71, 1958.

[Sto01] James H. Stock. Discussion of Cogley and Sargent `evolving post world war II US inflation dynamics'. *NBER Macroeconomics Annual*, 16:379–387, 2001.

[SW07] James H. Stock and Mark W. Watson. Why has U.S. inflation become harder to forecast? *Journal of Money, Credit and Banking*, 39(S1):3–33, 2007.

[Sto89] Nancy L Stokey. Reputation and time consistency. *The American Economic Review*, pages 134–139, 1989.

[STY04] Kjetil Storesletten, Christopher I Telmer, and Amir Yaron. Consumption and risk sharing over the life cycle. *Journal of Monetary Economics*, 51(3):609–633, 2004.

[Sun96] R K Sundaram. *A First Course in Optimization Theory*. Cambridge University Press, 1996.

[SB18] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[Tau86] George Tauchen. Finite state markov-chain approximations to univariate and vector autoregressions. *Economics Letters*, 20(2):177–181, 1986.

[Tay97] John B. Taylor. Comment on `america's peacetime inflation: the 1970s'. In Christina D. Romer and David H. Romer, editors, *Reducing Inflation: Motivation and Strategy*, pages 276–280. University of Chicago Press, 1997.

[The57] Henri Theil. A note on certainty equivalence in dynamic planning. *Econometrica: Journal of the Econometric Society*, pages 346–349, 1957.

[TPB99] Naftali Tishby, Fernando C. Pereira, and William Bialek. The Information Bottleneck Method. In *Proceedings of the 37th Annual Allerton Conference on Communication, Control, and Computing*, 368–377. 1999.

[Tob68] James Tobin. Discussion. In Stephen W. Rousseaus, editor, *Proceedings of a Symposium on Inflation: Its Causes, Consequences, and Control*. New York University, New York, 1968.

[Tow83] Robert M. Townsend. Forecasting the forecasts of others. *Journal of Political Economy*, 91(4):546–588, 1983. doi:10.1086/261166.

[Tre16] Daniel Treisman. Russia's billionaires. *The American Economic Review*, 106(5):236–241, 2016.

[TRL+14] J. H. Tu, C. W. Rowley, D. M. Luchtenburg, S. L. Brunton, and J. N. Kutz. On dynamic mode decomposition: theory and applications. *Journal of Computational Dynamics*, 1(2):391–421, 2014.

[VL11] Ngo Van Long. Dynamic games in the economics of natural resources: a survey. *Dynamic Games and Applications*, 1(1):115–148, 2011.

[Vic61] W. Vickrey. Counterspeculation, auctions, and competitive sealed tenders. *Journal of Finance*, 16:8–37, 1961.

[vN28] John von Neumann. Zur theorie der gesellschaftsspiele. *Mathematische annalen*, 100(1):295–320, 1928.

[vN37] John von Neumann. Uber ein okonomsiches gleichungssystem und eine verallgemeinering des browerschen fixpunktsatzes. In *Erge. Math. Kolloq.*, volume 8, 73–83. 1937.

[Wal47] Abraham Wald. *Sequential Analysis*. John Wiley and Sons, New York, 1947.

[Wal80] W Allen Wallis. The statistical research group, 1942–1945. *Journal of the American Statistical Association*, 75(370):320–330, 1980.

[War65] Stanley L Warner. Randomized response: a survey technique for eliminating evasive answer bias. *Journal of the American Statistical Association*, 60(309):63–69, 1965.

[Wec79] William E Wecker. Predicting the turning points of a time series. *Journal of business*, pages 35–50, 1979.

[Wei89] Philippe Weil. The equity premium puzzle and the risk-free rate puzzle. *Journal of Monetary Economics*, 24(3):401–421, 1989. doi:10.1016/0304-3932(89)90028-7.

[Whi83] Charles Whiteman. *Linear Rational Expectations Models: A User's Guide*. University of Minnesota Press, Minneapolis, Minnesota, 1983.

[Whi53] Peter Whittle. The analysis of multiple stationary time series. *Journal of the Royal Statistical Society, Series B*, 15(1):125–139, 1953.

[Whi81] Peter Whittle. Risk-sensitive linear/quadratic/gaussian control. *Advances in Applied Probability*, 13(4):764–777, 1981.

[Whi90] Peter Whittle. *Risk-Sensitive Optimal Control*. Wiley, New York, 1990.

[Wil19] Noah Williams. Escape dynamics in learning models. *Review of Economic Studies*, 86(2):882–912, 2019.

[Wil75] Oliver E. Williamson. *Markets and Hierarchies: Analysis and Antitrust Implications*. Free Press, New York, 1975.

[Woo90] Michael Woodford. Learning to believe in sunspots. *Econometrica*, 58(2):277–307, 1990.

[Woo15] Jeffrey M Wooldridge. *Introductory econometrics: A modern approach*. Nelson Education, 2015.

[YS05] G Alastair Young and Richard L Smith. *Essentials of statistical inference*. Cambridge University Press, 2005.

[vCencov81] Nikolai N. Čencov. *Statistical Decision Rules and Optimal Inference*. Volume 53 of Translations of Mathematical Monographs. American Mathematical Society, Providence, RI, 1981.

[DeGroot62] Morris H. DeGroot. Uncertainty, information, and sequential experiments. *Annals of Mathematical Statistics*, 33(2):404–419, 1962. doi:10.1214/aoms/1177704567.

PROOF INDEX

absolute_continuity
absolute_continuity (*merging_of_opinions*), 676

ak-aiyagari-algorithm
ak-aiyagari-algorithm (*ak_aiyagari*), 2234

algorithm-0
algorithm-0 (*two_computation*), 2262

ass-ms-a1
ass-ms-a1 (*ls_learning*), 2462

ass-ms-a2
ass-ms-a2 (*ls_learning*), 2462

ass-ms-a3
ass-ms-a3 (*ls_learning*), 2463

ass-ms-a4
ass-ms-a4 (*ls_learning*), 2463

ass-ms-a5
ass-ms-a5 (*ls_learning*), 2463

ass-ms-a6
ass-ms-a6 (*ls_learning*), 2463

ass-ms-a7
ass-ms-a7 (*ls_learning*), 2463

assm-cond-ind
assm-cond-ind (*unemployment_shocks*), 2703

assum-bk-borel
assum-bk-borel (*rational_learning_re*), 2449

assum-bk-continuity
assum-bk-continuity (*rational_learning_re*), 2450

assum-bk-identification
assum-bk-identification (*rational_learning_re*), 2450

assumption-mr-condition-4
assumption-mr-condition-4 (*misspecified_recovery*), 2599

assumption-mr-ergodicity
assumption-mr-ergodicity (*misspecified_recovery*), 2598

assumption-mr-markov-shocks
assumption-mr-markov-shocks (*misspecified_recovery*), 2594

assumption1
assumption1 (*von_neumann_model*), 749

assumption2
assumption2 (*von_neumann_model*), 749

blackwell_dubins
blackwell_dubins (*merging_of_opinions*), 679

com_eq_tax
com_eq_tax (*cass_fiscal*), 1997

cor-ms-exogenous
cor-ms-exogenous (*ls_learning*), 2466

cor-ms-projection-dichotomy
cor-ms-projection-dichotomy (*ls_learning*), 2466

cor-risk-neutral-recovery
cor-risk-neutral-recovery (*ross_recovery*), **??**

cp_admissible
cp_admissible (*phillips_credible_policies*), 1767

cp_prop_best
cp_prop_best (*phillips_credible_policies*), 1770

cp_prop_worst
cp_prop_worst (*phillips_credible_policies*), 1769

cp_selfenf
cp_selfenf (*phillips_credible_policies*), 1773

cp_selfgen
cp_selfgen (*phillips_credible_policies*), 1768

cp_spe
cp_spe (*phillips_credible_policies*), 1767

cp_strategy
cp_strategy (*phillips_credible_policies*), 1766

def-blackwell-sufficiency
def-blackwell-sufficiency (*blackwell_kihlstrom*), 578

def-circulant-matrix
def-circulant-matrix (*eig_circulant*), 56

def-degroot-uncertainty
def-degroot-uncertainty (*blackwell_kihlstrom*), 578

def-economic-criterion
def-economic-criterion (*blackwell_kihlstrom*), 577

def-markov-perfect-equilibrium
def-markov-perfect-equilibrium (*markov_perf*), 2125

def-sufficiency
def-sufficiency (*blackwell_kihlstrom*), 575

def-transition-independence
def-transition-independence (*ross_recovery*), **??**

definition-2
definition-2 (*von_neumann_model*), 749

definition-3
definition-3 (*von_neumann_model*), 752

definition-4
definition-4 (*von_neumann_model*), 752

definition-6
definition-6 (*von_neumann_model*), 753

definition-8
definition-8 (*von_neumann_model*), 759

dichotomy_at_infinity
dichotomy_at_infinity (*merging_of_opinions*), 694

equilibrium_asset_price
equilibrium_asset_price (*morris_learn*), 2427

hellinger_affinity
hellinger_affinity (*merging_of_opinions*), 690

ime_def_sufficiency
ime_def_sufficiency (*information_market_equilibrium*), 604

ime_lemma_posterior_sufficiency
ime_lemma_posterior_sufficiency (*information_market_equilibrium*), 604

ime_lemma_same_price_same_allocation
ime_lemma_same_price_same_allocation (*information_market_equilibrium*), 605

ime_lemma_unique_posterior
ime_lemma_unique_posterior (*information_market_equilibrium*), 606

ime_theorem_bayesian_convergence
ime_theorem_bayesian_convergence (*information_market_equilibrium*), 613

ime_theorem_invertibility_conditions
ime_theorem_invertibility_conditions (*information_market_equilibrium*), 607

ime_theorem_price_revelation
ime_theorem_price_revelation (*information_market_equilibrium*), 604

kakutani_dichotomy
kakutani_dichotomy (*merging_of_opinions*), 691

lc_prop1
lc_prop1 (*phillips_lost_conquest*), 1876

lc_prop2
lc_prop2 (*phillips_lost_conquest*), 1876

lrr-def-additive-functional

lrr-def-additive-functional (*long_run_risk_operator*), 2539

lrr-def-cash-flow-valuation-semigroup

lrr-def-cash-flow-valuation-semigroup (*long_run_risk_operator*), 2540

lrr-def-exponential-affine-eigenfunction

lrr-def-exponential-affine-eigenfunction (*long_run_risk_operator*), 2557

lrr-def-extended-generator

lrr-def-extended-generator (*long_run_risk_operator*), 2544

lrr-def-functional

lrr-def-functional (*long_run_risk_operator*), 2537

lrr-def-generator-eigenfunction

lrr-def-generator-eigenfunction (*long_run_risk_operator*), 2546

lrr-def-harris-recurrence

lrr-def-harris-recurrence (*long_run_risk_operator*), 2548

lrr-def-intensity-matrix

lrr-def-intensity-matrix (*long_run_risk_operator*), 2550

lrr-def-irreducible-skeleton

lrr-def-irreducible-skeleton (*long_run_risk_operator*), 2547

lrr-def-local-brownian-risk-price

lrr-def-local-brownian-risk-price (*long_run_risk_operator*), 2566

lrr-def-long-run-risk-price

lrr-def-long-run-risk-price (*long_run_risk_operator*), 2566

lrr-def-martingale-component

lrr-def-martingale-component (*long_run_risk_operator*), 2546

lrr-def-multiplicative-functional

lrr-def-multiplicative-functional (*long_run_risk_operator*), 2538

lrr-def-multiplicative-semigroup

lrr-def-multiplicative-semigroup (*long_run_risk_operator*), 2540

lrr-def-one-parameter-semigroup

lrr-def-one-parameter-semigroup (*long_run_risk_operator*), 2539

lrr-def-stationary-distribution

lrr-def-stationary-distribution (*long_run_risk_operator*), 2547

lrr-def-stochastic-discount-factor

lrr-def-stochastic-discount-factor (*long_run_risk_operator*), 2540

lrr-def-stochastic-growth-functional

lrr-def-stochastic-growth-functional (*long_run_risk_operator*), 2540

lrr-def-stochastic-stability

lrr-def-stochastic-stability (*long_run_risk_operator*), 2548

lrr-def-valuation-functional

lrr-def-valuation-functional (*long_run_risk_operator*), 2540

martingale_convergence

martingale_convergence (*merging_of_opinions*), 678

most_optimistic_valuation

most_optimistic_valuation (*morris_learn*), 2427

normalized_price

normalized_price (*morris_learn*), 2427

one_sided_vs_mutual

one_sided_vs_mutual (*merging_of_opinions*), 680

pa_prop

pa_prop (*phillips_adaptive*), 1787

pm_bmap

pm_bmap (*phillips_misspecified*), 1796

pm_equilibrium
pm_equilibrium (*phillips_misspecified*), 1796

prop-bk-event-convergence
prop-bk-event-convergence (*rational_learning_re*), 2449

prop-bk-measure-convergence
prop-bk-measure-convergence (*rational_learning_re*), 2450

prop-bk-sharpening
prop-bk-sharpening (*rational_learning_re*), 2451

prop-dmd-eigenvectors
prop-dmd-eigenvectors (*var_dmd*), 96

prop-loo-identity
prop-loo-identity (*unemployment_shocks*), 2704

prop-lqcs-bewley
prop-lqcs-bewley (*lq_robust_smoothing*), 1549

prop-mean-preservation
prop-mean-preservation (*blackwell_kihlstrom*), 579

prop-misspecified-recovery-martingale-component
prop-misspecified-recovery-martingale-component (*misspecified_recovery*), 2592

prop-mr-uniqueness
prop-mr-uniqueness (*misspecified_recovery*), 2598

prop-ms-convergence
prop-ms-convergence (*ls_learning*), 2464

prop-ms-jacobian-eigenvalues
prop-ms-jacobian-eigenvalues (*ls_learning*), 2464

prop-ms-necessity
prop-ms-necessity (*ls_learning*), 2465

prop:hst_oe
prop:hst_oe (*robust_permanent_income*), 1470

rate_dominance_beta
rate_dominance_beta (*morris_learn*), 2428

sc_def
sc_def (*phillips_self_confirming*), **??**

sharpness
sharpness (*merging_of_opinions*), 680

speculative_premium
speculative_premium (*morris_learn*), 2428

survival_conditions
survival_conditions (*survival_recursive_preferences*), 706

theorem-0
theorem-0 (*os_stochastic*), 1259

theorem1
theorem1 (*von_neumann_model*), 752

theorem2
theorem2 (*von_neumann_model*), 758

thm-blackwell
thm-blackwell (*blackwell_kihlstrom*), 579

thm-data-processing
thm-data-processing (*blackwell_kihlstrom*), 597

thm-kihlstrom
thm-kihlstrom (*blackwell_kihlstrom*), 580

thm-lqcs-oe1
thm-lqcs-oe1 (*lq_robust_smoothing*), 1541

thm-lqcs-oe2
thm-lqcs-oe2 (*lq_robust_smoothing*), 1545

thm-perron-frobenius
thm-perron-frobenius (*ross_recovery*), **??**

thm-ross-recovery
thm-ross-recovery (*ross_recovery*), **??**

total_variation_distance
total_variation_distance (*merging_of_opinions*), 677

two_comp_algo_buyout
two_comp_algo_buyout (*two_computation*), 2283

two_comp_algo_endo
two_comp_algo_endo (*two_computation*), 2290

two_comp_algo_govfund

two_comp_algo_govfund (*two_computation*), 2297

two_trader_optimist

two_trader_optimist (*morris_learn*), 2428

INDEX

A

A Bayesian Formulation of Friedman and Wald's Problem, 521
A Problem that Stumped Milton Friedman, 495
An Introduction to Job Search, 1053, 1069
Asset Pricing
    GMM Estimation, 2365
    MLE Estimation, 2335
Asset Pricing: Finite State Models, 2313

B

Bewley Model, 1523, 1527
Bounded Rationality, 1581
    Exchange Rate Indeterminacy, 1641
    Genetic Algorithms, 1657
    Money as a Medium of Exchange, 1677
    Overlapping Generations, 1599
    Parameterized Expectations, 1627
    Prospects, 1717

C

Central Limit Theorem, 181, 186
    Intuition, 186
    Multivariate Case, 191
Certainty Equivalence
    Robustness, 1485, 1493
CLT, 181
Complete Markets, 1523
    Arrow securities, 1529
Consumption Smoothing, 1509
Cross-Section Distributions
    consumption, 1525

D

Detection Error Probabilities, 1544
Distorted Expectations
    Stackelberg multiplier game, 1543
Dynamic Programming
    Computation, 1260
    Theory, 1259
    Unbounded Utility, 1260

E

Eigenvalues, 21, 37
Eigenvectors, 21, 37
Ergodicity, 763, 777

F

Finite Markov Asset Pricing
    Lucas Tree, 2321
Finite Markov Chains, 763, 764
    Stochastic Matrices, 764
Frequency Domain
    permanent income model, 1544

G

Gram-Schmidt process, 959

H

Hansen Sargent Tallarini
    model, 1539
hidden Markov model, 959
History Dependence, 1513

I

Irreducibility and Aperiodicity, 763, 770

J

Job Search VII: On-the-Job Search, 1133
Job Search VII: On-the-Job Search, 1133

K

Kalman Filter, 921, 955
    and vector autoregressions, 961
    Programming Implementation, 930
    Recursive Procedure, 928
Kalman Filter 2, 937
Kesten processes
    heavy tails, 892

L

Lake Model, 2055
Lake Model, Endogenous, 2075

Law of Large Numbers, 181, 182
    Illustration, 184
    Multivariate Case, 191
    Proof, 183
Linear Algebra, 21
    Differentiating Linear and Quadratic Forms, 41
    Eigenvalues, 37
    Eigenvectors, 37
    Matrices, 30
    Matrix Norms, 40
    Neumann's Theorem, 41
    Positive Definite Matrices, 41
    SciPy, 37
    Series Expansions, 40
    Spectral Radius, 41
    Vectors, 22
Linear Markov Perfect Equilibria, 2125
Linear State Space Models, 799, 891
    Distributions, 806
    Ergodicity, 813
    Martingale Difference Shocks, 801
    Moments, 806
    Moving Average Representations, 805
    Prediction, 819
    Seasonals, 804
    Stationarity, 813
    Time Trends, 804
    Univariate Autoregressive Processes, 802
    Vector Autoregressions, 804
LLN, 181
LQ Control, 1383
    Infinite Horizon, 1395
    Optimality (*Finite Horizon*), 1387
    Permanent Income, 1485, 1493
LQ Permanent Income Model, 1509
    cross section, 1523
    standard, 1510

M

Marginal Distributions, 763, 768
Markov Asset Pricing
    Overview, 2313
Markov Chains, 764
    Calculating Stationary Distributions, 775
    Convergence to Stationarity, 776
    Cross-Sectional Distributions, 770
    Ergodicity, 777
    Forecasting Future Values, 778
    Future Probabilities, 770
    Irreducibility, Aperiodicity, 770
    Marginal Distributions, 768
    Simulation, 766
    Stationary Distributions, 773
Markov Perfect Equilibrium, 2123
    Applications, 2128
    Background, 2124
    Overview, 2123
Markov process, inventory, 787
Matrix
    Determinants, 35
    Inverse, 35
    Maps, 32
    Numpy, 31
    Operations, 30
    Solving Systems of Equations, 32
Modeling
    Career Choice, 1119
Modeling COVID 19, 9
Models
    Harrison Kreps, 2411
    Linear State Space, 800
    Markov Asset Pricing, 2313
    McCall, 1030
    Morris Learning, 2423
    On-the-Job Search, 1133
    Permanent Income, 1431, 1447
    Pricing, 2314
    Sequential analysis, 495, 521

N

Neumann's Theorem, 41

O

Observational Equivalence
    robustness and discounting, 1539
    Theorem 1, 1541
    Theorem 2, 1545
On-the-Job Search
    Model, 1134
    Model Features, 1134
    Parameterization, 1134
    Programming Implementation, 1135
    Solving for Policies, 1138
Optimal Savings
    Computation, 1328, 1368
    Model, 1256
    Policy Function, 1267
    Policy Function Approach, 1257
    Problem, 1326, 1344
    Programming Implementation, 1332, 1349
Optimal Savings III: Stochastic Returns, 1255
Optimal Savings IV: Time Iteration, 1271
Optimal Savings V: The Endogenous Grid Method, 1283

Optimal Savings VI: EGM with JAX, 1291
Organization Capital, 981
    Bayesian Screening, 991
    Human Capital, 995
    Industry Equilibrium, 989
    Personnel Information, 982
    Team Information, 993
    Transfer Costs, 996

P

Pandas for Panel Data, 2623
Permanent Income Hypothesis
    Friedman, 1510
Permanent Income II: LQ Techniques, 1447
Permanent Income Model
    examples, 1513
    Hall's Representation, 1439
    Savings Problem, 1432
Phillips Curve
    Adaptive Expectations, 1783
    Adaptive Learning and Escape Dynamics, 1819
    Credibility Problem, 1749
    Credible Policies, 1761
    Drifts and Volatilities, 1881
    Escaping Nash Inflation, 1839
    Fed Policy in the 2020s, 1869
    Optimal Misspecified Beliefs, 1793
    Priors and Learning Cycles, 1853
    Rise and Fall of U.S. Inflation, 1731
    Self-Confirming Equilibria, 1805
Positive Definite Matrices, 41
Precautionary Savings
    robustness, 1537, 1539
    robustness vs convex marginal utility, 1542
Pricing Models, 2313, 2314
    Risk Aversion, 2314
    Risk-Neutral, 2314
Python
    Pandas, 2623
python, 99, 742

R

Rational Expectations Equilibrium, 2089
    Competitive Equilbrium (*w. Adjustment Costs*), 2092
    Computation, 2095
    Definition, 2092
    Planning Problem Approach, 2095
Riccati equation
    matrix difference, 958
Robust Bewley Model, 1548
Robust Control
    permanent income, 1537, 1539
Robustness
    payoff evaluation, 1545

S

spectral factorization identity, 961
Spectral Radius, 41
Stability in Linear Rational Expectations Models, 2103
Stationary Distributions, 763, 773
Stochastic Matrices, 764
Survival
    Recursive Preferences, 701

T

The Income Fluctuation Problem III: The Endogenous Grid Method, 1325
The Income Fluctuation Problem IV: Transient Income Shocks, 1343
The Income Fluctuation Problem V: Stochastic Returns on Assets, 1365
The Permanent Income Model, 1431

U

Unbounded Utility, 1260

V

Vector Autoregression
    and Kalman filter, 955
vector autoregression, 960
Vectors, 21, 22
    Inner Product, 26
    Linear Independence, 29
    Norm, 26
    Operations, 23
    Span, 26