Intermediate Quantitative Economics with Python

Thomas J. Sargent & John Stachurski

Jan 16, 2024

CONTENTS

I Tools and Techniques **5**

1 Geometric Series for Elementary Economics **7**
1.1 Overview . . . 7
1.2 Key Formulas . . . 8
1.3 Example: The Money Multiplier in Fractional Reserve Banking . . . 8
1.4 Example: The Keynesian Multiplier . . . 10
1.5 Example: Interest Rates and Present Values . . . 12
1.6 Back to the Keynesian Multiplier . . . 20

2 Modeling COVID 19 **25**
2.1 Overview . . . 25
2.2 The SIR Model . . . 26
2.3 Implementation . . . 27
2.4 Experiments . . . 28
2.5 Ending Lockdown . . . 32

3 Linear Algebra **35**
3.1 Overview . . . 35
3.2 Vectors . . . 36
3.3 Matrices . . . 44
3.4 Solving Systems of Equations . . . 46
3.5 Eigenvalues and Eigenvectors . . . 51
3.6 Further Topics . . . 54
3.7 Exercises . . . 56

4 QR Decomposition **59**
4.1 Overview . . . 59
4.2 Matrix Factorization . . . 59
4.3 Gram-Schmidt process . . . 59
4.4 Some Code . . . 61
4.5 Example . . . 62
4.6 Using QR Decomposition to Compute Eigenvalues . . . 64
4.7 QR and PCA . . . 65

5 Complex Numbers and Trigonometry **69**
5.1 Overview . . . 69
5.2 De Moivre's Theorem . . . 72
5.3 Applications of de Moivre's Theorem . . . 73

6 Circulant Matrices **79**
6.1 Overview . . . 79

6.2 Constructing a Circulant Matrix . . . 79
6.3 Connection to Permutation Matrix . . . 81
6.4 Examples with Python . . . 82
6.5 Associated Permutation Matrix . . . 86
6.6 Discrete Fourier Transform . . . 88

7 Singular Value Decomposition (SVD) 95
7.1 Overview . . . 95
7.2 The Setting . . . 95
7.3 Singular Value Decomposition . . . 96
7.4 Four Fundamental Subspaces . . . 97
7.5 Eckart-Young Theorem . . . 100
7.6 Full and Reduced SVD's . . . 101
7.7 Polar Decomposition . . . 104
7.8 Application: Principal Components Analysis (PCA) . . . 105
7.9 Relationship of PCA to SVD . . . 106
7.10 PCA with Eigenvalues and Eigenvectors . . . 107
7.11 Connections . . . 108
7.12 Exercises . . . 111

8 VARs and DMDs 113
8.1 First-Order Vector Autoregressions . . . 113
8.2 Dynamic Mode Decomposition (DMD) . . . 116
8.3 Representation 1 . . . 116
8.4 Representation 2 . . . 117
8.5 Representation 3 . . . 119
8.6 Source for Some Python Code . . . 123

9 Using Newton's Method to Solve Economic Models 125
9.1 Overview . . . 125
9.2 Fixed Point Computation Using Newton's Method . . . 126
9.3 Root-Finding in One Dimension . . . 132
9.4 Multivariate Newton's Method . . . 134
9.5 Exercises . . . 142

II Elementary Statistics 149

10 Elementary Probability with Matrices 151
10.1 Sketch of Basic Concepts . . . 152
10.2 What Does Probability Mean? . . . 152
10.3 Representing Probability Distributions . . . 153
10.4 Univariate Probability Distributions . . . 154
10.5 Bivariate Probability Distributions . . . 155
10.6 Marginal Probability Distributions . . . 156
10.7 Conditional Probability Distributions . . . 156
10.8 Statistical Independence . . . 157
10.9 Means and Variances . . . 157
10.10 Generating Random Numbers . . . 157
10.11 Some Discrete Probability Distributions . . . 162
10.12 Geometric distribution . . . 163
10.13 Continuous Random Variables . . . 166
10.14 A Mixed Discrete-Continuous Distribution . . . 168
10.15 Matrix Representation of Some Bivariate Distributions . . . 169
10.16 A Continuous Bivariate Random Vector . . . 176

10.17 Sum of Two Independently Distributed Random Variables . . . 184
10.18 Transition Probability Matrix . . . 185
10.19 Coupling . . . 186
10.20 Copula Functions . . . 187
10.21 Time Series . . . 192

11 LLN and CLT **193**
11.1 Overview . . . 193
11.2 Relationships . . . 194
11.3 LLN . . . 194
11.4 CLT . . . 198
11.5 Exercises . . . 204

12 Two Meanings of Probability **211**
12.1 Overview . . . 211
12.2 Frequentist Interpretation . . . 212
12.3 Bayesian Interpretation . . . 218
12.4 Role of a Conjugate Prior . . . 227

13 Multivariate Hypergeometric Distribution **229**
13.1 Overview . . . 229
13.2 The Administrator's Problem . . . 229
13.3 Usage . . . 233

14 Multivariate Normal Distribution **239**
14.1 Overview . . . 239
14.2 The Multivariate Normal Distribution . . . 240
14.3 Bivariate Example . . . 243
14.4 Trivariate Example . . . 247
14.5 One Dimensional Intelligence (IQ) . . . 248
14.6 Information as Surprise . . . 252
14.7 Cholesky Factor Magic . . . 254
14.8 Math and Verbal Intelligence . . . 254
14.9 Univariate Time Series Analysis . . . 257
14.10 Stochastic Difference Equation . . . 262
14.11 Application to Stock Price Model . . . 264
14.12 Filtering Foundations . . . 266
14.13 Classic Factor Analysis Model . . . 270
14.14 PCA and Factor Analysis . . . 272

15 Heavy-Tailed Distributions **277**

16 Fault Tree Uncertainties **279**
16.1 Overview . . . 279
16.2 Log normal distribution . . . 280
16.3 The Convolution Property . . . 281
16.4 Approximating Distributions . . . 282
16.5 Convolving Probability Mass Functions . . . 286
16.6 Failure Tree Analysis . . . 289
16.7 Application . . . 289
16.8 Failure Rates Unknown . . . 290
16.9 Waste Hoist Failure Rate . . . 290

17 Introduction to Artificial Neural Networks **295**
17.1 Overview . . . 295

17.2 A Deep (but not Wide) Artificial Neural Network . . . 296
17.3 Calibrating Parameters . . . 297
17.4 Back Propagation and the Chain Rule . . . 297
17.5 Training Set . . . 298
17.6 Example 1 . . . 301
17.7 How Deep? . . . 303
17.8 Example 2 . . . 303

18 Randomized Response Surveys **307**
18.1 Overview . . . 307
18.2 Warner's Strategy . . . 307
18.3 Comparing Two Survey Designs . . . 309
18.4 Concluding Remarks . . . 315

19 Expected Utilities of Random Responses **317**
19.1 Overview . . . 317
19.2 Privacy Measures . . . 317
19.3 Zoo of Concepts . . . 317
19.4 Respondent's Expected Utility . . . 319
19.5 Utilitarian View of Survey Design . . . 323
19.6 Criticisms of Proposed Privacy Measures . . . 326
19.7 Concluding Remarks . . . 331

III Linear Programming **333**

20 Linear Programming **335**
20.1 Overview . . . 335
20.2 Objective Function and Constraints . . . 335
20.3 Example 1: Production Problem . . . 336
20.4 Example 2: Investment Problem . . . 338
20.5 Standard Form . . . 340
20.6 Computations . . . 342
20.7 Duality . . . 345
20.8 Duality Theorems . . . 346

21 Optimal Transport **353**
21.1 Overview . . . 353
21.2 The Optimal Transport Problem . . . 354
21.3 The Linear Programming Approach . . . 355
21.4 The Dual Problem . . . 362
21.5 The Python Optimal Transport Package . . . 366

22 Von Neumann Growth Model (and a Generalization) **371**
22.1 Notation . . . 376
22.2 Model Ingredients and Assumptions . . . 377
22.3 Dynamic Interpretation . . . 378
22.4 Duality . . . 379
22.5 Interpretation as Two-player Zero-sum Game . . . 381

IV Introduction to Dynamics **387**

23 Dynamics in One Dimension **389**
23.1 Overview . . . 389

23.2 Some Definitions . . . 389
23.3 Graphical Analysis . . . 392
23.4 Exercises . . . 403

24 AR1 Processes **409**
24.1 Overview . . . 409
24.2 The AR(1) Model . . . 410
24.3 Stationarity and Asymptotic Stability . . . 412
24.4 Ergodicity . . . 414
24.5 Exercises . . . 415

25 Finite Markov Chains **421**
25.1 Overview . . . 421
25.2 Definitions . . . 422
25.3 Simulation . . . 424
25.4 Marginal Distributions . . . 427
25.5 Irreducibility and Aperiodicity . . . 429
25.6 Stationary Distributions . . . 432
25.7 Ergodicity . . . 435
25.8 Computing Expectations . . . 436
25.9 Exercises . . . 437

26 Inventory Dynamics **445**
26.1 Overview . . . 445
26.2 Sample Paths . . . 446
26.3 Marginal Distributions . . . 448
26.4 Exercises . . . 451

27 Linear State Space Models **455**
27.1 Overview . . . 455
27.2 The Linear State Space Model . . . 456
27.3 Distributions and Moments . . . 461
27.4 Stationarity and Ergodicity . . . 468
27.5 Noisy Observations . . . 473
27.6 Prediction . . . 473
27.7 Code . . . 474
27.8 Exercises . . . 475

28 Samuelson Multiplier-Accelerator **477**
28.1 Overview . . . 477
28.2 Details . . . 479
28.3 Implementation . . . 482
28.4 Stochastic Shocks . . . 492
28.5 Government Spending . . . 494
28.6 Wrapping Everything Into a Class . . . 497
28.7 Using the LinearStateSpace Class . . . 502
28.8 Pure Multiplier Model . . . 509
28.9 Summary . . . 514

29 Kesten Processes and Firm Dynamics **515**
29.1 Overview . . . 515
29.2 Kesten Processes . . . 516
29.3 Heavy Tails . . . 519
29.4 Application: Firm Dynamics . . . 521
29.5 Exercises . . . 522

30 Wealth Distribution Dynamics **527**
30.1 Overview . . . 527
30.2 Lorenz Curves and the Gini Coefficient . . . 528
30.3 A Model of Wealth Dynamics . . . 531
30.4 Implementation . . . 532
30.5 Applications . . . 535
30.6 Exercises . . . 538

31 A First Look at the Kalman Filter **541**
31.1 Overview . . . 541
31.2 The Basic Idea . . . 542
31.3 Convergence . . . 550
31.4 Implementation . . . 550
31.5 Exercises . . . 551

32 Another Look at the Kalman Filter **559**
32.1 A worker's output . . . 560
32.2 A firm's wage-setting policy . . . 560
32.3 A state-space representation . . . 561
32.4 An Innovations Representation . . . 562
32.5 Some Computational Experiments . . . 563
32.6 Future Extensions . . . 572

33 Shortest Paths **573**
33.1 Overview . . . 573
33.2 Outline of the Problem . . . 574
33.3 Finding Least-Cost Paths . . . 576
33.4 Solving for Minimum Cost-to-Go . . . 577
33.5 Exercises . . . 578

V Search **583**

34 Job Search I: The McCall Search Model **585**
34.1 Overview . . . 585
34.2 The McCall Model . . . 586
34.3 Computing the Optimal Policy: Take 1 . . . 588
34.4 Computing an Optimal Policy: Take 2 . . . 594
34.5 Exercises . . . 595

35 Job Search II: Search and Separation **601**
35.1 Overview . . . 601
35.2 The Model . . . 602
35.3 Solving the Model . . . 603
35.4 Implementation . . . 605
35.5 Impact of Parameters . . . 608
35.6 Exercises . . . 609

36 Job Search III: Fitted Value Function Iteration **613**
36.1 Overview . . . 613
36.2 The Algorithm . . . 614
36.3 Implementation . . . 616
36.4 Exercises . . . 618

37 Job Search IV: Correlated Wage Offers **621**

37.1 Overview . . . 621
37.2 The Model . . . 622
37.3 Implementation . . . 623
37.4 Unemployment Duration . . . 627
37.5 Exercises . . . 629

38 Job Search V: Modeling Career Choice **631**
38.1 Overview . . . 631
38.2 Model . . . 632
38.3 Implementation . . . 634
38.4 Exercises . . . 639

39 Job Search VI: On-the-Job Search **645**
39.1 Overview . . . 645
39.2 Model . . . 646
39.3 Implementation . . . 647
39.4 Solving for Policies . . . 650
39.5 Exercises . . . 653

40 Job Search VII: A McCall Worker Q-Learns **657**
40.1 Overview . . . 657
40.2 Review of McCall Model . . . 658
40.3 Implied Quality Function Q . . . 662
40.4 From Probabilities to Samples . . . 663
40.5 Q-Learning . . . 663
40.6 Employed Worker Can't Quit . . . 671
40.7 Possible Extensions . . . 673

VI Consumption, Savings and Capital **675**

41 Cass-Koopmans Model **677**
41.1 Overview . . . 677
41.2 The Model . . . 678
41.3 Planning Problem . . . 680
41.4 Shooting Algorithm . . . 683
41.5 Setting Initial Capital to Steady State Capital . . . 687
41.6 A Turnpike Property . . . 689
41.7 A Limiting Infinite Horizon Economy . . . 690
41.8 Concluding Remarks . . . 693

42 Cass-Koopmans Competitive Equilibrium **695**
42.1 Overview . . . 695
42.2 Review of Cass-Koopmans Model . . . 696
42.3 Competitive Equilibrium . . . 697
42.4 Market Structure . . . 698
42.5 Firm Problem . . . 698
42.6 Household Problem . . . 699
42.7 Computing a Competitive Equilibrium . . . 701
42.8 Yield Curves and Hicks-Arrow Prices . . . 709

43 Cake Eating I: Introduction to Optimal Saving **711**
43.1 Overview . . . 711
43.2 The Model . . . 712
43.3 The Value Function . . . 713

43.4 The Optimal Policy . . . 715
43.5 The Euler Equation . . . 716
43.6 Exercises . . . 718

44 Cake Eating II: Numerical Methods 721
44.1 Overview . . . 721
44.2 Reviewing the Model . . . 722
44.3 Value Function Iteration . . . 722
44.4 Time Iteration . . . 730
44.5 Exercises . . . 731

45 Optimal Growth I: The Stochastic Optimal Growth Model 737
45.1 Overview . . . 737
45.2 The Model . . . 738
45.3 Computation . . . 742
45.4 Exercises . . . 750

46 Optimal Growth II: Accelerating the Code with Numba 753
46.1 Overview . . . 753
46.2 The Model . . . 754
46.3 Computation . . . 755
46.4 Exercises . . . 759

47 Optimal Growth III: Time Iteration 765
47.1 Overview . . . 765
47.2 The Euler Equation . . . 766
47.3 Implementation . . . 768
47.4 Exercises . . . 773

48 Optimal Growth IV: The Endogenous Grid Method 777
48.1 Overview . . . 777
48.2 Key Idea . . . 778
48.3 Implementation . . . 779

49 The Income Fluctuation Problem I: Basic Model 785
49.1 Overview . . . 785
49.2 The Optimal Savings Problem . . . 786
49.3 Computation . . . 788
49.4 Implementation . . . 789
49.5 Exercises . . . 794

50 The Income Fluctuation Problem II: Stochastic Returns on Assets 801
50.1 Overview . . . 801
50.2 The Savings Problem . . . 802
50.3 Solution Algorithm . . . 803
50.4 Implementation . . . 805
50.5 Exercises . . . 810

VII Bayes Law 813

51 Non-Conjugate Priors 815
51.1 Unleashing MCMC on a Binomial Likelihood . . . 816
51.2 Prior Distributions . . . 818
51.3 Implementation . . . 822

51.4 Alternative Prior Distributions . . . 827
51.5 Posteriors Via MCMC and VI . . . 831
51.6 Non-conjugate Prior Distributions . . . 837

52 Posterior Distributions for AR(1) Parameters **855**
52.1 PyMC Implementation . . . 858
52.2 Numpyro Implementation . . . 860

53 Forecasting an AR(1) Process **865**
53.1 A Univariate First-Order Autoregressive Process . . . 866
53.2 Implementation . . . 867
53.3 Predictive Distributions of Path Properties . . . 868
53.4 A Wecker-Like Algorithm . . . 869
53.5 Using Simulations to Approximate a Posterior Distribution . . . 870
53.6 Calculating Sample Path Statistics . . . 871
53.7 Original Wecker Method . . . 872
53.8 Extended Wecker Method . . . 874
53.9 Comparison . . . 876

VIII Information **879**

54 Job Search VII: Search with Learning **881**
54.1 Overview . . . 881
54.2 Model . . . 882
54.3 Take 1: Solution by VFI . . . 885
54.4 Take 2: A More Efficient Method . . . 890
54.5 Another Functional Equation . . . 891
54.6 Solving the RWFE . . . 891
54.7 Implementation . . . 892
54.8 Exercises . . . 893
54.9 Solutions . . . 893
54.10 Appendix A . . . 895
54.11 Appendix B . . . 897
54.12 Examples . . . 901

55 Likelihood Ratio Processes **913**
55.1 Overview . . . 913
55.2 Likelihood Ratio Process . . . 914
55.3 Nature Permanently Draws from Density g . . . 915
55.4 Peculiar Property . . . 917
55.5 Nature Permanently Draws from Density f . . . 918
55.6 Likelihood Ratio Test . . . 919
55.7 Kullback–Leibler Divergence . . . 924
55.8 Sequels . . . 927

56 Computing Mean of a Likelihood Ratio Process **929**
56.1 Overview . . . 929
56.2 Mathematical Expectation of Likelihood Ratio . . . 930
56.3 Importance sampling . . . 932
56.4 Selecting a Sampling Distribution . . . 933
56.5 Approximating a cumulative likelihood ratio . . . 934
56.6 Distribution of Sample Mean . . . 935
56.7 More Thoughts about Choice of Sampling Distribution . . . 937

57 A Problem that Stumped Milton Friedman **943**
57.1 Overview . . . 943
57.2 Origin of the Problem . . . 944
57.3 A Dynamic Programming Approach . . . 945
57.4 Implementation . . . 950
57.5 Analysis . . . 952
57.6 Comparison with Neyman-Pearson Formulation . . . 958
57.7 Sequels . . . 960

58 Exchangeability and Bayesian Updating **961**
58.1 Overview . . . 961
58.2 Independently and Identically Distributed . . . 962
58.3 A Setting in Which Past Observations Are Informative . . . 963
58.4 Relationship Between IID and Exchangeable . . . 964
58.5 Exchangeability . . . 965
58.6 Bayes' Law . . . 965
58.7 More Details about Bayesian Updating . . . 966
58.8 Appendix . . . 969
58.9 Sequels . . . 975

59 Likelihood Ratio Processes and Bayesian Learning **977**
59.1 Overview . . . 977
59.2 The Setting . . . 978
59.3 Likelihood Ratio Process and Bayes' Law . . . 979
59.4 Behavior of posterior probability $\{\pi_t\}$ under the subjective probability distribution . . . 983
59.5 Initial Prior is Verified by Paths Drawn from Subjective Conditional Densities . . . 989
59.6 Drilling Down a Little Bit . . . 990
59.7 Sequels . . . 991

60 Incorrect Models **993**
60.1 Overview . . . 993
60.2 Sampling from Compound Lottery H . . . 996
60.3 Type 1 Agent . . . 998
60.4 What a type 1 Agent Learns when Mixture H Generates Data . . . 999
60.5 Kullback-Leibler Divergence Governs Limit of π_t . . . 1001
60.6 Type 2 Agent . . . 1005
60.7 Concluding Remarks . . . 1007

61 Bayesian versus Frequentist Decision Rules **1009**
61.1 Overview . . . 1010
61.2 Setup . . . 1010
61.3 Frequentist Decision Rule . . . 1013
61.4 Bayesian Decision Rule . . . 1018
61.5 Was the Navy Captain's Hunch Correct? . . . 1025
61.6 More Details . . . 1027
61.7 Distribution of Bayesian Decision Rule's Time to Decide . . . 1027
61.8 Probability of Making Correct Decision . . . 1030
61.9 Distribution of Likelihood Ratios at Frequentist's t . . . 1032

IX LQ Control **1035**

62 LQ Control: Foundations **1037**
62.1 Overview . . . 1037
62.2 Introduction . . . 1038

62.3 Optimality – Finite Horizon . . . 1040
62.4 Implementation . . . 1043
62.5 Extensions and Comments . . . 1048
62.6 Further Applications . . . 1050
62.7 Exercises . . . 1058

63 Lagrangian for LQ Control **1067**
63.1 Overview . . . 1067
63.2 Undiscounted LQ DP Problem . . . 1068
63.3 Lagrangian . . . 1069
63.4 State-Costate Dynamics . . . 1070
63.5 Reciprocal Pairs Property . . . 1070
63.6 Schur decomposition . . . 1071
63.7 Application . . . 1072
63.8 Other Applications . . . 1077
63.9 Discounted Problems . . . 1078

64 Eliminating Cross Products **1081**
64.1 Overview . . . 1081
64.2 Undiscounted Dynamic Programming Problem . . . 1081
64.3 Kalman Filter . . . 1082
64.4 Duality table . . . 1083

65 The Permanent Income Model **1085**
65.1 Overview . . . 1085
65.2 The Savings Problem . . . 1086
65.3 Alternative Representations . . . 1093
65.4 Two Classic Examples . . . 1096
65.5 Further Reading . . . 1099
65.6 Appendix: The Euler Equation . . . 1099

66 Permanent Income II: LQ Techniques **1101**
66.1 Overview . . . 1101
66.2 Setup . . . 1102
66.3 The LQ Approach . . . 1104
66.4 Implementation . . . 1105
66.5 Two Example Economies . . . 1108

67 Production Smoothing via Inventories **1119**
67.1 Overview . . . 1119
67.2 Example 1 . . . 1124
67.3 Inventories Not Useful . . . 1126
67.4 Inventories Useful but are Hardwired to be Zero Always . . . 1126
67.5 Example 2 . . . 1127
67.6 Example 3 . . . 1128
67.7 Example 4 . . . 1129
67.8 Example 5 . . . 1131
67.9 Example 6 . . . 1132
67.10 Exercises . . . 1135

X Multiple Agent Models 1141

68 Schelling's Segregation Model **1143**
68.1 Outline . . . 1143

68.2 The Model . . . 1144
68.3 Results . . . 1145
68.4 Exercises . . . 1149

69 A Lake Model of Employment and Unemployment 1157
69.1 Overview . . . 1157
69.2 The Model . . . 1158
69.3 Implementation . . . 1160
69.4 Dynamics of an Individual Worker . . . 1165
69.5 Endogenous Job Finding Rate . . . 1167
69.6 Exercises . . . 1175

70 Rational Expectations Equilibrium 1185
70.1 Overview . . . 1185
70.2 Rational Expectations Equilibrium . . . 1188
70.3 Computing an Equilibrium . . . 1191
70.4 Exercises . . . 1193

71 Stability in Linear Rational Expectations Models 1199
71.1 Overview . . . 1200
71.2 Linear Difference Equations . . . 1200
71.3 Illustration: Cagan's Model . . . 1202
71.4 Some Python Code . . . 1204
71.5 Alternative Code . . . 1206
71.6 Another Perspective . . . 1207
71.7 Log money Supply Feeds Back on Log Price Level . . . 1210
71.8 Big P, Little p Interpretation . . . 1213
71.9 Fun with SymPy . . . 1215

72 Markov Perfect Equilibrium 1219
72.1 Overview . . . 1219
72.2 Background . . . 1220
72.3 Linear Markov Perfect Equilibria . . . 1221
72.4 Application . . . 1223
72.5 Exercises . . . 1228

73 Uncertainty Traps 1237
73.1 Overview . . . 1237
73.2 The Model . . . 1238
73.3 Implementation . . . 1241
73.4 Results . . . 1242
73.5 Exercises . . . 1243

74 The Aiyagari Model 1251
74.1 Overview . . . 1251
74.2 The Economy . . . 1252
74.3 Firms . . . 1253
74.4 Code . . . 1254

XI Asset Pricing and Finance 1261

75 Asset Pricing: Finite State Models 1263
75.1 Overview . . . 1263
75.2 Pricing Models . . . 1264

75.3 Prices in the Risk-Neutral Case . . . 1265
75.4 Risk Aversion and Asset Prices . . . 1269
75.5 Exercises . . . 1278

76 Competitive Equilibria with Arrow Securities **1283**
76.1 Introduction . . . 1283
76.2 The setting . . . 1284
76.3 Recursive Formulation . . . 1285
76.4 State Variable Degeneracy . . . 1286
76.5 Markov Asset Prices . . . 1286
76.6 General Equilibrium . . . 1288
76.7 Python Code . . . 1292
76.8 Finite Horizon . . . 1303

77 Heterogeneous Beliefs and Bubbles **1309**
77.1 Overview . . . 1309
77.2 Structure of the Model . . . 1310
77.3 Solving the Model . . . 1312
77.4 Exercises . . . 1317

XII Data and Empirics 1321

78 Pandas for Panel Data **1323**
78.1 Overview . . . 1323
78.2 Slicing and Reshaping Data . . . 1324
78.3 Merging Dataframes and Filling NaNs . . . 1329
78.4 Grouping and Summarizing Data . . . 1334
78.5 Final Remarks . . . 1340
78.6 Exercises . . . 1340

79 Linear Regression in Python **1345**
79.1 Overview . . . 1345
79.2 Simple Linear Regression . . . 1346
79.3 Extending the Linear Regression Model . . . 1351
79.4 Endogeneity . . . 1353
79.5 Summary . . . 1357
79.6 Exercises . . . 1357

80 Maximum Likelihood Estimation **1361**
80.1 Overview . . . 1361
80.2 Set Up and Assumptions . . . 1362
80.3 Conditional Distributions . . . 1365
80.4 Maximum Likelihood Estimation . . . 1367
80.5 MLE with Numerical Methods . . . 1369
80.6 Maximum Likelihood Estimation with `statsmodels` . . . 1374
80.7 Summary . . . 1378
80.8 Exercises . . . 1379

XIII Auctions 1383

81 First-Price and Second-Price Auctions **1385**
81.1 First-Price Sealed-Bid Auction (FPSB) . . . 1385
81.2 Second-Price Sealed-Bid Auction (SPSB) . . . 1386

81.3 Characterization of SPSB Auction . . . 1386
81.4 Uniform Distribution of Private Values . . . 1387
81.5 Setup . . . 1387
81.6 First price sealed bid auction . . . 1387
81.7 Second Price Sealed Bid Auction . . . 1388
81.8 Python Code . . . 1388
81.9 Revenue Equivalence Theorem . . . 1390
81.10 Calculation of Bid Price in FPSB . . . 1392
81.11 χ^2 Distribution . . . 1393
81.12 5 Code Summary . . . 1396
81.13 References . . . 1401

82 Multiple Good Allocation Mechanisms **1403**
82.1 Overview . . . 1403
82.2 Ascending Bids Auction for Multiple Goods . . . 1403
82.3 A Benevolent Planner . . . 1404
82.4 Equivalence of Allocations . . . 1404
82.5 Ascending Bid Auction . . . 1404
82.6 Pseudocode . . . 1405
82.7 An Example . . . 1407
82.8 A Python Class . . . 1415
82.9 Robustness Checks . . . 1424
82.10 A Groves-Clarke Mechanism . . . 1436
82.11 An Example Solved by Hand . . . 1437
82.12 Another Python Class . . . 1440

XIV Other **1447**

83 Troubleshooting **1449**
83.1 Fixing Your Local Environment . . . 1449
83.2 Reporting an Issue . . . 1450

84 References **1451**

85 Execution Statistics **1453**

Bibliography **1457**

Index **1465**

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

- Tools and Techniques
  - *Geometric Series for Elementary Economics*
  - *Modeling COVID 19*
  - *Linear Algebra*
  - *QR Decomposition*
  - *Complex Numbers and Trigonometry*
  - *Circulant Matrices*
  - *Singular Value Decomposition (SVD)*
  - *VARs and DMDs*
  - *Using Newton's Method to Solve Economic Models*
- Elementary Statistics
  - *Elementary Probability with Matrices*
  - *LLN and CLT*
  - *Two Meanings of Probability*
  - *Multivariate Hypergeometric Distribution*
  - *Multivariate Normal Distribution*
  - *Heavy-Tailed Distributions*
  - *Fault Tree Uncertainties*
  - *Introduction to Artificial Neural Networks*
  - *Randomized Response Surveys*
  - *Expected Utilities of Random Responses*
- Linear Programming
  - *Linear Programming*
  - *Optimal Transport*
  - *Von Neumann Growth Model (and a Generalization)*
- Introduction to Dynamics
  - *Dynamics in One Dimension*
  - *AR1 Processes*
  - *Finite Markov Chains*
  - *Inventory Dynamics*
  - *Linear State Space Models*
  - *Samuelson Multiplier-Accelerator*
  - *Kesten Processes and Firm Dynamics*
  - *Wealth Distribution Dynamics*
  - *A First Look at the Kalman Filter*

  - *Another Look at the Kalman Filter*
  - *Shortest Paths*
- Search
  - *Job Search I: The McCall Search Model*
  - *Job Search II: Search and Separation*
  - *Job Search III: Fitted Value Function Iteration*
  - *Job Search IV: Correlated Wage Offers*
  - *Job Search V: Modeling Career Choice*
  - *Job Search VI: On-the-Job Search*
  - *Job Search VII: A McCall Worker Q-Learns*
- Consumption, Savings and Capital
  - *Cass-Koopmans Model*
  - *Cass-Koopmans Competitive Equilibrium*
  - *Cake Eating I: Introduction to Optimal Saving*
  - *Cake Eating II: Numerical Methods*
  - *Optimal Growth I: The Stochastic Optimal Growth Model*
  - *Optimal Growth II: Accelerating the Code with Numba*
  - *Optimal Growth III: Time Iteration*
  - *Optimal Growth IV: The Endogenous Grid Method*
  - *The Income Fluctuation Problem I: Basic Model*
  - *The Income Fluctuation Problem II: Stochastic Returns on Assets*
- Bayes Law
  - *Non-Conjugate Priors*
  - *Posterior Distributions for AR(1) Parameters*
  - *Forecasting an AR(1) Process*
- Information
  - *Job Search VII: Search with Learning*
  - *Likelihood Ratio Processes*
  - *Computing Mean of a Likelihood Ratio Process*
  - *A Problem that Stumped Milton Friedman*
  - *Exchangeability and Bayesian Updating*
  - *Likelihood Ratio Processes and Bayesian Learning*
  - *Incorrect Models*
  - *Bayesian versus Frequentist Decision Rules*
- LQ Control
  - *LQ Control: Foundations*

  - *Lagrangian for LQ Control*
  - *Eliminating Cross Products*
  - *The Permanent Income Model*
  - *Permanent Income II: LQ Techniques*
  - *Production Smoothing via Inventories*
- Multiple Agent Models
  - *Schelling's Segregation Model*
  - *A Lake Model of Employment and Unemployment*
  - *Rational Expectations Equilibrium*
  - *Stability in Linear Rational Expectations Models*
  - *Markov Perfect Equilibrium*
  - *Uncertainty Traps*
  - *The Aiyagari Model*
- Asset Pricing and Finance
  - *Asset Pricing: Finite State Models*
  - *Competitive Equilibria with Arrow Securities*
  - *Heterogeneous Beliefs and Bubbles*
- Data and Empirics
  - *Pandas for Panel Data*
  - *Linear Regression in Python*
  - *Maximum Likelihood Estimation*
- Auctions
  - *First-Price and Second-Price Auctions*
  - *Multiple Good Allocation Mechanisms*
- Other
  - *Troubleshooting*
  - *References*
  - *Execution Statistics*

Part I

Tools and Techniques

CHAPTER

ONE

GEOMETRIC SERIES FOR ELEMENTARY ECONOMICS

Contents

- *Geometric Series for Elementary Economics*
  - *Overview*
  - *Key Formulas*
  - *Example: The Money Multiplier in Fractional Reserve Banking*
  - *Example: The Keynesian Multiplier*
  - *Example: Interest Rates and Present Values*
  - *Back to the Keynesian Multiplier*

1.1 Overview

The lecture describes important ideas in economics that use the mathematics of geometric series.

Among these are

- the Keynesian **multiplier**
- the money **multiplier** that prevails in fractional reserve banking systems
- interest rates and present values of streams of payouts from assets

(As we shall see below, the term **multiplier** comes down to meaning **sum of a convergent geometric series**)

These and other applications prove the truth of the wise crack that

> "in economics, a little knowledge of geometric series goes a long way "

Below we'll use the following imports:

```
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
import sympy as sym
from sympy import init_printing
from matplotlib import cm
from mpl_toolkits.mplot3d import Axes3D
```

1.2 Key Formulas

To start, let c be a real number that lies strictly between -1 and 1.

- We often write this as $c \in (-1, 1)$.
- Here $(-1, 1)$ denotes the collection of all real numbers that are strictly less than 1 and strictly greater than -1.
- The symbol $\in$ means *in* or *belongs to the set after the symbol*.

We want to evaluate geometric series of two types – infinite and finite.

1.2.1 Infinite Geometric Series

The first type of geometric that interests us is the infinite series

$$1 + c + c^2 + c^3 + \cdots$$

Where $\cdots$ means that the series continues without end.

The key formula is

$$1 + c + c^2 + c^3 + \cdots = \frac{1}{1-c} \tag{1.1}$$

To prove key formula (1.1), multiply both sides by $(1-c)$ and verify that if $c \in (-1, 1)$, then the outcome is the equation $1 = 1$.

1.2.2 Finite Geometric Series

The second series that interests us is the finite geometric series

$$1 + c + c^2 + c^3 + \cdots + c^T$$

where T is a positive integer.

The key formula here is

$$1 + c + c^2 + c^3 + \cdots + c^T = \frac{1 - c^{T+1}}{1-c}$$

Remark: The above formula works for any value of the scalar c. We don't have to restrict c to be in the set $(-1, 1)$.

We now move on to describe some famous economic applications of geometric series.

1.3 Example: The Money Multiplier in Fractional Reserve Banking

In a fractional reserve banking system, banks hold only a fraction $r \in (0, 1)$ of cash behind each **deposit receipt** that they issue

- In recent times
  - cash consists of pieces of paper issued by the government and called dollars or pounds or ...
  - a *deposit* is a balance in a checking or savings account that entitles the owner to ask the bank for immediate payment in cash

- When the UK and France and the US were on either a gold or silver standard (before 1914, for example)
  - cash was a gold or silver coin
  - a *deposit receipt* was a *bank note* that the bank promised to convert into gold or silver on demand; (sometimes it was also a checking or savings account balance)

Economists and financiers often define the **supply of money** as an economy-wide sum of **cash** plus **deposits**.

In a **fractional reserve banking system** (one in which the reserve ratio r satisfies $0 < r < 1$), **banks create money** by issuing deposits *backed* by fractional reserves plus loans that they make to their customers.

A geometric series is a key tool for understanding how banks create money (i.e., deposits) in a fractional reserve system.

The geometric series formula (1.1) is at the heart of the classic model of the money creation process – one that leads us to the celebrated **money multiplier**.

1.3.1 A Simple Model

There is a set of banks named $i = 0, 1, 2, \ldots$.

Bank i's loans L_i, deposits D_i, and reserves R_i must satisfy the balance sheet equation (because **balance sheets balance**):

$$L_i + R_i = D_i \tag{1.2}$$

The left side of the above equation is the sum of the bank's **assets**, namely, the loans L_i it has outstanding plus its reserves of cash R_i.

The right side records bank i's liabilities, namely, the deposits D_i held by its depositors; these are IOU's from the bank to its depositors in the form of either checking accounts or savings accounts (or before 1914, bank notes issued by a bank stating promises to redeem note for gold or silver on demand).

Each bank i sets its reserves to satisfy the equation

$$R_i = rD_i \tag{1.3}$$

where $r \in (0, 1)$ is its **reserve-deposit ratio** or **reserve ratio** for short

- the reserve ratio is either set by a government or chosen by banks for precautionary reasons

Next we add a theory stating that bank $i + 1$'s deposits depend entirely on loans made by bank i, namely

$$D_{i+1} = L_i \tag{1.4}$$

Thus, we can think of the banks as being arranged along a line with loans from bank i being immediately deposited in $i + 1$

- in this way, the debtors to bank i become creditors of bank $i + 1$

Finally, we add an *initial condition* about an exogenous level of bank 0's deposits

$$D_0 \text{ is given exogenously}$$

We can think of D_0 as being the amount of cash that a first depositor put into the first bank in the system, bank number $i = 0$.

Now we do a little algebra.

Combining equations (1.2) and (1.3) tells us that

$$L_i = (1 - r)D_i \tag{1.5}$$

This states that bank i loans a fraction $(1-r)$ of its deposits and keeps a fraction r as cash reserves.

Combining equation (1.5) with equation (1.4) tells us that

$$D_{i+1} = (1-r)D_i \text{ for } i \geq 0$$

which implies that

$$D_i = (1-r)^i D_0 \text{ for } i \geq 0 \tag{1.6}$$

Equation (1.6) expresses D_i as the i th term in the product of D_0 and the geometric series

$$1, (1-r), (1-r)^2, \cdots$$

Therefore, the sum of all deposits in our banking system $i = 0, 1, 2, \ldots$ is

$$\sum_{i=0}^{\infty}(1-r)^i D_0 = \frac{D_0}{1-(1-r)} = \frac{D_0}{r} \tag{1.7}$$

1.3.2 Money Multiplier

The **money multiplier** is a number that tells the multiplicative factor by which an exogenous injection of cash into bank 0 leads to an increase in the total deposits in the banking system.

Equation (1.7) asserts that the **money multiplier** is $\frac{1}{r}$

- An initial deposit of cash of D_0 in bank 0 leads the banking system to create total deposits of $\frac{D_0}{r}$.
- The initial deposit D_0 is held as reserves, distributed throughout the banking system according to $D_0 = \sum_{i=0}^{\infty} R_i$.

1.4 Example: The Keynesian Multiplier

The famous economist John Maynard Keynes and his followers created a simple model intended to determine national income y in circumstances in which

- there are substantial unemployed resources, in particular **excess supply** of labor and capital
- prices and interest rates fail to adjust to make aggregate **supply equal demand** (e.g., prices and interest rates are frozen)
- national income is entirely determined by aggregate demand

1.4.1 Static Version

An elementary Keynesian model of national income determination consists of three equations that describe aggregate demand for y and its components.

The first equation is a national income identity asserting that consumption c plus investment i equals national income y:

$$c + i = y$$

The second equation is a Keynesian consumption function asserting that people consume a fraction $b \in (0, 1)$ of their income:

$$c = by$$

The fraction $b \in (0, 1)$ is called the **marginal propensity to consume**.

The fraction $1 - b \in (0, 1)$ is called the **marginal propensity to save**.

The third equation simply states that investment is exogenous at level i.

- *exogenous* means *determined outside this model*.

Substituting the second equation into the first gives $(1 - b)y = i$.

Solving this equation for y gives

$$y = \frac{1}{1-b} i$$

The quantity $\frac{1}{1-b}$ is called the **investment multiplier** or simply the **multiplier**.

Applying the formula for the sum of an infinite geometric series, we can write the above equation as

$$y = i \sum_{t=0}^{\infty} b^t$$

where t is a nonnegative integer.

So we arrive at the following equivalent expressions for the multiplier:

$$\frac{1}{1-b} = \sum_{t=0}^{\infty} b^t$$

The expression $\sum_{t=0}^{\infty} b^t$ motivates an interpretation of the multiplier as the outcome of a dynamic process that we describe next.

1.4.2 Dynamic Version

We arrive at a dynamic version by interpreting the nonnegative integer t as indexing time and changing our specification of the consumption function to take time into account

- we add a one-period lag in how income affects consumption

We let c_t be consumption at time t and i_t be investment at time t.

We modify our consumption function to assume the form

$$c_t = b y_{t-1}$$

so that b is the marginal propensity to consume (now) out of last period's income.

We begin with an initial condition stating that

$$y_{-1} = 0$$

We also assume that

$$i_t = i \text{ for all } t \geq 0$$

so that investment is constant over time.

It follows that

$$y_0 = i + c_0 = i + b y_{-1} = i$$

and

$$y_1 = c_1 + i = by_0 + i = (1+b)i$$

and

$$y_2 = c_2 + i = by_1 + i = (1+b+b^2)i$$

and more generally

$$y_t = by_{t-1} + i = (1+b+b^2+\cdots+b^t)i$$

or

$$y_t = \frac{1-b^{t+1}}{1-b}i$$

Evidently, as $t \to +\infty$,

$$y_t \to \frac{1}{1-b}i$$

Remark 1: The above formula is often applied to assert that an exogenous increase in investment of Δi at time 0 ignites a dynamic process of increases in national income by successive amounts

$$\Delta i, (1+b)\Delta i, (1+b+b^2)\Delta i, \cdots$$

at times $0, 1, 2, \ldots$.

Remark 2 Let g_t be an exogenous sequence of government expenditures.

If we generalize the model so that the national income identity becomes

$$c_t + i_t + g_t = y_t$$

then a version of the preceding argument shows that the **government expenditures multiplier** is also $\frac{1}{1-b}$, so that a permanent increase in government expenditures ultimately leads to an increase in national income equal to the multiplier times the increase in government expenditures.

1.5 Example: Interest Rates and Present Values

We can apply our formula for geometric series to study how interest rates affect values of streams of dollar payments that extend over time.

We work in discrete time and assume that $t = 0, 1, 2, \ldots$ indexes time.

We let $r \in (0, 1)$ be a one-period **net nominal interest rate**

- if the nominal interest rate is 5 percent, then $r = .05$

A one-period **gross nominal interest rate** R is defined as

$$R = 1 + r \in (1, 2)$$

- if $r = .05$, then $R = 1.05$

Remark: The gross nominal interest rate R is an **exchange rate** or **relative price** of dollars at between times t and $t+1$. The units of R are dollars at time $t+1$ per dollar at time t.

When people borrow and lend, they trade dollars now for dollars later or dollars later for dollars now.

The price at which these exchanges occur is the gross nominal interest rate.

- If I sell x dollars to you today, you pay me Rx dollars tomorrow.
- This means that you borrowed x dollars for me at a gross interest rate R and a net interest rate r.

We assume that the net nominal interest rate r is fixed over time, so that R is the gross nominal interest rate at times $t = 0, 1, 2, \ldots$.

Two important geometric sequences are

$$1, R, R^2, \cdots \tag{1.8}$$

and

$$1, R^{-1}, R^{-2}, \cdots \tag{1.9}$$

Sequence (1.8) tells us how dollar values of an investment **accumulate** through time.

Sequence (1.9) tells us how to **discount** future dollars to get their values in terms of today's dollars.

1.5.1 Accumulation

Geometric sequence (1.8) tells us how one dollar invested and re-invested in a project with gross one period nominal rate of return accumulates

- here we assume that net interest payments are reinvested in the project
- thus, 1 dollar invested at time 0 pays interest r dollars after one period, so we have $r + 1 = R$ dollars at time1
- at time 1 we reinvest $1 + r = R$ dollars and receive interest of rR dollars at time 2 plus the *principal* R dollars, so we receive $rR + R = (1 + r)R = R^2$ dollars at the end of period 2
- and so on

Evidently, if we invest x dollars at time 0 and reinvest the proceeds, then the sequence

$$x, xR, xR^2, \cdots$$

tells how our account accumulates at dates $t = 0, 1, 2, \ldots$.

1.5.2 Discounting

Geometric sequence (1.9) tells us how much future dollars are worth in terms of today's dollars.

Remember that the units of R are dollars at $t + 1$ per dollar at t.

It follows that

- the units of R^{-1} are dollars at t per dollar at $t + 1$
- the units of R^{-2} are dollars at t per dollar at $t + 2$
- and so on; the units of R^{-j} are dollars at t per dollar at $t + j$

So if someone has a claim on x dollars at time $t + j$, it is worth xR^{-j} dollars at time t (e.g., today).

1.5.3 Application to Asset Pricing

A **lease** requires a payments stream of x_t dollars at times $t = 0, 1, 2, \ldots$ where

$$x_t = G^t x_0$$

where $G = (1 + g)$ and $g \in (0, 1)$.

Thus, lease payments increase at g percent per period.

For a reason soon to be revealed, we assume that $G < R$.

The **present value** of the lease is

$$\begin{aligned} p_0 &= x_0 + x_1/R + x_2/(R^2) + \ddots \\ &= x_0(1 + GR^{-1} + G^2R^{-2} + \cdots) \\ &= x_0 \frac{1}{1 - GR^{-1}} \end{aligned}$$

where the last line uses the formula for an infinite geometric series.

Recall that $R = 1 + r$ and $G = 1 + g$ and that $R > G$ and $r > g$ and that r and g are typically small numbers, e.g., .05 or .03.

Use the Taylor series of $\frac{1}{1+r}$ about $r = 0$, namely,

$$\frac{1}{1+r} = 1 - r + r^2 - r^3 + \cdots$$

and the fact that r is small to approximate $\frac{1}{1+r} \approx 1 - r$.

Use this approximation to write p_0 as

$$\begin{aligned} p_0 &= x_0 \frac{1}{1 - GR^{-1}} \\ &= x_0 \frac{1}{1 - (1+g)(1-r)} \\ &= x_0 \frac{1}{1 - (1 + g - r - rg)} \\ &\approx x_0 \frac{1}{r - g} \end{aligned}$$

where the last step uses the approximation $rg \approx 0$.

The approximation

$$p_0 = \frac{x_0}{r - g}$$

is known as the **Gordon formula** for the present value or current price of an infinite payment stream $x_0 G^t$ when the nominal one-period interest rate is r and when $r > g$.

We can also extend the asset pricing formula so that it applies to finite leases.

Let the payment stream on the lease now be x_t for $t = 1, 2, \ldots, T$, where again

$$x_t = G^t x_0$$

The present value of this lease is:

$$\begin{aligned} p_0 &= x_0 + x_1/R + \cdots + x_T/R^T \\ &= x_0(1 + GR^{-1} + \cdots + G^T R^{-T}) \\ &= \frac{x_0(1 - G^{T+1}R^{-(T+1)})}{1 - GR^{-1}} \end{aligned}$$

Applying the Taylor series to $R^{-(T+1)}$ about $r = 0$ we get:

$$\frac{1}{(1+r)^{T+1}} = 1 - r(T+1) + \frac{1}{2}r^2(T+1)(T+2) + \cdots \approx 1 - r(T+1)$$

Similarly, applying the Taylor series to G^{T+1} about $g = 0$:

$$(1+g)^{T+1} = 1 + (T+1)g + \frac{T(T+1)}{2!}g^2 + \frac{(T-1)T(T+1)}{3!}g^3 + \cdots \approx 1 + (T+1)g$$

Thus, we get the following approximation:

$$p_0 = \frac{x_0(1 - (1 + (T+1)g)(1 - r(T+1)))}{1 - (1-r)(1+g)}$$

Expanding:

$$\begin{aligned}
p_0 &= \frac{x_0(1 - 1 + (T+1)^2 rg - r(T+1) + g(T+1))}{1 - 1 + r - g + rg} \\
&= \frac{x_0(T+1)((T+1)rg + r - g)}{r - g + rg} \\
&\approx \frac{x_0(T+1)(r-g)}{r-g} + \frac{x_0 rg(T+1)}{r-g} \\
&= x_0(T+1) + \frac{x_0 rg(T+1)}{r-g}
\end{aligned}$$

We could have also approximated by removing the second term $rgx_0(T+1)$ when T is relatively small compared to $1/(rg)$ to get $x_0(T+1)$ as in the finite stream approximation.

We will plot the true finite stream present-value and the two approximations, under different values of T, and g and r in Python.

First we plot the true finite stream present-value after computing it below

```
# True present value of a finite lease
def finite_lease_pv_true(T, g, r, x_0):
    G = (1 + g)
    R = (1 + r)
    return (x_0 * (1 - G**(T + 1) * R**(-T - 1))) / (1 - G * R**(-1))
# First approximation for our finite lease

def finite_lease_pv_approx_1(T, g, r, x_0):
    p = x_0 * (T + 1) + x_0 * r * g * (T + 1) / (r - g)
    return p

# Second approximation for our finite lease
def finite_lease_pv_approx_2(T, g, r, x_0):
    return (x_0 * (T + 1))

# Infinite lease
def infinite_lease(g, r, x_0):
    G = (1 + g)
    R = (1 + r)
    return x_0 / (1 - G * R**(-1))
```

Now that we have defined our functions, we can plot some outcomes.

First we study the quality of our approximations

```
def plot_function(axes, x_vals, func, args):
    axes.plot(x_vals, func(*args), label=func.__name__)

T_max = 50

T = np.arange(0, T_max+1)
g = 0.02
r = 0.03
x_0 = 1

our_args = (T, g, r, x_0)
funcs = [finite_lease_pv_true,
        finite_lease_pv_approx_1,
        finite_lease_pv_approx_2]
        ## the three functions we want to compare

fig, ax = plt.subplots()
ax.set_title('Finite Lease Present Value $T$ Periods Ahead')
for f in funcs:
    plot_function(ax, T, f, our_args)
ax.legend()
ax.set_xlabel('$T$ Periods Ahead')
ax.set_ylabel('Present Value, $p_0$')
plt.show()
```

Evidently our approximations perform well for small values of T.

However, holding g and r fixed, our approximations deteriorate as T increases.

Next we compare the infinite and finite duration lease present values over different lease lengths T.

```
# Convergence of infinite and finite
T_max = 1000
T = np.arange(0, T_max+1)
fig, ax = plt.subplots()
ax.set_title('Infinite and Finite Lease Present Value $T$ Periods Ahead')
```

(continues on next page)

(continued from previous page)

```
f_1 = finite_lease_pv_true(T, g, r, x_0)
f_2 = np.full(T_max+1, infinite_lease(g, r, x_0))
ax.plot(T, f_1, label='T-period lease PV')
ax.plot(T, f_2, '--', label='Infinite lease PV')
ax.set_xlabel('$T$ Periods Ahead')
ax.set_ylabel('Present Value, $p_0$')
ax.legend()
plt.show()
```

The graph above shows how as duration $T \to +\infty$, the value of a lease of duration T approaches the value of a perpetual lease.

Now we consider two different views of what happens as r and g covary

```
# First view
# Changing r and g
fig, ax = plt.subplots()
ax.set_title('Value of lease of length $T$')
ax.set_ylabel('Present Value, $p_0$')
ax.set_xlabel('$T$ periods ahead')
T_max = 10
T=np.arange(0, T_max+1)

rs, gs = (0.9, 0.5, 0.4001, 0.4), (0.4, 0.4, 0.4, 0.5),
comparisons = ('$\gg$', '$>$', r'$\approx$', '$<$')
for r, g, comp in zip(rs, gs, comparisons):
    ax.plot(finite_lease_pv_true(T, g, r, x_0), label=f'r(={r}) {comp} g(={g})')

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

This graph gives a big hint for why the condition $r > g$ is necessary if a lease of length $T = +\infty$ is to have finite value.

For fans of 3-d graphs the same point comes through in the following graph.

If you aren't enamored of 3-d graphs, feel free to skip the next visualization!

```
# Second view
fig = plt.figure()
T = 3
ax = plt.subplot(projection='3d')
r = np.arange(0.01, 0.99, 0.005)
g = np.arange(0.011, 0.991, 0.005)

rr, gg = np.meshgrid(r, g)
z = finite_lease_pv_true(T, gg, rr, x_0)

# Removes points where undefined
same = (rr == gg)
z[same] = np.nan
surf = ax.plot_surface(rr, gg, z, cmap=cm.coolwarm,
    antialiased=True, clim=(0, 15))
fig.colorbar(surf, shrink=0.5, aspect=5)
ax.set_xlabel('$r$')
ax.set_ylabel('$g$')
ax.set_zlabel('Present Value, $p_0$')
ax.view_init(20, 10)
ax.set_title('Three Period Lease PV with Varying $g$ and $r$')
plt.show()
```

We can use a little calculus to study how the present value p_0 of a lease varies with r and g.

We will use a library called SymPy.

SymPy enables us to do symbolic math calculations including computing derivatives of algebraic equations.

We will illustrate how it works by creating a symbolic expression that represents our present value formula for an infinite lease.

After that, we'll use SymPy to compute derivatives

```
# Creates algebraic symbols that can be used in an algebraic expression
g, r, x0 = sym.symbols('g, r, x0')
G = (1 + g)
R = (1 + r)
p0 = x0 / (1 - G * R**(-1))
init_printing(use_latex='mathjax')
print('Our formula is:')
p0
```

```
Our formula is:
```

$$\frac{x_0}{-\frac{g+1}{r+1} + 1}$$

```
print('dp0 / dg is:')
dp_dg = sym.diff(p0, g)
dp_dg
```

```
dp0 / dg is:
```

$$\frac{x_0}{(r+1)\left(-\frac{g+1}{r+1}+1\right)^2}$$

```
print('dp0 / dr is:')
dp_dr = sym.diff(p0, r)
dp_dr
```

```
dp0 / dr is:
```

$$-\frac{x_0\,(g+1)}{(r+1)^2\left(-\frac{g+1}{r+1}+1\right)^2}$$

We can see that for $\frac{\partial p_0}{\partial r} < 0$ as long as $r > g$, $r > 0$ and $g > 0$ and x_0 is positive, so $\frac{\partial p_0}{\partial r}$ will always be negative.

Similarly, $\frac{\partial p_0}{\partial g} > 0$ as long as $r > g$, $r > 0$ and $g > 0$ and x_0 is positive, so $\frac{\partial p_0}{\partial g}$ will always be positive.

1.6 Back to the Keynesian Multiplier

We will now go back to the case of the Keynesian multiplier and plot the time path of y_t, given that consumption is a constant fraction of national income, and investment is fixed.

```
# Function that calculates a path of y
def calculate_y(i, b, g, T, y_init):
    y = np.zeros(T+1)
    y[0] = i + b * y_init + g
    for t in range(1, T+1):
        y[t] = b * y[t-1] + i + g
    return y

# Initial values
i_0 = 0.3
g_0 = 0.3
# 2/3 of income goes towards consumption
b = 2/3
y_init = 0
T = 100

fig, ax = plt.subplots()
ax.set_title('Path of Aggregate Output Over Time')
ax.set_xlabel('$t$')
ax.set_ylabel('$y_t$')
ax.plot(np.arange(0, T+1), calculate_y(i_0, b, g_0, T, y_init))
# Output predicted by geometric series
ax.hlines(i_0 / (1 - b) + g_0 / (1 - b), xmin=-1, xmax=101, linestyles='--')
plt.show()
```

In this model, income grows over time, until it gradually converges to the infinite geometric series sum of income.

We now examine what will happen if we vary the so-called **marginal propensity to consume**, i.e., the fraction of income that is consumed

```
bs = (1/3, 2/3, 5/6, 0.9)

fig,ax = plt.subplots()
ax.set_title('Changing Consumption as a Fraction of Income')
ax.set_ylabel('$y_t$')
ax.set_xlabel('$t$')
x = np.arange(0, T+1)
for b in bs:
    y = calculate_y(i_0, b, g_0, T, y_init)
    ax.plot(x, y, label=r'$b=$'+f"{b:.2f}")
ax.legend()
plt.show()
```

Increasing the marginal propensity to consume b increases the path of output over time.

Now we will compare the effects on output of increases in investment and government spending.

```
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(6, 10))
fig.subplots_adjust(hspace=0.3)

x = np.arange(0, T+1)
values = [0.3, 0.4]

for i in values:
    y = calculate_y(i, b, g_0, T, y_init)
    ax1.plot(x, y, label=f"i={i}")
for g in values:
    y = calculate_y(i_0, b, g, T, y_init)
    ax2.plot(x, y, label=f"g={g}")

axes = ax1, ax2
param_labels = "Investment", "Government Spending"
for ax, param in zip(axes, param_labels):
    ax.set_title(f'An Increase in {param} on Output')
    ax.legend(loc ="lower right")
    ax.set_ylabel('$y_t$')
    ax.set_xlabel('$t$')
plt.show()
```

Notice here, whether government spending increases from 0.3 to 0.4 or investment increases from 0.3 to 0.4, the shifts in the graphs are identical.

CHAPTER

TWO

MODELING COVID 19

Contents

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

2.1 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
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
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.

2.2 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.

2.2.1 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

$$\begin{aligned} \dot{s}(t) &= -\beta(t)\, s(t)\, i(t) \\ \dot{e}(t) &= \beta(t)\, s(t)\, i(t) - \sigma e(t) \\ \dot{i}(t) &= \sigma e(t) - \gamma i(t) \end{aligned} \tag{2.1}$$

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 (2.1) can be written in vector form as

$$\dot{x} = F(x, t), \qquad x := (s, e, i) \tag{2.2}$$

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

2.2.2 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.)

2.3 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 (2.2)

```
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
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
```

2.4 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)
```

2.4.1 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):
        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)
```

2.4.2 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)
```

2.5 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

```
v = 0.01
```

This is the cumulative number of deaths:

```
paths = [path * v * 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

THREE

LINEAR ALGEBRA

Contents

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

3.1 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,

$$\begin{aligned} y_1 &= a_{11}x_1 + a_{12}x_2 + \cdots + a_{1k}x_k \\ &\vdots \\ y_n &= a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nk}x_k \end{aligned} \tag{3.1}$$

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
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
from matplotlib import cm
from mpl_toolkits.mplot3d import Axes3D
from scipy.linalg import inv, solve, det, eig
```

3.2 Vectors

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

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()
```

3.2.1 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.])
```

3.2.2 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
```

```
12.0
```

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

```
1.7320508075688772
```

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

```
1.7320508075688772
```

3.2.3 Span

Given a set of vectors $A := \{a_1, \dots, a_k\}$ 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 $A := \{a_1, \dots, a_k\}$ if

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

In this context, the values $\beta_1, \dots, \beta_k$ 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
```

(continues on next page)

(continued from previous page)

```
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 $\mathbb{R}^3$.

3.2.4 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 $A := \{a_1, \dots, a_k\}$ 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 $A := \{a_1, \dots, a_k\} \subset \mathbb{R}^n$

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 $\beta_1, \dots, \beta_k$, then $\beta_1 = \cdots = \beta_k = 0$.

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

3.2.5 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 $A := \{a_1, \dots, a_k\} \subset \mathbb{R}^n$ is linearly independent and

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

then no other coefficient sequence $\gamma_1, \dots, \gamma_k$ 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.

3.3 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.

3.3.1 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

$$Ax = \begin{bmatrix} a_{11} & \cdots & a_{1k} \\ \vdots & \vdots & \vdots \\ a_{n1} & \cdots & a_{nk} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_k \end{bmatrix} := \begin{bmatrix} a_{11}x_1 + \cdots + a_{1k}x_k \\ \vdots \\ a_{n1}x_1 + \cdots + a_{nk}x_k \end{bmatrix} \tag{3.2}$$

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 $IA = A$.

3.3.2 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.

3.3.3 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.

3.4 Solving Systems of Equations

Recall again the system of equations (3.1).

If we compare (3.1) and (3.2), we see that (3.1) can now be written more conveniently as

$$y = Ax \tag{3.3}$$

The problem we face is to determine a vector $x \in \mathbb{R}^k$ that solves (3.3), 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)
```

(continues on next page)

(continued from previous page)

```
y = f(x)
ya, yb = np.min(y), np.max(y)

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 (3.3) 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 $a_1, \dots, a_k$ 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 $\{a_1, \dots, a_k\}$ are linearly independent and $y = Ax = x_1 a_1 + \cdots + x_k a_k$, then no $z \neq x$ satisfies $y = Az$.

3.4.1 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.

3.4.2 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.

3.4.3 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.

3.4.4 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
```

```
-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)`.

3.5 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 $\lambda_1, \dots, \lambda_n$ are its eigenvalues, then the eigenvalues of A^{-1} are $1/\lambda_1, \dots, 1/\lambda_n$.

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.

3.5.1 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.

3.6 Further Topics

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

3.6.1 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].

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

$$(I - A)^{-1} = \sum_{k=0}^{\infty} A^k \tag{3.4}$$

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 (3.4) is valid.

3.6.2 Positive Definite Matrices

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

We say that A is

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

3.6.3 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] 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.

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. $\frac{\partial y'Bz}{\partial B} = yz'$

Exercise 3.7.1 below asks you to apply these formulas.

3.6.4 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 [Janich94].

3.7 Exercises

Exercise 3.7.1

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

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

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 to Exercise 3.7.1

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

FOUR

QR DECOMPOSITION

4.1 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.

4.2 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

4.3 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.

4.3.1 Gram-Schmidt process for square A

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

In particular, let

$$A = \left[\begin{array}{c|c|c|c} a_1 & a_2 & \cdots & a_n \end{array} \right]$$

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 **orgonalize** 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

$$A = \left[\begin{array}{c|c|c|c} a_1 & a_2 & \cdots & a_n \end{array} \right] = \left[\begin{array}{c|c|c|c} e_1 & e_2 & \cdots & e_n \end{array} \right] \begin{bmatrix} a_1 \cdot e_1 & a_2 \cdot e_1 & \cdots & a_n \cdot e_1 \\ 0 & a_2 \cdot e_2 & \cdots & a_n \cdot e_2 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_n \cdot e_n \end{bmatrix}$$

Thus, we have constructed the decomposision

$$A = QR$$

where

$$Q = \left[\begin{array}{c|c|c|c} a_1 & a_2 & \cdots & a_n \end{array} \right] = \left[\begin{array}{c|c|c|c} e_1 & e_2 & \cdots & e_n \end{array} \right]$$

and

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

4.3.2 A not square

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

Then a QR decomposition is

$$A = \left[\begin{array}{c|c|c|c} a_1 & a_2 & \cdots & a_m \end{array}\right] = \left[\begin{array}{c|c|c|c} e_1 & e_2 & \cdots & e_n \end{array}\right] \begin{bmatrix} a_1 \cdot e_1 & a_2 \cdot e_1 & \cdots & a_n \cdot e_1 & a_{n+1} \cdot e_1 & \cdots & a_m \cdot e_1 \\ 0 & a_2 \cdot e_2 & \cdots & a_n \cdot e_2 & a_{n+1} \cdot e_2 & \cdots & a_m \cdot e_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_n \cdot e_n & a_{n+1} \cdot e_n & \cdots & a_m \cdot e_n \end{bmatrix}$$

which implies that

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

4.4 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
```

4.5 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 ]]),
 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 ]]))
```

4.6 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))
```

```
[-0.3112636688246115, -0.21548621711105198, 1.4525659598104073]
```

Compare with the `scipy` package.

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

```
[(-0.2633749429678319-0.2466435691315821j),
 (-0.2633749429678319+0.2466435691315821j),
 (1.4525659598104061+0j)]
```

4.7 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

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

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

```
XX = X.T @ X

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

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

```
Λ̃, Λ̂_hat
```

```
(array([  61.42527244,  149.55734339,  329.29347096, 1341.76809767,
        9054.98958835]),
 array([  61.42527244,  149.55734339,  329.29347096, 1341.76809767,
        9054.98958835]))
```

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()
```

```
7.494005416219807e-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()
```

```
6.59383658785373e-12
```

CHAPTER

FIVE

COMPLEX NUMBERS AND TRIGONOMETRY

Contents

- *Complex Numbers and Trigonometry*
  - *Overview*
  - *De Moivre's Theorem*
  - *Applications of de Moivre's Theorem*

5.1 Overview

This lecture introduces some elementary mathematics and trigonometry.

Useful and interesting in its own right, these concepts reap substantial rewards when studying dynamics generated by linear difference equations or linear differential equations.

For example, these tools are keys to understanding outcomes attained by Paul Samuelson (1939) [Sam39] in his classic paper on interactions between the investment accelerator and the Keynesian consumption function, our topic in the lecture *Samuelson Multiplier Accelerator*.

In addition to providing foundations for Samuelson's work and extensions of it, this lecture can be read as a stand-alone quick reminder of key results from elementary high school trigonometry.

So let's dive in.

5.1.1 Complex Numbers

A complex number has a **real part** x and a purely **imaginary part** y.

The Euclidean, polar, and trigonometric forms of a complex number z are:

$$z = x + iy = re^{i\theta} = r(\cos\theta + i\sin\theta)$$

The second equality above is known as **Euler's formula**

- Euler contributed many other formulas too!

The complex conjugate $\bar{z}$ of z is defined as

$$\bar{z} = x - iy = re^{-i\theta} = r(\cos\theta - i\sin\theta)$$

The value x is the **real** part of z and y is the **imaginary** part of z.

The symbol $|z| = \sqrt{\bar{z} \cdot z} = r$ represents the **modulus** of z.

The value r is the Euclidean distance of vector (x, y) from the origin:

$$r = |z| = \sqrt{x^2 + y^2}$$

The value θ is the angle of (x, y) with respect to the real axis.

Evidently, the tangent of θ is $(\frac{y}{x})$.

Therefore,

$$\theta = \tan^{-1}\left(\frac{y}{x}\right)$$

Three elementary trigonometric functions are

$$\cos\theta = \frac{x}{r} = \frac{e^{i\theta} + e^{-i\theta}}{2}, \quad \sin\theta = \frac{y}{r} = \frac{e^{i\theta} - e^{-i\theta}}{2i}, \quad \tan\theta = \frac{y}{x}$$

We'll need the following imports:

```
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
from sympy import (Symbol, symbols, Eq, nsolve, sqrt, cos, sin, simplify,
                   init_printing, integrate)
```

5.1.2 An Example

Consider the complex number $z = 1 + \sqrt{3}i$.

For $z = 1 + \sqrt{3}i$, $x = 1$, $y = \sqrt{3}$.

It follows that $r = 2$ and $\theta = \tan^{-1}(\sqrt{3}) = \frac{\pi}{3} = 60^o$.

Let's use Python to plot the trigonometric form of the complex number $z = 1 + \sqrt{3}i$.

```
# Abbreviate useful values and functions
π = np.pi


# Set parameters
r = 2
θ = π/3
x = r * np.cos(θ)
x_range = np.linspace(0, x, 1000)
θ_range = np.linspace(0, θ, 1000)

# Plot
fig = plt.figure(figsize=(8, 8))
ax = plt.subplot(111, projection='polar')

ax.plot((0, θ), (0, r), marker='o', color='b')          # Plot r
ax.plot(np.zeros(x_range.shape), x_range, color='b')       # Plot x
ax.plot(θ_range, x / np.cos(θ_range), color='b')        # Plot y
```

(continues on next page)

(continued from previous page)

```
ax.plot(θ_range, np.full(θ_range.shape, 0.1), color='r')  # Plot θ

ax.margins(0) # Let the plot starts at origin

ax.set_title("Trigonometry of complex numbers", va='bottom',
    fontsize='x-large')

ax.set_rmax(2)
ax.set_rticks((0.5, 1, 1.5, 2))  # Less radial ticks
ax.set_rlabel_position(-88.5)    # Get radial labels away from plotted line

ax.text(θ, r+0.01 , r'$z = x + iy = 1 + \sqrt{3}\, i$')   # Label z
ax.text(θ+0.2, 1 , '$r = 2$')                             # Label r
ax.text(0-0.2, 0.5, '$x = 1$')                            # Label x
ax.text(0.5, 1.2, r'$y = \sqrt{3}$')                      # Label y
ax.text(0.25, 0.15, r'$\theta = 60^o$')                   # Label θ

ax.grid(True)
plt.show()
```

5.2 De Moivre's Theorem

de Moivre's theorem states that:

$$(r(\cos\theta + i\sin\theta))^n = r^n e^{in\theta} = r^n(\cos n\theta + i\sin n\theta)$$

To prove de Moivre's theorem, note that

$$(r(\cos\theta + i\sin\theta))^n = \left(re^{i\theta}\right)^n$$

and compute.

5.3 Applications of de Moivre's Theorem

5.3.1 Example 1

We can use de Moivre's theorem to show that $r = \sqrt{x^2 + y^2}$.

We have

$$\begin{aligned} 1 &= e^{i\theta} e^{-i\theta} \\ &= (\cos\theta + i\sin\theta)(\cos(\text{-}\theta) + i\sin(\text{-}\theta)) \\ &= (\cos\theta + i\sin\theta)(\cos\theta - i\sin\theta) \\ &= \cos^2\theta + \sin^2\theta \\ &= \frac{x^2}{r^2} + \frac{y^2}{r^2} \end{aligned}$$

and thus

$$x^2 + y^2 = r^2$$

We recognize this as a theorem of **Pythagoras**.

5.3.2 Example 2

Let $z = re^{i\theta}$ and $\bar{z} = re^{-i\theta}$ so that $\bar{z}$ is the **complex conjugate** of z.

$(z, \bar{z})$ form a **complex conjugate pair** of complex numbers.

Let $a = pe^{i\omega}$ and $\bar{a} = pe^{-i\omega}$ be another complex conjugate pair.

For each element of a sequence of integers $n = 0, 1, 2, \ldots,$.

To do so, we can apply de Moivre's formula.

Thus,

$$\begin{aligned} x_n &= az^n + \bar{a}\bar{z}^n \\ &= pe^{i\omega}(re^{i\theta})^n + pe^{-i\omega}(re^{-i\theta})^n \\ &= pr^n e^{i(\omega+n\theta)} + pr^n e^{-i(\omega+n\theta)} \\ &= pr^n[\cos(\omega+n\theta) + i\sin(\omega+n\theta) + \cos(\omega+n\theta) - i\sin(\omega+n\theta)] \\ &= 2pr^n\cos(\omega+n\theta) \end{aligned}$$

5.3.3 Example 3

This example provides machinery that is at the heard of Samuelson's analysis of his multiplier-accelerator model [Sam39].

Thus, consider a **second-order linear difference equation**

$$x_{n+2} = c_1 x_{n+1} + c_2 x_n$$

whose **characteristic polynomial** is

$$z^2 - c_1 z - c_2 = 0$$

or

$$(z^2 - c_1 z - c_2) = (z - z_1)(z - z_2) = 0$$

has roots z_1, z_1.

A **solution** is a sequence $\{x_n\}_{n=0}^{\infty}$ that satisfies the difference equation.

Under the following circumstances, we can apply our example 2 formula to solve the difference equation

- the roots z_1, z_2 of the characteristic polynomial of the difference equation form a complex conjugate pair
- the values x_0, x_1 are given initial conditions

To solve the difference equation, recall from example 2 that

$$x_n = 2pr^n \cos(\omega + n\theta)$$

where ω, p are coefficients to be determined from information encoded in the initial conditions x_1, x_0.

Since $x_0 = 2p\cos\omega$ and $x_1 = 2pr\cos(\omega + \theta)$ the ratio of x_1 to x_0 is

$$\frac{x_1}{x_0} = \frac{r\cos(\omega + \theta)}{\cos\omega}$$

We can solve this equation for ω then solve for p using $x_0 = 2pr^0 \cos(\omega + n\theta)$.

With the `sympy` package in Python, we are able to solve and plot the dynamics of x_n given different values of n.

In this example, we set the initial values: - $r = 0.9$ - $\theta = \frac{1}{4}\pi$ - $x_0 = 4$ - $x_1 = r \cdot 2\sqrt{2} = 1.8\sqrt{2}$.

We first numerically solve for ω and p using `nsolve` in the `sympy` package based on the above initial condition:

```
# Set parameters
r = 0.9
θ = π/4
x0 = 4
x1 = 2 * r * sqrt(2)

# Define symbols to be calculated
ω, p = symbols('ω p', real=True)

# Solve for ω
## Note: we choose the solution near 0
eq1 = Eq(x1/x0 - r * cos(ω+θ) / cos(ω), 0)
ω = nsolve(eq1, ω, 0)
ω = float(ω)
print(f'ω = {ω:1.3f}')

# Solve for p
eq2 = Eq(x0 - 2 * p * cos(ω), 0)
p = nsolve(eq2, p, 0)
p = float(p)
print(f'p = {p:1.3f}')
```

```
ω = 0.000
p = 2.000
```

Using the code above, we compute that $\omega = 0$ and $p = 2$.

Then we plug in the values we solve for ω and p and plot the dynamic.

```
# Define range of n
max_n = 30
n = np.arange(0, max_n+1, 0.01)

# Define x_n
x = lambda n: 2 * p * r**n * np.cos(ω + n * θ)

# Plot
fig, ax = plt.subplots(figsize=(12, 8))

ax.plot(n, x(n))
ax.set(xlim=(0, max_n), ylim=(-5, 5), xlabel='$n$', ylabel='$x_n$')

# Set x-axis in the middle of the plot
ax.spines['bottom'].set_position('center')
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.xaxis.set_ticks_position('bottom')
ax.yaxis.set_ticks_position('left')

ticklab = ax.xaxis.get_ticklabels()[0] # Set x-label position
trans = ticklab.get_transform()
ax.xaxis.set_label_coords(31, 0, transform=trans)

ticklab = ax.yaxis.get_ticklabels()[0] # Set y-label position
trans = ticklab.get_transform()
ax.yaxis.set_label_coords(0, 5, transform=trans)

ax.grid()
plt.show()
```

5.3.4 Trigonometric Identities

We can obtain a complete suite of trigonometric identities by appropriately manipulating polar forms of complex numbers.

We'll get many of them by deducing implications of the equality

$$e^{i(\omega+\theta)} = e^{i\omega}e^{i\theta}$$

For example, we'll calculate identities for

$\cos(\omega+\theta)$ and $\sin(\omega+\theta)$.

Using the sine and cosine formulas presented at the beginning of this lecture, we have:

$$\cos(\omega+\theta) = \frac{e^{i(\omega+\theta)} + e^{-i(\omega+\theta)}}{2}$$
$$\sin(\omega+\theta) = \frac{e^{i(\omega+\theta)} - e^{-i(\omega+\theta)}}{2i}$$

We can also obtain the trigonometric identities as follows:

$$\begin{aligned}\cos(\omega+\theta) + i\sin(\omega+\theta) &= e^{i(\omega+\theta)} \\ &= e^{i\omega}e^{i\theta} \\ &= (\cos\omega + i\sin\omega)(\cos\theta + i\sin\theta) \\ &= (\cos\omega\cos\theta - \sin\omega\sin\theta) + i(\cos\omega\sin\theta + \sin\omega\cos\theta)\end{aligned}$$

Since both real and imaginary parts of the above formula should be equal, we get:

$$\cos(\omega+\theta) = \cos\omega\cos\theta - \sin\omega\sin\theta$$
$$\sin(\omega+\theta) = \cos\omega\sin\theta + \sin\omega\cos\theta$$

The equations above are also known as the **angle sum identities**. We can verify the equations using the `simplify` function in the `sympy` package:

```
# Define symbols
ω, θ = symbols('ω θ', real=True)

# Verify
print("cos(ω)cos(θ) - sin(ω)sin(θ) =",
    simplify(cos(ω)*cos(θ) - sin(ω) * sin(θ)))
print("cos(ω)sin(θ) + sin(ω)cos(θ) =",
    simplify(cos(ω)*sin(θ) + sin(ω) * cos(θ)))
```

```
cos(ω)cos(θ) - sin(ω)sin(θ) = cos(θ + ω)
cos(ω)sin(θ) + sin(ω)cos(θ) = sin(θ + ω)
```

5.3.5 Trigonometric Integrals

We can also compute the trigonometric integrals using polar forms of complex numbers.

For example, we want to solve the following integral:

$$\int_{-\pi}^{\pi} \cos(\omega)\sin(\omega)\,d\omega$$

Using Euler's formula, we have:

$$
\begin{aligned}
\int \cos(\omega)\sin(\omega)\,d\omega &= \int \frac{(e^{i\omega}+e^{-i\omega})}{2}\frac{(e^{i\omega}-e^{-i\omega})}{2i}\,d\omega \\
&= \frac{1}{4i}\int e^{2i\omega}-e^{-2i\omega}\,d\omega \\
&= \frac{1}{4i}\left(\frac{-i}{2}e^{2i\omega}-\frac{i}{2}e^{-2i\omega}+C_1\right) \\
&= -\frac{1}{8}\left[\left(e^{i\omega}\right)^2+\left(e^{-i\omega}\right)^2-2\right]+C_2 \\
&= -\frac{1}{8}(e^{i\omega}-e^{-i\omega})^2+C_2 \\
&= \frac{1}{2}\left(\frac{e^{i\omega}-e^{-i\omega}}{2i}\right)^2+C_2 \\
&= \frac{1}{2}\sin^2(\omega)+C_2
\end{aligned}
$$

and thus:

$$
\int_{-\pi}^{\pi}\cos(\omega)\sin(\omega)\,d\omega = \frac{1}{2}\sin^2(\pi)-\frac{1}{2}\sin^2(-\pi)=0
$$

We can verify the analytical as well as numerical results using `integrate` in the `sympy` package:

```
# Set initial printing
init_printing()

ω = Symbol('ω')
print('The analytical solution for integral of cos(ω)sin(ω) is:')
integrate(cos(ω) * sin(ω), ω)
```

```
The analytical solution for integral of cos(ω)sin(ω) is:
```

$$
\frac{\sin^2(\omega)}{2}
$$

```
print('The numerical solution for the integral of cos(ω)sin(ω) \
from -π to π is:')
integrate(cos(ω) * sin(ω), (ω, -π, π))
```

```
The numerical solution for the integral of cos(ω)sin(ω) from -π to π is:
```

$$
0
$$

5.3.6 Exercises

Exercise 5.3.1

We invite the reader to verify analytically and with the `sympy` package the following two equalities:

$$\int_{-\pi}^{\pi} \cos(\omega)^2 \, d\omega = \pi$$

$$\int_{-\pi}^{\pi} \sin(\omega)^2 \, d\omega = \pi$$

Solution to Exercise 5.3.1

Let's import symbolic π from `sympy`

```
# Import symbolic π from sympy
from sympy import pi
```

```
print('The analytical solution for the integral of cos(ω)**2 \
from -π to π is:')

integrate(cos(ω)**2, (ω, -pi, pi))
```

```
The analytical solution for the integral of cos(ω)**2 from -π to π is:
```

$$\pi$$

```
print('The analytical solution for the integral of sin(ω)**2 \
from -π to π is:')

integrate(sin(ω)**2, (ω, -pi, pi))
```

```
The analytical solution for the integral of sin(ω)**2 from -π to π is:
```

$$\pi$$

CHAPTER

SIX

CIRCULANT MATRICES

6.1 Overview

This lecture describes circulant matrices and some of their properties.

Circulant matrices have a special structure that connects them to useful concepts including

- convolution
- Fourier transforms
- permutation matrices

Because of these connections, circulant matrices are widely used in machine learning, for example, in image processing.

We begin by importing some Python packages

```
import numpy as np
from numba import njit
import matplotlib.pyplot as plt
```

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

6.2 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:

$$C = \begin{bmatrix} c_0 & c_1 & c_2 & c_3 & c_4 & \cdots & c_{N-1} \\ c_{N-1} & c_0 & c_1 & c_2 & c_3 & \cdots & c_{N-2} \\ c_{N-2} & c_{N-1} & c_0 & c_1 & c_2 & \cdots & c_{N-3} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ c_3 & c_4 & c_5 & c_6 & c_7 & \cdots & c_2 \\ c_2 & c_3 & c_4 & c_5 & c_6 & \cdots & c_1 \\ c_1 & c_2 & c_3 & c_4 & c_5 & \cdots & c_0 \end{bmatrix} \tag{6.1}$$

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.

```
@njit
def construct_cirlulant(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_cirlulant(np.array([1., 2., 3.]))
```

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

6.2.1 Some Properties of Circulant Matrices

Here are some useful properties:

Suppose that A and B are both circulant matrices. 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

$$b_k = \sum_{i=0}^{n-1} c_{k-i} a_i \tag{6.2}$$

We use $*$ to denote **convolution** via the calculation described in equation (6.2).

It can be verified that the vector b satisfies

$$b = C^T a$$

where C^T is the transpose of the circulant matrix defined in equation (6.1).

6.3 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 $\{1, 2, \dots, n\}$ 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

$$P = \begin{bmatrix} 0 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 1 \\ 1 & 0 & 0 & 0 & \cdots & 0 \end{bmatrix} \tag{6.3}$$

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 (6.3) 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 (6.3) all equal 1.

It can be verified that permutation matrices are orthogonal matrices:

$$PP' = I$$

6.4 Examples with Python

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

```
@njit
def construct_P(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_P(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. +0.5j  0. -0.5j -0.5+0.j ]

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

λ2 = 0.0-1.0j
vec2 = [-0.5+0.j   0. -0.5j  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_P(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 Discete 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,
```

(continues on next page)

(continued from previous page)

```
          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,  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_P(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])
```

6.5 Associated Permutation Matrix

Next, we execute calculations to verify that the circulant matrix C defined in equation (6.1) 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([0.315, 0.091, 0.849, 0.769, 0.442, 0.324, 0.291, 0.411])
```

```
C8 = construct_cirlulant(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([[0.315, 0.091, 0.849, 0.769, 0.442, 0.324, 0.291, 0.411],
       [0.411, 0.315, 0.091, 0.849, 0.769, 0.442, 0.324, 0.291],
       [0.291, 0.411, 0.315, 0.091, 0.849, 0.769, 0.442, 0.324],
       [0.324, 0.291, 0.411, 0.315, 0.091, 0.849, 0.769, 0.442],
       [0.442, 0.324, 0.291, 0.411, 0.315, 0.091, 0.849, 0.769],
       [0.769, 0.442, 0.324, 0.291, 0.411, 0.315, 0.091, 0.849],
       [0.849, 0.769, 0.442, 0.324, 0.291, 0.411, 0.315, 0.091],
       [0.091, 0.849, 0.769, 0.442, 0.324, 0.291, 0.411, 0.315]])
```

```
C8
```

```
array([[0.315, 0.091, 0.849, 0.769, 0.442, 0.324, 0.291, 0.411],
       [0.411, 0.315, 0.091, 0.849, 0.769, 0.442, 0.324, 0.291],
       [0.291, 0.411, 0.315, 0.091, 0.849, 0.769, 0.442, 0.324],
       [0.324, 0.291, 0.411, 0.315, 0.091, 0.849, 0.769, 0.442],
       [0.442, 0.324, 0.291, 0.411, 0.315, 0.091, 0.849, 0.769],
       [0.769, 0.442, 0.324, 0.291, 0.411, 0.315, 0.091, 0.849],
       [0.849, 0.769, 0.442, 0.324, 0.291, 0.411, 0.315, 0.091],
       [0.091, 0.849, 0.769, 0.442, 0.324, 0.291, 0.411, 0.315]])
```

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

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

```
0.0
```

The kth column of P_8 associated with eigenvalue w^{k-1} is an eigenvector of C_8 associated with an eigenvalue $\sum_{h=0}^{7} c_j w^{hk}$.

```
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([ 3.491+0.j   , -0.545-0.646j, -0.383+0.766j,  0.29 +0.469j,
        0.303-0.j   ,  0.29 -0.469j, -0.383-0.766j, -0.545+0.646j])
```

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] - ?_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]
[-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]
```

6.6 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 number $\{x_j\}_{j=0}^{N-1}$.

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('j')

    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_transform(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_transform(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])
```

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 availble 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)

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 tranform 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 a 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

SEVEN

SINGULAR VALUE DECOMPOSITION (SVD)

7.1 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).

7.2 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)

7.3 Singular Value Decomposition

A **singular value decomposition** of an $m \times n$ matrix X of rank $p \leq \min(m, n)$ is

$$X = U \Sigma V^\top \tag{7.1}$$

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 (7.1) 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.

7.4 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 $u_i, i = 1, \dots, m$ be the m column vectors of U and let $v_i, i = 1, \dots, n$ be the n column vectors of V.

Let's write the full SVD of X as

$$X = \begin{bmatrix} U_L & U_R \end{bmatrix} \begin{bmatrix} \Sigma_p & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} V_L & V_R \end{bmatrix}^\top \tag{7.2}$$

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 (7.2) 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

$$\begin{aligned} XV_L &= U_L \Sigma_p \\ XV_R &= 0 \end{aligned} \tag{7.3}$$

or

$$\begin{aligned} Xv_i &= \sigma_i u_i, \quad i = 1, \dots, p \\ Xv_i &= 0, \quad i = p+1, \dots, n \end{aligned} \tag{7.4}$$

Equations (7.4) 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 (7.3) 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 (7.2) 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

$$\begin{aligned} X^\top U_L &= V_L \Sigma_p \\ X^\top U_R &= 0 \end{aligned} \tag{7.5}$$

or

$$\begin{aligned} X^\top u_i &= \sigma_i v_i, \quad i = 1, \dots, p \\ X^\top u_i &= 0 \quad i = p+1, \dots, m \end{aligned} \tag{7.6}$$

Notice how equations (7.6) 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 (7.5) 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 (7.3) and (7.5) describe the four fundamental subspaces of X in the following ways:

$$\begin{aligned}\mathcal{C}(X) &= \mathcal{C}(U_L)\\ \mathcal{N}(X^\top) &= \mathcal{C}(U_R)\\ \mathcal{R}(X) &\equiv \mathcal{C}(X^\top) = \mathcal{C}(V_L)\\ \mathcal{N}(X) &= \mathcal{C}(V_R)\end{aligned} \tag{7.7}$$

Since U and V are both orthonormal matrices, collection (7.7) 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 (7.7) simply by performing the multiplications called for by the right side of (7.2) and reading them.

The claims in (7.7) 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:
```

(continues on next page)

(continued from previous page)

```
 [[ 0.63 -0.06  0.06]
 [-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]]
```

7.5 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

$$\hat{X}_r = \sigma_1 U_1 V_1^\top + \sigma_2 U_2 V_2^\top + \cdots + \sigma_r U_r V_r^\top \tag{7.8}$$

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).

7.6 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([[-0.66,  0.29, -0.44, -0.46, -0.27],
        [-0.16, -0.14, -0.31,  0.75, -0.55],
        [-0.54, -0.24,  0.77,  0.  , -0.22],
        [-0.23,  0.77,  0.18,  0.43,  0.36],
        [-0.45, -0.49, -0.28,  0.19,  0.67]]),
 array([1.91, 0.9 ]),
 array([[-0.85, -0.52],
        [-0.52,  0.85]]))
```

```
print('Uhat, Shat, Vhat = ')
Uhat, Shat, Vhat
```

```
Uhat, Shat, Vhat =
```

```
(array([[-0.66,  0.29],
        [-0.16, -0.14],
        [-0.54, -0.24],
        [-0.23,  0.77],
        [-0.45, -0.49]]),
 array([1.91, 0.9 ]),
 array([[-0.85, -0.52],
        [-0.52,  0.85]]))
```

```
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,  1.04e-16,  5.69e-17,  1.43e-16,  1.66e-16],
        [ 1.04e-16,  1.00e+00,  1.52e-17,  6.82e-18,  5.37e-17],
        [ 5.69e-17,  1.52e-17,  1.00e+00, -3.82e-17,  6.97e-17],
        [ 1.43e-16,  6.82e-18, -3.82e-17,  1.00e+00, -1.32e-16],
        [ 1.66e-16,  5.37e-17,  6.97e-17, -1.32e-16,  1.00e+00]]),
 array([[ 1.00e+00, -7.93e-17,  1.19e-16,  1.75e-17, -2.25e-17],
        [-7.93e-17,  1.00e+00,  5.68e-18,  1.58e-16,  2.91e-17],
        [ 1.19e-16,  5.68e-18,  1.00e+00,  2.59e-17,  3.72e-17],
        [ 1.75e-17,  1.58e-16,  2.59e-17,  1.00e+00, -2.33e-17],
        [-2.25e-17,  2.91e-17,  3.72e-17, -2.33e-17,  1.00e+00]]))
```

```
UhatUhatT = Uhat@Uhat.T
UhatTUhat = Uhat.T@Uhat
print('UhatUhatT, UhatTUhat= ')
UhatUhatT, UhatTUhat
```

```
UhatUhatT, UhatTUhat=
```

```
(array([[ 0.51,  0.06,  0.28,  0.38,  0.15],
        [ 0.06,  0.04,  0.12, -0.07,  0.14],
        [ 0.28,  0.12,  0.35, -0.06,  0.36],
        [ 0.38, -0.07, -0.06,  0.65, -0.27],
        [ 0.15,  0.14,  0.36, -0.27,  0.44]]),
 array([[ 1.00e+00, -7.93e-17],
        [-7.93e-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([[ 0.69, -0.72],
        [ 0.72,  0.69]]),
 array([2.17, 0.42]),
 array([[ 0.57,  0.42,  0.4 ,  0.33,  0.47],
        [ 0.18, -0.11,  0.68, -0.66, -0.23],
```

(continues on next page)

(continued from previous page)

```
        [-0.71,  0.49,  0.44,  0.21, -0.09],
        [-0.01, -0.67,  0.42,  0.59, -0.15],
        [-0.36, -0.33,  0.09, -0.25,  0.83]]))
```

```
print('Uhat, Shat, Vhat = ')
Uhat, Shat, Vhat
```

```
Uhat, Shat, Vhat =
```

```
(array([[ 0.69, -0.72],
        [ 0.72,  0.69]]),
 array([2.17, 0.42]),
 array([[ 0.57,  0.42,  0.4 ,  0.33,  0.47],
        [ 0.18, -0.11,  0.68, -0.66, -0.23]]))
```

Let's verify that our reduced SVD accurately represents X

```
SShat=np.diag(Shat)
np.allclose(X, Uhat@SShat@Vhat)
```

```
True
```

7.7 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

7.8 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 $j = 1, \dots, n$ 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.

$$\bar{X}_j = \frac{1}{m} \sum_{i=1}^{m} x_{i,j}$$

We then create an average matrix out of these means:

$$\bar{X} = \begin{bmatrix} 1 \\ 1 \\ \dots \\ 1 \end{bmatrix} [\bar{X}_1 \mid \bar{X}_2 \mid \cdots \mid \bar{X}_n]$$

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.

$$C = \frac{1}{n} B^\top B$$

Step 3: Decompose the covariance matrix and arrange the singular values:

If the matrix C is diagonalizable, we can eigendecompose it, find its eigenvalues and rearrange the eigenvalue and eigenvector matrices in a decreasing other.

If C is not diagonalizable, we can perform an SVD of C:

$$\begin{aligned} B^T B &= V\Sigma^\top U^\top U \Sigma V^\top \\ &= V\Sigma^\top \Sigma V^\top \end{aligned}$$

$$C = \frac{1}{n} V\Sigma^\top \Sigma V^\top$$

We can then rearrange the columns in the matrices V 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:

$$\begin{aligned} T &= BV \\ &= U\Sigma V^\top \\ &= U\Sigma \end{aligned}$$

7.9 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.

$$X = U\Sigma V^\top = \sigma_1 U_1 V_1^\top + \sigma_2 U_2 V_2^\top + \cdots + \sigma_p U_p V_p^\top \tag{7.9}$$

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 (7.9), each of the $m \times n$ matrices $U_j V_j^\top$ is evidently of rank 1.

Thus, we have

$$X = \sigma_1 \begin{pmatrix} U_{11} V_1^\top \\ U_{21} V_1^\top \\ \ldots \\ U_{m1} V_1^\top \end{pmatrix} + \sigma_2 \begin{pmatrix} U_{12} V_2^\top \\ U_{22} V_2^\top \\ \ldots \\ U_{m2} V_2^\top \end{pmatrix} + \ldots + \sigma_p \begin{pmatrix} U_{1p} V_p^\top \\ U_{2p} V_p^\top \\ \ldots \\ U_{mp} V_p^\top \end{pmatrix} \tag{7.10}$$

Here is how we would interpret the objects in the matrix equation (7.10) 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**

- $U_j = \begin{bmatrix} U_{1k} \\ U_{2k} \\ \dots \\ U_{mk} \end{bmatrix}$ $k = 1, \dots, m$ is a vector of **loadings** of variables X_i on the kth principal component, $i = 1, \dots, m$
- σ_k for each $k = 1, \dots, p$ is the strength of kth **principal component**, where strength means contribution to the overall covariance of X.

7.10 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

$$XX^\top = P \Lambda P^\top. \tag{7.11}$$

It follows that we can represent the data matrix X as

$$X = [X_1 | X_2 | \dots | X_m] = [P_1 | P_2 | \dots | P_m] \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \dots \\ \epsilon_m \end{bmatrix} = P_1 \epsilon_1 + P_2 \epsilon_2 + \dots + P_m \epsilon_m$$

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

7.11 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:

$$\begin{aligned} XX^\top &= U \Sigma V^\top V \Sigma^\top U^\top \\ &\equiv U \Sigma \Sigma^\top U^\top \\ &\equiv U \Lambda U^\top \end{aligned} \tag{7.12}$$

Compare representation (7.12) with equation (7.11) 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):

        ?, P = LA.eigh(self.Ω)     # columns of P are eigenvectors

        ind = sorted(range(?.size), key=lambda x: ?[x], reverse=True)

        # sort by eigenvalues
        self.? = ?[ind]
        P = P[:, ind]
        self.P = P @ diag_sign(P)

        self.Λ = np.diag(self.?)

        self.explained_ratio_pca = np.cumsum(self.?) / self.?.sum()

        # compute the N by T matrix of principal components
        self.? = self.P.T @ self.X

        P = self.P[:, :self.r_component]
```

(continues on next page)

(continued from previous page)

```
        ▯ = self.▯[:self.r_component, :]

        # transform data
        self.X_pca = P @ ▯

    def svd(self):

        U, ▯, VT = LA.svd(self.X)

        ind = sorted(range(▯.size), key=lambda x: ▯[x], reverse=True)

        # sort by eigenvalues
        d = min(self.m, self.n)

        self.▯ = ▯[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.▯)

        ▯_sq = self.▯ ** 2
        self.explained_ratio_svd = np.cumsum(▯_sq) / ▯_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]
        ▯ = self.▯[:r_component, :]

        # transform data
        self.X_pca = P @ ▯

        # 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)))
```

(continues on next page)

(continued from previous page)

```
    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()
    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('$V^\top *\sqrt{\lambda}$')
    axs[1].set_xlabel('n')
    plt.show()
```

7.12 Exercises

Exercise 7.12.1

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 to Exercise 7.12.1

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

EIGHT

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

8.1 First-Order Vector Autoregressions

We want to fit a **first-order vector autoregression**

$$X_{t+1} = AX_t + C\epsilon_{t+1}, \quad \epsilon_{t+1} \perp X_t \tag{8.1}$$

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

$$X_t = \begin{bmatrix} X_{1,t} & X_{2,t} & \cdots & X_{m,t} \end{bmatrix}^\top \tag{8.2}$$

and where $\cdot^\top$ again denotes complex transposition and $X_{i,t}$ is variable i at time t.

We want to fit equation (8.1).

Our data are organized in an $m \times (n+1)$ matrix $\tilde{X}$

$$\tilde{X} = [X_1 \mid X_2 \mid \cdots \mid X_n \mid X_{n+1}]$$

where for $t = 1, \ldots, n+1$, the $m \times 1$ vector X_t is given by (8.2).

Thus, we want to estimate a system (8.1) that consists of m least squares regressions of **everything** on one lagged value of **everything**.

The i'th equation of (8.1) 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

$$X = [X_1 \mid X_2 \mid \cdots \mid X_n]$$

and

$$X' = [X_2 \mid X_3 \mid \cdots \mid X_{n+1}]$$

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

$$\hat{A} = X' X^+ \tag{8.3}$$

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 (8.3) for the least-squares estimator of the population matrix of regression coefficients A becomes

$$\hat{A} = X' X^\top (XX^\top)^{-1} \tag{8.4}$$

This formula for least-squares regression coefficients is widely used in econometrics.

It is used to estimate vector autorgressions.

The right side of formula (8.4) 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 (8.3) for a least-squares estimator of A becomes

$$\hat{A} = X' (X^\top X)^{-1} X^\top \tag{8.5}$$

Please compare formulas (8.4) and (8.5) for $\hat{A}$.

Here we are especially interested in formula (8.5).

The ith row of $\hat{A}$ is an $m \times 1$ vector of regression coefficients of $X_{i,t+1}$ on $X_{j,t}, j = 1, \dots, m$.

If we use formula (8.5) 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 (8.1).

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

$$\hat{A} = \text{argmin}_{\check{A}} ||X' - \check{A}X||_F \tag{8.6}$$

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 (8.6) is

$$\hat{A} = X' X^+ \tag{8.7}$$

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 (8.5).

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 $X_{j,t}, j = 1, \dots, m$.

An efficient way to compute the pseudo-inverse X^+ is to start with a singular value decomposition

$$X = U \Sigma V^\top \tag{8.8}$$

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.

$$\begin{aligned} X^+ &= (X^\top X)^{-1} X^\top \\ &= (V \Sigma U^\top U \Sigma V^\top)^{-1} V \Sigma U^\top \\ &= (V \Sigma \Sigma V^\top)^{-1} V \Sigma U^\top \\ &= V \Sigma^{-1} \Sigma^{-1} V^\top V \Sigma U^\top \\ &= V \Sigma^{-1} U^\top \end{aligned} \tag{8.9}$$

(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 (8.8) to compute

$$X^+ = V\Sigma^{-1}U^\top \tag{8.10}$$

where the matrix Σ^{-1} is constructed by replacing each non-zero element of Σ with σ_j^{-1}.

We can use formula (8.10) together with formula (8.7) 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

$$\hat{A} = X'V\Sigma^{-1}U^\top \tag{8.11}$$

8.2 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 [Sch10],

You can read about Dynamic Mode Decomposition here [KBBWP16] and here [BK19] (section 7.2).

Dynamic Mode Decomposition (DMD) computes a rank $r < p$ approximation to the least squares regression coefficients $\hat{A}$ described by formula (8.11).

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.

8.3 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

$$\tilde{b}_t = U^\top X_t. \tag{8.12}$$

The original data X_t can be represented as

$$X_t = U\tilde{b}_t \tag{8.13}$$

(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 (8.12) that we can reconstruct X_t from $\tilde{b}_t$.

In particular,

- Equation (8.12) serves as an **encoder** that **rotates** the $m \times 1$ vector X_t to become an $m \times 1$ vector $\tilde{b}_t$
- Equation (8.13) 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

$$\tilde{A} = U^\top \hat{A} U \tag{8.14}$$

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.

8.4 Representation 2

This representation is related to one originally proposed by [Sch10].

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 (8.14):

$$\tilde{A} = W\Lambda W^{-1} \tag{8.15}$$

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

$$\hat{A} = U\tilde{A}U^\top = UW\Lambda W^{-1}U^\top \tag{8.16}$$

According to equation (8.16), 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:

$$\overline{X}_{t+1} = UW\Lambda^t W^{-1}U^\top X_1 \tag{8.17}$$

In effect, [Sch10] defined an $m \times m$ matrix Φ_s as

$$\Phi_s = UW \tag{8.18}$$

and a generalized inverse

$$\Phi_s^+ = W^{-1}U^\top \tag{8.19}$$

[Sch10] then represented equation (8.17) as

$$\overline{X}_{t+1} = \Phi_s \Lambda^t \Phi_s^+ X_1 \tag{8.20}$$

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. [TRL+14].

It is more appropriate to use representation 3 when, as is often the case in practice, we want to use a reduced SVD.

8.5 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

$$\hat{A} = X'\tilde{V}\tilde{\Sigma}^{-1}\tilde{U}^\top \tag{8.21}$$

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

$$\tilde{A} = \tilde{U}^\top \hat{A}\tilde{U} \tag{8.22}$$

Interpretation as projection coefficients

[BK22] 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

$$\tilde{A} = \tilde{U}^\top \hat{A}\tilde{U} = \tilde{U}^\top X'\tilde{V}\tilde{\Sigma}^{-1}\tilde{U}^\top\tilde{U} = \tilde{U}^\top X'\tilde{V}\tilde{\Sigma}^{-1}\tilde{U}^\top \tag{8.23}$$

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 $\tilde{A}$:

$$\tilde{A} = \tilde{W}\Lambda\tilde{W}^{-1} \tag{8.24}$$

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

$$\tilde{\Phi}_s = \tilde{U}\tilde{W} \tag{8.25}$$

that corresponds to (8.18) for a full SVD.

At this point, where $\hat{A}$ is given by formula (8.21) 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

$$\Phi \equiv \hat{A}\tilde{\Phi}_s = X'\tilde{V}\tilde{\Sigma}^{-1}\tilde{W} \tag{8.26}$$

It turns out that columns of Φ **are** eigenvectors of $\hat{A}$.

This is a consequence of a result established by Tu et al. [TRL+14] that we now present.

Proposition The p columns of Φ are eigenvectors of $\hat{A}$.

Proof: From formula (8.26) 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

$$\hat{A}\Phi = \Phi\Lambda. \tag{8.27}$$

Let ϕ_i be the ith column of Φ and λ_i be the corresponding i eigenvalue of $\tilde{A}$ from decomposition (8.24).

Equating the $m \times 1$ vectors that appear on the two sides of equation (8.27) 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 $\hat{A}$.

This concludes the proof.

Also see [BK22] (p. 238)

8.5.1 Decoder of $\check{b}$ as a linear projection

From eigendecomposition (8.27) we can represent $\hat{A}$ as

$$\hat{A} = \Phi\Lambda\Phi^+. \tag{8.28}$$

From formula (8.28) we can deduce dynamics of the $p \times 1$ vector $\check{b}_t$:

$$\check{b}_{t+1} = \Lambda\check{b}_t$$

where

$$\check{b}_t = \Phi^+ X_t \tag{8.29}$$

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

$$\check{b} = (\Phi^\top \Phi)^{-1} \Phi^\top X \tag{8.30}$$

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

$$\check{X} = \Phi \check{b} \tag{8.31}$$

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

$$X = \Phi \check{b} + \epsilon \tag{8.32}$$

where ϵ is an $m \times n$ matrix of least squares errors satisfying the least squares orthogonality conditions $\epsilon^\top \Phi = 0$ or

$$(X - \Phi \check{b})^\top \Phi = 0_{m \times p} \tag{8.33}$$

Rearranging the orthogonality conditions (8.33) gives $X^\top \Phi = \check{b} \Phi^\top \Phi$, which implies formula (8.30).

8.5.2 An Approximation

We now describe a way to approximate the $p \times 1$ vector $\check{b}_t$ instead of using formula (8.29).

In particular, the following argument adapted from [BK22] (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 (8.32) we have

$$\check{X}_1 = \Phi \check{b}_1 \tag{8.34}$$

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 (8.32) 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 (8.32).

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 (8.23), $\tilde{A} = \tilde{U}^\top X' \tilde{V} \tilde{\Sigma}^{-1}$.

It then follows that

$$\hat{b}_1 = \tilde{A} \tilde{W} \check{b}_1$$

and therefore, by the eigendecomposition (8.24) 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

$$\hat{b}_1 = (\tilde{W} \Lambda)^{-1} \tilde{U}^\top X_1, \tag{8.35}$$

which is a computationally efficient approximation to the following instance of equation (8.29) for the initial vector $\check{b}_1$:

$$\check{b}_1 = \Phi^+ X_1 \tag{8.36}$$

(To highlight that (8.35) 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

$$\check{X}_{t+j} = \Phi \Lambda^j \Phi^+ X_t \tag{8.37}$$

or use compute a decoded $\hat{X}_{t+j}$ from approximate modes via

$$\hat{X}_{t+j} = \Phi \Lambda^j (\tilde{W} \Lambda)^{-1} \tilde{U}^\top X_t. \tag{8.38}$$

We can then use a decoded $\check{X}_{t+j}$ or $\hat{X}_{t+j}$ to forecast X_{t+j}.

8.5.3 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.

8.6 Source for Some Python Code

You can find a Python implementation of DMD here:

https://mathlab.sissa.it/pydmd

CHAPTER

NINE

USING NEWTON'S METHOD TO SOLVE ECONOMIC MODELS

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*

See also:

GPU: A version of this lecture which makes use of jax to run the code on a `GPU` is available here

9.1 Overview

Many economic problems involve finding fixed points or zeros (sometimes 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 multi-dimensional 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 multi-dimensional settings to solve market for equilibria with multiple goods.

At the end of the lecture we leverage the power of automatic differentiation in `autograd` to solve a very high-dimensional equilibrium problem

```
!pip install autograd
```

We use the following imports in this lecture

```
import matplotlib.pyplot as plt
from collections import namedtuple
from scipy.optimize import root
from autograd import jacobian
# Thinly-wrapped numpy to enable automatic differentiation
import autograd.numpy as np

plt.rcParams["figure.figsize"] = (10, 5.7)
```

9.2 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.

9.2.1 The Solow Model

In the Solow growth model, assuming Cobb-Douglas production technology and zero population growth, the law of motion for capital is

$$k_{t+1} = g(k_t) \quad \text{where} \quad g(k) := sAk^\alpha + (1-\delta)k \tag{9.1}$$

Here

- k_t is capital stock per worker,
- $A, \alpha > 0$ are production parameters, $\alpha < 1$
- $s > 0$ 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

$$k^* = \left(\frac{sA}{\delta}\right)^{1/(1-\alpha)}$$

9.2.2 Implementation

Let's store our parameters in `namedtuple` to help us keep our code clean and concise.

```
SolowParameters = namedtuple("SolowParameters", ('A', 's', 'α', 'δ'))
```

This function creates a suitable `namedtuple` 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 *(9.2.1)* 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 = np.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)
    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 arbitrary function f."
    k = k_0
    k_iterates = []
    for t in range(n):
```

(continues on next page)

(continued from previous page)

```
        k_iterates.append(k)
        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

$$\hat{g}(x) \approx g(x_0) + g'(x_0)(x - x_0) \tag{9.2}$$

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

$$x_{t+1} = \frac{g(x_t) - g'(x_t)x_t}{1 - g'(x_t)}, \quad x_0 \text{ given} \tag{9.3}$$

To implement Newton's method we observe that the derivative of the law of motion for capital *(9.2.1)* is

$$g'(k) = \alpha s A k^{\alpha-1} + (1 - \delta) \tag{9.4}$$

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 *(9.2.3)*.

```
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)
    ax2.plot(ks4, "-o", label="newton steps")
```

(continues on next page)

(continued from previous page)

```
    for ax in axes:
        ax.plot(k_star * np.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(np.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.

9.3 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.

9.3.1 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

$$x_{t+1} = x_t - \frac{f(x_t)}{f'(x_t)}, \quad x_0 \text{ given} \tag{9.5}$$

The following code implements the iteration *(9.3.1)*

```
def newton(f, Df, x_0, tol=1e-7, max_iter=100_000):
    x = x_0

    # Implement the zero-finding formula
    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 = np.abs(x - y)
        x = y
        print(f'iteration {n}, error = {error:.5f}')
    return x
```

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.)

9.3.2 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,
                              Df=lambda x: Dg(x, params) - 1,
                              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 the descent we saw in the graphs above: a very accurate result is reached with only 5 iterations.

9.4 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 Netwon's method.

9.4.1 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,

$$q_i^s(p) = b_i \sqrt{p_i}$$

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

$$A = \begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix}, \qquad b = \begin{pmatrix} b_0 \\ b_1 \end{pmatrix} \qquad \text{and} \qquad c = \begin{pmatrix} c_0 \\ c_1 \end{pmatrix}$$

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

$$e(p) = \begin{pmatrix} e_0(p) \\ e_1(p) \end{pmatrix}$$

The function below calculates the excess demand for given parameters

```
def e(p, A, b, c):
    return np.exp(- A @ p) + c - b * np.sqrt(p)
```

Our default parameter values will be

$$A = \begin{pmatrix} 0.5 & 0.4 \\ 0.8 & 0.2 \end{pmatrix}, \qquad b = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \qquad \text{and} \qquad c = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

```
A = np.array([
    [0.5, 0.4],
    [0.8, 0.2]
])
b = np.ones(2)
c = np.ones(2)
```

At a price level of $p = (1, 0.5)$, the excess demand is

```
ex_demand = e((1.0, 0.5), 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
```

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):

    # Create a 100x100 grid
    p_grid = np.linspace(0, grid_max, grid_size)
    z = np.empty((100, 100))

    for i, p_1 in enumerate(p_grid):
        for j, p_2 in enumerate(p_grid):
            z[i, j] = e((p_1, p_2), A, b, c)[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("$p_0$")
```

(continues on next page)

(continued from previous page)

```
    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 = np.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 271 µs, sys: 21 µs, total: 292 µs
Wall time: 265 µs
```

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 $e(p) \approx 0$:

```
np.max(np.abs(e(p, A, b, c)))
```

```
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

$$J(p) = \begin{pmatrix} \frac{\partial e_0}{\partial p_0}(p) & \frac{\partial e_0}{\partial p_1}(p) \\ \frac{\partial e_1}{\partial p_0}(p) & \frac{\partial e_1}{\partial p_1}(p) \end{pmatrix}$$

```
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 * np.exp(-a_00 * p_0) - (b[0]/2) * p_0**(-1/2)
    j_01 = -a_01 * np.exp(-a_01 * p_1)
    j_10 = -a_10 * np.exp(-a_10 * p_0)
    j_11 = -a_11 * np.exp(-a_11 * p_1) - (b[1]/2) * p_1**(-1/2)
    J = [[j_00, j_01],
         [j_10, j_11]]
    return np.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 448 µs, sys: 34 µs, total: 482 µs
Wall time: 396 µs
```

Now the solution is even more accurate (although, in this low-dimensional problem, the difference is quite small):

```
p = solution.x
np.max(np.abs(e(p, A, b, c)))
```

```
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

$$p_{n+1} = p_n - J_e(p_n)^{-1} e(p_n) \tag{9.6}$$

This is a multivariate version of *(9.3.1)*

(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 `autograd` library to auto-differentiate and calculate the Jacobian.

With only slight modification, we can generalize *our previous attempt* to multi-dimensional problems

```
def newton(f, x_0, tol=1e-5, max_iter=10):
    x = x_0
    q = lambda x: x - np.linalg.solve(jacobian(f)(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(np.isnan(y))):
            raise Exception('Solution not found with NaN generated')
        error = np.linalg.norm(x - y)
        x = y
        print(f'iteration {n}, error = {error:.5f}')
    print('\n' + f'Result = {x} \n')
    return x
```

```
def e(p, A, b, c):
    return np.exp(- np.dot(A, p)) + c - b * np.sqrt(p)
```

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 4.59 ms, sys: 33 µs, total: 4.62 ms
Wall time: 3.43 ms
```

```
np.max(np.abs(e(p, A, b, c)))
```

```
1.4632739464559563e-13
```

The result is very accurate.

With the larger overhead, the speed is not better than the optimized `scipy` function.

9.4.2 A High-Dimensional Problem

Our next step is to investigate a large market with 3,000 goods.

A JAX version of this section using GPU accelerated linear algebra and automatic differentiation is available here

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
np.random.seed(123)

# Create a random matrix A and normalize the rows to sum to one
A = np.random.rand(dim, dim)
A = np.asarray(A)
s = np.sum(A, axis=0)
A = A / s

# Set up b and c
b = np.ones(dim)
c = np.ones(dim)
```

Here's our initial condition

```
init_p = np.ones(dim)
```

```
%%time
p = newton(lambda p: e(p, A, b, c), init_p)
```

```
iteration 1, error = 23.22267
```

```
iteration 2, error = 3.94538
```

```
iteration 3, error = 0.08500
```

```
iteration 4, error = 0.00004
```

```
iteration 5, error = 0.00000

Result = [1.50185286 1.49865815 1.50028285 ... 1.50875149 1.48724784 1.48577532]

CPU times: user 2min 12s, sys: 1.59 s, total: 2min 13s
Wall time: 33.5 s
```

```
np.max(np.abs(e(p, A, b, c)))
```

```
6.661338147750939e-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: jacobian(e)(p, A, b, c),
                method='hybr',
                tol=1e-5)
```

```
CPU times: user 1min 21s, sys: 585 ms, total: 1min 22s
Wall time: 43 s
```

```
p = solution.x
np.max(np.abs(e(p, A, b, c)))
```

```
8.295585953721485e-07
```

9.5 Exercises

Exercise 9.5.1

Consider a three-dimensional extension of the Solow fixed point problem with

$$A = \begin{pmatrix} 2 & 3 & 3 \\ 2 & 4 & 2 \\ 1 & 5 & 1 \end{pmatrix}, \quad s = 0.2, \quad \alpha = 0.5, \quad \delta = 0.8$$

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:

$$\begin{aligned} k1_0 &= (1, 1, 1) \\ k2_0 &= (3, 5, 5) \\ k3_0 &= (50, 50, 50) \end{aligned}$$

Hint:

- The computation of the fixed point is equivalent to computing k^* such that $f(k^*) - k^* = 0$.
- If you are unsure about your solution, you can start with the solved example:

$$A = \begin{pmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{pmatrix}$$

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 to Exercise 9.5.1

Let's first define the parameters for this problem

```
A = np.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 = [np.ones(3),
          np.array([3.0, 5.0, 5.0]),
          np.repeat(50.0, 3)]
```

Then define the multivariate version of the formula for the *(9.2.1)*

```
def multivariate_solow(k, A=A, s=s, α=α, δ=δ):
    return (s * np.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 25.4 ms, sys: 0 ns, total: 25.4 ms
Wall time: 5.84 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]
```

(continues on next page)

(continued from previous page)

```
CPU times: user 16.7 ms, sys: 0 ns, total: 16.7 ms
Wall time: 3.82 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 16.2 ms, sys: 8.09 ms, total: 24.3 ms
Wall time: 5.55 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 substitute the output back to the formulate to check our last result

```
multivariate_solow(k) - k
```

```
array([ 0.0000000e+00, -4.4408921e-16,  8.8817842e-16])
```

Note the error is very small.

We can also test our results on the known solution

```
A = np.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 = np.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 17 ms, sys: 11 µs, total: 17 ms
Wall time: 3.9 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 21.9 ms, sys: 0 ns, total: 21.9 ms
Wall time: 5.02 ms
```

We can see it steps towards a more accurate solution.

Exercise 9.5.2

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:

$$A = \begin{pmatrix} 0.2 & 0.1 & 0.7 \\ 0.3 & 0.2 & 0.5 \\ 0.1 & 0.8 & 0.1 \end{pmatrix}, \qquad b = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \qquad \text{and} \qquad c = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$$

For this exercise, use the following extreme price vectors as initial values:

$$\begin{aligned} p1_0 &= (5, 5, 5) \\ p2_0 &= (1, 1, 1) \\ p3_0 &= (4.5, 0.1, 4) \end{aligned}$$

Set the tolerance to 0.0 for more accurate output.

Solution to Exercise 9.5.2

Define parameters and initial values

```
A = np.array([
    [0.2, 0.1, 0.7],
    [0.3, 0.2, 0.5],
    [0.1, 0.8, 0.1]
])

b = np.array([1.0, 1.0, 1.0])
c = np.array([1.0, 1.0, 1.0])

initLs = [np.repeat(5.0, 3),
          np.ones(3),
          np.array([4.5, 0.1, 4.0])]
```

Let's run through each initial guess and check the output

```
attempt = 1
for init in initLs:
    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
```

```
Attempt 1: Starting value is [5. 5. 5.]

iteration 1, error = 9.24381
```

```
/opt/conda/envs/quantecon/lib/python3.11/site-packages/autograd/tracer.py:48:␣
↪RuntimeWarning: invalid value encountered in sqrt
  return f_raw(*args, **kwargs)
/opt/conda/envs/quantecon/lib/python3.11/site-packages/autograd/numpy/numpy_vjps.
↪py:99: RuntimeWarning: invalid value encountered in power
  defvjp(anp.sqrt,    lambda ans, x : lambda g: g * 0.5 * x**-0.5)
```

```
---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
File <timed exec>:1

Cell In[34], line 12, in newton(f, x_0, tol, max_iter)
     10 y = q(x)
     11 if(any(np.isnan(y))):
---> 12     raise Exception('Solution not found with NaN generated')
     13 error = np.linalg.norm(x - y)
     14 x = y

Exception: Solution not found with NaN generated
```

```
----------------------------------------------------------------
Attempt 2: Starting value is [1. 1. 1.]

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]

CPU times: user 7.15 ms, sys: 0 ns, total: 7.15 ms
Wall time: 5.06 ms
----------------------------------------------------------------
Attempt 3: Starting value is [4.5 0.1 4. ]

iteration 1, error = 4.89202
iteration 2, error = 1.21206
iteration 3, error = 0.69421
```

(continues on next page)

(continued from previous page)

```
iteration 4, error = 0.16895
iteration 5, error = 0.00521
iteration 6, error = 0.00000
iteration 7, error = 0.00000
iteration 8, error = 0.00000

Result = [1.49744442 1.49744442 1.49744442]

CPU times: user 9.24 ms, sys: 59 µs, total: 9.3 ms
Wall time: 6.53 ms
----------------------------------------------------------------
```

We can find 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

```
e(p, A, b, c)
```

```
array([ 0.00000000e+00,  0.00000000e+00, -2.22044605e-16])
```

We can see the result is very accurate.

Part II

Elementary Statistics

CHAPTER

TEN

ELEMENTARY PROBABILITY WITH MATRICES

This lecture uses matrix algebra to illustrate some basic ideas about probability theory.

After providing somewhat informal definitions of the 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 probability distribution and a vector to represent a univariate 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')
```

10.1 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, but they are there underlying the other objects that we'll mainly focus on.

Let Ω be a set of possible underlying outcomes and let $\omega \in \Omega$ be a particular underlying outcomes.

Let $\mathcal{G} \subset \Omega$ be a subset of Ω.

Let $\mathcal{F}$ be a collection of such subsets $\mathcal{G} \subset \Omega$.

The pair $\Omega, \mathcal{F}$ forms our **probability space** on which we want to put a probability measure.

A **probability measure** μ maps a set of possible underlying outcomes $\mathcal{G} \in \mathcal{F}$ into a scalar number between 0 and 1

- this is the "probability" that X belongs to A, denoted by $\text{Prob}\{X \in A\}$.

A **random variable** $X(\omega)$ is a function of the underlying outcome $\omega \in \Omega$.

The random variable $X(\omega)$ has a **probability distribution** that is induced by the underlying probability measure μ and the function $X(\omega)$:

$$\text{Prob}(X \in A) = \int_{\mathcal{G}} \mu(\omega) d\omega \tag{10.1}$$

where $\mathcal{G}$ is the subset of Ω for which $X(\omega) \in A$.

We call this the induced probability distribution of random variable X.

10.2 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 the quantecon lectures https://python.quantecon.org/prob_meaning.html and https://python.quantecon.org/navy_captain.html.

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)**

Scalar example

Consider the following discrete distribution

$$X \sim \{f_i\}_{i=0}^{I-1}, \quad f_i \geqslant 0, \quad \sum_i f_i = 1$$

Draw a sample $x_0, x_1, \ldots, x_{N-1}$, N draws of X from $\{f_i\}_{i=1}^{I}$.

What do the "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 equal tthe product of marginal distributions, i.e.,

$$\begin{aligned}\text{Prob}\{x_0 = i_0, x_1 = i_1, \ldots, x_{N-1} = i_{N-1}\} &= \text{Prob}\{x_0 = i_0\} \cdot \cdots \cdot \text{Prob}\{x_{I-1} = i_{I-1}\} \\ &= f_{i_0} f_{i_1} \cdot \cdots \cdot f_{i_{N-1}}\end{aligned}$$

Consider the **empirical distribution**:

$$\begin{aligned} i &= 0, \ldots, I-1, \\ N_i &= \text{number of times } X = i, \\ N &= \sum_{i=0}^{I-1} N_i \quad \text{total number of draws}, \\ \tilde{f}_i &= \frac{N_i}{N} \sim \text{ frequency of draws for which } X = i \end{aligned}$$

Key ideas that justify connecting probability theory with statistics are laws of large numbers and central limit theorems

LLN:

- A Law of Large Numbers (LLN) states that $\tilde{f}_i \to f_i$ as $N \to \infty$

CLT:

- A Central Limit Theorem (CLT) describes a **rate** at which $\tilde{f}_i \to f_i$

Remarks

- For "frequentist" statisticians, **anticipated relative frequency** is **all** that a probability distribution means.
- But for a Bayesian it means something more or different.

10.3 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 **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 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
- we replace integration with summation in the formula like (10.1) that relates a CDF to a probability mass function

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.

10.4 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.

10.4.1 Discrete random variable

Let X be a discrete random variable that takes possible values: $i = 0, 1, \dots, I-1 = \bar{X}$.

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

$$f = \begin{bmatrix} f_0 \\ f_1 \\ \vdots \\ f_{I-1} \end{bmatrix} \tag{10.2}$$

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 (10.2) 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.

Remarks:

- The concept of **parameter** is intimately related to the notion of **sufficient statistic**.
- Sufficient statistics are nonlinear functions of a data set.
- Sufficient statistics are designed to summarize all **information** about parameters that is contained in a data set.
- They are important tools that AI uses to summarize a **big data** set
- R. A. Fisher provided a rigorous definition of **information** – see https://en.wikipedia.org/wiki/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

$$f_i(\theta) \geq 0, \sum_{i=0}^{\infty} f_i(\theta) = 1$$

10.4.2 Continuous random variable

Let X be a continous random variable that takes values $X \in \tilde{X} \equiv [X_U, X_L]$ whose distributions have 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

$$\text{Prob}\{X \in \tilde{X}\} = 1$$

10.5 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:

$$X \in \{0, \dots, I-1\}$$

$$Y \in \{0, \dots, J-1\}$$

Then their **joint distribution** is described by a matrix

$$F_{I \times J} = [f_{ij}]_{i \in \{0,\dots,I-1\}, j \in \{0,\dots,J-1\}}$$

whose elements are

$$f_{ij} = \text{Prob}\{X = i, Y = j\} \geq 0$$

where

$$\sum_i \sum_j f_{ij} = 1$$

10.6 Marginal Probability Distributions

The joint distribution induce marginal distributions

$$\text{Prob}\{X = i\} = \sum_{j=0}^{J-1} f_{ij} = \mu_i, \quad i = 0, \dots, I-1$$

$$\text{Prob}\{Y = j\} = \sum_{i=0}^{I-1} f_{ij} = \nu_j, \quad j = 0, \dots, J-1$$

For example, let a joint distribution over (X, Y) be

$$F = \begin{bmatrix} .25 & .1 \\ .15 & .5 \end{bmatrix} \tag{10.3}$$

The implied marginal distributions are:

$$\begin{aligned} \text{Prob}\{X = 0\} &= .25 + .1 = .35 \\ \text{Prob}\{X = 1\} &= .15 + .5 = .65 \\ \text{Prob}\{Y = 0\} &= .25 + .15 = .4 \\ \text{Prob}\{Y = 1\} &= .1 + .5 = .6 \end{aligned}$$

Digression: If two random variables X, Y are continuous and have joint density $f(x, y)$, then marginal distributions can be computed by

$$\begin{aligned} f(x) &= \int_{\mathbb{R}} f(x, y) dy \\ f(y) &= \int_{\mathbb{R}} f(x, y) dx \end{aligned}$$

10.7 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 $i = 0, \dots, I-1, \quad j = 0, \dots, J-1$.

Note that

$$\sum_i \text{Prob}\{X_i = i | Y_j = j\} = \frac{\sum_i f_{ij}}{\sum_i f_{ij}} = 1$$

Remark: The mathematics of conditional probability implies **Bayes' Law**:

$$\text{Prob}\{X = i | Y = j\} = \frac{\text{Prob}\{X = i, Y = j\}}{\text{Prob}\{Y = j\}} = \frac{\text{Prob}\{Y = j | X = i\}\text{Prob}\{X = i\}}{\text{Prob}\{Y = j\}}$$

For the joint distribution (10.3)

$$\text{Prob}\{X = 0 | Y = 1\} = \frac{.1}{.1 + .5} = \frac{.1}{.6}$$

10.8 Statistical Independence

Random variables X and Y are statistically **independent** if

$$\text{Prob}\{X = i, Y = j\} = f_i g_j$$

where

$$\text{Prob}\{X = i\} = f_i \geq 0 \ \sum f_i = 1$$
$$\text{Prob}\{Y = j\} = g_j \geq 0 \ \sum g_j = 1$$

Conditional distributions are

$$\text{Prob}\{X = i | Y = j\} = \frac{f_i g_j}{\sum_i f_i g_j} = \frac{f_i g_j}{g_i} = f_i$$
$$\text{Prob}\{Y = j | X = i\} = \frac{f_i g_j}{\sum_j f_i g_j} = \frac{f_i g_j}{f_i} = g_j$$

10.9 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

$$\mu_X \equiv \mathbb{E}[X] = \int_{-\infty}^{\infty} x f_X(x) dx$$
$$\sigma_X^2 \equiv \mathbb{D}[X] = \text{E}\left[(X - \mu_X)^2\right] = \int_{-\infty}^{\infty} (x - \mu_X)^2 f_X(x) dx$$

10.10 Generating Random Numbers

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 = np.random.rand(n)

# transform
x = -np.log(1-u)/λ

# draw geometric distributions
x_g = np.random.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 = np.random.rand(n)

# transform
x = np.ceil(np.log(1-u)/np.log(λ) - 1)

# draw geometric distributions
x_g = np.random.geometric(1-λ, n)

# plot and compare
plt.hist(x, bins=150, density=True)
plt.show()
```

```
np.random.geometric(1-λ, n).max()
```

```
70
```

```
np.log(0.4)/np.log(0.3)
```

```
0.7610560044063083
```

```
plt.hist(x_g, bins=150, density=True, alpha=0.6)
plt.show()
```

10.11 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

10.12 Geometric distribution

$$\text{Prob}(X = k) = (1-p)^{k-1}p, k = 1, 2, \ldots$$

$\Longrightarrow$

$$\mathbb{E}(X) = \frac{1}{p}$$
$$\mathbb{D}(X) = \frac{1-p}{p^2}$$

We draw observations from the distribution and compare the sample mean and variance with the theoretical results.

```
# specify parameters
p, n = 0.3, 1_000_000

# draw observations from the distribution
x = np.random.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:  3.329663
The sample variance is:  7.757961306431005

The population mean is:  3.3333333333333335
The population variance is:  7.777777777777778
```

10.12.1 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.array(range(1,10))

# mean
mean = np.sum(Benford_pmf * k)

# variance
var = np.sum([(k-mean)**2 * Benford_pmf])

# verify sum to 1
print(np.sum(Benford_pmf))
print(mean)
print(var)
```

```
0.9999999999999999
3.440236967123206
6.056512631375667
```

```
# plot distribution
plt.plot(range(1,10), Benford_pmf, 'o')
plt.title('Benford\'s distribution')
plt.show()
```

10.12.2 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 success.

Its distribution is

$$X \sim NB(r, p)$$
$$\text{Prob}(X = k; r, p) = \binom{k+r-1}{r-1} p^r (1-p)^k$$

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
```

(continues on next page)

(continued from previous page)

```
x = np.random.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:  23.326577
The sample variance is:  77.774148463071

The population mean is:  23.333333333333336
The population variance is:  77.77777777777779
```

10.13 Continuous Random Variables

10.13.1 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 = np.random.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:  -2.5696791128609393e-05
The sample standard deviation is:  0.10004898494410892
```

```
# compare
print(μ-μ_hat < 1e-3)
print(σ-σ_hat < 1e-3)
```

```
True
True
```

10.13.2 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)*np.random.rand(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:  14.998600037068522
The sample variance is:  8.334805948671056

The population mean is:  15.0
The population variance is:  8.333333333333334
```

10.14 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 = np.random.rand(1_000_000)
# x[x > 0.95] = 100*x[x > 0.95]+300
x[x > 0.95] = 100*np.random.rand(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:  17.555027820946638
The sample variance is:  5875.111605495466
```

The analytical mean and variance can be computed:

$$\begin{aligned}
\mu &= \int_{300}^{400} x f(x) dx \\
&= 0.0005 \int_{300}^{400} x dx \\
&= 0.0005 \times \frac{1}{2} x^2 \Big|_{300}^{400}
\end{aligned}$$

$$\begin{aligned}
\sigma^2 &= 0.95 \times (0 - 17.5)^2 + \int_{300}^{400} (x - 17.5)^2 f(x) dx \\
&= 0.95 \times 17.5^2 + 0.0005 \int_{300}^{400} (x - 17.5)^2 dx \\
&= 0.95 \times 17.5^2 + 0.0005 \times \frac{1}{3} (x - 17.5)^3 \Big|_{300}^{400}
\end{aligned}$$

```
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
```

10.15 Matrix Representation 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

$$\text{Prob}(X = i) = \sum_j f_{ij} = u_i$$

$$\text{Prob}(Y = j) = \sum_i f_{ij} = v_j$$

Below we draw some samples confirm that the "sampling" distribution agrees well with the "population" distribution.

Sample results:

```
# 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
p = np.random.rand(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)
```

```
[[ 0.  0.  0. ...  0.  1.  0.]
 [10. 10. 20. ... 10. 20. 10.]]
```

Here, we use exactly the inverse CDF technique to generate sample from the joint distribution F.

```
# marginal distribution
xp = np.sum(x[0, :] == xs[0])/1_000_000
yp = np.sum(x[1, :] == ys[0])/1_000_000
```

(continues on next page)

(continued from previous page)

```
# 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    | 0.499953 |
|    1    | 0.500047 |
+---------+----------+

marginal distribution for y
+---------+--------------------+
| y_value |       y_prob       |
+---------+--------------------+
|    10   |      0.400607      |
|    20   | 0.5993930000000001 |
+---------+--------------------+
```

```
# 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])
```

(continues on next page)

(continued from previous page)

```
yctb.add_row([xs[1], yc2p, 1-yc2p])
print(yctb)
```

```
conditional distribution for x
+---------+--------------------+--------------------+
| y_value |      prob(x=0)     |      prob(x=1)     |
+---------+--------------------+--------------------+
|    10   |  0.749602478239272 | 0.25039752176072805 |
|    20   | 0.33309865146906953 |  0.6669013485309305 |
+---------+--------------------+--------------------+

conditional distribution for y
+---------+--------------------+--------------------+
| x_value |      prob(y=10)    |      prob(y=20)    |
+---------+--------------------+--------------------+
|    0    |  0.6006484609553298 | 0.39935153904467025 |
|    1    | 0.20060314330452939 |  0.7993968566954706 |
+---------+--------------------+--------------------+
```

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 sample counterparts computed above.

Let's wrap some of the functions we have used in a Python class for a general 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
        '''
        xs = self.xs
        ys = self.ys
        f_cum = np.cumsum(self.f)
        p = np.random.rand(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))]
```

(continues on next page)

(continued from previous page)

```
        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]]
                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.

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    | 0.499926 |
|    1    | 0.500074 |
|   sum   |   1.0    |
+---------+----------+

marginal distribution for y
 +---------+----------+
| y_value |  y_prob  |
+---------+----------+
|    10   | 0.400563 |
|    20   | 0.599437 |
|   sum   |   1.0    |
+---------+----------+
```

```
# sample conditional
d.cond_dist()
```

```
conditional distribution for x
 +---------+---------------------+---------------------+-----+
| x_value |          0          |          1          | sum |
+---------+---------------------+---------------------+-----+
|    10   |  0.749073179499854  | 0.25092682050014603 | 1.0 |
|    20   | 0.33343787587352797 |  0.666562124126472  | 1.0 |
+---------+---------------------+---------------------+-----+

conditional distribution for y
 +---------+---------------------+--------------------+-----+
| y_value |          10         |         20         | sum |
+---------+---------------------+--------------------+-----+
|    0    |  0.6001908282425799 | 0.3998091717574201 | 1.0 |
|    1    | 0.20099425285057812 | 0.7990057471494219 | 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
 +---------+----------+
| x_value |  x_prob  |
+---------+----------+
|    10   | 0.300482 |
|    20   | 0.399327 |
|    30   | 0.300191 |
|   sum   |   1.0    |
+---------+----------+

marginal distribution for y
 +---------+----------+
| y_value |  y_prob  |
+---------+----------+
|    1    | 0.449434 |
|    2    | 0.550566 |
|   sum   |   1.0    |
+---------+----------+
```

```
d_new.cond_dist()
```

```
conditional distribution for x
 +---------+---------------------+---------------------+---------------------+-----
 ↪+
| x_value |          10         |          20         |          30         | sum |
+---------+---------------------+---------------------+---------------------+-----+
|    1    |  0.4452311129109057 | 0.22134729459720448 |  0.3334215924918898 | 1.0 |
|    2    | 0.18232146554636502 |  0.5446140880475728 | 0.27306444640606214 | 1.0 |
+---------+---------------------+---------------------+---------------------+-----+

conditional distribution for y
 +---------+--------------------+--------------------+-----+
| y_value |         1          |         2          | sum |
```

(continues on next page)

(continued from previous page)

```
+---------+--------------------+--------------------+-----+
|    10   | 0.6659367283231608 | 0.3340632716768392 | 1.0 |
|    20   | 0.2491216471713658 | 0.7508783528286341 | 1.0 |
|    30   | 0.4991855185531878 | 0.5008144814468122 | 1.0 |
+---------+--------------------+--------------------+-----+
```

10.16 A Continuous Bivariate Random Vector

A two-dimensional Gaussian distribution has joint density

$$f(x,y) = (2\pi\sigma_1\sigma_2\sqrt{1-\rho^2})^{-1}\exp\left[-\frac{1}{2(1-\rho^2)}\left(\frac{(x-\mu_1)^2}{\sigma_1^2} - \frac{2\rho(x-\mu_1)(y-\mu_2)}{\sigma_1\sigma_2} + \frac{(y-\mu_2)^2}{\sigma_2^2}\right)\right]$$

$$\frac{1}{2\pi\sigma_1\sigma_2\sqrt{1-\rho^2}}\exp\left[-\frac{1}{2(1-\rho^2)}\left(\frac{(x-\mu_1)^2}{\sigma_1^2} - \frac{2\rho(x-\mu_1)(y-\mu_2)}{\sigma_1\sigma_2} + \frac{(y-\mu_2)^2}{\sigma_2^2}\right)\right]$$

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
    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")
```

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 simulate from a built-in `numpy` function and 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 = np.random.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 = np.random.normal(μx_hat, σx_hat, 1_000_000)
plt.hist(x_sim, bins=1_000, alpha=0.4, histtype="step")
plt.show()
```

```
-0.0033915154669934187 2.2361944061298753
```

```
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 = np.random.normal(μy_hat, σy_hat, 1_000_000)
plt.hist(y_sim, bins=1_000, density=True, alpha=0.4, histtype="step")
plt.show()
```

```
4.99813369550081 0.9991098050285391
```

Conditional distribution

The population conditional distribution is

$$[X|Y=y] \sim \mathbb{N}\left[\mu_X + \rho\sigma_X \frac{y-\mu_Y}{\sigma_Y}, \sigma_X^2(1-\rho^2)\right]$$
$$[Y|X=x] \sim \mathbb{N}\left[\mu_Y + \rho\sigma_Y \frac{x-\mu_X}{\sigma_X}, \sigma_Y^2(1-\rho^2)\right]$$

Let's approximate the joint density by discretizing and mapping the approximating joint density into a matrix.

We can compute the discretized marginal density by just using matrix algebra and noting that

$$\text{Prob}\{X=i|Y=j\} = \frac{f_{ij}}{\sum_i f_{ij}}$$

Fix $y=0$.

```
# discretized marginal density
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 mean and variance are computed by

$$\mathbb{E}\left[X|Y=j\right] = \sum_i iProb\{X=i|Y=j\} = \sum_i i\frac{f_{ij}}{\sum_i f_{ij}}$$

$$\mathbb{D}\left[X|Y=j\right] = \sum_i \left(i-\mu_{X|Y=j}\right)^2 \frac{f_{ij}}{\sum_i f_{ij}}$$

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 = np.random.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(0, 10, 1_000_000)
z = func(x=1, y=y) / np.sum(func(x=1, y=y))
plt.plot(y,z)
plt.show()
```

```
# discretized mean and standard deviation
μy = np.dot(y,z)
σy = np.sqrt(np.dot((y - μy)**2, z))

# sample
zz = np.random.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)))
```

```
-0.9997518414498433 2.22658413316977
-1.0 2.227105745132009
5.039999456960768 0.9959851265795597
5.04 0.9959919678390986
```

10.17 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:

$$
\begin{aligned}
\bar{X} &= \{0, 1, \ldots, I-1\}; \qquad & f_i &= \text{Prob}\{X = i\} \\
\bar{Y} &= \{0, 1, \ldots, J-1\}; \qquad & g_j &= \text{Prob}\{Y = j\} \\
\bar{Z} &= \{0, 1, \ldots, I+J-2\}; \qquad & h_k &= \text{Prob}\{X + Y = k\}
\end{aligned}
$$

Independence of X and Y implies that

$$\begin{aligned} h_k &= \text{Prob}\{X = 0, Y = k\} + \text{Prob}\{X = 1, Y = k-1\} + ... + \text{Prob}\{X = k, Y = 0\} \\ h_k &= f_0 g_k + f_1 g_{k-1} + ... + f_{k-1} g_1 + f_k g_0 \qquad \text{for} \quad k = 0, 1, ..., I + J - 2 \end{aligned}$$

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.

10.18 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 $i = 0, ..., I - 1; j = 0, ..., J - 1$ and

$$\sum_i \sum_j \rho_{ij} = 1, \quad \rho_{ij} \geqslant 0.$$

An associated conditional distribution is

$$\text{Prob}\{Y = i | X = j\} = \frac{\rho_{ij}}{\sum_i \rho_{ij}} = \frac{\text{Prob}\{Y = j, X = i\}}{\text{Prob}\{X = i\}}$$

We can define a transition probability matrix

$$p_{ij} = \text{Prob}\{Y = j | X = i\} = \frac{\rho_{ij}}{\sum_j \rho_{ij}}$$

where

$$\begin{bmatrix} p_{11} & p_{12} \\ p_{21} & p_{22} \end{bmatrix}$$

The first row is the probability of $Y = j, j = 0, 1$ conditional on $X = 0$.

The second row is the probability of $Y = j, j = 0, 1$ conditional on $X = 1$.

Note that

- $\sum_j \rho_{ij} = \frac{\sum_j \rho_{ij}}{\sum_j \rho_{ij}} = 1$, so each row of ρ is a probability distribution (not so for each column.

10.19 Coupling

Start with a joint distribution

$$
\begin{aligned}
f_{ij} &= \text{Prob}\{X = i, Y = j\} \\
i &= 0, \cdots \ I-1 \\
j &= 0, \cdots \ J-1 \\
&\text{stacked to an } I \times J \text{ matrix} \\
&e.g. \quad I = 1, J = 1
\end{aligned}
$$

where

$$
\begin{bmatrix} f_{11} & f_{12} \\ f_{21} & f_{22} \end{bmatrix}
$$

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, can we usually construct more than one joint distribution that verifies these marginals.

Each of these joint distributions is called a **coupling** of the two martingal distributions.

Let's start with marginal distributions

$$
\begin{aligned}
\text{Prob}\{X = i\} &= \sum_j f_{ij} = \mu_i, i = 0, \cdots, I-1 \\
\text{Prob}\{Y = j\} &= \sum_j f_{ij} = \nu_j, j = 0, \cdots, J-1
\end{aligned}
$$

Given two marginal distribution, μ for X and ν for Y, a joint distribution f_{ij} is said to be a **coupling** of μ and ν.

Example:

Consider the following bivariate example.

$$
\begin{aligned}
\text{Prob}\{X = 0\} &= 1 - q = \mu_0 \\
\text{Prob}\{X = 1\} &= q = \mu_1 \\
\text{Prob}\{Y = 0\} &= 1 - r = \nu_0 \\
\text{Prob}\{Y = 1\} &= r = \nu_1 \\
\text{where } 0 \leq q &< r \leq 1
\end{aligned}
$$

We construct two couplings.

The first coupling if 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

$$
\begin{aligned}
(1-q)(1-r) + (1-q)r + q(1-r) + qr &= 1 \\
\mu_0 = (1-q)(1-r) + (1-q)r &= 1 - q \\
\mu_1 = q(1-r) + qr &= q \\
\nu_0 = (1-q)(1-r) + (1-r)q &= 1 - r \\
\mu_1 = r(1-q) + qr &= r
\end{aligned}
$$

A second coupling of our two marginal distributions is the joint distribution

$$f_{ij} = \begin{bmatrix} (1-r) & r-q \\ 0 & q \end{bmatrix}$$

The 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.

Remark:

- Couplings are important in optimal transport problems and in Markov processes.

10.20 Copula Functions

Suppose that $X_1, X_2, \dots, X_n$ are N random variables and that

- their marginal distributions are $F_1(x_1), F_2(x_2), \dots, F_N(x_N)$, and
- their joint distribution is $H(x_1, x_2, \dots, x_N)$

Then there exists a **copula function** $C(\cdot)$ that verifies

$$H(x_1, x_2, \dots, x_N) = C(F_1(x_1), F_2(x_2), \dots, F_N(x_N)).$$

We can obtain

$$C(u_1, u_2, \dots, u_n) = H[F_1^{-1}(u_1), F_2^{-1}(u_2), \dots, F_N^{-1}(u_N)]$$

In a reverse direction of logic, given univariate **marginal distributions** $F_1(x_1), F_2(x_2), \dots, F_N(x_N)$ and a copula function $C(\cdot)$, the function $H(x_1, x_2, \dots, x_N) = C(F_1(x_1), F_2(x_2), \dots, F_N(x_N))$ is a **coupling** of $F_1(x_1), F_2(x_2), \dots, F_N(x_N)$.

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.

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.

```
# define parameters
mu = np.array([0.6, 0.4])
nu = np.array([0.3, 0.7])

# number of draws
draws = 1_000_000

# generate draws from uniform distribution
p = np.random.rand(draws)

# generate draws of X and Y via uniform distribution
x = np.ones(draws)
y = np.ones(draws)
x[p <= mu[0]] = 0
x[p > mu[0]] = 1
y[p <= nu[0]] = 0
y[p > nu[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)

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    | 0.600646 |
|    1    | 0.399354 |
+---------+----------+
distribution for y
+---------+----------+
| y_value |  y_prob  |
+---------+----------+
|    0    | 0.300605 |
|    1    | 0.699395 |
+---------+----------+
```

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 = np.random.rand(draws1)

# generate draws of first copuling 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
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      | 0.179779 |
|     0      |     1      | 0.41976  |
|     1      |     0      | 0.120015 |
|     1      |     1      | 0.280446 |
+------------+------------+----------+
```

```
# 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      |  0.599539 |
|     1      |  0.400461 |
+------------+-----------+
marginal distribution for y
+------------+-----------+
| c1_y_value | c1_y_prob |
+------------+-----------+
|     0      |  0.299794 |
|     1      |  0.700206 |
+------------+-----------+
```

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 = np.random.rand(draws2)

# generate draws of first 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
```

(continues on next page)

(continued from previous page)

```
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("first 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)
```

```
first joint distribution for c2
+------------+------------+----------+
| c2_x_value | c2_y_value | c2_prob  |
+------------+------------+----------+
|     0      |     0      | 0.300361 |
|     0      |     1      | 0.299724 |
|     1      |     0      |   0.0    |
|     1      |     1      | 0.399915 |
+------------+------------+----------+
```

```
# 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      |  0.600085 |
|     1      |  0.399915 |
+------------+-----------+
marginal distribution for y
+------------+-----------+
| c2_y_value | c2_y_prob |
+------------+-----------+
|     0      |  0.300361 |
|     1      |  0.699639 |
+------------+-----------+
```

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.

10.21 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

$$\text{Prob}\{X(0) = i, X(1) = j\} = f_{ij} \geq 0 \; i = 0, \cdots, I-1$$
$$\sum_i \sum_j f_{ij} = 1$$

f_{ij} is a joint distribution over $[X(0), X(1)]$.

A conditional distribution is

$$\text{Prob}\{X(1) = j | X(0) = i\} = \frac{f_{ij}}{\sum_j f_{ij}}$$

Remark:

- This is a key formula for a theory of optimally predicting a time series.

CHAPTER

ELEVEN

LLN AND CLT

Contents

- *LLN and CLT*
  - *Overview*
  - *Relationships*
  - *LLN*
  - *CLT*
  - *Exercises*

11.1 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
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
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
```

11.2 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.

11.3 LLN

We begin with the law of large numbers, which tells us when sample averages will converge to their population means.

11.3.1 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

$$\mathbb{P}\left\{\bar{X}_n \to \mu \text{ as } n \to \infty\right\} = 1 \tag{11.1}$$

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, (11.1) 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.

11.3.2 Proof

The proof of Kolmogorov's strong law is nontrivial – see, for example, theorem 8.3.5 of [Dud02].

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

$$\mathbb{P}\left\{|\bar{X}_n - \mu| \geq \epsilon\right\} \to 0 \quad \text{as} \quad n \to \infty \tag{11.2}$$

(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

$$\mathbb{P}\left\{|\bar{X}_n - \mu| \geq \epsilon\right\} \leq \frac{\mathbb{E}[(\bar{X}_n - \mu)^2]}{\epsilon^2} \tag{11.3}$$

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 (11.3), we come to the estimate

$$\mathbb{P}\left\{|\bar{X}_n - \mu| \geq \epsilon\right\} \leq \frac{\sigma^2}{n\epsilon^2} \tag{11.4}$$

The claim in (11.2) 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.

11.3.3 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 $i = 1, \dots, 100$.

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 = '$\\bar{X}_n$ 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='$\mu$')
    ax.vlines(list(range(n)), m, data, lw=0.2)
    ax.legend(**legend_args, fontsize=12)

plt.show()
```

$\bar{X}_n$ for $X_i \sim \beta(2, 2)$
μ
0.8
0.6
0.4
0.2
0
20
40
60
80
100

$\bar{X}_n$ for $X_i \sim \gamma(5, 1/2)$
μ
22.5
20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0
20
40
60
80
100

$\bar{X}_n$ for $X_i \sim$ student's t with 10 degrees of freedom
μ
3
2
1
0
−1
−2
−3
0
20
40
60
80
100

The three distributions are chosen at random from a selection stored in the dictionary `distributions`.

11.4 CLT

Next, we turn to the central limit theorem, which tells us about the distribution of the deviation between sample averages and population means.

11.4.1 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 $X_1, \dots, X_n$ is IID, with common mean μ and common variance $\sigma^2 \in (0, \infty)$, then

$$\sqrt{n}(\bar{X}_n - \mu) \xrightarrow{d} N(0, \sigma^2) \quad \text{as} \quad n \to \infty \tag{11.5}$$

Here $\xrightarrow{d} N(0, \sigma^2)$ indicates convergence in distribution to a centered (i.e, zero mean) normal with standard deviation σ.

11.4.2 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 [Dud02]).

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 $X_1, \dots, X_n$ 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.

11.4.3 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='$N(0, \sigma^2)$')
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.

11.4.4 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):
    """
    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 = np.random.randint(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))

# 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)
# 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.

11.4.5 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 $(X_1, \dots, X_k)$.

Each realization of $\mathbf{X}$ is an element of $\mathbb{R}^k$.

A collection of random vectors $\mathbf{X}_1, \dots, \mathbf{X}_n$ is called independent if, given any n vectors $\mathbf{x}_1, \dots, \mathbf{x}_n$ in $\mathbb{R}^k$, we have

$$\mathbb{P}\{\mathbf{X}_1 \leq \mathbf{x}_1, \dots, \mathbf{X}_n \leq \mathbf{x}_n\} = \mathbb{P}\{\mathbf{X}_1 \leq \mathbf{x}_1\} \times \cdots \times \mathbb{P}\{\mathbf{X}_n \leq \mathbf{x}_n\}$$

(The vector inequality $\mathbf{X} \leq \mathbf{x}$ means that $X_j \leq x_j$ for $j = 1, \dots, k$)

Let $\mu_j := \mathbb{E}[X_j]$ for all $j = 1, \dots, k$.

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 $\mathbf{X}_1, \dots, \mathbf{X}_n$ 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

$$\mathbb{P}\left\{\bar{\mathbf{X}}_n \to \mu \text{ as } n \to \infty\right\} = 1 \tag{11.6}$$

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,

$$\sqrt{n}(\bar{\mathbf{X}}_n - \mu) \xrightarrow{d} N(\mathbf{0}, \Sigma) \quad \text{as} \quad n \to \infty \tag{11.7}$$

11.5 Exercises

Exercise 11.5.1

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

$$\sqrt{n}\{g(\bar{X}_n) - g(\mu)\} \xrightarrow{d} N(0, g'(\mu)^2 \sigma^2) \quad \text{as} \quad n \to \infty \tag{11.8}$$

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 to Exercise 11.5.1

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 = "$N(0, g'(\mu)^2  \sigma^2)$"
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 11.5.2

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. $\mathbf{X}_1, \dots, \mathbf{X}_n$ 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

$$\sqrt{n}(\bar{\mathbf{X}}_n - \mu) \xrightarrow{d} N(\mathbf{0}, \Sigma) \tag{11.9}$$

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

$$\mathbf{AZ}_n \xrightarrow{d} \mathbf{AZ}$$

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

$$n\|\mathbf{Q}(\bar{\mathbf{X}}_n - \mu)\|^2 \xrightarrow{d} \chi^2(k) \tag{11.10}$$

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 (11.10) 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 to Exercise 11.5.2

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

$$\mathbf{Q}\mathbf{Y}_n \xrightarrow{d} \mathbf{Q}\mathbf{Y}$$

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

TWELVE

TWO MEANINGS OF PROBABILITY

12.1 Overview

This lecture illustrates two distinct interpretations of a **probability distribution**

- A frequentist interpretation as **relative frequencies** anticipated to occur in a large i.i.d. sample
- A Bayesian interpretation as a **personal opinion** (about a parameter or list of parameters) after seeing a collection of observations

We recommend watching this video about **hypothesis testing** within the frequentist approach

https://youtu.be/8JIe_cz6qGA

After you watch that video, please watch the following video on the Bayesian approach to constructing **coverage intervals**

https://youtu.be/Pahyv9i_X2k

After you are familiar with the material in these videos, this lecture uses the Socratic method to to help consolidate your understanding of the different questions that are answered by

- a frequentist confidence interval
- a Bayesian coverage interval

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.

Code for answering questions:

In addition to what's in Anaconda, this lecture will deploy the following library:

```
pip install prettytable
```

To answer our coding questions, we'll start with some imports

```
import numpy as np
import pandas as pd
import prettytable as pt
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.

12.2 Frequentist Interpretation

Consider the following classic example.

The random variable X takes on possible values $k = 0, 1, 2, \ldots, n$ with probabilties

$$\text{Prob}(X = k|\theta) = \left(\frac{n!}{k!(n-k)!}\right)\theta^k(1-\theta)^{n-k}$$

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 $\sum_{h=1}^n y_h^i = k$:

$$f_k^I = \frac{\text{number of samples of length n for which } \sum_{h=1}^n y_h^i = k}{I}$$

The probability $\text{Prob}(X = k|\theta)$ 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 12.2.1

1. Please write a Python class to compute f_k^I
2. Please use your code to compute $f_k^I, k = 0, \ldots, n$ and compare them to $\text{Prob}(X = k|\theta)$ for various values of θ, n and I
3. With the Law of Large numbers in mind, use your code to say something

---

Solution to Exercise 12.2.1

Here is one solution:

```
class frequentist:

    def __init__(self, θ, n, I):

        '''
        initialization
        -----------------
        parameters:
        9 : probability that one toss of a coin will be a head with Y = 1
        n : number of independent flips in each independent sequence of draws
        I : number of independent sequence of draws

        '''

        self.θ, self.n, self.I = θ, n, I

    def binomial(self, k):

        '''compute the theoretical probability for specific input k'''

        θ, n = self.θ, self.n
        self.k = k
        self.P = binom.pmf(k, n, θ)

    def draw(self):

        '''draw n independent flips for I independent sequences'''

        θ, n, I = self.θ, self.n, self.I
        sample = np.random.rand(I, n)
        Y = (sample <= θ) * 1
        self.Y = Y

    def compute_fk(self, kk):

        '''compute f_{k}^I for specific input k'''

        Y, I = self.Y, self.I
        K = np.sum(Y, 1)
        f_kI = np.sum(K == kk) / I
        self.f_kI = f_kI
        self.kk = kk

    def compare(self):

        '''compute and print the comparison'''

        n = self.n
        comp = pt.PrettyTable()
        comp.field_names = ['k', 'Theoretical', 'Frequentist']
        self.draw()
        for i in range(n):
            self.binomial(i+1)
            self.compute_fk(i+1)
            comp.add_row([i+1, self.P, self.f_kI])
        print(comp)
```

```
θ, n, k, I = 0.7, 20, 10, 1_000_000

freq = frequentist(θ, n, I)

freq.compare()
```

```
+----+------------------------+-------------+
| k  |      Theoretical       | Frequentist |
+----+------------------------+-------------+
| 1  | 1.6271660538000033e-09 |     0.0     |
| 2  | 3.606884752589999e-08  |    1e-06    |
| 3  |  5.04963865362601e-07  |     0.0     |
| 4  | 5.007558331512455e-06  |    8e-06    |
| 5  | 3.7389768875293014e-05 |   4.7e-05   |
| 6  | 0.00021810698510587546 |   0.000215  |
| 7  |  0.001017832597160754  |   0.001005  |
| 8  |  0.003859281930901185  |   0.003811  |
| 9  |  0.012006654896137007  |   0.012057  |
| 10 |  0.030817080900085007  |   0.031069  |
| 11 |  0.06536956554563476   |   0.065284  |
| 12 |  0.11439673970486108   |   0.113936  |
| 13 |   0.1642619852172365   |   0.164163  |
| 14 |  0.19163898275344252   |   0.191442  |
| 15 |  0.17886305056987967   |   0.179241  |
| 16 |   0.1304209743738704   |   0.130724  |
| 17 |  0.07160367220526209   |   0.071519  |
| 18 |  0.027845872524268643  |   0.027851  |
| 19 |  0.006839337111223895  |   0.006827  |
| 20 | 0.0007979226629761189  |    0.0008   |
+----+------------------------+-------------+
```

From the table above, can you see the law of large numbers at work?

Let's do some more calculations.

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, npt = 0.01, 0.99, 50
thetas = np.linspace(θ_low, θ_high, npt)
P = []
f_kI = []
for i in range(npt):
    freq = frequentist(thetas[i], n, I)
    freq.binomial(k)
    freq.draw()
    freq.compute_fk(k)
    P.append(freq.P)
    f_kI.append(freq.f_kI)
```

```
fig, ax = plt.subplots(figsize=(8, 6))
ax.grid()
ax.plot(thetas, P, 'k-.', label='Theoretical')
ax.plot(thetas, f_kI, 'r--', label='Fraction')
plt.title(r'Comparison with different $\theta$', fontsize=16)
plt.xlabel(r'$\theta$', fontsize=15)
plt.ylabel('Fraction', fontsize=15)
plt.tick_params(labelsize=13)
plt.legend()
plt.show()
```

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, nn = 1, 100, 50
ns = np.linspace(n_low, n_high, nn, dtype='int')
P = []
f_kI = []
for i in range(nn):
    freq = frequentist(θ, ns[i], I)
    freq.binomial(k)
    freq.draw()
```

(continues on next page)

(continued from previous page)

```
    freq.compute_fk(k)
    P.append(freq.P)
    f_kI.append(freq.f_kI)
```

```
fig, ax = plt.subplots(figsize=(8, 6))
ax.grid()
ax.plot(ns, P, 'k-.', label='Theoretical')
ax.plot(ns, f_kI, 'r--', label='Frequentist')
plt.title(r'Comparison with different $n$', fontsize=16)
plt.xlabel(r'$n$', fontsize=15)
plt.ylabel('Fraction', fontsize=15)
plt.tick_params(labelsize=13)
plt.legend()
plt.show()
```

Comparison with different I

Now we fix $\theta = 0.7, n = 20, k = 10$ and vary $\log(I)$ from 2 to 7.

```
I_log_low, I_log_high, nI = 2, 6, 200
log_Is = np.linspace(I_log_low, I_log_high, nI)
Is = np.power(10, log_Is).astype(int)
P = []
```

(continues on next page)

(continued from previous page)

```
f_kI = []
for i in range(nI):
    freq = frequentist(θ, n, Is[i])
    freq.binomial(k)
    freq.draw()
    freq.compute_fk(k)
    P.append(freq.P)
    f_kI.append(freq.f_kI)
```

```
fig, ax = plt.subplots(figsize=(8, 6))
ax.grid()
ax.plot(Is, P, 'k-.', label='Theoretical')
ax.plot(Is, f_kI, 'r--', label='Fraction')
plt.title(r'Comparison with different $I$', fontsize=16)
plt.xlabel(r'$I$', fontsize=15)
plt.ylabel('Fraction', fontsize=15)
plt.tick_params(labelsize=13)
plt.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 θ.

The Law of Large Numbers is at work here.

For each draw of an independent sequence, Prob$(X_i = k|\theta)$ is the same, so aggregating all draws forms an i.i.d sequence of a binary random variable $\rho_{k,i}, i = 1, 2, ...I$, with a mean of Prob$(X = k|\theta)$ and a variance of

$$n \cdot \text{Prob}(X = k|\theta) \cdot (1 - \text{Prob}(X = k|\theta)).$$

So, by the LLN, the average of $P_{k,i}$ converges to:

$$E[\rho_{k,i}] = \text{Prob}(X = k|\theta) = \left(\frac{n!}{k!(n-k)!}\right)\theta^k(1-\theta)^{n-k}$$

as I goes to infinity.

12.3 Bayesian Interpretation

We again use a binomial distribution.

But now we don't regard θ as being a fixed number.

Instead, we think of it as a **random variable**.

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 i.i.d. 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

Thus, suppose that, before seeing any data, you have a personal prior probability distribution saying that

$$P(\theta) = \frac{\theta^{\alpha-1}(1-\theta)^{\beta-1}}{B(\alpha,\beta)}$$

where $B(\alpha, \beta)$ is a **beta function** , so that $P(\theta)$ is a **beta distribution** with parameters α, β.

Exercise 12.3.1

a) Please write down the **likelihood function** for a sample of length n from a binomial distribution with parameter θ.

b) Please write down the **posterior** distribution for θ after observing one flip of the coin.

c) Now pretend that the true value of $\theta = .4$ and that someone who doesn't know this has a beta prior distribution with parameters with $\beta = \alpha = .5$. Please write a Python class to simulate this person's personal posterior distribution for θ for a *single* sequence of n draws.

d) Please plot the posterior distribution for θ as a function of θ as n grows as $1, 2, ...$

e) For various n's, please describe and compute a Bayesian coverage interval for the interval $[.45, .55]$.

f) Please tell what question a Bayesian coverage interval answers.

g) Please compute the Posterior probabililty that $\theta \in [.45, .55]$ for various values of sample size n.

h) Please use your Python class to study what happens to the posterior distribution as $n \to +\infty$, again assuming that the true value of $\theta = .4$, though it is unknown to the person doing the updating via Bayes' Law.

Solution to Exercise 12.3.1

a) Please write down the **likelihood function** and the **posterior** distribution for θ after observing one flip of our coin.

Suppose the outcome is **Y**.

The likelihood function is:

$$L(Y|\theta) = \text{Prob}(X = Y|\theta) = \theta^Y (1-\theta)^{1-Y}$$

b) Please write the **posterior** distribution for θ after observing one flip of our coin.

The prior distribution is

$$\text{Prob}(\theta) = \frac{\theta^{\alpha-1}(1-\theta)^{\beta-1}}{B(\alpha,\beta)}$$

We can derive the posterior distribution for θ via

$$\begin{aligned}
\text{Prob}(\theta|Y) &= \frac{\text{Prob}(Y|\theta)\text{Prob}(\theta)}{\text{Prob}(Y)} \\
&= \frac{\text{Prob}(Y|\theta)\text{Prob}(\theta)}{\int_0^1 \text{Prob}(Y|\theta)\text{Prob}(\theta)d\theta} \\
&= \frac{\theta^Y(1-\theta)^{1-Y}\frac{\theta^{\alpha-1}(1-\theta)^{\beta-1}}{B(\alpha,\beta)}}{\int_0^1 \theta^Y(1-\theta)^{1-Y}\frac{\theta^{\alpha-1}(1-\theta)^{\beta-1}}{B(\alpha,\beta)}d\theta} \\
&= \frac{\theta^{Y+\alpha-1}(1-\theta)^{1-Y+\beta-1}}{\int_0^1 \theta^{Y+\alpha-1}(1-\theta)^{1-Y+\beta-1}d\theta}
\end{aligned}$$

which means that

$$\text{Prob}(\theta|Y) \sim \text{Beta}(\alpha + Y, \beta + (1-Y))$$

Now please pretend that the true value of $\theta = .4$ and that someone who doesn't know this has a beta prior with $\beta = \alpha = .5$.

c) Now pretend that the true value of $\theta = .4$ and that someone who doesn't know this has a beta prior distribution with parameters with $\beta = \alpha = .5$. Please write a Python class to simulate this person's personal posterior distribution for θ for a *single* sequence of n draws.

```
class Bayesian:

    def __init__(self, θ=0.4, n=1_000_000, α=0.5, β=0.5):
        """
        Parameters:
        ----------
        θ : float, ranging from [0,1].
           probability that one toss of a coin will be a head with Y = 1

        n : int.
           number of independent flips in an independent sequence of draws

        α&β : int or float.
```

(continues on next page)

(continued from previous page)

```
            parameters of the prior distribution on θ

        """
        self.θ, self.n, self.α, self.β = θ, n, α, β
        self.prior = st.beta(α, β)

    def draw(self):
        """
        simulate a single sequence of draws of length n, given probability θ

        """
        array = np.random.rand(self.n)
        self.draws = (array < self.θ).astype(int)

    def form_single_posterior(self, step_num):
        """
        form a posterior distribution after observing the first step_num elements of␣
↪the draws

        Parameters
        ----------
        step_num: int.
               number of steps observed to form a posterior distribution

        Returns
        ------
        the posterior distribution for sake of plotting in the subsequent steps

        """
        heads_num = self.draws[:step_num].sum()
        tails_num = step_num - heads_num

        return st.beta(self.α+heads_num, self.β+tails_num)

    def form_posterior_series(self,num_obs_list):
        """
        form a series of posterior distributions that form after observing different␣
↪number of draws.

        Parameters
        ----------
        num_obs_list: a list of int.
               a list of the number of observations used to form a series of␣
↪posterior distributions.

        """
        self.posterior_list = []
        for num in num_obs_list:
            self.posterior_list.append(self.form_single_posterior(num))
```

d) Please plot the posterior distribution for θ as a function of θ as n grows from $1, 2, \ldots$.

```
Bay_stat = Bayesian()
Bay_stat.draw()

num_list = [1, 2, 3, 4, 5, 10, 20, 30, 50, 70, 100, 300, 500, 1000, # this line for␣
↪finite n
```

(continues on next page)

(continued from previous page)

```
            5000, 10_000, 50_000, 100_000, 200_000, 300_000]  # this line for␣
↪approximately infinite n

Bay_stat.form_posterior_series(num_list)

θ_values = np.linspace(0.01, 1, 100)

fig, ax = plt.subplots(figsize=(10, 6))

ax.plot(θ_values, Bay_stat.prior.pdf(θ_values), label='Prior Distribution', color='k',
↪ linestyle='--')

for ii, num in enumerate(num_list[:14]):
    ax.plot(θ_values, Bay_stat.posterior_list[ii].pdf(θ_values), label='Posterior␣
↪with n = %d' % num)

ax.set_title('P.D.F of Posterior Distributions', fontsize=15)
ax.set_xlabel(r"$\theta$", fontsize=15)

ax.legend(fontsize=11)
plt.show()
```

e) For various n's, please describe and compute .05 and .95 quantiles for posterior probabilities.

```
upper_bound = [ii.ppf(0.05) for ii in Bay_stat.posterior_list[:14]]
lower_bound = [ii.ppf(0.95) for ii in Bay_stat.posterior_list[:14]]

interval_df = pd.DataFrame()
```

(continues on next page)

(continued from previous page)

```
interval_df['upper'] = upper_bound
interval_df['lower'] = lower_bound
interval_df.index = num_list[:14]
interval_df = interval_df.T
interval_df
```

```
            1         2         3         4         5         10        20    \
upper  0.228520  0.097308  0.235534  0.349293  0.260634  0.185116  0.324234
lower  0.998457  0.902692  0.937587  0.953993  0.872224  0.652678  0.675766

            30        50        70        100       300       500       1000
upper  0.293487  0.347833  0.321256  0.369972  0.396703  0.393947  0.378704
lower  0.582293  0.575503  0.512457  0.532139  0.490768  0.466650  0.429705
```

As n increases, we can see that Bayesian coverage intervals narrow and move toward 0.4.

f) Please tell what question a Bayesian coverage interval answers.

The Bayesian coverage interval tells the range of θ that corresponds to the $[p_1, p_2]$ quantiles of the cumulative probability distribution (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 $[p_1, p_2]$ is described by

$$F(a) = p_1, F(b) = p_2$$

g) Please compute the Posterior probabililty that $\theta \in [.45, .55]$ for various values of sample size n.

```
left_value, right_value = 0.45, 0.55

posterior_prob_list=[ii.cdf(right_value)-ii.cdf(left_value) for ii in Bay_stat.
 ↪posterior_list]

fig, ax = plt.subplots(figsize=(8, 5))
ax.plot(posterior_prob_list)
ax.set_title('Posterior Probabililty that '+ r"$\theta$" +' Ranges from %.2f to %.2f'
 ↪%(left_value, right_value),
             fontsize=13)
ax.set_xticks(np.arange(0, len(posterior_prob_list), 3))
ax.set_xticklabels(num_list[::3])
ax.set_xlabel('Number of Observations', fontsize=11)

plt.show()
```

Notice that in the graph above the posterior probabililty that $\theta \in [.45, .55]$ typically 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 probabililty.

However, $[.45, .55]$ actually excludes the true $\theta = .4$ that generates the data.

As a result, the posterior probabililty 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 $\theta \in [.45, .55]$ very unlikely.

That is why we see a nearly horizontal line when the number of observations exceeds 500.

h) Please use your Python class to study what happens to the posterior distribution as $n \to +\infty$, again assuming that the true value of $\theta = .4$, though it is unknown to the person doing the updating via Bayes' Law.

Using the Python class we made above, we can see the evolution of posterior distributions as n approaches infinity.

```
fig, ax = plt.subplots(figsize=(10, 6))

for ii, num in enumerate(num_list[14:]):
    ii += 14
    ax.plot(θ_values, Bay_stat.posterior_list[ii].pdf(θ_values),
            label='Posterior with n=%d thousand' % (num/1000))
```

(continues on next page)

(continued from previous page)

```
ax.set_title('P.D.F 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.

Here the posterior means converges to 0.4 while the posterior standard deviations converges to 0 from above.

To show this, we compute the means and variances statistics of the posterior distributions.

```
mean_list = [ii.mean() for ii in Bay_stat.posterior_list]
std_list = [ii.std() for ii in Bay_stat.posterior_list]

fig, ax = plt.subplots(1, 2, figsize=(14, 5))

ax[0].plot(mean_list)
ax[0].set_title('Mean Values of Posterior Distribution', fontsize=13)
ax[0].set_xticks(np.arange(0, len(mean_list), 3))
ax[0].set_xticklabels(num_list[::3])
ax[0].set_xlabel('Number of Observations', fontsize=11)

ax[1].plot(std_list)
ax[1].set_title('Standard Deviations of Posterior Distribution', fontsize=13)
ax[1].set_xticks(np.arange(0, len(std_list), 3))
ax[1].set_xticklabels(num_list[::3])
ax[1].set_xlabel('Number of Observations', fontsize=11)
```

(continues on next page)

(continued from previous page)

```
plt.show()
```

How shall we interpret the patterns above?

The answer is encoded in the Bayesian updating formulas.

It is natural to extend the one-step Bayesian update to an n-step Bayesian update.

$$\text{Prob}(\theta|k) = \frac{\text{Prob}(\theta, k)}{\text{Prob}(k)} = \frac{\text{Prob}(k|\theta) * \text{Prob}(\theta)}{\text{Prob}(k)} = \frac{\text{Prob}(k|\theta) * \text{Prob}(\theta)}{\int_0^1 \text{Prob}(k|\theta) * \text{Prob}(\theta) d\theta}$$

$$= \frac{\binom{N}{k}(1-\theta)^{N-k}\theta^k * \frac{\theta^{\alpha-1}(1-\theta)^{\beta-1}}{B(\alpha,\beta)}}{\int_0^1 \binom{N}{k}(1-\theta)^{N-k}\theta^k * \frac{\theta^{\alpha-1}(1-\theta)^{\beta-1}}{B(\alpha,\beta)} d\theta}$$

$$= \frac{(1-\theta)^{\beta+N-k-1} * \theta^{\alpha+k-1}}{\int_0^1 (1-\theta)^{\beta+N-k-1} * \theta^{\alpha+k-1} d\theta}$$

$$= Beta(\alpha + k, \beta + N - k)$$

A beta distribution with α and β has the following mean and variance.

The mean is $\frac{\alpha}{\alpha+\beta}$

The variance is $\frac{\alpha\beta}{(\alpha+\beta)^2(\alpha+\beta+1)}$

- α can be viewed as the number of successes
- β can be viewed as the number of failures

The random variables k and $N - k$ are governed by Binomial Distribution with $\theta = 0.4$.

Call this the true data generating process.

According to the Law of Large Numbers, for a large number of observations, observed frequencies of k and $N - k$ will be described by the true data generating process, i.e., the population probability distribution that we assumed when generating the observations on the computer. (See Exercise 12.2.1).

Consequently, the mean of the posterior distribution converges to 0.4 and the variance withers to zero.

```
upper_bound = [ii.ppf(0.95) for ii in Bay_stat.posterior_list]
lower_bound = [ii.ppf(0.05) for ii in Bay_stat.posterior_list]

fig, ax = plt.subplots(figsize=(10, 6))
ax.scatter(np.arange(len(upper_bound)), upper_bound, label='95 th Quantile')
ax.scatter(np.arange(len(lower_bound)), lower_bound, label='05 th Quantile')

ax.set_xticks(np.arange(0, len(upper_bound), 2))
ax.set_xticklabels(num_list[::2])
ax.set_xlabel('Number of Observations', fontsize=12)
ax.set_title('Bayesian Coverage Intervals of Posterior Distributions', 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 statististian comes to believe that θ is near $.4$.

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.

12.4 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 $\text{Prob}(X|\theta)$.

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 *this quantecon lecture*.

We also apply these powerful methods to approximating Bayesian posteriors for non-conjugate priors in *this quantecon lecture* and *this quantecon lecture*

CHAPTER

THIRTEEN

MULTIVARIATE HYPERGEOMETRIC DISTRIBUTION

Contents

- *Multivariate Hypergeometric Distribution*
  - *Overview*
  - *The Administrator's Problem*
  - *Usage*

13.1 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

13.2 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, $i = 1, 2, \dots, c$

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.

13.2.1 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

$$X = \begin{pmatrix} k_1 \\ k_2 \\ \vdots \\ k_c \end{pmatrix}.$$

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

$$X = \begin{pmatrix} k_1 \\ k_2 \\ \vdots \\ k_4 \end{pmatrix}.$$

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**.

13.2.2 Multivariate Hypergeometric Distribution

Let's start with some imports.

```
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
from scipy.special import comb
from scipy.stats import normaltest
from numba import njit, 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)
```

(continues on next page)

(continued from previous page)

```
    def pmf(self, k_arr):
        """
        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.
```

(continues on next page)

(continued from previous page)

```
        Parameters
        ----------
        n: int
            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
```

13.3 Usage

13.3.1 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

$$P(2 \text{ black}, 2 \text{ white}, 2 \text{ red}) = \frac{\binom{5}{2}\binom{10}{2}\binom{15}{2}}{\binom{30}{6}} = 0.079575596816976$$

```
# 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 $X = \begin{pmatrix} 2 & 2 & 2 \end{pmatrix}$

```
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]])
```

13.3.2 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([3.9581715, 0.2773806, 1.1594565, 0.6049914])
```

```
# variance covariance matrix
np.cov(sample.T)
```

```
array([[ 1.31818601, -0.1792688 , -0.74840555, -0.39051166],
       [-0.1792688 ,  0.25904483, -0.05221924, -0.02755678],
       [-0.74840555, -0.05221924,  0.91540862, -0.11478382],
       [-0.39051166, -0.02755678, -0.11478382,  0.53285226]])
```

Evidently, the sample means and covariances approximate their population counterparts well.

13.3.3 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)
```

(continues on next page)

(continued from previous page)

```
    return np.exp(-z / (2 * (1 - ρ**2))) / denom
```

```
@njit
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 (k_i, k_j) 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([0.48809858, 0.90163965, 0.22282301, 0.41117551])
```

The lesson to take away from this is that the normal approximation is imperfect.

CHAPTER

FOURTEEN

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*

14.1 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

14.2 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
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
from numba import njit
import statsmodels.api as sm
```

Assume that an $N \times 1$ random vector z has a multivariate normal probability density.

This means that the probability density takes the form

$$f(z; \mu, \Sigma) = (2\pi)^{-\left(\frac{N}{2}\right)} \det(\Sigma)^{-\frac{1}{2}} \exp\left(-.5 (z-\mu)' \Sigma^{-1} (z-\mu)\right)$$

where $\mu = Ez$ is the mean of the random vector z and $\Sigma = E(z-\mu)(z-\mu)'$ is the covariance matrix of z.

The covariance matrix Σ is symmetric and positive definite.

```
@njit
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 covarianece matrix of z, N by 1
    """

    z = np.atleast_2d(z)
```

(continues on next page)

(continued from previous page)

```
    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 $k \in \{1, \dots, N-1\}$, 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

$$\hat{\Sigma}_{11} = \Sigma_{11} - \Sigma_{12}\Sigma_{22}^{-1}\Sigma_{21} = \Sigma_{11} - \beta\Sigma_{22}\beta'$$

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.
```

(continues on next page)

(continued from previous page)

```
    Parameters
    ----------
    μ: ndarray(float, dim=1)
        the mean of z, N by 1
    Σ: ndarray(float, dim=2)
        the covarianece matrix of z, N by 1

    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.
        Σ_hat: ndarray(float, ndim=2)
            The conditional covariance matrix of z1 or z2.
        """
```

(continues on next page)

(continued from previous page)

```
        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.

14.3 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.

```
beta = multi_normal.βs

a1 = μ[0] - beta[0]*μ[1]
b1 = beta[0]

a2 = μ[1] - beta[1]*μ[0]
b2 = beta[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')
ax.spines['bottom'].set_position('zero')
ax.spines['right'].set_color('none')
```

(continues on next page)

(continued from previous page)

```
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.title('two regressions')
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()
```

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 = np.random.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([0.5013179]))
```

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]]), 0.7502937302551068)
```

Lastly, let's compute the estimate of $E\hat{z_1}|z_2$ and compare it with $\hat{\mu}_1$

```
μ1_hat, results.predict(z2 - μ2) + μ1
```

```
(array([2.5]), array([2.50527159]))
```

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.

14.4 Trivariate Example

Let's apply our code to a trivariate example.

We'll specify the mean vector and the covariance matrix as follows.

```
μ = np.random.random(3)
C = np.random.random((3, 3))
Σ = C @ C.T # positive semi-definite

multi_normal = MultivariateNormal(μ, Σ)
```

```
μ, Σ
```

```
(array([0.1004995 , 0.5664604 , 0.62990434]),
 array([[1.11682157, 0.72527344, 0.40007789],
        [0.72527344, 0.79010438, 0.50020188],
        [0.40007789, 0.50020188, 0.32838098]]))
```

```
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 = np.random.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([[ 4.11163385, -5.04465606]]), array([ 4.11465659, -5.04918581]))
```

```
Σ1_hat, results.resid @ results.resid.T / (n - 1)
```

```
(array([[0.15301807]]), 0.15323449102163417)
```

```
μ1_hat, results.predict(z2 - μ2) + μ1
```

```
(array([-16.05094011]), array([-16.06640233]))
```

Once again, sample analogues do a good job of approximating their populations counterparts.

14.5 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.

$$y_i = \theta + \sigma_y w_i, \quad i = 1, \dots, n$$

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.,
```

(continues on next page)

(continued from previous page)

```
        100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
        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.]]),
 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.]]))
```

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 = np.random.multivariate_normal(μ_IQ, Σ_IQ)
y = x[:-1] # test scores
θ = x[-1]  # IQ
```

```
# the true value
θ
```

```
101.94716848514508
```

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([101.76117969]), 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='$\hat{μ}_{θ}$')
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('$\hat{θ}$')
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 $\frac{1}{n^{.5}}$.

14.6 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

$$\Sigma \equiv DD' = CC'$$

and

$$E\epsilon\epsilon' = I.$$

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 $\left(X - \mu_\theta 1_{n+1}\right)$.

We can say that ϵ is an orthogonal basis for $\left(X - \mu_\theta 1_{n+1}\right)$.

Let c_i be the ith element in the last row of C.

Then we can write

$$\theta = \mu_\theta + c_1\epsilon_1 + c_2\epsilon_2 + \cdots + c_n\epsilon_n + c_{n+1}\epsilon_{n+1} \tag{14.1}$$

The mutual orthogonality of the ϵ_i's provides us with an informative way to interpret them in light of equation (14.1).

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 (14.1) 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 μ9 and Σ9 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([np.sum(C[n, i+1:n+1] ** 2) 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
```

```
True
```

```
# conditional variance
np.max(np.abs(Σθ_hat_arr - Σθ_hat_arr_C)) < 1e-10
```

```
True
```

14.7 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 (14.1),

- the random variable $c_i \epsilon_i$ is information about θ that is not contained by the information in $\epsilon_1, \epsilon_2, \dots, \epsilon_{i-1}$
- the coefficient c_i is the simple population regression coefficient of $\theta - \mu_\theta$ on ϵ_i

14.8 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
```

(continues on next page)

(continued from previous page)

```
    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 = np.random.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
```

```
(108.13476065193913, 96.68232460482335)
```

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
```

(continues on next page)

(continued from previous page)

```
multi_normal_IQ2d.partition(k)

multi_normal_IQ2d.cond_dist(1, [*y1, *y2])
```

```
(array([104.39841597, 103.07943049]),
 array([[33.33333333,  0.        ],
        [ 0.        , 33.33333333]]))
```

Now let's compute distributions of θ and μ 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 104.40 and 33.33␣
↪respectively
The mean and variance of θ conditional on y1, y2          are 104.40 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 103.08 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 103.08 and 33.33␣
↪respectively
```

Evidently, math tests provide no information about μ and language tests provide no information about η.

14.9 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:

$$\begin{aligned} x_0 &\sim N\left(0, \sigma_0^2\right) \\ x_{t+1} &= a x_t + b w_{t+1}, \quad w_{t+1} \sim N\left(0,1\right), t \geq 0 \\ y_t &= c x_t + d v_t, \quad v_t \sim N\left(0,1\right), t \geq 0 \end{aligned}$$

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

$$\Sigma_y = EYY' = C\Sigma_x C' + DD'$$

By stacking X and Y, we can write

$$Z = \begin{bmatrix} X \\ Y \end{bmatrix}$$

and

$$\Sigma_z = EZZ' = \begin{bmatrix} \Sigma_x & \Sigma_x C' \\ C\Sigma_x & \Sigma_y \end{bmatrix}$$

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   ],
```

(continues on next page)

(continued from previous page)

```
       [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   ,
        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 = np.random.multivariate_normal(μz, Σz)

x = z[:T+1]
y = z[T+1:]
```

14.9.1 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 =  [0.19135217 0.6333543  1.70366634 1.59828953]
Y =  [0.15897865 0.6156868  1.66989805 1.62642912]
 E [ X | Y] =
```

```
(array([0.15964452, 0.6170057 , 1.66740008, 1.62611548]),
 array([[2.48875094e-03, 5.57449314e-06, 1.24861729e-08, 2.80236945e-11],
```

(continues on next page)

(continued from previous page)

```
        [5.57449314e-06, 2.48876343e-03, 5.57452116e-06, 1.25113944e-08],
        [1.24861728e-08, 5.57452116e-06, 2.48876346e-03, 5.58575339e-06],
        [2.80236945e-11, 1.25113941e-08, 5.58575339e-06, 2.49377812e-03]]))
```

14.9.2 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 $[x_t, y_0, \dots, y_{t-2}, y_{t-1}]$.

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([1.50040666]), array([[1.00201996]]))
```

14.9.3 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]
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([0.49775287]), array([[1.81413617]]))
```

14.9.4 Constructing a Wold Representation

Now we'll apply Cholesky decomposition to decompose $\Sigma_y = HH'$ 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([0.1587803 , 0.47189755, 1.1143238 , 0.12573861])
```

```
y
```

```
array([0.15897865, 0.6156868 , 1.66989805, 1.62642912])
```

This example is an instance of what is known as a **Wold representation** in time series analysis.

14.10 Stochastic Difference Equation

Consider the stochastic second-order linear difference equation

$$y_t = \alpha_0 + \alpha_1 y_{y-1} + \alpha_2 y_{t-2} + u_t$$

where $u_t \sim N\left(0, \sigma_u^2\right)$ and

$$\left[\begin{array}{c} y_{-1} \\ y_0 \end{array}\right] \sim N\left(\mu_{\tilde{y}}, \Sigma_{\tilde{y}}\right)$$

It can be written as a stacked system

$$\underbrace{\left[\begin{array}{cccccccc} 1 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ -\alpha_1 & 1 & 0 & 0 & \cdots & 0 & 0 & 0 \\ -\alpha_2 & -\alpha_1 & 1 & 0 & \cdots & 0 & 0 & 0 \\ 0 & -\alpha_2 & -\alpha_1 & 1 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \cdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & -\alpha_2 & -\alpha_1 & 1 \end{array}\right]}_{\equiv A} \left[\begin{array}{c} y_1 \\ y_2 \\ y_3 \\ y_4 \\ \vdots \\ y_T \end{array}\right] = \underbrace{\left[\begin{array}{c} \alpha_0 + \alpha_1 y_0 + \alpha_2 y_{-1} \\ \alpha_0 + \alpha_2 y_0 \\ \alpha_0 \\ \alpha_0 \\ \vdots \\ \alpha_0 \end{array}\right]}_{\equiv b}$$

We can compute y by solving the system

$$y = A^{-1}\left(b + u\right)$$

We have

$$\begin{aligned} \mu_y &= A^{-1}\mu_b \\ \Sigma_y &= A^{-1} E\left[\left(b - \mu_b + u\right)\left(b - \mu_b + u\right)'\right]\left(A^{-1}\right)' \\ &= A^{-1}\left(\Sigma_b + \Sigma_u\right)\left(A^{-1}\right)' \end{aligned}$$

where

$$\mu_b = \left[\begin{array}{c} \alpha_0 + \alpha_1 \mu_{y_0} + \alpha_2 \mu_{y_{-1}} \\ \alpha_0 + \alpha_2 \mu_{y_0} \\ \alpha_0 \\ \vdots \\ \alpha_0 \end{array}\right]$$

$$\Sigma_b = \begin{bmatrix} C\Sigma_{\tilde{y}}C' & 0_{N-2\times N-2} \\ 0_{N-2\times 2} & 0_{N-2\times N-2} \end{bmatrix}, \quad C = \begin{bmatrix} \alpha_2 & \alpha_1 \\ 0 & \alpha_2 \end{bmatrix}$$

$$\Sigma_u = \begin{bmatrix} \sigma_u^2 & 0 & \cdots & 0 \\ 0 & \sigma_u^2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & \sigma_u^2 \end{bmatrix}$$

```
# set parameters
T = 80
T = 160
# coefficients of the second order difference equation
0 = 10
1 = 1.53
2 = -.9

# variance of u
σu = 1.
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^{\prime}
A = np.zeros((T, T))

for i in range(T):
    A[i, i] = 1

    if i-1 >= 0:
        A[i, i-1] = -1

    if i-2 >= 0:
        A[i, i-2] = -2

A_inv = np.linalg.inv(A)
```

```
# compute the mean vectors of b and y
μb = np.full(T, 0)
μb[0] += 1 * μy_tilde[1] + 2 * μy_tilde[0]
μb[1] += 2 * 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([[2, 1], [0, 2]])
Σb[:2, :2] = C @ Σy_tilde @ C.T

Σy = A_inv @ (Σb + Σu) @ A_inv.T
```

14.11 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

$$\mu_p = B\mu_y$$
$$\Sigma_p = B\Sigma_y B'$$

```
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

$$\mu_z = D\mu_y$$

$$\Sigma_z = D\Sigma_y D'$$

```
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 = np.random.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 green 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.

14.12 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

$$x_0 = \tilde{x}_0 + \zeta_0 \tag{14.2}$$

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 $E[\zeta_0 \zeta_0' | y_0] = \tilde{\Sigma}_0$.

14.12.1 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 (14.2), 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

$$K_0 = A\Sigma_0 G'(G\Sigma_0 G' + R)^{-1}$$

14.12.2 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

$$x_t | y^{t-1} \sim \mathcal{N}(A\tilde{x}_t, A\tilde{\Sigma}_t A' + CC')$$

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

$$P_{t-1} = R + A'P_tA - A'P_tB(B'P_tB + Q)^{-1}B'P_tA.$$

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 resemblences.

- 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.

14.12.3 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], [.3, 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([-0.078125,  0.803125]),
 array([[ 0.72098214, -0.203125  ],
        [-0.403125  ,  0.228125  ]]))
```

```
A = np.array([[0.5, 0.2], [-0.1, 0.3]])
C = np.array([[2.], [1.]])
```

(continues on next page)

(continued from previous page)

```
# conditional distribution of x1
x1_cond = A @ μ1_hat
Σ1_cond = C @ C.T + A @ Σ11 @ A.T
x1_cond, Σ1_cond
```

```
(array([0.1215625, 0.24875  ]),
 array([[4.12874554, 1.95523214],
        [1.92123214, 1.04592857]]))
```

14.12.4 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))

    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.        ],
        [0.1215625 , 0.24875   ],
        [0.18680212, 0.06904689],
        [0.75576875, 0.05558463]]),
 array([[[1.        , 0.5       ],
         [0.3       , 2.        ]],

        [[4.12874554, 1.95523214],
         [1.92123214, 1.04592857]],
```

(continues on next page)

(continued from previous page)

```
        [[4.08198663, 1.99218488],
         [1.98640488, 1.00886423]],

        [[4.06457628, 2.00041999],
         [1.99943739, 1.00275526]]]))
```

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*

14.13 Classic Factor Analysis Model

The factor analysis model widely used in psychology and other fields can be represented as

$$Y = \Lambda f + U$$

where

1. Y is $n \times 1$ random vector, $EUU' = D$ is a diagonal matrix,
2. Λ is $n \times k$ coefficient matrix,
3. f is $k \times 1$ random vector, $Eff' = I$,
4. U is $n \times 1$ random vector, and $U \perp f$ (i.e., $EUf' = 0$)
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

$$\begin{aligned} \Sigma_y = EYY' &= \Lambda\Lambda' + D \\ EYf' &= \Lambda \\ EfY' &= \Lambda' \end{aligned}$$

Thus, the covariance matrix Σ_Y is the sum of a diagonal matrix D and a positive semi-definite matrix $\Lambda\Lambda'$ 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

$$\Sigma_z = EZZ' = \begin{pmatrix} I & \Lambda' \\ \Lambda & \Lambda\Lambda' + D \end{pmatrix}$$

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))

Σz[:k, :k] = np.eye(k)
Σz[:k, k:] = Λ.T
Σz[k:, :k] = Λ
Σz[k:, k:] = Σy
```

```
z = np.random.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([-0.44536073,  0.25951946]),
 array([[0.04761905, 0.        ],
        [0.        , 0.04761905]]))
```

We can verify that the conditional mean $E[f \mid Y = y] = BY$ where $B = \Lambda' \Sigma_y^{-1}$.

```
B = Λ.T @ np.linalg.inv(Σy)

B @ y
```

```
array([-0.44536073,  0.25951946])
```

Similarly, we can compute the conditional distribution $Y \mid f$.

```
multi_normal_factor.cond_dist(1, f)
```

```
(array([-0.50897287, -0.50897287, -0.50897287, -0.50897287, -0.50897287,
         0.32994912,  0.32994912,  0.32994912,  0.32994912,  0.32994912]),
 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([-0.50897287, -0.50897287, -0.50897287, -0.50897287, -0.50897287,
        0.32994912,  0.32994912,  0.32994912,  0.32994912,  0.32994912])
```

14.14 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

$$\Sigma_y = P\tilde{\Lambda}P'$$

where $\tilde{\Lambda}$ is a diagonal matrix.

We have

$$Y = P\epsilon$$

and

$$\epsilon = P'Y$$

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()
```

```
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.  ],
       [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 =  [ 0.60931831 -1.04564972 -0.03220843  0.79887235 -0.19350376 -0.4574276
 -0.77014932 -1.17613046 -0.44359808 -0.38322672]
```

```
# print the values of the two factors

print('f = ', f)
```

```
f =  [-0.50897287  0.32994912]
```

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='$\epsilon$')
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([ 0.60931831, -1.04564972])
```

```
# compare with Ef|y
B @ y
```

```
array([-0.44536073,  0.25951946])
```

The fraction of variance in y_t explained by the first two principal components can be computed as below.

```
λ_tilde[:2].sum() / λ_tilde.sum()
```

```
0.84
```

Compute

$$\hat{Y} = P_j \epsilon_j + P_k \epsilon_k$$

where P_j and P_k correspond to the largest two eigenvalues.

```
y_hat = P[:, :2] @ ε[:2]
```

In this example, it turns out that the projection $\hat{Y}$ of Y on the first two principal components does a good job of approximating $Ef \mid y$.

We confirm this in the following plot of f, $Ey \mid f$, $Ef \mid y$, and $\hat{y}$ on the coordinate axis versus y on the ordinate axis.

```
plt.scatter(range(N), Λ @ f, label='$Ey|f$')
plt.scatter(range(N), y_hat, label='$\hat{y}$')
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='$Ef_{1}|y$')
plt.hlines(Efy[1], N//2, N-1, ls='-.', color='b', label='$Ef_{2}|y$')
plt.legend()

plt.show()
```

The covariance matrix of $\hat{Y}$ can be computed by first constructing the covariance matrix of ϵ and then use the upper left block for 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]
 [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]]
```

CHAPTER

FIFTEEN

HEAVY-TAILED DISTRIBUTIONS

New website

We have moved this lecture to a new lecture series.

See An Undergraduate Lecture Series for the Foundations of Computational Economics

CHAPTER

SIXTEEN

FAULT TREE UNCERTAINTIES

16.1 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 [ESAW18] and Greenfield and Sargent [GS93] 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 [Apo90] 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 /opt/conda/envs/quantecon/lib/python3.
↪11/site-packages (0.8.10)
```

```
WARNING: Running pip as the 'root' user can result in broken permissions and␣
↪conflicting behaviour with the system package manager. It is recommended to use␣
↪a virtual environment instead: https://pip.pypa.io/warnings/venv
```

```
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)
```

16.2 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

$$\begin{aligned}
\text{mean:} &\quad e^{\mu+\frac{1}{2}\sigma^2} \\
\text{variance:} &\quad (e^{\sigma^2}-1)e^{2\mu+\sigma^2} \\
\text{median:} &\quad e^{\mu} \\
\text{mode:} &\quad e^{\mu-\sigma^2} \\
\text{.95 quantile:} &\quad e^{\mu+1.645\sigma} \\
\text{.95-.05 quantile ratio:} &\quad e^{1.645\sigma}
\end{aligned}$$

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.

16.3 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.

16.4 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
```

```
(244.69193226422038, 246.68348900599963, 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
```

```
(244.69059898302908, 244.69193226422038)
```

16.5 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

$$x(\omega_j) = \sum_{t=0}^{T-1} x_t \exp(-i\omega_j t) \tag{16.1}$$

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

$$x_t = T^{-1} \sum_{j=0}^{T-1} x(\omega_j) \exp(i\omega_j t) \tag{16.2}$$

The sequences $\{x_t\}_{t=0}^{T-1}$ and $\{x(\omega_j)\}_{j=0}^{T-1}$ contain the same information.

The pair of equations (16.1) and (16.2) 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 =  45.73965891699936 ; time with fftconvolve =  0.
↪16953839599955245
```

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
```

```
(489.3810974093853, 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
```

```
(734.0714863312277, 734.0757967926611)
```

16.6 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 [ESAW18].

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.

16.7 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

$$P(F) \approx \sum_{i=1}^{n} P(A_i) \tag{16.3}$$

Probabilities for each event are recorded as failure rates per year.

16.8 Failure Rates Unknown

Now we come to the problem that really interests us, following [ESAW18] and Greenfield and Sargent [GS93] in the spirit of Apostolakis [Apo90].

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 $P(A_i), i = 1, \dots, n$ 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 $i = 1, \dots, n$ 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 (16.3).

16.9 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 [GS93].

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 [GS93] 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 [GS93] 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)
```

(continues on next page)

(continued from previous page)

```
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)
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 =  10.198475473999679
```

```
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}"],
```

(continues on next page)

(continued from previous page)

```
    ['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 [GS93].

Discrepancies are probably due to slight differences in the number of digits retained in inputting $\mu_i, \sigma_i, i = 1, \dots, 14$ and in the number of points deployed in the discretizations.

CHAPTER

SEVENTEEN

INTRODUCTION TO ARTIFICIAL NEURAL NETWORKS

```
!pip install --upgrade jax jaxlib
!conda install -y -c plotly plotly plotly-orca retrying
```

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.

17.1 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

17.2 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

$$x_{i+1} = h_i \circ l_i(x_i), \quad , i = 1, \ldots N \tag{17.1}$$

starting from $x_1 = \tilde{x}$.

The value of x_{N+1} that emerges from this iterative scheme equals $\hat{f}(\tilde{x})$.

17.3 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 $x_{i+1} = h_i\left(w_i x_i + b_i\right)$.

For a given **prediction** $\hat{y}(x)$ and **target** $y = f(x)$, consider the loss function

$$\mathcal{L}\left(\hat{y}, y\right)(x) = \frac{1}{2}\left(\hat{y} - y\right)^2(x).$$

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:

$$\min_{\{(w_i, b_i)\}_{i=1}^N} \int \mathcal{L}\left(x_{N+1}, y\right)(x) d\mu(x)$$

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:

$$p_{k+1} = p_k - \alpha \frac{d\mathcal{L}}{dx_{N+1}} \frac{dx_{N+1}}{dp_k} \tag{17.2}$$

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**.

17.4 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 (17.1) 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

$$dx = (I - L)^{-1} Ddp$$

which in turn implies

$$\begin{pmatrix} dx_{N+1}/dw_1 \\ dx_{N+1}/db_1 \\ \vdots \\ dx_{N+1}/dw_N \\ dx_{N+1}/db_N \end{pmatrix} = e_N \left(I - L\right)^{-1} D.$$

We can then solve the above problem by applying our update for p multiple times for a collection of input-output pairs $\left\{(x_1^i, y^i)\right\}_{i=1}^M$ that we'll call our "training set".

17.5 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 (17.2) 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))
    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
```

(continues on next page)

(continued from previous page)

```
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([[8.5726520e-03, 4.0850643e-04, 6.1021703e-01],
       [1.7145304e-03, 2.3785220e-01, 1.0000000e+00]], dtype=float32)
```

```
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)
```

```
# 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(1.4901161e-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([[-1.3489482]], dtype=float32), Array([0.37956238], dtype=float32)),
 (Array([[-0.00782906]], dtype=float32), Array([0.44972023], dtype=float32)),
 (Array([[0.22937916]], dtype=float32), Array([-0.04793657], dtype=float32))]
```

```
update_ad(params, x, y)
```

```
[(Array([[-1.3489482]], dtype=float32), Array([0.37956238], dtype=float32)),
 (Array([[-0.00782906]], dtype=float32), Array([0.44972023], dtype=float32)),
 (Array([[0.22937916]], dtype=float32), Array([-0.04793657], dtype=float32))]
```

17.6 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 45 s, sys: 4.78 s, total: 49.7 s
Wall time: 31.9 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
```

17.7 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.

17.8 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
```

```
## to check that gpu is activated in environment

from jax.lib import xla_bridge
print(xla_bridge.get_backend().platform)
```

```
gpu
```

Note: Cloud Environment: This lecture site is built in a server environment that doesn't have access to a `gpu` If you run this lecture locally this lets you know where your code is being executed, either via the `cpu` or the `gpu`

CHAPTER

EIGHTEEN

RANDOMIZED RESPONSE SURVEYS

18.1 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 [War65].

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)

18.2 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 [War65] 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

$$L = [\pi p + (1-\pi)(1-p)]^{n_1} \left[(1-\pi)p + \pi(1-p)\right]^{n-n_1} \tag{18.1}$$

The log of the likelihood function is:

$$\log(L) = n_1 \log\left[\pi p + (1-\pi)(1-p)\right] + (n-n_1)\log\left[(1-\pi)p + \pi(1-p)\right] \tag{18.2}$$

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

$$\pi p + (1-\pi)(1-p) = \frac{n_1}{n} \tag{18.3}$$

If $p \neq \frac{1}{2}$, then the maximum likelihood estimator (MLE) of π is:

$$\hat{\pi} = \frac{p-1}{2p-1} + \frac{n_1}{(2p-1)n} \tag{18.4}$$

We compute the mean and variance of the MLE estimator $\hat{\pi}$ to be:

$$\begin{aligned} \mathbb{E}(\hat{\pi}) &= \frac{1}{2p-1}\left[p - 1 + \frac{1}{n}\sum_{i=1}^{n}\mathbb{E}X_i\right] \\ &= \frac{1}{2p-1}\left[p - 1 + \pi p + (1-\pi)(1-p)\right] \\ &= \pi \end{aligned} \tag{18.5}$$

and

$$\begin{aligned} Var(\hat{\pi}) &= \frac{nVar(X_i)}{(2p-1)^2 n^2} \\ &= \frac{\left[\pi p + (1-\pi)(1-p)\right]\left[(1-\pi)p + \pi(1-p)\right]}{(2p-1)^2 n^2} \\ &= \frac{\frac{1}{4} + (2p^2 - 2p + \frac{1}{2})(-2\pi^2 + 2\pi - \frac{1}{2})}{(2p-1)^2 n^2} \\ &= \frac{1}{n}\left[\frac{1}{16(p-\frac{1}{2})^2} - (\pi - \frac{1}{2})^2\right] \end{aligned} \tag{18.6}$$

Equation (18.5) indicates that $\hat{\pi}$ is an **unbiased estimator** of π while equation (18.6) tell us the variance of the estimator.

To compute a confidence interval, first rewrite (18.6) as:

$$Var(\hat{\pi}) = \frac{\frac{1}{4} - (\pi - \frac{1}{2})^2}{n} + \frac{\frac{1}{16(p-\frac{1}{2})^2} - \frac{1}{4}}{n} \tag{18.7}$$

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 (18.1) 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 (18.5) and (18.7) we can deduce that:

- The MSE of $\hat{\pi}$ decreases as p increases.

18.3 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:

$$\hat{\pi} = \frac{\sum_{i=1}^{n} Y_i}{n} \tag{18.8}$$

We calculate the expectation, bias, and variance of the estimator to be:

$$\mathbb{E}(\hat{\pi}) = \pi T_a + [(1-\pi)(1-T_b)] \tag{18.9}$$

$$\begin{aligned} Bias(\hat{\pi}) &= \mathbb{E}(\hat{\pi} - \pi) \\ &= \pi[T_a + T_b - 2] + [1 - T_b] \end{aligned} \tag{18.10}$$

$$Var(\hat{\pi}) = \frac{[\pi T_a + (1-\pi)(1-T_b)]\,[1 - \pi T_a - (1-\pi)(1-T_b)]}{n} \tag{18.11}$$

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()
        np.random.seed(seed)
        sample = np.random.rand(size, self.n) <= A
        random_device = np.random.rand(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 = np.random.rand(size, self.n) <= irow.T_a
            truth_b = np.random.rand(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  \
T_a  T_b  Bias
0.95 1.00 -0.030060000000000087               5.76               1.36
0.90 1.00 -0.060045000000000015               1.73               0.41
0.70 1.00 -0.17952999999999997                0.21               0.05
0.50 1.00 -0.300077                           0.07               0.02
1.00 0.95 0.019769999999999954               10.59                2.5
     0.90 0.04005000000000014                 3.63               0.86
     0.70 0.12005199999999994                 0.46               0.11
     0.50 0.1997460000000001                  0.17               0.04
0.95 0.95 -0.010137000000000063              18.65               4.41
0.90 0.90 -0.020103000000000093              10.48               2.48
0.70 0.70 -0.060487999999999875               1.71                0.4
```

(continues on next page)

(continued from previous page)

```
0.50 0.50 -0.09934100000000001                0.66                0.16

                                   MSE Ratio: p = 0.8 MSE Ratio: p = 0.9
T_a  T_b  Bias
0.95 1.00 -0.030060000000000087               0.63                0.35
0.90 1.00 -0.060045000000000015               0.19                 0.1
0.70 1.00 -0.17952999999999997                0.02                0.01
0.50 1.00 -0.300077                           0.01                 0.0
1.00 0.95 0.019769999999999954                1.15                0.64
     0.90 0.04005000000000014                 0.39                0.22
     0.70 0.12005199999999994                 0.05                0.03
     0.50 0.1997460000000001                  0.02                0.01
0.95 0.95 -0.010137000000000063               2.02                1.12
0.90 0.90 -0.020103000000000093               1.14                0.63
0.70 0.70 -0.060487999999999875               0.19                 0.1
0.50 0.50 -0.09934100000000001                0.07                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 [War65]:

- $\pi_A = 0.5$
- $n = 1000$

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
```

(continues on next page)

(continued from previous page)

```
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
     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  \
T_a  T_b  Bias
0.95 1.00 -0.02523000000000003                   7.0               1.69
0.90 1.00 -0.05027900000000002                  2.23               0.54
0.70 1.00 -0.14986600000000005                  0.27               0.07
0.50 1.00 -0.250211                              0.1               0.02
1.00 0.95 0.024410000000000043                  7.38               1.78
     0.90 0.04983899999999997                   2.26               0.54
     0.70 0.14976900000000004                   0.27               0.07
     0.50 0.24985100000000005                    0.1               0.02
0.95 0.95 -0.00025999999999998247              24.29               5.86
0.90 0.90 -0.00010899999999997023              25.73                6.2
0.70 0.70 -0.0004390000000000782               25.75               6.21
0.50 0.50 0.0007679999999999909                24.91               6.01

                                  MSE Ratio: p = 0.8 MSE Ratio: p = 0.9
T_a  T_b  Bias
0.95 1.00 -0.02523000000000003                  0.75               0.44
0.90 1.00 -0.05027900000000002                  0.24               0.14
0.70 1.00 -0.14986600000000005                  0.03               0.02
0.50 1.00 -0.250211                             0.01               0.01
1.00 0.95 0.024410000000000043                  0.79               0.46
     0.90 0.04983899999999997                   0.24               0.14
     0.70 0.14976900000000004                   0.03               0.02
     0.50 0.24985100000000005                   0.01               0.01
0.95 0.95 -0.00025999999999998247               2.59               1.52
0.90 0.90 -0.00010899999999997023               2.74               1.61
0.70 0.70 -0.0004390000000000782                2.74               1.61
0.50 0.50 0.0007679999999999909                 2.65               1.56
```

We can also revisit a calculation in the concluding section of Warner [War65] 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
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  \
T_a  T_b  Bias
0.95 1.00 -0.03031649999999997                3.27                 0.8
0.90 1.00 -0.0603515                          0.91                0.22
0.70 1.00 -0.18008749999999996                0.11                0.03
0.50 1.00 -0.299849                           0.04                0.01
1.00 0.95 0.01973400000000003                  6.7                1.64
     0.90 0.03976600000000008                 2.01                0.49
     0.70 0.11978900000000003                 0.24                0.06
     0.50 0.2001385                           0.09                0.02
0.95 0.95 -0.010474500000000053              14.78                3.61
0.90 0.90 -0.020373499999999933               6.32                1.54
0.70 0.70 -0.059945499999999985               0.92                0.23
0.50 0.50 -0.10010349999999996                0.34                0.08

                                MSE Ratio: p = 0.8 MSE Ratio: p = 0.9
T_a  T_b  Bias
0.95 1.00 -0.03031649999999997                0.34                0.19
0.90 1.00 -0.0603515                          0.09                0.05
0.70 1.00 -0.18008749999999996                0.01                0.01
0.50 1.00 -0.299849                            0.0                 0.0
1.00 0.95 0.01973400000000003                 0.69                0.39
```

(continues on next page)

(continued from previous page)

```
     0.90 0.03976600000000008                0.21                0.12
     0.70 0.11978900000000003                0.02                0.01
     0.50 0.2001385                          0.01                 0.0
0.95 0.95 -0.010474500000000053              1.53                0.85
0.90 0.90 -0.020373499999999933              0.65                0.36
0.70 0.70 -0.059945499999999985               0.1                0.05
0.50 0.50 -0.10010349999999996               0.03                0.02
```

Evidently, as n increases, the randomized response method does better performance in more situations.

18.4 Concluding Remarks

This QuantEcon lecture describes some alternative randomized response surveys.

That lecture presents a utilitarian analysis of those alternatives conducted by Lars Ljungqvist [Lju93].

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

CHAPTER

NINETEEN

EXPECTED UTILITIES OF RANDOM RESPONSES

19.1 Overview

This QuantEcon lecture describes randomized response surveys in the tradition of Warner [War65] that are designed to protect respondents' privacy.

Lars Ljungqvist [Lju93] 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 [Lan75], [Lan76], [LW76], [And76], [FPS77], [GKAH77], [GAESH69].

19.2 Privacy Measures

We consider randomized response models with only two possible answers, "yes" and "no."

The design determines probabilities

$$\begin{aligned}\text{Pr}(\text{yes}|A) &= 1 - \text{Pr}(\text{no}|A)\\ \text{Pr}(\text{yes}|A^{'}) &= 1 - \text{Pr}(\text{no}|A^{'})\end{aligned}$$

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:

$$\text{Pr}(A|r) = \frac{\pi_A \text{Pr}(r|A)}{\pi_A \text{Pr}(r|A) + (1-\pi_A)\text{Pr}(r|A^{'})} \tag{19.1}$$

19.3 Zoo of Concepts

At this point we describe some concepts proposed by various researchers

19.3.1 Leysieffer and Warner(1976)

The response r is regarded as jeopardizing with respect to A or $A^{'}$ if

$$\begin{gathered}\Pr(A|r) > \pi_A \\ \text{or} \\ \Pr(A^{'}|r) > 1 - \pi_A\end{gathered} \tag{19.2}$$

From Bayes's rule:

$$\frac{\Pr(A|r)}{\Pr(A^{'}|r)} \times \frac{(1-\pi_A)}{\pi_A} = \frac{\Pr(r|A)}{\Pr(r|A^{'})} \tag{19.3}$$

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:

$$\begin{gathered}g(r|A) = \frac{\Pr(r|A)}{\Pr(r|A^{'})} \\ \text{and} \\ g(r|A^{'}) = \frac{\Pr(r|A^{'})}{\Pr(r|A)}\end{gathered} \tag{19.4}$$

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,

$$\begin{gathered}g(\text{yes}|A) > 1 \\ \text{and} \\ g(\text{no}|A^{'}) > 1\end{gathered}$$

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

$$\Pr(A|\text{no}) = 0$$

19.3.2 Lanke(1976)

Lanke (1975) [Lan75] 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) [Lan76] argued that an appropriate measure of protection is to minimize

$$\max\{\Pr(A|\text{yes}), \Pr(A|\text{no})\} \tag{19.5}$$

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.

19.3.3 2.3 Fligner, Policello, and Singh

Fligner, Policello, and Singh reached similar conclusion as Lanke (1976). [FPS77]

They measured "private protection" as

$$\frac{1 - \max\{\Pr(A|\text{yes}), \Pr(A|\text{no})\}}{1 - \pi_A} \tag{19.6}$$

19.3.4 2.4 Greenberg, Kuebler, Abernathy, and Horvitz (1977)

[GKAH77]

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:

$$\Pr(\text{yes}|A) \times \Pr(A|\text{yes}) + \Pr(\text{no}|A) \times \Pr(A|\text{no}) \tag{19.7}$$

Similarly, the hazard for an individual who does not belong to A would be

$$\Pr(\text{yes}|A^{'}) \times \Pr(A|\text{yes}) + \Pr(\text{no}|A^{'}) \times \Pr(A|\text{no}) \tag{19.8}$$

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

$$\Pr(\text{yes}|A) \times \Pr(A|\text{yes}) \tag{19.9}$$

and

$$\Pr(\text{yes}|A^{'}) \times \Pr(A|\text{yes}) \tag{19.10}$$

This measure is just the first term in (19.7), i.e., the probability that an individual answers "yes" and is perceived to belong to A.

19.4 Respondent's Expected Utility

19.4.1 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

$$\frac{\partial U_i\left(\Pr(A|r_i), \phi_i\right)}{\partial \Pr(A|r_i)} < 0, \text{ for } \phi_i \in \{\text{truth}, \text{lie}\} \tag{19.11}$$

and

$$U_i\left(\Pr(A|r_i), \text{truth}\right) > U_i\left(\Pr(A|r_i), \text{lie}\right), \text{ for } \Pr(A|r_i) \in [0, 1] \tag{19.12}$$

Suppose now that correct answer for individual i is "yes".

Individual i would choose to answer truthfully if

$$U_i\left(\Pr(A|\text{yes}), \text{truth}\right) \geq U_i\left(\Pr(A|\text{no}), \text{lie}\right) \tag{19.13}$$

If the correct answer is "no", individual i would volunteer the correct answer only if

$$U_i\left(\Pr(A|\text{no}), \text{truth}\right) \geq U_i\left(\Pr(A|\text{yes}), \text{lie}\right) \tag{19.14}$$

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 (19.14) holds for sure.

Consequently, constraint (19.13) becomes the single necessary condition for individual i always to answer truthfully.

At equality, constraint $(10.\text{a})$ determines conditional probabilities that make the individual indifferent between telling the truth and lying when the correct answer is "yes":

$$U_i\left(\Pr(A|\text{yes}), \text{truth}\right) = U_i\left(\Pr(A|\text{no}), \text{lie}\right) \tag{19.15}$$

Equation (19.15) defines a "truth border".

Differentiating (19.15) with respect to the conditional probabilities shows that the truth border has a positive slope in the space of conditional probabilities:

$$\frac{\partial \Pr(A|\text{no})}{\partial \Pr(A|\text{yes})} = \frac{\frac{\partial U_i(\Pr(A|\text{yes}),\text{truth})}{\partial \Pr(A|\text{yes})}}{\frac{\partial U_i(\Pr(A|\text{no}),\text{lie})}{\partial \Pr(A|\text{no})}} > 0 \tag{19.16}$$

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 (19.15)) is at least as high as the utility of lying on the right side of (19.15).
- 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.

19.4.2 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 (19.11) and (19.11) 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='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='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='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:

$$U_i(\text{Pr}(A|r_i), \phi_i) = -\text{Pr}(A|r_i) + f(\phi_i)$$

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='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()
```

19.5 Utilitarian View of Survey Design

19.5.1 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) [And76] showed that the minimum variance estimate in the two-response model has variance

$$V(\Pr(A|\text{yes}), \Pr(A|\text{no})) = \frac{{\pi_A}^2(1-\pi_A)^2}{n} \times \frac{1}{\Pr(A|\text{yes}) - \pi_A} \times \frac{1}{\pi_A - \Pr(A|\text{no})} \tag{19.17}$$

where the random sample with replacement consists of n individuals.

We can use Expression (19.17) to draw iso-variance curves.

The following inequalities restrict the shapes of iso-variance curves:

$$\left.\frac{d\,\Pr(A|\text{no})}{d\,\Pr(A|\text{yes})}\right|_{\text{constant variance}} = \frac{\pi_A - \Pr(A|\text{no})}{\Pr(A|\text{yes}) - \pi_A} > 0 \tag{19.18}$$

$$\left.\frac{d^2\,\Pr(A|\text{no})}{d\,\Pr(A|\text{yes})^2}\right|_{\text{constant variance}} = -\frac{2\left[\pi_A - \Pr(A|\text{no})\right]}{\left[\Pr(A|\text{yes}) - \pi_A\right]^2} < 0 \tag{19.19}$$

From expression (19.17), (19.18) and (19.19) 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.

19.5.2 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 [Lju93], 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()
```

19.5.3 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 (19.12) 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.

19.6 Criticisms of Proposed Privacy Measures

We can use a utilitarian approach to analyze some privacy measures.

We'll enlist Python Code to help us.

19.6.1 Analysis of Method of Lanke's (1976)

Lanke (1976) recommends a privacy protection criterion that minimizes:

$$\max\{\Pr(A|\text{yes}), \Pr(A|\text{no})\} \tag{19.20}$$

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()
```

19.6.2 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.

19.6.3 Analysis on the Method of Chaudhuri and Mukerjee's (1988)

[CM88]

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])
```

(continues on next page)

(continued from previous page)

```
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(0.8, 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()
```

19.6.4 Method of Greenberg et al. (1977)

[GKAH77]

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:

$$\Pr(\text{yes}|A) \times \Pr(A|\text{yes}) + \Pr(\text{no}|A) \times \Pr(A|\text{no}) \tag{19.21}$$

The hazard for an individual who does not belong to A is

$$\Pr(\text{yes}|A^{'}) \times \Pr(A|\text{yes}) + \Pr(\text{no}|A^{'}) \times \Pr(A|\text{no}) \tag{19.22}$$

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

$$\Pr(\text{yes}|A) \times \Pr(A|\text{yes}) \tag{19.23}$$

and

$$\Pr(\text{yes}|A^{'}) \times \Pr(A|\text{yes}) \tag{19.24}$$

According to Greenberg et al. (1977), a respondent commits himself or herself to answer truthfully on the basis of a probability in (19.21) or (19.23) **before** randomly selecting the question to be answered.

Suppose that the appropriate privacy measure is captured by the notion of "limited hazard" in (19.23) and (19.24).

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.

19.7 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 [Lju93] 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

Linear Programming

CHAPTER

TWENTY

LINEAR PROGRAMMING

20.1 Overview

Linear programming problems either maximize or minimize a linear objective function subject to a set of linear equality and/or inequality constraints.

Linear programs come in pairs:

- an original **primal** problem, and
- an associated **dual** problem.

If a primal problem involves **maximization**, the dual problem involves **minimization**.

If a primal problem involves **minimization**, the dual problem involves **maximization**.

We provide a standard form of a linear program and methods to transform other forms of linear programming problems into a standard form.

We tell how to solve a linear programming problem using SciPy.

We describe the important concept of complementary slackness and how it relates to the dual problem.

Let's start with some standard imports.

```
import numpy as np
from scipy.optimize import linprog
import matplotlib.pyplot as plt
from matplotlib.patches import Polygon
```

20.2 Objective Function and Constraints

We want to minimize a **cost function** $c'x = \sum_{i=1}^n c_i x_i$ over feasible values of $x = (x_1, x_2, \dots, x_n)'$.

Here

- $c = (c_1, c_2, \dots, c_n)'$ is a **unit cost vector**, and
- $x = (x_1, x_2, \dots, x_n)'$ is a vector of **decision variables**

Decision variables are restricted to satisfy a set of linear equality and/or inequality constraints.

We describe the constraints with the following collections of n-dimensional vectors a_i and scalars b_i and associated sets indexing the equality and inequality constraints:

- a_i for $i \in M_i$, where M_1, M_2, M_3 are each sets of indexes

and a collection of scalers

- b_i for $i \in N_i$, where N_1, N_2, N_3 are each sets of indexes.

A linear programming can be stated as [Ber97]:

$$
\begin{aligned}
\min_x \; & c'x \\
\text{subject to} \; & a_i'x \geq b_i, && i \in M_1 \\
& a_i'x \leq b_i, && i \in M_2 \\
& a_i'x = b_i, && i \in M_3 \\
& x_j \geq 0, && j \in N_1 \\
& x_j \leq 0, && j \in N_2 \\
& x_j \text{ unrestricted}, && j \in N_3
\end{aligned}
\tag{20.1}
$$

A vector x that satisfies all of the constraints is called a **feasible solution**.

A collection of all feasible solutions is called a **feasible set**.

A feasible solution x that minimizes the cost function is called an **optimal solution**.

The corresponding value of cost function $c'x$ is called the **optimal value**.

If the feasible set is empty, we say that solving the linear programming problem is **infeasible**.

If, for any $K \in \mathbb{R}$, there exists a feasible solution x such that $c'x < K$, we say that the problem is **unbounded** or equivalently that the optimal value is $-\infty$.

20.3 Example 1: Production Problem

This example was created by [Ber97]

Suppose that a factory can produce two goods called Product 1 and Product 2.

To produce each product requires both material and labor.

Selling each product generates revenue.

Required per unit material and labor inputs and revenues are shown in table below:

	Product 1	Product 2
Material	2	5
Labor	4	2
Revenue	3	4

30 units of material and 20 units of labor available.

A firm's problem is to construct a production plan that uses its 30 units of materials and 20 unites of labor to maximize its revenue.

Let x_i denote the quantity of Product i that the firm produces.

This problem can be formulated as:

$$
\begin{aligned}
\max_{x_1, x_2} \; & z = 3x_1 + 4x_2 \\
\text{subject to } \; & 2x_1 + 5x_2 \leq 30 \\
& 4x_1 + 2x_2 \leq 20 \\
& x_1, x_2 \geq 0
\end{aligned}
$$

The following graph illustrates the firm's constraints and iso-revenue lines.

```
fig, ax = plt.subplots(figsize=(8, 6))
ax.grid()

# Draw constraint lines
ax.hlines(0, -1, 17.5)
ax.vlines(0, -1, 12)
ax.plot(np.linspace(-1, 17.5, 100), 6-0.4*np.linspace(-1, 17.5, 100), color="c")
ax.plot(np.linspace(-1, 5.5, 100), 10-2*np.linspace(-1, 5.5, 100), color="c")
ax.text(1.5, 8, "$2x_1 + 5x_2 \leq 30$", size=12)
ax.text(10, 2.5, "$4x_1 + 2x_2 \leq 20$", size=12)
ax.text(-2, 2, "$x_2 \geq 0$", size=12)
ax.text(2.5, -0.7, "$x_1 \geq 0$", size=12)

# Draw the feasible region
feasible_set = Polygon(np.array([[0, 0],
                                 [0, 6],
                                 [2.5, 5],
                                 [5, 0]]),
                       color="cyan")
ax.add_patch(feasible_set)

# Draw the objective function
ax.plot(np.linspace(-1, 5.5, 100), 3.875-0.75*np.linspace(-1, 5.5, 100), color="orange
 ↪")
ax.plot(np.linspace(-1, 5.5, 100), 5.375-0.75*np.linspace(-1, 5.5, 100), color="orange
 ↪")
ax.plot(np.linspace(-1, 5.5, 100), 6.875-0.75*np.linspace(-1, 5.5, 100), color="orange
 ↪")
ax.arrow(-1.6, 5, 0, 2, width = 0.05, head_width=0.2, head_length=0.5, color="orange")
ax.text(5.7, 1, "$z = 3x_1 + 4x_2$", size=12)

# Draw the optimal solution
ax.plot(2.5, 5, "*", color="black")
ax.text(2.7, 5.2, "Optimal Solution", size=12)

plt.show()
```

The blue region is the feasible set within which all constraints are satisfied.

Parallel orange lines are iso-revenue lines.

The firm's objective is to find the parallel orange lines to the upper boundary of the feasible set.

The intersection of the feasible set and the highest orange line delineates the optimal set.

In this example, the optimal set is the point $(2.5, 5)$.

20.4 Example 2: Investment Problem

We now consider a problem posed and solved by [Hu18].

A mutual fund has $\$100,000$ to be invested over a three year horizon.

Three investment options are available:

1. **Annuity:** the fund can pay a same amount of new capital at the beginning of each of three years and receive a payoff of 130% of **total capital** invested at the end of the third year. Once the mutual fund decides to invest in this annuity, it has to keep investing in all subsequent years in the three year horizon.
2. **Bank account:** the fund can deposit any amount into a bank at the beginning of each year and receive its capital plus 6% interest at the end of that year. In addition, the mutual fund is permitted to borrow no more than $20,000 at the beginning of each year and is asked to pay back the amount borrowed plus 6% interest at the end of the year. The mutual fund can choose whether to deposit or borrow at the beginning of each year.

3. **Corporate bond:** At the beginning of the second year, a corporate bond becomes available. The fund can buy an amount that is no more than \$50,000 of this bond at the beginning of the second year and at the end of the third year receive a payout of 130% of the amount invested in the bond.

The mutual fund's objective is to maximize total payout that it owns at the end of the third year.

We can formulate this as a linear programming problem.

Let x_1 be the amount of put in the annuity, x_2, x_3, x_4 be bank deposit balances at the beginning of the three years, and x_5 be the amount invested in the corporate bond.

When x_2, x_3, x_4 are negative, it means that the mutual fund has borrowed from bank.

The table below shows the mutual fund's decision variables together with the timing protocol described above:

	Year 1	Year 2	Year 3
Annuity	x_1	x_1	x_1
Bank account	x_2	x_3	x_4
Corporate bond	0	x_5	0

The mutual fund's decision making proceeds according to the following timing protocol:

1. At the beginning of the first year, the mutual fund decides how much to invest in the annuity and how much to deposit in the bank. This decision is subject to the constraint:

$$x_1 + x_2 = 100,000$$

2. At the beginning of the second year, the mutual fund has a bank balance of $1.06x_2$. It must keep x_1 in the annuity. It can choose to put x_5 into the corporate bond, and put x_3 in the bank. These decisions are restricted by

$$x_1 + x_5 = 1.06x_2 - x_3$$

3. At the beginning of the third year, the mutual fund has a bank account balance equal to $1.06x_3$. It must again invest x_1 in the annuity, leaving it with a bank account balance equal to x_4. This situation is summarized by the restriction:

$$x_1 = 1.06x_3 - x_4$$

The mutual fund's objective function, i.e., its wealth at the end of the third year is:

$$1.30 \cdot 3x_1 + 1.06x_4 + 1.30x_5$$

Thus, the mutual fund confronts the linear program:

$$\begin{aligned}
\max_x \; & 1.30 \cdot 3x_1 + 1.06x_4 + 1.30x_5 \\
\text{subject to } \; & x_1 + x_2 = 100,000 \\
& x_1 - 1.06x_2 + x_3 + x_5 = 0 \\
& x_1 - 1.06x_3 + x_4 = 0 \\
& x_2 \geq -20,000 \\
& x_3 \geq -20,000 \\
& x_4 \geq -20,000 \\
& x_5 \leq 50,000 \\
& x_j \geq 0, \quad j = 1, 5 \\
& x_j \text{ unrestricted}, \quad j = 2, 3, 4
\end{aligned}$$

20.5 Standard Form

For purposes of

- unifying linear programs that are initially stated in superficially different forms, and
- having a form that is convenient to put into black-box software packages,

it is useful to devote some effort to describe a **standard form**.

Our standard form is:

$$
\begin{aligned}
\min_x \; & c_1 x_1 + c_2 x_2 + \cdots + c_n x_n \\
\text{subject to } \; & a_{11} x_1 + a_{12} x_2 + \cdots + a_{1n} x_n = b_1 \\
& a_{21} x_1 + a_{22} x_2 + \cdots + a_{2n} x_n = b_2 \\
& \quad \vdots \\
& a_{m1} x_1 + a_{m2} x_2 + \cdots + a_{mn} x_n = b_m \\
& x_1, x_2, \ldots, x_n \geq 0
\end{aligned}
$$

Let

$$
A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ & & \vdots & \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}, \quad b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}, \quad c = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix}, \quad x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}.
$$

The standard form LP problem can be expressed concisely as:

$$
\begin{aligned}
\min_x \; & c' x \\
\text{subject to } \; & Ax = b \\
& x >= 0
\end{aligned}
\tag{20.2}
$$

Here, $Ax = b$ means that the i-th entry of Ax equals the i-th entry of b for every i.

Similarly, $x >= 0$ means that x_j is greater than 0 for every j.

20.5.1 Useful Transformations

It is useful to know how to transform a problem that initially is not stated in the standard form into one that is.

By deploying the following steps, any linear programming problem can be transformed into an equivalent standard form linear programming problem.

1. **Objective Function:** If a problem is originally a constrained **maximization** problem, we can construct a new objective function that is the additive inverse of the original objective function. The transformed problem is then a **minimization** problem.
2. **Decision Variables:** Given a variable x_j satisfying $x_j \leq 0$, we can introduce a new variable $x_j' = -x_j$ and subsitute it into original problem. Given a free variable x_i with no restriction on its sign, we can introduce two new variables x_j^+ and x_j^- satisfying $x_j^+, x_j^- \geq 0$ and replace x_j by $x_j^+ - x_j^-$.
3. **Inequality constraints:** Given an inequality constraint $\sum_{j=1}^n a_{ij} x_j \leq 0$, we can introduce a new variable s_i, called a **slack variable** that satisfies $s_i \geq 0$ and replace the original constraint by $\sum_{j=1}^n a_{ij} x_j + s_i = 0$.

Let's apply the above steps to the two examples described above.

20.5.2 Example 1: Production Problem

The original problem is:

$$
\begin{aligned}
\max_{x_1, x_2} \; & 3x_1 + 4x_2 \\
\text{subject to} \; & 2x_1 + 5x_2 \leq 30 \\
& 4x_1 + 2x_2 \leq 20 \\
& x_1, x_2 \geq 0
\end{aligned}
$$

This problem is equivalent to the following problem with a standard form:

$$
\begin{aligned}
\min_{x_1, x_2} \; & -(3x_1 + 4x_2) \\
\text{subject to} \; & 2x_1 + 5x_2 + s_1 = 30 \\
& 4x_1 + 2x_2 + s_2 = 20 \\
& x_1, x_2, s_1, s_2 \geq 0
\end{aligned}
$$

20.5.3 Example 2: Investment Problem

The original problem is:

$$
\begin{aligned}
\max_{x} \; & 1.30 \cdot 3x_1 + 1.06x_4 + 1.30x_5 \\
\text{subject to} \; & x_1 + x_2 = 100,000 \\
& x_1 - 1.06x_2 + x_3 + x_5 = 0 \\
& x_1 - 1.06x_3 + x_4 = 0 \\
& x_2 \geq -20,000 \\
& x_3 \geq -20,000 \\
& x_4 \geq -20,000 \\
& x_5 \leq 50,000 \\
& x_j \geq 0, \quad j = 1, 5 \\
& x_j \text{ unrestricted}, \quad j = 2, 3, 4
\end{aligned}
$$

This problem is equivalent to the following problem with a standard form:

$$
\begin{aligned}
\min_{x} \; & -(1.30 \cdot 3x_1 + 1.06x_4^+ - 1.06x_4^- + 1.30x_5) \\
\text{subject to} \; & x_1 + x_2^+ - x_2^- = 100,000 \\
& x_1 - 1.06(x_2^+ - x_2^-) + x_3^+ - x_3^- + x_5 = 0 \\
& x_1 - 1.06(x_3^+ - x_3^-) + x_4^+ - x_4^- = 0 \\
& x_2^- - x_2^+ + s_1 = 20,000 \\
& x_3^- - x_3^+ + s_2 = 20,000 \\
& x_4^- - x_4^+ + s_3 = 20,000 \\
& x_5 + s_4 = 50,000 \\
& x_j \geq 0, \quad j = 1, 5 \\
& x_j^+, x_j^- \geq 0, \quad j = 2, 3, 4 \\
& s_j \geq 0, \quad j = 1, 2, 3, 4
\end{aligned}
$$

20.6 Computations

The package *scipy.optimize* provides a function ***linprog*** to solve linear programming problems with a form below:

$$\begin{aligned} \min_x \; & c'x \\ \text{subject to} \; & A_{ub}x \leq b_{ub} \\ & A_{eq}x = b_{eq} \\ & l \leq x \leq u \end{aligned}$$

Note: By default $l = 0$ and $u =$ None unless explicitly specified with the argument 'bounds'.

Let's apply this great Python tool to solve our two example problems.

20.6.1 Example 1: Production Problem

The problem is:

$$\begin{aligned} \max_{x_1, x_2} \; & 3x_1 + 4x_2 \\ \text{subject to} \; & 2x_1 + 5x_2 \leq 30 \\ & 4x_1 + 2x_2 \leq 20 \\ & x_1, x_2 \geq 0 \end{aligned}$$

```
# Construct parameters
c_ex1 = np.array([3, 4])

# Inequality constraints
A_ex1 = np.array([[2, 5],
                  [4, 2]])
b_ex1 = np.array([30,20])

# Solve the problem
# we put a negative sign on the objective as linprog does minimization
res_ex1 = linprog(-c_ex1, A_ub=A_ex1, b_ub=b_ex1)

res_ex1
```

```
        message: Optimization terminated successfully. (HiGHS Status 7: Optimal)
        success: True
         status: 0
            fun: -27.5
              x: [ 2.500e+00  5.000e+00]
            nit: 2
          lower:  residual: [ 2.500e+00  5.000e+00]
                 marginals: [ 0.000e+00  0.000e+00]
          upper:  residual: [       inf        inf]
                 marginals: [ 0.000e+00  0.000e+00]
          eqlin:  residual: []
                 marginals: []
        ineqlin:  residual: [ 0.000e+00  0.000e+00]
                 marginals: [-6.250e-01 -4.375e-01]
```

(continues on next page)

(continued from previous page)

```
mip_node_count: 0
mip_dual_bound: 0.0
       mip_gap: 0.0
```

The optimal plan tells the factory to produce 2.5 units of Product 1 and 5 units of Product 2; that generates a maximizing value of revenue of 27.5.

We are using the *linprog* function as a **black box**.

Inside it, Python first transforms the problem into standard form.

To do that, for each inequality constraint it generates one slack variable.

Here the vector of slack variables is a two-dimensional NumPy array that equals $b_{ub} - A_{ub}x$.

See the official documentation for more details.

Note: This problem is to maximize the objective, so that we need to put a minus sign in front of parameter vector c.

20.6.2 Example 2: Investment Problem

The problem is:

$$
\begin{aligned}
\max_x \; & 1.30 \cdot 3x_1 + 1.06x_4 + 1.30x_5 \\
\text{subject to } \; & x_1 + x_2 = 100,000 \\
& x_1 - 1.06x_2 + x_3 + x_5 = 0 \\
& x_1 - 1.06x_3 + x_4 = 0 \\
& x_2 \geq -20,000 \\
& x_3 \geq -20,000 \\
& x_4 \geq -20,000 \\
& x_5 \leq 50,000 \\
& x_j \geq 0, \quad j = 1,5 \\
& x_j \text{ unrestricted}, \quad j = 2,3,4
\end{aligned}
$$

Let's solve this problem using *linprog*.

```
# Construct parameters
rate = 1.06

# Objective function parameters
c_ex2 = np.array([1.30*3, 0, 0, 1.06, 1.30])

# Inequality constraints
A_ex2 = np.array([[1,  1,  0,  0,  0],
                  [1, -rate, 1, 0, 1],
                  [1, 0, -rate, 1, 0]])
b_ex2 = np.array([100000, 0, 0])

# Bounds on decision variables
bounds_ex2 = [(  0,    None),
              (-20000, None),
```

(continues on next page)

(continued from previous page)

```
              (-20000, None),
              (-20000, None),
              (  0,   50000)]

# Solve the problem
res_ex2 = linprog(-c_ex2, A_eq=A_ex2, b_eq=b_ex2,
                  bounds=bounds_ex2)

res_ex2
```

```
        message: Optimization terminated successfully. (HiGHS Status 7: Optimal)
        success: True
         status: 0
            fun: -141018.24349792697
              x: [ 2.493e+04  7.507e+04  4.649e+03 -2.000e+04  5.000e+04]
            nit: 0
          lower:  residual: [ 2.493e+04  9.507e+04  2.465e+04  0.000e+00
                              5.000e+04]
                 marginals: [ 0.000e+00  0.000e+00  0.000e+00  1.650e-01
                              0.000e+00]
          upper:  residual: [       inf        inf        inf        inf
                              0.000e+00]
                 marginals: [ 0.000e+00  0.000e+00  0.000e+00  0.000e+00
                             -1.470e-03]
          eqlin:  residual: [ 0.000e+00  0.000e+00  0.000e+00]
                 marginals: [-1.376e+00 -1.299e+00 -1.225e+00]
        ineqlin:  residual: []
                 marginals: []
  mip_node_count: 0
  mip_dual_bound: 0.0
         mip_gap: 0.0
```

Python tells us that the best investment strategy is:

1. At the beginning of the first year, the mutual fund should buy $\$24,927.75$ of the annuity. Its bank account balance should be $\$75,072.25$.
2. At the beginning of the second year, the mutual fund should buy $\$50,000$ of the corporate bond and keep invest in the annuity. Its bank account balance should be $\$4,648.83$.
3. At the beginning of the third year, the mutual fund should borrow $\$20,000$ from the bank and invest in the annuity.
4. At the end of the third year, the mutual fund will get payouts from the annuity and corporate bond and repay its loan from the bank. At the end it will own $141018.24, so that it's total net rate of return over the three periods is 41.02%.

20.7 Duality

Associated with a linear programming of form (20.1) with m constraints and n decision variables, there is an **dual** linear programming problem that takes the form (please see [Ber97])

$$
\begin{aligned}
\max_p \; & b'p \\
\text{subject to } \; & p_i \geq 0, && i \in M_1 \\
& p_i \leq 0, && i \in M_2 \\
& p_i \text{ unrestricted}, && i \in M_3 \\
& A_j'p \leq c_j, && j \in N_1 \\
& A_j'p \geq c_j, && j \in N_2 \\
& A_j'p = c_j, && j \in N_3
\end{aligned}
$$

Where A_j is j-th column of the m by n matrix A.

Note: In what follows, we shall use a_i' to denote the i-th row of A and A_j to denote the j-th column of A.

$$
A = \begin{bmatrix} a_1' \\ a_2' \\ \\ a_m' \end{bmatrix}.
$$

To construct the dual of linear programming problem (20.1), we proceed as follows:

1. For every constraint $a_i'x \geq (\leq or =) b_i$, $j = 1, 2, ..., m$, in the primal problem, we construct a corresponding dual variable p_i. p_i is restricted to be positive if $a_i'x \geq b_i$ or negative if $a_i'x \leq b_i$ or unrestricted if $a_i'x = b_i$. We construct the m-dimensional vector p with entries p_i.
2. For every variable x_j, $j = 1, 2, ..., n$, we construct a corresponding dual constraint $A_j'p \geq (\leq or =) c_j$. The constraint is $A_j'p \geq c_j$ if $x_j \leq 0$, $A_j'p \leq c_j$ if $x_j \geq 0$ or $A_j'p = c_j$ if x_j is unrestricted.
3. The dual problem is to **maximize** objective function $b'p$.

For a **maximization** problem, we can first transform it to an equivalent minimization problem and then follow the above steps above to construct the dual **minimization** problem.

We can easily verify that **the dual of a dual problem is the primal problem**.

The following table summarizes relationships between objects in primal and dual problems.

Objective: Min	Objective: Max
m constraints	m variables
constraint $\geq$	variable ≥ 0
constraint $\leq$	variable ≤ 0
constraint $=$	variable free
n variables	n constraints
variable ≥ 0	constraint $\leq$
variable ≤ 0	constraint $\geq$
variable free	constraint $=$

As an example, the dual problem of the standard form (20.2) is:

$$\begin{aligned} \max_p \; & b'p \\ \text{subject to} \;\; & A'p \leq c \end{aligned}$$

As another example, consider a linear programming problem with form:

$$\begin{aligned} \max_x \; & c'x \\ \text{subject to} \;\; & Ax \leq b \\ & x \geq 0 \end{aligned} \tag{20.3}$$

Its dual problem is:

$$\begin{aligned} \min_p \; & b'p \\ \text{subject to} \;\; & A'p \geq c \\ & p \geq 0 \end{aligned}$$

20.8 Duality Theorems

Primal and dual problems are linked by powerful **duality theorems** that have **weak** and **strong** forms.

The duality theorems provide the foundations of enlightening economic interpretations of linear programming problems.

Weak duality: For linear programming problem (20.1), if x and p are feasible solutions to the primal and the dual problems, respectively, then

$$b'p \leq c'x$$

Strong duality: For linear programming problem (20.1), if the primal problem has an optimal solution x, then the dual problem also has an optimal solution. Denote an optimal solution of the dual problem as p. Then

$$b'p = c'x$$

According to strong duality, we can find the optimal value for the primal problem by solving the dual problem.

But the dual problem tells us even more as we shall see next.

20.8.1 Complementary Slackness

Let x and p be feasible solutions to the primal problem (20.1) and its dual problem, respectively.

Then x and p are also optimal solutions of the primal and dual problems if and only if:

$$\begin{aligned} p_i(a_i'x - b_i) &= 0, \quad \forall i, \\ x_j(A_j'p - c_j) &= 0, \quad \forall j. \end{aligned}$$

This means that $p_i = 0$ if $a_i'x - b_i \neq 0$ and $x_j = 0$ if $A_j'p - c_j \neq 0$.

These are the celebrated **complementary slackness** conditions.

Let's interpret them.

20.8.2 Interpretations

Let's take a version of problem (20.3) as a production problem and consider its associated dual problem.

A factory produce n products with m types of resources.

Where $i = 1, 2, \dots, m$ and $j = 1, 2, \dots, n$, let

- x_j denote quantities of product j to be produced
- a_{ij} denote required amount of resource i to make one unit of product j,
- b_i denotes the avaliable amount of resource i
- c_j denotes the revenue generated by producing one unit of product j.

Dual variables: By strong duality, we have

$$c_1x_1 + c_2x_2 + \cdots + c_nx_n = b_1p_1 + b_2p_2 + \cdots + b_mp_m.$$

Evidently, a one unit change of b_i results in p_i units change of revenue.

Thus, a dual variable can be interpreted as the **value** of one unit of resource i.

This is why it is often called the **shadow price** of resource i.

For feasible but not optimal primal and dual solutions x and p, by weak duality, we have

$$c_1x_1 + c_2x_2 + \cdots + c_nx_n < b_1p_1 + b_2p_2 + \cdots + b_mp_m.$$

Note: Here, the expression is opposite to the statement above since primal problem is a minimization problem.

When a strict inequality holds, the solution is not optimal because it doesn't fully utilize all valuable resources.

Evidently,

- if a shadow price p_i is larger than the market price for Resource i, the factory should buy more Resource i and expand its scale to generate more revenue;
- if a shadow price p_i is less than the market price for Resource i, the factory should sell its Resource i.

Complementary slackness: If there exists i such that $a_i'x - b_i < 0$ for some i, then $p_i = 0$ by complementary slackness. $a_i'x - b_i < 0$ means that to achieve its optimal production, the factory doesn't require as much Resource i as it has. It is reasonable that the shadow price of Resource i is 0: some of its resource i is redundant.

If there exists j such that $A_j'p - c_j > 0$, then $x_j = 0$ by complementary slackness. $A_j'p - c_j > 0$ means that the value of all resources used when producing one unit of product j is greater than its cost.

This means that producing another product that can more efficiently utilize these resources is a better choice than producing product j

Since producing product j is not optimal, x_j should equal 0.

20.8.3 Example 1: Production Problem

This problem is one specific instance of the problem (20.3), whose economic meaning is interpreted above.

Its dual problem is:

$$
\begin{aligned}
\min_{x_1, x_2} \; & 30p_1 + 20p_2 \\
\text{subject to} \; & 2p_1 + 4p_2 \geq 3 \\
& 5p_1 + 2p_2 \geq 4 \\
& p_1, p_2 \geq 0
\end{aligned}
$$

We solve this dual problem by using the function *linprog*.

Since parameters used here are defined before when solving the primal problem, we won't define them here.

```
# Solve the dual problem
res_ex1_dual = linprog(b_ex1, A_ub=-A_ex1.T, b_ub=-c_ex1)

res_ex1_dual
```

```
        message: Optimization terminated successfully. (HiGHS Status 7: Optimal)
        success: True
         status: 0
            fun: 27.5
              x: [ 6.250e-01  4.375e-01]
            nit: 2
          lower:  residual: [ 6.250e-01  4.375e-01]
                 marginals: [ 0.000e+00  0.000e+00]
          upper:  residual: [       inf        inf]
                 marginals: [ 0.000e+00  0.000e+00]
          eqlin:  residual: []
                 marginals: []
        ineqlin:  residual: [ 0.000e+00  0.000e+00]
                 marginals: [-2.500e+00 -5.000e+00]
 mip_node_count: 0
 mip_dual_bound: 0.0
        mip_gap: 0.0
```

The optimal value for the dual problem equals 27.5.

This equals the optimal value of the primal problem, an illustration of strong duality.

Shadow prices for materials and labor are 0.625 and 0.4375, respectively.

20.8.4 Example 2: Investment Problem

The dual problem is:

$$
\begin{aligned}
\min_{p} \; & 100,000p_1 - 20,000p_4 - 20,000p_5 - 20,000p_6 + 50,000p_7 \\
\text{subject to} \; & p_1 + p_2 + p_3 \geq 1.30 \cdot 3 \\
& p_1 - 1.06p_2 + p_4 = 0 \\
& p_2 - 1.06p_3 + p_5 = 0 \\
& p_3 + p_6 = 1.06 \\
& p_2 + p_7 \geq 1.30 \\
& p_i \text{ unrestricted}, \quad i = 1, 2, 3 \\
& p_i \leq 0, \quad i = 4, 5, 6 \\
& p_7 \geq 0
\end{aligned}
$$

We solve this dual problem by using the function *linprog*.

```
# Objective function parameters
c_ex2_dual = np.array([100000, 0, 0, -20000, -20000, -20000, 50000])

# Equality constraints
A_eq_ex2_dual = np.array([[1, -1.06,     0,  1,  0,  0,  0],
                          [0,     1, -1.06,  0,  1,  0,  0],
                          [0,     0,     1,  0,  0,  1,  0]])
b_eq_ex2_dual = np.array([0, 0, 1.06])

# Inequality constraints
A_ub_ex2_dual = - np.array([[1, 1, 1, 0, 0, 0, 0],
                            [0, 1, 0, 0, 0, 0, 1]])
b_ub_ex2_dual = - np.array([1.30*3, 1.30])

# Bounds on decision variables
bounds_ex2_dual = [(None, None),
                   (None, None),
                   (None, None),
                   (None,    0),
                   (None,    0),
                   (None,    0),
                   (   0, None)]

# Solve the dual problem
res_ex2_dual = linprog(c_ex2_dual, A_eq=A_eq_ex2_dual, b_eq=b_eq_ex2_dual,
                       A_ub=A_ub_ex2_dual, b_ub=b_ub_ex2_dual, bounds=bounds_ex2_dual)

res_ex2_dual
```

```
        message: Optimization terminated successfully. (HiGHS Status 7: Optimal)
        success: True
         status: 0
            fun: 141018.2434979269
              x: [ 1.376e+00  1.299e+00  1.225e+00  0.000e+00  0.000e+00
                  -1.650e-01  1.470e-03]
            nit: 0
          lower:  residual: [       inf        inf        inf        inf
                             inf        inf  1.470e-03]
```

(continues on next page)

(continued from previous page)

```
                marginals: [ 0.000e+00  0.000e+00  0.000e+00  0.000e+00
                             0.000e+00  0.000e+00  0.000e+00]
         upper:  residual: [       inf        inf        inf  0.000e+00
                             0.000e+00  1.650e-01        inf]
                marginals: [ 0.000e+00  0.000e+00  0.000e+00 -9.507e+04
                            -2.465e+04  0.000e+00  0.000e+00]
         eqlin:  residual: [ 0.000e+00  0.000e+00  0.000e+00]
                marginals: [ 7.507e+04  4.649e+03 -2.000e+04]
       ineqlin:  residual: [ 0.000e+00  0.000e+00]
                marginals: [-2.493e+04 -5.000e+04]
mip_node_count: 0
mip_dual_bound: 0.0
       mip_gap: 0.0
```

The optimal value for the dual problem is 141018.24, which equals the value of the primal problem.

Now, let's interpret the dual variables.

By strong duality and also our numerical results, we have that optimal value is:

$$100,000p_1 - 20,000p_4 - 20,000p_5 - 20,000p_6 + 50,000p_7.$$

We know if b_i changes one dollor, then the optimal payoff in the end of the third year will change p_i dollars.

For $i = 1$, this means if the initial capital changes by one dollar, then the optimal payoff in the end of the third year will change p_1 dollars.

Thus, p_1 is the potential value of one more unit of initial capital, or the shadow price for initial capital.

We can also interpret p_1 as the prospective value in the end of the third year coming from having one more dollar to invest at the beginning of the first year.

If the mutual fund can raise money at a cost lower than $p_1 - 1$, then it should raise more money to increase its revenue.

But if it bears a cost of funds higher than $p_1 - 1$, the mutual fund shouldn't do that.

For $i = 4, 5, 6$, this means that if the amount of capital that the fund is permitted to borrow from the bank changes by one dollar, the optimal pay out at the end of the third year will change p_i dollars.

Thus, for $i = 4, 5, 6$, $|p_i|$ indicates the value of one dollar that the mutual fund can borrow from the bank at the beginning of the $i - 3$-th year.

$|p_i|$ is the shadow price for the loan amount. (We use absolute value here since $p_i \leq 0$.)

If the interest rate is lower than $|p_i|$, then the mutual fund should borrow to increase its optimal payoff; if the interest rate is higher, it is better to not do this.

For $i = 7$, this means that if the amount of the corporate bond the mutual fund can buy changes one dollar, then the optimal payoff will change p_7 dollars at the end of the third year. Again, p_7 is the shadow price for the amount of the corporate bond the mutual fund can buy.

As for numerical results

1. $p_1 = 1.38$, which means one dollar of initial capital is worth $\$1.38$ at the end of the third year.
2. $p_4 = p_5 = 0$, which means the loan amounts at the beginning of the first and second year are worth nothing. Recall that the optimal solution to the primal problem, $x_2, x_3 > 0$, which means at the beginning of the first and second year, the mutual fund has a postive bank account and borrows no capital from the bank. Thus, it is reasonable that the loan amounts at the beginning of the first and second year are valueless. This is what the complementary slackness conditions mean in this setting.

3. $p_6 = -0.16$, which means one dollar of the loan amount at the beginning of the third year is worth $\$0.16$. Since $|p_6|$ is higher than the interest rate 6%, the mutual fund should borrow as much as possible at the beginning of the third year. Recall that the optimal solution to the primal problem is $x_4 = -20,000$ which means the mutual fund borrows money from the bank as much as it can.

4. $p_7 = 0.0015$, which means one dollar of the amount of the corporate bond that the mutual fund can buy is worth $\$0.0015$.

CHAPTER

TWENTYONE

OPTIMAL TRANSPORT

21.1 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 [DSS58]. A modern reference about applications in economics is [Gal16].

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
```

21.2 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.
- $i = 1, 2, \dots, m$ and $j = 1, 2, \dots, n$.

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:

$$
\begin{aligned}
\min_{x_{ij}} \quad & \sum_{i=1}^{m} \sum_{j=1}^{n} c_{ij} x_{ij} \\
\text{subject to} \quad & \sum_{j=1}^{n} x_{ij} = p_i, \quad i = 1, 2, \dots, m \\
& \sum_{i=1}^{m} x_{ij} = q_j, \quad j = 1, 2, \dots, n \\
& x_{ij} \geq 0
\end{aligned}
\tag{21.1}
$$

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:

$$\sum_{j=1}^{n} q_j = \sum_{j=1}^{n}\sum_{i=1}^{m} x_{ij} = \sum_{i=1}^{m}\sum_{j=1}^{n} x_{ij} = \sum_{i=1}^{m} p_i \tag{21.2}$$

The presence of the restrictions in (21.2) will be the source of one redundancy in the complete set of restrictions that we describe below.

More about this later.

21.3 The Linear Programming Approach

In this section we discuss using using standard linear programming solvers to tackle the optimal transport problem.

21.3.1 Vectorizing a Matrix of Decision Variables

A *matrix* of decision variables x_{ij} appears in problem (21.1).

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 $(1, 1, \dots, 1)'$, 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

$$A \otimes B = \begin{pmatrix} a_{11}B & a_{12}B & \dots & a_{1s}B \\ a_{21}B & a_{22}B & \dots & a_{2s}B \\ & & \vdots & \\ a_{m1}B & a_{m2}B & \dots & a_{ms}B \end{pmatrix}.$$

$A \otimes B$ is an $mn \times st$ matrix.

It has the property that for any $m \times n$ matrix X

$$\text{vec}(A'XB) = (B' \otimes A')\,\text{vec}(X). \tag{21.3}$$

We can now express our constraints in terms of $\text{vec}(X)$.

Let $A = \mathbf{I}'_m, B = \mathbf{1}_n$.

By equation (21.3)

$$X\,\mathbf{1}_n = \text{vec}(X\,\mathbf{1}_n) = \text{vec}(\mathbf{I}_m X\,\mathbf{1}_n) = (\mathbf{1}'_n \otimes \mathbf{I}_m)\,\text{vec}(X).$$

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:

$$\begin{aligned} \min_z \;\; & \text{vec}(C)'z \\ \text{subject to} \;\; & Az = b \\ & z \geq 0 \end{aligned} \tag{21.4}$$

where

$$A = \begin{pmatrix} \mathbf{1}'_n \otimes \mathbf{I}_m \\ \mathbf{I}_n \otimes \mathbf{1}'_m \end{pmatrix} \quad \text{and} \quad b = \begin{pmatrix} p \\ q \end{pmatrix}$$

21.3.2 An Application

We now provide an example that takes the form (21.4) 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:

$$p = \begin{pmatrix} 50 \\ 100 \\ 150 \end{pmatrix}, \quad q = \begin{pmatrix} 25 \\ 115 \\ 60 \\ 30 \\ 70 \end{pmatrix} \quad \text{and} \quad C = \begin{pmatrix} 10 & 15 & 20 & 20 & 40 \\ 20 & 40 & 15 & 30 & 30 \\ 30 & 35 & 40 & 55 & 25 \end{pmatrix}.$$

Let's write Python code that sets up the problem and solves it.

```
# Define parameters
m = 3
n = 5

p = np.array([50, 100, 150])
```

(continues on next page)

(continued from previous page)

```
q = np.array([25, 115, 60, 30, 70])

C = np.array([[10, 15, 20, 20, 40],
              [20, 40, 15, 30, 30],
              [30, 35, 40, 55, 25]])

# 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 warning:

The above warning message from SciPy points out that A is not full rank.

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 (21.2).

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 (21.2).

One equality constraint here is redundant.

Below we 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 1.81 ms, sys: 0 ns, total: 1.81 ms
Wall time: 1.48 ms
```

```
        message: Optimization terminated successfully. (HiGHS Status 7: Optimal)
        success: True
```

(continues on next page)

(continued from previous page)

```
         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 1.5 ms, sys: 233 µs, total: 1.73 ms
Wall time: 1.48 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 = []

# simulate 1000 times
for i in range(1000):

    np.random.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:  (0.0, 10.0, 15.0, 50.0, 0.0, 65.0, 0.0, 60.0, 0.0, 0.0, 30.
↪0, 0.0, 0.0, 0.0, 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
                                    inf]
```

(continues on next page)

(continued from previous page)

```
                 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 $\text{vec}(X)$.

The minimized cost from the optimal transport plan is given by the fun variable.

21.3.3 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 this routine uses the same simplex algorithm as `scipy.optimize.linprog`, the code is accelerated by using a just-in-time compiler shipped in the `numba` library.

As you will see very soon, by using `scipy.optimize.linprog` 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)
```

Since the two LP solvers use the same simplex algorithm, we expect to get exactly the same solutions

```
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 0 ns, sys: 2 ms, total: 2 ms
Wall time: 1.67 ms
```

```
# quantecon.optimize.linprog_simplex
%time out = linprog_simplex(-c, A_eq=A_eq, b_eq=b_eq)
```

```
CPU times: user 100 µs, sys: 9 µs, total: 109 µs
Wall time: 117 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.)

21.4 The Dual Problem

Let u, v denotes vectors of dual decision variables with entries $(u_i), (v_j)$.

The **dual** to **minimization** problem (21.1) is the **maximization** problem:

$$\begin{aligned} \max_{u_i, v_j} \ & \sum_{i=1}^{m} p_i u_i + \sum_{j=1}^{n} q_j v_j \\ \text{subject to } \ & u_i + v_j \leq c_{ij}, \ i = 1, 2, \dots, m; \ j = 1, 2, \dots, n \end{aligned} \tag{21.5}$$

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

- $(\mathbf{1}_n' \otimes \mathbf{I}_m) \operatorname{vec}(X) = p$

and v is attached to constraints

- $(\mathbf{I}_n \otimes \mathbf{1}'_m)\,\mathrm{vec}(X) = q$.

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

$$\begin{aligned} \max_{u_i, v_j} \; & pu + qv \\ \text{subject to} \;\; & A' \begin{pmatrix} u \\ v \end{pmatrix} = \mathrm{vec}(C) \end{aligned} \tag{21.6}$$

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.]
```

We can also solve the dual problem using quantecon.optimize.linprog_simplex.

```
res_dual_qe = linprog_simplex(b_eq, A_ub=A_eq.T, b_ub=c)
```

And the shadow prices computed by the two programs are identical.

```
res_dual_qe.x
```

```
array([ 5., 15., 25.,  5., 10.,  0., 15.,  0.])
```

```
res_dual.x
```

```
array([-20., -10.,   0.,  30.,  35.,  25.,  40.,  25.])
```

We can compare computational times from using our two tools.

```
%time linprog(-b, A_ub=A.T, b_ub=C_vec, bounds=[(None, None)]*(m+n))
```

```
CPU times: user 2.25 ms, sys: 176 µs, total: 2.43 ms
Wall time: 1.89 ms
```

```
        message: Optimization terminated successfully. (HiGHS Status 7: Optimal)
        success: True
         status: 0
            fun: -7225.0
              x: [-2.000e+01 -1.000e+01  0.000e+00  3.000e+01  3.500e+01
                   2.500e+01  4.000e+01  2.500e+01]
            nit: 9
          lower:  residual: [       inf        inf        inf        inf
                                   inf        inf        inf        inf]
                 marginals: [ 0.000e+00  0.000e+00  0.000e+00  0.000e+00
                              0.000e+00  0.000e+00  0.000e+00  0.000e+00]
          upper:  residual: [       inf        inf        inf        inf
                                   inf        inf        inf        inf]
                 marginals: [ 0.000e+00  0.000e+00  0.000e+00  0.000e+00
                              0.000e+00  0.000e+00  0.000e+00  0.000e+00]
          eqlin:  residual: []
                 marginals: []
        ineqlin:  residual: [ 0.000e+00  0.000e+00 ...  1.500e+01
                              0.000e+00]
                 marginals: [-0.000e+00 -1.000e+01 ... -0.000e+00
                             -7.000e+01]
 mip_node_count: 0
 mip_dual_bound: 0.0
        mip_gap: 0.0
```

```
%time linprog_simplex(b_eq, A_ub=A_eq.T, b_ub=c)
```

```
CPU times: user 376 µs, sys: 29 µs, total: 405 µs
Wall time: 413 µs
```

```
SimplexResult(x=array([ 5., 15., 25.,  5., 10.,  0., 15.,  0.]), lambd=array([ 0.,␣
↪35.,  0., 15.,  0., 25.,  0., 60., 15.,  0.,  0., 80.,  0.,
        0., 70.]), fun=7225.0, success=True, status=0, num_iter=24)
```

`quantecon.optimize.linprog_simplex` solves the dual problem 10 times faster.

Just for completeness, let's solve the dual problems with nonsingular A matrices that we create by dropping a redundant equality constraint.

Try first leaving out the first constraint:

```
linprog(-b[1:], A_ub=A[1:].T, b_ub=C_vec,
        bounds=[(None, None)]*(m+n-1))
```

```
        message: Optimization terminated successfully. (HiGHS Status 7: Optimal)
        success: True
         status: 0
            fun: -7225.0
              x: [ 1.000e+01  2.000e+01  1.000e+01  1.500e+01  5.000e+00
                   2.000e+01  5.000e+00]
            nit: 12
          lower:  residual: [       inf        inf        inf        inf
                                   inf        inf        inf]
                 marginals: [ 0.000e+00  0.000e+00  0.000e+00  0.000e+00
```

(continues on next page)

(continued from previous page)

```
                                0.000e+00  0.000e+00  0.000e+00]
          upper:  residual: [       inf        inf        inf        inf
                                    inf        inf        inf]
                 marginals: [ 0.000e+00  0.000e+00  0.000e+00  0.000e+00
                                0.000e+00  0.000e+00  0.000e+00]
          eqlin:  residual: []
                 marginals: []
        ineqlin:  residual: [ 0.000e+00  0.000e+00 ...  1.500e+01
                                0.000e+00]
                 marginals: [-1.500e+01 -1.000e+01 ... -0.000e+00
                              -7.000e+01]
 mip_node_count: 0
 mip_dual_bound: 0.0
        mip_gap: 0.0
```

Not let's instead leave out the last constraint:

```
linprog(-b[:-1], A_ub=A[:-1].T, b_ub=C_vec,
        bounds=[(None, None)]*(m+n-1))
```

```
        message: Optimization terminated successfully. (HiGHS Status 7: Optimal)
        success: True
         status: 0
            fun: -7225.0
              x: [ 5.000e+00  1.500e+01  2.500e+01  5.000e+00  1.000e+01
                  -0.000e+00  1.500e+01]
            nit: 9
          lower:  residual: [       inf        inf        inf        inf
                                    inf        inf        inf]
                 marginals: [ 0.000e+00  0.000e+00  0.000e+00  0.000e+00
                                0.000e+00  0.000e+00  0.000e+00]
          upper:  residual: [       inf        inf        inf        inf
                                    inf        inf        inf]
                 marginals: [ 0.000e+00  0.000e+00  0.000e+00  0.000e+00
                                0.000e+00  0.000e+00  0.000e+00]
          eqlin:  residual: []
                 marginals: []
        ineqlin:  residual: [ 0.000e+00  0.000e+00 ...  1.500e+01
                                0.000e+00]
                 marginals: [-0.000e+00 -1.000e+01 ... -0.000e+00
                              -7.000e+01]
 mip_node_count: 0
 mip_dual_bound: 0.0
        mip_gap: 0.0
```

21.4.1 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 $u_i + v_j = c_{ij}$.

21.5 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.)

21.5.1 Replicating Previous Results

The following line of code solves the example application discussed above using linear programming.

```
X = ot.emd(p, q, C)
X
```

```
/tmp/ipykernel_3805/1617639716.py:1: UserWarning: Input histogram consists of␣
↪integer. The transport plan will be casted accordingly, possibly resulting in a␣
↪loss of precision. If this behaviour is unwanted, please make sure your input␣
↪histogram consists of floating point elements.
 X = ot.emd(p, q, C)
```

```
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.sum(X * C)
total_cost
```

```
7225
```

21.5.2 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 = []
    np.random.seed(seed)

    for i in range(n):

        if group == 'p':
            m = 1/n
            x = np.random.uniform(-2, 2)
            y = np.random.uniform(-2, 2)
        else:
            m = betabinom.pmf(i, n-1, 2, 2)
            x = 0.6 * np.random.uniform(-1.5, 1.5)
            y = 0.6 * np.random.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
```

(continues on next page)

(continued from previous page)

```
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 713 µs, sys: 55 µs, total: 768 µs
Wall time: 514 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)
```

(continues on next page)

(continued from previous page)

```
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',
                       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

TWENTYTWO

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 [vN37] that was generalized by Kemeny, Morgenstern and Thompson [KMT56].

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(object):

    """
    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
```

(continues on next page)

(continued from previous page)

```
        (x', v) >= (0', -inf)

    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 - γ * 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
```

(continues on next page)

(continued from previous page)

```
            res = linprog(c, A_ub=A_iq, b_ub=b_iq, A_eq=A_eq, b_eq=b_eq,
                          bounds=bounds)

        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
```

(continues on next page)

(continued from previous page)

```
        Outputs:
        --------
        beta: scalar
            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
```

22.1 Notation

We use the following notation.

$\mathbf{0}$ 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 and 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^T denotes the transpose of x (i.e., a row vector).

Let ι_n denote a column vector composed of n ones, i.e. $\iota_n = (1, 1, \dots, 1)^T$.

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.

22.2 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 I:** (every good that is consumed is also produced)

$$b_{.,j} > \mathbf{0} \quad \forall j = 1, 2, \dots, n$$

- **Assumption II:** (no free lunch)

$$a_{i,.} > \mathbf{0} \quad \forall i = 1, 2, \dots, m$$

A semi-positive *intensity* m-vector x denotes levels at which activities are operated.

Therefore,

- vector $x^T A$ gives the total amount of *goods used in production*
- vector $x^T B$ 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 $x^T B > x^T A$.

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 or 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: For an economy (A, B), the set of goods $S \subset \{1, 2, \dots, n\}$ 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 $\exists T \subset \{1, 2, \dots, m\}$ (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 [Gal89]

```
# (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],
```

(continues on next page)

(continued from previous page)

```
                [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
```

22.3 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 $x_t^T A$ in period t results in $x_t^T B$ amounts of output in period $t+1$.

These timing conventions imply the following feasibility condition:

$$x_t^T B \geq x_{t+1}^T A \qquad \forall t \geq 1$$

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 $t+1$.

22.3.1 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

$$x_t^T B \geq \alpha x_t^T A \qquad \forall t$$

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.

22.4 Duality

Two problems are connected by a remarkable dual relationship between technological and valuation characteristics of the economy:

Definition: 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

$$\begin{aligned} \max_{\alpha} \quad & \alpha \\ \text{s.t.} \quad & x^T B \geq \alpha x^T A \end{aligned}$$

Theorem 9.3 of David Gale's book [Gal89] asserts that if Assumptions I and II 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: 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

$$\begin{aligned} \min_{\beta} \quad & \beta \\ \text{s.t.} \quad & Bp \leq \beta Ap \end{aligned}$$

Assumptions I and II 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) [Gal89]) implies that under Assumptions I and II, $\beta_0 \leq \alpha_0$.

But to deduce that $\beta_0 \geq \alpha_0$, Assumptions I and II are not sufficient.

Therefore, von Neumann [vN37] went on to prove the following remarkable "duality" result that connects TEP and EEP.

Theorem 1 (von Neumann): If the economy (A, B) satisfies Assumptions I and II, 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 true

$$\begin{aligned} x_0^T B &\geq \gamma^* x_0^T A \\ Bp_0 &\leq \gamma^* Ap_0 \\ x_0^T \left(B - \gamma^* A\right) p_0 &= 0 \end{aligned}$$

Note: *Proof (Sketch):* Assumption I and II 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 $x^T B \geq \gamma^* x^T A$. 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 $x_0^T B \geq \alpha_0 x_0^T A \geq \gamma^* x_0^T A$. 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 $x_0^T \left(B - \gamma^* A\right) p_0 = 0$ 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 I ex encode all equilibrium conditions.

So Theorem I essentially states that under Assumptions I and II there always exists an equilibrium (γ^*, x_0, p_0) with balanced growth.

Note that Theorem I is silent about uniqueness of the equilibrium. In fact, it does not rule out (trivial) cases with $x_0^T Bp_0 = 0$ so that nothing of value is produced.

To exclude such uninteresting cases, Kemeny, Morgenstern and Thomspson [KMT56] add an extra requirement

$$x_0^T Bp_0 > 0$$

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.

22.5 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 solves 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 $\iota_m^T x = 1$
- the column player chooses the n-vector $p > \mathbf{0}$ subject to $\iota_n^T p = 1$.

Definition: The $m \times n$ matrix game C has the *solution* $(x^*, p^*, V(C))$ in mixed strategies if

$$(x^*)^T C e^j \geq V(C) \quad \forall j \in \{1, \dots, n\} \qquad \text{and} \qquad (e^i)^T C p^* \leq V(C) \quad \forall i \in \{1, \dots, m\}$$

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 stategy, the maximizing player can assure himself at least $V(C)$ (no matter what the column player chooses)
- by playing the appropriate mixed stategy, 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 [vN28] implies that

$$V(C) = \max_x \min_p \quad x^T C p = \min_p \max_x \quad x^T C p = (x^*)^T C p^*$$

22.5.1 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: $v \equiv \min_p x^T C p = \min_j x^T C e^j$
- For a fixed p, let u be the value of the maximization problem: $u \equiv \max_x x^T C p = \max_i (e^i)^T C p$

Then the *max-min problem* (the game from the maximizing player's point of view) can be written as the *primal* LP

$$\begin{aligned} V(C) = & \max \quad v \\ \text{s.t.} \quad & v \iota_n^T \leq x^T C \\ & x \geq \mathbf{0} \\ & \iota_n^T x = 1 \end{aligned}$$

while the *min-max problem* (the game from the minimizing player's point of view) is the *dual* LP

$$\begin{aligned} V(C) = & \min \quad u \\ \text{s.t.} \quad & u \iota_m \geq C p \\ & p \geq \mathbf{0} \\ & \iota_m^T p = 1 \end{aligned}$$

Hamburger, Thompson and Weil [HTW67] view the input-output pair of the economy as payoff matrices of two-player zero-sum games.

Using this interpretation, they restate Assumption I and II as follows

$$V(-A) < 0 \qquad \text{and} \qquad V(B) > 0$$

Note: *Proof (Sketch)*:

- $\Rightarrow V(B) > 0$ implies $x_0^T B \gg \mathbf{0}$, 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 I.
- $\Leftarrow$ From Assumption I 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 $x^T Bp > 0$ so that it must be the case that $V(B) > 0$.

In order to (re)state Theorem I 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. $[x^T M(\gamma)]_j < 0$ 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 I) the optimal intensity and price vectors x_0 and p_0 satisfy

$$x_0^T M(\gamma) \geq \mathbf{0}^T \qquad \text{and} \qquad M(\gamma)p_0 \leq \mathbf{0}$$

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 $(x')^T M(\gamma') \geq \mathbf{0}^T$ and so for all $\gamma < \gamma'$.

$$(x')^T M(\gamma) = (x')^T M(\gamma') + (x')^T (\gamma' - \gamma) A \geq \mathbf{0}^T$$

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. [HTW67] 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).

22.5.2 Algorithm

Hamburger, Thompson and Weil [HTW67] 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 $x^T(B - \alpha A) \geq \mathbf{0}^T$ 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 $\min_j\{[\iota_m^T(B - \beta A)]_j\} > 0$, then the EEP has no solution and so we can define **LB** as the β^* that solves $\min_j\{[\iota_m^T(B - \beta^* A)]_j\} = 0$.

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.

```
γ = 2

print(f'Value of the game with γ = {γ}')
print(n1.zerosum(γ=γ)[0])
print('Intensity vector (from the primal)')
print(n1.zerosum(γ=γ)[1])
print('Price vector (from the dual)')
print(n1.zerosum(γ=γ, 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='$\gamma$')
    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]
```

(continues on next page)

(continued from previous page)

```
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.

22.5.3 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, Assumptions I and II 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 $x_0^T B p_0 > 0$ 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 [Gal89]) asserts that imposing irreducibility is sufficient for uniqueness of (γ^*, x_0, p_0).

Theorem II: Adopt the conditions of Theorem 1. If the economy (A, B) is irreducible, then $\gamma^* = \alpha_0 = \beta_0$.

22.5.4 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: 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 [Gal89]): if x_0 and $\alpha_0 > 0$ solve the TEP with (A, I_n), then

$$x_0^T = \alpha_0 x_0^T A \qquad \Leftrightarrow \qquad x_0^T A = \left(\frac{1}{\alpha_0}\right) x_0^T$$

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 $S_1, \dots, S_k$. 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 IV

Introduction to Dynamics

CHAPTER

TWENTYTHREE

DYNAMICS IN ONE DIMENSION

Contents

- *Dynamics in One Dimension*
  - *Overview*
  - *Some Definitions*
  - *Graphical Analysis*
  - *Exercises*

23.1 Overview

In this lecture we give a quick introduction to discrete time dynamics in one dimension.

In one-dimensional models, the state of the system is described by a single variable.

Although most interesting dynamic models have two or more state variables, the one-dimensional setting is a good place to learn the foundations of dynamics and build intuition.

Let's start with some standard imports:

```
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
```

23.2 Some Definitions

This section sets out the objects of interest and the kinds of properties we study.

23.2.1 Difference Equations

A **time homogeneous first order difference equation** is an equation of the form

$$x_{t+1} = g(x_t) \tag{23.1}$$

where g is a function from some subset S of $\mathbb{R}$ to itself.

Here S is called the **state space** and x is called the **state variable**.

In the definition,

- time homogeneity means that g is the same at each time t
- first order means dependence on only one lag (i.e., earlier states such as x_{t-1} do not enter into (23.1)).

If $x_0 \in S$ is given, then (23.1) recursively defines the sequence

$$x_0, \quad x_1 = g(x_0), \quad x_2 = g(x_1) = g(g(x_0)), \quad \text{etc.} \tag{23.2}$$

This sequence is called the **trajectory** of x_0 under g.

If we define g^n to be n compositions of g with itself, then we can write the trajectory more simply as $x_t = g^t(x_0)$ for $t \geq 0$.

23.2.2 Example: A Linear Model

One simple example is the **linear difference equation**

$$x_{t+1} = ax_t + b, \qquad S = \mathbb{R}$$

where a, b are fixed constants.

In this case, given x_0, the trajectory (23.2) is

$$x_0, \quad ax_0 + b, \quad a^2 x_0 + ab + b, \quad \text{etc.} \tag{23.3}$$

Continuing in this way, and using our knowledge of *geometric series*, we find that, for any $t \geq 0$,

$$x_t = a^t x_0 + b\frac{1 - a^t}{1 - a} \tag{23.4}$$

This is about all we need to know about the linear model.

We have an exact expression for x_t for all t and hence a full understanding of the dynamics.

Notice in particular that $|a| < 1$, then, by (23.4), we have

$$x_t \to \frac{b}{1 - a} \text{ as } t \to \infty \tag{23.5}$$

regardless of x_0

This is an example of what is called global stability, a topic we return to below.

23.2.3 Example: A Nonlinear Model

In the linear example above, we obtained an exact analytical expression for x_t in terms of arbitrary t and x_0.

This made analysis of dynamics very easy.

When models are nonlinear, however, the situation can be quite different.

For example, recall how we previously studied the law of motion for the Solow growth model, a simplified version of which is

$$k_{t+1} = szk_t^\alpha + (1-\delta)k_t \tag{23.6}$$

Here k is capital stock and s, z, α, δ are positive parameters with $0 < \alpha, \delta < 1$.

If you try to iterate like we did in (23.3), you will find that the algebra gets messy quickly.

Analyzing the dynamics of this model requires a different method (see below).

23.2.4 Stability

A **steady state** of the difference equation $x_{t+1} = g(x_t)$ is a point x^* in S such that $x^* = g(x^*)$.

In other words, x^* is a **fixed point** of the function g in S.

For example, for the linear model $x_{t+1} = ax_t + b$, you can use the definition to check that

- $x^* := b/(1-a)$ is a steady state whenever $a \neq 1$.
- if $a = 1$ and $b = 0$, then every $x \in \mathbb{R}$ is a steady state.
- if $a = 1$ and $b \neq 0$, then the linear model has no steady state in $\mathbb{R}$.

A steady state x^* of $x_{t+1} = g(x_t)$ is called **globally stable** if, for all $x_0 \in S$,

$$x_t = g^t(x_0) \to x^* \text{ as } t \to \infty$$

For example, in the linear model $x_{t+1} = ax_t + b$ with $a \neq 1$, the steady state x^*

- is globally stable if $|a| < 1$ and
- fails to be globally stable otherwise.

This follows directly from (23.4).

A steady state x^* of $x_{t+1} = g(x_t)$ is called **locally stable** if there exists an $\epsilon > 0$ such that

$$|x_0 - x^*| < \epsilon \implies x_t = g^t(x_0) \to x^* \text{ as } t \to \infty$$

Obviously every globally stable steady state is also locally stable.

We will see examples below where the converse is not true.

23.3 Graphical Analysis

As we saw above, analyzing the dynamics for nonlinear models is nontrivial.

There is no single way to tackle all nonlinear models.

However, there is one technique for one-dimensional models that provides a great deal of intuition.

This is a graphical approach based on **45 degree diagrams**.

Let's look at an example: the Solow model with dynamics given in (23.6).

We begin with some plotting code that you can ignore at first reading.

The function of the code is to produce 45 degree diagrams and time series plots.

```
def subplots(fs):
    "Custom subplots with axes throught the origin"
    fig, ax = plt.subplots(figsize=fs)

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

    return fig, ax


def plot45(g, xmin, xmax, x0, num_arrows=6, var='x'):

    xgrid = np.linspace(xmin, xmax, 200)

    fig, ax = subplots((6.5, 6))
    ax.set_xlim(xmin, xmax)
    ax.set_ylim(xmin, xmax)

    hw = (xmax - xmin) * 0.01
    hl = 2 * hw
    arrow_args = dict(fc="k", ec="k", head_width=hw,
            length_includes_head=True, lw=1,
            alpha=0.6, head_length=hl)

    ax.plot(xgrid, g(xgrid), 'b-', lw=2, alpha=0.6, label='g')
    ax.plot(xgrid, xgrid, 'k-', lw=1, alpha=0.7, label='45')

    x = x0
    xticks = [xmin]
    xtick_labels = [xmin]

    for i in range(num_arrows):
        if i == 0:
            ax.arrow(x, 0.0, 0.0, g(x), **arrow_args) # x, y, dx, dy
        else:
            ax.arrow(x, x, 0.0, g(x) - x, **arrow_args)
            ax.plot((x, x), (0, x), 'k', ls='dotted')

        ax.arrow(x, g(x), g(x) - x, 0, **arrow_args)
```

(continues on next page)

(continued from previous page)

```
        xticks.append(x)
        xtick_labels.append(r'${}_{}$'.format(var, str(i)))

        x = g(x)
        xticks.append(x)
        xtick_labels.append(r'${}_{}$'.format(var, str(i+1)))
        ax.plot((x, x), (0, x), 'k', ls='dotted')

    xticks.append(xmax)
    xtick_labels.append(xmax)
    ax.set_xticks(xticks)
    ax.set_yticks(xticks)
    ax.set_xticklabels(xtick_labels)
    ax.set_yticklabels(xtick_labels)

    bbox = (0., 1.04, 1., .104)
    legend_args = {'bbox_to_anchor': bbox, 'loc': 'upper right'}

    ax.legend(ncol=2, frameon=False, **legend_args, fontsize=14)
    plt.show()

def ts_plot(g, xmin, xmax, x0, ts_length=6, var='x'):
    fig, ax = subplots((7, 5.5))
    ax.set_ylim(xmin, xmax)
    ax.set_xlabel(r'$t$', fontsize=14)
    ax.set_ylabel(r'${}_t$'.format(var), fontsize=14)
    x = np.empty(ts_length)
    x[0] = x0
    for t in range(ts_length-1):
        x[t+1] = g(x[t])
    ax.plot(range(ts_length),
            x,
            'bo-',
            alpha=0.6,
            lw=2,
            label=r'${}_t$'.format(var))
    ax.legend(loc='best', fontsize=14)
    ax.set_xticks(range(ts_length))
    plt.show()
```

Let's create a 45 degree diagram for the Solow model with a fixed set of parameters

```
A, s, alpha, delta = 2, 0.3, 0.3, 0.4
```

Here's the update function corresponding to the model.

```
def g(k):
    return A * s * k**alpha + (1 - delta) * k
```

Here is the 45 degree plot.

```
xmin, xmax = 0, 4  # Suitable plotting region.

plot45(g, xmin, xmax, 0, num_arrows=0)
```

The plot shows the function g and the 45 degree line.

Think of k_t as a value on the horizontal axis.

To calculate k_{t+1}, we can use the graph of g to see its value on the vertical axis.

Clearly,

- If g lies above the 45 degree line at this point, then we have $k_{t+1} > k_t$.
- If g lies below the 45 degree line at this point, then we have $k_{t+1} < k_t$.
- If g hits the 45 degree line at this point, then we have $k_{t+1} = k_t$, so k_t is a steady state.

For the Solow model, there are two steady states when $S = \mathbb{R}_+ = [0, \infty)$.

- the origin $k = 0$
- the unique positive number such that $k = szk^\alpha + (1-\delta)k$.

By using some algebra, we can show that in the second case, the steady state is

$$k^* = \left(\frac{sz}{\delta}\right)^{1/(1-\alpha)}$$

23.3.1 Trajectories

By the preceding discussion, in regions where g lies above the 45 degree line, we know that the trajectory is increasing.

The next figure traces out a trajectory in such a region so we can see this more clearly.

The initial condition is $k_0 = 0.25$.

```
k0 = 0.25

plot45(g, xmin, xmax, k0, num_arrows=5, var='k')
```

We can plot the time series of capital corresponding to the figure above as follows:

```
ts_plot(g, xmin, xmax, k0, var='k')
```

Here's a somewhat longer view:

```
ts_plot(g, xmin, xmax, k0, ts_length=20, var='k')
```

When capital stock is higher than the unique positive steady state, we see that it declines:

```
k0 = 2.95

plot45(g, xmin, xmax, k0, num_arrows=5, var='k')
```

Here is the time series:

```
ts_plot(g, xmin, xmax, k0, var='k')
```

23.3.2 Complex Dynamics

The Solow model is nonlinear but still generates very regular dynamics.

One model that generates irregular dynamics is the **quadratic map**

$$g(x) = 4x(1-x), \qquad x \in [0,1]$$

Let's have a look at the 45 degree diagram.

```
xmin, xmax = 0, 1
g = lambda x: 4 * x * (1 - x)

x0 = 0.3
plot45(g, xmin, xmax, x0, num_arrows=0)
```

Now let's look at a typical trajectory.

```
plot45(g, xmin, xmax, x0, num_arrows=6)
```

Notice how irregular it is.

Here is the corresponding time series plot.

```
ts_plot(g, xmin, xmax, x0, ts_length=6)
```

The irregularity is even clearer over a longer time horizon:

```
ts_plot(g, xmin, xmax, x0, ts_length=20)
```

23.4 Exercises

Exercise 23.4.1

Consider again the linear model $x_{t+1} = ax_t + b$ with $a \neq 1$.

The unique steady state is $b/(1-a)$.

The steady state is globally stable if $|a| < 1$.

Try to illustrate this graphically by looking at a range of initial conditions.

What differences do you notice in the cases $a \in (-1, 0)$ and $a \in (0, 1)$?

Use $a = 0.5$ and then $a = -0.5$ and study the trajectories

Set $b = 1$ throughout.

Solution to Exercise 23.4.1

We will start with the case $a = 0.5$.

Let's set up the model and plotting region:

```
a, b = 0.5, 1
xmin, xmax = -1, 3
g = lambda x: a * x + b
```

Now let's plot a trajectory:

```
x0 = -0.5
plot45(g, xmin, xmax, x0, num_arrows=5)
```

Here is the corresponding time series, which converges towards the steady state.

```
ts_plot(g, xmin, xmax, x0, ts_length=10)
```

Now let's try $a = -0.5$ and see what differences we observe.

Let's set up the model and plotting region:

```
a, b = -0.5, 1
xmin, xmax = -1, 3
g = lambda x: a * x + b
```

Now let's plot a trajectory:

```
x0 = -0.5
plot45(g, xmin, xmax, x0, num_arrows=5)
```

Here is the corresponding time series, which converges towards the steady state.

```
ts_plot(g, xmin, xmax, x0, ts_length=10)
```

Once again, we have convergence to the steady state but the nature of convergence differs.

In particular, the time series jumps from above the steady state to below it and back again.

In the current context, the series is said to exhibit **damped oscillations**.

CHAPTER

TWENTYFOUR

AR1 PROCESSES

Contents

- *AR1 Processes*
  - *Overview*
  - *The AR(1) Model*
  - *Stationarity and Asymptotic Stability*
  - *Ergodicity*
  - *Exercises*

24.1 Overview

In this lecture we are going to study a very simple class of stochastic models called AR(1) processes.

These simple models are used again and again in economic research to represent the dynamics of series such as

- labor income
- dividends
- productivity, etc.

AR(1) processes can take negative values but are easily converted into positive processes when necessary by a transformation such as exponentiation.

We are going to study AR(1) processes partly because they are useful and partly because they help us understand important concepts.

Let's start with some imports:

```
import numpy as np
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
```

24.2 The AR(1) Model

The **AR(1) model** (autoregressive model of order 1) takes the form

$$X_{t+1} = aX_t + b + cW_{t+1} \tag{24.1}$$

where a, b, c are scalar-valued parameters.

This law of motion generates a time series $\{X_t\}$ as soon as we specify an initial condition X_0.

This is called the **state process** and the state space is $\mathbb{R}$.

To make things even simpler, we will assume that

- the process $\{W_t\}$ is IID and standard normal,
- the initial condition X_0 is drawn from the normal distribution $N(\mu_0, v_0)$ and
- the initial condition X_0 is independent of $\{W_t\}$.

24.2.1 Moving Average Representation

Iterating backwards from time t, we obtain

$$X_t = aX_{t-1} + b + cW_t = a^2 X_{t-2} + ab + acW_{t-1} + b + cW_t = \cdots$$

If we work all the way back to time zero, we get

$$X_t = a^t X_0 + b \sum_{j=0}^{t-1} a^j + c \sum_{j=0}^{t-1} a^j W_{t-j} \tag{24.2}$$

Equation (24.2) shows that X_t is a well defined random variable, the value of which depends on

- the parameters,
- the initial condition X_0 and
- the shocks $W_1, \ldots W_t$ from time $t = 1$ to the present.

Throughout, the symbol ψ_t will be used to refer to the density of this random variable X_t.

24.2.2 Distribution Dynamics

One of the nice things about this model is that it's so easy to trace out the sequence of distributions $\{\psi_t\}$ corresponding to the time series $\{X_t\}$.

To see this, we first note that X_t is normally distributed for each t.

This is immediate from (24.2), since linear combinations of independent normal random variables are normal.

Given that X_t is normally distributed, we will know the full distribution ψ_t if we can pin down its first two moments.

Let μ_t and v_t denote the mean and variance of X_t respectively.

We can pin down these values from (24.2) or we can use the following recursive expressions:

$$\mu_{t+1} = a\mu_t + b \quad \text{and} \quad v_{t+1} = a^2 v_t + c^2 \tag{24.3}$$

These expressions are obtained from (24.1) by taking, respectively, the expectation and variance of both sides of the equality.

In calculating the second expression, we are using the fact that X_t and W_{t+1} are independent.

(This follows from our assumptions and (24.2).)

Given the dynamics in (24.2) and initial conditions μ_0, v_0, we obtain μ_t, v_t and hence

$$\psi_t = N(\mu_t, v_t)$$

The following code uses these facts to track the sequence of marginal distributions $\{\psi_t\}$.

The parameters are

```
a, b, c = 0.9, 0.1, 0.5

mu, v = -3.0, 0.6  # initial conditions mu_0, v_0
```

Here's the sequence of distributions:

```
from scipy.stats import norm

sim_length = 10
grid = np.linspace(-5, 7, 120)

fig, ax = plt.subplots()

for t in range(sim_length):
    mu = a * mu + b
    v = a**2 * v + c**2
    ax.plot(grid, norm.pdf(grid, loc=mu, scale=np.sqrt(v)),
            label=f"$\psi_{t}$",
            alpha=0.7)

ax.legend(bbox_to_anchor=[1.05,1],loc=2,borderaxespad=1)

plt.show()
```

24.3 Stationarity and Asymptotic Stability

Notice that, in the figure above, the sequence $\{\psi_t\}$ seems to be converging to a limiting distribution.

This is even clearer if we project forward further into the future:

```
def plot_density_seq(ax, mu_0=-3.0, v_0=0.6, sim_length=60):
    mu, v = mu_0, v_0
    for t in range(sim_length):
        mu = a * mu + b
        v = a**2 * v + c**2
        ax.plot(grid,
                norm.pdf(grid, loc=mu, scale=np.sqrt(v)),
                alpha=0.5)

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

Moreover, the limit does not depend on the initial condition.

For example, this alternative density sequence also converges to the same limit.

```
fig, ax = plt.subplots()
plot_density_seq(ax, mu_0=3.0)
plt.show()
```

In fact it's easy to show that such convergence will occur, regardless of the initial condition, whenever $|a| < 1$.

To see this, we just have to look at the dynamics of the first two moments, as given in (24.3).

When $|a| < 1$, these sequences converge to the respective limits

$$\mu^* := \frac{b}{1-a} \quad \text{and} \quad v^* = \frac{c^2}{1-a^2} \tag{24.4}$$

(See our *lecture on one dimensional dynamics* for background on deterministic convergence.)

Hence

$$\psi_t \to \psi^* = N(\mu^*, v^*) \quad \text{as } t \to \infty \tag{24.5}$$

We can confirm this is valid for the sequence above using the following code.

```
fig, ax = plt.subplots()
plot_density_seq(ax, mu_0=3.0)

mu_star = b / (1 - a)
std_star = np.sqrt(c**2 / (1 - a**2))  # square root of v_star
psi_star = norm.pdf(grid, loc=mu_star, scale=std_star)
ax.plot(grid, psi_star, 'k-', lw=2, label="$\psi^*$")
ax.legend()

plt.show()
```

As claimed, the sequence $\{\psi_t\}$ converges to ψ^*.

24.3.1 Stationary Distributions

A stationary distribution is a distribution that is a fixed point of the update rule for distributions.

In other words, if ψ_t is stationary, then $\psi_{t+j} = \psi_t$ for all j in $\mathbb{N}$.

A different way to put this, specialized to the current setting, is as follows: a density ψ on $\mathbb{R}$ is **stationary** for the AR(1) process if

$$X_t \sim \psi \quad \Longrightarrow \quad aX_t + b + cW_{t+1} \sim \psi$$

The distribution ψ^* in (24.5) has this property — checking this is an exercise.

(Of course, we are assuming that $|a| < 1$ so that ψ^* is well defined.)

In fact, it can be shown that no other distribution on $\mathbb{R}$ has this property.

Thus, when $|a| < 1$, the AR(1) model has exactly one stationary density and that density is given by ψ^*.

24.4 Ergodicity

The concept of ergodicity is used in different ways by different authors.

One way to understand it in the present setting is that a version of the Law of Large Numbers is valid for $\{X_t\}$, even though it is not IID.

In particular, averages over time series converge to expectations under the stationary distribution.

Indeed, it can be proved that, whenever $|a| < 1$, we have

$$\frac{1}{m}\sum_{t=1}^{m} h(X_t) \to \int h(x)\psi^*(x)dx \quad \text{as } m \to \infty \tag{24.6}$$

whenever the integral on the right hand side is finite and well defined.

Notes:

- In (24.6), convergence holds with probability one.
- The textbook by [MT09] is a classic reference on ergodicity.

For example, if we consider the identity function $h(x) = x$, we get

$$\frac{1}{m}\sum_{t=1}^{m} X_t \to \int x\psi^*(x)dx \quad \text{as } m \to \infty$$

In other words, the time series sample mean converges to the mean of the stationary distribution.

As will become clear over the next few lectures, ergodicity is a very important concept for statistics and simulation.

24.5 Exercises

Exercise 24.5.1

Let k be a natural number.

The k-th central moment of a random variable is defined as

$$M_k := \mathbb{E}[(X - \mathbb{E}X)^k]$$

When that random variable is $N(\mu, \sigma^2)$, it is known that

$$M_k = \begin{cases} 0 & \text{if } k \text{ is odd} \\ \sigma^k (k-1)!! & \text{if } k \text{ is even} \end{cases}$$

Here $n!!$ is the double factorial.

According to (24.6), we should have, for any $k \in \mathbb{N}$,

$$\frac{1}{m}\sum_{t=1}^{m}(X_t - \mu^*)^k \approx M_k$$

when m is large.

Confirm this by simulation at a range of k using the default parameters from the lecture.

Solution to Exercise 24.5.1

Here is one solution:

```
from numba import njit
from scipy.special import factorial2

@njit
def sample_moments_ar1(k, m=100_000, mu_0=0.0, sigma_0=1.0, seed=1234):
    np.random.seed(seed)
    sample_sum = 0.0
    x = mu_0 + sigma_0 * np.random.randn()
    for t in range(m):
        sample_sum += (x - mu_star)**k
        x = a * x + b + c * np.random.randn()
```

(continues on next page)

(continued from previous page)

```
    return sample_sum / m

def true_moments_ar1(k):
    if k % 2 == 0:
        return std_star**k * factorial2(k - 1)
    else:
        return 0

k_vals = np.arange(6) + 1
sample_moments = np.empty_like(k_vals)
true_moments = np.empty_like(k_vals)

for k_idx, k in enumerate(k_vals):
    sample_moments[k_idx] = sample_moments_ar1(k)
    true_moments[k_idx] = true_moments_ar1(k)

fig, ax = plt.subplots()
ax.plot(k_vals, true_moments, label="true moments")
ax.plot(k_vals, sample_moments, label="sample moments")
ax.legend()

plt.show()
```

Exercise 24.5.2

Write your own version of a one dimensional kernel density estimator, which estimates a density from a sample.

Write it as a class that takes the data X and bandwidth h when initialized and provides a method f such that

$$f(x) = \frac{1}{hn} \sum_{i=1}^{n} K\left(\frac{x - X_i}{h}\right)$$

For K use the Gaussian kernel (K is the standard normal density).

Write the class so that the bandwidth defaults to Silverman's rule (see the "rule of thumb" discussion on this page). Test the class you have written by going through the steps

1. simulate data $X_1, \ldots, X_n$ from distribution ϕ
2. plot the kernel density estimate over a suitable range
3. plot the density of ϕ on the same figure

for distributions ϕ of the following types

- beta distribution with $\alpha = \beta = 2$
- beta distribution with $\alpha = 2$ and $\beta = 5$
- beta distribution with $\alpha = \beta = 0.5$

Use $n = 500$.

Make a comment on your results. (Do you think this is a good estimator of these distributions?)

Solution to Exercise 24.5.2

Here is one solution:

```
K = norm.pdf

class KDE:

    def __init__(self, x_data, h=None):

        if h is None:
            c = x_data.std()
            n = len(x_data)
            h = 1.06 * c * n**(-1/5)
        self.h = h
        self.x_data = x_data

    def f(self, x):
        if np.isscalar(x):
            return K((x - self.x_data) / self.h).mean() * (1/self.h)
        else:
            y = np.empty_like(x)
            for i, x_val in enumerate(x):
                y[i] = K((x_val - self.x_data) / self.h).mean() * (1/self.h)
            return y
```

```
def plot_kde(ϕ, x_min=-0.2, x_max=1.2):
    x_data = ϕ.rvs(n)
    kde = KDE(x_data)

    x_grid = np.linspace(-0.2, 1.2, 100)
    fig, ax = plt.subplots()
    ax.plot(x_grid, kde.f(x_grid), label="estimate")
    ax.plot(x_grid, ϕ.pdf(x_grid), label="true density")
    ax.legend()
    plt.show()
```

```
from scipy.stats import beta

n = 500
```

(continues on next page)

(continued from previous page)

```
parameter_pairs= (2, 2), (2, 5), (0.5, 0.5)
for α, β in parameter_pairs:
    plot_kde(beta(α, β))
```

We see that the kernel density estimator is effective when the underlying distribution is smooth but less so otherwise.

Exercise 24.5.3

In the lecture we discussed the following fact: for the $AR(1)$ process

$$X_{t+1} = aX_t + b + cW_{t+1}$$

with $\{W_t\}$ iid and standard normal,

$$\psi_t = N(\mu, s^2) \implies \psi_{t+1} = N(a\mu + b, a^2 s^2 + c^2)$$

Confirm this, at least approximately, by simulation. Let

- $a = 0.9$
- $b = 0.0$
- $c = 0.1$
- $\mu = -3$
- $s = 0.2$

First, plot ψ_t and ψ_{t+1} using the true distributions described above.

Second, plot ψ_{t+1} on the same figure (in a different color) as follows:

1. Generate n draws of X_t from the $N(\mu, s^2)$ distribution
2. Update them all using the rule $X_{t+1} = aX_t + b + cW_{t+1}$
3. Use the resulting sample of X_{t+1} values to produce a density estimate via kernel density estimation.

Try this for $n = 2000$ and confirm that the simulation based estimate of ψ_{t+1} does converge to the theoretical distribution.

Solution to Exercise 24.5.3

Here is our solution

```
a = 0.9
b = 0.0
c = 0.1
μ = -3
s = 0.2
```

```
μ_next = a * μ + b
s_next = np.sqrt(a**2 * s**2 + c**2)
```

```
ψ = lambda x: K((x - μ) / s)
ψ_next = lambda x: K((x - μ_next) / s_next)
```

```
ψ = norm(μ, s)
ψ_next = norm(μ_next, s_next)
```

```
n = 2000
x_draws = ψ.rvs(n)
x_draws_next = a * x_draws + b + c * np.random.randn(n)
kde = KDE(x_draws_next)

x_grid = np.linspace(μ - 1, μ + 1, 100)
fig, ax = plt.subplots()

ax.plot(x_grid, ψ.pdf(x_grid), label="$\psi_t$")
ax.plot(x_grid, ψ_next.pdf(x_grid), label="$\psi_{t+1}$")
ax.plot(x_grid, kde.f(x_grid), label="estimate of $\psi_{t+1}$")

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

The simulated distribution approximately coincides with the theoretical distribution, as predicted.

CHAPTER

TWENTYFIVE

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
```

25.1 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
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import quantecon as qe
import numpy as np
from mpl_toolkits.mplot3d import Axes3D
```

25.2 Definitions

The following concepts are fundamental.

25.2.1 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 $k \in \mathbb{N}$.

25.2.2 Markov Chains

There is a close connection between stochastic matrices and Markov chains.

To begin, let S be a finite set with n elements $\{x_1, \dots, x_n\}$.

The set S is called the **state space** and $x_1, \dots, x_n$ 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 $y \in S$,

$$\mathbb{P}\{X_{t+1} = y \,|\, X_t\} = \mathbb{P}\{X_{t+1} = y \,|\, X_t, X_{t-1}, \dots\} \tag{25.1}$$

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

$$P(x, y) := \mathbb{P}\{X_{t+1} = y \,|\, X_t = x\} \qquad (x, y \in S) \tag{25.2}$$

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:

[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.

- draw X_0 from a marginal distribution ψ
- for each $t = 0, 1, ...$, draw X_{t+1} from $P(X_t, \cdot)$

By construction, the resulting process satisfies (25.2).

25.2.3 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

$$P = \begin{pmatrix} 1 - \alpha & \alpha \\ \beta & 1 - \beta \end{pmatrix} \tag{25.3}$$

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.

25.2.4 Example 2

From US unemployment data, Hamilton [Ham05] 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.

25.3 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 $S = 0, \dots, n-1$.

25.3.1 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([1, 1, 1, 1, 0])
```

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
    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)
```

```
0.24843
```

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.

25.3.2 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)
```

```
0.250722
```

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 1.25 s, sys: 0 ns, total: 1.25 s
Wall time: 1.25 s
```

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

```
%time mc.simulate(ts_length=1_000_000) # qe code version
```

```
CPU times: user 14.4 ms, sys: 9.77 ms, total: 24.2 ms
Wall time: 23.8 ms
```

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

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', 'employed', 'unemployed'], dtype='<U10')
```

```
mc.simulate(ts_length=4, init='unemployed')
```

```
array(['unemployed', 'employed', 'unemployed', 'unemployed'], dtype='<U10')
```

```
mc.simulate(ts_length=4)  # Start at randomly chosen initial state
```

```
array(['unemployed', 'unemployed', 'unemployed', 'unemployed'],
      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([0, 1, 1, 0])
```

25.4 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 $t = 0, 1, 2, \ldots$.

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

$$\psi_{t+1} = \psi_t P \tag{25.4}$$

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 (25.4), 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

$$X_0 \sim \psi_0 \quad \implies \quad X_m \sim \psi_0 P^m \tag{25.5}$$

and, more generally,

$$X_t \sim \psi_t \quad \implies \quad X_{t+m} \sim \psi_t P^m \tag{25.6}$$

25.4.1 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 (25.6), 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 (25.6), we see that, conditional on $X_t = x$, the distribution of X_{t+m} is the x-th row of P^m.

In particular

$$\mathbb{P}\{X_{t+m} = y \mid X_t = x\} = P^m(x, y) = (x, y)\text{-th element of } P^m$$

25.4.2 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

$$\psi P^6 \cdot \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}$$

25.4.3 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 (25.3).

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.

25.5 Irreducibility and Aperiodicity

Irreducibility and aperiodicity are central concepts of modern Markov chain theory.

Let's see what they're about.

25.5.1 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.

25.5.2 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
```

25.6 Stationary Distributions

As seen in (25.4), 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]])
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 [Haggstrom02].

(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.

25.6.1 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 $\alpha \to 0$.

25.6.2 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

$$\psi(I_n - P) = 0 \tag{25.7}$$

for ψ, where I_n is the $n \times n$ identity.

But the zero vector solves system (25.7), 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 (25.7) 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]])
```

25.6.3 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 $t = 1, 2, \ldots$.
- The black dot is ψ^*.

You might like to try experimenting with different initial conditions.

25.7 Ergodicity

Under irreducibility, yet another important result obtains: for all $x \in S$,

$$\frac{1}{m} \sum_{t=1}^{m} \mathbf{1}\{X_t = x\} \to \psi^*(x) \quad \text{as } m \to \infty \tag{25.8}$$

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 (25.8) is valid.

The convergence asserted in (25.8) is a special case of a law of large numbers result for Markov chains — see EDTC, section 4.3.4 for some additional information.

25.7.1 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.

25.8 Computing Expectations

We sometimes want to compute mathematical expectations of functions of X_t of the form

$$\mathbb{E}[h(X_t)] \tag{25.9}$$

and conditional expectations such as

$$\mathbb{E}[h(X_{t+k}) \mid X_t = x] \tag{25.10}$$

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 (25.9) 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 (25.10), 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

$$\mathbb{E}[h(X_{t+k}) \mid X_t = x] = (P^k h)(x) \tag{25.11}$$

The vector $P^k h$ stores the conditional expectation $\mathbb{E}[h(X_{t+k}) \mid X_t = x]$ over all x.

25.8.1 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 (25.6)).

To verify the law of iterated expectations, use equation (25.11) 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 $\psi_t P^k h = \psi_{t+k} h = \mathbb{E}[h(X_{t+k})]$.

25.8.2 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

$$\mathbb{E}[\sum_{j=0}^{\infty} \beta^j h(X_{t+j}) \mid X_t = x\Big] = [(I - \beta P)^{-1} h](x)$$

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**".

25.9 Exercises

Exercise 25.9.1

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 to Exercise 25.9.1

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=f'$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 25.9.2

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 [DLP13].)

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;
```

(continues on next page)

(continued from previous page)

```
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('\w', 'x +++ y ****** z')  # \w matches alphanumerics
```

```
['x', 'y', 'z']
```

```
re.findall('\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 to Exercise 25.9.2

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('\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):
```

(continues on next page)

(continued from previous page)

```
    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 25.9.3

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 [Tau86] 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 $\{x_0, \dots, x_{n-1}\} \subset \mathbb{R}$ 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 to Exercise 25.9.3

A solution from the QuantEcon.py library can be found here.

CHAPTER

TWENTYSIX

INVENTORY DYNAMICS

Contents

- *Inventory Dynamics*
  - *Overview*
  - *Sample Paths*
  - *Marginal Distributions*
  - *Exercises*

26.1 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 also optimal in certain circumstances.

A review of early literature and some macroeconomic implications can be found in [Cap85].

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

```
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
from numba import njit, float64, prange
from numba.experimental import jitclass
```

26.2 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.

```
firm_data = [
    ('s', float64),          # restock trigger level
    ('S', float64),          # capacity
    ('mu', float64),         # shock location parameter
    ('sigma', float64)       # shock scale parameter
]


@jitclass(firm_data)
class Firm:

    def __init__(self, s=10, S=100, mu=1.0, sigma=0.5):

        self.s, self.S, self.mu, self.sigma = s, S, mu, sigma

    def update(self, x):
        "Update the state from t to t+1 given current state x."

        Z = np.random.randn()
        D = np.exp(self.mu + self.sigma * Z)
        if x <= self.s:
            return max(self.S - D, 0)
        else:
            return max(x - D, 0)

    def sim_inventory_path(self, x_init, sim_length):

        X = np.empty(sim_length)
        X[0] = x_init

        for t in range(sim_length-1):
            X[t+1] = self.update(X[t])
        return X
```

Let's run a first simulation, of a single path:

```
firm = Firm()

s, S = firm.s, firm.S
sim_length = 100
x_init = 50

X = firm.sim_inventory_path(x_init, sim_length)

fig, ax = plt.subplots()
bbox = (0., 1.02, 1., .102)
legend_args = {'ncol': 3,
               'bbox_to_anchor': bbox,
               'loc': 3,
               'mode': 'expand'}

ax.plot(X, label="inventory")
ax.plot(np.full(sim_length, s), 'k--', label="$s$")
ax.plot(np.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(np.full(sim_length, s), 'k--', label="$s$")
ax.plot(np.full(sim_length, S), 'k-', label="$S$")
ax.set_ylim(0, S+10)
ax.legend(**legend_args)
```

(continues on next page)

(continued from previous page)

```
for i in range(400):
    X = firm.sim_inventory_path(x_init, sim_length)
    ax.plot(X, 'b', alpha=0.2, lw=0.5)

plt.show()
```

26.3 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)

ax.set_xticks((T,))
```

(continues on next page)

(continued from previous page)

```
ax.set_xticklabels((r'$T$',))

sample = np.empty(M)
for m in range(M):
    X = firm.sim_inventory_path(x_init, 2 * T)
    ax.plot(X, 'b-', lw=1, alpha=0.5)
    ax.plot((T,), (X[T+1],), 'ko', alpha=0.5)
    sample[m] = X[T+1]

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()

sample = np.empty(M)
for m in range(M):
    X = firm.sim_inventory_path(x_init, T+1)
    sample[m] = X[T]

ax.hist(sample,
```

(continues on next page)

(continued from previous page)

```
         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

```
from sklearn.neighbors import KernelDensity

def plot_kde(sample, ax, label=''):

    xmin, xmax = 0.9 * min(sample), 1.1 * max(sample)
    xgrid = np.linspace(xmin, xmax, 200)
    kde = KernelDensity(kernel='gaussian').fit(sample[:, None])
    log_dens = kde.score_samples(xgrid[:, None])

    ax.plot(xgrid, np.exp(log_dens), label=label)
```

```
fig, ax = plt.subplots()
plot_kde(sample, ax)
plt.show()
```

The allocation of probability mass is similar to what was shown by the histogram just above.

26.4 Exercises

Exercise 26.4.1

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 to Exercise 26.4.1

Below is one possible solution:

The computations involve a lot of CPU cycles so we have tried to write the code efficiently.

This meant writing a specialized function rather than using the class above.

```
s, S, mu, sigma = firm.s, firm.S, firm.mu, firm.sigma

@njit(parallel=True)
def shift_firms_forward(current_inventory_levels, num_periods):
```

(continues on next page)

(continued from previous page)

```
    num_firms = len(current_inventory_levels)
    new_inventory_levels = np.empty(num_firms)

    for f in prange(num_firms):
        x = current_inventory_levels[f]
        for t in range(num_periods):
            Z = np.random.randn()
            D = np.exp(mu + sigma * Z)
            if x <= s:
                x = max(S - D, 0)
            else:
                x = max(x - D, 0)
        new_inventory_levels[f] = x

    return new_inventory_levels
```

```
x_init = 50
num_firms = 50_000

sample_dates = 0, 10, 50, 250, 500, 750

first_diffs = np.diff(sample_dates)

fig, ax = plt.subplots()

X = np.full(num_firms, x_init)

current_date = 0
for d in first_diffs:
    X = shift_firms_forward(X, d)
    current_date += d
    plot_kde(X, ax, label=f't = {current_date}')

ax.set_xlabel('inventory')
ax.set_ylabel('probability')
ax.legend()
plt.show()
```

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 26.4.2

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 to Exercise 26.4.2

Here is one solution.

Again, the computations are relatively intensive so we have written a a specialized function rather than using the class above.

We will also use parallelization across firms.

```
@njit(parallel=True)
def compute_freq(sim_length=50, x_init=70, num_firms=1_000_000):

    firm_counter = 0  # Records number of firms that restock 2x or more
    for m in prange(num_firms):
        x = x_init
        restock_counter = 0  # Will record number of restocks for firm m

        for t in range(sim_length):
            Z = np.random.randn()
            D = np.exp(mu + sigma * Z)
            if x <= s:
```

(continues on next page)

(continued from previous page)

```
                x = max(S - D, 0)
                restock_counter += 1
            else:
                x = max(x - D, 0)

        if restock_counter > 1:
            firm_counter += 1

    return firm_counter / num_firms
```

Note the time the routine takes to run, as well as the output.

```
%%time

freq = compute_freq()
print(f"Frequency of at least two stock outs = {freq}")
```

```
Frequency of at least two stock outs = 0.447604
CPU times: user 3.88 s, sys: 7.35 ms, total: 3.89 s
Wall time: 942 ms
```

Try switching the `parallel` flag to `False` in the jitted function above.

Depending on your system, the difference can be substantial.

(On our desktop machine, the speed up is by a factor of 5.)

CHAPTER

TWENTYSEVEN

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
```

27.1 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
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
from quantecon import LinearStateSpace
from scipy.stats import norm
import random
```

27.2 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

$$
\begin{aligned}
x_{t+1} &= Ax_t + Cw_{t+1} \\
y_t &= Gx_t \\
x_0 &\sim N(\mu_0, \Sigma_0)
\end{aligned}
$$

27.2.1 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 (27.1) 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 $w_{t+1} \sim N(0, I)$.

27.2.2 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

$$y_{t+1} = \phi_0 + \phi_1 y_t + \phi_2 y_{t-1} \quad \text{s.t.} \quad y_0, y_{-1} \text{ given} \tag{27.1}$$

To map (27.1) into our state space system (27.1), 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, (27.1) and (27.1) 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 (27.1) to represent the model

$$y_{t+1} = \phi_1 y_t + \phi_2 y_{t-1} + \phi_3 y_{t-2} + \phi_4 y_{t-3} + \sigma w_{t+1} \tag{27.2}$$

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 (27.2) is termed a *vector autoregression*.

To map this into (27.1), 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 (27.1) 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 (27.2).

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

$$A = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \qquad C = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \qquad G = \begin{bmatrix} a & b \end{bmatrix} \tag{27.3}$$

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.

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.

[1] The eigenvalues of A are $(1, -1, i, -i)$.

27.2.3 Moving Average Representations

A nonrecursive expression for x_t as a function of $x_0, w_1, w_2, \dots, w_t$ can be found by using (27.1) 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 (27.4) 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 (27.4), 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*.

27.3 Distributions and Moments

27.3.1 Unconditional Moments

Using (27.1), 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

$$\mu_{t+1} = A\mu_t \quad \text{with} \quad \mu_0 \text{ given} \tag{27.4}$$

Here μ_0 is a primitive given in (27.1).

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

$$\Sigma_{t+1} = A\Sigma_t A' + CC' \quad \text{with} \quad \Sigma_0 \text{ given} \tag{27.5}$$

As with μ_0, the matrix Σ_0 is a primitive given in (27.1).

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

$$\mathbb{E}[y_t] = \mathbb{E}[Gx_t] = G\mu_t \tag{27.6}$$

The variance-covariance matrix of y_t is easily shown to be

$$\text{Var}[y_t] = \text{Var}[Gx_t] = G\Sigma_t G' \tag{27.7}$$

27.3.2 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

$$u \sim N(\bar{u}, S) \quad \text{and} \quad v = a + Bu \implies v \sim N(a + B\bar{u}, BSB') \tag{27.8}$$

In particular, given our Gaussian assumptions on the primitives and the linearity of (27.1) 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 (27.4) and (27.5).

Letting μ_t and Σ_t be as defined by these equations, we have

$$x_t \sim N(\mu_t, \Sigma_t) \tag{27.9}$$

By similar reasoning combined with (27.6) and (27.7),

$$y_t \sim N(G\mu_t, G\Sigma_t G') \tag{27.10}$$

[2] The correct way to argue this is by induction. Suppose that x_t is Gaussian. Then (27.1) and (27.8) 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.

27.3.3 Ensemble Interpretations

How should we interpret the distributions defined by (27.9)–(27.10)?

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 (27.2).

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)
    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
```

(continues on next page)

(continued from previous page)

```
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), scale=float(np.sqrt(Sigma_y)))
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=r'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 (27.10).

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 $t = 0, \dots, 50$.

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))

ax.plot(population_means, color='g', lw=2, alpha=0.8, label='$G\mu_t$')
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

$$\frac{1}{I}\sum_{i=1}^{I}(x_T^i - \bar{x}_T)(x_T^i - \bar{x}_T)' \to \Sigma_T \qquad (I \to \infty)$$

27.3.4 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

$$p(x, y) = p(y \mid x)p(x) \qquad \text{(joint } = \text{ conditional } \times \text{ marginal)}$$

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 (27.1), 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*

$$\Sigma_{t+j,t} := \mathbb{E}[(x_{t+j} - \mu_{t+j})(x_t - \mu_t)'] \tag{27.11}$$

Elementary calculations show that

$$\Sigma_{t+j,t} = A^j \Sigma_t \tag{27.12}$$

Notice that $\Sigma_{t+j,t}$ in general depends on both j, the gap between the two dates, and t, the earlier date.

27.4 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.

27.4.1 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()
    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)
```

(continues on next page)

(continued from previous page)

```
    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*.

27.4.2 Stationary Distributions

In our setting, a distribution ψ_∞ is said to be *stationary* for x_t if

$$x_t \sim \psi_\infty \quad \text{and} \quad x_{t+1} = Ax_t + Cw_{t+1} \quad \implies \quad x_{t+1} \sim \psi_\infty$$

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 (27.4) and (27.5) respectively.

27.4.3 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 (27.4) and (27.5) 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 (27.12), 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.

27.4.4 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 (27.5) 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

$$\mu_{1,t+1} = A_1 \mu_{1,t} + a \tag{27.13}$$

Assume now that the moduli of the eigenvalues of A_1 are all strictly less than one.

Then (27.13) 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 (27.5) converge to the fixed point of the *discrete Lyapunov equation* in the first line of (27.14).

27.4.5 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.

27.5 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 (27.6) and (27.7) now become

$$\mathbb{E}[y_t] = \mathbb{E}[Gx_t + Hv_t] = G\mu_t \tag{27.14}$$

The variance-covariance matrix of y_t is easily shown to be

$$\text{Var}[y_t] = \text{Var}[Gx_t + Hv_t] = G\Sigma_t G' + HH' \tag{27.15}$$

The distribution of y_t is therefore

$$y_t \sim N(G\mu_t, G\Sigma_t G' + HH')$$

27.6 Prediction

The theory of prediction for linear state space systems is elegant and simple.

27.6.1 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

$$\mathbb{E}_t[x_{t+1}] := \mathbb{E}[x_{t+1} \mid x_t, x_{t-1}, \dots, x_0] = Ax_t$$

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 $x_t, x_{t-1}, \dots, x_0$.

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

$$x_{t+j} = A^j x_t + A^{j-1} C w_{t+1} + A^{j-2} C w_{t+2} + \cdots + A^0 C w_{t+j}$$

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

$$\mathbb{E}_t[y_{t+j}] = \mathbb{E}_t[G x_{t+j} + H v_{t+j}] = G A^j x_t$$

27.6.2 Covariance of Prediction Errors

It is useful to obtain the covariance matrix of the vector of j-step-ahead prediction errors

$$x_{t+j} - \mathbb{E}_t[x_{t+j}] = \sum_{s=0}^{j-1} A^s C w_{t-s+j} \tag{27.16}$$

Evidently,

$$V_j := \mathbb{E}_t[(x_{t+j} - \mathbb{E}_t[x_{t+j}])(x_{t+j} - \mathbb{E}_t[x_{t+j}])'] = \sum_{k=0}^{j-1} A^k C C' A^{k'} \tag{27.17}$$

V_j defined in (27.17) can be calculated recursively via $V_1 = CC'$ and

$$V_j = CC' + A V_{j-1} A', \quad j \geq 2 \tag{27.18}$$

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

$$V_\infty = CC' + A V_\infty A' \tag{27.19}$$

Equation (27.19) 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.

27.7 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.

27.8 Exercises

Exercise 27.8.1

In several contexts, we want to compute forecasts of geometric sums of future random variables governed by the linear state-space system (27.1).

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 to Exercise 27.8.1

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

TWENTYEIGHT

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
```

28.1 Overview

This lecture creates non-stochastic and stochastic versions of Paul Samuelson's celebrated multiplier accelerator model [Sam39].

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
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
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
```

28.1.1 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 [Sar87].)

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 [Sar87].)

28.2 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.
- a is the marginal propensity to consume in the Keynesian consumption function $C_t = aY_{t-1} + \gamma$.
- b is the "accelerator coefficient" in the "investment accelerator" $I_t = b(Y_{t-1} - Y_{t-2})$.
- $\{\epsilon_t\}$ 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

$$C_t = aY_{t-1} + \gamma \tag{28.1}$$

with the investment accelerator

$$I_t = b(Y_{t-1} - Y_{t-2}) \tag{28.2}$$

and the national income identity

$$Y_t = C_t + I_t + G_t \tag{28.3}$$

- The parameter a is peoples' *marginal propensity to consume* out of income - equation (28.1) asserts that people consume a fraction of $a \in (0, 1)$ of each additional dollar of income.
- The parameter $b > 0$ is the investment accelerator coefficient - equation (28.2) asserts that people invest in physical capital when income is increasing and disinvest when it is decreasing.

Equations (28.1), (28.2), and (28.3) imply the following second-order linear difference equation for national income:

$$Y_t = (a + b)Y_{t-1} - bY_{t-2} + (\gamma + G_t)$$

or

$$Y_t = \rho_1 Y_{t-1} + \rho_2 Y_{t-2} + (\gamma + G_t) \tag{28.4}$$

where $\rho_1 = (a + b)$ and $\rho_2 = -b$.

To complete the model, we require two **initial conditions**.

If the model is to generate time series for $t = 0, \dots, T$, we require initial values

$$Y_{-1} = \bar{Y}_{-1}, \quad Y_{-2} = \bar{Y}_{-2}$$

We'll ordinarily set the parameters (a, b) 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.

28.2.1 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 (28.4), leading to the **second-order scalar linear stochastic difference equation**:

$$Y_t = G_t + a(1-b)Y_{t-1} - abY_{t-2} + \sigma\epsilon_t \tag{28.5}$$

28.2.2 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 (28.5) as

$$Y_t = \rho_1 Y_{t-1} + \rho_2 Y_{t-2}$$

or

$$Y_{t+2} - \rho_1 Y_{t+1} - \rho_2 Y_t = 0 \tag{28.6}$$

To discover the properties of the solution of (28.6), it is useful first to form the **characteristic polynomial** for (28.6):

$$z^2 - \rho_1 z - \rho_2 \tag{28.7}$$

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 (28.7), the characteristic polynomial (28.7) equals zero:

$$z^2 - \rho_1 z - \rho_2 = (z - \lambda_1)(z - \lambda_2) = 0 \tag{28.8}$$

Equation (28.8) 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 (28.6).

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

$$c_1 = ve^{i\theta}, \;\; c_2 = ve^{-i\theta}$$

So we can write

$$\begin{aligned} Y_t &= ve^{i\theta} r^t e^{i\omega t} + ve^{-i\theta} r^t e^{-i\omega t} \\ &= vr^t[e^{i(\omega t+\theta)} + e^{-i(\omega t+\theta)}] \\ &= 2vr^t \cos(\omega t + \theta) \end{aligned}$$

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** $\check{p} = \frac{2\pi}{\omega}$ and **damping factor** r.

We say that $\check{p}$ 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 [Sam39], we want to choose the parameters a, b 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.

28.2.3 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 a, b 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 (28.7) 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:

$$A = \begin{bmatrix} 1 & 0 & 0 \\ \gamma + G & \rho_1 & \rho_2 \\ 0 & 1 & 0 \end{bmatrix}$$

28.3 Implementation

We'll start by drawing an informative graph from page 189 of [Sar87]

```
def param_plot():

    """This function creates the graph on page 189 of
    Sargent Macroeconomic Theory, second edition, 1987.
    """

    fig, ax = plt.subplots(figsize=(10, 6))
    ax.set_aspect('equal')

    # Set axis
    xmin, ymin = -3, -2
    xmax, ymax = -xmin, -ymin
    plt.axis([xmin, xmax, ymin, ymax])

    # Set axis labels
    ax.set(xticks=[], yticks=[])
    ax.set_xlabel(r'$\rho_2$', fontsize=16)
    ax.xaxis.set_label_position('top')
    ax.set_ylabel(r'$\rho_1$', rotation=0, fontsize=16)
    ax.yaxis.set_label_position('right')

    # Draw (t1, t2) points
    ρ1 = np.linspace(-2, 2, 100)
    ax.plot(ρ1, -abs(ρ1) + 1, c='black')
    ax.plot(ρ1, np.full_like(ρ1, -1), c='black')
    ax.plot(ρ1, -(ρ1**2 / 4), c='black')

    # Turn normal axes off
    for spine in ['left', 'bottom', 'top', 'right']:
        ax.spines[spine].set_visible(False)
```

(continues on next page)

(continued from previous page)

```
    # Add arrows to represent axes
    axes_arrows = {'arrowstyle': '<|-|>', 'lw': 1.3}
    ax.annotate('', xy=(xmin, 0), xytext=(xmax, 0), arrowprops=axes_arrows)
    ax.annotate('', xy=(0, ymin), xytext=(0, ymax), arrowprops=axes_arrows)

    # Annotate the plot with equations
    plot_arrowsl = {'arrowstyle': '-|>', 'connectionstyle': "arc3, rad=-0.2"}
    plot_arrowsr = {'arrowstyle': '-|>', 'connectionstyle': "arc3, rad=0.2"}
    ax.annotate(r'$\rho_1 + \rho_2 < 1$', xy=(0.5, 0.3), xytext=(0.8, 0.6),
                arrowprops=plot_arrowsr, fontsize='12')
    ax.annotate(r'$\rho_1 + \rho_2 = 1$', xy=(0.38, 0.6), xytext=(0.6, 0.8),
                arrowprops=plot_arrowsr, fontsize='12')
    ax.annotate(r'$\rho_2 < 1 + \rho_1$', xy=(-0.5, 0.3), xytext=(-1.3, 0.6),
                arrowprops=plot_arrowsl, fontsize='12')
    ax.annotate(r'$\rho_2 = 1 + \rho_1$', xy=(-0.38, 0.6), xytext=(-1, 0.8),
                arrowprops=plot_arrowsl, fontsize='12')
    ax.annotate(r'$\rho_2 = -1$', xy=(1.5, -1), xytext=(1.8, -1.3),
                arrowprops=plot_arrowsl, fontsize='12')
    ax.annotate(r'${\rho_1}^2 + 4\rho_2 = 0$', xy=(1.15, -0.35),
                xytext=(1.5, -0.3), arrowprops=plot_arrowsr, fontsize='12')
    ax.annotate(r'${\rho_1}^2 + 4\rho_2 < 0$', xy=(1.4, -0.7),
                xytext=(1.8, -0.6), arrowprops=plot_arrowsr, fontsize='12')

    # Label categories of solutions
    ax.text(1.5, 1, 'Explosive\n growth', ha='center', fontsize=16)
    ax.text(-1.5, 1, 'Explosive\n oscillations', ha='center', fontsize=16)
    ax.text(0.05, -1.5, 'Explosive oscillations', ha='center', fontsize=16)
    ax.text(0.09, -0.5, 'Damped oscillations', ha='center', fontsize=16)

    # Add small marker to y-axis
    ax.axhline(y=1.005, xmin=0.495, xmax=0.505, c='black')
    ax.text(-0.12, -1.12, '-1', fontsize=10)
    ax.text(-0.12, 0.98, '1', fontsize=10)

    return fig

param_plot()
plt.show()
```

The graph portrays regions in which the (λ_1, λ_2) root pairs implied by the $(\rho_1 = (a + b), \rho_2 = -b)$ 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 (a, b).

28.3.1 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:
        print('Explosive oscillations')
    elif ρ1 + ρ2 > 1:
        print('Explosive growth')
    elif discriminant < 0:
        print('Roots are complex with modulus less than one; \
```

(continues on next page)

(continued from previous page)

```
therefore damped oscillations')
    else:
        print('Roots are real and absolute values are less than one; \
therefore get smooth convergence to a steady state')
```

```
### Test the categorize_solution function

categorize_solution(1.3, -.4)
```

```
Roots are real and absolute values are less than one; therefore get smooth␣
↪convergence to a steady state
```

28.3.2 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('Time $t$')
    plt.ylabel('$Y_t$', rotation=0)
    plt.grid()
    plt.show()
```

28.3.3 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)

The function also plots a Y_t starting from initial conditions that we set

```
# This is a 'manual' method

def y_nonstochastic(y_0=100, y_1=80, α=.92, β=.5, γ=10, n=80):

    """Takes values of parameters and computes the roots of characteristic
    polynomial. It tells whether they are real or complex and whether they
    are less than unity in absolute value.It also computes a simulation of
    length n starting from the two given initial conditions for national
    income
    """

    roots = []

    ρ1 = α + β
    ρ2 = -β
```

(continues on next page)

(continued from previous page)

```
    print(f'ρ_1 is {ρ1}')
    print(f'ρ_2 is {ρ2}')

    discriminant = ρ1 ** 2 + 4 * ρ2

    if discriminant == 0:
        roots.append(-ρ1 / 2)
        print('Single real root: ')
        print(''.join(str(roots)))
    elif discriminant > 0:
        roots.append((-ρ1 + sqrt(discriminant).real) / 2)
        roots.append((-ρ1 - sqrt(discriminant).real) / 2)
        print('Two real roots: ')
        print(''.join(str(roots)))
    else:
        roots.append((-ρ1 + sqrt(discriminant)) / 2)
        roots.append((-ρ1 - sqrt(discriminant)) / 2)
        print('Two complex roots: ')
        print(''.join(str(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 -0.5
Two real roots:
[-0.6459687576256715, -0.7740312423743284]
Absolute values of roots are less than one
```

28.3.4 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 (a, b) and (ρ_1, ρ_2), pairs that would generate those roots

```
### code to reverse-engineer a cycle
### y_t = r^t (c_1 cos(ϕ t) + c2 sin(ϕ t))
###

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 a and the accelerator coefficient b
    that verifies those roots.
    """
    g1 = cmath.rect(r, ϕ)  # Generate two complex roots
    g2 = cmath.rect(r, -ϕ)
    ρ1 = g1 + g2           # Implied ρ1, ρ2
    ρ2 = -g1 * g2
    b = -ρ2                # Reverse-engineer a and b that validate these
    a = ρ1 - b
```

(continues on next page)

(continued from previous page)

```
    return ρ1, ρ2, a, b

## Now let's use the function in an example
## Here are the example parameters

r = .95
period = 10                 # Length of cycle in units of time
ϕ = 2 * math.pi/period

## Apply the function

ρ1, ρ2, a, b = f(r, ϕ)

print(f"a, b = {a}, {b}")
print(f"ρ1, ρ2 = {ρ1}, {ρ2}")
```

```
a, b = (0.6346322893124001+0j), (0.9024999999999999-0j)
ρ1, ρ2 = (1.5371322893124+0j), (-0.9024999999999999+0j)
```

```
## Print the real components of ρ1 and ρ2

ρ1 = ρ1.real
ρ2 = ρ2.real

ρ1, ρ2
```

```
(1.5371322893124, -0.9024999999999999)
```

28.3.5 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}, {ϕ}")
print(f"p1, p2 = {p1}, {p2}")
# print(f"g1, g2 = {g1}, {g2}")

print(f"a, b = {a}, {b}")
print(f"ρ1, ρ2 = {ρ1}, {ρ2}")
```

```
r, ϕ = 0.95, 0.6283185307179586
p1, p2 = (0.95, 0.6283185307179586), (0.95, -0.6283185307179586)
a, b = (0.6346322893124001+0j), (0.9024999999999999-0j)
ρ1, ρ2 = 1.5371322893124, -0.9024999999999999
```

```python
##=== This method uses numpy to calculate roots ===#


def y_nonstochastic(y_0=100, y_1=80, α=.9, β=.8, γ=10, n=80):

    """ Rather than computing the roots of the characteristic
    polynomial by hand as we did earlier, this function
    enlists numpy to do the work for us
    """

    # Useful constants
    ρ1 = α + β
    ρ2 = -β

    categorize_solution(ρ1, ρ2)

    # Find roots of polynomial
    roots = np.roots([1, -ρ1, -ρ2])
    print(f'Roots are {roots}')

    # Check if real or complex
    if all(isinstance(root, complex) for root in roots):
        print('Roots are complex')
    else:
        print('Roots are real')

    # Check if roots are less than one
    if all(abs(root) < 1 for root in roots):
        print('Roots are less than one')
    else:
        print('Roots are not less than one')

    # Define transition equation
    def transition(x, t): return ρ1 * x[t - 1] + ρ2 * x[t - 2] + γ

    # Set initial conditions
    y_t = [y_0, y_1]

    # Generate y_t series
    for t in range(2, n):
        y_t.append(transition(y_t, t))

    return y_t

plot_y(y_nonstochastic())
```

```
Roots are complex with modulus less than one; therefore damped oscillations
Roots are [0.85+0.27838822j 0.85-0.27838822j]
Roots are complex
Roots are less than one
```

28.3.6 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, a, b = f(r, ϕ)

# Drop the imaginary part so that it is a valid input into y_nonstochastic
a = a.real
b = b.real

print(f"a, b = {a}, {b}")

ytemp = y_nonstochastic(α=a, β=b, y_0=20, y_1=30)
plot_y(ytemp)
```

```
a, b = 0.6180339887498949, 1.0
Roots are complex with modulus less than one; therefore damped oscillations
Roots are [0.80901699+0.58778525j 0.80901699-0.58778525j]
Roots are complex
Roots are not less than one
```

28.3.7 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)
```

$$\left[\frac{\rho_1}{2} - \frac{\sqrt{\rho_1^2 + 4\rho_2}}{2}, \ \frac{\rho_1}{2} + \frac{\sqrt{\rho_1^2 + 4\rho_2}}{2} \right]$$

```
a = Symbol("α")
b = Symbol("β")
r1 = a + b
r2 = -b

sympy.solve(z**2 - r1*z - r2, z)
```

$$\left[\frac{\alpha}{2} + \frac{\beta}{2} - \frac{\sqrt{\alpha^2 + 2\alpha\beta + \beta^2 - 4\beta}}{2}, \ \frac{\alpha}{2} + \frac{\beta}{2} + \frac{\sqrt{\alpha^2 + 2\alpha\beta + \beta^2 - 4\beta}}{2} \right]$$

28.4 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 and proceeds to analyze the roots of the characteristic
    polynomial and also generate a simulation.
    """

    # Useful constants
    ρ1 = α + β
    ρ2 = -β

    # Categorize solution
    categorize_solution(ρ1, ρ2)

    # Find roots of polynomial
    roots = np.roots([1, -ρ1, -ρ2])
    print(roots)

    # Check if real or complex
    if all(isinstance(root, complex) for root in roots):
        print('Roots are complex')
    else:
        print('Roots are real')

    # Check if roots are less than one
    if all(abs(root) < 1 for root in roots):
        print('Roots are less than one')
    else:
        print('Roots are not less than one')

    # Generate shocks
    ϵ = np.random.normal(0, 1, n)

    # Define transition equation
    def transition(x, t): return ρ1 * \
        x[t - 1] + ρ2 * x[t - 2] + γ + σ * ϵ[t]

    # Set initial conditions
    y_t = [y_0, y_1]

    # Generate y_t series
    for t in range(2, n):
        y_t.append(transition(y_t, t))

    return y_t

plot_y(y_stochastic())
```

```
Roots are real and absolute values are less than one; therefore get smooth␣
↪convergence to a steady state
[0.7236068 0.2763932]
```

(continues on next page)

(continued from previous page)

```
Roots are real
Roots are less than one
```

Let's do a simulation in which there are shocks and the characteristic polynomial has complex roots

```
r = .97

period = 10   #  Length of cycle in units of time
ϕ = 2 * math.pi/period

### Apply the  reverse-engineering function f

ρ1, ρ2, a, b = f(r, ϕ)

# Drop the imaginary part so that it is a valid input into y_nonstochastic
a = a.real
b = b.real

print(f"a, b = {a}, {b}")
plot_y(y_stochastic(y_0=40, y_1 = 42, α=a, β=b, σ=2, n=100))
```

```
a, b = 0.6285929690873979, 0.9409000000000001
Roots are complex with modulus less than one; therefore damped oscillations
[0.78474648+0.57015169j 0.78474648-0.57015169j]
Roots are complex
Roots are less than one
```

28.5 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
    """

    # Useful constants
    ρ1 = α + β
    ρ2 = -β

    # Categorize solution
    categorize_solution(ρ1, ρ2)

    # Find roots of polynomial
    roots = np.roots([1, -ρ1, -ρ2])
    print(roots)
```

(continues on next page)

(continued from previous page)

```
    # Check if real or complex
    if all(isinstance(root, complex) for root in roots):
        print('Roots are complex')
    else:
        print('Roots are real')

    # Check if roots are less than one
    if all(abs(root) < 1 for root in roots):
        print('Roots are less than one')
    else:
        print('Roots are not less than one')

    # Generate shocks
    ϵ = np.random.normal(0, 1, n)

    def transition(x, t, g):

        # Non-stochastic - separated to avoid generating random series
        # when not needed
        if σ == 0:
            return ρ1 * x[t - 1] + ρ2 * x[t - 2] + γ + g

        # Stochastic
        else:
            ϵ = np.random.normal(0, 1, n)
            return ρ1 * x[t - 1] + ρ2 * x[t - 2] + γ + g + σ * ϵ[t]

    # Create list and set initial conditions
    y_t = [y_0, y_1]

    # Generate y_t series
    for t in range(2, n):

        # No government spending
        if g == 0:
            y_t.append(transition(y_t, t))

        # Government spending (no shock)
        elif g != 0 and duration == None:
            y_t.append(transition(y_t, t))

        # Permanent government spending shock
        elif duration == 'permanent':
            if t < g_t:
                y_t.append(transition(y_t, t, g=0))
            else:
                y_t.append(transition(y_t, t, g=g))

        # One-off government spending shock
        elif duration == 'one-off':
            if t == g_t:
                y_t.append(transition(y_t, t, g=g))
            else:
                y_t.append(transition(y_t, t, g=0))
    return y_t
```

A permanent government spending shock can be simulated as follows

```
plot_y(y_stochastic_g(g=10, g_t=20, duration='permanent'))
```

```
Roots are real and absolute values are less than one; therefore get smooth␣
↪convergence to a steady state
[0.7236068 0.2763932]
Roots are real
Roots are less than one
```

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'))
```

```
Roots are real and absolute values are less than one; therefore get smooth␣
↪convergence to a steady state
[0.7236068 0.2763932]
Roots are real
Roots are less than one
```

28.6 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
    multiple-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
    n : int
```

(continues on next page)

(continued from previous page)

```
        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
        self.ρ1 = α + β
        self.ρ2 = -β
        self.roots = np.roots([1, -self.ρ1, -self.ρ2])

    def root_type(self):
        if all(isinstance(root, complex) for root in self.roots):
            return 'Complex conjugate'
        elif len(self.roots) > 1:
            return 'Double real'
        else:
            return 'Single real'

    def root_less_than_one(self):
        if all(abs(root) < 1 for root in self.roots):
            return True

    def solution_type(self):
        ρ1, ρ2 = self.ρ1, self.ρ2
        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:
```

(continues on next page)

(continued from previous page)

```
            return 'Steady state'

    def _transition(self, x, t, g):

        # Non-stochastic - separated to avoid generating random series
        # when not needed
        if self.σ == 0:
            return self.ρ1 * x[t - 1] + self.ρ2 * x[t - 2] + self.γ + g

        # Stochastic
        else:
            ϵ = np.random.normal(0, 1, self.n)
            return self.ρ1 * x[t - 1] + self.ρ2 * x[t - 2] + self.γ + g \
                + self.σ * ϵ[t]

    def generate_series(self):

        # Create list and set initial conditions
        y_t = [self.y_0, self.y_1]

        # Generate y_t series
        for t in range(2, self.n):

            # No government spending
            if self.g == 0:
                y_t.append(self._transition(y_t, t))

            # Government spending (no shock)
            elif self.g != 0 and self.duration == None:
                y_t.append(self._transition(y_t, t))

            # Permanent government spending shock
            elif self.duration == 'permanent':
                if t < self.g_t:
                    y_t.append(self._transition(y_t, t, g=0))
                else:
                    y_t.append(self._transition(y_t, t, g=self.g))

            # One-off government spending shock
            elif self.duration == 'one-off':
                if t == self.g_t:
                    y_t.append(self._transition(y_t, t, g=self.g))
                else:
                    y_t.append(self._transition(y_t, t, g=0))
        return y_t

    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:
            print('Absolute value of roots is less than one')
        else:
            print('Absolute value of roots is not less than one')
```

(continues on next page)

(continued from previous page)

```
        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):
        fig, ax = plt.subplots(figsize=(10, 6))
        ax.plot(self.generate_series())
        ax.set(xlabel='Iteration', xlim=(0, self.n))
        ax.set_ylabel('$Y_t$', rotation=0)
        ax.grid()

        # Add parameter values to plot
        paramstr = f'$\\alpha={self.α:.2f}$ \n $\\beta={self.β:.2f}$ \n \
        $\\gamma={self.γ:.2f}$ \n $\\sigma={self.σ:.2f}$ \n \
        $\\rho_1={self.ρ1:.2f}$ \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):

        # Uses the param_plot() function defined earlier (it is then able
        # to be used standalone or as part of the model)

        fig = param_plot()
        ax = fig.gca()

        # Add λ values to legend
        for i, root in enumerate(self.roots):
            if isinstance(root, complex):
                # Need to fill operator for positive as string is split apart
                operator = ['+', '']
                label = rf'$\lambda_{i+1} = {sam.roots[i].real:.2f} {operator[i]}
↪{sam.roots[i].imag:.2f}i$'
            else:
                label = rf'$\lambda_{i+1} = {sam.roots[i].real:.2f}$'
            ax.scatter(0, 0, 0, label=label) # dummy to add to legend

        # Add ρ pair to plot
        ax.scatter(self.ρ1, self.ρ2, 100, 'red', '+',
            label=r'$(\ \rho_1, \ \rho_2 \ )$', zorder=5)

        plt.legend(fontsize=12, loc=3)

        return fig
```

28.6.1 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()
```

28.6.2 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()
```

28.7 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

```
"""This script maps the Samuelson model in the the
``LinearStateSpace`` class
"""
α = 0.8
β = 0.9
ρ1 = α + β
ρ2 = -β
γ = 10
σ = 1
```

(continues on next page)

(continued from previous page)

```
g = 10
n = 100

A = [[1,         0,       0],
     [γ + g,    ρ1,      ρ2],
     [0,         1,       0]]

G = [[γ + g, ρ1,    ρ2],          # this is Y_{t+1}
     [γ,      α,     0],          # this is C_{t+1}
     [0,      β,    -β]]          # this is I_{t+1}

μ_0 = [1, 100, 50]
C = np.zeros((3,1))
C[1] = σ # stochastic

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))
    ax.grid()

axes[-1].set_xlabel('Iteration')

plt.show()
```

28.7.1 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(w)
```

```
[0.85+0.42130749j 0.85-0.42130749j 1.  +0.j        ]
```

28.7.2 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

        # Define intial conditions
        self.μ_0 = [1, y_0, y_1]

        self.ρ1 = α + β
        self.ρ2 = -β

        # Define transition matrix
        self.A = [[1,                  0,          0],
                  [γ + g,        self.ρ1,    self.ρ2],
                  [0,                  1,          0]]

        # Define output matrix
        self.G = [[γ + g, self.ρ1, self.ρ2],          # this is Y_{t+1}
                  [γ,            α,        0],          # this is C_{t+1}
                  [0,            β,       -β]]          # this is I_{t+1}

        self.C = np.zeros((3, 1))
        self.C[1] = σ  # stochastic

        # Initialize LSS with parameters from Samuelson model
        LinearStateSpace.__init__(self, self.A, self.C, self.G, mu_0=self.μ_0)

    def plot_simulation(self, ts_length=100, stationary=True):

        # Temporarily store original parameters
        temp_mu = self.mu_0
        temp_Sigma = self.Sigma_0

        # Set distribution parameters equal to their stationary
        # values for simulation
        if stationary == True:
            try:
                self.mu_x, self.mu_y, self.Sigma_x, self.Sigma_y, self.Sigma_yx = \
```

(continues on next page)

(continued from previous page)

```
                    self.stationary_distributions()
                self.mu_0 = self.mu_x
                self.Sigma_0 = self.Sigma_x
            # Exception where no convergence achieved when
            #calculating stationary distributions
            except ValueError:
                print('Stationary distribution does not exist')

        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))
            ax.grid()

        axes[-1].set_xlabel('Iteration')

        # Reset distribution parameters to their initial values
        self.mu_0 = temp_mu
        self.Sigma_0 = temp_Sigma

        return fig

    def plot_irf(self, j=5):

        x, y = self.impulse_response(j)

        # Reshape into 3 x j matrix for plotting purposes
        yimf = np.array(y).flatten().reshape(j+1, 3).T

        fig, axes = plt.subplots(3, 1, sharex=True, figsize=(12, 8))
        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)
            ax.grid()

        axes[0].set_title('Impulse Response Functions')
        axes[-1].set_xlabel('Iteration')

        return fig

    def multipliers(self, j=5):
        x, y = self.impulse_response(j)
        return np.sum(np.array(y).flatten().reshape(j+1, 3), axis=0)
```

28.7.3 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])
```

28.8 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()
```

Output (Y_t)
404
402
400
398
Consumption (C_t)
394
392
390
388
Investment (I_t)
0.04
0.02
0.00
−0.02
−0.04
0
20
40
60
80
100
Iteration

Output (Y_t)
404
402
400
398
Consumption (C_t)
394
392
390
388
Investment (I_t)
0.04
0.02
0.00
−0.02
−0.04
0
20
40
60
80
100
Iteration

```
pure_multiplier = SamuelsonLSS(α=0.8, β=0)
```

```
pure_multiplier.plot_simulation()
```

```
pure_multiplier.plot_irf(100)
```

Impulse Response Functions
Y_t
C_t
I_t
Iteration

Impulse Response Functions
Y_t
C_t
I_t
Iteration

28.9 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 [Sam39].

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

TWENTYNINE

KESTEN PROCESSES AND FIRM DYNAMICS

Contents

- *Kesten Processes and Firm Dynamics*
  - *Overview*
  - *Kesten Processes*
  - *Heavy Tails*
  - *Application: Firm Dynamics*
  - *Exercises*

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install quantecon
!pip install --upgrade yfinance
```

29.1 Overview

Previously 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
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
import quantecon as qe
```

The following two lines are only added to avoid a `FutureWarning` caused by compatibility issues between pandas and matplotlib.

```
from pandas.plotting import register_matplotlib_converters
register_matplotlib_converters()
```

Additional technical background related to this lecture can be found in the monograph of [BDM+16].

29.2 Kesten Processes

A **Kesten process** is a stochastic process of the form

$$X_{t+1} = a_{t+1} X_t + \eta_{t+1} \tag{29.1}$$

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.

29.2.1 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.

```
import yfinance as yf

s = yf.download('^IXIC', '2006-1-1', '2019-11-1')['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

$$\sigma_{t+1}^2 = \alpha_0 + \sigma_t^2(\alpha_1 \xi_{t+1}^2 + \beta) \tag{29.2}$$

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

$$r_t = \sigma_t \zeta_t \tag{29.3}$$

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.

29.2.2 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

$$w_{t+1} = R_{t+1} s w_t + y_{t+1} \tag{29.4}$$

where $\{R_t\}$ is the gross rate of return on assets.

If $\{R_t\}$ and $\{y_t\}$ are both IID, then (29.4) is a Kesten process.

29.2.3 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 (29.1) if

$$X_t \sim F^* \quad \Longrightarrow \quad a_{t+1} X_t + \eta_{t+1} \sim F^* \tag{29.5}$$

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

$$F^*(y) = \int \mathbb{P}\{a_{t+1} x + \eta_{t+1} \leq y\} F^*(dx) \quad \text{for all } y \geq 0. \tag{29.6}$$

The left hand side is the distribution of the next period state when the current state is drawn from F^*.

The equality in (29.6) states that this distribution is unchanged.

29.2.4 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 (29.4),
- 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

$$\int \mathbb{P}\{R_{t+1} s w + y_{t+1} \leq y\} F^*(dw)$$

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.

29.2.5 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

$$\mathbb{E} \ln a_t < 0 \quad \text{and} \quad \mathbb{E} \eta_t < \infty \tag{29.7}$$

then a unique stationary distribution exists on $\mathbb{R}_+$.

- See, for example, theorem 2.1.3 of [BDM+16], which provides slightly weaker conditions.

As one application of this result, we see that the wealth process (29.4) will have a unique stationary distribution whenever labor income has finite mean and $\mathbb{E} \ln R_t + \ln s < 0$.

29.3 Heavy Tails

Under certain conditions, the stationary distribution of a Kesten process has a Pareto tail.

(See our earlier lecture on heavy-tailed distributions for background.)

This fact is significant for economics because of the prevalence of Pareto-tailed distributions.

29.3.1 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., [BDM+16], theorem 2.4.4) states that if

1. the stationarity conditions in (29.7) 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.

29.3.2 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 of 10 paths when a_t and b_t are lognormal.

```
μ = -0.5
σ = 1.0

def kesten_ts(ts_length=100):
    x = np.zeros(ts_length)
    for t in range(ts_length-1):
        a = np.exp(μ + σ * np.random.randn())
        b = np.exp(np.random.randn())
        x[t+1] = a * x[t] + b
    return x

fig, ax = plt.subplots()

num_paths = 10
np.random.seed(12)

for i in range(num_paths):
    ax.plot(kesten_ts())

ax.set(xlabel='time', ylabel='$X_t$')
plt.show()
```

29.4 Application: Firm Dynamics

As noted in our lecture on heavy tails, for common measures of firm size such as revenue or employment, the US firm size distribution exhibits a Pareto tail (see, e.g., [Axt01], [Gab16]).

Let us try to explain this rather striking fact using the Kesten–Goldie Theorem.

29.4.1 Gibrat's Law

It was postulated many years ago by Robert Gibrat [Gib31] 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

$$\frac{s_{t+1}}{s_t} = a_{t+1} \tag{29.8}$$

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., [Eva87] and [Hal87]) and
2. the growth rate of small firms is more volatile than that of large firms [DRS89].

On the other hand, Gibrat's law is generally found to be a reasonable approximation for large firms [Eva87].

We can accommodate these empirical findings by modifying (29.8) to

$$s_{t+1} = a_{t+1} s_t + b_{t+1} \tag{29.9}$$

where $\{a_t\}$ and $\{b_t\}$ are both IID and independent of each other.

In the exercises you are asked to show that (29.9) is more consistent with the empirical findings presented above than Gibrat's law in (29.8).

29.4.2 Heavy Tails

So what has this to do with Pareto tails?

The answer is that (29.9) 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.

29.5 Exercises

Exercise 29.5.1

Simulate and plot 15 years of daily returns (consider each year as having 250 working days) using the GARCH(1, 1) process in (29.2)–(29.3).

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 to Exercise 29.5.1

Here is one solution:

```
α_0 = 1e-5
α_1 = 0.1
β = 0.9

years = 15
days = years * 250

def garch_ts(ts_length=days):
    σ2 = 0
    r = np.zeros(ts_length)
    for t in range(ts_length-1):
        ξ = np.random.randn()
        σ2 = α_0 + σ2 * (α_1 * ξ**2 + β)
        r[t] = np.sqrt(σ2) * np.random.randn()
    return r

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

(continues on next page)

(continued from previous page)

```
np.random.seed(12)

ax.plot(garch_ts(), alpha=0.7)

ax.set(xlabel='time', ylabel='$\\sigma_t^2$')
plt.show()
```

Exercise 29.5.2

In our discussion of firm dynamics, it was claimed that (29.9) is more consistent with the empirical literature than Gibrat's law in (29.8).

(The empirical literature was reviewed immediately above (29.9).)

In what sense is this true (or false)?

Solution to Exercise 29.5.2

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 (29.9) are more consistent with points 1-2 than Gibrat's law.

To see why, we rewrite (29.9) in terms of growth dynamics:

$$\frac{s_{t+1}}{s_t} = a_{t+1} + \frac{b_{t+1}}{s_t} \tag{29.10}$$

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 (29.10) clearly approaches Gibrat's law (29.8) as s_t gets large.

Exercise 29.5.3

Consider an arbitrary Kesten process as given in (29.1).

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 to Exercise 29.5.3

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 29.5.4

One unrealistic aspect of the firm dynamics specified in (29.9) 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 [Hop92].

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

$$s_{t+1} = e_{t+1}\mathbb{1}\{s_t < \bar{s}\} + (a_{t+1}s_t + b_{t+1})\mathbb{1}\{s_t \geq \bar{s}\} \tag{29.11}$$

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 (29.11) 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 (29.9).

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 (29.11) 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 to Exercise 29.5.4

Here's one solution. First we generate the observations:

```
from numba import njit, prange
from numpy.random import randn


@njit(parallel=True)
def generate_draws(μ_a=-0.5,
                   σ_a=0.1,
                   μ_b=0.0,
                   σ_b=0.5,
                   μ_e=0.0,
                   σ_e=0.5,
                   s_bar=1.0,
                   T=500,
```

(continues on next page)

(continued from previous page)

```
                    M=1_000_000,
                    s_init=1.0):

    draws = np.empty(M)
    for m in prange(M):
        s = s_init
        for t in range(T):
            if s < s_bar:
                new_s = np.exp(μ_e + σ_e *  randn())
            else:
                a = np.exp(μ_a + σ_a * randn())
                b = np.exp(μ_b + σ_b * randn())
                new_s = a * s + b
            s = new_s
        draws[m] = s

    return draws

data = generate_draws()
```

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

THIRTY

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 a `GPU` is available here

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install quantecon
```

30.1 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, [BB18].

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.

30.1.1 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
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
import quantecon as qe
from numba import njit, float64, prange
from numba.experimental import jitclass
```

30.2 Lorenz Curves and the Gini Coefficient

Before we investigate wealth dynamics, we briefly review some measures of inequality.

30.2.1 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

```
n = 10_000                      # size of sample
w = np.exp(np.random.randn(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 = np.random.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.

30.2.2 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 = np.random.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')
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.

30.3 A Model of Wealth Dynamics

Having discussed inequality measures, let us now turn to wealth dynamics.

The model we will study is

$$w_{t+1} = (1 + r_{t+1})s(w_t) + y_{t+1} \tag{30.1}$$

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

$$s(w) = s_0 w \cdot \mathbb{1}\{w \geq \hat{w}\} \tag{30.2}$$

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.

30.4 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,
                 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
```

(continues on next page)

(continued from previous page)

```
        # 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.

```
@njit
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
```

(continues on next page)

(continued from previous page)

```
    """
    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.

```
@njit(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.

30.5 Applications

Let's try simulating the model at different parameter values and investigate the implications for the wealth distribution.

30.5.1 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*.

30.5.2 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
```

(continues on next page)

(continued from previous page)

```
    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=f'$\psi^*$ 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 28s, sys: 35.2 ms, total: 1min 28s
Wall time: 12.3 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("$\mu_r$")
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=f'$\psi^*$ 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 27s, sys: 43.7 ms, total: 1min 28s
Wall time: 11.4 s
```

We see that greater volatility has the effect of increasing inequality in this model.

30.6 Exercises

Exercise 30.6.1

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 to Exercise 30.6.1

Here is one solution, which produces a good match between theory and simulation.

```
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()
```

(continues on next page)

(continued from previous page)

```
for i, a in enumerate(a_vals):
    y = np.random.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 30.6.2

The wealth process (30.1) is similar to a *Kesten process*.

This is because, according to (30.2), 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 (30.1) 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 to Exercise 30.6.2

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()
```

CHAPTER

THIRTYONE

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
```

31.1 Overview

This lecture provides a simple and intuitive introduction to the Kalman filter, for those 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

For additional (more advanced) reading on the Kalman filter, see

- [LS18], section 2.7
- [AM05]

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
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
from scipy import linalg
import numpy as np
import matplotlib.cm as cm
```

(continues on next page)

(continued from previous page)

```
from quantecon import Kalman, LinearStateSpace
from scipy.stats import norm
from scipy.integrate import quad
from scipy.linalg import eigvals
```

31.2 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 country Y and our mission is to track it.

Let $x \in \mathbb{R}^2$ denote the current location of the missile—a pair indicating latitude-longitude coordinates on a map.

At the present moment in time, the precise location x is unknown, but we do have some beliefs about x.

One way to summarize our knowledge is a point prediction $\hat{x}$

- But what if the President wants to know the probability that the missile is currently over the Sea of Japan?
- Then it is better to summarize our initial beliefs with a bivariate probability density p
  - $\int_E p(x)dx$ indicates the probability that we attach to the missile being in region E.

The density p is called our *prior* for the random variable x.

To keep things tractable in our example, we assume that our prior is Gaussian.

In particular, we take

$$p = N(\hat{x}, \Sigma) \tag{31.1}$$

where $\hat{x}$ is the mean of the distribution and Σ is a 2×2 covariance matrix. In our simulations, we will suppose that

$$\hat{x} = \begin{pmatrix} 0.2 \\ -0.2 \end{pmatrix}, \qquad \Sigma = \begin{pmatrix} 0.4 & 0.3 \\ 0.3 & 0.45 \end{pmatrix} \tag{31.2}$$

This density $p(x)$ is shown below as a contour map, with the center of the red ellipse being equal to $\hat{x}$.

```
# Set up the Gaussian prior density p
Σ = [[0.4, 0.3], [0.3, 0.45]]
Σ = np.matrix(Σ)
x_hat = np.matrix([0.2, -0.2]).T
# Define the matrices G and R from the equation y = G x + N(0, R)
G = [[1, 0], [0, 1]]
G = np.matrix(G)
R = 0.5 * Σ
# The matrices A and Q
A = [[1.2, 0], [0, -0.2]]
A = np.matrix(A)
Q = 0.3 * Σ
# The observed value of y
y = np.matrix([2.3, -1.9]).T

# 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)
```

(continues on next page)

(continued from previous page)

```
def bivariate_normal(x, y, σ_x=1.0, σ_y=1.0, μ_x=0.0, μ_y=0.0, σ_xy=0.0):
    """
    Compute and return the probability density function of bivariate normal
    distribution of normal random variables x and y

    Parameters
    ----------
    x : array_like(float)
        Random variable

    y : array_like(float)
        Random variable

    σ_x : array_like(float)
          Standard deviation of random variable x

    σ_y : array_like(float)
          Standard deviation of random variable y

    μ_x : scalar(float)
          Mean value of random variable x

    μ_y : scalar(float)
          Mean value of random variable y

    σ_xy : array_like(float)
           Covariance of random variables x and y

    """

    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

def gen_gaussian_plot_vals(μ, C):
    "Z values for plotting the bivariate Gaussian N(μ, C)"
    m_x, m_y = float(μ[0]), float(μ[1])
    s_x, s_y = np.sqrt(C[0, 0]), np.sqrt(C[1, 1])
    s_xy = C[0, 1]
    return bivariate_normal(X, Y, s_x, s_y, m_x, m_y, s_xy)

# Plot the figure

fig, ax = plt.subplots(figsize=(10, 8))
ax.grid()

Z = gen_gaussian_plot_vals(x_hat, Σ)
ax.contourf(X, Y, Z, 6, alpha=0.6, cmap=cm.jet)
cs = ax.contour(X, Y, Z, 6, colors="black")
ax.clabel(cs, inline=1, fontsize=10)

plt.show()
```

31.2.1 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 $y = (2.3, -1.9)$.

The next figure shows the original prior $p(x)$ and the new reported location y

```
fig, ax = plt.subplots(figsize=(10, 8))
ax.grid()

Z = gen_gaussian_plot_vals(x_hat, Σ)
ax.contourf(X, Y, Z, 6, alpha=0.6, cmap=cm.jet)
cs = ax.contour(X, Y, Z, 6, colors="black")
ax.clabel(cs, inline=1, fontsize=10)
ax.text(float(y[0]), float(y[1]), "$y$", fontsize=20, color="black")

plt.show()
```

The bad news is that our sensors are imprecise.

In particular, we should interpret the output of our sensor not as $y = x$, but rather as

$$y = Gx + v, \quad \text{where} \quad v \sim N(0, R) \tag{31.3}$$

Here G and R are 2×2 matrices with R positive definite. Both are assumed known, and the noise term v is assumed to be independent of x.

How then should we combine our prior $p(x) = N(\hat{x}, \Sigma)$ and this new information y to improve our understanding of the location of the missile?

As you may have guessed, the answer is to use Bayes' theorem, which tells us to update our prior $p(x)$ to $p(x \,|\, y)$ via

$$p(x \,|\, y) = \frac{p(y \,|\, x)\, p(x)}{p(y)}$$

where $p(y) = \int p(y \,|\, x)\, p(x) dx$.

In solving for $p(x \,|\, y)$, we observe that

- $p(x) = N(\hat{x}, \Sigma)$.
- In view of (31.3), the conditional density $p(y \,|\, x)$ is $N(Gx, R)$.
- $p(y)$ does not depend on x, and enters into the calculations only as a normalizing constant.

Because we are in a linear and Gaussian framework, the updated density can be computed by calculating population linear regressions.

In particular, the solution is known[1] to be

$$p(x \mid y) = N(\hat{x}^F, \Sigma^F)$$

where

$$\hat{x}^F := \hat{x} + \Sigma G'(G\Sigma G' + R)^{-1}(y - G\hat{x}) \quad \text{and} \quad \Sigma^F := \Sigma - \Sigma G'(G\Sigma G' + R)^{-1}G\Sigma \tag{31.4}$$

Here $\Sigma G'(G\Sigma G' + R)^{-1}$ is the matrix of population regression coefficients of the hidden object $x - \hat{x}$ on the surprise $y - G\hat{x}$.

This new density $p(x \mid y) = N(\hat{x}^F, \Sigma^F)$ 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(figsize=(10, 8))
ax.grid()

Z = gen_gaussian_plot_vals(x_hat, 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)
x_hat_F = x_hat + M * (y - G * x_hat)
Σ_F = Σ - M * G * Σ
new_Z = gen_gaussian_plot_vals(x_hat_F, Σ_F)
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=cm.jet)
ax.text(float(y[0]), float(y[1]), "$y$", fontsize=20, color="black")

plt.show()
```

[1] See, for example, page 93 of [Bis06]. To get from his expressions to the ones used above, you will also need to apply the Woodbury matrix identity.

Our new density twists the prior $p(x)$ in a direction determined by the new information $y - G\hat{x}$.

In generating the figure, we set G to the identity matrix and $R = 0.5\Sigma$ for Σ defined in (31.2).

31.2.2 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.

- $p(x \mid y) = N(\hat{x}^F, \Sigma^F)$ is called the *filtering distribution*

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,

$$x_{t+1} = Ax_t + w_{t+1}, \quad \text{where} \quad w_t \sim N(0, Q) \tag{31.5}$$

Our aim is to combine this law of motion and our current distribution $p(x \mid y) = N(\hat{x}^F, \Sigma^F)$ to come up with a new *predictive* distribution for the location in one unit of time.

In view of (31.5), all we have to do is introduce a random vector $x^F \sim N(\hat{x}^F, \Sigma^F)$ 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.

Elementary calculations and the expressions in (31.4) tell us that

$$\mathbb{E}[Ax^F + w] = A\mathbb{E}x^F + \mathbb{E}w = A\hat{x}^F = A\hat{x} + A\Sigma G'(G\Sigma G' + R)^{-1}(y - G\hat{x})$$

and

$$\text{Var}[Ax^F + w] = A\,\text{Var}[x^F]A' + Q = A\Sigma^F A' + Q = A\Sigma A' - A\Sigma G'(G\Sigma G' + R)^{-1}G\Sigma A' + Q$$

The matrix $A\Sigma G'(G\Sigma G' + R)^{-1}$ 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 or $\hat{x}$.

Using this notation, we can summarize our results as follows.

Our updated prediction is the density $N(\hat{x}_{new}, \Sigma_{new})$ where

$$\begin{aligned} \hat{x}_{new} &:= A\hat{x} + K_\Sigma(y - G\hat{x}) \\ \Sigma_{new} &:= A\Sigma A' - K_\Sigma G\Sigma A' + Q \end{aligned}$$

- The density $p_{new}(x) = N(\hat{x}_{new}, \Sigma_{new})$ 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(figsize=(10, 8))
ax.grid()

# Density 1
Z = gen_gaussian_plot_vals(x_hat, 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)
x_hat_F = x_hat + M * (y - G * x_hat)
Σ_F = Σ - M * G * Σ
Z_F = gen_gaussian_plot_vals(x_hat_F, Σ_F)
cs2 = ax.contour(X, Y, Z_F, 6, colors="black")
ax.clabel(cs2, inline=1, fontsize=10)

# Density 3
new_x_hat = A * x_hat_F
new_Σ = A * Σ_F * A.T + Q
new_Z = gen_gaussian_plot_vals(new_x_hat, 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=cm.jet)
ax.text(float(y[0]), float(y[1]), "$y$", fontsize=20, color="black")

plt.show()
```

31.2.3 The Recursive Procedure

Let's look back at what we've done.

We started the current period with a prior $p(x)$ for the location x of the missile.

We then used the current measurement y to update to $p(x \mid y)$.

Finally, we used the law of motion (31.5) for $\{x_t\}$ to update to $p_{new}(x)$.

If we now step into the next period, we are ready to go round again, taking $p_{new}(x)$ as the current prior.

Swapping notation $p_t(x)$ for $p(x)$ and $p_{t+1}(x)$ for $p_{new}(x)$, the full recursive procedure is:

1. Start the current period with prior $p_t(x) = N(\hat{x}_t, \Sigma_t)$.
2. Observe current measurement y_t.
3. Compute the filtering distribution $p_t(x \mid y) = N(\hat{x}_t^F, \Sigma_t^F)$ from $p_t(x)$ and y_t, applying Bayes rule and the conditional distribution (31.3).
4. Compute the predictive distribution $p_{t+1}(x) = N(\hat{x}_{t+1}, \Sigma_{t+1})$ from the filtering distribution and (31.5).
5. Increment t by one and go to step 1.

Repeating (31.6), the dynamics for $\hat{x}_t$ and Σ_t are as follows

$$\begin{aligned}\hat{x}_{t+1} &= A\hat{x}_t + K_{\Sigma_t}(y_t - G\hat{x}_t) \\ \Sigma_{t+1} &= A\Sigma_t A' - K_{\Sigma_t} G\Sigma_t A' + Q\end{aligned}$$

These are the standard dynamic equations for the Kalman filter (see, for example, [LS18], page 58).

31.3 Convergence

The matrix Σ_t is a measure of the uncertainty of our prediction $\hat{x}_t$ 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 $\hat{x}_t$ is made based on information available at $t-1$, not t.

Even if we know the precise value of x_{t-1} (which we don't), the transition equation (31.5) 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 (31.6):

$$\Sigma_{t+1} = A\Sigma_t A' - A\Sigma_t G'(G\Sigma_t G' + R)^{-1} G\Sigma_t A' + Q \tag{31.6}$$

This is a nonlinear difference equation in Σ_t.

A fixed point of (31.6) is a constant matrix Σ such that

$$\Sigma = A\Sigma A' - A\Sigma G'(G\Sigma G' + R)^{-1} G\Sigma A' + Q \tag{31.7}$$

Equation (31.6) is known as a discrete-time Riccati difference equation.

Equation (31.7) 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 [AHMS96] and [AM05], chapter 4.

A sufficient (but not necessary) condition is that all the eigenvalues λ_i of A satisfy $|\lambda_i| < 1$ (cf. e.g., [AM05], p. 77).

(This strong condition assures that the unconditional distribution of x_t converges as $t \to +\infty$.)

In this case, for any initial choice of Σ_0 that is both non-negative and symmetric, the sequence $\{\Sigma_t\}$ in (31.6) converges to a non-negative symmetric matrix Σ that solves (31.7).

31.4 Implementation

The class `Kalman` from the QuantEcon.py package implements the Kalman filter

- Instance data consists of:
  - the moments $(\hat{x}_t, \Sigma_t)$ of the current prior.
  - An instance of the LinearStateSpace class from QuantEcon.py.

The latter represents a linear state space model of the form

$$\begin{aligned}x_{t+1} &= Ax_t + Cw_{t+1} \\ y_t &= Gx_t + Hv_t\end{aligned}$$

where the shocks w_t and v_t are IID standard normals.

To connect this with the notation of this lecture we set

$$Q := CC' \quad \text{and} \quad R := HH'$$

- 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 $(\hat{x}_t, \Sigma_t)$ to $(\hat{x}_t^F, \Sigma_t^F)$
  - `filtered_to_forecast`, which updates the filtering distribution to the predictive distribution – which becomes the new prior $(\hat{x}_{t+1}, \Sigma_{t+1})$
  - `update`, which combines the last two methods
  - a `stationary_values`, which computes the solution to (31.7) and the corresponding (stationary) Kalman gain

You can view the program on GitHub.

31.5 Exercises

Exercise 31.5.1

Consider the following simple application of the Kalman filter, loosely based on [LS18], 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 (31.5) 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 to simulate the model and, using the code from `kalman.py`, plot the first five predictive densities $p_t(x) = N(\hat{x}_t, \Sigma_t)$.

As shown in [LS18], sections 2.9.1–2.9.2, these distributions asymptotically put all mass on the unknown value θ.

In the simulation, take $\theta = 10$, $\hat{x}_0 = 8$ and $\Sigma_0 = 1$.

Your figure should – modulo randomness – look something like this

Solution to Exercise 31.5.1

```
# 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
x_hat_0, Σ_0 = 8, 1
kalman = Kalman(ss, x_hat_0, Σ_0)
```

(continues on next page)

First 5 densities when $\theta = 10.0$
$t = 0$
$t = 1$
$t = 2$
$t = 3$
$t = 4$
0.0
0.2
0.4
0.6
0.8
5
6
7
8
9
10
11
12

(continued from previous page)

```
# 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(figsize=(10,8))
xgrid = np.linspace(θ - 5, θ + 2, 200)

for i in range(N):
    # Record the current predicted mean and variance
    m, v = [float(z) for z in (kalman.x_hat, kalman.Sigma)]
    # 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 31.5.2

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

$$z_t := 1 - \int_{\theta-\epsilon}^{\theta+\epsilon} p_t(x)dx$$

for $t = 0, 1, 2, \ldots, T$.

Plot z_t against T, setting $\epsilon = 0.1$ and $T = 600$.

Your figure should show error erratically declining something like this

Solution to Exercise 31.5.2

```
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)
```

(continues on next page)

(continued from previous page)

```
x_hat_0, Σ_0 = 8, 1
kalman = Kalman(ss, x_hat_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 = [float(temp) for temp in (kalman.x_hat, kalman.Sigma)]

    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(figsize=(9, 7))
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 31.5.3

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 $\hat{x}_t$ made by the Kalman filter against a competitor who **is** allowed to observe x_{t-1}.

This competitor will use the conditional expectation $\mathbb{E}[x_t \,|\, x_{t-1}]$, 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 $g^*(x_{t-1}) := \mathbb{E}[x_t \,|\, x_{t-1}]$)

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 squared error.

In particular, your task is to generate a graph plotting observations of both $\|x_t - Ax_{t-1}\|^2$ and $\|x_t - \hat{x}_t\|^2$ against t for $t = 1, \ldots, 50$.

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 $\hat{x}_0 = (8, 8)$.

Finally, set $x_0 = (0, 0)$.

You should end up with a figure similar to the following (modulo randomness)

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.

Solution to Exercise 31.5.3

```
# 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]]
```

(continues on next page)

(continued from previous page)

```
Σ = np.array(Σ)
x_hat = np.array([8, 8])

# Initialize the Kalman filter
kn = Kalman(ss, x_hat, 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])
    e1[t-1] = np.sum((x[:, t] - kn.x_hat.flatten())**2)
```

(continues on next page)

(continued from previous page)

```
    e2[t-1] = np.sum((x[:, t] - A @ x[:, t-1])**2)

fig, ax = plt.subplots(figsize=(9,6))
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]]
```

Exercise 31.5.4

Try varying the coefficient 0.3 in $Q = 0.3I$ up and down.

Observe how the diagonal values in the stationary solution Σ (see (31.7)) 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

THIRTYTWO

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 this quantecon lecture *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.

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 the much firm pays the worker each period as a function of the firm's information each period.

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*.

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
from quantecon import Kalman, LinearStateSpace
from collections import namedtuple
from scipy.stats import multivariate_normal
import matplotlib as mpl

mpl.rcParams['text.usetex'] = True
mpl.rcParams['text.latex.preamble'] = r'\usepackage{{amsmath}}'
```

32.1 A worker's output

A representative worker is permanently employed at a firm.

The workers' output is described by the following dynamic process:

$$\begin{aligned} h_{t+1} &= \alpha h_t + \beta u_t + c w_{t+1}, \quad c_{t+1} \sim \mathcal{N}(0,1) \\ u_{t+1} &= u_t \\ y_t &= g h_t + v_t, \quad v_t \sim \mathcal{N}(0,R) \end{aligned} \tag{32.1}$$

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
- $h_0 \sim \mathcal{N}(\hat{h}_0, \sigma_{h,0})$
- $u_0 \sim \mathcal{N}(\hat{u}_0, \sigma_{u,0})$

Parameters of the model are $\alpha, \beta, c, R, g, \hat{h}_0, \hat{u}_0, \sigma_h, \sigma_u$.

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 $t = 0, 1, 2, \ldots$.

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 $u_0 \sim \mathcal{N}(\hat{u}_0, \sigma_{u,0})$.

The h_t part of a worker's "type" moves over time, but the effort component of the worker's type is $u_t = u_0$.

This means that from the firm's point of view, the worker's effort is effectively an unknown fixed "parameter".

At time $t \geq 1$, for a particular worker the firm observed $y^{t-1} = [y_{t-1}, y_{t-2}, \ldots, y_0]$.

The firm does not observe the worker's "type" (h_0, u_0).

But the firm does observe the worker's output y_t at time t and remembers the worker's past outputs y^{t-1}.

32.2 A firm's wage-setting policy

Based on information about the worker that the firm has at time $t \geq 1$, the firm pays the worker log wage

$$w_t = g E[h_t | y^{t-1}], \quad t \geq 1$$

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.

32.3 A state-space representation

Write system *(32.1.1)* in the state-space form

$$\begin{bmatrix} h_{t+1} \\ u_{t+1} \end{bmatrix} = \begin{bmatrix} \alpha & \beta \\ 0 & 1 \end{bmatrix} \begin{bmatrix} h_t \\ u_t \end{bmatrix} + \begin{bmatrix} c \\ 0 \end{bmatrix} w_{t+1}$$
$$y_t = \begin{bmatrix} g & 0 \end{bmatrix} \begin{bmatrix} h_t \\ u_t \end{bmatrix} + v_t$$

which is equivalent with

$$\begin{aligned} x_{t+1} &= Ax_t + Cw_{t+1} \\ y_t &= Gx_t + v_t \\ x_0 &\sim \mathcal{N}(\hat{x}_0, \Sigma_0) \end{aligned} \tag{32.2}$$

where

$$x_t = \begin{bmatrix} h_t \\ u_t \end{bmatrix}, \quad \hat{x}_0 = \begin{bmatrix} \hat{h}_0 \\ \hat{u}_0 \end{bmatrix}, \quad \Sigma_0 = \begin{bmatrix} \sigma_{h,0} & 0 \\ 0 & \sigma_{u,0} \end{bmatrix}$$

To compute the firm's wage setting policy, we first we 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=4, σ_u=4):

    A = np.array([[α, β],
                  [0, 1]])
    C = np.array([[c],
                  [0]])
    G = np.array([g, 1])

    # Define initial state and covariance matrix
    xhat_0 = np.array([[hhat_0],
                       [uhat_0]])

    Σ_0 = np.array([[σ_h, 0],
                    [0, σ_u]])

    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 (32.2).

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
```

(continues on next page)

(continued from previous page)

```
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]
```

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".

32.4 An Innovations Representation

We have all the objects in hand required to form an innovations representation for the output process $\{y_t\}_{t=0}^T$ 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 K_t is the Kalman gain matrix at time t.

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-1))
y_hat_t = np.zeros(T-1)
x_hat_t = np.zeros((2, T-1))

for t in range(1, T):
    kalman.update(y[t])
    x_hat, Σ = kalman.x_hat, kalman.Sigma
    Σ_t[:, :, t-1] = Σ
    x_hat_t[:, t-1] = x_hat.reshape(-1)
    y_hat_t[t-1] = worker.G @ x_hat

x_hat_t = np.concatenate((x[:, 1][:, np.newaxis],
                    x_hat_t), axis=1)
Σ_t = np.concatenate((worker.Σ_0[:, :, np.newaxis],
                    Σ_t), axis=2)
u_hat_t = x_hat_t[1, :]
```

For a draw of h_0, u_0, we plot $Ey_t = G\hat{x}_t$ where $\hat{x}_t = E[x_t|y^{t-1}]$.

We also plot $E[u_0|y^{t-1}]$, which is the firm 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 as the firm's inference $E[u_0|y^{t-1}]$ of the worker's work ethic converges toward the hidden u_0, which is not directly observed by the firm.

```
fig, ax = plt.subplots(1, 2)

ax[0].plot(y_hat_t, label=r'$E[y_t| y^{t-1}]$')
ax[0].set_xlabel('Time')
ax[0].set_ylabel(r'$E[y_t]$')
ax[0].set_title(r'$E[y_t]$ over time')
ax[0].legend()

ax[1].plot(u_hat_t, label=r'$E[u_t|y^{t-1}]$')
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'$E[u_t|y^{t-1}]$')
ax[1].set_title('Inferred work ethic over time')
ax[1].legend()

fig.tight_layout()
plt.show()
```

32.5 Some Computational Experiments

Let's look at Σ_0 and Σ_T 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.]]
```

```
print(Σ_t[:, :, -1])
```

```
[[0.08805027 0.00100377]
 [0.00100377 0.00398351]]
```

Evidently, entries in the conditional covariance matrix become smaller over time.

It is enlightening to portray how conditional covariance matrices Σ_t evolve by plotting confidence ellipsoides around $E[x_t|y^{t-1}]$ 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
    mu = x_hat_t[:, t]
    cov = Σ_t[:, :, t]
    mvn = multivariate_normal(mean=mu, 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+1}')
    axs[i].set_xlabel(r'$h_{{{}}}$'.format(str(t+1)))
    axs[i].set_ylabel(r'$u_{{{}}}$'.format(str(t+1)))

    cov_latex = r'$\Sigma_{{{}}}= \begin{{bmatrix}} {:.2f} & {:.2f} \\ {:.2f} & {:.2f}
 ↪ \end{{bmatrix}}$'.format(
        t+1, 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 affects the shape of the confidence ellipsoid 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.

```
# For example, we might want h_0 = 0 and u_0 = 4
mu_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=mu_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 and u_0 = uhhat_0 = 4.0:
worker = create_worker(hhat_0=0.0, uhat_0=4.0)

ss_example = LinearStateSpace(A, C, G, np.sqrt(R),
                              # This line takes h_0=hhat_0 and u_0=uhhat_0
                              mu_0=worker.xhat_0,
                              # This line forces exact h_0=hhat_0 and u_0=uhhat_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
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.

```
# First we compute the Kalman filter with initial xhat_0 and Σ_0
kalman = Kalman(ss, xhat_0, Σ_0)
Σ_t = []
y_hat_t = np.zeros(T-1)
u_hat_t = np.zeros(T-1)

# Then we iteratively update the Kalman filter class using
# observation y based on the linear state model above:
for t in range(1, T):
    kalman.update(y[t])
    x_hat, Σ = kalman.x_hat, kalman.Sigma
    Σ_t.append(Σ)
    y_hat_t[t-1] = worker.G @ x_hat
    u_hat_t[t-1] = x_hat[1]


# Generate plots for y_hat_t and u_hat_t
fig, ax = plt.subplots(1, 2)

ax[0].plot(y_hat_t, label=r'$E[y_t| y^{t-1}]$')
ax[0].set_xlabel('Time')
ax[0].set_ylabel(r'$E[y_t]$')
ax[0].set_title(r'$E[y_t]$ over time')
ax[0].legend()

ax[1].plot(u_hat_t, label=r'$E[u_t|y^{t-1}]$')
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'$E[u_t|y^{t-1}]$')
ax[1].set_title('Inferred work ethic over time')
```

(continues on next page)

(continued from previous page)

```
ax[1].legend()

fig.tight_layout()
plt.show()
```

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., 1.]), R=0.5, xhat_0=array([[  7.],
       [100.]]), Σ_0=array([[2.5, 0. ],
       [0. , 3.2]]))
```

We can also simulate the system for $T = 50$ periods for different workers.

The difference between the inferred work ethics and true work ethics converges to 0 over time.

This shows that the filter is gradually teaching the worker and firm about the worker's effort.

```
num_workers = 3
T = 50
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)
ax.set_ylim(ymin=-2, ymax=2)
plt.show()
```

```
# We can also generate plots of u_t:

T = 50
fig, ax = plt.subplots(figsize=(7, 7))

uhat_0s = [2, -2, 1]
αs = [0.2, 0.3, 0.5]
βs = [0.1, 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,
                     # By setting diff=False, it will give u_t
                     diff=False, name=r'$u_{{{}, t}}$'.format(i))

ax.axhline(y=u_0, xmin=0, xmax=0, color='grey',
```

(continues on next page)

(continued from previous page)

```
            linestyle='dashed', label=r'$u_{i, 0}$')
ax.legend(bbox_to_anchor=(1, 0.5))
plt.show()
```

```
# We can also use exact u_0=1 and h_0=2 for all workers

T = 50
fig, ax = plt.subplots(figsize=(7, 7))

# These two lines set u_0=1 and h_0=2 for all workers
mu_0 = np.array([[1],
                 [2]])
Sigma_0 = np.zeros((2,2))
```

(continues on next page)

(continued from previous page)

```
uhat_0s = [2, -2, 1]
αs = [0.2, 0.3, 0.5]
βs = [0.1, 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, mu_0=mu_0, Sigma_0=Sigma_0,
                     diff=False, name=r'$u_{{{}, t}}$'.format(i))

# This controls the boundary of plots
ax.set_ylim(ymin=-3, ymax=3)
ax.axhline(y=u_0, xmin=0, xmax=0, color='grey',
           linestyle='dashed', label=r'$u_{i, 0}$')
ax.legend(bbox_to_anchor=(1, 0.5))
plt.show()
```

```
# We can generate a plot for only one of the workers:

T = 50
fig, ax = plt.subplots(figsize=(7, 7))

mu_0_1 = np.array([[1],
                  [100]])
mu_0_2 = np.array([[1],
                  [30]])
Sigma_0 = np.zeros((2,2))

uhat_0s = 100
αs = 0.5
βs = 0.3

worker = create_worker(uhat_0=uhat_0, α=α, β=β)
simulate_workers(worker, T, ax, mu_0=mu_0_1, Sigma_0=Sigma_0,
                 diff=False, name=r'Hard-working worker')
simulate_workers(worker, T, ax, mu_0=mu_0_2, Sigma_0=Sigma_0,
                 diff=False,
                 title='A hard-working worker and a less hard-working worker',
                 name=r'Normal worker')
ax.axhline(y=u_0, xmin=0, xmax=0, color='grey',
           linestyle='dashed', label=r'$u_{i, 0}$')
ax.legend(bbox_to_anchor=(1, 0.5))
plt.show()
```

32.6 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

THIRTYTHREE

SHORTEST PATHS

Contents

- *Shortest Paths*
  - *Overview*
  - *Outline of the Problem*
  - *Finding Least-Cost Paths*
  - *Solving for Minimum Cost-to-Go*
  - *Exercises*

33.1 Overview

The shortest path problem is a classic problem in mathematics and computer science with applications in

- Economics (sequential decision making, analysis of social networks, etc.)
- Operations research and transportation
- Robotics and artificial intelligence
- Telecommunication network design and routing
- etc., etc.

Variations of the methods we discuss in this lecture are used millions of times every day, in applications such as

- Google Maps
- routing packets on the internet

For us, the shortest path problem also provides a nice introduction to the logic of **dynamic programming**.

Dynamic programming is an extremely powerful optimization technique that we apply in many lectures on this site.

The only scientific library we'll need in what follows is NumPy:

```
import numpy as np
```

33.2 Outline of the Problem

The shortest path problem is one of finding how to traverse a graph from one specified node to another at minimum cost.

Consider the following graph

We wish to travel from node (vertex) A to node G at minimum cost

- Arrows (edges) indicate the movements we can take.
- Numbers on edges indicate the cost of traveling that edge.

(Graphs such as the one above are called weighted directed graphs.)

Possible interpretations of the graph include

- Minimum cost for supplier to reach a destination.
- Routing of packets on the internet (minimize time).
- Etc., etc.

For this simple graph, a quick scan of the edges shows that the optimal paths are

- A, C, F, G at cost 8
- A, D, F, G at cost 8

A
B
C
D
E
F
G
5
3
1
9
2
4
6
8
1
4

A
B
C
D
E
F
G
5
3
1
9
2
4
6
8
1
4

33.3 Finding Least-Cost Paths

For large graphs, we need a systematic solution.

Let $J(v)$ denote the minimum cost-to-go from node v, understood as the total cost from v if we take the best route.

Suppose that we know $J(v)$ for each node v, as shown below for the graph from the preceding example

Note that $J(G) = 0$.

The best path can now be found as follows

1. Start at node $v = A$
2. From current node v, move to any node that solves

$$\min_{w \in F_v} \{c(v, w) + J(w)\} \tag{33.1}$$

where

- F_v is the set of nodes that can be reached from v in one step.
- $c(v, w)$ is the cost of traveling from v to w.

Hence, if we know the function J, then finding the best path is almost trivial.

But how can we find the cost-to-go function J?

Some thought will convince you that, for every node v, the function J satisfies

$$J(v) = \min_{w \in F_v} \{c(v, w) + J(w)\} \tag{33.2}$$

This is known as the *Bellman equation*, after the mathematician Richard Bellman.

The Bellman equation can be thought of as a restriction that J must satisfy.

What we want to do now is use this restriction to compute J.

33.4 Solving for Minimum Cost-to-Go

Let's look at an algorithm for computing J and then think about how to implement it.

33.4.1 The Algorithm

The standard algorithm for finding J is to start an initial guess and then iterate.

This is a standard approach to solving nonlinear equations, often called the method of **successive approximations**.

Our initial guess will be

$$J_0(v) = 0 \text{ for all } v \tag{33.3}$$

Now

1. Set $n = 0$
2. Set $J_{n+1}(v) = \min_{w \in F_v} \{c(v, w) + J_n(w)\}$ for all v
3. If J_{n+1} and J_n are not equal then increment n, go to 2

This sequence converges to J.

Although we omit the proof, we'll prove similar claims in our other lectures on dynamic programming.

33.4.2 Implementation

Having an algorithm is a good start, but we also need to think about how to implement it on a computer.

First, for the cost function c, we'll implement it as a matrix Q, where a typical element is

$$Q(v, w) = \begin{cases} c(v, w) \text{ if } w \in F_v \\ +\infty \text{ otherwise} \end{cases}$$

In this context Q is usually called the **distance matrix**.

We're also numbering the nodes now, with $A = 0$, so, for example

$$Q(1, 2) = \text{ the cost of traveling from B to C}$$

For example, for the simple graph above, we set

```
from numpy import inf

Q = np.array([[inf, 1,   5,   3,   inf, inf, inf],
              [inf, inf, inf, 9,   6,   inf, inf],
              [inf, inf, inf, inf, inf, 2,   inf],
              [inf, inf, inf, inf, inf, 4,   8],
              [inf, inf, inf, inf, inf, inf, 4],
              [inf, inf, inf, inf, inf, inf, 1],
              [inf, inf, inf, inf, inf, inf, 0]])
```

Notice that the cost of staying still (on the principle diagonal) is set to

- np.inf for non-destination nodes — moving on is required.
- 0 for the destination node — here is where we stop.

For the sequence of approximations $\{J_n\}$ of the cost-to-go functions, we can use NumPy arrays.

Let's try with this example and see how we go:

```
nodes = range(7)                              # Nodes = 0, 1, ..., 6
J = np.zeros_like(nodes, dtype=int)        # Initial guess
next_J = np.empty_like(nodes, dtype=int)   # Stores updated guess

max_iter = 500
i = 0

while i < max_iter:
    for v in nodes:
        # minimize Q[v, w] + J[w] over all choices of w
        lowest_cost = inf
        for w in nodes:
            cost = Q[v, w] + J[w]
            if cost < lowest_cost:
                lowest_cost = cost
        next_J[v] = lowest_cost
    if np.equal(next_J, J).all():
        break
    else:
        J[:] = next_J   # Copy contents of next_J to J
        i += 1

print("The cost-to-go function is", J)
```

```
The cost-to-go function is [ 8 10  3  5  4  1  0]
```

This matches with the numbers we obtained by inspection above.

But, importantly, we now have a methodology for tackling large graphs.

33.5 Exercises

Exercise 33.5.1

The text below describes a weighted directed graph.

The line `node0, node1 0.04, node8 11.11, node14 72.21` means that from node0 we can go to

- node1 at cost 0.04
- node8 at cost 11.11
- node14 at cost 72.21

No other nodes can be reached directly from node0.

Other lines have a similar interpretation.

Your task is to use the algorithm given above to find the optimal path and its cost.

Note: You will be dealing with floating point numbers now, rather than integers, so consider replacing `np.equal()` with `np.allclose()`.

```
%%file graph.txt
node0, node1 0.04, node8 11.11, node14 72.21
node1, node46 1247.25, node6 20.59, node13 64.94
node2, node66 54.18, node31 166.80, node45 1561.45
node3, node20 133.65, node6 2.06, node11 42.43
node4, node75 3706.67, node5 0.73, node7 1.02
node5, node45 1382.97, node7 3.33, node11 34.54
node6, node31 63.17, node9 0.72, node10 13.10
node7, node50 478.14, node9 3.15, node10 5.85
node8, node69 577.91, node11 7.45, node12 3.18
node9, node70 2454.28, node13 4.42, node20 16.53
node10, node89 5352.79, node12 1.87, node16 25.16
node11, node94 4961.32, node18 37.55, node20 65.08
node12, node84 3914.62, node24 34.32, node28 170.04
node13, node60 2135.95, node38 236.33, node40 475.33
node14, node67 1878.96, node16 2.70, node24 38.65
node15, node91 3597.11, node17 1.01, node18 2.57
node16, node36 392.92, node19 3.49, node38 278.71
node17, node76 783.29, node22 24.78, node23 26.45
node18, node91 3363.17, node23 16.23, node28 55.84
node19, node26 20.09, node20 0.24, node28 70.54
node20, node98 3523.33, node24 9.81, node33 145.80
node21, node56 626.04, node28 36.65, node31 27.06
node22, node72 1447.22, node39 136.32, node40 124.22
node23, node52 336.73, node26 2.66, node33 22.37
node24, node66 875.19, node26 1.80, node28 14.25
node25, node70 1343.63, node32 36.58, node35 45.55
node26, node47 135.78, node27 0.01, node42 122.00
node27, node65 480.55, node35 48.10, node43 246.24
node28, node82 2538.18, node34 21.79, node36 15.52
node29, node64 635.52, node32 4.22, node33 12.61
node30, node98 2616.03, node33 5.61, node35 13.95
node31, node98 3350.98, node36 20.44, node44 125.88
node32, node97 2613.92, node34 3.33, node35 1.46
node33, node81 1854.73, node41 3.23, node47 111.54
node34, node73 1075.38, node42 51.52, node48 129.45
node35, node52 17.57, node41 2.09, node50 78.81
node36, node71 1171.60, node54 101.08, node57 260.46
node37, node75 269.97, node38 0.36, node46 80.49
node38, node93 2767.85, node40 1.79, node42 8.78
node39, node50 39.88, node40 0.95, node41 1.34
node40, node75 548.68, node47 28.57, node54 53.46
node41, node53 18.23, node46 0.28, node54 162.24
node42, node59 141.86, node47 10.08, node72 437.49
node43, node98 2984.83, node54 95.06, node60 116.23
node44, node91 807.39, node46 1.56, node47 2.14
node45, node58 79.93, node47 3.68, node49 15.51
node46, node52 22.68, node57 27.50, node67 65.48
node47, node50 2.82, node56 49.31, node61 172.64
node48, node99 2564.12, node59 34.52, node60 66.44
node49, node78 53.79, node50 0.51, node56 10.89
node50, node85 251.76, node53 1.38, node55 20.10
node51, node98 2110.67, node59 23.67, node60 73.79
node52, node94 1471.80, node64 102.41, node66 123.03
node53, node72 22.85, node56 4.33, node67 88.35
node54, node88 967.59, node59 24.30, node73 238.61
node55, node84 86.09, node57 2.13, node64 60.80
```

(continues on next page)

(continued from previous page)

```
node56, node76 197.03, node57 0.02, node61 11.06
node57, node86 701.09, node58 0.46, node60 7.01
node58, node83 556.70, node64 29.85, node65 34.32
node59, node90 820.66, node60 0.72, node71 0.67
node60, node76 48.03, node65 4.76, node67 1.63
node61, node98 1057.59, node63 0.95, node64 4.88
node62, node91 132.23, node64 2.94, node76 38.43
node63, node66 4.43, node72 70.08, node75 56.34
node64, node80 47.73, node65 0.30, node76 11.98
node65, node94 594.93, node66 0.64, node73 33.23
node66, node98 395.63, node68 2.66, node73 37.53
node67, node82 153.53, node68 0.09, node70 0.98
node68, node94 232.10, node70 3.35, node71 1.66
node69, node99 247.80, node70 0.06, node73 8.99
node70, node76 27.18, node72 1.50, node73 8.37
node71, node89 104.50, node74 8.86, node91 284.64
node72, node76 15.32, node84 102.77, node92 133.06
node73, node83 52.22, node76 1.40, node90 243.00
node74, node81 1.07, node76 0.52, node78 8.08
node75, node92 68.53, node76 0.81, node77 1.19
node76, node85 13.18, node77 0.45, node78 2.36
node77, node80 8.94, node78 0.98, node86 64.32
node78, node98 355.90, node81 2.59
node79, node81 0.09, node85 1.45, node91 22.35
node80, node92 121.87, node88 28.78, node98 264.34
node81, node94 99.78, node89 39.52, node92 99.89
node82, node91 47.44, node88 28.05, node93 11.99
node83, node94 114.95, node86 8.75, node88 5.78
node84, node89 19.14, node94 30.41, node98 121.05
node85, node97 94.51, node87 2.66, node89 4.90
node86, node97 85.09
node87, node88 0.21, node91 11.14, node92 21.23
node88, node93 1.31, node91 6.83, node98 6.12
node89, node97 36.97, node99 82.12
node90, node96 23.53, node94 10.47, node99 50.99
node91, node97 22.17
node92, node96 10.83, node97 11.24, node99 34.68
node93, node94 0.19, node97 6.71, node99 32.77
node94, node98 5.91, node96 2.03
node95, node98 6.17, node99 0.27
node96, node98 3.32, node97 0.43, node99 5.87
node97, node98 0.30
node98, node99 0.33
node99,
```

```
Overwriting graph.txt
```

Solution to Exercise 33.5.1

First let's write a function that reads in the graph data above and builds a distance matrix.

```
num_nodes = 100
destination_node = 99
```

(continues on next page)

(continued from previous page)

```
def map_graph_to_distance_matrix(in_file):

    # First let's set of the distance matrix Q with inf everywhere
    Q = np.full((num_nodes, num_nodes), np.inf)

    # Now we read in the data and modify Q
    with open(in_file) as infile:
        for line in infile:
            elements = line.split(',')
            node = elements.pop(0)
            node = int(node[4:])     # convert node description to integer
            if node != destination_node:
                for element in elements:
                    destination, cost = element.split()
                    destination = int(destination[4:])
                    Q[node, destination] = float(cost)
            Q[destination_node, destination_node] = 0
    return Q
```

In addition, let's write

1. a "Bellman operator" function that takes a distance matrix and current guess of J and returns an updated guess of J, and
2. a function that takes a distance matrix and returns a cost-to-go function.

We'll use the algorithm described above.

The minimization step is vectorized to make it faster.

```
def bellman(J, Q):
    num_nodes = Q.shape[0]
    next_J = np.empty_like(J)
    for v in range(num_nodes):
        next_J[v] = np.min(Q[v, :] + J)
    return next_J


def compute_cost_to_go(Q):
    num_nodes = Q.shape[0]
    J = np.zeros(num_nodes)       # Initial guess
    max_iter = 500
    i = 0

    while i < max_iter:
        next_J = bellman(J, Q)
        if np.allclose(next_J, J):
            break
        else:
            J[:] = next_J    # Copy contents of next_J to J
            i += 1

    return(J)
```

We used np.allclose() rather than testing exact equality because we are dealing with floating point numbers now.

Finally, here's a function that uses the cost-to-go function to obtain the optimal path (and its cost).

```
def print_best_path(J, Q):
    sum_costs = 0
    current_node = 0
    while current_node != destination_node:
        print(current_node)
        # Move to the next node and increment costs
        next_node = np.argmin(Q[current_node, :] + J)
        sum_costs += Q[current_node, next_node]
        current_node = next_node

    print(destination_node)
    print('Cost: ', sum_costs)
```

Okay, now we have the necessary functions, let's call them to do the job we were assigned.

```
Q = map_graph_to_distance_matrix('graph.txt')
J = compute_cost_to_go(Q)
print_best_path(J, Q)
```

```
0
8
11
18
23
33
41
53
56
57
60
67
70
73
76
85
87
88
93
94
96
97
98
99
Cost:  160.55000000000007
```

The total cost of the path should agree with $J[0]$ so let's check this.

```
J[0]
```

```
160.55
```

Part V

Search

CHAPTER

THIRTYFOUR

JOB SEARCH I: THE MCCALL SEARCH MODEL

Contents

- *Job Search I: The McCall Search Model*
  - *Overview*
  - *The McCall Model*
  - *Computing the Optimal Policy: Take 1*
  - *Computing an Optimal Policy: Take 2*
  - *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
```

34.1 Overview

The McCall search model [McC70] 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
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
from numba import jit, float64
from numba.experimental import jitclass
import quantecon as qe
from quantecon.distributions import BetaBinomial
```

34.2 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

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t y_t$$

The constant β lies in $(0, 1)$ and is called a **discount factor**.

The smaller is β, the more the agent discounts future utility relative to current utility.

The variable y_t is income, equal to

- his/her wage w_t when employed
- unemployment compensation c when unemployed

34.2.1 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 optimally 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.

34.2.2 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.

More precisely, $v^*(w)$ denotes the value of the objective function (35.1) when an agent in this situation makes *optimal* decisions now and at all future 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!

But think of v^* as a function that assigns to each possible wage s the maximal lifetime value that can be obtained with that offer in hand.

A crucial observation is that this function v^* must satisfy the recursion

$$v^*(w) = \max \left\{ \frac{w}{1-\beta}, \, c + \beta \sum_{w' \in \mathbb{W}} v^*(w') q(w') \right\} \tag{34.1}$$

for every possible w in $\mathbb{W}$.

This important equation 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
$$\frac{w}{1-\beta} = w + \beta w + \beta^2 w + \cdots$$
- 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 (34.1).

34.2.3 The Optimal Policy

Suppose for now that we are able to solve (34.1) for the unknown function v^*.

Once we have this function in hand we can behave optimally (i.e., make the right choice between accept and reject).

All we have to do is select the maximal choice on the right-hand side of (34.1).

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 (34.1).

Thus, we have a map from $\mathbb{R}$ 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

$$\bar{w} := (1-\beta)\left\{ c + \beta \sum_{w'} v^*(w') q(w') \right\} \tag{34.2}$$

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 (34.2), we can compute this reservation wage if we can compute the value function.

34.3 Computing the Optimal Policy: Take 1

To put the above ideas into action, we need to compute the value function at each possible state $w \in \mathbb{W}$.

To simplify notation, let's set

$$\mathbb{W} := \{w_1, \dots, w_n\} \quad \text{and} \quad v^*(i) := v^*(w_i)$$

The value function is then represented by the vector $v^* = (v^*(i))_{i=1}^n$.

In view of (34.1), this vector satisfies the nonlinear system of equations

$$v^*(i) = \max\left\{ \frac{w(i)}{1-\beta}, \, c + \beta \sum_{1 \leq j \leq n} v^*(j) q(j) \right\} \quad \text{for } i = 1, \dots, n \tag{34.3}$$

34.3.1 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

$$v'(i) = \max\left\{ \frac{w(i)}{1-\beta}, \, c + \beta \sum_{1 \leq j \leq n} v(j) q(j) \right\} \quad \text{for } i = 1, \dots, n \tag{34.4}$$

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.

34.3.2 Fixed Point Theory

What's the mathematics behind these ideas?

First, one defines a mapping T from $\mathbb{R}^n$ to itself via

$$(Tv)(i) = \max \left\{ \frac{w(i)}{1-\beta}, \, c + \beta \sum_{1 \leq j \leq n} v(j) q(j) \right\} \quad \text{for } i = 1, \dots, n \tag{34.5}$$

(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.

34.3.3 Implementation

Our default for q, the distribution of the state process, will be Beta-binomial.

```
n, a, b = 50, 200, 100                        # default parameters
q_default = BetaBinomial(n, a, b).pdf()       # default choice of q
```

Our default set of values for wages will be

```
w_min, w_max = 10, 60
w_default = np.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 are going to use Numba to accelerate our code.

- See, in particular, the discussion of `@jitclass` in our lecture on Numba.

The following helps Numba by providing some type

```
mccall_data = [
    ('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
]
```

Here's a class that stores the data and computes the values of state-action pairs, i.e. the value in the maximum bracket on the right hand side of the Bellman equation (34.4), given the current state and an arbitrary feasible action.

Default parameter values are embedded in the class.

```
@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_default, q_default

    def state_action_values(self, i, v):
        """
        The values of state-action pairs.
        """
        # 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 * np.sum(v * q)

        return np.array([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.

Here's a function to implement this:

```
def plot_value_function_seq(mcm, ax, num_plots=6):
    """
    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 j in range(n):
            v_next[j] = np.max(mcm.state_action_values(j, v))
        v[:] = v_next  # copy contents into v

    ax.legend(loc='lower right')
```

Now let's create an instance of `McCallModel` and watch iterations $T^k v$ converge from below:

```
mcm = McCallModel()

fig, ax = plt.subplots()
ax.set_xlabel('wage')
ax.set_ylabel('value')
plot_value_function_seq(mcm, ax)
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.

We'll be using JIT compilation via Numba to turbocharge our loops.

```
@jit(nopython=True)
def compute_reservation_wage(mcm,
                             max_iter=500,
                             tol=1e-6):

    # Simplify names
    c, β, w, q = mcm.c, mcm.β, mcm.w, mcm.q

    # == First compute the value function == #

    n = len(w)
    v = w / (1 - β)          # initial guess
    v_next = np.empty_like(v)
    j = 0
    error = tol + 1
    while j < max_iter and error > tol:

        for j in range(n):
            v_next[j] = np.max(mcm.state_action_values(j, v))

        error = np.max(np.abs(v_next - v))
        j += 1

        v[:] = v_next  # copy contents into v

    # == Now compute the reservation wage == #

    return (1 - β) * (c + β * np.sum(v * q))
```

The next line computes the reservation wage at default parameters

```
compute_reservation_wage(mcm)
```

```
47.316499710024964
```

34.3.4 Comparative Statics

Now that we know how to compute the reservation wage, let's see how it varies with parameters.

In particular, let's look at what happens when we change β and c.

```
grid_size = 25
R = np.empty((grid_size, grid_size))

c_vals = np.linspace(10.0, 30.0, grid_size)
β_vals = np.linspace(0.9, 0.99, grid_size)

for i, c in enumerate(c_vals):
```

(continues on next page)

(continued from previous page)

```
    for j, β in enumerate(β_vals):
        mcm = McCallModel(c=c, β=β)
        R[i, j] = compute_reservation_wage(mcm)
```

```
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)


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 both with patience and with unemployment compensation.

34.4 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:

$$h = c + \beta \sum_{s'} v^*(s') q(s') \tag{34.6}$$

The Bellman equation can now be written as

$$v^*(s') = \max \left\{ \frac{w(s')}{1-\beta}, h \right\}$$

Substituting this last equation into (34.6) gives

$$h = c + \beta \sum_{s' \in \mathbb{S}} \max \left\{ \frac{w(s')}{1-\beta}, h \right\} q(s') \tag{34.7}$$

This is a nonlinear equation 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

$$h' = c + \beta \sum_{s' \in \mathbb{S}} \max \left\{ \frac{w(s')}{1-\beta}, h \right\} q(s') \tag{34.8}$$

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:

```
@jit(nopython=True)
def compute_reservation_wage_two(mcm,
                                 max_iter=500,
                                 tol=1e-5):

    # Simplify names
    c, β, w, q = mcm.c, mcm.β, mcm.w, mcm.q

    # == First compute h == #

    h = np.sum(w * q) / (1 - β)
    i = 0
    error = tol + 1
    while i < max_iter and error > tol:

        s = np.maximum(w / (1 - β), h)
        h_next = c + β * np.sum(s * q)
```

(continues on next page)

(continued from previous page)

```
        error = np.abs(h_next - h)
        i += 1

        h = h_next

    # == Now compute the reservation wage == #

    return (1 - β) * h
```

You can use this code to solve the exercise below.

34.5 Exercises

Exercise 34.5.1

Compute the average duration of unemployment when $\beta = 0.99$ and c takes the following values

```
c_vals = np.linspace(10, 40, 25)
```

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`.

Solution to Exercise 34.5.1

Here's one solution

```
cdf = np.cumsum(q_default)

@jit(nopython=True)
def compute_stopping_time(w_bar, seed=1234):

    np.random.seed(seed)
    t = 1
    while True:
        # Generate a wage draw
        w = w_default[qe.random.draw(cdf)]
        # Stop when the draw is above the reservation wage
        if w >= w_bar:
            stopping_time = t
            break
        else:
            t += 1
    return stopping_time

@jit(nopython=True)
def compute_mean_stopping_time(w_bar, num_reps=100000):
    obs = np.empty(num_reps)
    for i in range(num_reps):
        obs[i] = compute_stopping_time(w_bar, seed=i)
```

(continues on next page)

(continued from previous page)

```
    return obs.mean()

c_vals = np.linspace(10, 40, 25)
stop_times = np.empty_like(c_vals)
for i, c in enumerate(c_vals):
    mcm = McCallModel(c=c)
    w_bar = compute_reservation_wage_two(mcm)
    stop_times[i] = compute_mean_stopping_time(w_bar)

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()
```

Exercise 34.5.2

The purpose of this exercise is to show how to replace the discrete wage offer distribution used above with a continuous distribution.

This is a significant topic because many convenient distributions are continuous (i.e., have a density).

Fortunately, the theory changes little in our simple model.

Recall that h in (34.6) 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 (34.6) by

$$h = c + \beta \int v^*(s')q(s')ds'. \tag{34.9}$$

Equation (34.7) becomes

$$h = c + \beta \int \max\left\{\frac{w(s')}{1-\beta}, h\right\} q(s')ds' \tag{34.10}$$

The aim is to solve this nonlinear equation by iteration, and from it obtain the reservation wage.

Try to carry this out, setting

- the state sequence $\{s_t\}$ to be IID and standard normal and
- the wage function to be $w(s) = \exp(\mu + \sigma s)$.

You will need to implement a new version of the `McCallModel` class that assumes a lognormal wage distribution.

Calculate the integral by Monte Carlo, by averaging over a large number of wage draws.

For default parameters, use `c=25, β=0.99, σ=0.5, μ=2.5`.

Once your code is working, investigate how the reservation wage changes with c and β.

Solution to Exercise 34.5.2

Here is one solution:

```
mccall_data_continuous = [
    ('c', float64),          # unemployment compensation
    ('β', float64),          # discount factor
    ('σ', float64),          # scale parameter in lognormal distribution
    ('μ', float64),          # location parameter in lognormal distribution
    ('w_draws', float64[:])  # draws of wages for Monte Carlo
]

@jitclass(mccall_data_continuous)
class McCallModelContinuous:

    def __init__(self, c=25, β=0.99, σ=0.5, μ=2.5, mc_size=1000):

        self.c, self.β, self.σ, self.μ = c, β, σ, μ

        # Draw and store shocks
        np.random.seed(1234)
        s = np.random.randn(mc_size)
        self.w_draws = np.exp(μ+ σ * s)


@jit(nopython=True)
def compute_reservation_wage_continuous(mcmc, max_iter=500, tol=1e-5):

    c, β, σ, μ, w_draws = mcmc.c, mcmc.β, mcmc.σ, mcmc.μ, mcmc.w_draws

    h = np.mean(w_draws) / (1 - β)  # initial guess
    i = 0
    error = tol + 1
    while i < max_iter and error > tol:

        integral = np.mean(np.maximum(w_draws / (1 - β), h))
        h_next = c + β * integral
```

(continues on next page)

(continued from previous page)

```
        error = np.abs(h_next - h)
        i += 1

        h = h_next

    # == Now compute the reservation wage == #

    return (1 - β) * h
```

Now we investigate how the reservation wage changes with c and β.

We will do this using a contour plot.

```
grid_size = 25
R = np.empty((grid_size, grid_size))

c_vals = np.linspace(10.0, 30.0, grid_size)
β_vals = np.linspace(0.9, 0.99, grid_size)

for i, c in enumerate(c_vals):
    for j, β in enumerate(β_vals):
        mcmc = McCallModelContinuous(c=c, β=β)
        R[i, j] = compute_reservation_wage_continuous(mcmc)
```

```
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)


ax.set_title("reservation wage")
ax.set_xlabel("$c$", fontsize=16)
ax.set_ylabel("$β$", fontsize=16)

ax.ticklabel_format(useOffset=False)

plt.show()
```

reservation wage
0.99
0.98
0.97
0.96
0.95
0.94
0.93
0.92
0.91
0.90
β
10.0
12.5
15.0
17.5
20.0
22.5
25.0
27.5
30.0
c
35.0
32.5
30.0
27.5
25.0
22.5
37.5
35.0
32.5
30.0
27.5
25.0
22.5
20.0
17.5

CHAPTER

THIRTYFIVE

JOB SEARCH II: SEARCH AND SEPARATION

Contents

- *Job Search II: Search and Separation*
  - *Overview*
  - *The Model*
  - *Solving the Model*
  - *Implementation*
  - *Impact of Parameters*
  - *Exercises*

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install quantecon
```

35.1 Overview

Previously *we looked* at the McCall job search model [McC70] as a way of understanding unemployment and worker decisions.

One unrealistic feature of the model is that every job is permanent.

In this lecture, we extend the McCall 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
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
from numba import njit, float64
```

(continues on next page)

(continued from previous page)

```
from numba.experimental import jitclass
from quantecon.distributions import BetaBinomial
```

35.2 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

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t u(y_t) \tag{35.1}$$

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 $u'' < 0$.

35.2.1 The Wage Process

For now we will drop the separation of state process and wage process that we maintained for the *baseline model*.

In particular, we simply suppose that wage offers $\{w_t\}$ are IID with common distribution q.

The set of possible wage values is denoted by $\mathbb{W}$.

(Later we will go back to having a separate state process $\{s_t\}$ driving random outcomes, since this formulation is usually convenient in more sophisticated models.)

35.2.2 Timing and Decisions

At the start of each period, the agent can be either

- unemployed or
- employed at some existing wage level w_e.

At the start of a given period, the current wage offer w_t is observed.

If currently *employed*, the worker

1. receives utility $u(w_e)$ and
2. is fired with some (small) probability α.

If currently *unemployed*, the worker either accepts or rejects the current offer w_t.

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*.

35.3 Solving the Model

We drop time subscripts in what follows and primes denote next period values.

Let

- $v(w_e)$ be total lifetime value accruing to a worker who enters the current period *employed* with existing wage w_e
- $h(w)$ be total lifetime value accruing to a worker who who enters the current period *unemployed* and receives wage offer w.

Here *value* means the value of the objective function (35.1) when the worker makes optimal decisions at all future points in time.

Our first aim is to obtain these functions.

35.3.1 The Bellman Equations

Suppose for now that the worker can calculate the functions v and h and use them in his decision making.

Then v and h should satisfy

$$v(w_e) = u(w_e) + \beta \left[(1-\alpha) v(w_e) + \alpha \sum_{w' \in \mathbb{W}} h(w') q(w') \right] \tag{35.2}$$

and

$$h(w) = \max \left\{ v(w),\, u(c) + \beta \sum_{w' \in \mathbb{W}} h(w') q(w') \right\} \tag{35.3}$$

Equation (35.2) expresses the value of being employed at wage w_e in terms of

- current reward $u(w_e)$ plus
- discounted expected reward tomorrow, given the α probability of being fired

Equation (35.3) 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.

Accepting transitions the worker to employment and hence yields reward $v(w)$.

Rejecting leads to unemployment compensation and unemployment tomorrow.

Equations (35.2) and (35.3) are the Bellman equations for this model.

They provide enough information to solve for both v and h.

35.3.2 A Simplifying Transformation

Rather than jumping straight into solving these equations, let's see if we can simplify them somewhat.

(This process will be analogous to our *second pass* at the plain vanilla McCall model, where we simplified the Bellman equation.)

First, let

$$d := \sum_{w' \in \mathbb{W}} h(w')q(w') \tag{35.4}$$

be the expected value of unemployment tomorrow.

We can now write (35.3) as

$$h(w) = \max\{v(w),\, u(c) + \beta d\}$$

or, shifting time forward one period

$$\sum_{w' \in \mathbb{W}} h(w')q(w') = \sum_{w' \in \mathbb{W}} \max\{v(w'),\, u(c) + \beta d\}\, q(w')$$

Using (35.4) again now gives

$$d = \sum_{w' \in \mathbb{W}} \max\{v(w'),\, u(c) + \beta d\}\, q(w') \tag{35.5}$$

Finally, (35.2) can now be rewritten as

$$v(w) = u(w) + \beta\left[(1-\alpha)v(w) + \alpha d\right] \tag{35.6}$$

In the last expression, we wrote w_e as w to make the notation simpler.

35.3.3 The Reservation Wage

Suppose we can use (35.5) and (35.6) to solve for d and v.

(We will do this soon.)

We can then determine optimal behavior for the worker.

From (35.3), we see that an unemployed agent accepts current offer w if $v(w) \geq u(c) + \beta d$.

This means precisely that the value of accepting is higher than the expected value of rejecting.

It is clear that v is (at least weakly) increasing in w, since the agent is never made worse off by a higher wage offer.

Hence, we can express the optimal choice as accepting wage offer w if and only if

$$w \geq \bar{w} \quad \text{where} \quad \bar{w} \text{ solves } v(\bar{w}) = u(c) + \beta d$$

35.3.4 Solving the Bellman Equations

We'll use the same iterative approach to solving the Bellman equations that we adopted in the *first job search lecture*.

Here this amounts to

1. make guesses for d and v
2. plug these guesses into the right-hand sides of (35.5) and (35.6)

3. update the left-hand sides from this rule and then repeat

In other words, we are iterating using the rules

$$d_{n+1} = \sum_{w' \in \mathbb{W}} \max\left\{v_n(w'),\, u(c) + \beta d_n\right\} q(w') \tag{35.7}$$

$$v_{n+1}(w) = u(w) + \beta\left[(1-\alpha)v_n(w) + \alpha d_n\right] \tag{35.8}$$

starting from some initial conditions d_0, v_0.

As before, the system always converges to the true solutions—in this case, the v and d that solve (35.5) and (35.6).

(A proof can be obtained via the Banach contraction mapping theorem.)

35.4 Implementation

Let's implement this iterative process.

In the code, you'll see that we use a class to store the various parameters and other objects associated with a given model.

This helps to tidy up the code and provides an object that's easy to pass to functions.

The default utility function is a CRRA utility function

```
@njit
def u(c, σ=2.0):
    return (c**(1 - σ) - 1) / (1 - σ)
```

Also, here's a default wage distribution, based around the BetaBinomial distribution:

```
n = 60                                  # n possible outcomes for w
w_default = np.linspace(10, 20, n)      # wages between 10 and 20
a, b = 600, 400                         # shape parameters
dist = BetaBinomial(n-1, a, b)
q_default = dist.pdf()
```

Here's our jitted class for the McCall model with separation.

```
mccall_data = [
    ('α', float64),      # job separation rate
    ('β', float64),      # discount factor
    ('c', float64),      # unemployment compensation
    ('w', float64[:]),   # list of wage values
    ('q', float64[:])    # pmf of random variable w
]

@jitclass(mccall_data)
class McCallModel:
    """
    Stores the parameters and functions associated with a given model.
    """

    def __init__(self, α=0.2, β=0.98, c=6.0, w=w_default, q=q_default):

        self.α, self.β, self.c, self.w, self.q = α, β, c, w, q
```

(continues on next page)

(continued from previous page)

```
    def update(self, v, d):

        α, β, c, w, q = self.α, self.β, self.c, self.w, self.q

        v_new = np.empty_like(v)

        for i in range(len(w)):
            v_new[i] = u(w[i]) + β * ((1 - α) * v[i] + α * d)

        d_new = np.sum(np.maximum(v, u(c) + β * d) * q)

        return v_new, d_new
```

Now we iterate until successive realizations are closer together than some small tolerance level.

We then return the current iterate as an approximate solution.

```
@njit
def solve_model(mcm, tol=1e-5, max_iter=2000):
    """
    Iterates to convergence on the Bellman equations

    * mcm is an instance of McCallModel
    """

    v = np.ones_like(mcm.w)     # Initial guess of v
    d = 1                       # Initial guess of d
    i = 0
    error = tol + 1

    while error > tol and i < max_iter:
        v_new, d_new = mcm.update(v, d)
        error_1 = np.max(np.abs(v_new - v))
        error_2 = np.abs(d_new - d)
        error = max(error_1, error_2)
        v = v_new
        d = d_new
        i += 1

    return v, d
```

35.4.1 The Reservation Wage: First Pass

The optimal choice of the agent is summarized by the reservation wage.

As discussed above, the reservation wage is the $\bar{w}$ that solves $v(\bar{w}) = h$ where $h := u(c) + \beta d$ is the continuation value.

Let's compare v and h to see what they look like.

We'll use the default parameterizations found in the code above.

```
mcm = McCallModel()
v, d = solve_model(mcm)
h = u(mcm.c) + mcm.β * d
```

(continues on next page)

(continued from previous page)

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

ax.plot(mcm.w, v, 'b-', lw=2, alpha=0.7, label='$v$')
ax.plot(mcm.w, [h] * len(mcm.w),
        'g-', lw=2, alpha=0.7, label='$h$')
ax.set_xlim(min(mcm.w), max(mcm.w))
ax.legend()

plt.show()
```

The value v is increasing because higher w generates a higher wage flow conditional on staying employed.

35.4.2 The Reservation Wage: Computation

Here's a function `compute_reservation_wage` that takes an instance of `McCallModel` and returns the associated reservation wage.

```
@njit
def compute_reservation_wage(mcm):
    """
    Computes the reservation wage of an instance of the McCall model
    by finding the smallest w such that v(w) >= h.

    If no such w exists, then w_bar is set to np.inf.
    """

    v, d = solve_model(mcm)
    h = u(mcm.c) + mcm.β * d

    i = np.searchsorted(v, h, side='right')
    w_bar = mcm.w[i]

    return w_bar
```

Next we will investigate how the reservation wage varies with parameters.

35.5 Impact of Parameters

In each instance below, we'll show you a figure and then ask you to reproduce it in the exercises.

35.5.1 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 `McCallModel` 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.

35.5.2 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.

35.5.3 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.

35.6 Exercises

Exercise 35.6.1

Reproduce all the reservation wage figures shown above.

Regarding the values on the horizontal axis, use

```
grid_size = 25
c_vals = np.linspace(2, 12, grid_size)          # unemployment compensation
beta_vals = np.linspace(0.8, 0.99, grid_size)   # discount factors
alpha_vals = np.linspace(0.05, 0.5, grid_size)  # separation rate
```

Solution to Exercise 35.6.1

Here's the first figure.

```
mcm = McCallModel()

w_bar_vals = np.empty_like(c_vals)

fig, ax = plt.subplots()

for i, c in enumerate(c_vals):
    mcm.c = c
    w_bar = compute_reservation_wage(mcm)
    w_bar_vals[i] = w_bar

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()
```

Here's the second one.

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

for i, β in enumerate(beta_vals):
    mcm.β = β
    w_bar = compute_reservation_wage(mcm)
    w_bar_vals[i] = w_bar

ax.set(xlabel='discount factor', ylabel='reservation wage')
ax.plot(beta_vals, w_bar_vals, label=r'$\bar w$ as a function of $\beta$')
ax.legend()

plt.show()
```

Here's the third.

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

for i, α in enumerate(alpha_vals):
    mcm.α = α
    w_bar = compute_reservation_wage(mcm)
    w_bar_vals[i] = w_bar

ax.set(xlabel='separation rate', ylabel='reservation wage')
ax.plot(alpha_vals, w_bar_vals, label=r'$\bar w$ as a function of $\alpha$')
ax.legend()

plt.show()
```

CHAPTER

THIRTYSIX

JOB SEARCH III: FITTED VALUE FUNCTION ITERATION

Contents

- *Job Search III: Fitted Value Function Iteration*
  - *Overview*
  - *The Algorithm*
  - *Implementation*
  - *Exercises*

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install interpolation
```

36.1 Overview

In this lecture we again study the *McCall job search model with separation*, but now with a continuous wage distribution.

While we already considered continuous wage distributions briefly in the exercises of the *first job search lecture*, the change was relatively trivial in that case.

This is because we were able to reduce the problem to solving for a single scalar value (the continuation value).

Here, with separation, 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.

We will use the following imports:

```
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
from interpolation import interp
```

(continues on next page)

(continued from previous page)

```
from numba import njit, float64
from numba.experimental import jitclass
```

```
/opt/conda/envs/quantecon/lib/python3.11/site-packages/numba/core/decorators.
↪py:262: NumbaDeprecationWarning: numba.generated_jit is deprecated. Please see␣
↪the documentation at: https://numba.readthedocs.io/en/stable/reference/
↪deprecation.html#deprecation-of-generated-jit for more information and advice on␣
↪a suitable replacement.
  warnings.warn(msg, NumbaDeprecationWarning)
```

36.2 The Algorithm

The model is the same as the McCall model with job separation we *studied before*, except that the wage offer distribution is continuous.

We are going to start with the two Bellman equations we obtained for the model with job separation after *a simplifying transformation*.

Modified to accommodate continuous wage draws, they take the following form:

$$d = \int \max\left\{v(w'),\, u(c) + \beta d\right\} q(w') dw' \tag{36.1}$$

and

$$v(w) = u(w) + \beta\left[(1-\alpha)v(w) + \alpha d\right] \tag{36.2}$$

The unknowns here are the function v and the scalar d.

The difference between these and the pair of Bellman equations we previously worked on are

1. in (36.1), what used to be a sum over a finite number of wage values is an integral over an infinite set.
2. The function v in (36.2) is defined over all $w \in \mathbb{R}_+$.

The function q in (36.1) is the density of the wage offer distribution.

Its support is taken as equal to $\mathbb{R}_+$.

36.2.1 Value Function Iteration

In theory, we should now proceed as follows:

1. Begin with a guess v, d for the solutions to (36.1)–(36.2).
2. Plug v, d into the right hand side of (36.1)–(36.2) and compute the left hand side to obtain updates v', d'
3. Unless some stopping condition is satisfied, set $(v, d) = (v', d')$ 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 (36.2).

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.

36.2.2 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., [Gor95] or [Sta08]) and
2. preserves useful shape properties such as monotonicity and concavity/convexity.

Linear interpolation will be implemented using a JIT-aware Python interpolation library called interpolation.py.

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 * np.cos(6 * x) + np.sin(14 * x)
    return y1 + 2.5

c_grid = np.linspace(0, 1, 6)
f_grid = np.linspace(0, 1, 150)

def Af(x):
    return interp(c_grid, f(c_grid), x)

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()
```

36.3 Implementation

The first step is to build a jitted class for the McCall model with separation and a continuous wage offer distribution.

We will take the utility function to be the log function for this application, with $u(c) = \ln c$.

We will adopt the lognormal distribution for wages, with $w = \exp(\mu + \sigma z)$ when z is standard normal and μ, σ are parameters.

```
@njit
def lognormal_draws(n=1000, μ=2.5, σ=0.5, seed=1234):
    np.random.seed(seed)
    z = np.random.randn(n)
    w_draws = np.exp(μ + σ * z)
    return w_draws
```

Here's our class.

```
mccall_data_continuous = [
    ('c', float64),          # unemployment compensation
    ('α', float64),          # job separation rate
    ('β', float64),          # discount factor
    ('w_grid', float64[:]),  # grid of points for fitted VFI
    ('w_draws', float64[:])  # draws of wages for Monte Carlo
]

@jitclass(mccall_data_continuous)
class McCallModelContinuous:

    def __init__(self,
                 c=1,
                 α=0.1,
                 β=0.96,
                 grid_min=1e-10,
                 grid_max=5,
```

(continues on next page)

(continued from previous page)

```
                 grid_size=100,
                 w_draws=lognormal_draws()):

        self.c, self.α, self.β = c, α, β

        self.w_grid = np.linspace(grid_min, grid_max, grid_size)
        self.w_draws = w_draws

    def update(self, v, d):

        # Simplify names
        c, α, β = self.c, self.α, self.β
        w = self.w_grid
        u = lambda x: np.log(x)

        # Interpolate array represented value function
        vf = lambda x: interp(w, v, x)

        # Update d using Monte Carlo to evaluate integral
        d_new = np.mean(np.maximum(vf(self.w_draws), u(c) + β * d))

        # Update v
        v_new = u(w) + β * ((1 - α) * v + α * d)

        return v_new, d_new
```

We then return the current iterate as an approximate solution.

```
@njit
def solve_model(mcm, tol=1e-5, max_iter=2000):
    """
    Iterates to convergence on the Bellman equations

    * mcm is an instance of McCallModel
    """

    v = np.ones_like(mcm.w_grid)    # Initial guess of v
    d = 1                           # Initial guess of d
    i = 0
    error = tol + 1

    while error > tol and i < max_iter:
        v_new, d_new = mcm.update(v, d)
        error_1 = np.max(np.abs(v_new - v))
        error_2 = np.abs(d_new - d)
        error = max(error_1, error_2)
        v = v_new
        d = d_new
        i += 1

    return v, d
```

Here's a function `compute_reservation_wage` that takes an instance of `McCallModelContinuous` and returns the associated reservation wage.

If $v(w) < h$ for all w, then the function returns np.inf

```
@njit
def compute_reservation_wage(mcm):
    """
    Computes the reservation wage of an instance of the McCall model
    by finding the smallest w such that v(w) >= h.

    If no such w exists, then w_bar is set to np.inf.
    """
    u = lambda x: np.log(x)

    v, d = solve_model(mcm)
    h = u(mcm.c) + mcm.β * d

    w_bar = np.inf
    for i, wage in enumerate(mcm.w_grid):
        if v[i] > h:
            w_bar = wage
            break

    return w_bar
```

The exercises ask you to explore the solution and how it changes with parameters.

36.4 Exercises

Exercise 36.4.1

Use the code above to explore what happens to the reservation wage when the wage parameter μ changes.

Use the default parameters and μ in `mu_vals = np.linspace(0.0, 2.0, 15)`.

Is the impact on the reservation wage as you expected?

Solution to Exercise 36.4.1

Here is one solution

```
mcm = McCallModelContinuous()
mu_vals = np.linspace(0.0, 2.0, 15)
w_bar_vals = np.empty_like(mu_vals)

fig, ax = plt.subplots()

for i, m in enumerate(mu_vals):
    mcm.w_draws = lognormal_draws(μ=m)
    w_bar = compute_reservation_wage(mcm)
    w_bar_vals[i] = w_bar

ax.set(xlabel='mean', ylabel='reservation wage')
ax.plot(mu_vals, w_bar_vals, label=r'$\bar w$ as a function of $\mu$')
ax.legend()

plt.show()
```

Not surprisingly, the agent is more inclined to wait when the distribution of offers shifts to the right.

Exercise 36.4.2

Let us now consider how the agent responds to an increase in volatility.

To try to understand this, compute the reservation wage when the wage offer distribution is uniform on $(m - s, m + s)$ and s varies.

The idea here is that we are holding the mean constant and spreading the support.

(This is a form of *mean-preserving spread*.)

Use `s_vals = np.linspace(1.0, 2.0, 15)` and `m = 2.0`.

State how you expect the reservation wage to vary with s.

Now compute it. Is this as you expected?

Solution to Exercise 36.4.2

Here is one solution

```
mcm = McCallModelContinuous()
s_vals = np.linspace(1.0, 2.0, 15)
m = 2.0
w_bar_vals = np.empty_like(s_vals)

fig, ax = plt.subplots()

for i, s in enumerate(s_vals):
    a, b = m - s, m + s
    mcm.w_draws = np.random.uniform(low=a, high=b, size=10_000)
    w_bar = compute_reservation_wage(mcm)
    w_bar_vals[i] = w_bar
```

(continues on next page)

(continued from previous page)

```
ax.set(xlabel='volatility', ylabel='reservation wage')
ax.plot(s_vals, w_bar_vals, label=r'$\bar w$ as a function of wage volatility')
ax.legend()

plt.show()
```

The reservation wage increases with volatility.

One might think that higher volatility would make the agent more inclined to take a given offer, since doing so represents certainty and waiting represents risk.

But job search is like holding an option: the worker is only exposed to upside risk (since, in a free market, no one can force them to take a bad offer).

More volatility means higher upside potential, which encourages the agent to wait.

CHAPTER

THIRTYSEVEN

JOB SEARCH IV: CORRELATED WAGE OFFERS

Contents

- *Job Search IV: Correlated Wage Offers*
  - *Overview*
  - *The Model*
  - *Implementation*
  - *Unemployment Duration*
  - *Exercises*

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install quantecon
!pip install interpolation
```

37.1 Overview

In this lecture we solve a *McCall style job search model* with persistent and transitory components to wages.

In other words, we relax the unrealistic assumption that randomness in wages is independent over time.

At the same time, we will go back to assuming that jobs are permanent and no separation occurs.

This is to keep the model relatively simple as we study the impact of correlation.

We will use the following imports:

```
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
import quantecon as qe
from interpolation import interp
from numpy.random import randn
from numba import njit, prange, float64
from numba.experimental import jitclass
```

```
/opt/conda/envs/quantecon/lib/python3.11/site-packages/numba/core/decorators.
↪py:262: NumbaDeprecationWarning: numba.generated_jit is deprecated. Please see␣
↪the documentation at: https://numba.readthedocs.io/en/stable/reference/
↪deprecation.html#deprecation-of-generated-jit for more information and advice on␣
↪a suitable replacement.
  warnings.warn(msg, NumbaDeprecationWarning)
```

37.2 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 express, u is a utility function and $\mathbb{E}_z$ is 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.

37.2.1 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 $u(c) = \ln(c)$.

The reservation wage is the wage where equality holds in the last expression.

That is,

$$\bar{w}(z) := \exp(f^*(z)(1-\beta)) \tag{37.1}$$

Our main aim is to solve for the reservation rule and study its properties and implications.

37.3 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.

The following list helps Numba by providing some type information about the data we will work with.

```
job_search_data = [
     ('μ', float64),              # transient shock log mean
     ('s', float64),              # transient shock log variance
     ('d', float64),              # shift coefficient of persistent state
     ('ρ', float64),              # correlation coefficient of persistent state
     ('σ', float64),              # state volatility
     ('β', float64),              # discount factor
     ('c', float64),              # unemployment compensation
     ('z_grid', float64[:]),      # grid over the state space
     ('e_draws', float64[:,:])    # Monte Carlo draws for integration
]
```

Here's a class that stores the data and the right hand side of the Bellman equation.

Default parameter values are embedded in the class.

```
@jitclass(job_search_data)
class JobSearch:

    def __init__(self,
                 μ=0.0,        # transient shock log mean
```

(continues on next page)

(continued from previous page)

```
                 s=1.0,        # transient shock log variance
                 d=0.0,        # shift coefficient of persistent state
                 ρ=0.9,        # correlation coefficient of persistent state
                 σ=0.1,        # state volatility
                 β=0.98,       # discount factor
                 c=5,          # unemployment compensation
                 mc_size=1000,
                 grid_size=100):

        self.μ, self.s, self.d,  = μ, s, d,
        self.ρ, self.σ, self.β, self.c = ρ, σ, β, c

        # Set up grid
        z_mean = d / (1 - ρ)
        z_sd = σ / np.sqrt(1 - ρ**2)
        k = 3  # std devs from mean
        a, b = z_mean - k * z_sd, z_mean + k * z_sd
        self.z_grid = np.linspace(a, b, grid_size)

        # Draw and store shocks
        np.random.seed(1234)
        self.e_draws = randn(2, mc_size)

    def parameters(self):
        """
        Return all parameters as a tuple.
        """
        return self.μ, self.s, self.d, \
                self.ρ, self.σ, self.β, self.c
```

Next we implement the Q operator.

```
@njit(parallel=True)
def Q(js, f_in, f_out):
    """
    Apply the operator Q.

        * js is an instance of JobSearch
        * f_in and f_out are arrays that represent f and Qf respectively

    """

    μ, s, d, ρ, σ, β, c = js.parameters()
    M = js.e_draws.shape[1]

    for i in prange(len(js.z_grid)):
        z = js.z_grid[i]
        expectation = 0.0
        for m in range(M):
            e1, e2 = js.e_draws[:, m]
            z_next = d + ρ * z + σ * e1
            go_val = interp(js.z_grid, f_in, z_next)      # f(z')
            y_next = np.exp(μ + s * e2)                   # y' draw
            w_next = np.exp(z_next) + y_next              # w' draw
            stop_val = np.log(w_next) / (1 - β)
            expectation += max(stop_val, go_val)
```

(continues on next page)

(continued from previous page)

```
        expectation = expectation / M
        f_out[i] = np.log(c) + β * expectation
```

Here's a function to compute an approximation to the fixed point of Q.

```
def compute_fixed_point(js,
                        use_parallel=True,
                        tol=1e-4,
                        max_iter=1000,
                        verbose=True,
                        print_skip=25):

    f_init = np.full(len(js.z_grid), np.log(js.c))
    f_out = np.empty_like(f_init)

    # Set up loop
    f_in = f_init
    i = 0
    error = tol + 1

    while i < max_iter and error > tol:
        Q(js, f_in, f_out)
        error = np.max(np.abs(f_in - f_out))
        i += 1
        if verbose and i % print_skip == 0:
            print(f"Error at iteration {i} is {error}.")
        f_in[:] = f_out

    if error > tol:
        print("Failed to converge!")
    elif verbose:
        print(f"\nConverged in {i} iterations.")

    return f_out
```

Let's try generating an instance and solving the model.

```
js = JobSearch()

qe.tic()
f_star = compute_fixed_point(js, verbose=True)
qe.toc()
```

```
Error at iteration 25 is 0.576247783958749.
```

```
Error at iteration 50 is 0.11808817939665062.
```

```
Error at iteration 75 is 0.0285774413852522.
```

```
Error at iteration 100 is 0.007158336385160169.
```

```
Error at iteration 125 is 0.0018027870994501427.
```

```
Error at iteration 150 is 0.00045489087412420304.
```

```
Error at iteration 175 is 0.00011479050300522431.

Converged in 178 iterations.
TOC: Elapsed: 0:00:5.02
```

```
5.027941942214966
```

Next we will compute and plot the reservation wage function defined in (37.1).

```
res_wage_function = np.exp(f_star * (1 - js.β))

fig, ax = plt.subplots()
ax.plot(js.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 look at its impact on the reservation wage:

```
c_vals = 1, 2, 3

fig, ax = plt.subplots()

for c in c_vals:
    js = JobSearch(c=c)
    f_star = compute_fixed_point(js, verbose=False)
```

(continues on next page)

(continued from previous page)

```
    res_wage_function = np.exp(f_star * (1 - js.β))
    ax.plot(js.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.

37.4 Unemployment Duration

Next we study how mean unemployment duration varies with unemployment compensation.

For simplicity we'll fix the initial state at $z_t = 0$.

```
def compute_unemployment_duration(js, seed=1234):

    f_star = compute_fixed_point(js, verbose=False)
    μ, s, d, ρ, σ, β, c = js.parameters()
    z_grid = js.z_grid
    np.random.seed(seed)

    @njit
    def f_star_function(z):
        return interp(z_grid, f_star, z)

    @njit
    def draw_tau(t_max=10_000):
        z = 0
        t = 0

        unemployed = True
        while unemployed and t < t_max:
            # draw current wage
```

(continues on next page)

(continued from previous page)

```
            y = np.exp(μ + s * np.random.randn())
            w = np.exp(z) + y
            res_wage = np.exp(f_star_function(z) * (1 - β))
            # if optimal to stop, record t
            if w >= res_wage:
                unemployed = False
                τ = t
            # else increment data and state
            else:
                z = ρ * z + d + σ * np.random.randn()
                t += 1
        return τ

    @njit(parallel=True)
    def compute_expected_tau(num_reps=100_000):
        sum_value = 0
        for i in prange(num_reps):
            sum_value += draw_tau()
        return sum_value / num_reps

    return compute_expected_tau()
```

Let's test this out with some possible values for unemployment compensation.

```
c_vals = np.linspace(1.0, 10.0, 8)
durations = np.empty_like(c_vals)
for i, c in enumerate(c_vals):
    js = JobSearch(c=c)
    τ = compute_unemployment_duration(js)
    durations[i] = τ
```

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.

37.5 Exercises

Exercise 37.5.1

Investigate how mean unemployment duration varies with the discount factor β.

- What is your prior expectation?
- Do your results match up?

Solution to Exercise 37.5.1

Here is one solution

```
beta_vals = np.linspace(0.94, 0.99, 8)
durations = np.empty_like(beta_vals)
for i, β in enumerate(beta_vals):
    js = JobSearch(β=β)
    τ = compute_unemployment_duration(js)
    durations[i] = τ
```

```
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

THIRTYEIGHT

JOB SEARCH V: MODELING CAREER CHOICE

Contents

- *Job Search V: Modeling Career Choice*
  - *Overview*
  - *Model*
  - *Implementation*
  - *Exercises*

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install quantecon
```

38.1 Overview

Next, we study a computational problem concerning career and job choices.

The model is originally due to Derek Neal [Nea99].

This exposition draws on the presentation in [LS18], section 6.5.

We begin with some imports:

```
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
import quantecon as qe
from numba import njit, prange
from quantecon.distributions import BetaBinomial
from scipy.special import binom, beta
from mpl_toolkits.mplot3d.axes3d import Axes3D
from matplotlib import cm
```

38.1.1 Model Features

- Career and job within career both chosen to maximize expected discounted wage flow.
- Infinite horizon dynamic programming with two state variables.

38.2 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

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t w_t \tag{38.1}$$

subject to the choice restrictions specified above.

Let $v(\theta, \epsilon)$ denote the value function, which is the maximum of (38.1) overall feasible (career, job) policies, given the initial state (θ, ϵ).

The value function obeys

$$v(\theta, \epsilon) = \max\{I, II, III\}$$

where

$$\begin{aligned}
I &= \theta + \epsilon + \beta v(\theta, \epsilon) \\
II &= \theta + \int \epsilon' G(d\epsilon') + \beta \int v(\theta, \epsilon') G(d\epsilon') \\
III &= \int \theta' F(d\theta') + \int \epsilon' G(d\epsilon') + \beta \int \int v(\theta', \epsilon') G(d\epsilon') F(d\theta')
\end{aligned}$$

Evidently I, II and III correspond to "stay put", "new job" and "new life", respectively.

38.2.1 Parameterization

As in [LS18], 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 `np.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 `np.linspace(0, B, grid_size)`.

A very useful family of discrete distributions is the Beta-binomial family, with probability mass function

$$p(k \mid n, a, b) = \binom{n}{k} \frac{B(k+a, n-k+b)}{B(a,b)}, \qquad k = 0, \dots, n$$

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 $n = 50$.

```
def gen_probs(n, a, b):
    probs = np.zeros(n+1)
    for k in range(n+1):
        probs[k] = binom(n, k) * beta(k + a, n - k + b) / beta(a, b)
    return probs

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(list(range(0, n+1)), gen_probs(n, a, b), '-o', label=ab_label)
ax.legend()
plt.show()
```

38.3 Implementation

We will first create a class `CareerWorkerProblem` which will hold the default parameterizations of the model and an initial guess for the value function.

```
class CareerWorkerProblem:

    def __init__(self,
                 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):

        self.β, self.grid_size, self.B = β, grid_size, B

        self.θ = np.linspace(0, B, grid_size)      # Set of θ values
        self.ϵ = np.linspace(0, B, grid_size)      # Set of ϵ values

        self.F_probs = BetaBinomial(grid_size - 1, F_a, F_b).pdf()
        self.G_probs = BetaBinomial(grid_size - 1, G_a, G_b).pdf()
        self.F_mean = np.sum(self.θ * self.F_probs)
        self.G_mean = np.sum(self.ϵ * self.G_probs)

        # Store these parameters for str and repr methods
        self._F_a, self._F_b = F_a, F_b
        self._G_a, self._G_b = G_a, G_b
```

The following function takes an instance of `CareerWorkerProblem` and returns the corresponding Bellman operator T and the greedy policy function.

In this model, T is defined by $Tv(\theta, \epsilon) = \max\{I, II, III\}$, where I, II and III are as given in (38.2).

```
def operator_factory(cw, parallel_flag=True):

    """
    Returns jitted versions of the Bellman operator and the
    greedy policy function

    cw is an instance of ``CareerWorkerProblem``
    """

    θ, ϵ, β = cw.θ, cw.ϵ, cw.β
    F_probs, G_probs = cw.F_probs, cw.G_probs
    F_mean, G_mean = cw.F_mean, cw.G_mean

    @njit(parallel=parallel_flag)
    def T(v):
        "The Bellman operator"

        v_new = np.empty_like(v)

        for i in prange(len(v)):
            for j in prange(len(v)):
                v1 = θ[i] + ϵ[j] + β * v[i, j]                    # Stay put
                v2 = θ[i] + G_mean + β * v[i, :] @ G_probs        # New job
                v3 = G_mean + F_mean + β * F_probs @ v @ G_probs  # New life
                v_new[i, j] = max(v1, v2, v3)

        return v_new

    @njit
    def get_greedy(v):
        "Computes the v-greedy policy"

        σ = np.empty(v.shape)

        for i in range(len(v)):
            for j in range(len(v)):
                v1 = θ[i] + ϵ[j] + β * v[i, j]
                v2 = θ[i] + G_mean + β * v[i, :] @ G_probs
                v3 = G_mean + F_mean + β * F_probs @ v @ G_probs
                if v1 > max(v2, v3):
                    action = 1
                elif v2 > max(v1, v3):
                    action = 2
                else:
                    action = 3
                σ[i, j] = action

        return σ

    return T, get_greedy
```

Lastly, `solve_model` will take an instance of `CareerWorkerProblem` and iterate using the Bellman operator to find the fixed point of the Bellman equation.

```
def solve_model(cw,
                use_parallel=True,
                tol=1e-4,
                max_iter=1000,
                verbose=True,
                print_skip=25):

    T, _ = operator_factory(cw, parallel_flag=use_parallel)

    # Set up loop
    v = np.full((cw.grid_size, cw.grid_size), 100.)  # Initial guess
    i = 0
    error = tol + 1

    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
```

Here's the solution to the model – an approximate value function

```
cw = CareerWorkerProblem()
T, get_greedy = operator_factory(cw)
v_star = solve_model(cw, verbose=False)
greedy_star = get_greedy(v_star)

fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111, projection='3d')
tg, eg = np.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()
```

And here is the optimal policy

```
fig, ax = plt.subplots(figsize=(6, 6))
tg, eg = np.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 variables to be large, and the worker cannot change careers without changing jobs.

- Sometimes a good job must be sacrificed in order to change to a better career.

38.4 Exercises

Exercise 38.4.1

Using the default parameterization in the class `CareerWorkerProblem`, 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 generate the draws from the distributions F and G, use `quantecon.random.draw()`.

Solution to Exercise 38.4.1

Simulate job/career paths.

In reading the code, recall that `optimal_policy[i, j]` = policy at (θ_i, ϵ_j) = either 1, 2 or 3; meaning 'stay put', 'new job' and 'new life'.

```
F = np.cumsum(cw.F_probs)
G = np.cumsum(cw.G_probs)
v_star = solve_model(cw, verbose=False)
T, get_greedy = operator_factory(cw)
greedy_star = get_greedy(v_star)

def gen_path(optimal_policy, F, G, t=20):
    i = j = 0
    θ_index = []
    ϵ_index = []
    for t in range(t):
        if optimal_policy[i, j] == 1:       # Stay put
            pass

        elif greedy_star[i, j] == 2:     # New job
            j = qe.random.draw(G)

        else:                            # New life
            i, j = qe.random.draw(F), qe.random.draw(G)
        θ_index.append(i)
        ϵ_index.append(j)
    return cw.θ[θ_index], cw.ϵ[ϵ_index]


fig, axes = plt.subplots(2, 1, figsize=(10, 8))
for ax in axes:
    θ_path, ϵ_path = gen_path(greedy_star, F, G)
    ax.plot(ϵ_path, label='ϵ')
    ax.plot(θ_path, label='θ')
    ax.set_ylim(0, 6)

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

Exercise 38.4.2

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 to Exercise 38.4.2

The median for the original parameterization can be computed as follows

```
cw = CareerWorkerProblem()
F = np.cumsum(cw.F_probs)
G = np.cumsum(cw.G_probs)
T, get_greedy = operator_factory(cw)
v_star = solve_model(cw, verbose=False)
greedy_star = get_greedy(v_star)

@njit
def passage_time(optimal_policy, F, G):
    t = 0
    i = j = 0
    while True:
        if optimal_policy[i, j] == 1:    # Stay put
            return t
        elif optimal_policy[i, j] == 2:  # New job
            j = qe.random.draw(G)
        else:                            # New life
            i, j  = qe.random.draw(F), qe.random.draw(G)
        t += 1

@njit(parallel=True)
def median_time(optimal_policy, F, G, M=25000):
    samples = np.empty(M)
    for i in prange(M):
        samples[i] = passage_time(optimal_policy, F, G)
    return np.median(samples)

median_time(greedy_star, F, G)
```

```
7.0
```

To compute the median with $\beta = 0.99$ instead of the default value $\beta = 0.95$, replace `cw = CareerWorkerProblem()` with `cw = CareerWorkerProblem(β=0.99)`.

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 38.4.3

Set the parameterization to `G_a = G_b = 100` and generate a new optimal policy figure – interpret.

Solution to Exercise 38.4.3

Here is one solution

```
cw = CareerWorkerProblem(G_a=100, G_b=100)
T, get_greedy = operator_factory(cw)
v_star = solve_model(cw, verbose=False)
greedy_star = get_greedy(v_star)

fig, ax = plt.subplots(figsize=(6, 6))
tg, eg = np.meshgrid(cw.θ, cw.ϵ)
lvls = (0.5, 1.5, 2.5, 3.5)
```

(continues on next page)

(continued from previous page)

```
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

THIRTYNINE

JOB SEARCH VI: ON-THE-JOB SEARCH

Contents

- *Job Search VI: On-the-Job Search*
  - *Overview*
  - *Model*
  - *Implementation*
  - *Solving for Policies*
  - *Exercises*

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install interpolation
```

39.1 Overview

In this section, we solve a simple on-the-job search model

- based on [LS18], exercise 6.18, and [Jov79]

Let's start with some imports:

```
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
import scipy.stats as stats
from interpolation import interp
from numba import njit, prange
```

```
/opt/conda/envs/quantecon/lib/python3.11/site-packages/numba/core/decorators.
↪py:262: NumbaDeprecationWarning: numba.generated_jit is deprecated. Please see␣
↪the documentation at: https://numba.readthedocs.io/en/stable/reference/
↪deprecation.html#deprecation-of-generated-jit for more information and advice on␣
↪a suitable replacement.
  warnings.warn(msg, NumbaDeprecationWarning)
```

39.1.1 Model Features

- job-specific human capital accumulation combined with on-the-job search
- infinite-horizon dynamic programming with one state variable and two controls

39.2 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

$$x_{t+1} = (1 - b_{t+1})g(x_t, \phi_t) + b_{t+1} \max\{g(x_t, \phi_t), u_{t+1}\} \tag{39.1}$$

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 (39.1), the Bellman equation for this problem can be written as

$$v(x) = \max_{s+\phi \leq 1} \left\{ x(1 - s - \phi) + \beta(1 - \pi(s))v[g(x, \phi)] + \beta\pi(s) \int v[g(x, \phi) \vee u] f(du) \right\} \tag{39.2}$$

Here nonnegativity of s and ϕ is understood, while $a \vee b := \max\{a, b\}$.

39.2.1 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 $\text{Beta}(2, 2)$ distribution is supported on $(0, 1)$ - it has a unimodal, symmetric density peaked at 0.5.

39.2.2 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 (39.1) 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.

39.3 Implementation

We will set up a class `JVWorker` that holds the parameters of the model described above

```
class JVWorker:
    r"""
    A Jovanovic-type model of employment with on-the-job search.

    """

    def __init__(self,
                 A=1.4,
                 α=0.6,
                 β=0.96,          # Discount factor
                 π=np.sqrt,       # Search effort function
                 a=2,             # Parameter of f
                 b=2,             # Parameter of f
                 grid_size=50,
```

(continues on next page)

(continued from previous page)

```
                 mc_size=100,
                 ε=1e-4):

        self.A, self.α, self.β, self.π = A, α, β, π
        self.mc_size, self.ε = mc_size, ε

        self.g = njit(lambda x, ϕ: A * (x * ϕ)**α)     # Transition function
        self.f_rvs = np.random.beta(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)
        ε = 1e-4
        grid_max = max(A**(1 / (1 - α)), stats.beta(a, b).ppf(1 - ε))

        # Human capital
        self.x_grid = np.linspace(ε, grid_max, grid_size)
```

The function `operator_factory` takes an instance of this class and returns a jitted version of the Bellman operator `T`, i.e.

$$Tv(x) = \max_{s+\phi \leq 1} w(s, \phi)$$

where

$$w(s, \phi) := x(1 - s - \phi) + \beta(1 - \pi(s))v[g(x, \phi)] + \beta\pi(s) \int v[g(x, \phi) \vee u] f(du) \tag{39.3}$$

When we represent v, it will be with a NumPy array `v` giving values on grid `x_grid`.

But to evaluate the right-hand side of (39.3), we need a function, so we replace the arrays `v` and `x_grid` with a function `v_func` that gives linear interpolation of `v` on `x_grid`.

Inside the `for` loop, for each `x` in the grid over the state space, we set up the function $w(z) = w(s, \phi)$ defined in (39.3).

The function is maximized over all feasible (s, ϕ) pairs.

Another function, `get_greedy` returns the optimal choice of s and ϕ at each x, given a value function.

```
def operator_factory(jv, parallel_flag=True):

    """
    Returns a jitted version of the Bellman operator T

    jv is an instance of JVWorker

    """

    π, β = jv.π, jv.β
    x_grid, ε, mc_size = jv.x_grid, jv.ε, jv.mc_size
    f_rvs, g = jv.f_rvs, jv.g

    @njit
    def state_action_values(z, x, v):
        s, ϕ = z
        v_func = lambda x: interp(x_grid, v, x)

        integral = 0
        for m in range(mc_size):
```

(continues on next page)

(continued from previous page)

```
            u = f_rvs[m]
            integral += v_func(max(g(x, ϕ), u))
        integral = integral / mc_size

        q = π(s) * integral + (1 - π(s)) * v_func(g(x, ϕ))
        return x * (1 - ϕ - s) + β * q

    @njit(parallel=parallel_flag)
    def T(v):
        """
        The Bellman operator
        """

        v_new = np.empty_like(v)
        for i in prange(len(x_grid)):
            x = x_grid[i]

            # Search on a grid
            search_grid = np.linspace(ε, 1, 15)
            max_val = -1
            for s in search_grid:
                for ϕ in search_grid:
                    current_val = state_action_values((s, ϕ), x, v) if s + ϕ <= 1␣
↪else -1
                    if current_val > max_val:
                        max_val = current_val
            v_new[i] = max_val

        return v_new

    @njit
    def get_greedy(v):
        """
        Computes the v-greedy policy of a given function v
        """
        s_policy, ϕ_policy = np.empty_like(v), np.empty_like(v)

        for i in range(len(x_grid)):
            x = x_grid[i]
            # Search on a grid
            search_grid = np.linspace(ε, 1, 15)
            max_val = -1
            for s in search_grid:
                for ϕ in search_grid:
                    current_val = state_action_values((s, ϕ), x, v) if s + ϕ <= 1␣
↪else -1
                    if current_val > max_val:
                        max_val = current_val
                        max_s, max_ϕ = s, ϕ
                        s_policy[i], ϕ_policy[i] = max_s, max_ϕ
        return s_policy, ϕ_policy

    return T, get_greedy
```

To solve the model, we will write a function that uses the Bellman operator and iterates to find a fixed point.

```
def solve_model(jv,
                use_parallel=True,
                tol=1e-4,
                max_iter=1000,
                verbose=True,
                print_skip=25):

    """
    Solves the model by value function iteration

    * jv is an instance of JVWorker

    """

    T, _ = operator_factory(jv, parallel_flag=use_parallel)

    # Set up loop
    v = jv.x_grid * 0.5  # Initial condition
    i = 0
    error = tol + 1

    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
```

39.4 Solving for Policies

Let's generate the optimal policies and see what they look like.

```
jv = JVWorker()
T, get_greedy = operator_factory(jv)
v_star = solve_model(jv)
s_star, ϕ_star = get_greedy(v_star)
```

```
Error at iteration 25 is 0.1511117438778582.
Error at iteration 50 is 0.054460176346580624.
```

```
Error at iteration 75 is 0.01962726875481735.
Error at iteration 100 is 0.007073603220129954.
```

```
Error at iteration 125 is 0.002549303376893519.
Error at iteration 150 is 0.0009187605673108834.
```

```
Error at iteration 175 is 0.0003311182920384681.
Error at iteration 200 is 0.00011933394534402453.

Converged in 205 iterations.
```

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()

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.

39.5 Exercises

Exercise 39.5.1

Let's look at the dynamics for the state process $\{x_t\}$ associated with these policies.

The dynamics are given by (39.1) 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 = JVWorker(grid_size=25, mc_size=50)
plot_grid_max, plot_grid_size = 1.2, 100
plot_grid = np.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 to Exercise 39.5.1

Here's code to produce the 45 degree diagram

```
jv = JVWorker(grid_size=25, mc_size=50)
π, g, f_rvs, x_grid = jv.π, jv.g, jv.f_rvs, jv.x_grid
T, get_greedy = operator_factory(jv)
v_star = solve_model(jv, verbose=False)
s_policy, ϕ_policy = get_greedy(v_star)

# Turn the policy function arrays into actual functions
s = lambda y: interp(x_grid, s_policy, y)
ϕ = lambda y: interp(x_grid, ϕ_policy, y)

def h(x, b, u):
    return (1 - b) * g(x, ϕ(x)) + b * max(g(x, ϕ(x)), u)


plot_grid_max, plot_grid_size = 1.2, 100
plot_grid = np.linspace(0, plot_grid_max, plot_grid_size)
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
for x in plot_grid:
    for i in range(jv.mc_size):
```

(continues on next page)

(continued from previous page)

```
        b = 1 if np.random.uniform(0, 1) < π(s(x)) else 0
        u = f_rvs[i]
        y = h(x, b, u)
        ax.plot(x, y, '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 39.5.2

In Exercise 39.5.1, 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 to Exercise 39.5.2

The figure can be produced as follows

```
jv = JVWorker()

def xbar(ϕ):
    A, α = jv.A, jv.α
    return (A * ϕ**α)**(1 / (1 - α))

ϕ_grid = np.linspace(0, 1, 100)
fig, ax = plt.subplots(figsize=(9, 7))
ax.set(xlabel='$\phi$')
ax.plot(ϕ_grid, [xbar(ϕ) * (1 - ϕ) for ϕ in ϕ_grid], label='$w^*(\phi)$')
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 39.5.1.

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

FORTY

JOB SEARCH VII: A MCCALL WORKER Q-LEARNS

40.1 Overview

This lecture illustrates a powerful machine learning technique called Q-learning.

[SB18] 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)
```

40.2 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

$$V(w) = \max_{\text{accept, reject}} \left\{ \frac{w}{1-\beta}, c + \beta \int V(w')\, dF(w') \right\} \tag{40.1}$$

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:

$$v'(i) = \max \left\{ \frac{w(i)}{1-\beta}, c + \beta \sum_{1 \leq j \leq n} v(j) q(j) \right\} \quad \text{for } i = 1, \dots, n$$

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[:]),    # array of wage values, w[i] = wage at state i
    ('q', float64[:]),     # 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):
```

(continues on next page)

(continued from previous page)

```
        """
        The values of state-action pairs.
        """
        # 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 * np.sum(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()
```

(continues on next page)

(continued from previous page)

```
valfunc_VFI, flag = mcm.VFI()

fig, ax = plt.subplots(figsize=(10,6))
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 5322.27935875 5500.
 6000.        ]
```

40.3 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

$$\begin{aligned} Q\left(w, \text{accept}\right) &= \frac{w}{1-\beta} \\ Q\left(w, \text{reject}\right) &= c + \beta \int \max_{\text{accept, reject}} \left\{ \frac{w'}{1-\beta}, Q\left(w', \text{reject}\right) \right\} dF\left(w'\right) \end{aligned} \tag{40.2}$$

Note that the first equation of system (40.2) 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 (40.2), 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 (40.2) 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

$$V\left(w\right) = \max \left\{ \frac{w}{1-\beta}, Q_r \right\}$$

40.4 From Probabilities to Samples

We noted above that the optimal Q function for our McCall worker satisfies the Bellman equations

$$\begin{aligned} w + \beta \max_{\text{accept, reject}} \{Q(w, \text{accept}), Q(w, \text{reject})\} - Q(w, \text{accept}) = 0 \\ c + \beta \int \max_{\text{accept, reject}} \{Q(w', \text{accept}), Q\left(w', \text{reject}\right)\} \, dF\left(w'\right) - Q\left(w, \text{reject}\right) = 0 \end{aligned} \tag{40.3}$$

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 (40.3) but replaces the second by removing integration over $F(w')$:

$$\begin{aligned} w + \beta \max_{\text{accept, reject}} \{Q(w, \text{accept}), Q(w, \text{reject})\} - Q(w, \text{accept}) = 0 \\ c + \beta \max_{\text{accept, reject}} \{Q(w', \text{accept}), Q\left(w', \text{reject}\right)\} - Q\left(w, \text{reject}\right) \approx 0 \end{aligned} \tag{40.4}$$

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"

$$\begin{aligned} w + \beta \max_{\text{accept, reject}} \left\{\hat{Q}_t(w_t, \text{accept}), \hat{Q}_t(w_t, \text{reject})\right\} - \hat{Q}_t(w_t, \text{accept}) = \text{diff}_{\text{accept},t} \\ c + \beta \max_{\text{accept, reject}} \left\{\hat{Q}_t(w_{t+1}, \text{accept}), \hat{Q}_t\left(w_{t+1}, \text{reject}\right)\right\} - \hat{Q}_t\left(w_t, \text{reject}\right) = \text{diff}_{\text{reject},t} \end{aligned} \tag{40.5}$$

The adaptive learning scheme would then be some version of

$$\hat{Q}_{t+1} = \hat{Q}_t + \alpha \, \text{diff}_t \tag{40.6}$$

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 (40.5).

This informal argument takes us to the threshold of Q-learning.

40.5 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

$$\widetilde{Q}^{new}(w,a) = \widetilde{Q}^{old}(w,a) + \alpha \widetilde{TD}(w,a) \tag{40.7}$$

where

$$\begin{aligned} \widetilde{TD}(w,\text{accept}) &= \left[w + \beta \max_{a' \in \mathcal{A}} \widetilde{Q}^{old}(w,a')\right] - \widetilde{Q}^{old}(w,\text{accept}) \\ \widetilde{TD}(w,\text{reject}) &= \left[c + \beta \max_{a' \in \mathcal{A}} \widetilde{Q}^{old}(w',a')\right] - \widetilde{Q}^{old}(w,\text{reject}), \;\; w' \sim F \end{aligned} \tag{40.8}$$

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 (40.6).

An aspect of the algorithm not yet captured by equation system (40.8) is random **experimentation** that we add by occasionally randomly replacing

$$\text{argmax}_{a' \in \mathcal{A}} \widetilde{Q}^{old}(w,a')$$

with

$$\text{argmin}_{a' \in \mathcal{A}} \widetilde{Q}^{old}(w,a')$$

and occasionally replacing

$$\text{argmax}_{a' \in \mathcal{A}} \widetilde{Q}^{old}(w',a')$$

with

$$\text{argmin}_{a' \in \mathcal{A}} \widetilde{Q}^{old}(w',a')$$

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 (40.8) and update $\widetilde{Q}$ according to (40.7).
5. Either draw a new state w' if required or else take existing wage if and update the Q-table again according to (40.7).
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 (40.7) 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):
```

(continues on next page)

(continued from previous page)

```
        """
        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

        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
```

(continues on next page)

(continued from previous page)

```
            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(nopython=True)
def run_epochs(N, qlmc, qtable):
    """
    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)
```

```
[[5289.78842525   24.75124688]
 [5294.44003453 5107.08638883]
```

(continues on next page)

(continued from previous page)

```
 [5356.255516   5368.63597532]
 [5337.59712822 5268.99062573]
 [5315.60883035 5269.31325509]
 [5402.83928755 5314.59243986]
 [5287.03027055 5249.56781036]
 [5352.54785416 5311.3335083 ]
 [5393.31582072 5315.97159717]
 [5394.13294999 5500.00000041]
 [5379.57512675 6000.        ]]
```

```
# inspect value function
valfunc_qlr = valfunc_from_qtable(qtable)

print(valfunc_qlr)
```

```
[5289.78842525 5294.44003453 5368.63597532 5337.59712822 5315.60883035
 5402.83928755 5287.03027055 5352.54785416 5393.31582072 5500.00000041
 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()

# 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,
```

(continues on next page)

(continued from previous page)

```
       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:
            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

40.6 Employed Worker Can't Quit

The preceding version of temporal difference Q-learning described in equation system (40.8) 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 [LS18], 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:

$$\begin{aligned}\widetilde{TD}\left(w,\text{accept}\right) &= \left[w+\beta\widetilde{Q}^{old}\left(w,\text{accept}\right)\right]-\widetilde{Q}^{old}\left(w,\text{accept}\right)\\ \widetilde{TD}\left(w,\text{reject}\right) &= \left[c+\beta\max_{a'\in\mathcal{A}}\widetilde{Q}^{old}\left(w',a'\right)\right]-\widetilde{Q}^{old}\left(w,\text{reject}\right),\ w'\sim F\end{aligned} \tag{40.9}$$

It turns out that formulas (40.9) combined with our Q-learning recursion (40.7) 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
```

40.7 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 VI

Consumption, Savings and Capital

CHAPTER

FORTYONE

CASS-KOOPMANS MODEL

Contents

- *Cass-Koopmans Model*
  - *Overview*
  - *The Model*
  - *Planning Problem*
  - *Shooting Algorithm*
  - *Setting Initial Capital to Steady State Capital*
  - *A Turnpike Property*
  - *A Limiting Infinite Horizon Economy*
  - *Concluding Remarks*

41.1 Overview

This lecture and *Cass-Koopmans Competitive Equilibrium* describe a model that Tjalling Koopmans [Koo65] and David Cass [Cas65] used to analyze optimal growth.

The model can be viewed as an extension of the model of Robert Solow described in an earlier lecture but adapted to make the saving rate be a choice.

(Solow assumed a constant saving rate determined outside the model.)

We describe two versions of the model, one in this lecture and the other in *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 that describes optimal paths for long but finite-horizon economies.

Let's start with some standard imports:

```
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
from numba import njit, float64
from numba.experimental import jitclass
import numpy as np
```

41.2 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 $\vec{K} = \{K_0, \ldots, K_{T+1}\}$.

41.2.1 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

$$C = \int_0^1 c(\omega) d\omega. \tag{41.1}$$

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 (41.1) 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:

$$U(\vec{C}) = \sum_{t=0}^{T} \beta^t \frac{C_t^{1-\gamma}}{1-\gamma} \tag{41.2}$$

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

$$u(C_t) = \frac{C_t^{1-\gamma}}{1-\gamma} \tag{41.3}$$

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

$$F(K_t, N_t) = AK_t^{\alpha} N_t^{1-\alpha} \tag{41.4}$$

with $0 < \alpha < 1$, $A > 0$.

A feasible allocation $\vec{C}, \vec{K}$ satisfies

$$C_t + K_{t+1} \leq F(K_t, N_t) + (1-\delta)K_t \quad \text{for all } t \in \{0, 1, \dots, T\} \tag{41.5}$$

where $\delta \in (0, 1)$ is a depreciation rate of capital.

41.3 Planning Problem

A planner chooses an allocation $\{\vec{C}, \vec{K}\}$ to maximize (41.2) subject to (41.5).

Let $\vec{\mu} = \{\mu_0, \dots, \mu_T\}$ be a sequence of nonnegative **Lagrange multipliers**.

To find an optimal allocation, form a Lagrangian

$$\mathcal{L}(\vec{C}, \vec{K}, \vec{\mu}) = \sum_{t=0}^{T} \beta^t \left\{ u(C_t) + \mu_t \left(F(K_t, 1) + (1-\delta)K_t - C_t - K_{t+1} \right) \right\} \tag{41.6}$$

and pose the following min-max problem:

$$\min_{\vec{\mu}} \max_{\vec{C}, \vec{K}} \mathcal{L}(\vec{C}, \vec{K}, \vec{\mu}) \tag{41.7}$$

- **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.

41.3.1 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

$$\begin{aligned} \frac{\partial F(K_t, N_t)}{\partial K_t} &= \frac{\partial N_t f\left(\frac{K_t}{N_t}\right)}{\partial K_t} \\ &= N_t f'\left(\frac{K_t}{N_t}\right) \frac{1}{N_t} \quad \text{(Chain rule)} \\ &= f'\left(\frac{K_t}{N_t}\right)\Big|_{N_t=1} \\ &= f'(K_t) \end{aligned} \tag{41.8}$$

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 $K_t = \frac{K_t}{N_t}$.)

41.3.2 First-order necessary conditions

We now compute **first-order necessary conditions** for extremization of Lagrangian (41.6):

$$C_t: \qquad u'(C_t) - \mu_t = 0 \qquad \text{for all} \quad t = 0, 1, \ldots, T \tag{41.9}$$

$$K_t: \qquad \beta\mu_t\left[(1-\delta) + f'(K_t)\right] - \mu_{t-1} = 0 \qquad \text{for all} \quad t = 1, 2, \ldots, T \tag{41.10}$$

$$\mu_t: \qquad F(K_t, 1) + (1-\delta)K_t - C_t - K_{t+1} = 0 \qquad \text{for all} \quad t = 0, 1, \ldots, T \tag{41.11}$$

$$K_{T+1}: \qquad -\mu_T \leq 0, \ \leq 0 \text{ if } K_{T+1} = 0; \ = 0 \text{ if } K_{T+1} > 0 \tag{41.12}$$

In computing (41.10) we recognize that K_t appears in both the time t and time $t-1$ feasibility constraints (41.5).

Restrictions (41.12) come from differentiating with respect to K_{T+1} and applying the following **Karush-Kuhn-Tucker condition** (KKT) (see Karush-Kuhn-Tucker conditions):

$$\mu_T K_{T+1} = 0 \tag{41.13}$$

Combining (41.9) and (41.10) 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

$$\beta u'\left(C_{t+1}\right)\left[(1-\delta) + f'\left(K_{t+1}\right)\right] = u'\left(C_t\right) \quad \text{for all } t = 0, 1, \ldots, T \tag{41.14}$$

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 (41.3) becomes the consumption **Euler equation**

$$C_{t+1} = \left(\beta C_t^{\gamma}[f'(K_{t+1}) + (1-\delta)]\right)^{1/\gamma}$$

which we can combine with the feasibility constraint (41.5) to get

$$\begin{aligned} C_{t+1} &= C_t\left(\beta[f'(F(K_t, 1) + (1-\delta)K_t - C_t) + (1-\delta)]\right)^{1/\gamma} \\ K_{t+1} &= F(K_t, 1) + (1-\delta)K_t - C_t. \end{aligned}$$

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()
```

41.4 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 (41.9), (41.10), and (41.11) 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 (41.13)

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 (41.9) and then k_1 from equation (41.11) and μ_1 from equation (41.10).
- 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 (41.13) 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.)

```
@njit
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
```

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.

```
@njit
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:
                print('Converged successfully on iteration ', i+1)
            return c_vec, k_vec

        i += 1
        if i == max_iter:
```

(continues on next page)

(continued from previous page)

```
            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
```

```
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 = ['$c_t$', '$k_t$', '$\mu_t$']
    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]);
```

41.5 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 (41.5) at $\bar{K}$ gives

$$f(\bar{K}) - \delta\bar{K} = \bar{C} \tag{41.15}$$

Substituting $K_t = \bar{K}$ and $C_t = \bar{C}$ for all t into (41.14) 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 (41.4), 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);
```

41.6 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.

Let's calculate and plot the saving rate.

```
@njit
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)
```

41.7 A Limiting Infinite Horizon Economy

We want to set $T = +\infty$.

The appropriate thing to do is to replace terminal condition (41.12) 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 (41.15) 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 $f'(K) = \rho + \delta$.

41.7.1 Exercise

Exercise 41.7.1

- 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?

Solution to Exercise 41.7.1

```
plot_saving_rate(pp, 0.3, k_ss*1.5, [130], k_ter=k_ss, k_ss=k_ss, s_ss=s_ss)
```

41.8 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.

CHAPTER

FORTYTWO

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*

42.1 Overview

This lecture continues our analysis in this lecture *Cass-Koopmans Planning Model* about the model that Tjalling Koopmans [Koo65] and David Cass [Cas65] 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
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
from numba import njit, float64
from numba.experimental import jitclass
import numpy as np
```

42.2 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.

42.2.1 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 (41.2) subject to (41.5).

The allocation that solves the planning problem reappears in a competitive equilibrium, as we shall see below.

42.3 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.

42.4 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 $t = 1, 2, \ldots, T$.

42.4.1 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**.

42.5 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

$$F(\tilde{k}_t, \tilde{n}_t) = A \tilde{k}_t^\alpha \tilde{n}_t^{1-\alpha}$$

42.5.1 Zero Profit Conditions

Zero-profits conditions for capital and labor are

$$F_k(\tilde{k}_t, \tilde{n}_t) = \eta_t$$

and

$$F_n(\tilde{k}_t, \tilde{n}_t) = w_t \tag{42.1}$$

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 $\vec{w} = \{w_0, \dots, w_T\}$ and $\vec{\eta} = \{\eta_0, \dots, \eta_T\}$.

42.6 Household Problem

A representative household lives at $t = 0, 1, \dots, T$.

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 $\vec{c} = \{c_0, \dots, c_T\}$ and let $\vec{k} = \{k_1, \dots, k_{T+1}\}$.

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:

$$\begin{aligned} &\max_{\vec{c},\vec{k}} \sum_{t=0}^{T} \beta^t u(c_t) \\ \text{subject to } &\sum_{t=0}^{T} q_t^0 \left(c_t + (k_{t+1} - (1-\delta)k_t) - (w_t - \eta_t k_t)\right) \leq 0 \end{aligned}$$

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

42.6.1 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.

42.7 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.

42.7.1 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 $t = 0, 1, \dots, T$.

Thus, we guess that for $t = 0, \dots, T$:

$$q_t^0 = \beta^t u'(C_t) \tag{42.2}$$

$$w_t = f(K_t) - K_t f'(K_t) \tag{42.3}$$

$$\eta_t = f'(K_t) \tag{42.4}$$

At these prices, let capital chosen by the household be

$$k_t^*(\vec{q}, \vec{w}, \vec{\eta}), \quad t \geq 0 \tag{42.5}$$

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

$$k_t^* = \tilde{k}_t^* \tag{42.6}$$

$$1 = \tilde{n}_t^* \tag{42.7}$$

$$c_t^* + k_{t+1}^* - (1-\delta)k_t^* = F(\tilde{k}_t^*, \tilde{n}_t^*)$$

We shall verify that for $t = 0, \dots, T$ allocations chosen by the household and the firm both equal the allocation that solves the planning problem:

$$k_t^* = \tilde{k}_t^* = K_t, \tilde{n}_t = 1, c_t^* = C_t \tag{42.8}$$

42.7.2 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.

42.7.3 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

$$c_t: \quad \beta^t u'(c_t) - \lambda q_t^0 = 0 \quad t = 0, 1, \ldots, T \tag{42.9}$$

$$k_t: \quad -\lambda q_t^0 \left[(1-\delta) + \eta_t \right] + \lambda q_{t-1}^0 = 0 \quad t = 1, 2, \ldots, T+1 \tag{42.10}$$

$$\lambda: \quad \left(\sum_{t=0}^{T} q_t^0 \left(c_t + (k_{t+1} - (1-\delta)k_t) - w_t - \eta_t k_t \right) \right) \leq 0 \tag{42.11}$$

$$k_{T+1}: \quad -\lambda q_0^{T+1} \leq 0, \leq 0 \text{ if } k_{T+1} = 0; = 0 \text{ if } k_{T+1} > 0 \tag{42.12}$$

Now we plug in our guesses of prices and do some algebra in the hope of recovering all first-order necessary conditions (41.9)-(41.12) for the planning problem from this lecture *Cass-Koopmans Planning Model*.

Combining (42.9) and (42.2), we get:

$$u'(C_t) = \mu_t$$

which is (41.9).

Combining (42.10), (42.2), and (42.4), we get:

$$-\lambda \beta^t \mu_t \left[(1-\delta) + f'(K_t) \right] + \lambda \beta^{t-1} \mu_{t-1} = 0 \tag{42.13}$$

Rewriting (42.13) 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 (41.10).

Combining (42.11), (42.2), (42.3) and (42.4) after multiplying both sides of (42.11) 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 $t = 0, \dots, T$, it follows that

$$C_t + K_{t+1} - (1-\delta)K_t - F(K_t, 1) = 0 \quad \text{for all } t \text{ in } \{0, 1, \dots, T\}$$

which is (41.11).

Combining (42.12) and (42.2), 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 (41.12) 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.

42.7.4 Representative Firm's Problem

We now turn to the problem faced by a firm in a competitive equilibrium:

If we plug (42.8) into (42.1) for all t, we get

$$\frac{\partial F(K_t, 1)}{\partial K_t} = f'(K_t) = \eta_t$$

which is (42.4).

If we now plug (42.8) into (42.1) 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 (42.5).

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 (42.6) and (42.7) 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
```

```
@njit
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
```

```
@njit
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
```

(continues on next page)

(continued from previous page)

```
        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:
                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.

```
@njit
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

@njit
def w(pp, k_path):
    w_path = pp.f(k_path) - k_path * pp.f_prime(k_path)
    return w_path

@njit
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))
titles = ['Arrow-Hicks Prices', 'Labor Rental Rate', 'Capital Rental Rate',
          'Consumption', 'Capital', 'Lagrange Multiplier']
ylabels = ['$q_t^0$', '$w_t$', '$\eta_t$', '$c_t$', '$k_t$', '$\mu_t$']

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))

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=f'$\gamma = {γ}$')
        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.

42.8 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 $j = 1, 2, \ldots$ against $j = 1, 2, \ldots$.

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

```
@njit
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

@njit
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

FORTYTHREE

CAKE EATING I: INTRODUCTION TO OPTIMAL SAVING

Contents

- *Cake Eating I: Introduction to Optimal Saving*
  - *Overview*
  - *The Model*
  - *The Value Function*
  - *The Optimal Policy*
  - *The Euler Equation*
  - *Exercises*

43.1 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?

Although the topic sounds trivial, this kind of trade-off between current and future utility is at the heart of many savings and consumption problems.

Once we master the ideas in this simple environment, we will apply them to progressively more challenging—and useful—problems.

The main tool we will use to solve the cake eating problem is dynamic programming.

Readers might find it helpful to review the following lectures before reading this one:

- The *shortest paths lecture*
- The *basic McCall model*
- The *McCall model with separation*
- The *McCall model with separation and a continuous wage distribution*

In what follows, we require the following imports:

```
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
```

43.2 The Model

We consider an infinite time horizon $t = 0, 1, 2, 3..$

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 there is

$$x_{t+1} = x_t - c_t$$

left in period $t + 1$.

Consuming quantity c of the cake gives current utility $u(c)$.

We adopt the CRRA utility function

$$u(c) = \frac{c^{1-\gamma}}{1-\gamma} \qquad (\gamma > 0,\ \gamma \neq 1) \tag{43.1}$$

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

$$\max_{\{c_t\}} \sum_{t=0}^{\infty} \beta^t u(c_t) \tag{43.2}$$

subject to

$$x_{t+1} = x_t - c_t \quad \text{and} \quad 0 \leq c_t \leq x_t \tag{43.3}$$

for all t.

A consumption path $\{c_t\}$ satisfying (43.3) where $x_0 = \bar{x}$ 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**

43.2.1 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.

43.2.2 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.

First, higher β implies less discounting, and hence the agent is more patient, which should reduce the rate of consumption.

Second, higher γ implies that marginal utility $u'(c) = c^{-\gamma}$ falls faster with c.

This suggests more smoothing, and hence a lower rate of consumption.

In summary, we expect the rate of consumption to be *decreasing in both parameters*.

Let's see if this is true.

43.3 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.

43.3.1 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,

$$v(x) = \max \sum_{t=0}^{\infty} \beta^t u(c_t) \tag{43.4}$$

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

$$v(x) = \max_{0 \leq c \leq x} \{u(c) + \beta v(x - c)\} \quad \text{for any given } x \geq 0. \tag{43.5}$$

The intuition here is essentially the same 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 (43.5), 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 (43.5), as claimed.

43.3.2 An Analytical Solution

It has been shown that, with u as the CRRA utility function in (43.1), the function

$$v^*(x_t) = \left(1 - \beta^{1/\gamma}\right)^{-\gamma} u(x_t) \tag{43.6}$$

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.

The solution (43.6) 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 when the time comes.

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()
```

43.4 The Optimal Policy

Now that we have the value function, 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 (43.5).

$$c^* = \arg\max_c \{u(c) + \beta v(x - c)\}$$

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

$$\sigma^*(x) := \arg\max_c \{u(c) + \beta v(x - c)\} \quad \text{for all } x$$

If we plug the analytical expression (43.6) for the value function into the right hand side and compute the optimum, we find that

$$\sigma^*(x) = \left(1 - \beta^{1/\gamma}\right) x \tag{43.7}$$

Now let's recall our intuition on the impact of parameters.

We guessed that the consumption rate would be decreasing in both parameters.

This is in fact the case, as can be seen from (43.7).

Here's 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()
```

43.5 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.

43.5.1 Statement and Implications

The Euler equation for the present problem can be stated as

$$u'(c_t^*) = \beta u'(c_{t+1}^*) \tag{43.8}$$

This is necessary condition for the optimal path.

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 $x > 0$,

$$u'(\sigma(x)) = \beta u'(\sigma(x - \sigma(x))) \tag{43.9}$$

Evidently (43.9) is just the policy equivalent of (43.8).

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 (43.7) 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 [MST20].

The following arguments focus on necessity, explaining why an optimal path or policy should satisfy the Euler equation.

43.5.2 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.

43.5.3 Derivation II: Using the Bellman Equation

Another way to derive the Euler equation is to use the Bellman equation (43.5).

Taking the derivative on the right hand side of the Bellman equation with respect to c and setting it to zero, we get

$$u'(c) = \beta v'(x - c) \tag{43.10}$$

To obtain $v'(x - c)$, we set $g(c, x) = u(c) + \beta v(x - c)$, so that, at the optimal choice of consumption,

$$v(x) = g(c, x) \tag{43.11}$$

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

$$v'(x) = \frac{\partial g(c, x)}{\partial x} = \frac{\partial}{\partial x} \beta v(x - c) = \beta v'(x - c) \tag{43.12}$$

(This argument is an example of the Envelope Theorem.)

But now an application of (43.10) gives

$$u'(c) = v'(x) \tag{43.13}$$

Thus, the derivative of the value function is equal to marginal utility.

Combining this fact with (43.12) recovers the Euler equation.

43.6 Exercises

Exercise 43.6.1

How does one obtain the expressions for the value function and optimal policy given in (43.6) and (43.7) 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 (43.6) and (43.7).

In doing so, you will need to use the definition of the value function and the Bellman equation.

Solution to Exercise 43.6.1

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,

$$\begin{aligned} v(x) &= \max_{0 \leq c \leq x} \left\{ u(c) + \beta \frac{\theta^{1-\gamma}}{1-\beta(1-\theta)^{1-\gamma}} \cdot u(x-c) \right\} \\ &= \max_{0 \leq c \leq x} \left\{ \frac{c^{1-\gamma}}{1-\gamma} + \beta \frac{\theta^{1-\gamma}}{1-\beta(1-\theta)^{1-\gamma}} \cdot \frac{(x-c)^{1-\gamma}}{1-\gamma} \right\} \end{aligned}$$

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.

CHAPTER

FORTYFOUR

CAKE EATING II: NUMERICAL METHODS

Contents

- *Cake Eating II: Numerical Methods*
  - *Overview*
  - *Reviewing the Model*
  - *Value Function Iteration*
  - *Time Iteration*
  - *Exercises*

In addition to what's in Anaconda, this lecture will require the following library:

```
!pip install interpolation
```

44.1 Overview

In this lecture we continue the study of *the cake eating problem*.

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.

We will use the following imports:

```
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
from interpolation import interp
from scipy.optimize import minimize_scalar, bisect
```

```
/opt/conda/envs/quantecon/lib/python3.11/site-packages/numba/core/decorators.
↪py:262: NumbaDeprecationWarning: numba.generated_jit is deprecated. Please see␣
↪the documentation at: https://numba.readthedocs.io/en/stable/reference/
↪deprecation.html#deprecation-of-generated-jit for more information and advice on␣
↪a suitable replacement.
  warnings.warn(msg, NumbaDeprecationWarning)
```

44.2 Reviewing the Model

You might like to *review the details* before we start.

Recall in particular that the Bellman equation is

$$v(x) = \max_{0 \leq c \leq x} \{u(c) + \beta v(x - c)\} \quad \text{for all } x \geq 0. \tag{44.1}$$

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.

44.3 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 arbitary intial guess of v.
2. Obtain an update w defined by

$$w(x) = \max_{0 \leq c \leq x} \{u(c) + \beta v(x - c)\}$$

3. Stop if w is approximately equal to v, otherwise set $v = w$ and go back to step 2.

Let's write this a bit more mathematically.

44.3.1 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.

44.3.2 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 $\{v_0, \dots, v_I\}$ representing the values of some initial function v on the grid points $\{x_0, \dots, x_I\}$.
2. Build a function $\hat{v}$ on the state space $\mathbb{R}_+$ by linear interpolation, based on these data points.
3. Obtain and record the value $T\hat{v}(x_i)$ on each grid point x_i by repeatedly solving the maximization problem in the Bellman equation.
4. Unless some stopping condition is satisfied, set $\{v_0, \dots, v_I\} = \{T\hat{v}(x_0), \dots, T\hat{v}(x_I)\}$ and go to step 2.

In step 2 we'll use continuous piecewise linear interpolation.

44.3.3 Implementation

The `maximize` function below is a small helper function that converts a SciPy minimization routine into a maximization routine.

```
def maximize(g, a, b, args):
    """
    Maximize the function g over the interval [a, b].

    We use the fact that the maximizer of g on any interval is
    also the minimizer of -g.  The tuple args collects any extra
    arguments to g.

    Returns the maximal value and the maximizer.
    """

    objective = lambda x: -g(x, *args)
    result = minimize_scalar(objective, bounds=(a, b), method='bounded')
```

(continues on next page)

(continued from previous page)

```
    maximizer, maximum = result.x, -result.fun
    return maximizer, maximum
```

We'll store the parameters β and γ in a class called `CakeEating`.

The same class will also provide a method called `state_action_value` that returns the value of a consumption choice given a particular state and guess of v.

```
class CakeEating:

    def __init__(self,
                 β=0.96,           # discount factor
                 γ=1.5,            # degree of relative risk aversion
                 x_grid_min=1e-3,  # exclude zero for numerical stability
                 x_grid_max=2.5,   # size of cake
                 x_grid_size=120):

        self.β, self.γ = β, γ

        # Set up grid
        self.x_grid = np.linspace(x_grid_min, x_grid_max, x_grid_size)

    # Utility function
    def u(self, c):

        γ = self.γ

        if γ == 1:
            return np.log(c)
        else:
            return (c ** (1 - γ)) / (1 - γ)

    # first derivative of utility function
    def u_prime(self, c):

        return c ** (-self.γ)

    def state_action_value(self, c, x, v_array):
        """
        Right hand side of the Bellman equation given x and c.
        """

        u, β = self.u, self.β
        v = lambda x: interp(self.x_grid, v_array, x)

        return u(c) + β * v(x - c)
```

We now define the Bellman operation:

```
def T(v, ce):
    """
    The Bellman operator.  Updates the guess of the value function.

    * ce is an instance of CakeEating
    * v is an array representing a guess of the value function
```

(continues on next page)

(continued from previous page)

```
    """
    v_new = np.empty_like(v)

    for i, x in enumerate(ce.x_grid):
        # Maximize RHS of Bellman equation at state x
        v_new[i] = maximize(ce.state_action_value, 1e-10, x, (x, v))[1]

    return v_new
```

After defining the Bellman operator, we are ready to solve the model.

Let's start by creating a `CakeEating` instance using the default parameterization.

```
ce = CakeEating()
```

Now let's see the iteration of the value function in action.

We start from guess v given by $v(x) = u(x)$ for every x grid point.

```
x_grid = ce.x_grid
v = ce.u(x_grid)        # Initial guess
n = 12                  # Number of iterations

fig, ax = plt.subplots()

ax.plot(x_grid, v, color=plt.cm.jet(0),
        lw=2, alpha=0.6, label='Initial guess')

for i in range(n):
    v = T(v, ce)  # Apply the Bellman operator
    ax.plot(x_grid, v, color=plt.cm.jet(i / n), lw=2, alpha=0.6)

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 do this more systematically, we introduce a wrapper function called `compute_value_function` that iterates until some convergence conditions are satisfied.

```
def compute_value_function(ce,
                           tol=1e-4,
                           max_iter=1000,
                           verbose=True,
                           print_skip=25):

    # Set up loop
    v = np.zeros(len(ce.x_grid)) # Initial guess
    i = 0
    error = tol + 1

    while i < max_iter and error > tol:
        v_new = T(v, ce)

        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
```

Now let's call it, noting that it takes a little while to run.

```
v = compute_value_function(ce)
```

```
Error at iteration 25 is 23.8003755134813.
```

```
Error at iteration 50 is 8.577577195046615.
```

```
Error at iteration 75 is 3.091330659691039.
```

```
Error at iteration 100 is 1.1141054204751981.
```

```
Error at iteration 125 is 0.4015199357729671.
```

```
Error at iteration 150 is 0.14470646660561215.
```

```
Error at iteration 175 is 0.052151735472762084.
```

```
Error at iteration 200 is 0.018795314242879613.
```

```
Error at iteration 225 is 0.006773769545588948.
```

```
Error at iteration 250 is 0.0024412443051460286.
```

```
Error at iteration 275 is 0.000879816432870939.
```

```
Error at iteration 300 is 0.00031708295398402697.
```

```
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(ce.x_grid, ce.β, ce.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.

44.3.4 Policy Function

Let's see how this plays out in terms of computing the optimal policy.

In the *first lecture on 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

$$\sigma(x) = \arg\max_{0 \leq c \leq x} \{u(c) + \beta v(x - c)\}$$

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 σ(ce, v):
    """
    The optimal policy function. Given the value function,
    it finds optimal consumption in each state.

    * ce is an instance of CakeEating
    * v is a value function array

    """
    c = np.empty_like(v)

    for i in range(len(ce.x_grid)):
        x = ce.x_grid[i]
        # Maximize RHS of Bellman equation at state x
        c[i] = maximize(ce.state_action_value, 1e-10, x, (x, v))[0]

    return c
```

Now let's pass the approximate value function and compute optimal consumption:

```
c = σ(ce, v)
```

Let's plot this next to the true analytical solution

```
c_analytical = c_star(ce.x_grid, ce.β, ce.γ)

fig, ax = plt.subplots()

ax.plot(ce.x_grid, c_analytical, label='analytical')
ax.plot(ce.x_grid, c, 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 next.

44.4 Time Iteration

Now let's look at a different strategy to compute the optimal policy.

Recall that the optimal policy satisfies the Euler equation

$$u'(\sigma(x)) = \beta u'(\sigma(x - \sigma(x))) \quad \text{for all } x > 0 \tag{44.2}$$

Computationally, we can start with any initial guess of σ_0 and now choose c to solve

$$u'(c) = \beta u'(\sigma_0(x - c))$$

Choosing c to satisfy this equation at all $x > 0$ produces a function of x.

Call this new function σ_1, treat it as the new guess and repeat.

This is called **time iteration**.

As with value function iteration, we can view the update step as action of an operator, this time denoted by K.

- In particular, $K\sigma$ is the policy updated from σ using the procedure just described.
- We will use this terminology in the exercises below.

The main advantage of time iteration relative to value function iteration is that it operates in policy space rather than value function space.

This is helpful because the policy function has less curvature, and hence is easier to approximate.

In the exercises you are asked to implement time iteration and compare it to value function iteration.

You should find that the method is faster and more accurate.

This is due to

1. the curvature issue mentioned just above and
2. the fact that we are using more information — in this case, the first order conditions.

44.5 Exercises

Exercise 44.5.1

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 to Exercise 44.5.1

We need to create a class to hold our primitives and return the right hand side of the Bellman equation.

We will use inheritance to maximize code reuse.

```
class OptimalGrowth(CakeEating):
    """
    A subclass of CakeEating that adds the parameter α and overrides
    the state_action_value method.
    """

    def __init__(self,
                 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):

        self.α = α
        CakeEating.__init__(self, β, γ, x_grid_min, x_grid_max, x_grid_size)

    def state_action_value(self, c, x, v_array):
        """
        Right hand side of the Bellman equation given x and c.
        """

        u, β, α = self.u, self.β, self.α
        v = lambda x: interp(self.x_grid, v_array, x)
```

(continues on next page)

(continued from previous page)

```
        return u(c) + β * v((x - c)**α)
```

```
og = OptimalGrowth()
```

Here's the computed value function.

```
v = compute_value_function(og, verbose=False)

fig, ax = plt.subplots()

ax.plot(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 $\alpha = 1$.

```
c_new = σ(og, v)

fig, ax = plt.subplots()

ax.plot(ce.x_grid, c_analytical, label=r'$\alpha=1$ solution')
ax.plot(ce.x_grid, c_new, label=fr'$\alpha={og.α}$ 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.

Exercise 44.5.2

Implement time iteration, returning to the original case (i.e., dropping the modification in the exercise above).

Solution to Exercise 44.5.2

Here's one way to implement time iteration.

```
def K(σ_array, ce):
    """
    The policy function operator. Given the policy function,
    it updates the optimal consumption using Euler equation.

    * σ_array is an array of policy function values on the grid
    * ce is an instance of CakeEating

    """

    u_prime, β, x_grid = ce.u_prime, ce.β, ce.x_grid
    σ_new = np.empty_like(σ_array)

    σ = lambda x: interp(x_grid, σ_array, x)

    def euler_diff(c, x):
        return u_prime(c) - β * u_prime(σ(x - c))

    for i, x in enumerate(x_grid):

        # handle small x separately --- helps numerical stability
        if x < 1e-12:
            σ_new[i] = 0.0
```

(continues on next page)

(continued from previous page)

```
        # handle other x
        else:
            σ_new[i] = bisect(euler_diff, 1e-10, x - 1e-10, x)

    return σ_new
```

```
def iterate_euler_equation(ce,
                           max_iter=500,
                           tol=1e-5,
                           verbose=True,
                           print_skip=25):

    x_grid = ce.x_grid

    σ = np.copy(x_grid)         # initial guess

    i = 0
    error = tol + 1
    while i < max_iter and error > tol:

        σ_new = K(σ, ce)

        error = np.max(np.abs(σ_new - σ))
        i += 1

        if verbose and i % print_skip == 0:
            print(f"Error at iteration {i} is {error}.")

        σ = σ_new

    if error > tol:
        print("Failed to converge!")
    elif verbose:
        print(f"\nConverged in {i} iterations.")

    return σ
```

```
ce = CakeEating(x_grid_min=0.0)
c_euler = iterate_euler_equation(ce)
```

```
Error at iteration 25 is 0.0036456675931543225.
```

```
Error at iteration 50 is 0.0008283185047067848.
```

```
Error at iteration 75 is 0.00030791132300957147.
```

```
Error at iteration 100 is 0.00013555502390599772.
```

```
Error at iteration 125 is 6.417740905302616e-05.
```

```
Error at iteration 150 is 3.1438019047758115e-05.
```

```
Error at iteration 175 is 1.5658492883291464e-05.
```

```
Converged in 192 iterations.
```

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

ax.plot(ce.x_grid, c_analytical, label='analytical solution')
ax.plot(ce.x_grid, c_euler, label='time iteration solution')

ax.set_ylabel('consumption')
ax.set_xlabel('$x$')
ax.legend(fontsize=12)

plt.show()
```

CHAPTER

FORTYFIVE

OPTIMAL GROWTH I: THE STOCHASTIC OPTIMAL GROWTH MODEL

Contents

- *Optimal Growth I: The Stochastic Optimal Growth Model*
  - *Overview*
  - *The Model*
  - *Computation*
  - *Exercises*

45.1 Overview

In this lecture, we're going to study a simple optimal growth model with one agent.

The model is a version of the standard one sector infinite horizon growth model studied in

- [SLP89], chapter 2
- [LS18], section 3.1
- EDTC, chapter 1
- [Sun96], chapter 12

It is an extension of the simple *cake eating problem* we looked at earlier.

The extension involves

- nonlinear returns to saving, through a production function, and
- stochastic returns, due to shocks to production.

Despite these additions, the model is still relatively simple.

We regard it as a stepping stone to more sophisticated models.

We solve the model using dynamic programming and a range of numerical techniques.

In this first lecture on optimal growth, the solution method will be value function iteration (VFI).

While the code in this first lecture runs slowly, we will use a variety of techniques to drastically improve execution time over the next few lectures.

Let's start with some imports:

```
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
from scipy.interpolate import interp1d
from scipy.optimize import minimize_scalar
```

45.2 The Model

Consider an agent who owns an amount $y_t \in \mathbb{R}_+ := [0, \infty)$ of a consumption good at time t.

This output can either be consumed or invested.

When the good is invested, it is transformed one-for-one into capital.

The resulting capital stock, denoted here by k_{t+1}, will then be 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

$$y_{t+1} := f(k_{t+1})\xi_{t+1}$$

where $f \colon \mathbb{R}_+ \to \mathbb{R}_+$ is called the production function.

The resource constraint is

$$k_{t+1} + c_t \leq y_t \tag{45.1}$$

and all variables are required to be nonnegative.

45.2.1 Assumptions and Comments

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.
- Depreciation of capital is not made explicit but can be incorporated into the production function.

While many other treatments of the stochastic growth model use k_t as the state variable, we will use y_t.

This will allow us to treat a stochastic model while maintaining only one state variable.

We consider alternative states and timing specifications in some of our other lectures.

45.2.2 Optimization

Taking y_0 as given, the agent wishes to maximize

$$\mathbb{E}\left[\sum_{t=0}^{\infty} \beta^t u(c_t)\right] \tag{45.2}$$

subject to

$$y_{t+1} = f(y_t - c_t)\xi_{t+1} \quad \text{and} \quad 0 \leq c_t \leq y_t \quad \text{for all } t \tag{45.3}$$

where

- u is a bounded, continuous and strictly increasing utility function and
- $\beta \in (0, 1)$ is a discount factor.

In (45.3) we are assuming that the resource constraint (45.1) holds with equality — which is reasonable because u is strictly increasing and no output will be wasted at the optimum.

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 in the sense of (45.1),
3. optimal, in the sense that it maximizes (45.2) relative to all other feasible consumption sequences, and
4. *adapted*, in the sense that the action c_t depends only on observable outcomes, not on future outcomes such as ξ_{t+1}.

In the present context

- y_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.

45.2.3 The Policy Function Approach

One way to think about solving this problem is to look for the best **policy function**.

A policy function is a map from past and present observables into current action.

We'll be particularly interested in **Markov policies**, which are maps from the current state y_t into a current action c_t.

For dynamic programming problems such as this one (in fact for any Markov decision process), the optimal policy is always a Markov policy.

In other words, the current state y_t provides a sufficient statistic for the history in terms of making an optimal decision today.

This is quite intuitive, but if you wish you can find proofs in texts such as [SLP89] (section 4.1).

Hereafter we focus on finding the best Markov policy.

In our context, a Markov policy is a function $\sigma \colon \mathbb{R}_+ \to \mathbb{R}_+$, with the understanding that states are mapped to actions via

$$c_t = \sigma(y_t) \quad \text{for all } t$$

In what follows, we will call σ a *feasible consumption policy* if it satisfies

$$0 \leq \sigma(y) \leq y \quad \text{for all} \quad y \in \mathbb{R}_+ \tag{45.4}$$

In other words, a feasible consumption policy is a Markov policy that respects the resource constraint.

The set of all feasible consumption policies will be denoted by Σ.

Each $\sigma \in \Sigma$ determines a continuous state Markov process $\{y_t\}$ for output via

$$y_{t+1} = f(y_t - \sigma(y_t))\xi_{t+1}, \quad y_0 \text{ given} \tag{45.5}$$

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

$$\mathbb{E}\left[\sum_{t=0}^{\infty} \beta^t u(c_t)\right] = \mathbb{E}\left[\sum_{t=0}^{\infty} \beta^t u(\sigma(y_t))\right] \tag{45.6}$$

This is the total expected present value of following policy σ forever, given initial income y_0.

The aim is to select a policy that makes this number as large as possible.

The next section covers these ideas more formally.

45.2.4 Optimality

The σ associated with a given policy σ is the mapping defined by

$$v_\sigma(y) = \mathbb{E}\left[\sum_{t=0}^{\infty} \beta^t u(\sigma(y_t))\right] \tag{45.7}$$

when $\{y_t\}$ is given by (45.5) with $y_0 = y$.

In other words, it is the lifetime value of following policy σ starting at initial condition y.

The **value function** is then defined as

$$v^*(y) := \sup_{\sigma \in \Sigma} \; v_\sigma(y) \tag{45.8}$$

The value function gives the maximal value that can be obtained from state y, after considering all feasible policies.

A policy $\sigma \in \Sigma$ is called **optimal** if it attains the supremum in (45.8) for all $y \in \mathbb{R}_+$.

45.2.5 The Bellman Equation

With our assumptions on utility and production functions, the value function as defined in (45.8) also satisfies a **Bellman equation**.

For this problem, the Bellman equation takes the form

$$v(y) = \max_{0 \leq c \leq y} \left\{ u(c) + \beta \int v(f(y-c)z)\phi(dz) \right\} \qquad (y \in \mathbb{R}_+) \tag{45.9}$$

This is a *functional equation in* v.

The term $\int v(f(y-c)z)\phi(dz)$ can be understood as the expected next period value when

- v is used to measure value
- the state is y
- consumption is set to c

As shown in EDTC, theorem 10.1.11 and a range of other texts

The value function v^* satisfies the Bellman equation

In other words, (45.9) 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 gives us more information about the value function.

It also suggests a way of computing the value function, which we discuss below.

45.2.6 Greedy Policies

The primary importance of the value function is that we can use it to compute optimal policies.

The details are as follows.

Given a continuous function v on $\mathbb{R}_+$, we say that $\sigma \in \Sigma$ is v-**greedy** if $\sigma(y)$ is a solution to

$$\max_{0 \leq c \leq y} \left\{ u(c) + \beta \int v(f(y-c)z)\phi(dz) \right\} \tag{45.10}$$

for every $y \in \mathbb{R}_+$.

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

- A feasible consumption policy is optimal if and only if it is v^*-greedy.

The intuition is similar to the intuition for the Bellman equation, which was provided after (45.9).

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.

45.2.7 The Bellman Operator

How, then, should we compute the value function?

One way is to use the so-called **Bellman operator**.

(An operator is a map that sends functions into functions.)

The Bellman operator is denoted by T and defined by

$$Tv(y) := \max_{0 \leq c \leq y} \left\{ u(c) + \beta \int v(f(y-c)z)\phi(dz) \right\} \qquad (y \in \mathbb{R}_+) \tag{45.11}$$

In other words, T sends the function v into the new function Tv defined by (45.11).

By construction, the set of solutions to the Bellman equation (45.9) *exactly coincides with* the set of fixed points of T.

For example, if $Tv = v$, then, for any $y \geq 0$,

$$v(y) = Tv(y) = \max_{0 \leq c \leq y} \left\{ u(c) + \beta \int v^*(f(y-c)z)\phi(dz) \right\}$$

which says precisely that v is a solution to the Bellman equation.

It follows that v^* is a fixed point of T.

45.2.8 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

$$\rho(g, h) = \sup_{y \geq 0} |g(y) - h(y)|$$

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 (see EDTC, theorem 10.1.11 if you get stuck).

Hence, at least one optimal policy exists.

Our problem now is how to compute it.

45.2.9 Unbounded Utility

The results stated above assume that the utility function is bounded.

In practice economists often work with unbounded utility functions — and so will we.

In the unbounded setting, various optimality theories exist.

Unfortunately, they tend to be case-specific, as opposed to valid for a large range of applications.

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").

Consult, for example, section 12.2 of EDTC, [Kam12] or [MdRV10].

45.3 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.

Both of these things are helpful, but they do cost us some speed — as you will see when you run the code.

Later we will sacrifice some of this clarity and flexibility in order to accelerate our code with just-in-time (JIT) compilation.

The algorithm we will use is fitted value function iteration, which was described in earlier lectures *the McCall model* and *cake eating*.

The algorithm will be

1. Begin with an array of values $\{v_1, \dots, v_I\}$ representing the values of some initial function v on the grid points $\{y_1, \dots, y_I\}$.
2. Build a function $\hat{v}$ on the state space $\mathbb{R}_+$ by linear interpolation, based on these data points.
3. Obtain and record the value $T\hat{v}(y_i)$ on each grid point y_i by repeatedly solving (45.11).
4. Unless some stopping condition is satisfied, set $\{v_1, \dots, v_I\} = \{T\hat{v}(y_1), \dots, T\hat{v}(y_I)\}$ and go to step 2.

45.3.1 Scalar Maximization

To maximize the right hand side of the Bellman equation (45.9), we are going to use the `minimize_scalar` routine from SciPy.

Since we are maximizing rather than minimizing, we will use the fact that the maximizer of g on the interval $[a, b]$ is the minimizer of $-g$ on the same interval.

To this end, and to keep the interface tidy, we will wrap `minimize_scalar` in an outer function as follows:

```
def maximize(g, a, b, args):
    """
    Maximize the function g over the interval [a, b].

    We use the fact that the maximizer of g on any interval is
    also the minimizer of -g.  The tuple args collects any extra
    arguments to g.

    Returns the maximal value and the maximizer.
    """

    objective = lambda x: -g(x, *args)
    result = minimize_scalar(objective, bounds=(a, b), method='bounded')
    maximizer, maximum = result.x, -result.fun
    return maximizer, maximum
```

45.3.2 Optimal Growth Model

We will assume for now that ϕ is the distribution of $\xi := \exp(\mu + s\zeta)$ where

- ζ is standard normal,
- μ is a shock location parameter and
- s is a shock scale parameter.

We will store this and other primitives of the optimal growth model in a class.

The class, defined below, combines both parameters and a method that realizes the right hand side of the Bellman equation (45.9).

```
class OptimalGrowthModel:

    def __init__(self,
                 u,             # utility function
                 f,             # production function
                 β=0.96,        # discount factor
                 μ=0,           # shock location parameter
```

(continues on next page)

(continued from previous page)

```
                 s=0.1,         # shock scale parameter
                 grid_max=4,
                 grid_size=120,
                 shock_size=250,
                 seed=1234):

        self.u, self.f, self.β, self.μ, self.s = u, f, β, μ, s

        # Set up grid
        self.grid = np.linspace(1e-4, grid_max, grid_size)

        # Store shocks (with a seed, so results are reproducible)
        np.random.seed(seed)
        self.shocks = np.exp(μ + s * np.random.randn(shock_size))

    def state_action_value(self, c, y, v_array):
        """
        Right hand side of the Bellman equation.
        """

        u, f, β, shocks = self.u, self.f, self.β, self.shocks

        v = interp1d(self.grid, v_array)

        return u(c) + β * np.mean(v(f(y - c) * shocks))
```

In the second last line we are using linear interpolation.

In the last line, the expectation in (45.11) is computed via Monte Carlo, using the approximation

$$\int v(f(y-c)z)\phi(dz) \approx \frac{1}{n}\sum_{i=1}^{n} v(f(y-c)\xi_i)$$

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., [PalS13].)

45.3.3 The Bellman Operator

The next function implements the Bellman operator.

(We could have added it as a method to the `OptimalGrowthModel` class, but we prefer small classes rather than monolithic ones for this kind of numerical work.)

```
def T(v, og):
    """
    The Bellman operator.  Updates the guess of the value function
    and also computes a v-greedy policy.

      * og is an instance of OptimalGrowthModel
      * v is an array representing a guess of the value function
```

(continues on next page)

(continued from previous page)

```
    """
    v_new = np.empty_like(v)
    v_greedy = np.empty_like(v)

    for i in range(len(grid)):
        y = grid[i]

        # Maximize RHS of Bellman equation at state y
        c_star, v_max = maximize(og.state_action_value, 1e-10, y, (y, v))
        v_new[i] = v_max
        v_greedy[i] = c_star

    return v_greedy, v_new
```

45.3.4 An Example

Let's suppose now that

$$f(k) = k^\alpha \quad \text{and} \quad u(c) = \ln c$$

For this particular problem, an exact analytical solution is available (see [LS18], section 3.1.2), with

$$v^*(y) = \frac{\ln(1-\alpha\beta)}{1-\beta} + \frac{(\mu + \alpha \ln(\alpha\beta))}{1-\alpha}\left[\frac{1}{1-\beta} - \frac{1}{1-\alpha\beta}\right] + \frac{1}{1-\alpha\beta}\ln y \tag{45.12}$$

and optimal consumption policy

$$\sigma^*(y) = (1-\alpha\beta)y$$

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(y, 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(y)

def σ_star(y, α, β):
    """
    True optimal policy
    """
    return (1 - α * β) * y
```

Next let's create an instance of the model with the above primitives and assign it to the variable `og`.

```
α = 0.4
def fcd(k):
    return k**α

og = OptimalGrowthModel(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.

```
grid = og.grid

v_init = v_star(grid, α, og.β, og.μ)     # Start at the solution
v_greedy, v = T(v_init, og)              # Apply T once

fig, ax = plt.subplots()
ax.set_ylim(-35, -24)
ax.plot(grid, v, lw=2, alpha=0.6, label='$Tv^*$')
ax.plot(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, $v(y) = 5\ln(y)$.

```
v = 5 * np.log(grid)  # An initial condition
n = 35

fig, ax = plt.subplots()

ax.plot(grid, v, color=plt.cm.jet(0),
        lw=2, alpha=0.6, label='Initial condition')

for i in range(n):
    v_greedy, v = T(v, og)  # Apply the Bellman operator
    ax.plot(grid, v, color=plt.cm.jet(i / n), lw=2, alpha=0.6)

ax.plot(grid, v_star(grid, α, og.β, og.μ), 'k-', lw=2,
        alpha=0.8, label='True value function')
```

(continues on next page)

(continued from previous page)

```
ax.legend()
ax.set(ylim=(-40, 10), xlim=(np.min(grid), np.max(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.

45.3.5 Iterating to Convergence

We can write a function that iterates until the difference is below a particular tolerance level.

```
def solve_model(og,
                tol=1e-4,
                max_iter=1000,
                verbose=True,
                print_skip=25):
    """
    Solve model by iterating with the Bellman operator.

    """

    # Set up loop
    v = og.u(og.grid)  # Initial condition
    i = 0
    error = tol + 1

    while i < max_iter and error > tol:
        v_greedy, v_new = T(v, og)
```

(continues on next page)

(continued from previous page)

```
        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_greedy, v_new
```

Let's use this function to compute an approximate solution at the defaults.

```
v_greedy, v_solution = solve_model(og)
```

```
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 0.006912744396029069.
```

```
Error at iteration 150 is 0.002491330384817303.
```

```
Error at iteration 175 is 0.000897867291303811.
```

```
Error at iteration 200 is 0.00032358842396718046.
```

```
Error at iteration 225 is 0.00011662020561331587.
```

```
Converged in 229 iterations.
```

Now we check our result by plotting it against the true value:

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

ax.plot(grid, v_solution, lw=2, alpha=0.6,
        label='Approximate value function')

ax.plot(grid, v_star(grid, α, og.β, og.μ), lw=2,
        alpha=0.6, label='True value function')

ax.legend()
```

(continues on next page)

(continued from previous page)

```
ax.set_ylim(-35, -24)
plt.show()
```

The figure shows that we are pretty much on the money.

45.3.6 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 $\sigma(y) = (1 - \alpha\beta)y$

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

ax.plot(grid, v_greedy, lw=2,
        alpha=0.6, label='approximate policy function')

ax.plot(grid, σ_star(grid, α, og.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.

45.4 Exercises

Exercise 45.4.1

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 growth model with this utility specification.

Setting $\gamma = 1.5$, compute and plot an estimate of the optimal policy.

Time how long this function takes to run, so you can compare it to faster code developed in the *next lecture*.

Solution to Exercise 45.4.1

Here we set up the model.

```
γ = 1.5   # Preference parameter

def u_crra(c):
    return (c**(1 - γ) - 1) / (1 - γ)

og = OptimalGrowthModel(u=u_crra, f=fcd)
```

Now let's run it, with a timer.

```
%%time
v_greedy, v_solution = solve_model(og)
```

```
Error at iteration 25 is 0.5528151810417512.
```

```
Error at iteration 50 is 0.19923228425590978.
```

```
Error at iteration 75 is 0.07180266113800826.
```

```
Error at iteration 100 is 0.025877443335843964.
```

```
Error at iteration 125 is 0.009326145618970827.
```

```
Error at iteration 150 is 0.003361112262005861.
```

```
Error at iteration 175 is 0.0012113338243295857.
```

```
Error at iteration 200 is 0.0004365607333056687.
```

```
Error at iteration 225 is 0.00015733505506432266.
```

```
Converged in 237 iterations.
CPU times: user 39.3 s, sys: 150 ms, total: 39.4 s
Wall time: 39 s
```

Let's plot the policy function just to see what it looks like:

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

ax.plot(grid, v_greedy, lw=2,
        alpha=0.6, label='Approximate optimal policy')

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

Exercise 45.4.2

Time how long it takes to iterate with the Bellman operator 20 times, starting from initial condition $v(y) = u(y)$.

Use the model specification in the previous exercise.

(As before, we will compare this number with that for the faster code developed in the *next lecture*.)

Solution to Exercise 45.4.2

Let's set up:

```
og = OptimalGrowthModel(u=u_crra, f=fcd)
v = og.u(og.grid)
```

Here's the timing:

```
%%time

for i in range(20):
    v_greedy, v_new = T(v, og)
    v = v_new
```

```
CPU times: user 3.33 s, sys: 32.5 ms, total: 3.36 s
Wall time: 3.3 s
```

CHAPTER

FORTYSIX

OPTIMAL GROWTH II: ACCELERATING THE CODE WITH NUMBA

Contents

- *Optimal Growth II: Accelerating the Code with Numba*
  - *Overview*
  - *The Model*
  - *Computation*
  - *Exercises*

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install quantecon
!pip install interpolation
```

46.1 Overview

Previously, we studied a stochastic optimal growth model with one representative agent.

We solved the model using dynamic programming.

In writing our code, we focused on clarity and flexibility.

These are important, but there's often a trade-off between flexibility and speed.

The reason is that, when code is less flexible, we can exploit structure more easily.

(This is true about algorithms and mathematical problems more generally: more specific problems have more structure, which, with some thought, can be exploited for better results.)

So, in this lecture, we are going to accept less flexibility while gaining speed, using just-in-time (JIT) compilation to accelerate our code.

Let's start with some imports:

```
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
from interpolation import interp
from numba import jit, njit
from quantecon.optimize.scalar_maximization import brent_max
```

```
/opt/conda/envs/quantecon/lib/python3.11/site-packages/numba/core/decorators.
↪py:262: NumbaDeprecationWarning: numba.generated_jit is deprecated. Please see␣
↪the documentation at: https://numba.readthedocs.io/en/stable/reference/
↪deprecation.html#deprecation-of-generated-jit for more information and advice on␣
↪a suitable replacement.
  warnings.warn(msg, NumbaDeprecationWarning)
```

We are using an interpolation function from interpolation.py because it helps us JIT-compile our code.

The function `brent_max` is also designed for embedding in JIT-compiled code.

These are alternatives to similar functions in SciPy (which, unfortunately, are not JIT-aware).

46.2 The Model

The model is the same as discussed in our *previous lecture* on optimal growth.

We will start with log utility:

$$u(c) = \ln(c)$$

We continue to assume that

- $f(k) = k^\alpha$
- ϕ is the distribution of $\xi := \exp(\mu + s\zeta)$ when ζ is standard normal

We will once again use value function iteration to solve the model.

In particular, the algorithm is unchanged, and the only difference is in the implementation itself.

As before, we will be able to compare with the true solutions

```
def v_star(y, 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(y)

def σ_star(y, α, β):
    """
    True optimal policy
    """
    return (1 - α * β) * y
```

46.3 Computation

We will again store the primitives of the optimal growth model in a class.

But now we are going to use Numba's `@jitclass` decorator to target our class for JIT compilation.

Because we are going to use Numba to compile our class, we need to specify the data types.

You will see this as a list called `opt_growth_data` above our class.

Unlike in the *previous lecture*, we hardwire the production and utility specifications into the class.

This is where we sacrifice flexibility in order to gain more speed.

```
from numba import float64
from numba.experimental import jitclass

opt_growth_data = [
    ('α', float64),          # Production parameter
    ('β', float64),          # Discount factor
    ('μ', float64),          # Shock location parameter
    ('s', float64),          # Shock scale parameter
    ('grid', float64[:]),    # Grid (array)
    ('shocks', float64[:])   # Shock draws (array)
]

@jitclass(opt_growth_data)
class OptimalGrowthModel:

    def __init__(self,
                 α=0.4,
                 β=0.96,
                 μ=0,
                 s=0.1,
                 grid_max=4,
                 grid_size=120,
                 shock_size=250,
                 seed=1234):

        self.α, self.β, self.μ, self.s = α, β, μ, s

        # Set up grid
        self.grid = np.linspace(1e-5, grid_max, grid_size)

        # Store shocks (with a seed, so results are reproducible)
        np.random.seed(seed)
        self.shocks = np.exp(μ + s * np.random.randn(shock_size))


    def f(self, k):
        "The production function"
        return k**self.α


    def u(self, c):
        "The utility function"
        return np.log(c)

    def f_prime(self, k):
```

(continues on next page)

(continued from previous page)

```
        "Derivative of f"
        return self.α * (k**(self.α - 1))


    def u_prime(self, c):
        "Derivative of u"
        return 1/c

    def u_prime_inv(self, c):
        "Inverse of u'"
        return 1/c
```

The class includes some methods such as `u_prime` that we do not need now but will use in later lectures.

46.3.1 The Bellman Operator

We will use JIT compilation to accelerate the Bellman operator.

First, here's a function that returns the value of a particular consumption choice `c`, given state `y`, as per the Bellman equation (45.9).

```
@njit
def state_action_value(c, y, v_array, og):
    """
    Right hand side of the Bellman equation.

     * c is consumption
     * y is income
     * og is an instance of OptimalGrowthModel
     * v_array represents a guess of the value function on the grid

    """

    u, f, β, shocks = og.u, og.f, og.β, og.shocks

    v = lambda x: interp(og.grid, v_array, x)

    return u(c) + β * np.mean(v(f(y - c) * shocks))
```

Now we can implement the Bellman operator, which maximizes the right hand side of the Bellman equation:

```
@jit(nopython=True)
def T(v, og):
    """
    The Bellman operator.

     * og is an instance of OptimalGrowthModel
     * v is an array representing a guess of the value function

    """

    v_new = np.empty_like(v)
    v_greedy = np.empty_like(v)
```

(continues on next page)

(continued from previous page)

```
    for i in range(len(og.grid)):
        y = og.grid[i]

        # Maximize RHS of Bellman equation at state y
        result = brent_max(state_action_value, 1e-10, y, args=(y, v, og))
        v_greedy[i], v_new[i] = result[0], result[1]

    return v_greedy, v_new
```

We use the `solve_model` function to perform iteration until convergence.

```
def solve_model(og,
                tol=1e-4,
                max_iter=1000,
                verbose=True,
                print_skip=25):
    """
    Solve model by iterating with the Bellman operator.

    """

    # Set up loop
    v = og.u(og.grid)  # Initial condition
    i = 0
    error = tol + 1

    while i < max_iter and error > tol:
        v_greedy, v_new = T(v, og)
        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_greedy, v_new
```

Let's compute the approximate solution at the default parameters.

First we create an instance:

```
og = OptimalGrowthModel()
```

Now we call `solve_model`, using the `%%time` magic to check how long it takes.

```
%%time
v_greedy, v_solution = solve_model(og)
```

```
Error at iteration 25 is 0.41372668361363196.
```

```
Error at iteration 50 is 0.14767653072604503.
```

```
Error at iteration 75 is 0.053221715530355596.
```

```
Error at iteration 100 is 0.019180931418503633.
```

```
Error at iteration 125 is 0.006912744709538288.
```

```
Error at iteration 150 is 0.002491330497818467.
```

```
Error at iteration 175 is 0.0008978673320712005.
```

```
Error at iteration 200 is 0.0003235884386754151.
```

```
Error at iteration 225 is 0.00011662021095304453.

Converged in 229 iterations.
CPU times: user 8.42 s, sys: 143 ms, total: 8.56 s
Wall time: 8.55 s
```

You will notice that this is *much* faster than our *original implementation*.

Here is a plot of the resulting policy, compared with the true policy:

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

ax.plot(og.grid, v_greedy, lw=2,
        alpha=0.8, label='approximate policy function')

ax.plot(og.grid, σ_star(og.grid, og.α, og.β), 'k--',
        lw=2, alpha=0.8, label='true policy function')

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

Again, the fit is excellent — this is as expected since we have not changed the algorithm.

The maximal absolute deviation between the two policies is

```
np.max(np.abs(v_greedy - σ_star(og.grid, og.α, og.β)))
```

```
0.0010480539639137199
```

46.4 Exercises

Exercise 46.4.1

Time how long it takes to iterate with the Bellman operator 20 times, starting from initial condition $v(y) = u(y)$.

Use the default parameterization.

Solution to Exercise 46.4.1

Let's set up the initial condition.

```
v = og.u(og.grid)
```

Here's the timing:

```
%%time

for i in range(20):
    v_greedy, v_new = T(v, og)
    v = v_new
```

```
CPU times: user 577 ms, sys: 0 ns, total: 577 ms
Wall time: 576 ms
```

Compared with our *timing* for the non-compiled version of value function iteration, the JIT-compiled code is usually an order of magnitude faster.

Exercise 46.4.2

Modify the optimal growth model to use the CRRA utility specification.

$$u(c) = \frac{c^{1-\gamma}}{1-\gamma}$$

Set `γ = 1.5` as the default value and maintaining other specifications.

(Note that `jitclass` currently does not support inheritance, so you will have to copy the class and change the relevant parameters and methods.)

Compute an estimate of the optimal policy, plot it and compare visually with the same plot from the *analogous exercise* in the first optimal growth lecture.

Compare execution time as well.

Solution to Exercise 46.4.2

Here's our CRRA version of `OptimalGrowthModel`:

```
from numba import float64
from numba.experimental import jitclass

opt_growth_data = [
    ('α', float64),          # Production parameter
    ('β', float64),          # Discount factor
    ('μ', float64),          # Shock location parameter
    ('γ', float64),          # Preference parameter
    ('s', float64),          # Shock scale parameter
    ('grid', float64[:]),    # Grid (array)
    ('shocks', float64[:])   # Shock draws (array)
]

@jitclass(opt_growth_data)
class OptimalGrowthModel_CRRA:

    def __init__(self,
                 α=0.4,
                 β=0.96,
                 μ=0,
                 s=0.1,
                 γ=1.5,
                 grid_max=4,
                 grid_size=120,
                 shock_size=250,
                 seed=1234):

        self.α, self.β, self.γ, self.μ, self.s = α, β, γ, μ, s

        # Set up grid
        self.grid = np.linspace(1e-5, grid_max, grid_size)
```

(continues on next page)

(continued from previous page)

```
        # Store shocks (with a seed, so results are reproducible)
        np.random.seed(seed)
        self.shocks = np.exp(μ + s * np.random.randn(shock_size))


    def f(self, k):
        "The production function."
        return k**self.α

    def u(self, c):
        "The utility function."
        return c**(1 - self.γ) / (1 - self.γ)

    def f_prime(self, k):
        "Derivative of f."
        return self.α * (k**(self.α - 1))

    def u_prime(self, c):
        "Derivative of u."
        return c**(-self.γ)

    def u_prime_inv(c):
        return c**(-1 / self.γ)
```

Let's create an instance:

```
og_crra = OptimalGrowthModel_CRRA()
```

Now we call `solve_model`, using the `%%time` magic to check how long it takes.

```
%%time
v_greedy, v_solution = solve_model(og_crra)
```

```
Error at iteration 25 is 1.6201897527234905.
```

```
Error at iteration 50 is 0.459106047057503.
```

```
Error at iteration 75 is 0.165423522162655.
```

```
Error at iteration 100 is 0.05961808343499797.
```

```
Error at iteration 125 is 0.021486161531569792.
```

```
Error at iteration 150 is 0.007743542074422294.
```

```
Error at iteration 175 is 0.002790747140650751.
```

```
Error at iteration 200 is 0.001005776107120937.
```

```
Error at iteration 225 is 0.0003624784085332067.
```

```
Error at iteration 250 is 0.00013063602793295104.

Converged in 257 iterations.
CPU times: user 8.47 s, sys: 99.5 ms, total: 8.57 s
Wall time: 8.55 s
```

Here is a plot of the resulting policy:

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

ax.plot(og.grid, v_greedy, lw=2,
        alpha=0.6, label='Approximate value function')

ax.legend(loc='lower right')
plt.show()
```

This matches the solution that we obtained in our non-jitted code, *in the exercises*.

Execution time is an order of magnitude faster.

Exercise 46.4.3

In this exercise we return to the original log utility specification.

Once an optimal consumption policy σ is given, income follows

$$y_{t+1} = f(y_t - \sigma(y_t))\xi_{t+1}$$

The next figure shows a simulation of 100 elements of this sequence for three different discount factors (and hence three different policies).

In each sequence, the initial condition is $y_0 = 0.1$.

The discount factors are `discount_factors = (0.8, 0.9, 0.98)`.

We have also dialed down the shocks a bit with `s = 0.05`.

Otherwise, the parameters and primitives are the same as the log-linear model discussed earlier in the lecture.

Notice that more patient agents typically have higher wealth.

Replicate the figure modulo randomness.

Solution to Exercise 46.4.3

Here's one solution:

```
def simulate_og(σ_func, og, y0=0.1, ts_length=100):
    '''
    Compute a time series given consumption policy σ.
    '''
    y = np.empty(ts_length)
    ξ = np.random.randn(ts_length-1)
    y[0] = y0
    for t in range(ts_length-1):
        y[t+1] = (y[t] - σ_func(y[t]))**og.α * np.exp(og.μ + og.s * ξ[t])
    return y
```

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

for β in (0.8, 0.9, 0.98):

    og = OptimalGrowthModel(β=β, s=0.05)

    v_greedy, v_solution = solve_model(og, verbose=False)

    # Define an optimal policy function
```

(continues on next page)

(continued from previous page)

```
    σ_func = lambda x: interp(og.grid, v_greedy, x)
    y = simulate_og(σ_func, og)
    ax.plot(y, lw=2, alpha=0.6, label=rf'$\beta = {β}$')

ax.legend(loc='lower right')
plt.show()
```

CHAPTER

FORTYSEVEN

OPTIMAL GROWTH III: TIME ITERATION

Contents

- *Optimal Growth III: 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
!pip install interpolation
```

47.1 Overview

In this lecture, we'll continue our *earlier* study of the stochastic optimal growth model.

In that lecture, we solved the associated dynamic programming problem using value function iteration.

The beauty of this technique is its broad applicability.

With numerical problems, however, we can often attain higher efficiency in specific applications by deriving methods that are carefully tailored to the application at hand.

The stochastic optimal growth model has plenty of structure to exploit for this purpose, especially when we adopt some concavity and smoothness assumptions over primitives.

We'll use this structure to obtain an Euler equation based method.

This will be an extension of the time iteration method considered in our elementary lecture on *cake eating*.

In a *subsequent lecture*, we'll see that time iteration can be further adjusted to obtain even more efficiency.

Let's start with some imports:

```
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
from interpolation import interp
```

(continues on next page)

(continued from previous page)

```
from quantecon.optimize import brentq
from numba import njit
```

```
/opt/conda/envs/quantecon/lib/python3.11/site-packages/numba/core/decorators.
↪py:262: NumbaDeprecationWarning: numba.generated_jit is deprecated. Please see␣
↪the documentation at: https://numba.readthedocs.io/en/stable/reference/
↪deprecation.html#deprecation-of-generated-jit for more information and advice on␣
↪a suitable replacement.
  warnings.warn(msg, NumbaDeprecationWarning)
```

47.2 The Euler Equation

Our first step is to derive the Euler equation, which is a generalization of the Euler equation we obtained in the *lecture on cake eating*.

We take the model set out in *the stochastic growth model lecture* 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

$$v^*(y) = \max_{0 \leq c \leq y} \left\{ u(c) + \beta \int v^*(f(y-c)z)\phi(dz) \right\} \quad \text{for all} \quad y \in \mathbb{R}_+ \tag{47.1}$$

Let the optimal consumption policy be denoted by σ^*.

We know that σ^* is a v^*-greedy policy so that $\sigma^*(y)$ is the maximizer in (47.1).

The conditions above imply that

- σ^* is the unique optimal policy for the stochastic optimal growth model
- the optimal policy is continuous, strictly increasing and also **interior**, in the sense that $0 < \sigma^*(y) < y$ for all strictly positive y, and
- the value function is strictly concave and continuously differentiable, with

$$(v^*)'(y) = u'(\sigma^*(y)) := (u' \circ \sigma^*)(y) \tag{47.2}$$

The last result is called the **envelope condition** due to its relationship with the envelope theorem.

To see why (47.2) holds, write the Bellman equation in the equivalent form

$$v^*(y) = \max_{0 \leq k \leq y} \left\{ u(y-k) + \beta \int v^*(f(k)z)\phi(dz) \right\},$$

Differentiating with respect to y, and then evaluating at the optimum yields (47.2).

(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 (47.1), which is

$$u'(\sigma^*(y)) = \beta \int (v^*)'(f(y - \sigma^*(y))z) f'(y - \sigma^*(y)) z \phi(dz) \tag{47.3}$$

Combining (47.2) and the first-order condition (47.3) gives the **Euler equation**

$$(u' \circ \sigma^*)(y) = \beta \int (u' \circ \sigma^*)(f(y - \sigma^*(y))z) f'(y - \sigma^*(y)) z \phi(dz) \tag{47.4}$$

We can think of the Euler equation as a functional equation

$$(u' \circ \sigma)(y) = \beta \int (u' \circ \sigma)(f(y - \sigma(y))z) f'(y - \sigma(y)) z \phi(dz) \tag{47.5}$$

over interior consumption policies σ, one solution of which is the optimal policy σ^*.

Our aim is to solve the functional equation (47.5) and hence obtain σ^*.

47.2.1 The Coleman-Reffett Operator

Recall the Bellman operator

$$Tv(y) := \max_{0 \leq c \leq y} \left\{ u(c) + \beta \int v(f(y - c)z) \phi(dz) \right\} \tag{47.6}$$

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 $K\sigma(y)$ is the $c \in (0, y)$ that solves.

$$u'(c) = \beta \int (u' \circ \sigma)(f(y - c)z) f'(y - c) z \phi(dz) \tag{47.7}$$

We call this operator the **Coleman-Reffett operator** to acknowledge the work of [Col90] and [Ref96].

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 (47.5).

In particular, the optimal policy σ^* is a fixed point.

Indeed, for fixed y, the value $K\sigma^*(y)$ is the c that solves

$$u'(c) = \beta \int (u' \circ \sigma^*)(f(y - c)z) f'(y - c) z \phi(dz)$$

In view of the Euler equation, this is exactly $\sigma^*(y)$.

47.2.2 Is the Coleman-Reffett Operator Well Defined?

In particular, is there always a unique $c \in (0, y)$ that solves (47.7)?

The answer is yes, under our assumptions.

For any $\sigma \in \mathcal{P}$, the right side of (47.7)

- is continuous and strictly increasing in c on $(0, y)$
- diverges to $+\infty$ as $c \uparrow y$

The left side of (47.7)

- is continuous and strictly decreasing in c on $(0, y)$
- 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 $(0, y)$.

With a bit more analysis, one can show in addition that $K\sigma \in \mathcal{P}$ whenever $\sigma \in \mathcal{P}$.

47.2.3 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, the two operators are *topologically conjugate*.

Loosely speaking, this means 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.

Examples are given below.

47.3 Implementation

As in our *previous study*, we continue to assume that

- $u(c) = \ln c$
- $f(k) = k^\alpha$
- ϕ is the distribution of $\xi := \exp(\mu + s\zeta)$ when ζ is standard normal

This will allow us to compare our results to the analytical solutions

```
def v_star(y, 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(y)
```

(continues on next page)

(continued from previous page)

```
def σ_star(y, α, β):
    """
    True optimal policy
    """
    return (1 - α * β) * y
```

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'.

These are available in a class called `OptimalGrowthModel` that we constructed in an *earlier lecture*.

```
from numba import float64
from numba.experimental import jitclass

opt_growth_data = [
    ('α', float64),          # Production parameter
    ('β', float64),          # Discount factor
    ('μ', float64),          # Shock location parameter
    ('s', float64),          # Shock scale parameter
    ('grid', float64[:]),    # Grid (array)
    ('shocks', float64[:])   # Shock draws (array)
]

@jitclass(opt_growth_data)
class OptimalGrowthModel:

    def __init__(self,
                 α=0.4,
                 β=0.96,
                 μ=0,
                 s=0.1,
                 grid_max=4,
                 grid_size=120,
                 shock_size=250,
                 seed=1234):

        self.α, self.β, self.μ, self.s = α, β, μ, s

        # Set up grid
        self.grid = np.linspace(1e-5, grid_max, grid_size)

        # Store shocks (with a seed, so results are reproducible)
        np.random.seed(seed)
        self.shocks = np.exp(μ + s * np.random.randn(shock_size))


    def f(self, k):
        "The production function"
        return k**self.α


    def u(self, c):
        "The utility function"
        return np.log(c)

    def f_prime(self, k):
```

(continues on next page)

(continued from previous page)

```
        "Derivative of f"
        return self.α * (k**(self.α - 1))


    def u_prime(self, c):
        "Derivative of u"
        return 1/c

    def u_prime_inv(self, c):
        "Inverse of u'"
        return 1/c
```

Now we implement a method called `euler_diff`, which returns

$$u'(c) - \beta \int (u' \circ \sigma)(f(y-c)z) f'(y-c) z \phi(dz) \tag{47.8}$$

```
@njit
def euler_diff(c, σ, y, og):
    """
    Set up a function such that the root with respect to c,
    given y and σ, is equal to Kσ(y).

    """

    β, shocks, grid = og.β, og.shocks, og.grid
    f, f_prime, u_prime = og.f, og.f_prime, og.u_prime

    # First turn σ into a function via interpolation
    σ_func = lambda x: interp(grid, σ, x)

    # Now set up the function we need to find the root of.
    vals = u_prime(σ_func(f(y - c) * shocks)) * f_prime(y - c) * 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 (47.8) for c given state y and σ, the current guess of the policy.

Here's the operator K, that implements the root-finding step.

```
@njit
def K(σ, og):
    """
    The Coleman-Reffett operator

     Here og is an instance of OptimalGrowthModel.
    """

    β = og.β
    f, f_prime, u_prime = og.f, og.f_prime, og.u_prime
    grid, shocks = og.grid, og.shocks

    σ_new = np.empty_like(σ)
    for i, y in enumerate(grid):
        # Solve for optimal c at y
```

(continues on next page)

(continued from previous page)

```
        c_star = brentq(euler_diff, 1e-10, y-1e-10, args=(σ, y, og))[0]
        σ_new[i] = c_star

    return σ_new
```

47.3.1 Testing

Let's generate an instance and plot some iterates of K, starting from $\sigma(y) = y$.

```
og = OptimalGrowthModel()
grid = og.grid

n = 15
σ = grid.copy()  # Set initial condition

fig, ax = plt.subplots()
lb = 'initial condition $\sigma(y) = y$'
ax.plot(grid, σ, color=plt.cm.jet(0), alpha=0.6, label=lb)

for i in range(n):
    σ = K(σ, og)
    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, og)
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 *the previous lecture*.

Here is a function called `solve_model_time_iter` that takes an instance of `OptimalGrowthModel` and returns

an approximation to the optimal policy, using time iteration.

```
def solve_model_time_iter(model,     # Class with model information
                          σ,         # Initial condition
                          tol=1e-4,
                          max_iter=1000,
                          verbose=True,
                          print_skip=25):

    # Set up loop
    i = 0
    error = tol + 1

    while i < max_iter and error > tol:
        σ_new = K(σ, model)
        error = np.max(np.abs(σ - σ_new))
        i += 1
        if verbose and i % print_skip == 0:
            print(f"Error at iteration {i} is {error}.")
        σ = σ_new

    if error > tol:
        print("Failed to converge!")
    elif verbose:
        print(f"\nConverged in {i} iterations.")

    return σ_new
```

Let's call it:

```
σ_init = np.copy(og.grid)
σ = solve_model_time_iter(og, σ_init)
```

```
Converged in 11 iterations.
```

Here is a plot of the resulting policy, compared with the true policy:

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

ax.plot(og.grid, σ, lw=2,
        alpha=0.8, label='approximate policy function')

ax.plot(og.grid, σ_star(og.grid, og.α, og.β), 'k--',
        lw=2, alpha=0.8, label='true policy function')

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

Again, the fit is excellent.

The maximal absolute deviation between the two policies is

```
np.max(np.abs(σ - σ_star(og.grid, og.α, og.β)))
```

```
2.5329106132954138e-05
```

How long does it take to converge?

```
%%timeit -n 3 -r 1
σ = solve_model_time_iter(og, σ_init, verbose=False)
```

```
198 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 3 loops each)
```

Convergence is very fast, even compared to our *JIT-compiled value function iteration*.

Overall, we find that time iteration provides a very high degree of efficiency and accuracy, at least for this model.

47.4 Exercises

Exercise 47.4.1

Solve the model with CRRA utility

$$u(c) = \frac{c^{1-\gamma}}{1-\gamma}$$

Set `γ = 1.5`.

Compute and plot the optimal policy.

Solution to Exercise 47.4.1

We use the class `OptimalGrowthModel_CRRA` from our *VFI lecture*.

```
from numba import float64
from numba.experimental import jitclass

opt_growth_data = [
    ('α', float64),          # Production parameter
    ('β', float64),          # Discount factor
    ('μ', float64),          # Shock location parameter
    ('γ', float64),          # Preference parameter
    ('s', float64),          # Shock scale parameter
    ('grid', float64[:]),    # Grid (array)
    ('shocks', float64[:])   # Shock draws (array)
]

@jitclass(opt_growth_data)
class OptimalGrowthModel_CRRA:

    def __init__(self,
                 α=0.4,
                 β=0.96,
                 μ=0,
                 s=0.1,
                 γ=1.5,
                 grid_max=4,
                 grid_size=120,
                 shock_size=250,
                 seed=1234):

        self.α, self.β, self.γ, self.μ, self.s = α, β, γ, μ, s

        # Set up grid
        self.grid = np.linspace(1e-5, grid_max, grid_size)

        # Store shocks (with a seed, so results are reproducible)
        np.random.seed(seed)
        self.shocks = np.exp(μ + s * np.random.randn(shock_size))


    def f(self, k):
        "The production function."
        return k**self.α

    def u(self, c):
        "The utility function."
        return c**(1 - self.γ) / (1 - self.γ)

    def f_prime(self, k):
        "Derivative of f."
        return self.α * (k**(self.α - 1))

    def u_prime(self, c):
        "Derivative of u."
        return c**(-self.γ)

    def u_prime_inv(c):
        return c**(-1 / self.γ)
```

Let's create an instance:

```
og_crra = OptimalGrowthModel_CRRA()
```

Now we solve and plot the policy:

```
%%time
σ = solve_model_time_iter(og_crra, σ_init)


fig, ax = plt.subplots()

ax.plot(og.grid, σ, lw=2,
        alpha=0.8, label='approximate policy function')

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

```
Converged in 13 iterations.
```

```
CPU times: user 2 s, sys: 92.1 ms, total: 2.09 s
Wall time: 2.09 s
```

CHAPTER

FORTYEIGHT

OPTIMAL GROWTH IV: THE ENDOGENOUS GRID METHOD

Contents

- *Optimal Growth IV: The Endogenous Grid Method*
  - *Overview*
  - *Key Idea*
  - *Implementation*

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install interpolation
```

48.1 Overview

Previously, we solved the stochastic optimal growth model 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 [Car06].

Let's start with some standard imports:

```
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
from interpolation import interp
from numba import njit
```

```
/opt/conda/envs/quantecon/lib/python3.11/site-packages/numba/core/decorators.
↪py:262: NumbaDeprecationWarning: numba.generated_jit is deprecated. Please see␣
↪the documentation at: https://numba.readthedocs.io/en/stable/reference/
↪deprecation.html#deprecation-of-generated-jit for more information and advice on␣
↪a suitable replacement.
  warnings.warn(msg, NumbaDeprecationWarning)
```

48.2 Key Idea

Let's start by reminding ourselves of the theory and then see how the numerics fit in.

48.2.1 Theory

Take the model set out in *the time iteration lecture*, following the same terminology and notation.

The Euler equation is

$$(u' \circ \sigma^*)(y) = \beta \int (u' \circ \sigma^*)(f(y - \sigma^*(y))z) f'(y - \sigma^*(y)) z \phi(dz) \tag{48.1}$$

As we saw, the Coleman-Reffett operator is a nonlinear operator K engineered so that 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 $(K\sigma)(y)$ is the $c \in (0, \infty)$ that solves

$$u'(c) = \beta \int (u' \circ \sigma)(f(y - c)z) f'(y - c) z \phi(dz) \tag{48.2}$$

48.2.2 Exogenous Grid

As discussed in *the lecture on time iteration*, to implement the method on a computer, we need a numerical approximation.

In particular, 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.

Previously, to obtain a finite representation of an updated consumption policy, we

- fixed a grid of income points $\{y_i\}$
- calculated the consumption value c_i corresponding to each y_i using (48.2) and a root-finding routine

Each c_i is then interpreted as the value of the function $K\sigma$ at y_i.

Thus, with the points $\{y_i, c_i\}$ in hand, we can reconstruct $K\sigma$ via approximation.

Iteration then continues…

48.2.3 Endogenous Grid

The method discussed above requires a root-finding routine to find the c_i corresponding to a given income value y_i.

Root-finding is costly because it typically involves a significant number of function evaluations.

As pointed out by Carroll [Car06], we can avoid this if y_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 $\{k_i\}$ for capital ($k = y - c$).
- Then we obtain c_i via

$$c_i = (u')^{-1} \left\{ \beta \int (u' \circ \sigma)(f(k_i)z) \, f'(k_i) \, z \, \phi(dz) \right\} \tag{48.3}$$

- Finally, for each c_i we set $y_i = c_i + k_i$.

It is clear that each (y_i, c_i) pair constructed in this manner satisfies (48.2).

With the points $\{y_i, c_i\}$ in hand, we can reconstruct $K\sigma$ via approximation as before.

The name EGM comes from the fact that the grid $\{y_i\}$ is determined **endogenously**.

48.3 Implementation

As *before*, we will start with a simple setting where

- $u(c) = \ln c$,
- production is Cobb-Douglas, and
- the shocks are lognormal.

This will allow us to make comparisons with the analytical solutions

```
def v_star(y, 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(y)

def σ_star(y, α, β):
    """
    True optimal policy
    """
    return (1 - α * β) * y
```

We reuse the `OptimalGrowthModel` class

```python
from numba import float64
from numba.experimental import jitclass

opt_growth_data = [
    ('α', float64),          # Production parameter
    ('β', float64),          # Discount factor
    ('μ', float64),          # Shock location parameter
    ('s', float64),          # Shock scale parameter
    ('grid', float64[:]),    # Grid (array)
    ('shocks', float64[:])   # Shock draws (array)
]

@jitclass(opt_growth_data)
class OptimalGrowthModel:

    def __init__(self,
                 α=0.4,
                 β=0.96,
                 μ=0,
                 s=0.1,
                 grid_max=4,
                 grid_size=120,
                 shock_size=250,
                 seed=1234):

        self.α, self.β, self.μ, self.s = α, β, μ, s

        # Set up grid
        self.grid = np.linspace(1e-5, grid_max, grid_size)

        # Store shocks (with a seed, so results are reproducible)
        np.random.seed(seed)
        self.shocks = np.exp(μ + s * np.random.randn(shock_size))


    def f(self, k):
        "The production function"
        return k**self.α


    def u(self, c):
        "The utility function"
        return np.log(c)

    def f_prime(self, k):
        "Derivative of f"
        return self.α * (k**(self.α - 1))


    def u_prime(self, c):
        "Derivative of u"
        return 1/c

    def u_prime_inv(self, c):
        "Inverse of u'"
        return 1/c
```

48.3.1 The Operator

Here's an implementation of K using EGM as described above.

```
@njit
def K(σ_array, og):
    """
    The Coleman-Reffett operator using EGM

    """

    # Simplify names
    f, β = og.f, og.β
    f_prime, u_prime = og.f_prime, og.u_prime
    u_prime_inv = og.u_prime_inv
    grid, shocks = og.grid, og.shocks

    # Determine endogenous grid
    y = grid + σ_array  # y_i = k_i + c_i

    # Linear interpolation of policy using endogenous grid
    σ = lambda x: interp(y, σ_array, x)

    # Allocate memory for new consumption array
    c = np.empty_like(grid)

    # Solve for updated consumption value
    for i, k in enumerate(grid):
        vals = u_prime(σ(f(k) * shocks)) * f_prime(k) * shocks
        c[i] = u_prime_inv(β * np.mean(vals))

    return c
```

Note the lack of any root-finding algorithm.

48.3.2 Testing

First we create an instance.

```
og = OptimalGrowthModel()
grid = og.grid
```

Here's our solver routine:

```
def solve_model_time_iter(model,    # Class with model information
                          σ,        # Initial condition
                          tol=1e-4,
                          max_iter=1000,
                          verbose=True,
                          print_skip=25):

    # Set up loop
    i = 0
    error = tol + 1
```

(continues on next page)

(continued from previous page)

```
    while i < max_iter and error > tol:
        σ_new = K(σ, model)
        error = np.max(np.abs(σ - σ_new))
        i += 1
        if verbose and i % print_skip == 0:
            print(f"Error at iteration {i} is {error}.")
        σ = σ_new

    if error > tol:
        print("Failed to converge!")
    elif verbose:
        print(f"\nConverged in {i} iterations.")

    return σ_new
```

Let's call it:

```
σ_init = np.copy(grid)
σ = solve_model_time_iter(og, σ_init)
```

```
Converged in 12 iterations.
```

Here is a plot of the resulting policy, compared with the true policy:

```
y = grid + σ  # y_i = k_i + c_i

fig, ax = plt.subplots()

ax.plot(y, σ, lw=2,
        alpha=0.8, label='approximate policy function')

ax.plot(y, σ_star(y, og.α, og.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(σ - σ_star(y, og.α, og.β)))
```

```
1.530274914252061e-05
```

How long does it take to converge?

```
%%timeit -n 3 -r 1
σ = solve_model_time_iter(og, σ_init, verbose=False)
```

```
30 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 3 loops each)
```

Relative to time iteration, which as already found to be highly efficient, EGM has managed to shave off still more run time without compromising accuracy.

This is due to the lack of a numerical root-finding step.

We can now solve the optimal growth model at given parameters extremely fast.

CHAPTER

FORTYNINE

THE INCOME FLUCTUATION PROBLEM I: BASIC MODEL

Contents

- *The Income Fluctuation Problem I: Basic Model*
  - *Overview*
  - *The Optimal Savings Problem*
  - *Computation*
  - *Implementation*
  - *Exercises*

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install quantecon
!pip install interpolation
```

49.1 Overview

In this lecture, we study an optimal savings problem for an infinitely lived consumer—the "common ancestor" described in [LS18], section 1.3.

This is an essential sub-problem for many representative macroeconomic models

- [Aiy94]
- [Hug93]
- etc.

It is related to the decision problem in the *stochastic optimal growth model* and yet differs in important ways.

For example, the choice problem for the agent includes an additive income term that leads to an occasionally binding constraint.

Moreover, in this and the following lectures, we will inject more realistic features such as correlated shocks.

To solve the model we will use Euler equation based time iteration, which proved to be *fast and accurate* in our investigation of the *stochastic optimal growth model*.

Time iteration is globally convergent under mild assumptions, even when utility is unbounded (both above and below).

We'll need the following imports:

```
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
from quantecon.optimize import brentq
from interpolation import interp
from numba import njit, float64
from numba.experimental import jitclass
from quantecon import MarkovChain
```

```
/opt/conda/envs/quantecon/lib/python3.11/site-packages/numba/core/decorators.
↪py:262: NumbaDeprecationWarning: numba.generated_jit is deprecated. Please see␣
↪the documentation at: https://numba.readthedocs.io/en/stable/reference/
↪deprecation.html#deprecation-of-generated-jit for more information and advice on␣
↪a suitable replacement.
  warnings.warn(msg, NumbaDeprecationWarning)
```

49.1.1 References

Our presentation is a simplified version of [MST20].

Other references include [Dea91], [DH10], [Kuh13], [Rab02], [Rei09] and [SE77].

49.2 The Optimal Savings Problem

Let's write down the model and then discuss how to solve it.

49.2.1 Set-Up

Consider 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

$$a_{t+1} \leq R(a_t - c_t) + Y_{t+1}, \quad c_t \geq 0, \quad a_t \geq 0 \quad t = 0, 1, \ldots \tag{49.1}$$

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 chooses consumption c_t.
2. Labor is supplied by the household throughout the period and labor income Y_{t+1} is received at the end of period t.

3. Financial income $R(a_t - c_t)$ is received at the end of period t.
4. Time shifts to $t+1$ and the process repeats.

Non-capital income Y_t is given by $Y_t = y(Z_t)$, where $\{Z_t\}$ is an exogeneous state process.

As is common in the literature, we take $\{Z_t\}$ to be a finite state Markov chain taking values in Z with Markov matrix P.

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 $\lim_{c \to \infty} u'(c) = 0$

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 (49.1), and
3. measurability, 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.

49.2.2 Value Function and Euler Equation

The *value function* $V \colon \mathsf{S} \to \mathbb{R}$ is defined by

$$V(a, z) := \max \mathbb{E} \left\{ \sum_{t=0}^{\infty} \beta^t u(c_t) \right\} \tag{49.2}$$

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 attains the supremum in (49.2).

To pin down such paths we can use a version of the Euler equation, which in the present setting is

$$u'(c_t) \geq \beta R \, \mathbb{E}_t u'(c_{t+1}) \tag{49.3}$$

and

$$c_t < a_t \implies u'(c_t) = \beta R \, \mathbb{E}_t u'(c_{t+1}) \tag{49.4}$$

When $c_t = a_t$ we obviously have $u'(c_t) = u'(a_t)$,

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 lower boundary case $c_t = 0$ never arises at the optimum because $u'(0) = \infty$.)

With some thought, one can show that (49.3) and (49.4) are equivalent to

$$u'(c_t) = \max \left\{ \beta R \, \mathbb{E}_t u'(c_{t+1}) \, , \; u'(a_t) \right\} \tag{49.5}$$

49.2.3 Optimality Results

As shown in [MST20],

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 equality (49.5) and the transversality condition

$$\lim_{t \to \infty} \beta^t \, \mathbb{E} \left[u'(c_t) a_{t+1} \right] = 0 \tag{49.6}$$

Moreover, there exists an *optimal consumption function* $\sigma^* \colon \mathsf{S} \to \mathbb{R}_+$ 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 (49.5) and (49.6), and hence is the unique optimal path from (a, z).

Thus, to solve the optimization problem, we need to compute the policy σ^*.

49.3 Computation

There are two standard ways to solve for σ^*

1. time iteration using the Euler equality and
2. value function iteration.

Our investigation of the cake eating problem and stochastic optimal growth model suggests that time iteration will be faster and more accurate.

This is the approach that we apply below.

49.3.1 Time Iteration

We can rewrite (49.5) to make it a statement about functions rather than random variables.

In particular, consider the functional equation

$$(u' \circ \sigma)(a, z) = \max \left\{ \beta R \, \mathbb{E}_z (u' \circ \sigma)[R(a - \sigma(a, z)) + \hat{Y}, \, \hat{Z}] \, , \; u'(a) \right\} \tag{49.7}$$

where

- $(u' \circ \sigma)(s) := u'(\sigma(s))$.
- $\mathbb{E}_z$ conditions on current state z and $\hat{X}$ indicates next period value of random variable X and
- σ is the unknown function.

We need a suitable class of candidate solutions for the optimal consumption policy.

The right way to pick such a class is to consider what properties the solution is likely to have, in order to restrict the search space and ensure that iteration is well behaved.

To this end, let $\mathcal{C}$ be the space of continuous functions $\sigma \colon \mathbf{S} \to \mathbb{R}$ such that σ is increasing in the first argument, $0 < \sigma(a, z) \leq a$ for all $(a, z) \in \mathbf{S}$, and

$$\sup_{(a,z) \in \mathbf{S}} |(u' \circ \sigma)(a, z) - u'(a)| < \infty \tag{49.8}$$

This will be our candidate class.

In addition, let $K \colon \mathcal{C} \to \mathcal{C}$ be defined as follows.

For given $\sigma \in \mathcal{C}$, the value $K\sigma(a, z)$ is the unique $c \in [0, a]$ that solves

$$u'(c) = \max \left\{ \beta R \, \mathbb{E}_z (u' \circ \sigma) \left[R(a - c) + \hat{Y}, \, \hat{Z} \right], \; u'(a) \right\} \tag{49.9}$$

We refer to K as the Coleman–Reffett operator.

The operator K is constructed so that fixed points of K coincide with solutions to the functional equation (49.7).

It is shown in [MST20] that the unique optimal policy can be computed by picking any $\sigma \in \mathcal{C}$ and iterating with the operator K defined in (49.9).

49.3.2 Some Technical Details

The proof of the last statement is somewhat technical but here is a quick summary:

It is shown in [MST20] that K is a contraction mapping on $\mathcal{C}$ under the metric

$$\rho(c, d) := \| \, u' \circ \sigma_1 - u' \circ \sigma_2 \, \| := \sup_{s \in S} | \, u'(\sigma_1(s)) - u'(\sigma_2(s)) \, | \qquad (\sigma_1, \sigma_2 \in \mathcal{C})$$

which evaluates the maximal difference in terms of marginal utility.

(The benefit of this measure of distance is that, while elements of $\mathcal{C}$ are not generally bounded, ρ is always finite under our assumptions.)

It is also shown that the metric ρ is complete on $\mathcal{C}$.

In consequence, K has a unique fixed point $\sigma^* \in \mathcal{C}$ and $K^n c \to \sigma^*$ as $n \to \infty$ for any $\sigma \in \mathcal{C}$.

By the definition of K, the fixed points of K in $\mathcal{C}$ coincide with the solutions to (49.7) in $\mathcal{C}$.

As a consequence, the path $\{c_t\}$ generated from $(a_0, z_0) \in S$ using policy function σ^* is the unique optimal path from $(a_0, z_0) \in S$.

49.4 Implementation

We use the CRRA utility specification

$$u(c) = \frac{c^{1-\gamma}}{1 - \gamma}$$

The exogeneous state process $\{Z_t\}$ defaults to a two-state Markov chain with state space $\{0, 1\}$ and transition matrix P.

Here we build a class called `IFP` that stores the model primitives.

```
ifp_data = [
    ('R', float64),              # Interest rate 1 + r
    ('β', float64),              # Discount factor
    ('γ', float64),              # Preference parameter
    ('P', float64[:, :]),        # Markov matrix for binary Z_t
    ('y', float64[:]),           # Income is Y_t = y[Z_t]
    ('asset_grid', float64[:])   # Grid (array)
]

@jitclass(ifp_data)
class IFP:
```

(continues on next page)

(continued from previous page)

```
    def __init__(self,
                 r=0.01,
                 β=0.96,
                 γ=1.5,
                 P=((0.6, 0.4),
                    (0.05, 0.95)),
                 y=(0.0, 2.0),
                 grid_max=16,
                 grid_size=50):

        self.R = 1 + r
        self.β, self.γ = β, γ
        self.P, self.y = np.array(P), np.array(y)
        self.asset_grid = np.linspace(0, grid_max, grid_size)

        # Recall that we need R β < 1 for convergence.
        assert self.R * self.β < 1, "Stability condition violated."

    def u_prime(self, c):
        return c**(-self.γ)
```

Next we provide a function to compute the difference

$$u'(c) - \max\left\{\beta R\, \mathbb{E}_z (u' \circ \sigma)\left[R(a-c) + \hat{Y},\, \hat{Z}\right],\; u'(a)\right\} \tag{49.10}$$

```
@njit
def euler_diff(c, a, z, σ_vals, ifp):
    """
    The difference between the left- and right-hand side
    of the Euler Equation, given current policy σ.

        * c is the consumption choice
        * (a, z) is the state, with z in {0, 1}
        * σ_vals is a policy represented as a matrix.
        * ifp is an instance of IFP

    """

    # Simplify names
    R, P, y, β, γ  = ifp.R, ifp.P, ifp.y, ifp.β, ifp.γ
    asset_grid, u_prime = ifp.asset_grid, ifp.u_prime
    n = len(P)

    # Convert policy into a function by linear interpolation
    def σ(a, z):
        return interp(asset_grid, σ_vals[:, z], a)

    # Calculate the expectation conditional on current z
    expect = 0.0
    for z_hat in range(n):
        expect += u_prime(σ(R * (a - c) + y[z_hat], z_hat)) * P[z, z_hat]

    return u_prime(c) - max(β * R * expect, u_prime(a))
```

Note that we use linear interpolation along the asset grid to approximate the policy function.

The next step is to obtain the root of the Euler difference.

```
@njit
def K(σ, ifp):
    """
    The operator K.

    """
    σ_new = np.empty_like(σ)
    for i, a in enumerate(ifp.asset_grid):
        for z in (0, 1):
            result = brentq(euler_diff, 1e-8, a, args=(a, z, σ, ifp))
            σ_new[i, z] = result.root

    return σ_new
```

With the operator K in hand, we can choose an initial condition and start to iterate.

The following function iterates to convergence and returns the approximate optimal policy.

```
def solve_model_time_iter(model,    # Class with model information
                          σ,        # Initial condition
                          tol=1e-4,
                          max_iter=1000,
                          verbose=True,
                          print_skip=25):

    # Set up loop
    i = 0
    error = tol + 1

    while i < max_iter and error > tol:
        σ_new = K(σ, model)
        error = np.max(np.abs(σ - σ_new))
        i += 1
        if verbose and i % print_skip == 0:
            print(f"Error at iteration {i} is {error}.")
        σ = σ_new

    if error > tol:
        print("Failed to converge!")
    elif verbose:
        print(f"\nConverged in {i} iterations.")

    return σ_new
```

Let's carry this out using the default parameters of the `IFP` class:

```
ifp = IFP()

# Set up initial consumption policy of consuming all assets at all z
z_size = len(ifp.P)
a_grid = ifp.asset_grid
a_size = len(a_grid)
σ_init = np.repeat(a_grid.reshape(a_size, 1), z_size, axis=1)

σ_star = solve_model_time_iter(ifp, σ_init)
```

```
Error at iteration 25 is 0.011629589188247191.
Error at iteration 50 is 0.0003857183099467143.

Converged in 60 iterations.
```

Here's a plot of the resulting policy for each exogeneous state z.

```
fig, ax = plt.subplots()
for z in range(z_size):
    label = rf'$\sigma^*(\cdot, {z})$'
    ax.plot(a_grid, σ_star[:, z], label=label)
ax.set(xlabel='assets', ylabel='consumption')
ax.legend()
plt.show()
```

The following exercises walk you through several applications where policy functions are computed.

49.4.1 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 cake eating problem.

We know that, in this case, 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):
```

(continues on next page)

(continued from previous page)

```
    return (1 - β**(1 / γ))**(-γ) * (x**(1-γ) / (1-γ))
```

Let's see if we match up:

```
ifp_cake_eating = IFP(r=0.0, γ=(0.0, 0.0))

σ_star = solve_model_time_iter(ifp_cake_eating, σ_init)

fig, ax = plt.subplots()
ax.plot(a_grid, σ_star[:, 0], label='numerical')
ax.plot(a_grid, c_star(a_grid, ifp.β, ifp.γ), '--', label='analytical')

ax.set(xlabel='assets', ylabel='consumption')
ax.legend()

plt.show()
```

```
Error at iteration 25 is 0.023332272630545603.
Error at iteration 50 is 0.005301238424249788.
Error at iteration 75 is 0.0019706324625650695.
Error at iteration 100 is 0.0008675521337955794.
Error at iteration 125 is 0.00041073542212249903.
Error at iteration 150 is 0.00020120334010509389.
Error at iteration 175 is 0.00010021430795081887.

Converged in 176 iterations.
```

Success!

49.5 Exercises

Exercise 49.5.1

Let's consider how the interest rate affects consumption.

Reproduce the following figure, which shows (approximately) optimal consumption policies for different interest rates

- Other than `r`, all parameters are at their default values.
- `r` steps through `np.linspace(0, 0.04, 4)`.
- Consumption is plotted against assets for income shock fixed at the smallest value.

The figure shows that higher interest rates boost savings and hence suppress consumption.

Solution to Exercise 49.5.1

Here's one solution:

```
r_vals = np.linspace(0, 0.04, 4)

fig, ax = plt.subplots()
for r_val in r_vals:
    ifp = IFP(r=r_val)
    σ_star = solve_model_time_iter(ifp, σ_init, verbose=False)
    ax.plot(ifp.asset_grid, σ_star[:, 0], label=f'$r = {r_val:.3f}$')

ax.set(xlabel='asset level', ylabel='consumption (low income)')
ax.legend()
plt.show()
```

Exercise 49.5.2

Now let's consider the long run asset levels held by households under the default parameters.

The following figure is a 45 degree diagram showing the law of motion for assets when consumption is optimal

```
ifp = IFP()

σ_star = solve_model_time_iter(ifp, σ_init, verbose=False)
a = ifp.asset_grid
R, y = ifp.R, ifp.y

fig, ax = plt.subplots()
for z, lb in zip((0, 1), ('low income', 'high income')):
    ax.plot(a, R * (a - σ_star[:, z]) + y[z] , label=lb)

ax.plot(a, a, '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

$$a \mapsto R(a - \sigma^*(a, z)) + y(z)$$

The dashed line is the 45 degree line.

We can see from the figure that the dynamics will be stable — assets do not diverge even in the highest state.

In fact there is a unique stationary distribution of assets that we can calculate by simulation

- Can be proved via theorem 2 of [HP92].
- It represents the long run dispersion of assets across households when households have idiosyncratic shocks.

Ergodicity is valid here, so stationary probabilities can be calculated by averaging over a single long time series.

Hence to approximate the stationary distribution we can simulate a long time series for assets and histogram it.

Your task is to generate such a histogram.

- Use a single time series $\{a_t\}$ of length 500,000.
- Given the length of this time series, the initial condition (a_0, z_0) will not matter.
- You might find it helpful to use the `MarkovChain` class from `quantecon`.

Solution to Exercise 49.5.2

First we write a function to compute a long asset series.

```
def compute_asset_series(ifp, T=500_000, seed=1234):
    """
    Simulates a time series of length T for assets, given optimal
    savings behavior.

    ifp is an instance of IFP
    """
    P, y, R = ifp.P, ifp.y, ifp.R  # Simplify names
```

(continues on next page)

(continued from previous page)

```
    # Solve for the optimal policy
    σ_star = solve_model_time_iter(ifp, σ_init, verbose=False)
    σ = lambda a, z: interp(ifp.asset_grid, σ_star[:, z], a)

    # Simulate the exogeneous state process
    mc = MarkovChain(P)
    z_seq = mc.simulate(T, random_state=seed)

    # Simulate the asset path
    a = np.zeros(T+1)
    for t in range(T):
        z = z_seq[t]
        a[t+1] = R * (a[t] - σ(a[t], z)) + y[z]
    return a
```

Now we call the function, generate the series and then histogram it:

```
ifp = IFP()
a = compute_asset_series(ifp)

fig, ax = plt.subplots()
ax.hist(a, bins=20, alpha=0.5, density=True)
ax.set(xlabel='assets')
plt.show()
```

The shape of the asset distribution is unrealistic.

Here it is left skewed when in reality it has a long right tail.

In a *subsequent lecture* we will rectify this by adding more realistic features to the model.

Exercise 49.5.3

Following on from exercises 1 and 2, let's look at how savings and aggregate asset holdings vary with the interest rate

Note: [LS18] 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.

Following tradition, put the price (i.e., interest rate) on the vertical axis.

On the horizontal axis put aggregate capital, computed as the mean of the stationary distribution given the interest rate.

Solution to Exercise 49.5.3

Here's one solution

```
M = 25
r_vals = np.linspace(0, 0.02, M)
fig, ax = plt.subplots()

asset_mean = []
for r in r_vals:
    print(f'Solving model at r = {r}')
    ifp = IFP(r=r)
    mean = np.mean(compute_asset_series(ifp, T=250_000))
    asset_mean.append(mean)
ax.plot(asset_mean, r_vals)

ax.set(xlabel='capital', ylabel='interest rate')

plt.show()
```

```
Solving model at r = 0.0
```

```
Solving model at r = 0.0008333333333333334
```

```
Solving model at r = 0.0016666666666666668
```

```
Solving model at r = 0.0025
```

```
Solving model at r = 0.0033333333333333335
```

```
Solving model at r = 0.004166666666666667
```

```
Solving model at r = 0.005
```

```
Solving model at r = 0.005833333333333334
```

```
Solving model at r = 0.006666666666666667
```

```
Solving model at r = 0.007500000000000001
```

```
Solving model at r = 0.008333333333333333
```

```
Solving model at r = 0.009166666666666667
```

```
Solving model at r = 0.01
```

```
Solving model at r = 0.010833333333333334
```

```
Solving model at r = 0.011666666666666667
```

```
Solving model at r = 0.0125
```

```
Solving model at r = 0.013333333333333334
```

```
Solving model at r = 0.014166666666666668
```

```
Solving model at r = 0.015000000000000001
```

```
Solving model at r = 0.015833333333333335
```

```
Solving model at r = 0.016666666666666666
```

```
Solving model at r = 0.0175
```

```
Solving model at r = 0.018333333333333333
```

```
Solving model at r = 0.01916666666666667
```

```
Solving model at r = 0.02
```

As expected, aggregate savings increases with the interest rate.

CHAPTER

FIFTY

THE INCOME FLUCTUATION PROBLEM II: STOCHASTIC RETURNS ON ASSETS

Contents

- *The Income Fluctuation Problem II: Stochastic Returns on Assets*
  - *Overview*
  - *The Savings Problem*
  - *Solution Algorithm*
  - *Implementation*
  - *Exercises*

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install quantecon
!pip install interpolation
```

50.1 Overview

In this lecture, we continue our study of the *income fluctuation problem*.

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., [BBZ15] or [ST19]).

Theoretical properties of the household savings model presented here are analyzed in detail in [MST20].

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
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
from interpolation import interp
```

(continues on next page)

(continued from previous page)

```
from numba import njit, float64
from numba.experimental import jitclass
from quantecon import MarkovChain
```

```
/opt/conda/envs/quantecon/lib/python3.11/site-packages/numba/core/decorators.
↪py:262: NumbaDeprecationWarning: numba.generated_jit is deprecated. Please see␣
↪the documentation at: https://numba.readthedocs.io/en/stable/reference/
↪deprecation.html#deprecation-of-generated-jit for more information and advice on␣
↪a suitable replacement.
  warnings.warn(msg, NumbaDeprecationWarning)
```

50.2 The Savings Problem

In this section we review the household problem and optimality results.

50.2.1 Set Up

A household chooses a consumption-asset path $\{(c_t, a_t)\}$ to maximize

$$\mathbb{E}\left\{\sum_{t=0}^{\infty} \beta^t u(c_t)\right\} \tag{50.1}$$

subject to

$$a_{t+1} = R_{t+1}(a_t - c_t) + Y_{t+1} \text{ and } 0 \leq c_t \leq a_t, \tag{50.2}$$

with initial condition $(a_0, Z_0) = (a, z)$ treated as given.

Note that $\{R_t\}_{t\geq 1}$, the gross rate of return on wealth, is allowed to be stochastic.

The sequence $\{Y_t\}_{t\geq 1}$ is non-financial income.

The stochastic components of the problem obey

$$R_t = R(Z_t, \zeta_t) \quad \text{and} \quad Y_t = Y(Z_t, \eta_t), \tag{50.3}$$

where

- the maps 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 an irreducible time-homogeneous Markov chain on a finite set Z

Let P represent the Markov matrix for the chain $\{Z_t\}_{t\geq 0}$.

Our assumptions on preferences are the same as our *previous lecture* on the income fluctuation problem.

As before, $\mathbb{E}_z \hat{X}$ means expectation of next period value $\hat{X}$ given current value $Z = z$.

50.2.2 Assumptions

We need restrictions to ensure that the objective (50.1) 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 $\beta R < 1$.

Now it is stochastic, we require that

$$\beta G_R < 1, \quad \text{where} \quad G_R := \lim_{n \to \infty} \left(\mathbb{E} \prod_{t=1}^{n} R_t \right)^{1/n} \tag{50.4}$$

Notice that, when $\{R_t\}$ takes some constant value R, this reduces to the previous restriction $\beta R < 1$

The value G_R can be thought of as the long run (geometric) average gross rate of return.

More intuition behind (50.4) is provided in [MST20].

Discussion on how to check it is given 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 [BBZ15].

50.2.3 Optimality

Let the class of candidate consumption policies $\mathcal{C}$ be defined *as before*.

In [MST20] 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

$$(u' \circ \sigma)(a, z) = \max \left\{ \beta \, \mathbb{E}_z \, \hat{R} \, (u' \circ \sigma)[\hat{R}(a - \sigma(a, z)) + \hat{Y}, \, \hat{Z}], \, u'(a) \right\} \tag{50.5}$$

(Intuition and derivation are similar to our *earlier lecture* on the income fluctuation problem.)

We again solve the Euler equation using time iteration, iterating with a Coleman–Reffett operator K defined to match the Euler equation (50.5).

50.3 Solution Algorithm

50.3.1 A Time Iteration Operator

Our definition of the candidate class $\sigma \in \mathcal{C}$ of consumption policies is the same as in our *earlier lecture* on the income fluctuation problem.

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

$$u'(\xi) = \max \left\{ \beta \, \mathbb{E}_z \, \hat{R} \, (u' \circ \sigma)[\hat{R}(a - \xi) + \hat{Y}, \, \hat{Z}], \, u'(a) \right\} \tag{50.6}$$

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 [MST20] for more details).

50.3.2 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).

50.3.3 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 (50.5) drops out, and we solve for

$$c_i = (u')^{-1} \left\{ \beta \, \mathbb{E}_z \hat{R} (u' \circ \sigma) \left[\hat{R} s_i + \hat{Y}, \, \hat{Z} \right] \right\} \tag{50.7}$$

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.

50.3.4 Testing the Assumptions

Convergence of time iteration is dependent on the condition $\beta G_R < 1$ being satisfied.

One can check this using the fact that G_R is equal to the spectral radius of the matrix L defined by

$$L(z, \hat{z}) := P(z, \hat{z}) \int R(\hat{z}, x)\phi(x)dx$$

This identity is proved in [MST20], where ϕ is the density of the innovation ζ_t to returns on assets.

(Remember that Z is a finite set, so this expression defines a matrix.)

Checking the condition is even easier when $\{R_t\}$ 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.

50.4 Implementation

We will assume that $R_t = \exp(a_r \zeta_t + b_r)$ where a_r, b_r are constants and $\{\zeta_t\}$ is IID standard normal.

We allow labor income to be correlated, with

$$Y_t = \exp(a_y \eta_t + Z_t b_y)$$

where $\{\eta_t\}$ is also IID standard normal and $\{Z_t\}$ is a Markov chain taking values in $\{0, 1\}$.

```
ifp_data = [
    ('γ', float64),              # utility parameter
    ('β', float64),              # discount factor
    ('P', float64[:, :]),        # transition probs for z_t
    ('a_r', float64),            # scale parameter for R_t
    ('b_r', float64),            # additive parameter for R_t
    ('a_y', float64),            # scale parameter for Y_t
    ('b_y', float64),            # additive parameter for Y_t
    ('s_grid', float64[:]),      # Grid over savings
    ('η_draws', float64[:]),     # Draws of innovation η for MC
    ('ζ_draws', float64[:])      # Draws of innovation ζ for MC
]
```

```
@jitclass(ifp_data)
class IFP:
    """
    A class that stores primitives for the income fluctuation
```

(continues on next page)

(continued from previous page)

```
    problem.
    """

    def __init__(self,
                 γ=1.5,
                 β=0.96,
                 P=np.array([(0.9, 0.1),
                             (0.1, 0.9)]),
                 a_r=0.1,
                 b_r=0.0,
                 a_y=0.2,
                 b_y=0.5,
                 shock_draw_size=50,
                 grid_max=10,
                 grid_size=100,
                 seed=1234):

        np.random.seed(seed)  # arbitrary seed

        self.P, self.γ, self.β = P, γ, β
        self.a_r, self.b_r, self.a_y, self.b_y = a_r, b_r, a_y, b_y
        self.η_draws = np.random.randn(shock_draw_size)
        self.ζ_draws = np.random.randn(shock_draw_size)
        self.s_grid = np.linspace(0, grid_max, grid_size)

        # Test stability assuming {R_t} is IID and adopts the lognormal
        # specification given below.  The test is then β E R_t < 1.
        ER = np.exp(b_r + a_r**2 / 2)
        assert β * ER < 1, "Stability condition failed."

    # Marginal utility
    def u_prime(self, c):
        return c**(-self.γ)

    # Inverse of marginal utility
    def u_prime_inv(self, c):
        return c**(-1/self.γ)

    def R(self, z, ζ):
        return np.exp(self.a_r * ζ + self.b_r)

    def Y(self, z, η):
        return np.exp(self.a_y * η + (z * self.b_y))
```

Here's the Coleman-Reffett operator based on EGM:

```
@njit
def K(a_in, σ_in, ifp):
    """
    The Coleman--Reffett operator for the income fluctuation problem,
    using the endogenous grid method.

        * ifp is an instance of IFP
        * a_in[i, z] is an asset grid
        * σ_in[i, z] is consumption at a_in[i, z]
    """
```

(continues on next page)

(continued from previous page)

```
    # Simplify names
    u_prime, u_prime_inv = ifp.u_prime, ifp.u_prime_inv
    R, Y, P, β = ifp.R, ifp.Y, ifp.P, ifp.β
    s_grid, η_draws, ζ_draws = ifp.s_grid, ifp.η_draws, ifp.ζ_draws
    n = len(P)

    # Create consumption function by linear interpolation
    σ = lambda a, z: interp(a_in[:, z], σ_in[:, z], a)

    # Allocate memory
    σ_out = np.empty_like(σ_in)

    # Obtain c_i at each s_i, z, store in σ_out[i, z], computing
    # the expectation term by Monte Carlo
    for i, s in enumerate(s_grid):
        for z in range(n):
            # Compute expectation
            Ez = 0.0
            for z_hat in range(n):
                for η in ifp.η_draws:
                    for ζ in ifp.ζ_draws:
                        R_hat = R(z_hat, ζ)
                        Y_hat = Y(z_hat, η)
                        U = u_prime(σ(R_hat * s + Y_hat, z_hat))
                        Ez += R_hat * U * P[z, z_hat]
            Ez = Ez / (len(η_draws) * len(ζ_draws))
            σ_out[i, z] =  u_prime_inv(β * Ez)

    # Calculate endogenous asset grid
    a_out = np.empty_like(σ_out)
    for z in range(n):
        a_out[:, z] = s_grid + σ_out[:, z]

    # Fixing a consumption-asset pair at (0, 0) improves interpolation
    σ_out[0, :] = 0
    a_out[0, :] = 0

    return a_out, σ_out
```

The next function solves for an approximation of the optimal consumption policy via time iteration.

```
def solve_model_time_iter(model,        # Class with model information
                          a_vec,        # Initial condition for assets
                          σ_vec,        # Initial condition for consumption
                          tol=1e-4,
                          max_iter=1000,
                          verbose=True,
                          print_skip=25):

    # Set up loop
    i = 0
    error = tol + 1

    while i < max_iter and error > tol:
        a_new, σ_new = K(a_vec, σ_vec, model)
```

(continues on next page)

(continued from previous page)

```
        error = np.max(np.abs(σ_vec - σ_new))
        i += 1
        if verbose and i % print_skip == 0:
            print(f"Error at iteration {i} is {error}.")
        a_vec, σ_vec = np.copy(a_new), np.copy(σ_new)

    if error > tol:
        print("Failed to converge!")
    elif verbose:
        print(f"\nConverged in {i} iterations.")

    return a_new, σ_new
```

Now we are ready to create an instance at the default parameters.

```
ifp = IFP()
```

Next we set up an initial condition, which corresponds to consuming all assets.

```
# Initial guess of σ = consume all assets
k = len(ifp.s_grid)
n = len(ifp.P)
σ_init = np.empty((k, n))
for z in range(n):
    σ_init[:, z] = ifp.s_grid
a_init = np.copy(σ_init)
```

Let's generate an approximation solution.

```
a_star, σ_star = solve_model_time_iter(ifp, a_init, σ_init, print_skip=5)
```

```
Error at iteration 5 is 0.5081944529506557.
```

```
Error at iteration 10 is 0.1057246950930697.
```

```
Error at iteration 15 is 0.03658262202883744.
```

```
Error at iteration 20 is 0.013936729965906114.
```

```
Error at iteration 25 is 0.005292165269711546.
```

```
Error at iteration 30 is 0.0019748126990770665.
```

```
Error at iteration 35 is 0.0007219210463285108.
```

```
Error at iteration 40 is 0.0002590544496094971.
```

```
Error at iteration 45 is 9.163966595426842e-05.

Converged in 45 iterations.
```

Here's a plot of the resulting consumption policy.

```
fig, ax = plt.subplots()
for z in range(len(ifp.P)):
    ax.plot(a_star[:, z], σ_star[:, z], label=f"consumption when $z={z}$")

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

Notice that we consume all assets in the lower range of the asset space.

This is because we anticipate income Y_{t+1} tomorrow, which makes the need to save less urgent.

Can you explain why consuming all assets ends earlier (for lower values of assets) when $z = 0$?

50.4.1 Law of Motion

Let's try to get some idea of what will happen to assets over the long run under this consumption policy.

As with our *earlier lecture* on the income fluctuation problem, we begin by producing a 45 degree diagram showing the law of motion for assets

```
# Good and bad state mean labor income
Y_mean = [np.mean(ifp.Y(z, ifp.η_draws)) for z in (0, 1)]
# Mean returns
R_mean = np.mean(ifp.R(z, ifp.ζ_draws))

a = a_star
fig, ax = plt.subplots()
for z, lb in zip((0, 1), ('bad state', 'good state')):
    ax.plot(a[:, z], R_mean * (a[:, z] - σ_star[:, z]) + Y_mean[z] , label=lb)
```

(continues on next page)

(continued from previous page)

```
ax.plot(a[:, 0], a[:, 0], 'k--')
ax.set(xlabel='current assets', ylabel='next period assets')

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

The unbroken lines represent, for each z, an average update function for assets, given by

$$a \mapsto \bar{R}(a - \sigma^*(a, z)) + \bar{Y}(z)$$

Here

- $\bar{R} = \mathbb{E}R_t$, which is mean returns and
- $\bar{Y}(z) = \mathbb{E}_z Y(z, \eta_t)$, which is mean labor income in state z.

The dashed line is the 45 degree line.

We can see from the figure that the dynamics will be stable — assets do not diverge even in the highest state.

50.5 Exercises

Exercise 50.5.1

Let's repeat our *earlier exercise* on the long-run cross sectional distribution of assets.

In that exercise, we used a relatively simple income fluctuation model.

In the solution, we found the shape of the asset distribution to be unrealistic.

In particular, we failed to match the long right tail of the wealth distribution.

Your task is to try again, repeating the exercise, but now with our more sophisticated model.

Use the default parameters.

Solution to Exercise 50.5.1

First we write a function to compute a long asset series.

Because we want to JIT-compile the function, we code the solution in a way that breaks some rules on good programming style.

For example, we will pass in the solutions `a_star, σ_star` along with `ifp`, even though it would be more natural to just pass in `ifp` and then solve inside the function.

The reason we do this is that `solve_model_time_iter` is not JIT-compiled.

```
@njit
def compute_asset_series(ifp, a_star, σ_star, z_seq, T=500_000):
    """
    Simulates a time series of length T for assets, given optimal
    savings behavior.

        * ifp is an instance of IFP
        * a_star is the endogenous grid solution
        * σ_star is optimal consumption on the grid
        * z_seq is a time path for {Z_t}

    """

    # Create consumption function by linear interpolation
    σ = lambda a, z: interp(a_star[:, z], σ_star[:, z], a)

    # Simulate the asset path
    a = np.zeros(T+1)
    for t in range(T):
        z = z_seq[t]
        ζ, η = np.random.randn(), np.random.randn()
        R = ifp.R(z, ζ)
        Y = ifp.Y(z, η)
        a[t+1] = R * (a[t] - σ(a[t], z)) + Y
    return a
```

Now we call the function, generate the series and then histogram it, using the solutions computed above.

```
T = 1_000_000
mc = MarkovChain(ifp.P)
z_seq = mc.simulate(T, random_state=1234)

a = compute_asset_series(ifp, a_star, σ_star, z_seq, T=T)

fig, ax = plt.subplots()
ax.hist(a, bins=40, alpha=0.5, density=True)
ax.set(xlabel='assets')
plt.show()
```

Now we have managed to successfully replicate the long right tail of the wealth distribution.

Here's another view of this using a horizontal violin plot.

```
fig, ax = plt.subplots()
ax.violinplot(a, vert=False, showmedians=True)
ax.set(xlabel='assets')
plt.show()
```

Part VII

Bayes Law

CHAPTER

FIFTYONE

NON-CONJUGATE PRIORS

This lecture is a sequel to the *quantecon lecture*.

That lecture offers a Bayesian interpretation of probability in a setting in which the likelihood function and the prior distribution over parameters just happened to form a **conjugate** pair in which

- application of Bayes' Law produces a posterior distribution that has the same functional form as the prior

Having a likelihood and prior that are conjugate can simplify calculation of a posterior, faciltating analytical or nearly analytical calculations.

But in many situations the likelihood and prior need not form a conjugate pair.

- after all, a person's prior is his or her own business and would take a form conjugate to a likelihood only by remote coincidence

In these situations, computing a posterior can become very challenging.

In this lecture, we illustrate how modern Bayesians confront non-conjugate priors by using Monte Carlo techniques that involve

- first cleverly forming a Markov chain whose invariant distribution is the posterior distribution we want
- simulating the Markov chain until it has converged and then sampling from the invariant distribution to approximate the posterior

We shall illustrate the approach by deploying two powerful Python modules that implement this approach as well as another closely related one to be described below.

The two Python modules are

- `numpyro`
- `pymc4`

As usual, we begin by importing some Python code.

```
# install dependencies
!pip install numpyro pyro-ppl torch jax
```

```
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from scipy.stats import binom
import scipy.stats as st
import torch

# jax
```

(continues on next page)

(continued from previous page)

```
import jax.numpy as jnp
from jax import lax, random

# pyro
import pyro
from pyro import distributions as dist
import pyro.distributions.constraints as constraints
from pyro.infer import MCMC, NUTS, SVI, ELBO, Trace_ELBO
from pyro.optim import Adam

# numpyro
import numpyro
from numpyro import distributions as ndist
import numpyro.distributions.constraints as nconstraints
from numpyro.infer import MCMC as nMCMC
from numpyro.infer import NUTS as nNUTS
from numpyro.infer import SVI as nSVI
from numpyro.infer import ELBO as nELBO
from numpyro.infer import Trace_ELBO as nTrace_ELBO
from numpyro.optim import Adam as nAdam
```

51.1 Unleashing MCMC on a Binomial Likelihood

This lecture begins with the binomial example in the *quantecon lecture*.

That lecture computed a posterior

- analytically via choosing the conjugate priors,

This lecture instead computes posteriors

- numerically by sampling from the posterior distribution through MCMC methods, and
- using a variational inference (VI) approximation.

We use both the packages `pyro` and `numpyro` with assistance from `jax` to approximate a posterior distribution

We use several alternative prior distributions

We compare computed posteriors with ones associated with a conjugate prior as described in *the quantecon lecture*

51.1.1 Analytical Posterior

Assume that the random variable $X \sim Binom\left(n, \theta\right)$.

This defines a likelihood function

$$L\left(Y|\theta\right) = \text{Prob}(X = k|\theta) = \left(\frac{n!}{k!(n-k)!}\right)\theta^k(1-\theta)^{n-k}$$

where $Y = k$ is an observed data point.

We view θ as a random variable for which we assign a prior distribution having density $f(\theta)$.

We will try alternative priors later, but for now, suppose the prior is distributed as $\theta \sim Beta\left(\alpha, \beta\right)$, i.e.,

$$f(\theta) = \text{Prob}(\theta) = \frac{\theta^{\alpha-1}(1-\theta)^{\beta-1}}{B(\alpha, \beta)}$$

We choose this as our prior for now because we know that a conjugate prior for the binomial likelihood function is a beta distribution.

After observing k successes among N sample observations, the posterior probability distributionof θ is

$$\text{Prob}(\theta|k) = \frac{\text{Prob}(\theta, k)}{\text{Prob}(k)} = \frac{\text{Prob}(k|\theta)\text{Prob}(\theta)}{\text{Prob}(k)} = \frac{\text{Prob}(k|\theta)\text{Prob}(\theta)}{\int_0^1 \text{Prob}(k|\theta)\text{Prob}(\theta)d\theta}$$

$$= \frac{\binom{N}{k}(1-\theta)^{N-k}\theta^k \frac{\theta^{\alpha-1}(1-\theta)^{\beta-1}}{B(\alpha,\beta)}}{\int_0^1 \binom{N}{k}(1-\theta)^{N-k}\theta^k \frac{\theta^{\alpha-1}(1-\theta)^{\beta-1}}{B(\alpha,\beta)}d\theta}$$

$$= \frac{(1-\theta)^{\beta+N-k-1}\theta^{\alpha+k-1}}{\int_0^1 (1-\theta)^{\beta+N-k-1}\theta^{\alpha+k-1}d\theta}.$$

Thus,

$$\text{Prob}(\theta|k) \sim Beta(\alpha + k, \beta + N - k)$$

The analytical posterior for a given conjugate beta prior is coded in the following Python code.

```
def simulate_draw(theta, n):
    """
    Draws a Bernoulli sample of size n with probability P(Y=1) = theta
    """
    rand_draw = np.random.rand(n)
    draw = (rand_draw < theta).astype(int)
    return draw


def analytical_beta_posterior(data, alpha0, beta0):
    """
    Computes analytically the posterior distribution with beta prior parametrized by␣
 ↪(alpha, beta)
    given # num observations

    Parameters
    ---------
    num : int.
        the number of observations after which we calculate the posterior
    alpha0, beta0 : float.
        the parameters for the beta distribution as a prior

    Returns
    ---------
    The posterior beta distribution
    """
    num = len(data)
    up_num = data.sum()
    down_num = num - up_num
    return st.beta(alpha0 + up_num, beta0 + down_num)
```

51.1.2 Two Ways to Approximate Posteriors

Suppose that we don't have a conjugate prior.

Then we can't compute posteriors analytically.

Instead, we use computational tools to approximate the posterior distribution for a set of alternative prior distributions using both `Pyro` and `Numpyro` packages in Python.

We first use the **Markov Chain Monte Carlo** (MCMC) algorithm .

We implement the NUTS sampler to sample from the posterior.

In that way we construct a sampling distribution that approximates the posterior.

After doing that we deply another procedure called **Variational Inference** (VI).

In particular, we implement Stochastic Variational Inference (SVI) machinery in both `Pyro` and `Numpyro`.

The MCMC algorithm supposedly generates a more accurate approximation since in principle it directly samples from the posterior distribution.

But it can be computationally expensive, especially when dimension is large.

A VI approach can be cheaper, but it is likely to produce an inferior approximation to the posterior, for the simple reason that it requires guessing a parametric **guide functional form** that we use to approximate a posterior.

This guide function is likely at best to be an imperfect approximation.

By paying the cost of restricting the putative posterior to have a restricted functional form, the problem of approximating a posteriors is transformed to a well-posed optimization problem that seeks parameters of the putative posterior that minimize a Kullback-Leibler (KL) divergence between true posterior and the putatitive posterior distribution.

- minimizing the KL divergence is equivalent with maximizing a criterion called the **Evidence Lower Bound** (ELBO), as we shall verify soon.

51.2 Prior Distributions

In order to be able to apply MCMC sampling or VI, `Pyro` and `Numpyro` require that a prior distribution satisfy special properties:

- we must be able sample from it;
- we must be able to compute the log pdf pointwise;
- the pdf must be differentiable with respect to the parameters.

We'll want to define a distribution `class`.

We will use the following priors:

- a uniform distribution on $[\underline{\theta}, \overline{\theta}]$, where $0 \leq \underline{\theta} < \overline{\theta} \leq 1$.
- a truncated log-normal distribution with support on $[0, 1]$ with parameters (μ, σ).
  - To implement this, let $Z \sim Normal(\mu, \sigma)$ and $\tilde{Z}$ be truncated normal with support $[\log(0), \log(1)]$, then $\exp(Z)$ has a log normal distribution with bounded support $[0, 1]$. This can be easily coded since `Numpyro` has a built-in truncated normal distribution, and `Torch` provides a `TransformedDistribution` class that includes an exponential transformation.
  - Alternatively, we can use a rejection sampling strategy by assigning the probability rate to 0 outside the bounds and rescaling accepted samples, i.e., realizations that are within the bounds, by the total probability computed

via CDF of the original distribution. This can be implemented by defining a truncated distribution class with `pyro`'s `dist.Rejector` class.

  - We implement both methods in the below section and verify that they produce the same result.

- a shifted von Mises distribution that has support confined to $[0, 1]$ with parameter (μ, κ).
  - Let $X \sim vonMises(0, \kappa)$. We know that X has bounded support $[-\pi, \pi]$. We can define a shifted von Mises random variable $\tilde{X} = a + bX$ where $a = 0.5, b = 1/(2\pi)$ so that $\tilde{X}$ is supported on $[0, 1]$.
  - This can be implemented using `Torch`'s `TransformedDistribution` class with its `AffineTransform` method.
  - If instead, we want the prior to be von-Mises distributed with center $\mu = 0.5$, we can choose a high concentration level κ so that most mass is located between 0 and 1. Then we can truncate the distribution using the above strategy. This can be implemented using `pyro`'s `dist.Rejector` class. We choose $\kappa > 40$ in this case.
- a truncated Laplace distribution.
  - We also considered a truncated Laplace distribution because its density comes in a piece-wise non-smooth form and has a distinctive spiked shape.
  - The truncated Laplace can be created using `Numpyro`'s `TruncatedDistribution` class.

```
# used by Numpyro
def TruncatedLogNormal_trans(loc, scale):
    """
    Obtains the truncated log normal distribution using numpyro's TruncatedNormal and␣
↪ExpTransform
    """
    base_dist = ndist.TruncatedNormal(low=jnp.log(0), high=jnp.log(1), loc=loc,␣
↪scale=scale)
    return ndist.TransformedDistribution(
        base_dist,ndist.transforms.ExpTransform()
        )

def ShiftedVonMises(kappa):
    """
    Obtains the shifted von Mises distribution using AffineTransform
    """
    base_dist = ndist.VonMises(0, kappa)
    return ndist.TransformedDistribution(
        base_dist, ndist.transforms.AffineTransform(loc=0.5, scale=1/(2*jnp.pi))
        )

def TruncatedLaplace(loc, scale):
    """
    Obtains the truncated Laplace distribution on [0,1]
    """
    base_dist = ndist.Laplace(loc, scale)
    return ndist.TruncatedDistribution(
        base_dist, low=0.0, high=1.0
    )

# used by Pyro
class TruncatedLogNormal(dist.Rejector):
    """
    Define a TruncatedLogNormal distribution through rejection sampling in Pyro
    """
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, loc, scale_0, upp=1):
        self.upp = upp
        propose = dist.LogNormal(loc, scale_0)

        def log_prob_accept(x):
            return (x < upp).type_as(x).log()

        log_scale = dist.LogNormal(loc, scale_0).cdf(torch.as_tensor(upp)).log()
        super(TruncatedLogNormal, self).__init__(propose, log_prob_accept, log_scale)

    @constraints.dependent_property
    def support(self):
        return constraints.interval(0, self.upp)


class TruncatedvonMises(dist.Rejector):
    """
    Define a TruncatedvonMises distribution through rejection sampling in Pyro
    """
    def __init__(self, kappa, mu=0.5, low=0.0, upp=1.0):
        self.low, self.upp = low, upp
        propose = dist.VonMises(mu, kappa)

        def log_prob_accept(x):
            return ((x > low) & (x < upp)).type_as(x).log()

        log_scale = torch.log(
            torch.tensor(
                st.vonmises(kappa=kappa, loc=mu).cdf(upp)
                - st.vonmises(kappa=kappa, loc=mu).cdf(low))
        )
        super(TruncatedvonMises, self).__init__(propose, log_prob_accept, log_scale)

    @constraints.dependent_property
    def support(self):
        return constraints.interval(self.low, self.upp)
```

51.2.1 Variational Inference

Instead of directly sampling from the posterior, the **variational inference** methodw approximates an unknown posterior distribution with a family of tractable distributions/densities.

It then seeks to minimizes a measure of statistical discrepancy between the approximating and true posteriors.

Thus variational inference (VI) approximates a posterior by solving a minimization problem.

Let the latent parameter/variable that we want to infer be θ.

Let the prior be $p(\theta)$ and the likelihood be $p\left(Y|\theta\right)$.

We want $p\left(\theta|Y\right)$.

Bayes' rule implies

$$p\left(\theta|Y\right) = \frac{p\left(Y,\theta\right)}{p\left(Y\right)} = \frac{p\left(Y|\theta\right)p\left(\theta\right)}{p\left(Y\right)}$$

where

$$p\left(Y\right)=\int d\theta p\left(Y\mid\theta\right)p\left(Y\right). \tag{51.1}$$

The integral on the right side of (51.1) is typically difficult to compute.

Consider a **guide distribution** $q_\phi(\theta)$ parameterized by ϕ that we'll use to approximate the posterior.

We choose parameters ϕ of the guide distribution to minimize a Kullback-Leibler (KL) divergence between the approximate posterior $q_\phi(\theta)$ and the posterior:

$$D_{KL}(q(\theta;\phi)\parallel p(\theta\mid Y))\equiv-\int d\theta q(\theta;\phi)\log\frac{p(\theta\mid Y)}{q(\theta;\phi)}$$

Thus, we want a **variational distribution** q that solves

$$\min_{\phi}\quad D_{KL}(q(\theta;\phi)\parallel p(\theta\mid Y))$$

Note that

$$\begin{aligned}
D_{KL}(q(\theta;\phi)\parallel p(\theta\mid Y))&=-\int d\theta q(\theta;\phi)\log\frac{P(\theta\mid Y)}{q(\theta;\phi)}\\
&=-\int d\theta q(\theta)\log\frac{\frac{p(\theta,Y)}{p(Y)}}{q(\theta)}\\
&=-\int d\theta q(\theta)\log\frac{p(\theta,Y)}{p(\theta)q(Y)}\\
&=-\int d\theta q(\theta)\left[\log\frac{p(\theta,Y)}{q(\theta)}-\log p(Y)\right]\\
&=-\int d\theta q(\theta)\log\frac{p(\theta,Y)}{q(\theta)}+\int d\theta q(\theta)\log p(Y)\\
&=-\int d\theta q(\theta)\log\frac{p(\theta,Y)}{q(\theta)}+\log p(Y)\\
\log p(Y)&=D_{KL}(q(\theta;\phi)\parallel p(\theta\mid Y))+\int d\theta q_\phi(\theta)\log\frac{p(\theta,Y)}{q_\phi(\theta)}
\end{aligned}$$

For observed data Y, $p(\theta,Y)$ is a constant, so minimizing KL divergence is equivalent to maximizing

$$ELBO\equiv\int d\theta q_\phi(\theta)\log\frac{p(\theta,Y)}{q_\phi(\theta)}=\mathbb{E}_{q_\phi(\theta)}\left[\log p(\theta,Y)-\log q_\phi(\theta)\right] \tag{51.2}$$

Formula (51.2) is called the evidence lower bound (ELBO).

A standard optimization routine can used to search for the optimal ϕ in our parametrized distribution $q_\phi(\theta)$.

The parameterized distribution $q_\phi(\theta)$ is called the **variational distribution**.

We can implement Stochastic Variational Inference (SVI) in Pyro and Numpyro using the `Adam` gradient descent algorithm to approximate posterior.

We use two sets of variational distributions: Beta and TruncatedNormal with support $[0,1]$

- Learnable parameters for the Beta distribution are (alpha, beta), both of which are positive.
- Learnable parameters for the Truncated Normal distribution are (loc, scale).

We restrict the truncated Normal paramter 'loc' to be in the interval $[0,1]$.

51.3 Implementation

We have constructed a Python class `BaysianInference` that requires the following arguments to be initialized:

- `param`: a tuple/scalar of parameters dependent on distribution types
- `name_dist`: a string that specifies distribution names

The (`param`, `name_dist`) pair includes:

- ('beta', alpha, beta)
- ('uniform', upper_bound, lower_bound)
- ('lognormal', loc, scale)
  - Note: This is the truncated log normal.
- ('vonMises', kappa), where kappa denotes concentration parameter, and center location is set to 0.5.
  - Note: When using `Pyro`, this is the truncated version of the original vonMises distribution;
  - Note: When using `Numpyro`, this is the **shifted** distribution.
- ('laplace', loc, scale)
  - Note: This is the truncated Laplace

The class `BaysianInference` has several key methods :

- `sample_prior`:
  - This can be used to draw a single sample from the given prior distribution.
- `show_prior`:
  - Plots the approximate prior distribution by repeatedly drawing samples and fitting a kernal density curve.
- `MCMC_sampling`:
  - INPUT: (data, num_samples, num_warmup=1000)
  - Take a `np.array` data and generate MCMC sampling of posterior of size `num_samples`.
- `SVI_run`:
  - INPUT: (data, guide_dist, n_steps=10000)
  - guide_dist = 'normal' - use a **truncated** normal distribution as the parametrized guide
  - guide_dist = 'beta' - use a beta distribution as the parametrized guide
  - RETURN: (params, losses) - the learned parameters in a `dict` and the vector of loss at each step.

```
class BayesianInference:
    def __init__(self, param, name_dist, solver):
        """
        Parameters
        ---------
        param : tuple.
            a tuple object that contains all relevant parameters for the distribution
        dist : str.
            name of the distribution - 'beta', 'uniform', 'lognormal', 'vonMises',
↪'tent'
        solver : str.
            either pyro or numpyro
```

(continues on next page)

(continued from previous page)

```
        """
        self.param = param
        self.name_dist = name_dist
        self.solver = solver

        # jax requires explicit PRNG state to be passed
        self.rng_key = random.PRNGKey(0)


    def sample_prior(self):
        """
        Define the prior distribution to sample from in Pyro/Numpyro models.
        """
        if self.name_dist=='beta':
            # unpack parameters
            alpha0, beta0 = self.param
            if self.solver=='pyro':
                sample = pyro.sample('theta', dist.Beta(alpha0, beta0))
            else:
                sample = numpyro.sample('theta', ndist.Beta(alpha0, beta0), rng_
↪key=self.rng_key)

        elif self.name_dist=='uniform':
            # unpack parameters
            lb, ub = self.param
            if self.solver=='pyro':
                sample = pyro.sample('theta', dist.Uniform(lb, ub))
            else:
                sample = numpyro.sample('theta', ndist.Uniform(lb, ub), rng_key=self.
↪rng_key)

        elif self.name_dist=='lognormal':
            # unpack parameters
            loc, scale = self.param
            if self.solver=='pyro':
                sample = pyro.sample('theta', TruncatedLogNormal(loc, scale))
            else:
                sample = numpyro.sample('theta', TruncatedLogNormal_trans(loc, scale),
↪ rng_key=self.rng_key)

        elif self.name_dist=='vonMises':
            # unpack parameters
            kappa = self.param
            if self.solver=='pyro':
                sample = pyro.sample('theta', TruncatedvonMises(kappa))
            else:
                sample = numpyro.sample('theta', ShiftedVonMises(kappa), rng_key=self.
↪rng_key)

        elif self.name_dist=='laplace':
            # unpack parameters
            loc, scale = self.param
            if self.solver=='pyro':
                print("WARNING: Please use Numpyro for truncated Laplace.")
                sample = None
            else:
```

(continues on next page)

(continued from previous page)

```
                sample = numpyro.sample('theta', TruncatedLaplace(loc, scale), rng_
↪key=self.rng_key)

        return sample


    def show_prior(self, size=1e5, bins=20, disp_plot=1):
        """
        Visualizes prior distribution by sampling from prior and plots the␣
↪approximated sampling distribution
        """
        self.bins = bins

        if self.solver=='pyro':
            with pyro.plate('show_prior', size=size):
                sample = self.sample_prior()
            # to numpy
            sample_array = sample.numpy()

        elif self.solver=='numpyro':
            with numpyro.plate('show_prior', size=size):
                sample = self.sample_prior()
            # to numpy
            sample_array=jnp.asarray(sample)

        # plot histogram and kernel density
        if disp_plot==1:
            sns.displot(sample_array, kde=True, stat='density', bins=bins, height=5,␣
↪aspect=1.5)
            plt.xlim(0, 1)
            plt.show()
        else:
            return sample_array


    def model(self, data):
        """
        Define the probabilistic model by specifying prior, conditional likelihood,␣
↪and data conditioning
        """
        if not torch.is_tensor(data):
            data = torch.tensor(data)
        # set prior
        theta = self.sample_prior()

        # sample from conditional likelihood
        if self.solver=='pyro':
            output = pyro.sample('obs', dist.Binomial(len(data), theta), obs=torch.
↪sum(data))
        else:
            # Note: numpyro.sample() requires obs=np.ndarray
            output = numpyro.sample('obs', ndist.Binomial(len(data), theta),␣
↪obs=torch.sum(data).numpy())
        return output
```

(continues on next page)

(continued from previous page)

```
    def MCMC_sampling(self, data, num_samples, num_warmup=1000):
        """
        Computes numerically the posterior distribution with beta prior parametrized␣
↪by (alpha0, beta0)
        given data using MCMC
        """
        # tensorize
        data = torch.tensor(data)

        # use pyro
        if self.solver=='pyro':

            nuts_kernel = NUTS(self.model)
            mcmc = MCMC(nuts_kernel, num_samples=num_samples, warmup_steps=num_warmup,
↪ disable_progbar=True)
            mcmc.run(data)

        # use numpyro
        elif self.solver=='numpyro':

            nuts_kernel = nNUTS(self.model)
            mcmc = nMCMC(nuts_kernel, num_samples=num_samples, num_warmup=num_warmup,␣
↪progress_bar=False)
            mcmc.run(self.rng_key, data=data)

        # collect samples
        samples = mcmc.get_samples()['theta']
        return samples


    def beta_guide(self, data):
        """
        Defines the candidate parametrized variational distribution that we train to␣
↪approximate posterior with Pyro/Numpyro
        Here we use parameterized beta
        """
        if self.solver=='pyro':
            alpha_q = pyro.param('alpha_q', torch.tensor(0.5),
                                 constraint=constraints.positive)
            beta_q = pyro.param('beta_q', torch.tensor(0.5),
                                 constraint=constraints.positive)
            pyro.sample('theta', dist.Beta(alpha_q, beta_q))

        else:
            alpha_q = numpyro.param('alpha_q', 10,
                                 constraint=nconstraints.positive)
            beta_q = numpyro.param('beta_q', 10,
                                 constraint=nconstraints.positive)

            numpyro.sample('theta', ndist.Beta(alpha_q, beta_q))


    def truncnormal_guide(self, data):
        """
        Defines the candidate parametrized variational distribution that we train to␣
↪approximate posterior with Pyro/Numpyro
```

(continues on next page)

(continued from previous page)

```
        Here we use truncated normal on [0,1]
        """
        loc = numpyro.param('loc', 0.5,
                            constraint=nconstraints.interval(0.0, 1.0))
        scale = numpyro.param('scale', 1,
                            constraint=nconstraints.positive)
        numpyro.sample('theta', ndist.TruncatedNormal(loc, scale, low=0.0, high=1.0))


    def SVI_init(self, guide_dist, lr=0.0005):
        """
        Initiate SVI training mode with Adam optimizer
        NOTE: truncnormal_guide can only be used with numpyro solver
        """
        adam_params = {"lr": lr}

        if guide_dist=='beta':
            if self.solver=='pyro':
                optimizer = Adam(adam_params)
                svi = SVI(self.model, self.beta_guide, optimizer, loss=Trace_ELBO())

            elif self.solver=='numpyro':
                optimizer = nAdam(step_size=lr)
                svi = nSVI(self.model, self.beta_guide, optimizer, loss=nTrace_ELBO())

        elif guide_dist=='normal':
            # only allow numpyro
            if self.solver=='pyro':
                print("WARNING: Please use Numpyro with TruncatedNormal guide")
                svi = None

            elif self.solver=='numpyro':
                optimizer = nAdam(step_size=lr)
                svi = nSVI(self.model, self.truncnormal_guide, optimizer, loss=nTrace_
↪ELBO())
        else:
            print("WARNING: Please input either 'beta' or 'normal'")
            svi = None

        return svi

    def SVI_run(self, data, guide_dist, n_steps=10000):
        """
        Runs SVI and returns optimized parameters and losses

        Returns
        --------
        params : the learned parameters for guide
        losses : a vector of loss at each step
        """
        # tensorize data
        if not torch.is_tensor(data):
            data = torch.tensor(data)

        # initiate SVI
        svi = self.SVI_init(guide_dist=guide_dist)
```

(continues on next page)

(continued from previous page)

```
        # do gradient steps
        if self.solver=='pyro':
            # store loss vector
            losses = np.zeros(n_steps)
            for step in range(n_steps):
                losses[step] = svi.step(data)

            # pyro only supports beta VI distribution
            params = {
                'alpha_q': pyro.param('alpha_q').item(),
                'beta_q': pyro.param('beta_q').item()
                }

        elif self.solver=='numpyro':
            result = svi.run(self.rng_key, n_steps, data, progress_bar=False)
            params = dict(
                (key, np.asarray(value)) for key, value in result.params.items()
                )
            losses = np.asarray(result.losses)

        return params, losses
```

51.4 Alternative Prior Distributions

Let's see how well our sampling algorithm does in approximating

- a log normal distribution
- a uniform distribution

To examine our alternative prior distributions, we'll plot approximate prior distributions below by calling the `show_prior` method.

We verify that the rejection sampling strategy under `Pyro` produces the same log normal distribution as the truncated normal transformation under `Numpyro`.

```
# truncated log normal
exampleLN = BayesianInference(param=(0,2), name_dist='lognormal', solver='numpyro')
exampleLN.show_prior(size=100000,bins=20)

# truncated uniform
exampleUN = BayesianInference(param=(0.1,0.8), name_dist='uniform', solver='numpyro')
exampleUN.show_prior(size=100000,bins=20)
```

```
CUDA backend failed to initialize: Found CUDA version 12010, but JAX was built␣
↪against version 12020, which is newer. The copy of CUDA that is installed must␣
↪be at least as new as the version against which JAX was built. (Set TF_CPP_MIN_
↪LOG_LEVEL=0 and rerun for more info.)
```

```
/opt/conda/envs/quantecon/lib/python3.11/site-packages/seaborn/axisgrid.py:118:␣
↪UserWarning: The figure layout has changed to tight
  self._figure.tight_layout(*args, **kwargs)
```

```
/opt/conda/envs/quantecon/lib/python3.11/site-packages/seaborn/axisgrid.py:118:␣
↪UserWarning: The figure layout has changed to tight
  self._figure.tight_layout(*args, **kwargs)
```

The above graphs show that sampling seems to work well with both distributions.

Now let's see how well things work with a couple of von Mises distributions.

```
# shifted von Mises
exampleVM = BayesianInference(param=10, name_dist='vonMises', solver='numpyro')
exampleVM.show_prior(size=100000,bins=20)

# truncated von Mises
exampleVM_trunc = BayesianInference(param=20, name_dist='vonMises', solver='pyro')
exampleVM_trunc.show_prior(size=100000,bins=20)
```

```
/opt/conda/envs/quantecon/lib/python3.11/site-packages/seaborn/axisgrid.py:118:␣
↪UserWarning: The figure layout has changed to tight
  self._figure.tight_layout(*args, **kwargs)
```

```
/opt/conda/envs/quantecon/lib/python3.11/site-packages/seaborn/axisgrid.py:118:␣
↪UserWarning: The figure layout has changed to tight
  self._figure.tight_layout(*args, **kwargs)
```

These graphs look good too.

Now let's try with a Laplace distribution.

```
# truncated Laplace
exampleLP = BayesianInference(param=(0.5,0.05), name_dist='laplace', solver='numpyro')
exampleLP.show_prior(size=100000,bins=40)
```

```
/opt/conda/envs/quantecon/lib/python3.11/site-packages/seaborn/axisgrid.py:118:␣
↪UserWarning: The figure layout has changed to tight
  self._figure.tight_layout(*args, **kwargs)
```

Having assured ourselves that our sampler seems to do a good job, let's put it to work in using MCMC to compute posterior probabilities.

51.5 Posteriors Via MCMC and VI

We construct a class `BayesianInferencePlot` to implement MCMC or VI algorithms and plot multiple posteriors for different updating data sizes and different possible prior.

This class takes as inputs the true data generating parameter 'theta', a list of updating data sizes for multiple posterior plotting, and a defined and parametrized `BayesianInference` class.

It has two key methods:

- `BayesianInferencePlot.MCMC_plot()` takes wanted MCMC sample size as input and plot the output posteriors together with the prior defined in `BayesianInference` class.
- `BayesianInferencePlot.SVI_plot()` takes wanted VI distribution class ('beta' or 'normal') as input and plot the posteriors together with the prior.

```
class BayesianInferencePlot:
    """
    Easily implement the MCMC and VI inference for a given instance of␣
↪BayesianInference class and
    plot the prior together with multiple posteriors

    Parameters
    ----------
    theta : float.
        the true DGP parameter
```

(continues on next page)

(continued from previous page)

```
    N_list : list.
        a list of sample size
    BayesianInferenceClass : class.
        a class initiated using BayesianInference()

    """

    def __init__(self, theta, N_list, BayesianInferenceClass, binwidth=0.02):
        """
        Enter Parameters for data generation and plotting
        """
        self.theta = theta
        self.N_list = N_list
        self.BayesianInferenceClass = BayesianInferenceClass

        # plotting parameters
        self.binwidth = binwidth
        self.linewidth=0.05
        self.colorlist = sns.color_palette(n_colors=len(N_list))

        # data generation
        N_max = max(N_list)
        self.data = simulate_draw(theta, N_max)


    def MCMC_plot(self, num_samples, num_warmup=1000):
        """
        Parameters as in MCMC_sampling except that data is already defined
        """
        fig, ax = plt.subplots(figsize=(10, 6))

        # plot prior
        prior_sample = self.BayesianInferenceClass.show_prior(disp_plot=0)
        sns.histplot(
            data=prior_sample, kde=True, stat='density',
            binwidth=self.binwidth,
            color='#4C4E52',
            linewidth=self.linewidth,
            alpha=0.1,
            ax=ax,
            label='Prior Distribution'
            )

        # plot posteriors
        for id, n in enumerate(self.N_list):
            samples = self.BayesianInferenceClass.MCMC_sampling(
                self.data[:n], num_samples, num_warmup
            )
            sns.histplot(
                samples, kde=True, stat='density',
                binwidth=self.binwidth,
                linewidth=self.linewidth,
                alpha=0.2,
                color=self.colorlist[id-1],
                label=f'Posterior with $n={n}$'
                )
```

(continues on next page)

(continued from previous page)

```
        ax.legend()
        ax.set_title('MCMC Sampling density of Posterior Distributions', fontsize=15)
        plt.xlim(0, 1)
        plt.show()


    def SVI_fitting(self, guide_dist, params):
        """
        Fit the beta/truncnormal curve using parameters trained by SVI.
        I create plot using PDF given by scipy.stats distributions since torch.dist␣
↪do not have embedded PDF methods.
        """
        # create x axis
        xaxis = np.linspace(0,1,1000)
        if guide_dist=='beta':
            y = st.beta.pdf(xaxis, a=params['alpha_q'], b=params['beta_q'])

        elif guide_dist=='normal':

            # rescale upper/lower bound. See Scipy's truncnorm doc
            lower, upper = (0, 1)
            loc, scale = params['loc'], params['scale']
            a, b = (lower - loc) / scale, (upper - loc) / scale

            y = st.truncnorm.pdf(xaxis, a=a, b=b, loc=params['loc'], scale=params[
↪'scale'])
        return (xaxis, y)


    def SVI_plot(self, guide_dist, n_steps=2000):
        """
        Parameters as in SVI_run except that data is already defined
        """
        fig, ax = plt.subplots(figsize=(10, 6))

        # plot prior
        prior_sample = self.BayesianInferenceClass.show_prior(disp_plot=0)
        sns.histplot(
            data=prior_sample, kde=True, stat='density',
            binwidth=self.binwidth,
            color='#4C4E52',
            linewidth=self.linewidth,
            alpha=0.1,
            ax=ax,
            label='Prior Distribution'
            )

        # plot posteriors
        for id, n in enumerate(self.N_list):
            (params, losses) = self.BayesianInferenceClass.SVI_run(self.data[:n],␣
↪guide_dist, n_steps)
            x, y = self.SVI_fitting(guide_dist, params)
            ax.plot(x, y,
                alpha=1,
                color=self.colorlist[id-1],
                label=f'Posterior with $n={n}$'
```

(continues on next page)

(continued from previous page)

```
                )
        ax.legend()
        ax.set_title(f'SVI density of Posterior Distributions with {guide_dist} guide
↪', fontsize=15)
        plt.xlim(0, 1)
        plt.show()
```

Let's set some parameters that we'll use in all of the examples below.

To save computer time at first, notice that we'll set `MCMC_num_samples = 2000` and `SVI_num_steps = 5000`.

(Later, to increase accuracy of approximations, we'll want to increase these.)

```
num_list = [5,10,50,100,1000]
MCMC_num_samples = 2000
SVI_num_steps = 5000

# theta is the data generating process
true_theta = 0.8
```

51.5.1 Beta Prior and Posteriors:

Let's compare outcomes when we use a Beta prior.

For the same Beta prior, we shall

- compute posteriors analytically
- compute posteriors using MCMC via `Pyro` and `Numpyro`.
- compute posteriors using VI via `Pyro` and `Numpyro`.

Let's start with the analytical method that we described in this quantecon lecture https://python.quantecon.org/prob_meaning.html

```
# First examine Beta priors
BETA_pyro = BayesianInference(param=(5,5), name_dist='beta', solver='pyro')
BETA_numpyro = BayesianInference(param=(5,5), name_dist='beta', solver='numpyro')

BETA_pyro_plot = BayesianInferencePlot(true_theta, num_list, BETA_pyro)
BETA_numpyro_plot = BayesianInferencePlot(true_theta, num_list, BETA_numpyro)


# plot analytical Beta prior and posteriors
xaxis = np.linspace(0,1,1000)
y_prior = st.beta.pdf(xaxis, 5, 5)

fig, ax = plt.subplots(figsize=(10, 6))
# plot analytical beta prior
ax.plot(xaxis, y_prior, label='Analytical Beta Prior', color='#4C4E52')

data, colorlist, N_list = BETA_pyro_plot.data, BETA_pyro_plot.colorlist, BETA_pyro_
↪plot.N_list
# plot analytical beta posteriors
for id, n in enumerate(N_list):
    func = analytical_beta_posterior(data[:n], alpha0=5, beta0=5)
```

(continues on next page)

(continued from previous page)

```
    y_posterior = func.pdf(xaxis)
    ax.plot(
        xaxis, y_posterior, color=colorlist[id-1], label=f'Analytical Beta Posterior␣
↪with $n={n}$')
ax.legend()
ax.set_title('Analytical Beta Prior and Posterior', fontsize=15)
plt.xlim(0, 1)
plt.show()
```

Now let's use MCMC while still using a beta prior.

We'll do this for both MCMC and VI.

```
BayesianInferencePlot(true_theta, num_list, BETA_pyro).MCMC_plot(num_samples=MCMC_num_
↪samples)
BayesianInferencePlot(true_theta, num_list, BETA_numpyro).SVI_plot(guide_dist='beta',␣
↪n_steps=SVI_num_steps)
```

Here the MCMC approximation looks good.

But the VI approximation doesn't look so good.

- even though we use the beta distribution as our guide, the VI approximated posterior distributions do not closely resemble the posteriors that we had just computed analytically.

(Here, our initial parameter for Beta guide is (0.5, 0.5).)

But if we increase the number of steps from 5000 to 10000 in VI as we now shall do, we'll get VI-approximated posteriors will be more accurate, as we shall see next.

(Increasing the step size increases computational time though).

```
BayesianInferencePlot(true_theta, num_list, BETA_numpyro).SVI_plot(guide_dist='beta',␣
↪n_steps=100000)
```

51.6 Non-conjugate Prior Distributions

Having assured ourselves that our MCMC and VI methods can work well when we have conjugate prior and so can also compute analytically, we next proceed to situations in which our prior is not a beta distribution, so we don't have a conjugate prior.

So we will have non-conjugate priors and are cast into situations in which we can't calculate posteriors analytically.

51.6.1 MCMC

First, we implement and display MCMC.

We first initialize the `BayesianInference` classes and then can directly call `BayesianInferencePlot` to plot both MCMC and SVI approximating posteriors.

```
# Initialize BayesianInference classes
# try uniform
STD_UNIFORM_pyro = BayesianInference(param=(0,1), name_dist='uniform', solver='pyro')
UNIFORM_numpyro = BayesianInference(param=(0.2,0.7), name_dist='uniform', solver=
↪'numpyro')
```

(continues on next page)

(continued from previous page)

```
# try truncated lognormal
LOGNORMAL_numpyro = BayesianInference(param=(0,2), name_dist='lognormal', solver=
↪'numpyro')
LOGNORMAL_pyro = BayesianInference(param=(0,2), name_dist='lognormal', solver='pyro')

# try von Mises
# shifted von Mises
VONMISES_numpyro = BayesianInference(param=10, name_dist='vonMises', solver='numpyro')
# truncated von Mises
VONMISES_pyro = BayesianInference(param=40, name_dist='vonMises', solver='pyro')

# try laplace
LAPLACE_numpyro = BayesianInference(param=(0.5, 0.07), name_dist='laplace', solver=
↪'numpyro')
```

```
# Uniform
example_CLASS = STD_UNIFORM_pyro
print(f'=======INFO=======\nParameters: {example_CLASS.param}\nPrior Dist: {example_
↪CLASS.name_dist}\nSolver: {example_CLASS.solver}')
BayesianInferencePlot(true_theta, num_list, example_CLASS).MCMC_plot(num_samples=MCMC_
↪num_samples)

example_CLASS = UNIFORM_numpyro
print(f'=======INFO=======\nParameters: {example_CLASS.param}\nPrior Dist: {example_
↪CLASS.name_dist}\nSolver: {example_CLASS.solver}')
BayesianInferencePlot(true_theta, num_list, example_CLASS).MCMC_plot(num_samples=MCMC_
↪num_samples)
```

```
=======INFO=======
Parameters: (0, 1)
Prior Dist: uniform
Solver: pyro
```

```
=======INFO=======
Parameters: (0.2, 0.7)
Prior Dist: uniform
Solver: numpyro
```

In the situation depicted above, we have assumed a $Uniform(\underline{\theta}, \overline{\theta})$ prior that puts zero probability outside a bounded

support that excludes the true value.

Consequently, the posterior cannot put positive probability above $\overline{\theta}$ or below $\underline{\theta}$.

Note how when the true data-generating θ is located at 0.8 as it is here, when n gets large, the posterior concentrate on the upper bound of the support of the prior, 0.7 here.

```
# Log Normal
example_CLASS = LOGNORMAL_numpyro
print(f'=======INFO=======\nParameters: {example_CLASS.param}\nPrior Dist: {example_
↪CLASS.name_dist}\nSolver: {example_CLASS.solver}')
BayesianInferencePlot(true_theta, num_list, example_CLASS).MCMC_plot(num_samples=MCMC_
↪num_samples)

example_CLASS = LOGNORMAL_pyro
print(f'=======INFO=======\nParameters: {example_CLASS.param}\nPrior Dist: {example_
↪CLASS.name_dist}\nSolver: {example_CLASS.solver}')
BayesianInferencePlot(true_theta, num_list, example_CLASS).MCMC_plot(num_samples=MCMC_
↪num_samples)
```

```
=======INFO=======
Parameters: (0, 2)
Prior Dist: lognormal
Solver: numpyro
```

```
=======INFO=======
Parameters: (0, 2)
Prior Dist: lognormal
Solver: pyro
```

```
# Von Mises
example_CLASS = VONMISES_numpyro
print(f'=======INFO=======\nParameters: {example_CLASS.param}\nPrior Dist: {example_
↪CLASS.name_dist}\nSolver: {example_CLASS.solver}')
print('\nNOTE: Shifted von Mises')
BayesianInferencePlot(true_theta, num_list, example_CLASS).MCMC_plot(num_samples=MCMC_
↪num_samples)

example_CLASS = VONMISES_pyro
print(f'=======INFO=======\nParameters: {example_CLASS.param}\nPrior Dist: {example_
↪CLASS.name_dist}\nSolver: {example_CLASS.solver}')
print('\nNOTE: Truncated von Mises')
BayesianInferencePlot(true_theta, num_list, example_CLASS).MCMC_plot(num_samples=MCMC_
↪num_samples)
```

```
=======INFO=======
Parameters: 10
Prior Dist: vonMises
Solver: numpyro

NOTE: Shifted von Mises
```

```
=======INFO=======
Parameters: 40
Prior Dist: vonMises
Solver: pyro

NOTE: Truncated von Mises
```

```
# Laplace
example_CLASS = LAPLACE_numpyro
print(f'=======INFO=======\nParameters: {example_CLASS.param}\nPrior Dist: {example_
↪CLASS.name_dist}\nSolver: {example_CLASS.solver}')
BayesianInferencePlot(true_theta, num_list, example_CLASS).MCMC_plot(num_samples=MCMC_
↪num_samples)
```

```
=======INFO=======
Parameters: (0.5, 0.07)
Prior Dist: laplace
Solver: numpyro
```

To get more accuracy we will now increase the number of steps for Variational Inference (VI)

```
SVI_num_steps = 50000
```

VI with a Truncated Normal Guide

```
# Uniform
example_CLASS = BayesianInference(param=(0,1), name_dist='uniform', solver='numpyro')
print(f'=======INFO=======\nParameters: {example_CLASS.param}\nPrior Dist: {example_
↪CLASS.name_dist}\nSolver: {example_CLASS.solver}')
BayesianInferencePlot(true_theta, num_list, example_CLASS).SVI_plot(guide_dist='normal
↪', n_steps=SVI_num_steps)
```

```
=======INFO=======
Parameters: (0, 1)
Prior Dist: uniform
Solver: numpyro
```

```
# Log Normal
example_CLASS = LOGNORMAL_numpyro
print(f'=======INFO=======\nParameters: {example_CLASS.param}\nPrior Dist: {example_
↪CLASS.name_dist}\nSolver: {example_CLASS.solver}')
BayesianInferencePlot(true_theta, num_list, example_CLASS).SVI_plot(guide_dist='normal
↪', n_steps=SVI_num_steps)
```

```
=======INFO=======
Parameters: (0, 2)
Prior Dist: lognormal
Solver: numpyro
```

```
# Von Mises
example_CLASS = VONMISES_numpyro
print(f'=======INFO=======\nParameters: {example_CLASS.param}\nPrior Dist: {example_
↪CLASS.name_dist}\nSolver: {example_CLASS.solver}')
print('\nNB: Shifted von Mises')
BayesianInferencePlot(true_theta, num_list, example_CLASS).SVI_plot(guide_dist='normal
↪', n_steps=SVI_num_steps)
```

```
=======INFO=======
Parameters: 10
Prior Dist: vonMises
Solver: numpyro

NB: Shifted von Mises
```

```
# Laplace
example_CLASS = LAPLACE_numpyro
print(f'=======INFO=======\nParameters: {example_CLASS.param}\nPrior Dist: {example_
↪CLASS.name_dist}\nSolver: {example_CLASS.solver}')
BayesianInferencePlot(true_theta, num_list, example_CLASS).SVI_plot(guide_dist='normal
↪', n_steps=SVI_num_steps)
```

```
=======INFO=======
Parameters: (0.5, 0.07)
Prior Dist: laplace
Solver: numpyro
```

Variational Inference with a Beta Guide Distribution

```
# Uniform
example_CLASS = STD_UNIFORM_pyro
print(f'=======INFO=======\nParameters: {example_CLASS.param}\nPrior Dist: {example_
↪CLASS.name_dist}\nSolver: {example_CLASS.solver}')
BayesianInferencePlot(true_theta, num_list, example_CLASS).SVI_plot(guide_dist='beta',
↪ n_steps=SVI_num_steps)
```

```
=======INFO=======
Parameters: (0, 1)
Prior Dist: uniform
Solver: pyro
```

```
# Log Normal
example_CLASS = LOGNORMAL_numpyro
print(f'=======INFO=======\nParameters: {example_CLASS.param}\nPrior Dist: {example_
↪CLASS.name_dist}\nSolver: {example_CLASS.solver}')
BayesianInferencePlot(true_theta, num_list, example_CLASS).SVI_plot(guide_dist='beta',
↪ n_steps=SVI_num_steps)

example_CLASS = LOGNORMAL_pyro
print(f'=======INFO=======\nParameters: {example_CLASS.param}\nPrior Dist: {example_
↪CLASS.name_dist}\nSolver: {example_CLASS.solver}')
BayesianInferencePlot(true_theta, num_list, example_CLASS).SVI_plot(guide_dist='beta',
↪ n_steps=SVI_num_steps)
```

```
=======INFO=======
Parameters: (0, 2)
Prior Dist: lognormal
Solver: numpyro
```

```
=======INFO=======
Parameters: (0, 2)
Prior Dist: lognormal
Solver: pyro
```

```
# Von Mises
example_CLASS = VONMISES_numpyro
print(f'=======INFO=======\nParameters: {example_CLASS.param}\nPrior Dist: {example_
↪CLASS.name_dist}\nSolver: {example_CLASS.solver}')
print('\nNB: Shifted von Mises')
BayesianInferencePlot(true_theta, num_list, example_CLASS).SVI_plot(guide_dist='beta',
↪ n_steps=SVI_num_steps)

example_CLASS = VONMISES_pyro
print(f'=======INFO=======\nParameters: {example_CLASS.param}\nPrior Dist: {example_
↪CLASS.name_dist}\nSolver: {example_CLASS.solver}')
print('\nNB: Truncated von Mises')
BayesianInferencePlot(true_theta, num_list, example_CLASS).SVI_plot(guide_dist='beta',
↪ n_steps=SVI_num_steps)
```

```
=======INFO=======
Parameters: 10
Prior Dist: vonMises
Solver: numpyro

NB: Shifted von Mises
```

```
=======INFO=======
Parameters: 40
Prior Dist: vonMises
Solver: pyro

NB: Truncated von Mises
```

```
# Laplace
example_CLASS = LAPLACE_numpyro
print(f'=======INFO=======\nParameters: {example_CLASS.param}\nPrior Dist: {example_
↪CLASS.name_dist}\nSolver: {example_CLASS.solver}')
BayesianInferencePlot(true_theta, num_list, example_CLASS).SVI_plot(guide_dist='beta',
↪ n_steps=SVI_num_steps)
```

```
=======INFO=======
Parameters: (0.5, 0.07)
Prior Dist: laplace
Solver: numpyro
```

SVI density of Posterior Distributions with beta guide
Posterior with $n = 5$
Posterior with $n = 10$
Posterior with $n = 50$
Posterior with $n = 100$
Posterior with $n = 1000$
Prior Distribution
Density
0
2
4
6
8
10
12
0.0
0.2
0.4
0.6
0.8
1.0

CHAPTER

FIFTYTWO

POSTERIOR DISTRIBUTIONS FOR AR(1) PARAMETERS

We'll begin with some Python imports.

```
!pip install arviz pymc numpyro jax
```

```
import arviz as az
import pymc as pmc
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)
```

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

The first component of the statistical model is

$$y_{t+1} = \rho y_t + \sigma_x \epsilon_{t+1}, \quad t \geq 0 \tag{52.1}$$

where the scalars ρ and σ_x satisfy $|\rho| < 1$ and $\sigma_x > 0$; $\{\epsilon_{t+1}\}$ is a sequence of i.i.d. normal random variables with mean 0 and variance 1.

The second component of the statistical model is

$$y_0 \sim N(\mu_0, \sigma_0^2) \tag{52.2}$$

Consider a sample $\{y_t\}_{t=0}^T$ governed by this statistical model.

The model implies that the likelihood function of $\{y_t\}_{t=0}^T$ can be **factored**:

$$f(y_T, y_{T-1}, \dots, y_0) = f(y_T|y_{T-1}) f(y_{T-1}|y_{T-2}) \cdots f(y_1|y_0) f(y_0)$$

where we use f to denote a generic probability density.

The statistical model (52.1)-(52.2) implies

$$f(y_t|y_{t-1}) \sim \mathcal{N}(\rho y_{t-1}, \sigma_x^2)$$
$$f(y_0) \sim \mathcal{N}(\mu_0, \sigma_0^2)$$

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.

Below, we study two widely used alternative assumptions:

- $(\mu_0, \sigma_0) = (y_0, 0)$ which means that y_0 is drawn from the distribution $\mathcal{N}(y_0, 0)$; in effect, we are **conditioning on an observed initial value**.
- μ_0, σ_0 are functions of ρ, σ_x because y_0 is drawn from the stationary distribution implied by ρ, σ_x.

Note: We do **not** treat a third possible case in which μ_0, σ_0 are free parameters to be estimated.

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 $\{y_t\}_{t=0}^T$.

The notebook uses `pymc4` and `numpyro` to compute a posterior distribution of ρ, σ_x. We will use NUTS samplers to generate samples from the posterior in a chain. Both of these libraries support NUTS samplers.

NUTS is a form of Monte Carlo Markov Chain (MCMC) algorithm that bypasses random walk behaviour and allows for convergence to a target distribution more quickly. This not only has the advantage of speed, but allows for complex models to be fitted without having to employ specialised knowledge regarding the theory underlying those fitting methods.

Thus, we explore consequences of making these alternative assumptions about the distribution of y_0:

- A first procedure is to condition on whatever value of y_0 is observed. This amounts to assuming that the probability distribution of the random variable y_0 is a Dirac delta function that puts probability one on the observed value of y_0.
- A second procedure assumes that y_0 is drawn from the stationary distribution of a process described by (52.1) so that $y_0 \sim N\left(0, \frac{\sigma_x^2}{(1-\rho)^2}\right)$

When the initial value y_0 is far out in a tail of the stationary distribution, conditioning on an initial value gives a posterior that is **more accurate** in a sense that we'll explain.

Basically, when y_0 happens to be in a tail of the stationary distribution and we **don't condition on** y_0, the likelihood function for $\{y_t\}_{t=0}^T$ adjusts the posterior distribution of the parameter pair ρ, σ_x to make the observed value of y_0 more likely than it really is under the stationary distribution, thereby adversely twisting the posterior in short samples.

An example below shows how not conditioning on y_0 adversely shifts the posterior probability distribution of ρ toward larger values.

We begin by solving a **direct problem** that simulates an AR(1) process.

How we select the initial value y_0 matters.

- If we think y_0 is drawn from the stationary distribution $\mathcal{N}(0, \frac{\sigma_x^2}{1-\rho^2})$, then it is a good idea to use this distribution as $f(y_0)$. Why? Because y_0 contains information about ρ, σ_x.
- If we suspect that y_0 is far in the tails of the stationary distribution – so that variation in early observations in the sample have a significant **transient component** – it is better to condition on y_0 by setting $f(y_0) = 1$.

To illustrate the issue, we'll begin by choosing an initial y_0 that is far out in a tail of the stationary distribution.

```
def ar1_simulate(rho, sigma, y0, T):

    # Allocate space and draw epsilons
    y = np.empty(T)
    eps = np.random.normal(0.,sigma,T)

    # Initial condition and step forward
    y[0] = y0
    for t in range(1, T):
        y[t] = rho*y[t-1] + eps[t]

    return y

sigma =  1.
rho = 0.5
T = 50

np.random.seed(145353452)
y = ar1_simulate(rho, sigma, 10, T)
```

```
plt.plot(y)
plt.tight_layout()
```

Now we shall use Bayes' law to construct a posterior distribution, conditioning on the initial value of y_0.

(Later we'll assume that y_0 is drawn from the stationary distribution, but not now.)

First we'll use **pymc4**.

52.1 PyMC Implementation

For a normal distribution in pymc, $var = 1/\tau = \sigma^2$.

```
AR1_model = pmc.Model()

with AR1_model:

    # Start with priors
    rho = pmc.Uniform('rho', lower=-1., upper=1.) # Assume stable rho
    sigma = pmc.HalfNormal('sigma', sigma = np.sqrt(10))

    # Expected value of y at the next period (rho * y)
    yhat = rho * y[:-1]

    # Likelihood of the actual realization
    y_like = pmc.Normal('y_obs', mu=yhat, sigma=sigma, observed=y[1:])
```

pmc.sample by default uses the NUTS samplers to generate samples as shown in the below cell:

```
with AR1_model:
    trace = pmc.sample(50000, tune=10000, return_inferencedata=True)
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

```
with AR1_model:
    az.plot_trace(trace, figsize=(17,6))
```

Evidently, the posteriors aren't centered on the true values of $.5, 1$ that we used to generate the data.

This is a symptom of the classic **Hurwicz bias** for first order autoregressive processes (see Leonid Hurwicz [Hur50].)

The Hurwicz bias is worse the smaller is the sample (see [OW69]).

Be that as it may, here is more information about the posterior.

```
with AR1_model:
    summary = az.summary(trace, round_to=4)

summary
```

```
         mean      sd  hdi_3%  hdi_97%  mcse_mean  mcse_sd     ess_bulk  \
rho    0.5366  0.0711  0.4023   0.6701     0.0002   0.0001  164427.6658
sigma  1.0104  0.1070  0.8175   1.2106     0.0003   0.0002  166676.2733

          ess_tail  r_hat
rho    121123.6126    1.0
sigma  136679.2441    1.0
```

Now we shall compute a posterior distribution after seeing the same data but instead assuming that y_0 is drawn from the stationary distribution.

This means that

$$y_0 \sim N\left(0, \frac{\sigma_x^2}{1-\rho^2}\right)$$

We alter the code as follows:

```
AR1_model_y0 = pmc.Model()

with AR1_model_y0:

    # Start with priors
    rho = pmc.Uniform('rho', lower=-1., upper=1.) # Assume stable rho
    sigma = pmc.HalfNormal('sigma', sigma=np.sqrt(10))

    # Standard deviation of ergodic y
    y_sd = sigma / np.sqrt(1 - rho**2)

    # yhat
    yhat = rho * y[:-1]
    y_data = pmc.Normal('y_obs', mu=yhat, sigma=sigma, observed=y[1:])
    y0_data = pmc.Normal('y0_obs', mu=0., sigma=y_sd, observed=y[0])
```

```
with AR1_model_y0:
    trace_y0 = pmc.sample(50000, tune=10000, return_inferencedata=True)

# Grey vertical lines are the cases of divergence
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

```
with AR1_model_y0:
    az.plot_trace(trace_y0, figsize=(17,6))
```

```
with AR1_model:
    summary_y0 = az.summary(trace_y0, round_to=4)

summary_y0
```

```
         mean      sd  hdi_3%  hdi_97%  mcse_mean  mcse_sd     ess_bulk  \
rho    0.8757  0.0815  0.7313   0.9989     0.0002   0.0002  111428.1296
sigma  1.4048  0.1471  1.1355   1.6784     0.0004   0.0003  121931.4394

          ess_tail   r_hat
rho     83801.1839  1.0001
sigma  104282.7663  1.0001
```

Please note how the posterior for ρ has shifted to the right relative to when we conditioned on y_0 instead of assuming that y_0 is drawn from the stationary distribution.

Think about why this happens.

Hint: It is connected to how Bayes Law (conditional probability) solves an **inverse problem** by putting high probability on parameter values that make observations more likely.

We'll return to this issue after we use `numpyro` to compute posteriors under our two alternative assumptions about the distribution of y_0.

We'll now repeat the calculations using `numpyro`.

52.2 Numpyro Implementation

```
def plot_posterior(sample):
    """
    Plot trace and histogram
    """
    # To np array
    rhos = sample['rho']
    sigmas = sample['sigma']
    rhos, sigmas, = np.array(rhos), np.array(sigmas)
```

(continues on next page)

(continued from previous page)

```
    fig, axs = plt.subplots(2, 2, figsize=(17, 6))
    # Plot trace
    axs[0, 0].plot(rhos)    # rho
    axs[1, 0].plot(sigmas) # sigma

    # Plot posterior
    axs[0, 1].hist(rhos, bins=50, density=True, alpha=0.7)
    axs[0, 1].set_xlim([0, 1])
    axs[1, 1].hist(sigmas, bins=50, density=True, alpha=0.7)

    axs[0, 0].set_title("rho")
    axs[0, 1].set_title("rho")
    axs[1, 0].set_title("sigma")
    axs[1, 1].set_title("sigma")
    plt.show()
```

```
def AR1_model(data):
    # set prior
    rho = numpyro.sample('rho', dist.Uniform(low=-1., high=1.))
    sigma = numpyro.sample('sigma', dist.HalfNormal(scale=np.sqrt(10)))

    # Expected value of y at the next period (rho * y)
    yhat = rho * data[:-1]

    # Likelihood of the actual realization.
    y_data = numpyro.sample('y_obs', dist.Normal(loc=yhat, scale=sigma), obs=data[1:])
```

```
# Make jnp array
y = jnp.array(y)

# Set NUTS kernal
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)
```

```
plot_posterior(mcmc.get_samples())
```

```
mcmc.print_summary()
```

```
                mean       std    median      5.0%     95.0%     n_eff     r_hat
         rho    0.54      0.07      0.54      0.42      0.66  34735.88      1.00
       sigma    1.01      0.11      1.00      0.83      1.18  33245.42      1.00

Number of divergences: 0
```

Next, we again compute the posterior under the assumption that y_0 is drawn from the stationary distribution, so that

$$y_0 \sim N\left(0, \frac{\sigma_x^2}{1-\rho^2}\right)$$

Here's the new code to achieve this.

```
def AR1_model_y0(data):
    # Set prior
    rho = numpyro.sample('rho', dist.Uniform(low=-1., high=1.))
    sigma = numpyro.sample('sigma', dist.HalfNormal(scale=np.sqrt(10)))

    # Standard deviation of ergodic y
    y_sd = sigma / jnp.sqrt(1 - rho**2)

    # Expected value of y at the next period (rho * y)
    yhat = rho * data[:-1]

    # Likelihood of the actual realization.
    y_data = numpyro.sample('y_obs', dist.Normal(loc=yhat, scale=sigma), obs=data[1:])
    y0_data = numpyro.sample('y0_obs', dist.Normal(loc=0., scale=y_sd), obs=data[0])
```

```
# Make jnp array
y = jnp.array(y)

# Set NUTS kernal
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)
```

```
plot_posterior(mcmc2.get_samples())
```

```
mcmc2.print_summary()
```

```
                mean       std    median      5.0%     95.0%     n_eff     r_hat
       rho      0.88      0.08      0.89      0.76      1.00  29063.44      1.00
     sigma      1.40      0.15      1.39      1.17      1.64  25674.09      1.00

Number of divergences: 0
```

Look what happened to the posterior!

It has moved far from the true values of the parameters used to generate the data because of how Bayes' Law (i.e., conditional probability) is telling `numpyro` to explain what it interprets as "explosive" observations early in the sample.

Bayes' Law is able to generate a plausible likelihood for the first observation by driving $\rho \to 1$ and $\sigma \uparrow$ in order to raise the variance of the stationary distribution.

Our example illustrates the importance of what you assume about the distribution of initial conditions.

CHAPTER

FIFTYTHREE

FORECASTING AN AR(1) PROCESS

```
!pip install arviz pymc
```

This lecture describes methods for forecasting statistics that are functions of future values of a univariate autogressive process.

The methods are designed to take into account two possible sources of uncertainty about these statistics:

- random shocks that impinge of 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 [Wec79].

To acknowledge uncertainty about parameters, we'll deploy `pymc` to construct a Bayesian joint posterior distribution for unknown parameters.

Let's start with some imports.

```
import numpy as np
import arviz as az
import pymc as pmc
import matplotlib.pyplot as plt
import seaborn as sns

sns.set_style('white')
colors = sns.color_palette()

import logging
logging.basicConfig()
logger = logging.getLogger('pymc')
logger.setLevel(logging.CRITICAL)
```

53.1 A Univariate First-Order Autoregressive Process

Consider the univariate AR(1) model:

$$y_{t+1} = \rho y_t + \sigma \epsilon_{t+1}, \quad t \geq 0 \tag{53.1}$$

where the scalars ρ and σ satisfy $|\rho| < 1$ and $\sigma > 0$; $\{\epsilon_{t+1}\}$ is a sequence of i.i.d. normal random variables with mean 0 and variance 1.

The initial condition y_0 is a known number.

Equation (53.1) implies that for $t \geq 0$, the conditional density of y_{t+1} is

$$f(y_{t+1}|y_t; \rho, \sigma) \sim \mathcal{N}(\rho y_t, \sigma^2) \tag{53.2}$$

Further, equation (53.1) also implies that for $t \geq 0$, the conditional density of y_{t+j} for $j \geq 1$ is

$$f(y_{t+j}|y_t; \rho, \sigma) \sim \mathcal{N}\left(\rho^j y_t, \sigma^2 \frac{1-\rho^{2j}}{1-\rho^2}\right) \tag{53.3}$$

The predictive distribution (53.3) that 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 (53.3) with respect to a joint posterior distribution $\pi_t(\rho, \sigma|y^t)$ that conditions on an observed history $y^t = \{y_s\}_{s=0}^t$:

$$f(y_{t+j}|y^t) = \int f(y_{t+j}|y_t; \rho, \sigma)\pi_t(\rho, \sigma|y^t) d\rho d\sigma \tag{53.4}$$

Predictive distribution (53.3) assumes that parameters (ρ, σ) are known.

Predictive distribution (53.4) assumes that parameters (ρ, σ) are uncertain, but have known probability distribution $\pi_t(\rho, \sigma|y^t)$.

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 [Wec79] 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 three "sample path statistics";
- finally, plot the desired statistics from the N samples as an empirical distribution.

53.2 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 .9 and .95 coverage intervals using conditional distribution (53.3) 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.

```
def AR1_simulate(rho, sigma, y0, T):

    # Allocate space and draw epsilons
    y = np.empty(T)
    eps = np.random.normal(0, sigma, T)

    # Initial condition and step forward
    y[0] = y0
    for t in range(1, T):
        y[t] = rho * y[t-1] + eps[t]

    return y


def plot_initial_path(initial_path):
    """
    Plot the initial path and the preceding predictive densities
    """
    # Compute .9 confidence interval]
    y0 = initial_path[-1]
    center = np.array([rho**j * y0 for j in range(T1)])
    vars = np.array([sigma**2 * (1 - rho**(2 * j)) / (1 - rho**2) for j in range(T1)])
    y_bounds1_c95, y_bounds2_c95 = center + 1.96 * np.sqrt(vars), center - 1.96 * np.
↪sqrt(vars)
    y_bounds1_c90, y_bounds2_c90 = center + 1.65 * np.sqrt(vars), center - 1.65 * np.
↪sqrt(vars)

    # Plot
    fig, ax = plt.subplots(1, 1, figsize=(12, 6))
    ax.set_title("Initial Path and Predictive Densities", fontsize=15)
    ax.plot(np.arange(-T0 + 1, 1), initial_path)
    ax.set_xlim([-T0, T1])
    ax.axvline(0, linestyle='--', alpha=.4, color='k', lw=1)

    # Simulate future paths
    for i in range(10):
        y_future = AR1_simulate(rho, sigma, y0, T1)
        ax.plot(np.arange(T1), y_future, color='grey', alpha=.5)

    # Plot 90% CI
    ax.fill_between(np.arange(T1), y_bounds1_c95, y_bounds2_c95, alpha=.3, label='95%␣
↪CI')
    ax.fill_between(np.arange(T1), y_bounds1_c90, y_bounds2_c90, alpha=.35, label='90
↪% CI')
    ax.plot(np.arange(T1), center, color='red', alpha=.7, label='expected mean')
    ax.legend(fontsize=12)
    plt.show()
```

(continues on next page)

(continued from previous page)

```
sigma =  1
rho = 0.9
T0, T1 = 100, 100
y0 = 10

# Simulate
np.random.seed(145)
initial_path = AR1_simulate(rho, sigma, y0, T0)

# Plot
plot_initial_path(initial_path)
```

As functions of forecast horizon, the coverage intervals have shapes like those described in https://python.quantecon.org/perm_income_cons.html

53.3 Predictive Distributions of Path Properties

Wecker [Wec79] proposed using simulation techniques to characterize 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.

To examine this statistic, let Z be an indicator process

$$Z_t(Y(\omega)) := \begin{cases} 1 & \text{if } Y_t(\omega) < Y_{t-1}(\omega) < Y_{t-2}(\omega) \geq Y_{t-3}(\omega) \\ 0 & \text{otherwise} \end{cases}$$

Then the random variable **time until the next turning point** is defined as the following **stopping time** with respect to Z:

$$W_t(\omega) := \inf\{k \geq 1 \mid Z_{t+k}(\omega) = 1\}$$

Wecker [Wec79] 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

$$T_t(Y(\omega)) := \begin{cases} 1 & \text{if } Y_{t-2}(\omega) > Y_{t-1}(\omega) > Y_t(\omega) \text{ and } Y_t(\omega) < Y_{t+1}(\omega) < Y_{t+2}(\omega) \\ -1 & \text{if } Y_{t-2}(\omega) < Y_{t-1}(\omega) < Y_t(\omega) \text{ and } Y_t(\omega) > Y_{t+1}(\omega) > Y_{t+2}(\omega) \\ 0 & \text{otherwise} \end{cases}$$

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"

Following [Wec79], we can use simulations to calculate probabilities of P_t and N_t for each period t.

53.4 A Wecker-Like Algorithm

The procedure consists of the following steps:

- index a sample path by ω_i
- for 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 $W_t(\omega_i), W_{t+1}(\omega_i), \dots$
- consider the sets $\{W_t(\omega_i)\}_{i=1}^{T}$, $\{W_{t+1}(\omega_i)\}_{i=1}^{T}$, ... , $\{W_{t+N}(\omega_i)\}_{i=1}^{T}$ as samples from the predictive distributions $f(W_{t+1} \mid y_t, \dots)$, $f(W_{t+2} \mid y_t, y_{t-1}, \dots)$, ..., $f(W_{t+N} \mid y_t, y_{t-1}, \dots)$.

53.5 Using Simulations to Approximate a Posterior Distribution

The next code cells use `pymc` to compute the time t posterior distribution of ρ, σ.

Note that in defining the likelihood function, we choose to condition on the initial value y_0.

```
def draw_from_posterior(sample):
    """
    Draw a sample of size N from the posterior distribution.
    """

    AR1_model = pmc.Model()

    with AR1_model:

        # Start with priors
        rho = pmc.Uniform('rho',lower=-1.,upper=1.)  # Assume stable rho
        sigma = pmc.HalfNormal('sigma', sigma = np.sqrt(10))

        # Expected value of y at the next period (rho * y)
        yhat = rho * sample[:-1]

        # Likelihood of the actual realization.
        y_like = pmc.Normal('y_obs', mu=yhat, sigma=sigma, observed=sample[1:])

    with AR1_model:
        trace = pmc.sample(10000, tune=5000)

    # check condition
    with AR1_model:
        az.plot_trace(trace, figsize=(17, 6))

    rhos = trace.posterior.rho.values.flatten()
    sigmas = trace.posterior.sigma.values.flatten()

    post_sample = {
        'rho': rhos,
        'sigma': sigmas
    }

    return post_sample

post_samples = draw_from_posterior(initial_path)
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

The graphs on the left portray posterior marginal distributions.

53.6 Calculating Sample Path Statistics

Our next step is to prepare Python code to compute our sample path statistics.

```
# define statistics
def next_recession(omega):
    n = omega.shape[0] - 3
    z = np.zeros(n, dtype=int)

    for i in range(n):
        z[i] = int(omega[i] <= omega[i+1] and omega[i+1] > omega[i+2] and omega[i+2] >
 ↪ omega[i+3])

    if np.any(z) == False:
        return 500
    else:
        return np.where(z==1)[0][0] + 1

def minimum_value(omega):
    return min(omega[:8])

def severe_recession(omega):
    z = np.diff(omega)
    n = z.shape[0]

    sr = (z < -.02).astype(int)
    indices = np.where(sr == 1)[0]

    if len(indices) == 0:
        return T1
    else:
        return indices[0] + 1

def next_turning_point(omega):
    """
    Suppose that omega is of length 6
```

(continues on next page)

(continued from previous page)

```
        y_{t-2}, y_{t-1}, y_{t}, y_{t+1}, y_{t+2}, y_{t+3}

    that is sufficient for determining the value of P/N
    """

    n = np.asarray(omega).shape[0] - 4
    T = np.zeros(n, dtype=int)

    for i in range(n):
        if ((omega[i] > omega[i+1]) and (omega[i+1] > omega[i+2]) and
            (omega[i+2] < omega[i+3]) and (omega[i+3] < omega[i+4])):
            T[i] = 1
        elif ((omega[i] < omega[i+1]) and (omega[i+1] < omega[i+2]) and
            (omega[i+2] > omega[i+3]) and (omega[i+3] > omega[i+4])):
            T[i] = -1

    up_turn = np.where(T == 1)[0][0] + 1 if (1 in T) == True else T1
    down_turn = np.where(T == -1)[0][0] + 1 if (-1 in T) == True else T1

    return up_turn, down_turn
```

53.7 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(initial_path, N, ax):
    """
    Plot the predictive distributions from "pure" Wecker's method.
    """
    # Store outcomes
    next_reces = np.zeros(N)
    severe_rec = np.zeros(N)
    min_vals = np.zeros(N)
    next_up_turn, next_down_turn = np.zeros(N), np.zeros(N)

    # Compute .9 confidence interval]
    y0 = initial_path[-1]
    center = np.array([rho**j * y0 for j in range(T1)])
    vars = np.array([sigma**2 * (1 - rho**(2 * j)) / (1 - rho**2) for j in range(T1)])
    y_bounds1_c95, y_bounds2_c95 = center + 1.96 * np.sqrt(vars), center - 1.96 * np.
↪sqrt(vars)
    y_bounds1_c90, y_bounds2_c90 = center + 1.65 * np.sqrt(vars), center - 1.65 * np.
↪sqrt(vars)

    # Plot
    ax[0, 0].set_title("Initial path and predictive densities", fontsize=15)
    ax[0, 0].plot(np.arange(-T0 + 1, 1), initial_path)
    ax[0, 0].set_xlim([-T0, T1])
    ax[0, 0].axvline(0, linestyle='--', alpha=.4, color='k', lw=1)

    # Plot 90% CI
    ax[0, 0].fill_between(np.arange(T1), y_bounds1_c95, y_bounds2_c95, alpha=.3)
```

(continues on next page)

(continued from previous page)

```
    ax[0, 0].fill_between(np.arange(T1), y_bounds1_c90, y_bounds2_c90, alpha=.35)
    ax[0, 0].plot(np.arange(T1), center, color='red', alpha=.7)

    # Simulate future paths
    for n in range(N):
        sim_path = AR1_simulate(rho, sigma, initial_path[-1], T1)
        next_reces[n] = next_recession(np.hstack([initial_path[-3:-1], sim_path]))
        severe_rec[n] = severe_recession(sim_path)
        min_vals[n] = minimum_value(sim_path)
        next_up_turn[n], next_down_turn[n] = next_turning_point(sim_path)

        if n%(N/10) == 0:
            ax[0, 0].plot(np.arange(T1), sim_path, color='gray', alpha=.3, lw=1)

    # Return next_up_turn, next_down_turn
    sns.histplot(next_reces, kde=True, stat='density', ax=ax[0, 1], alpha=.8, label=
↪'True parameters')
    ax[0, 1].set_title("Predictive distribution of time until the next recession",␣
↪fontsize=13)

    sns.histplot(severe_rec, kde=False, stat='density', ax=ax[1, 0], binwidth=0.9,␣
↪alpha=.7, label='True parameters')
    ax[1, 0].set_title(r"Predictive distribution of stopping time of growth$<-2\%$",␣
↪fontsize=13)

    sns.histplot(min_vals, kde=True, stat='density', ax=ax[1, 1], alpha=.8, label=
↪'True parameters')
    ax[1, 1].set_title("Predictive distribution of minimum value in the next 8 periods
↪", fontsize=13)

    sns.histplot(next_up_turn, kde=True, stat='density', ax=ax[2, 0], alpha=.8, label=
↪'True parameters')
    ax[2, 0].set_title("Predictive distribution of time until the next positive turn",
↪ fontsize=13)

    sns.histplot(next_down_turn, kde=True, stat='density', ax=ax[2, 1], alpha=.8,␣
↪label='True parameters')
    ax[2, 1].set_title("Predictive distribution of time until the next negative turn",
↪ fontsize=13)

fig, ax = plt.subplots(3, 2, figsize=(15,12))
plot_Wecker(initial_path, 1000, ax)
plt.show()
```

53.8 Extended Wecker Method

Now we apply we apply our "extended" Wecker method based on predictive densities of y defined by (53.4) that acknowledge posterior uncertainty in the parameters ρ, σ.

To approximate the intergration on the right side of (53.4), we repeatedly draw parameters from the joint posterior distribution each time we simulate a sequence of future values from model (53.1).

```
def plot_extended_Wecker(post_samples, initial_path, N, ax):
    """
    Plot the extended Wecker's predictive distribution
    """
    # Select a sample
    index = np.random.choice(np.arange(len(post_samples['rho'])), N + 1,␣
↪replace=False)
    rho_sample = post_samples['rho'][index]
    sigma_sample = post_samples['sigma'][index]

    # Store outcomes
    next_reces = np.zeros(N)
    severe_rec = np.zeros(N)
    min_vals = np.zeros(N)
```

(continues on next page)

(continued from previous page)

```
    next_up_turn, next_down_turn = np.zeros(N), np.zeros(N)

    # Plot
    ax[0, 0].set_title("Initial path and future paths simulated from posterior draws",
 ↪ fontsize=15)
    ax[0, 0].plot(np.arange(-T0 + 1, 1), initial_path)
    ax[0, 0].set_xlim([-T0, T1])
    ax[0, 0].axvline(0, linestyle='--', alpha=.4, color='k', lw=1)

    # Simulate future paths
    for n in range(N):
        sim_path = AR1_simulate(rho_sample[n], sigma_sample[n], initial_path[-1], T1)
        next_reces[n] = next_recession(np.hstack([initial_path[-3:-1], sim_path]))
        severe_rec[n] = severe_recession(sim_path)
        min_vals[n] = minimum_value(sim_path)
        next_up_turn[n], next_down_turn[n] = next_turning_point(sim_path)

        if n % (N / 10) == 0:
            ax[0, 0].plot(np.arange(T1), sim_path, color='gray', alpha=.3, lw=1)

    # Return next_up_turn, next_down_turn
    sns.histplot(next_reces, kde=True, stat='density', ax=ax[0, 1], alpha=.6,␣
 ↪color=colors[1], label='Sampling from posterior')
    ax[0, 1].set_title("Predictive distribution of time until the next recession",␣
 ↪fontsize=13)

    sns.histplot(severe_rec, kde=False, stat='density', ax=ax[1, 0], binwidth=.9,␣
 ↪alpha=.6, color=colors[1], label='Sampling from posterior')
    ax[1, 0].set_title(r"Predictive distribution of stopping time of growth$<-2\%$",␣
 ↪fontsize=13)

    sns.histplot(min_vals, kde=True, stat='density', ax=ax[1, 1], alpha=.6,␣
 ↪color=colors[1], label='Sampling from posterior')
    ax[1, 1].set_title("Predictive distribution of minimum value in the next 8 periods
 ↪", fontsize=13)

    sns.histplot(next_up_turn, kde=True, stat='density', ax=ax[2, 0], alpha=.6,␣
 ↪color=colors[1], label='Sampling from posterior')
    ax[2, 0].set_title("Predictive distribution of time until the next positive turn",
 ↪ fontsize=13)

    sns.histplot(next_down_turn, kde=True, stat='density', ax=ax[2, 1], alpha=.6,␣
 ↪color=colors[1], label='Sampling from posterior')
    ax[2, 1].set_title("Predictive distribution of time until the next negative turn",
 ↪ fontsize=13)

fig, ax = plt.subplots(3, 2, figsize=(15, 12))
plot_extended_Wecker(post_samples, initial_path, 1000, ax)
plt.show()
```

53.9 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=(15,12))
plot_Wecker(initial_path, 1000, ax)
ax[0, 0].clear()
plot_extended_Wecker(post_samples, initial_path, 1000, ax)
plt.legend()
plt.show()
```

Initial path and future paths simulated from posterior draws
Predictive distribution of time until the next recession
Density
Predictive distribution of stopping time of growth < −2%
Predictive distribution of minimum value in the next 8 periods
Predictive distribution of time until the next positive turn
Predictive distribution of time until the next negative turn
True parameters
Sampling from posterior

Part VIII

Information

CHAPTER

FIFTYFOUR

JOB SEARCH VII: SEARCH WITH LEARNING

Contents

- *Job Search VII: 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
```

54.1 Overview

In this lecture, we consider an extension of the previously studied job search model of McCall [McC70].

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 [LS18], section 6.6.

Let's start with some imports

```
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
from numba import njit, prange, vectorize
from interpolation import mlinterp, interp
from math import gamma
import numpy as np
from matplotlib import cm
import scipy.optimize as op
from scipy.stats import cumfreq, beta
```

```
/opt/conda/envs/quantecon/lib/python3.11/site-packages/numba/core/decorators.
↪py:262: NumbaDeprecationWarning: numba.generated_jit is deprecated. Please see␣
↪the documentation at: https://numba.readthedocs.io/en/stable/reference/
↪deprecation.html#deprecation-of-generated-jit for more information and advice on␣
↪a suitable replacement.
  warnings.warn(msg, NumbaDeprecationWarning)
```

54.1.1 Model Features

- Infinite horizon dynamic programming with two states and one binary control.
- Bayesian updating to learn the unknown distribution.

54.2 Model

Let's first review the basic McCall model [McC70] and then add the variation we want to consider.

54.2.1 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

$$v(w) = \max\left\{\frac{w}{1-\beta}, c + \beta\int v(w')q(w')dw'\right\} \tag{54.1}$$

The optimal policy has the form $\mathbf{1}\{w \geq \bar{w}\}$, where $\bar{w}$ is a constant called the *reservation wage*.

54.2.2 Offer Distribution Unknown

Now let's extend the model by considering the variation presented in [LS18], 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

$$\pi_{t+1} = \frac{\pi_t f(w_{t+1})}{\pi_t f(w_{t+1}) + (1 - \pi_t) g(w_{t+1})} \tag{54.2}$$

This last expression follows from Bayes' rule, which tells us that

$$\mathbb{P}\{q = f \,|\, W = w\} = \frac{\mathbb{P}\{W = w \,|\, q = f\}\mathbb{P}\{q = f\}}{\mathbb{P}\{W = w\}} \quad \text{and} \quad \mathbb{P}\{W = w\} = \sum_{\omega \in \{f, g\}} \mathbb{P}\{W = w \,|\, q = \omega\}\mathbb{P}\{q = \omega\}$$

The fact that (54.2) is recursive allows us to progress to a recursive solution method.

Letting

$$q_\pi(w) := \pi f(w) + (1 - \pi) g(w) \quad \text{and} \quad \kappa(w, \pi) := \frac{\pi f(w)}{\pi f(w) + (1 - \pi) g(w)}$$

we can express the value function for the unemployed worker recursively as follows

$$v(w, \pi) = \max \left\{ \frac{w}{1 - \beta}, \, c + \beta \int v(w', \pi') \, q_\pi(w') \, dw' \right\} \quad \text{where} \quad \pi' = \kappa(w', \pi) \tag{54.3}$$

Notice that the current guess π is a state variable, since it affects the worker's perception of probabilities for future rewards.

54.2.3 Parameterization

Following section 6.6 of [LS18], our baseline parameterization will be

- f is $\text{Beta}(1, 1)$
- g is $\text{Beta}(3, 1.2)$

- $\beta = 0.95$ 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()
```

54.2.4 Looking Forward

What kind of optimal policy might result from (54.3) 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 $\bar{w}$.

54.3 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 = njit(lambda x: p(x, F_a, F_b))
        self.g = njit(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

    @njit
    def v_func(x, y, v):
        return mlinterp((w_grid, π_grid), v, (x, y))

    @njit
    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

    @njit(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

    @njit(parallel=parallel_flag)
    def get_greedy(v):
        """"
        Compute optimal actions taking v as the value function.

        """
        σ = np.empty_like(v)
```

(continues on next page)

(continued from previous page)

```
        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

                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
```

(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
```

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)
ax.set(xlabel='$\pi$', ylabel='$w$')

plt.show()
```

```
Error at iteration 5 is 0.6352728434103625.
```

```
Error at iteration 10 is 0.11119458294702866.
```

```
Error at iteration 15 is 0.025347512515775605.
```

```
Error at iteration 20 is 0.00588337804527761.
```

```
Error at iteration 25 is 0.0013653197403939998.
```

```
Error at iteration 30 is 0.000319287578243177.
```

```
Converged in 34 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")
ax.set(xlabel='$\pi$', 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.

54.4 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.

54.5 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 (54.3) have equal value:

$$\frac{\bar{w}(\pi)}{1-\beta} = c + \beta \int v(w', \pi')\, q_\pi(w')\, dw' \tag{54.4}$$

Together, (54.3) and (54.4) give

$$v(w, \pi) = \max\left\{\frac{w}{1-\beta}, \frac{\bar{w}(\pi)}{1-\beta}\right\} \tag{54.5}$$

Combining (54.4) and (54.5), 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

$$\bar{w}(\pi) = (1-\beta)c + \beta \int \max\left\{w', \bar{w} \circ \kappa(w', \pi)\right\}\, q_\pi(w')\, dw' \tag{54.6}$$

Equation (54.6) 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.

54.6 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

$$(Q\omega)(\pi) = (1-\beta)c + \beta \int \max\left\{w', \omega \circ \kappa(w', \pi)\right\}\, q_\pi(w')\, dw' \tag{54.7}$$

Comparing (54.6) and (54.7), 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 (54.6) and vice versa.

Moreover, for any $\omega, \omega' \in b[0,1]$, basic algebra and the triangle inequality for integrals tells us that

$$|(Q\omega)(\pi) - (Q\omega')(\pi)| \leq \beta \int |\max\left\{w', \omega \circ \kappa(w', \pi)\right\} - \max\left\{w', \omega' \circ \kappa(w', \pi)\right\}|\, q_\pi(w')\, dw' \tag{54.8}$$

Working case by case, it is easy to check that for real numbers a, b, c we always have

$$|\max\{a, b\} - \max\{a, c\}| \leq |b - c| \tag{54.9}$$

Combining (54.8) and (54.9) yields

$$|(Q\omega)(\pi) - (Q\omega')(\pi)| \leq \beta \int |\omega \circ \kappa(w', \pi) - \omega' \circ \kappa(w', \pi)| \; q_\pi(w') \, dw' \leq \beta \|\omega - \omega'\| \tag{54.10}$$

Taking the supremum over π now gives us

$$\|Q\omega - Q\omega'\| \leq \beta \|\omega - \omega'\| \tag{54.11}$$

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 $\omega \in b[0,1]$.

54.7 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

    @njit
    def ω_func(p, ω):
        return interp(π_grid, ω, p)

    @njit
    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

    @njit(parallel=parallel_flag)
    def Q(ω):
        """

        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
```

(continues on next page)

(continued from previous page)

```
            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 $\bar{w}$.

54.8 Exercises

Exercise 54.8.1

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.

54.9 Solutions

Solution to Exercise 54.8.1

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)
```

(continues on next page)

(continued from previous page)

```
    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='$\pi$', ylabel='$w$')
ax.grid()
plt.show()
```

```
Error at iteration 5 is 0.02236386911901056.
Error at iteration 10 is 0.0067711396559273496.
Error at iteration 15 is 0.0016356227647515986.
Error at iteration 20 is 0.0003751580882346284.
Error at iteration 25 is 8.418971314749779e-05.

Converged in 25 iterations.
```

54.10 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 = njit(lambda x: interp(π_grid, w_bar, x))
```

(continues on next page)

(continued from previous page)

```
@njit
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

@njit
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')
```

(continues on next page)

(continued from previous page)

```
ax.set_xlabel('Time')
ax.set_title('Unemployment rate')
ax.legend()
plt.show()
```

54.11 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.

```
@njit
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.
```

(continues on next page)

(continued from previous page)

```
    Parameters
    ----------

    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 >= interp(π_grid, w_bar, π):
                    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
```

(continues on next page)

(continued from previous page)

```
    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)
```

(continues on next page)

(continued from previous page)

```
    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)))
    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='$\pi$', 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='$\pi$', ylabel='$\mathbb{P}\{w > \overline{w} (\pi)\}$')

    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")
```

(continues on next page)

(continued from previous page)

```
    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.

54.12 Examples

54.12.1 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()
```

$l(w) = f(w)/g(w)$
l
accept
reject
π
$f(w), g(w)$
f
g
$\mathbb{P}\{w > \overline{w}(\pi)\}$

54.12.2 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
π
f(w), g(w)
f
g
$\mathbb{P}\{w > \overline{w}(\pi)\}$

54.12.3 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)
```

$l(w) = f(w)/g(w)$
l
accept
reject
π
w
$f(w), g(w)$
f
g
$\mathbb{P}\{w > \overline{w}(\pi)\}$

54.12.4 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)
```

$l(w) = f(w)/g(w)$
l
w
accept
reject
π
$f(w), g(w)$
f
g
$\mathbb{P}^{\pi}\{w > \overline{w}(\pi)\}$

54.12.5 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)
```

l
$l(w) = f(w)/g(w)$
w
accept
reject
π
f
g
$f(w), g(w)$
$\mathbb{P}^{\pi}\{w > \overline{w}(\pi)\}$

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

CHAPTER

FIFTYFIVE

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*
  - *Kullback–Leibler Divergence*
  - *Sequels*

55.1 Overview

This lecture describes likelihood ratio processes and some of their uses.

We'll use a setting described in *this lecture*.

Among things that we'll learn are

- A peculiar property of likelihood ratio processes
- 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 during World War II the United States Navy devised a decision rule that Captain Garret L. Schyler challenged and asked Milton Friedman to justify to him, a topic to be studied in *this lecture*

Let's start by importing some Python tools.

```
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
from numba import vectorize, njit
```

(continues on next page)

(continued from previous page)

```
from math import gamma
from scipy.integrate import quad
```

55.2 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 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 key component of a likelihood ratio process, namely, the time t likelihood ratio as the random variable

$$\ell(w_t) = \frac{f\left(w_t\right)}{g\left(w_t\right)}, \quad t \geq 1.$$

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

$$L\left(w^t\right) = \prod_{i=1}^t \ell(w_i),$$

where $w^t = \{w_1, \dots, w_t\}$ 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* 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 [NP33].

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 distributionss, 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 = njit(lambda x: p(x, F_a, F_b))
g = njit(lambda x: p(x, G_a, G_b))
```

```
@njit
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
```

55.3 Nature Permanently Draws from Density g

We first simulate the likelihood ratio process when nature permanently draws from g.

```
l_arr_g = simulate(G_a, G_b)
l_seq_g = np.cumprod(l_arr_g, axis=1)
```

```
N, T = l_arr_g.shape

for i in range(N):

    plt.plot(range(T), l_seq_g[i, :], color='b', lw=0.8, alpha=0.5)

plt.ylim([0, 3])
plt.title("$L(w^{t})$ paths");
```

Evidently, as sample length T grows, most probability mass shifts toward zero

To see it this more clearly clearly, we plot over time the fraction of paths $L\left(w^t\right)$ that fall in the interval $[0, 0.01]$.

```
plt.plot(range(T), np.sum(l_seq_g <= 0.01, axis=0) / N)
```

```
[<matplotlib.lines.Line2D at 0x7f380235e350>]
```

Despite the evident convergence of most probability mass to a very small interval near 0, the unconditional mean of $L\left(w^t\right)$ under probability density g is identically 1 for all t.

To verify this assertion, first notice that as mentioned earlier the unconditional mean $E\left[\ell\left(w_t\right) \mid q=g\right]$ is 1 for all t:

$$\begin{aligned} E\left[\ell\left(w_t\right) \mid q=g\right] &= \int \frac{f\left(w_t\right)}{g\left(w_t\right)} g\left(w_t\right) d w_t \\ &= \int f\left(w_t\right) d w_t \\ &= 1, \end{aligned}$$

which immediately implies

$$\begin{aligned} E\left[L\left(w^1\right) \mid q=g\right] &= E\left[\ell\left(w_1\right) \mid q=g\right] \\ &= 1. \end{aligned}$$

Because $L(w^t) = \ell(w_t)L(w^{t-1})$ and $\{w_t\}_{t=1}^t$ is an IID sequence, we have

$$\begin{aligned} E\left[L\left(w^t\right) \mid q=g\right] &= E\left[L\left(w^{t-1}\right) \ell\left(w_t\right) \mid q=g\right] \\ &= E\left[L\left(w^{t-1}\right) E\left[\ell\left(w_t\right) \mid q=g, w^{t-1}\right] \mid q=g\right] \\ &= E\left[L\left(w^{t-1}\right) E\left[\ell\left(w_t\right) \mid q=g\right] \mid q=g\right] \\ &= E\left[L\left(w^{t-1}\right) \mid q=g\right] \end{aligned}$$

for any $t \geq 1$.

Mathematical induction implies $E\left[L\left(w^t\right) \mid q=g\right]=1$ for all $t \geq 1$.

55.4 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 $t \rightarrow+\infty$?

The answer has to be that as $t \rightarrow+\infty$, 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 = simulate(G_a, G_b, N=50000)
l_seq_g = np.cumprod(l_arr_g, axis=1)
```

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 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 a way to 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.

55.5 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 $\ell(w_t)$ under density g is 1, its mean under the density f exceeds one.

To see this, we compute

$$
\begin{aligned}
E\left[\ell(w_t) \mid q=f\right] &= \int \frac{f(w_t)}{g(w_t)} f(w_t)\, dw_t \\
&= \int \frac{f(w_t)}{g(w_t)} \frac{f(w_t)}{g(w_t)} g(w_t)\, dw_t \\
&= \int \ell(w_t)^2 g(w_t)\, dw_t \\
&= E\left[\ell(w_t)^2 \mid q=g\right] \\
&= E\left[\ell(w_t) \mid q=g\right]^2 + Var\left(\ell(w_t) \mid q=g\right) \\
&> E\left[\ell(w_t) \mid q=g\right]^2 = 1
\end{aligned}
$$

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.

```
l_arr_f = simulate(F_a, F_b, N=50000)
l_seq_f = np.cumprod(l_arr_f, axis=1)
```

```
N, T = l_arr_f.shape
plt.plot(range(T), np.mean(l_seq_f, axis=0))
```

```
[<matplotlib.lines.Line2D at 0x7f380311f5d0>]
```

We also plot the probability that $L(w^t)$ 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)
```

```
[<matplotlib.lines.Line2D at 0x7f3802ed9b90>]
```

55.6 Likelihood Ratio Test

We now describe how to employ the machinery of Neyman and Pearson [NP33] to test the hypothesis that history w^t is generated by repeated IID draws from density g.

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:

- reject H_0 if $L(W^t) < c$,
- accept H_0 otherwise.

where c is a given discrimination threshold, to be chosen in a way we'll soon describe.

This test is *best* in the sense that it is a **uniformly most powerful** test.

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 detection (= power = 1 minus probability of Type II error):

$$1 - \beta \equiv \Pr\left\{L\left(w^t\right) < c \mid q = g\right\}$$

- Probability of false alarm (= significance level = probability of Type I error):

$$\alpha \equiv \Pr\left\{L\left(w^t\right) < c \mid q = f\right\}$$

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**.

To have made a good inference, 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 q is what makes it possible to distinguish $q = f$ from $q = g$.

```
fig, axs = plt.subplots(2, 2, figsize=(12, 8))
fig.suptitle('distribution of $log(L(w^t))$ under f or g', fontsize=15)

for i, t in enumerate([1, 7, 14, 21]):
    nr = i // 2
    nc = 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")

    for i, (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 i == 0 else x[1:] > np.log(c)
        axs[nr, nc].fill_between(x[1:][ind], hist[ind], alpha=0.5, label=label)
```

(continues on next page)

(continued from previous page)

```
    axs[nr, nc].legend()
    axs[nr, nc].set_title(f"t={t}")

plt.show()
```

The graph below shows more clearly that, when we hold the threshold c fixed, the probability of detection monotonically increases with increases in t and that the probability of a false alarm monotonically decreases.

```
PD = np.empty(T)
PFA = np.empty(T)

for t in range(T):
    PD[t] = np.sum(l_seq_g[:, t] < c) / N
    PFA[t] = np.sum(l_seq_f[:, t] < c) / N

plt.plot(range(T), PD, label="Probability of detection")
plt.plot(range(T), PFA, label="Probability of false alarm")
plt.xlabel("t")
plt.title("$c=1$")
plt.legend()
plt.show()
```

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 what is called a receiver operating characteristic curve.

Below, we plot receiver operating characteristic curves for different sample sizes t.

```
PFA = np.arange(0, 100, 1)

for t in range(1, 15, 4):
    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("Receiver Operating Characteristic Curve")
plt.show()
```

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.

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)

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*.

55.7 Kullback–Leibler Divergence

Now let's consider a case in which neither g nor f generates the data.

Instead, a third distribution h does.

Let's watch how how the cumulated likelihood ratios f/g behave when h governs the data.

A key tool here is called **Kullback–Leibler divergence**.

It is also called **relative entropy**.

It measures how one probability distribution differs from another.

In our application, we want to measure how f or g diverges from h

The two Kullback–Leibler divergences pertinent for us are K_f and K_g defined as

$$
\begin{aligned}
K_f &= E_h \left[\log \left(\frac{f(w)}{h(w)} \right) \frac{f(w)}{h(w)} \right] \\
&= \int \log \left(\frac{f(w)}{h(w)} \right) \frac{f(w)}{h(w)} h(w) \, dw \\
&= \int \log \left(\frac{f(w)}{h(w)} \right) f(w) \, dw
\end{aligned}
$$

$$
\begin{aligned}
K_g &= E_h \left[\log \left(\frac{g(w)}{h(w)} \right) \frac{g(w)}{h(w)} \right] \\
&= \int \log \left(\frac{g(w)}{h(w)} \right) \frac{g(w)}{h(w)} h(w) \, dw \\
&= \int \log \left(\frac{g(w)}{h(w)} \right) g(w) \, dw
\end{aligned}
$$

When $K_g < K_f$, g is closer to h than f is.

- In that case we'll find that $L(w^t) \to 0$.

When $K_g > K_f$, f is closer to h than g is.

- In that case we'll find that $L(w^t) \to +\infty$

We'll now experiment with an h is also a beta distribution

We'll start by setting parameters G_a and G_b so that h is closer to g

```
H_a, H_b = 3.5, 1.8

h = njit(lambda x: p(x, H_a, H_b))
```

```
x_range = np.linspace(0, 1, 100)
plt.plot(x_range, f(x_range), label='f')
plt.plot(x_range, g(x_range), label='g')
plt.plot(x_range, h(x_range), label='h')

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

Let's compute the Kullback–Leibler discrepancies by quadrature integration.

```
def KL_integrand(w, q, h):

    m = q(w) / h(w)

    return np.log(m) * q(w)
```

```
def compute_KL(h, f, g):

    Kf, _ = quad(KL_integrand, 0, 1, args=(f, h))
    Kg, _ = quad(KL_integrand, 0, 1, args=(g, h))

    return Kf, Kg
```

```
Kf, Kg = compute_KL(h, f, g)
Kf, Kg
```

```
(0.7902536603660161, 0.08554075759988769)
```

We have $K_g < K_f$.

Next, we can verify our conjecture about $L\left(w^t\right)$ by simulation.

```
l_arr_h = simulate(H_a, H_b)
l_seq_h = np.cumprod(l_arr_h, axis=1)
```

The figure below plots over time the fraction of paths $L\left(w^t\right)$ that fall in the interval $[0, 0.01]$.

Notice that it converges to 1 as expected when g is closer to h than f is.

```
N, T = l_arr_h.shape
plt.plot(range(T), np.sum(l_seq_h <= 0.01, axis=0) / N)
```

```
[<matplotlib.lines.Line2D at 0x7f3802cf5f50>]
```

We can also try an h that is closer to f than is g so that now K_g is larger than K_f.

```
H_a, H_b = 1.2, 1.2
h = njit(lambda x: p(x, H_a, H_b))
```

```
Kf, Kg = compute_KL(h, f, g)
Kf, Kg
```

```
(0.01239249754452668, 0.35377684280997646)
```

```
l_arr_h = simulate(H_a, H_b)
l_seq_h = np.cumprod(l_arr_h, axis=1)
```

Now probability mass of $L(w^t)$ falling above 10000 diverges to $+\infty$.

```
N, T = l_arr_h.shape
plt.plot(range(T), np.sum(l_seq_h > 10000, axis=0) / N)
```

```
[<matplotlib.lines.Line2D at 0x7f38005cdf50>]
```

55.8 Sequels

Likelihood processes play an important role in Bayesian learning, as described in *this lecture* and as applied in *this lecture*.

Likelihood ratio processes appear again in this lecture, which contains another illustration of the **peculiar property** of likelihood ratio processes described above.

CHAPTER

FIFTYSIX

COMPUTING MEAN OF A LIKELIHOOD RATIO PROCESS

Contents

- *Computing 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*
  - *More Thoughts about Choice of Sampling Distribution*

56.1 Overview

In *this lecture* we described a peculiar property of a likelihood ratio process, namely, that it's 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 $L(\omega^t) = \prod_{i=1}^{t} \ell(\omega_i)$.

We start by importing some Python packages.

```
import numpy as np
from numba import njit, vectorize, prange
import matplotlib.pyplot as plt
from math import gamma
```

56.2 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 = njit(lambda w: p(w, F_a, F_b))
g = njit(lambda w: p(w, G_a, G_b))
```

```
w_range = np.linspace(1e-5, 1-1e-5, 1000)

plt.plot(w_range, g(w_range), label='g')
plt.plot(w_range, f(w_range), label='f')
plt.xlabel('$\omega$')
plt.legend()
plt.title('density functions $f$ and $g$')
plt.show()
```

The likelihood ratio is `l(w)=f(w)/g(w)`.

```
l = njit(lambda w: f(w) / g(w))
```

```
plt.plot(w_range, l(w_range))
plt.title('$\ell(\omega)$')
plt.xlabel('$\omega$')
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.

56.3 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:

$$E^g\left[\ell\left(\omega\right)\right] = \int_\Omega \ell(\omega)g(\omega)d\omega = \int_\Omega \ell(\omega)\frac{g(\omega)}{h(\omega)}h(\omega)d\omega = E^h\left[\ell\left(\omega\right)\frac{g(\omega)}{h(\omega)}\right]$$

56.4 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()
```

56.5 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.

```
@njit(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)
```

```
0.9727039894749833
```

For our importance sampling estimate, we set $q = h$.

```
estimate(g_a, g_b, h_a, h_b, T=1, N=10000)
```

```
1.0043287471442088
```

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)
```

```
0.3977809631297837
```

```
estimate(g_a, g_b, h_a, h_b, T=10, N=10000)
```

```
1.043725489325251
```

56.6 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.

```
@njit(parallel=True)
def simulate(p_a, p_b, q_a, q_b, N_simu, T=1):
```

(continues on next page)

(continued from previous page)

```
    μ_L_p = np.empty(N_simu)
    μ_L_q = np.empty(N_simu)

    for i in prange(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)
```

```
(0.9981753342395336, 0.005403634415149236)
```

```
# importance sampling (mean and std)
np.nanmean(μ_L_q), np.nanvar(μ_L_q)
```

```
(1.0000558248698244, 2.596159010001882e-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')
    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],
```

(continues on next page)

(continued from previous page)

```
                                    [n_q, bins_q, μ_hat_q, σ_hat_q]]:
        idx = np.argmax(n)
        axs[row, col].text(bins[idx], n[idx], '$\hat{μ}$='+f'{μ_hat:.4g}'+', $\hat{σ}=
↪$'+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.

56.7 More Thoughts about Choice of Sampling Distribution

Above, we arbitraily 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)
```

```
(1.0, 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'p=Beta({g_a}, {g_b})')
plt.plot(w_range, p(w_range, a_list[0], b_list[0]), label=f'g=Beta({a_list[0]}, {b_
↪list[0]})')
plt.plot(w_range, p(w_range, a_list[1], b_list[1]), label=f'g=Beta({a_list[1]}, {b_
↪list[1]})')
plt.plot(w_range, p(w_range, a_list[2], b_list[2]), label=f'g=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 additonal 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], '$\hat{μ}$='+f'{μ_hat:.4g}'+', $\hat{σ}=$'+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))
```

(continues on next page)

(continued from previous page)

```
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)

    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], '$\hat{μ}$='+f'{μ_hat:.4g}'+', $\hat{σ}=$'+f'
↪{σ_hat:.4g}')

plt.show()
```

However, h_3 is evidently a poor importance sampling distribution forpir problem, with a mean estimate far away from 1 for $T = 20$.

Notice that evan at $T = 1$, the mean estimate with importance sampling is more biased than just sampling with g itself.

Thus, our simulations suggest that we would be better off simply using Monte Carlo approximations under g than using h_3 as an importance sampling distribution for our problem.

CHAPTER

FIFTYSEVEN

A PROBLEM THAT STUMPED MILTON FRIEDMAN

(and that Abraham Wald solved by inventing sequential analysis)

Contents

- *A Problem that Stumped Milton Friedman*
  - *Overview*
  - *Origin of the Problem*
  - *A Dynamic Programming Approach*
  - *Implementation*
  - *Analysis*
  - *Comparison with Neyman-Pearson Formulation*
  - *Sequels*

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install interpolation
```

57.1 Overview

This lecture describes a 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.

This problem led Abraham Wald [Wal47] to formulate **sequential analysis**, an approach to statistical decision problems intimately related to dynamic programming.

In this lecture, we apply dynamic programming algorithms to Friedman and Wallis and Wald's problem.

Key ideas in play will be:

- Bayes' Law
- Dynamic programming
- 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 occures 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**

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 interpolation import interp
from math import gamma
```

```
/opt/conda/envs/quantecon/lib/python3.11/site-packages/numba/core/decorators.
↪py:262: NumbaDeprecationWarning: numba.generated_jit is deprecated. Please see␣
↪the documentation at: https://numba.readthedocs.io/en/stable/reference/
↪deprecation.html#deprecation-of-generated-jit for more information and advice on␣
↪a suitable replacement.
  warnings.warn(msg, NumbaDeprecationWarning)
```

This lecture uses ideas studied in *this lecture*, *this lecture*. and *this lecture*.

57.2 Origin of the Problem

On pages 137-139 of his 1998 book *Two Lucky People* with Rose Friedman [FF98], 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.

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. Schyler, the captain objected that such a test, to quote from Allen's account, may prove wasteful. If a wise and seasoned ordnance officer like Schyler 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 struggled with the problem but, after realizing that they were not able to solve it, described the problem to Abraham Wald.

That started Wald on the path that led him to *Sequential Analysis* [Wal47].

We'll formulate the problem using dynamic programming.

57.3 A Dynamic Programming Approach

The following presentation of the problem closely follows Dmitri Berskekas's treatment in **Dynamic Programming and Stochastic Control** [Ber75].

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

$$\pi_{-1} = \mathbb{P}\{f = f_0 \mid \text{ no observations}\} \in (0,1)$$

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_0 to

$$\pi_k = \mathbb{P}\{f = f_0 \mid z_k, z_{k-1}, \ldots, z_0\}$$

which is calculated recursively by applying Bayes' law:

$$\pi_{k+1} = \frac{\pi_k f_0(z_{k+1})}{\pi_k f_0(z_{k+1}) + (1-\pi_k) f_1(z_{k+1})}, \quad k = -1, 0, 1, \ldots$$

After observing $z_k, z_{k-1}, \ldots, z_0$, the decision-maker believes that z_{k+1} has probability distribution

$$f_{\pi_k}(v) = \pi_k f_0(v) + (1-\pi_k) f_1(v),$$

which is a mixture of distributions f_0 and f_1, with the weight on f_0 being the posterior probability that $f = f_0$[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 [Kre88] chapter 11, for discussions of exchangeability.

```
@jit(nopython=True)
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 = π * f0(grid) + (1 - π) * f1(grid)
    axes[1].plot(y, lw=2, label=f"$\pi_k$ = {π}")

for ax in axes:
    ax.legend()
    ax.set(xlabel="$z$ values", ylabel="probability of $z_k$")

plt.tight_layout()
plt.show()
```

57.3.1 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

57.3.2 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.

57.3.3 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 $f = f_0$.

If, on the other hand, π is close to 0, then $f = f_1$ 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 decision rule such as the one shown in the figure

As we'll see, this is indeed the correct form of the decision rule.

Our problem is to determine threshold values α, β 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 α or β.

57.3.4 A Bellman Equation

Let $J(\pi)$ be the total loss for a decision-maker with current belief π who chooses optimally.

With some thought, you will agree that J should satisfy the Bellman equation

$$J(\pi) = \min \{(1-\pi)L_0,\ \pi L_1,\ c + \mathbb{E}[J(\pi')]\} \tag{57.1}$$

where π' is the random variable defined by Bayes' Law

$$\pi' = \kappa(z', \pi) = \frac{\pi f_0(z')}{\pi f_0(z') + (1-\pi) f_1(z')}$$

when π is fixed and z' is drawn from the current best guess, which is the distribution f defined by

$$f_\pi(v) = \pi f_0(v) + (1-\pi) f_1(v)$$

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

$$J(\pi) = \min\{(1-\pi)L_0,\ \pi L_1,\ h(\pi)\} \tag{57.2}$$

where $\pi \in [0,1]$ and

- $(1-\pi)L_0$ is the expected loss associated with accepting f_0 (i.e., the cost of making a type II error).
- πL_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 $\alpha, \beta \in (0,1) \times (0,1)$ that satisfy

$$(1-\pi)L_0 < \min\{\pi L_1, c + \mathbb{E}[J(\pi')]\} \text{ if } \pi \geq \alpha$$

and

$$\pi L_1 < \min\{(1-\pi)L_0, c + \mathbb{E}[J(\pi')]\} \text{ if } \pi \leq \beta$$

The optimal decision rule is then

$$\begin{aligned} \text{accept } f = f_0 \text{ if } \pi \geq \alpha \\ \text{accept } f = f_1 \text{ if } \pi \leq \beta \\ \text{draw another } z \text{ if } \beta \leq \pi \leq \alpha \end{aligned}$$

Our aim is to compute the cost function J, and from it the associated cutoffs α and β.

To make our computations manageable, using (57.2), we can write the continuation cost $h(\pi)$ as

$$\begin{aligned} h(\pi) &= c + \mathbb{E}[J(\pi')] \\ &= c + \mathbb{E}_{\pi'} \min\{(1-\pi')L_0, \pi' L_1, h(\pi')\} \\ &= c + \int \min\{(1-\kappa(z',\pi))L_0, \kappa(z',\pi)L_1, h(\kappa(z',\pi))\} f_\pi(z') dz' \end{aligned} \tag{57.3}$$

The equality

$$h(\pi) = c + \int \min\{(1-\kappa(z',\pi))L_0, \kappa(z',\pi)L_1, h(\kappa(z',\pi))\} f_\pi(z') dz' \tag{57.4}$$

is a **functional equation** in an unknown function h.

Using the functional equation, (57.4), for the continuation cost, we can back out optimal choices using the right side of (57.2).

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

$$Qh(\pi) = c + \int \min\{(1-\kappa(z',\pi))L_0, \kappa(z',\pi)L_1, h(\kappa(z',\pi))\} f_\pi(z') dz'$$

57.4 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 = π * f0(z), (1 - π) * f1(z)
        π_new = π_f0 / (π_f0 + π_f1)

        return π_new
```

As in the *optimal growth lecture*, 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: interp(π_grid, h, p)

    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

        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(nopython=True)
def solve_model(wf, tol=1e-4, max_iter=1000):
    """
    Compute the continuation cost function
```

(continues on next page)

(continued from previous page)

```
    * 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
```

57.5 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()
```

57.5.1 Value 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 α and β and plot these on our cost function plot

```
@jit(nopython=True)
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
    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)
    β = π_grid[np.searchsorted(
                              payoff_f1 - np.minimum(h, payoff_f0),
                              1e-10)
```

(continues on next page)

(continued from previous page)

```
                - 1]
    α = π_grid[np.searchsorted(
                                np.minimum(h, payoff_f1) - payoff_f0,
                                1e-10)
                - 1]

    return (β, α)

β, α = 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='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='$J(\pi)$')

ax.annotate(r"$\beta$", xy=(β + 0.01, 0.5), fontsize=14)
ax.annotate(r"$\alpha$", xy=(α + 0.01, 0.5), fontsize=14)

plt.vlines(β, 0, β * wf.L0, linestyle="--")
plt.vlines(α, 0, (1 - α) * wf.L1, linestyle="--")

ax.set(xlim=(0, 1), ylim=(0, 0.5 * max(wf.L0, wf.L1)), ylabel="cost",
       xlabel="$\pi$", title="Cost function $J(\pi)$")

plt.legend(borderpad=1.1)
plt.show()
```

The cost function J equals πL_1 for $\pi \leq \beta$, and $(1-\pi)L_0$ for $\pi \geq \alpha$.

The slopes of the two linear pieces of the cost function $J(\pi)$ are determined by L_1 and $-L_0$.

The cost function J is smooth in the interior region, where the posterior probability assigned to f_0 is in the indecisive region $\pi \in (\beta, \alpha)$.

The decision-maker continues to sample until the probability that he attaches to model f_0 falls below β or above α.

57.5.2 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
    β, α = 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:
        # Maybe should specify which distribution is correct one so that
        # the draws come from the "right" distribution
        z = f_rvs()
        t = t + 1
        π = κ(z, π)
        if π < β:
            decision_made = True
            decision = 1
        elif π > α:
            decision_made = True
            decision = 0

    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
```

(continues on next page)

(continued from previous page)

```
    return cdist, tdist

def simulation_plot(wf):
    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)
```

57.5.3 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.

57.5.4 A Notebook Implementation

To facilitate comparative statics, we provide a Jupyter notebook that generates the same plots, but with sliders.

With these sliders, you can adjust parameters and immediately observe

- 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 piece-wise 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.

57.6 Comparison with Neyman-Pearson Formulation

For several reasons, it is useful to describe the theory underlying the test that Navy Captain G. S. Schuyler had been told to use and that led him to approach Milton Friedman and Allan Wallis to convey his conjecture that superior practical procedures existed.

Evidently, the Navy had told Captail Schuyler to use what it knew to be a state-of-the-art Neyman-Pearson test.

We'll rely on Abraham Wald's [Wal47] elegant summary of Neyman-Pearson theory.

For our purposes, watch for there 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 the probabilities α and β defined in the Neyman-Pearson theory

Recall that in the sequential analytic formulation above, that

- The sample size n is not fixed but rather an object to be chosen; technically n is a random variable.
- The parameters β and α characterize cut-off rules used to determine n as a random variable.

- Laws of large numbers make no appearances in the sequential construction.

In chapter 1 of **Sequential Analysis** [Wal47] 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 possible 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.

57.7 Sequels

We'll dig deeper into some of the ideas used here in the following lectures:

- *this lecture* discusses the key concept of **exchangeability** that rationalizes statistical learning
- *this lecture* describes **likelihood ratio processes** and their role in frequentist and Bayesian statistical theories
- *this lecture* discusses the role of likelihood ratio processes in **Bayesian learning**
- *this lecture* returns to the subject of this lecture and studies whether the Captain's hunch that the (frequentist) decision rule that the Navy had ordered him to use can be expected to be better or worse than the rule sequential rule that Abraham Wald designed

CHAPTER

FIFTYEIGHT

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*

58.1 Overview

This lecture studies learning via Bayes' Law.

We touch foundations of Bayesian statistical inference invented by Bruno DeFinetti [dF37].

The relevance of DeFinetti's work for economists is presented forcefully in chapter 11 of [Kre88] by David Kreps.

An example that we study in this lecture is a key component of *this lecture* that augments the *classic* job search model of McCall [McC70] 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
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
from numba import njit, vectorize
from math import gamma
import scipy.optimize as op
from scipy.integrate import quad
import numpy as np
```

58.2 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 $W_0, W_1, \ldots$ 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 $W_0, W_1, \ldots$ is **independently and identically distributed** (IID) if in addition the marginal density of W_t is the same for all $t = 0, 1, \ldots$.

For example, let $p(W_0, W_1, \ldots)$ be the **joint density** of the sequence and let $p(W_t)$ be the **marginal density** for a particular W_t for all $t = 0, 1, \ldots$.

Then the joint density of the sequence $W_0, W_1, \ldots$ is IID if

$$p(W_0, W_1, \ldots) = p(W_0)p(W_1)\cdots$$

so that the joint density is the product of a sequence of identical marginal densities.

58.2.1 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,

$$p(W_t | W_{t-1}, \dots, W_0) = p(W_t)$$

and 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.

58.3 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
- not the F or G that nature drew once-and-for-all at $t = -1$

So our decision maker does not know which of the two distributions nature selected.

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**.

58.4 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

$$h(W_0, W_1, \ldots) \equiv \tilde{\pi}[f(W_0)f(W_1)\cdots] + (1-\tilde{\pi})[g(W_0)g(W_1)\cdots] \tag{58.1}$$

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.

But what and how?

58.5 Exchangeability

While the sequence $W_0, W_1, \ldots$ is not IID, it can be verified that it is **exchangeable**, which means that the ``re-ordered" joint distributions $h(W_0, W_1)$ and $h(W_1, W_0)$ 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 (58.1) 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.

For a Bayesian statistician, the mixing parameter $\tilde{\pi} \in (0, 1)$ has a special interpretation as a subjective **prior probability** that nature selected probability distribution F.

DeFinetti [dF37] 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.

58.6 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

$$\pi_{t+1} = \frac{\pi_t f(w_{t+1})}{\pi_t f(w_{t+1}) + (1-\pi_t) g(w_{t+1})} \tag{58.2}$$

Equation (58.2) 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

$$\mathbb{P}\{W = w\} = \sum_{a \in \{f,g\}} \mathbb{P}\{W = w \,|\, q = a\}\mathbb{P}\{q = a\}$$

58.7 More Details about Bayesian Updating

Let's stare at and rearrange Bayes' Law as represented in equation (58.2) 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 (58.2) 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

$$\frac{\pi_{t+1}}{\pi_t} = \frac{l\left(w_{t+1}\right)}{\pi_t l\left(w_{t+1}\right) + \left(1-\pi_t\right)} \begin{cases} > 1 & \text{if } l\left(w_{t+1}\right) > 1 \\ \leq 1 & \text{if } l\left(w_{t+1}\right) \leq 1 \end{cases} \tag{58.3}$$

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 (58.3) 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))
    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):
```

(continues on next page)

(continued from previous page)

```
    """
    A wrapper function that displays the updating rule of belief π,
    given the parameters which specify F and G distributions.
    """

    f = njit(lambda x: p(x, F_a, F_b))
    g = njit(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)

    W = np.arange(0.01, 0.99, 0.08)
    Π = np.arange(0.01, 0.99, 0.08)
```

(continues on next page)

(continued from previous page)

```
    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='$\pi$', 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.

58.8 Appendix

58.8.1 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 = njit(lambda x: p(x, F_a, F_b))
    g = njit(lambda x: p(x, G_a, G_b))

    @njit
    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

    @njit
    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 $\pi_t \to 0$.

58.8.2 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.

58.8.3 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
```

(continues on next page)

(continued from previous page)

```
    f = njit(lambda x: p(x, F_a, F_b))
    g = njit(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)

    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("$π_t$")
    plt.ylabel("$E[\pi_{t+1}/\pi_t]$")
    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)
```

58.9 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

FIFTYNINE

LIKELIHOOD RATIO PROCESSES AND BAYESIAN LEARNING

59.1 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
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
from numba import vectorize, njit, prange
from math import gamma
import pandas as pd

import seaborn as sns
colors = sns.color_palette()

@njit
def set_seed():
    np.random.seed(142857)
set_seed()
```

59.2 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

$$\ell(w_t) = \frac{f\left(w_t\right)}{g\left(w_t\right)}, \quad t \geq 1.$$

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

$$L\left(w^t\right) = \prod_{i=1}^t \ell(w_i),$$

where $w^t = \{w_1, \dots, w_t\}$ 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* 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 [NP33].

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
```

(continues on next page)

(continued from previous page)

```
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 = njit(lambda x: p(x, F_a, F_b))
g = njit(lambda x: p(x, G_a, G_b))
```

```
@njit
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)
```

59.3 Likelihood Ratio Process and Bayes' Law

Let π_t be a Bayesian posterior defined as

$$\pi_t = \text{Prob}(q = f | w^t)$$

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**.

Bayes' law implies that $\{\pi_t\}$ obeys the recursion

$$\pi_t = \frac{\pi_{t-1} l_t(w_t)}{\pi_{t-1} l_t(w_t) + 1 - \pi_{t-1}} \tag{59.1}$$

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 (59.1)

```
@njit
def update(π, l):
    "Update π using likelihood l"

    # Update belief
    π = π * l / (π * l + 1 - π)

    return π
```

Formula (59.1) 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

$$\pi_{t+1} = \frac{\pi_0 L\left(w^{t+1}\right)}{\pi_0 L\left(w^{t+1}\right) + 1 - \pi_0}. \tag{59.2}$$

Formula (59.2) generalizes formula (59.1).

Formula (59.2) can be regarded as a one step revision of prior probability π_0 after seeing the batch of data $\{w_i\}_{i=1}^{t+1}$.

Formula (59.2) 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 (59.2) 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.

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=f"$\pi_0$={π_seq_f[i, 0]}")

ax1.set_ylabel("$\pi_t$")
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()

for i in range(2):
    ax1.plot(range(T+1), π_seq_g[i, :], label=f"$\pi_0$={π_seq_g[i, 0]}")

ax1.set_ylabel("$\pi_t$")
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
```

```
True
```

We thus conclude that the likelihood ratio process is a key ingredient of the formula (59.2) for a Bayesian's posteior probabilty that nature has drawn history w^t as repeated draws from density g.

59.4 Behavior of posterior probability $\{\pi_t\}$ under the subjective probability distribution

We'll end this lecture by briefly studying what our Baysian 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 assumes 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.

59.4.1 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

$$\pi_t = \pi_t(w_t | \pi_{t-1}) \equiv \frac{\pi_{t-1} f(w_t)/g(w_t)}{\pi_{t-1} f(w_t)/g(w_t) + (1 - \pi_{t-1})} \tag{59.3}$$

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 $w_t, w_{t-1}, \dots, w_0$ 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 $w_{t-1}, \dots, w_0$ or equivalently conditional on the *sufficient statistic* π_{t-1} for $w_{t-1}, \dots, w_0$.

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, it 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 (59.1) or (59.3) , 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 (59.3) 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.

59.4.2 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.

```
@njit
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)
```

(continues on next page)

(continued from previous page)

```
    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('$\pi_t$')
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('$\pi_T$')
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('$\pi_T$')
```

(continues on next page)

(continued from previous page)

```
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 push 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=f'$\pi$_path, {j}-th␣
↪simulation')
    ax.plot(range(1,T+1), w_path[j,:], color=colors[i], label=f'$w$_path, {j}-th␣
↪simulation', alpha=0.3)

ax.legend(loc='upper right')
ax.set_xlabel('$t$')
ax.set_ylabel('$\pi_t$')
ax2 = ax.twinx()
ax2.set_ylabel("$w_t$")
plt.show()
```

59.5 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 as 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  0.8929  0.1071
0.2  0.7994  0.2006
0.3  0.7014  0.2986
0.4  0.5939  0.4061
0.5  0.5038  0.4962
0.6  0.3982  0.6018
0.7  0.3092  0.6908
0.8  0.1963  0.8037
0.9  0.0963  0.9037
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.

59.6 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

$$\sigma^2(\pi_t|\pi_{t-1}) = \int \left[\frac{\pi_{t-1}f(w)}{\pi_{t-1}f(w) + (1-\pi_{t-1})g(w)} - \pi_{t-1}\right]^2 \left[\pi_{t-1}f(w) + (1-\pi_{t-1})g(w)\right] dw$$

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.

```
@njit
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('$\pi_{t-1}$')
ax.set_ylabel('$\sigma^{2}(\pi_{t}\\vert \pi_{t-1})$')
plt.show()
```

The shape of the 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.

59.7 Sequels

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

SIXTY

INCORRECT MODELS

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install numpyro jax
```

60.1 Overview

This is a sequel to *this quantecon lecture*.

We discuss two ways to create 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 possible probability distributions will occur each period

The statistical setting is close but not identical to the problem studied in that quantecon lecture.

In that lecture, there were two i.i.d. 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 $w^t = \{w_s\}_{t=0}^{t}$ to form

$$\pi_t = E[\text{nature chose distribution} f | w^t], \quad t = 0, 1, 2, ...$$

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 *quantecon lecture* specification 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, naturally perpetually draws from the **mixture distribution** with c.d.f.

$$H(w) = \alpha F(w) + (1-\alpha)G(w), \quad \alpha \in (0,1)$$

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 *this quantecon lecture*.

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 turn 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 https://en.wikipedia.org/wiki/Kullback–Leibler_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
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
from numba import vectorize, njit
from math import gamma
import pandas as pd
import scipy.stats as sp
from scipy.integrate import quad

import seaborn as sns
colors = sns.color_palette()
```

(continues on next page)

(continued from previous page)

```
import numpyro
import numpyro.distributions as dist
from numpyro.infer import MCMC, NUTS

import jax.numpy as jnp
from jax import random

np.random.seed(142857)

@njit
def set_seed():
    np.random.seed(142857)
set_seed()
```

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

@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 = njit(lambda x: p(x, F_a, F_b))
g = njit(lambda x: p(x, G_a, G_b))
```

```
@njit
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)
```

60.2 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 the numpy.random.choice function 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 the quantecon lecture https://python.quantecon.org/prob_matrix.html:

- If a random variable X has c.d.f. $F(X)$, then a random variable $F^{-1}(U)$ also has c.d.f. $F(x)$, 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 https://numpy.org/doc/stable/reference/generated/numpy.searchsorted.html for 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.

```
@njit
def draw_lottery(p, N):
    "Draw from the compound lottery directly."

    draws = []
    for i in range(0, N):
        if np.random.rand()<=p:
            draws.append(np.random.beta(F_a, F_b))
        else:
            draws.append(np.random.beta(G_a, G_b))

    return np.array(draws)

def draw_lottery_MC(p, N):
    "Draw from the compound lottery using the Monte Carlo trick."

    xs = np.linspace(1e-8,1-(1e-8),10000)
    CDF = p*sp.beta.cdf(xs, F_a, F_b) + (1-p)*sp.beta.cdf(xs, G_a, G_b)

    Us = np.random.rand(N)
    draws = xs[np.searchsorted(CDF[:-1], Us)]
    return draws
```

```
# verify
N = 100000
α = 0.0

sample1 = draw_lottery(α, N)
sample2 = draw_lottery_MC(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')

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

```
# %%timeit     # compare speed
# sample1 = draw_lottery(α, N=int(1e6))
```

```
# %%timeit
# sample2 = draw_lottery_MC(α, N=int(1e6))
```

Note: With numba acceleration the first method is actually only slightly slower than the second when we generated 1,000,000 samples.

60.3 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 *this quantecon lecture*.

We'll briefly review that learning algorithm now.

Let π_t be a Bayesian posterior defined as

$$\pi_t = \text{Prob}(q = f | w^t)$$

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

$$\pi_t = \frac{\pi_{t-1} l_t(w_t)}{\pi_{t-1} l_t(w_t) + 1 - \pi_{t-1}} \tag{60.1}$$

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 (60.1)

```
@njit
def update(π, l):
    "Update π using likelihood l"

    # Update belief
    π = π * l / (π * l + 1 - π)

    return π
```

Formula (60.1) 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

$$\begin{aligned}
\frac{1}{\pi_{t+1}} &= \frac{\pi_t \ell\left(w_{t+1}\right) + \left(1 - \pi_t\right)}{\pi_t \ell\left(w_{t+1}\right)} \\
&= 1 - \frac{1}{\ell\left(w_{t+1}\right)} + \frac{1}{\ell\left(w_{t+1}\right)} \frac{1}{\pi_t}.
\end{aligned}$$

$$\Rightarrow \frac{1}{\pi_{t+1}} - 1 = \frac{1}{\ell\left(w_{t+1}\right)} \left(\frac{1}{\pi_t} - 1\right).$$

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

$$\pi_{t+1} = \frac{\pi_0 L\left(w^{t+1}\right)}{\pi_0 L\left(w^{t+1}\right) + 1 - \pi_0}. \tag{60.2}$$

Formula (60.2) generalizes formula (60.1).

Formula (60.2) can be regarded as a one step revision of prior probability π_0 after seeing the batch of data $\{w_i\}_{i=1}^{t+1}$.

60.4 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.

A submartingale or supermartingale continues to describe π_t

It raises its ugly head and causes π_t to converge either to 0 or to 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(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;α
    """

    w_s = draw_lottery(α, N*T).reshape(N, T)
    l_arr = f(w_s) / g(w_s)

    return l_arr

def plot_π_seq(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(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):
```

(continues on next page)

(continued from previous page)

```
        ax1.plot(range(T+1), π_seq_mixed[i, :], label=f"$\pi_0$={π_seq_mixed[i, 0]}")

    ax1.plot(np.nan, np.nan,  '--', color='b', label='Log likelihood ratio process')
    ax1.set_ylabel("$\pi_t$")
    ax1.set_xlabel("t")
    ax1.legend()
    ax1.set_title("when $\\alpha G + (1-\\alpha)$ F governs data")

    ax2 = ax1.twinx()
    ax2.plot(range(1, T+1), np.log(l_seq_mixed[0, :]), '--', color='b')
    ax2.set_ylabel("$log(L(w^{t}))$")

    plt.show()
```

```
plot_π_seq(α = 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(α = 0.2)
```

Evidently, α is having a big effect on the destination of π_t as $t \to +\infty$

60.5 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

$$KL_g(\alpha) = \int \log\left(\frac{g(w)}{h(w)}\right) h(w) dw$$

and

$$KL_f(\alpha) = \int \log\left(\frac{f(w)}{h(w)}\right) h(w) dw$$

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 $\to +\infty$, π_t goes to one if and only if $KL_f < KL_g$

```
@vectorize
def KL_g(α):
    "Compute the KL divergence between g and h."
    err = 1e-8                          # to avoid 0 at end points
    ws = np.linspace(err, 1-err, 10000)
    gs, fs = g(ws), f(ws)
    hs = α*fs + (1-α)*gs
```

(continues on next page)

(continued from previous page)

```
    return np.sum(np.log(gs/hs)*hs)/10000

@vectorize
def KL_f(α):
    "Compute the KL divergence between f and h."
    err = 1e-8                          # to avoid 0 at end points
    ws = np.linspace(err, 1-err, 10000)
    gs, fs = g(ws), f(ws)
    hs = α*fs + (1-α)*gs
    return np.sum(np.log(fs/hs)*hs)/10000


# compute KL using quad in Scipy
def KL_g_quad(α):
    "Compute the KL divergence between g and h using scipy.integrate."
    h = lambda x: α*f(x) + (1-α)*g(x)
    return quad(lambda x: np.log(g(x)/h(x))*h(x), 0, 1)[0]

def KL_f_quad(α):
    "Compute the KL divergence between f and h using scipy.integrate."
    h = lambda x: α*f(x) + (1-α)*g(x)
    return quad(lambda x: np.log(f(x)/h(x))*h(x), 0, 1)[0]

# vectorize
KL_g_quad_v = np.vectorize(KL_g_quad)
KL_f_quad_v = np.vectorize(KL_f_quad)


# Let us find the limit point
def π_lim(α, T=5000, π_0=0.4):
    "Find limit of π sequence."
    π_seq = np.zeros(T+1)
    π_seq[0] = π_0
    l_arr = simulate_mixed(α, T, N=1)[0]

    for t in range(T):
        π_seq[t+1] = update(π_seq[t], l_arr[t])
    return π_seq[-1]

π_lim_v = np.vectorize(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(α_arr)
KL_f_arr = KL_f(α_arr)

fig, ax = plt.subplots(1, figsize=[10, 6])

ax.plot(α_arr, KL_g_arr, label='KL(g, h)')
ax.plot(α_arr, KL_f_arr, label='KL(f, h)')
ax.set_ylabel('K-L divergence')
ax.set_xlabel(r'$\alpha$')

ax.legend(loc='upper right')
plt.show()
```

```
# # using Scipy to compute KL divergence

# α_arr = np.linspace(0, 1, 100)
# KL_g_arr = KL_g_quad_v(α_arr)
# KL_f_arr = KL_f_quad_v(α_arr)

# fig, ax = plt.subplots(1, figsize=[10, 6])

# ax.plot(α_arr, KL_g_arr, label='KL(g, h)')
# ax.plot(α_arr, KL_f_arr, label='KL(f, h)')
# ax.set_ylabel('K-L divergence')

# 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
α_arr[np.argmin(np.abs(KL_g_arr-KL_f_arr))]
```

```
0.31313131313131315
```

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<0.28)|(α_arr>0.38)]
π_lim_arr = π_lim_v(α_arr_x)
```

(continues on next page)

(continued from previous page)

```
# plot
fig, ax = plt.subplots(1, figsize=[10, 6])

ax.plot(α_arr, KL_g_arr, label='KL(g, h)')
ax.plot(α_arr, KL_f_arr, label='KL(f, h)')
ax.set_ylabel('K-L divergence')
ax.set_xlabel(r'$\alpha$')

# plot KL
ax2 = ax.twinx()
# plot limit point
ax2.scatter(α_arr_x, π_lim_arr, facecolors='none', edgecolors='tab:blue', label='$\pi
↪$ lim')
ax2.set_ylabel('π lim')

ax.legend(loc=[0.85, 0.8])
ax2.legend(loc=[0.85, 0.73])
plt.show()
```

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.

60.6 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 acknowledges 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 person starts out with a prior probabilty $\pi_0(\alpha)$ on $\alpha \in (0,1)$ where the prior has one of the forms that we deployed in *this quantecon lecture*.

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 the quantecon lecture https://python.quantecon.org/bayes_nonconj.html.

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.

```
α = 0.8

# simulate data with true α
data = draw_lottery(α, 1000)
sizes = [5, 20, 50, 200, 1000, 25000]

def model(w):
    α = numpyro.sample('α', dist.Uniform(low=0.0, high=1.0))

    y_samp = numpyro.sample('w',
        dist.Mixture(dist.Categorical(jnp.array([α, 1-α])), [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"

    kernal = NUTS(model)
    mcmc = MCMC(kernal, num_samples=5000, num_warmup=1000, progress_bar=False)

    mcmc.run(rng_key=random.PRNGKey(142857), w=jnp.array(ws))
    sample = mcmc.get_samples()
    return sample['α']
```

The following code generates the graph below that displays Bayesian posteriors for α at various history lengths.

```
fig, ax = plt.subplots(figsize=(10, 6))

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.set_title('$\pi_t(\\alpha)$ as $t$ increases')
ax.legend()
ax.set_xlabel('$\\alpha$')
plt.show()
```

```
CUDA backend failed to initialize: Found CUDA version 12010, but JAX was built␣
↪against version 12020, which is newer. The copy of CUDA that is installed must␣
↪be at least as new as the version against which JAX was built. (Set TF_CPP_MIN_
↪LOG_LEVEL=0 and rerun for more info.)
```

Evidently, the Bayesian posterior narrows in on the true value $\alpha = .8$ of the mixing parameter as the length of a history of observations grows.

60.7 Concluding Remarks

Our type 1 person 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 interprests 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.

CHAPTER

SIXTYONE

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 Frequentist's t*

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install interpolation
```

```
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
from numba import njit, prange, float64, int64
from numba.experimental import jitclass
from interpolation import interp
from math import gamma
from scipy.optimize import minimize
```

```
/opt/conda/envs/quantecon/lib/python3.11/site-packages/numba/core/decorators.
↪py:262: NumbaDeprecationWarning: numba.generated_jit is deprecated. Please see␣
↪the documentation at: https://numba.readthedocs.io/en/stable/reference/
↪deprecation.html#deprecation-of-generated-jit for more information and advice on␣
↪a suitable replacement.
  warnings.warn(msg, NumbaDeprecationWarning)
```

61.1 Overview

This lecture follows up on ideas presented in the following lectures:

- *A Problem that Stumped Milton Friedman*
- *Exchangeability and Bayesian Updating*
- *Likelihood Ratio Processes*

In *A Problem that Stumped Milton Friedman* we described a problem that a Navy Captain presented to Milton Friedman during World War II.

The Navy had instructed the Captain to use a decision rule for quality control that the Captain suspected could be dominated by a better rule.

(The Navy had ordered the Captain to use an instance of a **frequentist decision rule**.)

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.

One of the members of the group, the great mathematician 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 **frequentist** and **Bayesian** decision rules for an instance of the Navy Captain's decision problem.

The simulations validate the Navy Captain's hunch that there is a better rule than the one the Navy had ordered him to use.

61.2 Setup

To formalize the problem of the Navy Captain whose questions posed the problem that Milton Friedman and Allan Wallis handed over to Abraham Wald, we consider a setting with the following parts.

- Each period a decision maker draws a non-negative random variable Z from a probability distribution that he does not completely understand. 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 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 the distribution chosen by nature.

The decision maker wants to decide which distribution actually governs Z and is worried by two types of errors and the losses that they impose on him.

- a loss $\bar{L}_1$ from a **type I error** that occurs when he decides that $f = f_1$ when actually $f = f_0$
- a loss $\bar{L}_0$ from a **type II error** that occurs when 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 Problem that Stumped Milton Friedman* except that we increase both $\bar{L}_0$ and $\bar{L}_1$ from 25 to 100 to encourage the frequentist Navy Captain 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.

```
@njit
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):
```

(continues on next page)

(continued from previous page)

```
        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)

        return π_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

61.3 Frequentist Decision Rule

The Navy told the Captain to use a frequentist decision rule.

In particular, it gave him a decision rule that the Navy had designed by using frequentist statistical theory to minimize an expected loss function.

That decision rule is characterized by a sample size t and a cutoff d associated with 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 $\left\{z_i\right\}_{i=0}^{t}$.

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

To understand how that rule was engineered, 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

$$\bar{V}_{fre}\left(t,d\right) = ct + \pi^* PFA \times \bar{L}_1 + \left(1-\pi^*\right)\left(1-PD\right) \times \bar{L}_0 \tag{61.1}$$

$$\text{where} \quad \begin{aligned} PFA &= \Pr\left\{L\left(z^t\right) < d \mid q = f_0\right\} \\ PD &= \Pr\left\{L\left(z^t\right) < d \mid q = f_1\right\} \end{aligned}$$

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 $\left(PFA, PD\right)$ lie on a **receiver operating characteristic curve** and can be uniquely pinned down by choosing d.

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.

```
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")
plt.show()
```

Our frequentist minimizes the expected total loss presented in equation (61.1) 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.

```
@njit
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='$\min_{d} \overline{V}_{fre}(t,d)$')
plt.xlabel('t')
plt.title('$\pi^*=0.5$')
plt.legend()
plt.show()
```

```
t_optimal = np.argmin(V_fre_arr) + 1
```

```
msg = f"The above graph indicates that minimizing over t tells the frequentist to␣
↪draw {t_optimal} observations and then decide."
print(msg)
```

```
The above graph indicates that minimizing over t tells the frequentist to draw 9␣
↪observations and then decide.
```

(continues on next page)

(continued from previous page)

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('$\pi^*$')
plt.title('$\overline{V}_{fre}$')

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('$\pi^*$')
```

(continues on next page)

(continued from previous page)

```
axs[0].set_title('optimal sample size given $\pi^*$')

axs[1].plot(π_star_arr, PFA_optimal_arr, label='$PFA^*(\pi^*)$')
axs[1].plot(π_star_arr, PD_optimal_arr, label='$PD^*(\pi^*)$')
axs[1].set_xlabel('$\pi^*$')
axs[1].legend()
axs[1].set_title('optimal PFA and PD given $\pi^*$')

plt.show()
```

61.4 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.

We presented a Bayesian procedure that instructed the Captain to makes decisions by comparing his current Bayesian posterior probability π with two cutoff probabilities called α and β.

To proceed, we borrow some Python code from the quantecon lecture *A Problem that Stumped Milton Friedman* that computes α and β.

```
@njit(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: interp(π_grid, h, p)

    for i in prange(len(π_grid)):
        π = π_grid[i]
```

(continues on next page)

(continued from previous page)

```
        # 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

        h_new[i] = c + integral

    return h_new
```

```
@njit
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)
```

```
@njit
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
```

(continues on next page)

(continued from previous page)

```
    # 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)
    β = π_grid[np.searchsorted(
                              payoff_f1 - np.minimum(h, payoff_f0),
                              1e-10)
               - 1]
    α = π_grid[np.searchsorted(
                              np.minimum(h, payoff_f1) - payoff_f0,
                              1e-10)
               - 1]

    return (β, α)

β, α = 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"$\beta$", xy=(β + 0.01, 0.5), fontsize=14)
ax.annotate(r"$\alpha$", xy=(α + 0.01, 0.5), fontsize=14)

plt.vlines(β, 0, β * wf.L0, linestyle="--")
plt.vlines(α, 0, (1 - α) * wf.L1, linestyle="--")

ax.set(xlim=(0, 1), ylim=(0, 0.5 * max(wf.L0, wf.L1)), ylabel="cost",
       xlabel="$\pi$", 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 probabilities α and β.

The Bayesian decision rule is:

- accept H_0 if $\pi \geq \alpha$
- accept H_1 if $\pi \leq \beta$
- delay deciding and draw another z if $\beta \leq \pi \leq \alpha$

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,

$$V^0(\pi) = \begin{cases} 0 & \text{if } \alpha \leq \pi, \\ c + EV^0(\pi') & \text{if } \beta \leq \pi < \alpha, \\ \bar{L}_1 & \text{if } \pi < \beta. \end{cases}$$

2. under f_1

$$V^1(\pi) = \begin{cases} \bar{L}_0 & \text{if } \alpha \leq \pi, \\ c + EV^1(\pi') & \text{if } \beta \leq \pi < \alpha, \\ 0 & \text{if } \pi < \beta. \end{cases}$$

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.

```
@njit(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 < β] = wf.L1
    else:
        z_arr = wf.z1
        V[wf.π_grid >= α] = wf.L0

    V_old = np.empty_like(V)

    while True:
        V_old[:] = V[:]
        V[(β <= wf.π_grid) & (wf.π_grid < α)] = 0

        for i in prange(len(wf.π_grid)):
            π = wf.π_grid[i]

            if π >= α or π < β:
                continue

            for j in prange(len(z_arr)):
                π_next = wf.κ(z_arr[j], π)
                V[i] += wf.c + interp(wf.π_grid, V_old, π_next)

            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(β, 0, wf.L0, linestyle='--')
plt.text(β+0.01, wf.L0/2, 'β')
plt.vlines(α, 0, wf.L0, linestyle='--')
plt.text(α+0.01, wf.L0/2, 'α')
plt.xlabel('$\pi$')
plt.title('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 9 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,
                           '${\pi_0^*}=$'+f'{π_optimal:0.2f}')
    axs[row_i, col_i].set_xlabel('$\pi$')
    axs[row_i, col_i].set_ylabel('$\overline{V}_{baye}(\pi)$')
    axs[row_i, col_i].set_title('$\pi^*=$' + f'{π_star}')
```

(continues on next page)

(continued from previous page)

```
fig.suptitle('$\overline{V}_{baye}(\pi)=\pi^*V^0(\pi) + (1-\pi^*)V^1(\pi)$',␣
↪fontsize=16)
plt.show()
```

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('$\pi^*$')
axs[0].set_title('$\overline{V}_{baye}$')

axs[1].plot(π_star_arr, π_optimal_arr, label='optimal prior')
```

(continues on next page)

(continued from previous page)

```
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('$\pi^*$')
axs[1].set_title('optimal prior given $\pi^*$')
axs[1].legend()

plt.show()
```

61.5 Was the Navy Captain's Hunch Correct?

We now compare average (i.e., frequentist) losses obtained by the frequentist 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 + π_star * V1
V_baye_bar = V_baye.min()
```

```
plt.plot(range(T), V_fre_arr, label='$\min_{d} \overline{V}_{fre}(t,d)$')
plt.plot([0, T], [V_baye_bar, V_baye_bar], label='$\overline{V}_{baye}$')
plt.xlabel('t')
plt.title('$\pi^*=0.5$')
plt.legend()
plt.show()
```

Evidently, there is no sample size t at which the frequentist 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='$\overline{V}_{fre}$')
axs[0].plot(π_star_arr, V_baye_bar_arr, label='$\overline{V}_{baye}$')
axs[0].legend()
axs[0].set_xlabel('$\pi^*$')

axs[1].plot(π_star_arr, V_fre_bar_arr - V_baye_bar_arr, label='$diff$')
axs[1].legend()
axs[1].set_xlabel('$\pi^*$')

plt.show()
```

The right panel of the above graph plots the difference $\bar{V}_{fre} - \bar{V}_{Bayes}$.

It is always positive.

61.6 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 decision rule sets a sample size `t_optimal` **ex ante**.

For our parameter settings, we can compute its value:

```
t_optimal
```

```
9
```

For convenience, let's define `t_idx` as the Python array index corresponding to `t_optimal` sample size.

```
t_idx = t_optimal - 1
```

61.7 Distribution of Bayesian Decision Rule's Time to Decide

By using simulations, we compute the frequency distribution of time to deciding 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 $q = f_1$.

```
@njit(parallel=True)
def check_results(L_arr, α, β, 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] < β) or (π_arr[i, t] > α):
                time_arr[i] = t + 1
                correctness[i] = (flag == 0 and π_arr[i, t] > α) or (flag == 1 and π_
↪arr[i, t] < β)
                break

    return time_arr, correctness
```

```
time_arr0, correctness0 = check_results(L0_arr, α, β, 0, π_star)
time_arr1, correctness1 = check_results(L1_arr, α, β, 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 two decision rules.

To begin, let's look at simulations of the Bayesian's beliefs over time.

We can easily compute the 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')
```

(continues on next page)

(continued from previous page)

```
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('$E(\pi_t)$ 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('var($\pi_t$)')
axs[1].set_title('Variance of beliefs after drawing t observations')
axs[1].legend()

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 size 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()
```

61.8 Probability of Making Correct Decision

Now we use simulations to compute the fraction of samples in which the Bayesian and the frequentist decision rules decide correctly.

For the frequentist 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],
        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()
```

61.9 Distribution of Likelihood Ratios at Frequentist'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 α and β.

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.

```
Lα = (1 - π_star) *  α / (π_star - π_star * α)
Lβ = (1 - π_star) *  β / (π_star - π_star * β)
```

```
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(Lβ), 0, max(n0.max(), n1.max()), linestyle='--', color='r', label=
 ↪'log($L_β$)')
plt.vlines(np.log(Lα), 0, max(n0.max(), n1.max()), linestyle='--', color='b', label=
 ↪'log($L_α$)')
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(Lβ), 0, max(n0.max(), n1.max()), linestyle='--', color='r', label=
 ↪'log($L_β$)')
plt.vlines(np.log(Lα), 0, max(n0.max(), n1.max()), linestyle='--', color='b', label=
 ↪'log($L_α$)')
plt.legend()

plt.xlabel('log(L)')
plt.ylabel('n')
plt.title('Uncond. distribution of log likelihood ratio at frequentist  t')

plt.show()
```

Part IX

LQ Control

CHAPTER

SIXTYTWO

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
```

62.1 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 this lecture and this lecture).

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 *this lecture* and *this lecture*)

For additional reading on LQ control, see, for example,

- [LS18], chapter 5
- [HS08], chapter 4
- [HLL96], 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
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
from quantecon import LQ
```

62.2 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.

62.2.1 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

$$x_{t+1} = Ax_t + Bu_t + Cw_{t+1}, \qquad t = 0, 1, 2, \ldots \tag{62.1}$$

Here

- u_t is a "control" vector, incorporating choices available to a decision-maker confronting the current state x_t
- $\{w_t\}$ is an uncorrelated zero mean shock process satisfying $\mathbb{E} w_t w_t' = I$, 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 (62.1), 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

$$a_{t+1} = (1+r)a_t - u_t - \bar{c} + \sigma w_{t+1} + \mu \tag{62.2}$$

How can we write this new system in the form of equation (62.1)?

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

$$\begin{pmatrix} a_{t+1} \\ 1 \end{pmatrix} = \begin{pmatrix} 1+r & -\bar{c}+\mu \\ 0 & 1 \end{pmatrix} \begin{pmatrix} a_t \\ 1 \end{pmatrix} + \begin{pmatrix} -1 \\ 0 \end{pmatrix} u_t + \begin{pmatrix} \sigma \\ 0 \end{pmatrix} w_{t+1} \tag{62.3}$$

then the first row is equivalent to (62.2).

Moreover, the model is now linear and can be written in the form of (62.1) by setting

$$x_t := \begin{pmatrix} a_t \\ 1 \end{pmatrix}, \quad A := \begin{pmatrix} 1+r & -\bar{c}+\mu \\ 0 & 1 \end{pmatrix}, \quad B := \begin{pmatrix} -1 \\ 0 \end{pmatrix}, \quad C := \begin{pmatrix} \sigma \\ 0 \end{pmatrix} \tag{62.4}$$

In effect, we've bought ourselves linearity by adding another state.

62.2.2 Preferences

In the LQ model, the aim is to minimize flow of losses, where time-t loss is given by the quadratic expression

$$x_t' R x_t + u_t' Q u_t \tag{62.5}$$

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 [HS08] for details.

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

$$x_t' I x_t + u_t' I u_t = \|x_t\|^2 + \|u_t\|^2$$

Thus, for both the state and the control, loss is measured as squared distance from the origin.

(In fact, the general case (62.5) 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

$$x_t' R x_t + u_t' Q u_t = u_t^2 = (c_t - \bar{c})^2$$

Under this specification, the household's current loss is the squared deviation of consumption from the ideal level $\bar{c}$.

62.3 Optimality – Finite Horizon

Let's now be precise about the optimization problem we wish to consider, and look at how to solve it.

62.3.1 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

$$\mathbb{E} \left\{ \sum_{t=0}^{T-1} \beta^t (x_t' R x_t + u_t' Q u_t) + \beta^T x_T' R_f x_T \right\} \tag{62.6}$$

subject to the law of motion (62.1) 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 $x' R_f x$ 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 $w_1, \dots, w_T$.

62.3.2 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.

62.3.3 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 *this lecture*.

For reasons that will soon become clear, we first introduce the notation $J_T(x) = x' R_f x$.

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

$$\min_u \{x'_{T-1} R x_{T-1} + u' Q u + \beta \, \mathbb{E} J_T(A x_{T-1} + B u + C w_T)\}$$

At this stage, it is convenient to define the function

$$J_{T-1}(x) = \min_u \{x' R x + u' Q u + \beta \, \mathbb{E} J_T(A x + B u + C w_T)\} \tag{62.7}$$

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 $J_T(x) = x' R_f x$ 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

$$\min_u \{x'_{T-2} R x_{T-2} + u' Q u + \beta \, \mathbb{E} J_{T-1}(A x_{T-2} + B u + C w_{T-1})\}$$

Letting

$$J_{T-2}(x) = \min_u \{x' R x + u' Q u + \beta \, \mathbb{E} J_{T-1}(A x + B u + C w_{T-1})\}$$

the pattern for backward induction is now clear.

In particular, we define a sequence of value functions $\{J_0, \ldots, J_T\}$ via

$$J_{t-1}(x) = \min_u \{x' R x + u' Q u + \beta \, \mathbb{E} J_t(A x + B u + C w_t)\} \quad \text{and} \quad J_T(x) = x' R_f x$$

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 $J_t(x) = x' P_t x + d_t$ 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, (62.7) becomes

$$J_{T-1}(x) = \min_u \{x' R x + u' Q u + \beta \, \mathbb{E} (A x + B u + C w_T)' P_T (A x + B u + C w_T)\} \tag{62.8}$$

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

$$u = -(Q + \beta B' P_T B)^{-1} \beta B' P_T A x \tag{62.9}$$

Plugging this back into (62.8) and rearranging yields

$$J_{T-1}(x) = x' P_{T-1} x + d_{T-1}$$

where

$$P_{T-1} = R - \beta^2 A' P_T B (Q + \beta B' P_T B)^{-1} B' P_T A + \beta A' P_T A \tag{62.10}$$

and

$$d_{T-1} := \beta \, \text{trace}(C' P_T C) \tag{62.11}$$

(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 $J_t(x) = x' P_t x + d_t$ as claimed, where $\{P_t\}$ and $\{d_t\}$ satisfy the recursions

$$P_{t-1} = R - \beta^2 A' P_t B (Q + \beta B' P_t B)^{-1} B' P_t A + \beta A' P_t A \quad \text{with} \quad P_T = R_f \tag{62.12}$$

and

$$d_{t-1} = \beta (d_t + \text{trace}(C' P_t C)) \quad \text{with} \quad d_T = 0 \tag{62.13}$$

Recalling (62.9), the minimizers from these backward steps are

$$u_t = -F_t x_t \quad \text{where} \quad F_t := (Q + \beta B' P_{t+1} B)^{-1} \beta B' P_{t+1} A \tag{62.14}$$

These are the linear optimal control policies we *discussed above*.

In particular, the sequence of controls given by (62.14) and (62.1) solves our finite horizon LQ problem.

Rephrasing this more precisely, the sequence $u_0, \dots, u_{T-1}$ given by

$$u_t = -F_t x_t \quad \text{with} \quad x_{t+1} = (A - BF_t) x_t + C w_{t+1} \tag{62.15}$$

for $t = 0, \dots, T-1$ attains the minimum of (62.6) subject to our constraints.

62.4 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 (62.12), (62.13) and (62.14)
  - `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

62.4.1 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, [Fri56] or [MB54].)

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

$$\mathbb{E}\left\{\sum_{t=0}^{T-1}\beta^t(c_t-\bar{c})^2+\beta^T q a_T^2\right\} \tag{62.16}$$

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 (62.2).

We saw how this constraint could be manipulated into the LQ formulation $x_{t+1}=Ax_t+Bu_t+Cw_{t+1}$ by setting $x_t=(a_t\ 1)'$ and using the definitions in (62.4).

To match with this state and control, the objective function (62.16) can be written in the form of (62.6) by choosing

$$Q:=1,\quad R:=\begin{pmatrix}0&0\\0&0\end{pmatrix},\quad\text{and}\quad R_f:=\begin{pmatrix}q&0\\0&0\end{pmatrix}$$

Now that the problem is expressed in LQ form, we can proceed to the solution by applying (62.12) and (62.14).

After generating shocks $w_1,\dots,w_T$, the dynamics for assets and consumption can be simulated via (62.15).

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.

```
# Model parameters
r = 0.05
β = 1/(1 + r)
T = 45
c_bar = 2
σ = 0.25
```

(continues on next page)

(continued from previous page)

```
μ = 1
q = 1e6

# Formulate as an LQ problem
Q = 1
R = np.zeros((2, 2))
Rf = np.zeros((2, 2))
Rf[0, 0] = q
A = [[1 + r, -c_bar + μ],
    [0,              1]]
B = [[-1],
    [ 0]]
C = [[σ],
    [0]]

# 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}

axes[0].plot(list(range(1, T+1)), income, 'g-', label="non-financial income",
            **p_args)
axes[0].plot(list(range(T)), c, 'k-', label="consumption", **p_args)

axes[1].plot(list(range(1, T+1)), np.cumsum(income - μ), 'r-',
            label="cumulative unanticipated income", **p_args)
axes[1].plot(list(range(T+1)), assets, 'b-', label="assets", **p_args)
axes[1].plot(list(range(T)), np.zeros(T), 'k-')

for ax in axes:
    ax.grid()
    ax.set_xlabel('Time')
    ax.legend(ncol=2, **legend_args)

plt.show()
```

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.

```
# Compute solutions and simulate
lq = LQ(Q, R, A, B, C, beta=0.96, 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}

axes[0].plot(list(range(1, T+1)), income, 'g-', label="non-financial income",
             **p_args)
axes[0].plot(list(range(T)), c, 'k-', label="consumption", **p_args)

axes[1].plot(list(range(1, T+1)), np.cumsum(income - μ), 'r-',
             label="cumulative unanticipated income", **p_args)
axes[1].plot(list(range(T+1)), assets, 'b-', label="assets", **p_args)
axes[1].plot(list(range(T)), np.zeros(T), 'k-')

for ax in axes:
    ax.grid()
    ax.set_xlabel('Time')
    ax.legend(ncol=2, **legend_args)

plt.show()
```

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.

62.5 Extensions and Comments

Let's now consider a number of standard extensions to the LQ problem treated above.

62.5.1 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 [HS13], section 2.4.

62.5.2 Adding a Cross-Product Term

In some LQ problems, preferences include a cross-product term $u_t' N x_t$, so that the objective function becomes

$$\mathbb{E}\left\{\sum_{t=0}^{T-1} \beta^t (x_t' R x_t + u_t' Q u_t + 2 u_t' N x_t) + \beta^T x_T' R_f x_T\right\} \tag{62.17}$$

Our results extend to this case in a straightforward way.

The sequence $\{P_t\}$ from (62.12) becomes

$$P_{t-1} = R - (\beta B' P_t A + N)'(Q + \beta B' P_t B)^{-1}(\beta B' P_t A + N) + \beta A' P_t A \quad \text{with} \quad P_T = R_f \tag{62.18}$$

The policies in (62.14) are modified to

$$u_t = -F_t x_t \quad \text{where} \quad F_t := (Q + \beta B' P_{t+1} B)^{-1}(\beta B' P_{t+1} A + N) \tag{62.19}$$

The sequence $\{d_t\}$ is unchanged from (62.13).

We leave interested readers to confirm these results (the calculations are long but not overly difficult).

62.5.3 Infinite Horizon

Finally, we consider the infinite horizon case, with *cross-product term*, unchanged dynamics and objective function given by

$$\mathbb{E}\left\{\sum_{t=0}^{\infty} \beta^t (x_t' R x_t + u_t' Q u_t + 2 u_t' N x_t)\right\} \tag{62.20}$$

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 $u_t = -F x_t$ 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.

$$P = R - (\beta B' P A + N)'(Q + \beta B' P B)^{-1}(\beta B' P A + N) + \beta A' P A \tag{62.21}$$

Equation (62.21) is also called the *LQ Bellman equation*, and the map that sends a given P into the right-hand side of (62.21) is called the *LQ Bellman operator*.

The stationary optimal policy for this model is

$$u = -Fx \quad \text{where} \quad F = (Q + \beta B'PB)^{-1}(\beta B'PA + N) \tag{62.22}$$

The sequence $\{d_t\}$ from (62.13) is replaced by the constant value

$$d := \text{trace}(C'PC)\frac{\beta}{1-\beta} \tag{62.23}$$

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*.

62.5.4 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 (62.22) or (62.19).

It follows that we can ignore uncertainty when solving for optimal behavior, and plug it back in when examining optimal state dynamics.

62.6 Further Applications

62.6.1 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

$$\mathbb{E}\left\{\sum_{t=0}^{T-1}\beta^t(c_t - \bar{c})^2 + \beta^T q a_T^2\right\} \tag{62.24}$$

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

$$a_{t+1} = (1+r)a_t - u_t - \bar{c} + m_1 t + m_2 t^2 + \sigma w_{t+1} \tag{62.25}$$

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

$$x_t := \begin{pmatrix} a_t \\ 1 \\ t \\ t^2 \end{pmatrix}, \quad A := \begin{pmatrix} 1+r & -\bar{c} & m_1 & m_2 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 2 & 1 \end{pmatrix}, \quad B := \begin{pmatrix} -1 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \quad C := \begin{pmatrix} \sigma \\ 0 \\ 0 \\ 0 \end{pmatrix} \tag{62.26}$$

If you expand the expression $x_{t+1} = Ax_t + Bu_t + Cw_{t+1}$ using this specification, you will find that assets follow (62.25) as desired and that the other state variables also update appropriately.

To implement preference specification (62.24) we take

$$Q := 1, \quad R := \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \quad \text{and} \quad R_f := \begin{pmatrix} q & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \tag{62.27}$$

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 62.7.1 gives the full set of parameters used here and asks you to replicate the figure.

62.6.2 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

$$y_t = \begin{cases} p(t) + \sigma w_{t+1} & \text{if } t \leq K \\ s & \text{otherwise} \end{cases} \tag{62.28}$$

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 (62.16).

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 (62.28) 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 62.7.2, 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.

62.6.3 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

$$\mathbb{E}\left\{\sum_{t=0}^{\infty} \beta^t \pi_t\right\} \quad \text{where} \quad \pi_t := p_t q_t - c q_t - \gamma(q_{t+1} - q_t)^2 \tag{62.29}$$

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 $x_t = (\bar{q}_t \;\; q_t \;\; 1)'$, 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

$\bar{q}_t$
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 (62.29), 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 (62.29) 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 (62.29), 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

$$\min \mathbb{E} \sum_{t=0}^{\infty} \beta^t \left\{ a_1 (q_t - \bar{q}_t)^2 + \gamma u_t^2 \right\} \tag{62.30}$$

It's now relatively straightforward to find R and Q such that (62.30) can be written as (62.20).

Furthermore, the matrices A, B and C from (62.1) can be found by writing down the dynamics of each element of the state.

Exercise 62.7.3 asks you to complete this process, and reproduce the preceding figures.

62.7 Exercises

Exercise 62.7.1

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 to Exercise 62.7.1

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
```

(continues on next page)

(continued from previous page)

```
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 62.7.2

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 (62.26).

- 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 (62.27).

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 to Exercise 62.7.2

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],
```

(continues on next page)

(continued from previous page)

```
     [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 62.7.3

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 to Exercise 62.7.3

The first task is to find the matrices A, B, C, Q, R that define the LQ problem.

Recall that $x_t = (\bar{q}_t \;\; q_t \;\; 1)'$, 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)
```

(continues on next page)

(continued from previous page)

```
s = f'dynamics with $\gamma = {γ}$'
ax.text(max(time) * 0.6, 1 * q_bar.max(), s, fontsize=14)
plt.show()
```

CHAPTER

SIXTYTHREE

LAGRANGIAN FOR LQ CONTROL

```
!pip install quantecon
```

```
import numpy as np
from quantecon import LQ
from scipy.linalg import schur
```

63.1 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.

63.2 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

$$-x'Px = \max_u\{-x'Rx - u'Qu - (Ax + Bu)'P(Ax + Bu)\} \tag{63.1}$$

The first-order necessary conditions for the maximum problem on the right side of equation (63.1) 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 (63.1) and rearranging gives

$$P = R + A'PA - A'PB(Q + B'PB)^{-1}B'PA. \tag{63.2}$$

Equation (63.2) is called an **algebraic matrix Riccati** equation.

There are multiple solutions of equation (63.2).

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

$$\frac{\partial V(x)}{\partial x} = -2Px \tag{63.3}$$

We shall use fact (63.3) later.

63.3 Lagrangian

For the undiscounted optimal linear regulator problem, form the Lagrangian

$$L = -\sum_{t=0}^{\infty}\left\{x_t' R x_t + u_t' Q u_t + 2\mu_{t+1}'[Ax_t + Bu_t - x_{t+1}]\right\} \tag{63.4}$$

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 (63.3).)

First-order conditions for maximization with respect to $\{u_t, x_{t+1}\}_{t=0}^{\infty}$ are

$$\begin{aligned} 2Qu_t + 2B'\mu_{t+1} &= 0 \ , \ t \geq 0 \\ \mu_t &= Rx_t + A'\mu_{t+1} \ , \ t \geq 1. \end{aligned} \tag{63.5}$$

Define μ_0 to be a vector of shadow prices of x_0 and apply an envelope condition to (63.4) to deduce that

$$\mu_0 = Rx_0 + A'\mu_1,$$

which is a time $t = 0$ counterpart to the second equation of system (63.5).

An important fact is that

$$\mu_{t+1} = Px_{t+1} \tag{63.6}$$

where P is a positive define matrix that solves the algebraic Riccati equation (63.2).

Thus, from equations (63.3) and (63.6), $-2\mu_t$ 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 (63.5) 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 (63.5) into the form

$$L \begin{pmatrix} x_{t+1} \\ \mu_{t+1} \end{pmatrix} = N \begin{pmatrix} x_t \\ \mu_t \end{pmatrix} \ , \ t \geq 0, \tag{63.7}$$

where

$$L = \begin{pmatrix} I & BQ^{-1}B' \\ 0 & A' \end{pmatrix}, \quad N = \begin{pmatrix} A & 0 \\ -R & I \end{pmatrix}.$$

When L is of full rank (i.e., when A is of full rank), we can write system (63.7) as

$$\begin{pmatrix} x_{t+1} \\ \mu_{t+1} \end{pmatrix} = M \begin{pmatrix} x_t \\ \mu_t \end{pmatrix} \tag{63.8}$$

where

$$M \equiv L^{-1}N = \begin{pmatrix} A + BQ^{-1}B'A'^{-1}R & -BQ^{-1}B'A'^{-1} \\ -A'^{-1}R & A'^{-1} \end{pmatrix}. \tag{63.9}$$

63.4 State-Costate Dynamics

We seek to solve the difference equation system (63.8) 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

$$-\sum_{t=0}^{\infty}[x_t' R x_t + u_t' Q u_t],$$

which requires that $x_t' R x_t$ converge to zero as $t \to +\infty$.

63.5 Reciprocal Pairs Property

To proceed, we study properties of the $(2n \times 2n)$ matrix M defined in (63.9).

It helps to introduce a $(2n \times 2n)$ matrix

$$J = \begin{pmatrix} 0 & -I_n \\ I_n & 0 \end{pmatrix}.$$

The rank of J is $2n$.

Definition: A matrix M is called **symplectic** if

$$MJM' = J. \tag{63.10}$$

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 (63.9) is symplectic.

It follows from equation (63.10) and from the fact $J^{-1} = J' = -J$ that for any symplectic matrix M,

$$M' = J^{-1} M^{-1} J. \tag{63.11}$$

Equation (63.11) 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 (63.11) that the eigenvalues of M occur in reciprocal pairs: if λ is an eigenvalue of M, so is λ^{-1}.

Write equation (63.8) as

$$y_{t+1} = M y_t \tag{63.12}$$

where $y_t = \begin{pmatrix} x_t \\ \mu_t \end{pmatrix}$.

Consider a **triangularization** of M

$$V^{-1}MV = \begin{pmatrix} W_{11} & W_{12} \\ 0 & W_{22} \end{pmatrix} \tag{63.13}$$

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

63.6 Schur decomposition

The **Schur decomposition** and the **eigenvalue decomposition** are two decompositions of the form (63.13).

Write equation (63.12) as

$$y_{t+1} = VWV^{-1}y_t. \tag{63.14}$$

A solution of equation (63.14) for arbitrary initial condition y_0 is evidently

$$y_t = V \begin{bmatrix} W_{11}^t & W_{12,t} \\ 0 & W_{22}^t \end{bmatrix} V^{-1}y_0 \tag{63.15}$$

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 (63.15) as

$$\begin{pmatrix} y_{1t}^* \\ y_{2t}^* \end{pmatrix} = \begin{bmatrix} W_{11}^t & W_{12,t} \\ 0 & W_{22}^t \end{bmatrix} \begin{pmatrix} y_{10}^* \\ y_{20}^* \end{pmatrix}$$

where $y_t^* = V^{-1}y_t$, and in particular where

$$y_{2t}^* = V^{21}x_t + V^{22}\mu_t, \tag{63.16}$$

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 (63.16) 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

$$(V^{21}\ V^{22}) \quad \begin{pmatrix} V_{11} \\ V_{21} \end{pmatrix} = 0$$

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

$$P = V_{21} V_{11}^{-1}. \tag{63.17}$$

Remarkably, formula (63.17) provides us with a computationally efficient way of computing the positive definite matrix P that solves the algebraic Riccati equation (63.2) that emerges from dynamic programming.

This same method can be applied to compute the solution of any system of the form (63.8) 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 [LS18], ch 12.

63.7 Application

Here we demonstrate the computation with an example which is the deterministic version of an example borrowed from this quantecon lecture.

```
# Model parameters
r = 0.05
c_bar = 2
μ = 1

# Formulate as an LQ problem
Q = np.array([[1]])
```

(continues on next page)

(continued from previous page)

```
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)
```

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, k = lq.n, lq.k

    # 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
```

```
L, N, M = construct_LNM(lq.A, lq.B, lq.Q, lq.R)
```

```
M
```

```
array([[ 1.05      , -1.        , -0.95238095,  0.        ],
       [ 0.        ,  1.        ,  0.        ,  0.        ],
       [ 0.        ,  0.        ,  0.95238095,  0.        ],
       [ 0.        ,  0.        ,  0.95238095,  1.        ]])
```

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([[-1.32169408e-17,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00],
       [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
```

(continues on next page)

(continued from previous page)

```
          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]])
```

We can compute the eigenvalues of M using `np.linalg.eigvals`, arranged in ascending order.

```
eigvals = sorted(np.linalg.eigvals(M))
eigvals
```

```
[0.9523809523809523, 1.0, 1.0, 1.05]
```

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.

```
stable_eigvals = eigvals[:n]

def sort_fun(x):
    "Sort the eigenvalues with modules smaller than 1 to the top-left."

    if x in stable_eigvals:
        stable_eigvals.pop(stable_eigvals.index(x))
        return True
    else:
        return False

W, V, _ = schur(M, sort=sort_fun)
```

```
W
```

```
array([[ 1.        , -0.02316402, -1.00085948, -0.95000594],
       [ 0.        ,  0.95238095, -0.00237501, -0.95325452],
       [ 0.        ,  0.        ,  1.05      ,  0.02432222],
       [ 0.        ,  0.        ,  0.        ,  1.        ]])
```

```
V
```

```
array([[ 0.99875234,  0.00121459, -0.04992284,  0.        ],
       [ 0.04993762, -0.02429188,  0.99845688,  0.        ],
       [ 0.        ,  0.04992284,  0.00121459,  0.99875234],
       [ 0.        , -0.99845688, -0.02429188,  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([1.        , 0.95238095])
```

```
# W22
np.diag(W[n:, n:])
```

```
array([1.05, 1.  ])
```

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
    stable_eigvals = list(sorted(np.linalg.eigvals(M))[:n])

    def sort_fun(x):
        "Sort the eigenvalues with modules smaller than 1 to the top-left."

        if x in stable_eigvals:
            stable_eigvals.pop(stable_eigvals.index(x))
            return True
        else:
            return False

    W, V, _ = schur(M, sort=sort_fun)
    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}
    P = V[n:, :n] @ np.linalg.inv(V[:n, :n])

    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: [1.         0.95238095]
    W22: [1.05 1.  ]
```

```
array([[ 0.1025, -2.05  ],
       [-2.05  , 41.    ]])
```

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([[ 0.1025, -2.05  ],
        [-2.05  , 41.01  ]]),
 array([[-0.09761905,  1.95238095]]),
 0)
```

Using a Schur decomposition is much more efficient.

```
%%timeit
stationary_P(lq, verbose=False)
```

```
159 µs ± 1.14 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

```
%%timeit
lq.stationary_values()
```

```
2.53 ms ± 3.26 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

63.8 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]])
```

63.9 Discounted Problems

63.9.1 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.

```
β = 1 / (1 + r)
lq.beta = β
```

```
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.  ]]),
 0.0)
```

63.9.2 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

$$L = -\sum_{t=0}^{\infty} \beta^t \Big\{ x_t' R x_t + u_t' Q u_t + 2\beta \mu_{t+1}' [A x_t + B u_t - x_{t+1}] \Big\} \tag{63.18}$$

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

$$\begin{aligned} 2Qu_t + 2\beta B' \mu_{t+1} &= 0 \ , \ t \geq 0 \\ \mu_t &= R x_t + \beta A' \mu_{t+1} \ , \ t \geq 1. \end{aligned} \tag{63.19}$$

Define $2\mu_0$ to be the vector of shadow prices of x_0 and apply an envelope condition to (63.18) to deduce that

$$\mu_0 = R x_0 + \beta A' \mu_1,$$

which is a time $t = 0$ counterpart to the second equation of system (63.19).

Proceeding as we did above with the undiscounted system (63.5), we can rearrange the first-order conditions into the system

$$\begin{bmatrix} I & \beta B Q^{-1} B' \\ 0 & \beta A' \end{bmatrix} \begin{bmatrix} x_{t+1} \\ \mu_{t+1} \end{bmatrix} = \begin{bmatrix} A & 0 \\ -R & I \end{bmatrix} \begin{bmatrix} x_t \\ \mu_t \end{bmatrix} \tag{63.20}$$

which in the special case that $\beta = 1$ agrees with equation (63.5), as expected.

By staring at system (63.20), 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 (63.20) 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

$$F = \beta(Q + \beta B'PB)^{-1}B'PA \tag{63.21}$$

it follows that

$$A - BF = (I - \beta B(Q + \beta B'PB)^{-1}B'P)A.$$

Thus, our two expressions for the closed loop dynamics agree if and only if

$$(I + \beta BQ^{-1}B'P)^{-1} = (I - \beta B(Q + \beta B'PB)^{-1}B'P). \tag{63.22}$$

Matrix equation (63.22) 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:

$$\tilde{P} = (R + F'QF) + \beta(A - BF)'P(A - BF). \tag{63.23}$$

Evidently, $\tilde{P} = P$ only when F obeys formula (63.21).

Next, note that the second equation of system (63.20) 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

$$P = R + \beta A'P(A - BF) \tag{63.24}$$

where we must require that F obeys equation (63.21).

Equations (63.23) and (63.24) provide different perspectives on the optimal value function.

CHAPTER

SIXTYFOUR

ELIMINATING CROSS PRODUCTS

64.1 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

64.2 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

$$\begin{aligned}
A^* &= A - BQ^{-1}H, \\
R^* &= R - H'Q^{-1}H, \\
P &= R^* + A^{*\prime}PA - (A^{*\prime}PB)(Q + B'PB)^{-1}B'PA^*, \\
F^* &= (Q + B'PB)^{-1}B'PA^*, \\
F &= F^* + Q^{-1}H.
\end{aligned}$$

64.3 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

$$\begin{aligned}
K(\Sigma_t) &= (A\Sigma_t D' + BF')(D\Sigma_t D' + FF')^{-1}, \\
\Sigma_{t+1} &= A\Sigma_t A' + BB' - (A\Sigma_t D' + BF')(D\Sigma_t D' + FF')^{-1}(D\Sigma_t A' + FB').
\end{aligned}$$

Define tranformed matrices

$$\begin{aligned}
A^* &= A - BF'(FF')^{-1}D, \\
B^*B^{*\prime} &= BB' - BF'(FF')^{-1}FB'.
\end{aligned}$$

64.3.1 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

$$\begin{aligned} K^* &= (A^*\Sigma D')(D\Sigma D' + FF')^{-1} \\ \Sigma &= A^*\Sigma A^{*\prime} + B^*B^{*\prime} - (A^*\Sigma D')(D\Sigma D' + FF')^{-1}(D\Sigma A^{*\prime}). \end{aligned}$$

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.

64.4 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

SIXTYFIVE

THE PERMANENT INCOME MODEL

Contents

- *The Permanent Income Model*
  - *Overview*
  - *The Savings Problem*
  - *Alternative Representations*
  - *Two Classic Examples*
  - *Further Reading*
  - *Appendix: The Euler Equation*

65.1 Overview

This lecture describes a rational expectations version of the famous permanent income model of Milton Friedman [Fri56].

Robert Hall cast Friedman's model within a linear-quadratic setting [Hal78].

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 [Hal78] and chapter 2 of [LS18].

Let's start with some imports

```
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
import random
from numba import njit
```

65.2 The Savings Problem

In this section, we state and solve the savings and consumption problem faced by the consumer.

65.2.1 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).

65.2.2 The Decision Problem

A consumer has preferences over consumption streams that are ordered by the utility functional

$$\mathbb{E}_0 \left[\sum_{t=0}^{\infty} \beta^t u(c_t) \right] \tag{65.1}$$

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 (65.1) by choosing a consumption, borrowing plan $\{c_t, b_{t+1}\}_{t=0}^{\infty}$ subject to the sequence of budget constraints

$$c_t + b_t = \frac{1}{1+r} b_{t+1} + y_t \quad t \geq 0 \tag{65.2}$$

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.

65.2.3 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*

$$\begin{aligned} z_{t+1} &= Az_t + Cw_{t+1} \\ y_t &= Uz_t \end{aligned} \tag{65.3}$$

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

$$\mathbb{E}_0 \left[\sum_{t=0}^{\infty} \beta^t b_t^2 \right] < \infty \tag{65.4}$$

This condition rules out an always-borrow scheme that would allow the consumer to enjoy bliss consumption forever.

65.2.4 First-Order Conditions

First-order conditions for maximizing (65.1) subject to (65.2) are

$$\mathbb{E}_t[u'(c_{t+1})] = u'(c_t), \qquad t = 0, 1, \ldots \tag{65.5}$$

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, (65.5) has the striking implication that consumption follows a martingale:

$$\mathbb{E}_t[c_{t+1}] = c_t \tag{65.6}$$

(In fact, quadratic preferences are *necessary* for this conclusion[1].)

One way to interpret (65.6) is that consumption will change only when "new information" about permanent income is revealed.

These ideas will be clarified below.

65.2.5 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 (65.6)
2. the period-by-period budget constraint (65.2), and
3. the boundary condition (65.4)

To accomplish this, observe first that (65.4) implies $\lim_{t\to\infty} \beta^{\frac{t}{2}} b_{t+1} = 0$.

Using this restriction on the debt path and solving (65.2) forward yields

$$b_t = \sum_{j=0}^{\infty} \beta^j (y_{t+j} - c_{t+j}) \tag{65.7}$$

Take conditional expectations on both sides of (65.7) and use the martingale property of consumption and the *law of iterated expectations* to deduce

$$b_t = \sum_{j=0}^{\infty} \beta^j \mathbb{E}_t[y_{t+j}] - \frac{c_t}{1-\beta} \tag{65.8}$$

Expressed in terms of c_t we get

$$c_t = (1-\beta)\left[\sum_{j=0}^{\infty} \beta^j \mathbb{E}_t[y_{t+j}] - b_t\right] = \frac{r}{1+r}\left[\sum_{j=0}^{\infty} \beta^j \mathbb{E}_t[y_{t+j}] - b_t\right] \tag{65.9}$$

[1] A linear marginal utility is essential for deriving (65.6) from (65.5). 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 (65.5). 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 (65.5) 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.

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 (65.9) gives

$$c_t = \frac{r}{1+r} \left[U(I - \beta A)^{-1} z_t - b_t \right] \tag{65.10}$$

Using this equality to eliminate c_t in the budget constraint (65.2) gives

$$\begin{aligned} b_{t+1} &= (1+r)(b_t + c_t - y_t) \\ &= (1+r)b_t + r[U(I - \beta A)^{-1} z_t - b_t] - (1+r)Uz_t \\ &= b_t + U[r(I - \beta A)^{-1} - (1+r)I]z_t \\ &= b_t + U(I - \beta A)^{-1}(A - I)z_t \end{aligned}$$

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:

$$\begin{aligned} z_{t+1} &= Az_t + Cw_{t+1} \\ b_{t+1} &= b_t + U[(I-\beta A)^{-1}(A-I)]z_t \\ y_t &= Uz_t \\ c_t &= (1-\beta)[U(I-\beta A)^{-1}z_t - b_t] \end{aligned} \tag{65.11}$$

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 (65.11) as

$$\begin{aligned} x_{t+1} &= \tilde{A}x_t + \tilde{C}w_{t+1} \\ \tilde{y}_t &= \tilde{U}x_t \end{aligned} \tag{65.12}$$

We can use the following formulas from *linear state space models* to compute population mean $\mu_t = \mathbb{E}x_t$ and covariance $\Sigma_t := \mathbb{E}[(x_t - \mu_t)(x_t - \mu_t)']$

$$\mu_{t+1} = \tilde{A}\mu_t \quad \text{with} \quad \mu_0 \text{ given} \tag{65.13}$$

$$\Sigma_{t+1} = \tilde{A}\Sigma_t\tilde{A}' + \tilde{C}\tilde{C}' \quad \text{with} \quad \Sigma_0 \text{ given} \tag{65.14}$$

We can then compute the mean and covariance of $\tilde{y}_t$ from

$$\begin{aligned} \mu_{y,t} &= \tilde{U}\mu_t \\ \Sigma_{y,t} &= \tilde{U}\Sigma_t\tilde{U}' \end{aligned} \tag{65.15}$$

A Simple Example with IID Income

To gain some preliminary intuition on the implications of (65.11), 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

@njit
def time_path(T):
    w = np.random.randn(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

w, b, c = time_path(T)

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)  # 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()
```

65.3 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.

65.3.1 Hall's Representation

Hall [Hal78] suggested an insightful way to summarize the implications of LQ permanent income theory.

First, to represent the solution for b_t, shift (65.9) forward one period and eliminate b_{t+1} by using (65.2) 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

$$c_{t+1} - c_t = (1-\beta)\sum_{j=0}^{\infty}\beta^j\left\{\mathbb{E}_{t+1}[y_{t+j+1}] - \mathbb{E}_t[y_{t+j+1}]\right\} \tag{65.16}$$

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 (65.16) and (65.8), which we repeat:

$$b_t = \sum_{j=0}^{\infty}\beta^j\mathbb{E}_t[y_{t+j}] - \frac{1}{1-\beta}c_t \tag{65.17}$$

Equation (65.17) 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

$$\begin{aligned}
\mathbb{E}_t \sum_{j=0}^{\infty} \beta^j y_{t+j} &= U(I - \beta A)^{-1} z_t \\
\mathbb{E}_{t+1} \sum_{j=0}^{\infty} \beta^j y_{t+j+1} &= U(I - \beta A)^{-1} z_{t+1} \\
\mathbb{E}_t \sum_{j=0}^{\infty} \beta^j y_{t+j+1} &= U(I - \beta A)^{-1} A z_t
\end{aligned}$$

Using these formulas together with (65.3) and substituting into (65.16) and (65.17) gives the following representation for the consumer's optimum decision rule:

$$\begin{aligned}
c_{t+1} &= c_t + (1-\beta) U (I - \beta A)^{-1} C w_{t+1} \\
b_t &= U(I - \beta A)^{-1} z_t - \frac{1}{1-\beta} c_t \\
y_t &= U z_t \\
z_{t+1} &= A z_t + C w_{t+1}
\end{aligned} \tag{65.18}$$

Representation (65.18) 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 (65.6).

65.3.2 Cointegration

Representation (65.18) reveals that the joint process $\{c_t, b_t\}$ possesses the property that Engle and Granger [EG87] 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 (65.11) 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 (65.18) we have

$$(1-\beta) b_t + c_t = (1-\beta) U (I - \beta A)^{-1} z_t \tag{65.19}$$

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 (65.19) can be rearranged to take the form

$$(1-\beta)b_t + c_t = (1-\beta)\mathbb{E}_t \sum_{j=0}^{\infty} \beta^j y_{t+j} \tag{65.20}$$

Equation (65.20) asserts that the *cointegrating residual* on the left side equals the conditional expectation of the geometric sum of future incomes on the right[4].

65.3.3 Cross-Sectional Implications

Consider again (65.18), this time in light of our discussion of distribution dynamics in the *lecture on linear systems*.

The dynamics of c_t are given by

$$c_{t+1} = c_t + (1-\beta)U(I-\beta A)^{-1}Cw_{t+1} \tag{65.21}$$

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

$$\text{Var}[c_t] = \text{Var}[c_0] + t\,\hat{\sigma}^2 \tag{65.22}$$

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 (65.22) 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., [DP94], [STY04]).

65.3.4 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 [JYC88], [LL01], [LL04] for interesting applications of related ideas.

65.3.5 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 (65.3) has the moving average representation

$$y_{t+1} = d(L)w_{t+1} \tag{65.23}$$

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

$$\mathbb{E}_{t+1}[y_{t+j}] - \mathbb{E}_t[y_{t+j}] = d_{j-1} w_{t+1} \tag{65.24}$$

Using (65.24) in (65.16) gives

$$c_{t+1} - c_t = (1-\beta) d(\beta) w_{t+1} \tag{65.25}$$

The object $d(\beta)$ is the **present value of the moving average coefficients** in the representation for the endowment process y_t.

65.4 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.

65.4.1 Example 1

Assume as before that the consumer observes the state z_t at time t.

In view of (65.18) we have

$$c_{t+1} - c_t = \sigma_1 w_{1t+1} + (1-\beta)\sigma_2 w_{2t+1} \tag{65.26}$$

Formula (65.26) shows how an increment $\sigma_1 w_{1t+1}$ to the permanent component of income z_{1t+1} leads to

[5] Representation (65.3) 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 (65.11) to this example shows that

$$b_{t+1} - b_t = -z_{2t} = -\sigma_2 w_{2t} \tag{65.27}$$

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

@njit
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.

65.4.2 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 (65.18).

The discussion in sections 2.9.1 and 2.11.3 of [LS18] 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
- $a_t := y_t - E[y_t \,|\, y_{t-1}, \dots, y_0]$

In the same discussion in [LS18] 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 (65.18) implies

$$c_{t+1} - c_t = [1 - \beta(1 - K)]a_{t+1} \tag{65.28}$$

where the endowment process can now be represented in terms of the univariate innovation to y_t as

$$y_{t+1} - y_t = a_{t+1} - (1 - K)a_t \tag{65.29}$$

Equation (65.29) 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 (65.29), the first difference of income is a first-order moving average.

Equation (65.28) asserts that the first difference of consumption is IID.

Application of formula to this example shows that

$$b_{t+1} - b_t = (K - 1)a_t \tag{65.30}$$

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.

65.5 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., [HM82], [Par99], [Dea91], [Car01].

65.6 Appendix: The Euler Equation

Where does the first-order condition (65.5) 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 (65.2) 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 (65.5) in the two-period case.

The proof for the general case is similar.

CHAPTER

SIXTYSIX

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
```

66.1 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 [Hal78] 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 [Bar79] 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 [Bew86].

(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
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import quantecon as qe
import numpy as np
import scipy.linalg as la
```

66.2 Setup

Let's recall the basic features of the model discussed in the *permanent income model*.

Consumer preferences are ordered by

$$E_0 \sum_{t=0}^{\infty} \beta^t u(c_t) \tag{66.1}$$

where $u(c) = -(c-\gamma)^2$.

The consumer maximizes (66.1) by choosing a consumption, borrowing plan $\{c_t, b_{t+1}\}_{t=0}^{\infty}$ subject to the sequence of budget constraints

$$c_t + b_t = \frac{1}{1+r} b_{t+1} + y_t, \quad t \geq 0 \tag{66.2}$$

and the no-Ponzi condition

$$E_0 \sum_{t=0}^{\infty} \beta^t b_t^2 < \infty \tag{66.3}$$

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

$$\begin{aligned} z_{t+1} &= Az_t + Cw_{t+1} \\ y_t &= Uz_t \end{aligned} \tag{66.4}$$

The restrictions on the shock process and parameters are the same as in our *previous lecture*.

66.2.1 Digression on a Useful Isomorphism

The LQ permanent income model of consumption is mathematically isomorphic with a version of Barro's [Bar79] 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.

66.2.2 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 (66.4), as discussed in our lecture on *linear models*.

To do so we take

$$z_t = \begin{bmatrix} 1 \\ y_t \\ y_{t-1} \end{bmatrix}, \quad A = \begin{bmatrix} 1 & 0 & 0 \\ \alpha & \rho_1 & \rho_2 \\ 0 & 1 & 0 \end{bmatrix}, \quad C = \begin{bmatrix} 0 \\ \sigma \\ 0 \end{bmatrix}, \quad \text{and} \quad U = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}$$

66.3 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

66.3.1 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

$$x_{t+1} = \tilde{A} x_t + \tilde{B} u_t + \tilde{C} w_{t+1}, \qquad t \geq 0, \tag{66.5}$$

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 (66.4).

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 $x_{t+1} = (\tilde{A} - \tilde{B} F) x_t + \tilde{C} w_{t+1}$.

66.3.2 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 (66.5) 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 (66.5) 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 (66.3) 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.

66.4 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.

66.4.1 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)
```

(continues on next page)

(continued from previous page)

```
μ_z, μ_y, Σ_z, Σ_y, Σ_yx = Lz.stationary_distributions()

# 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]
```

(continues on next page)

(continued from previous page)

```
 [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
```

66.4.2 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
```

(continues on next page)

(continued from previous page)

```
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],
       [-9.51248175e-06,  9.51247915e-08,  0.00000000e+00,
        -1.99999923e-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.

66.5 Two Example Economies

In the spirit of Bewley models [Bew86], 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.

66.5.1 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)
```

66.5.2 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], Σ_y[1, 1]
        debt_mean[t], debt_var[t] = μ_x[3], Σ_x[3, 3]

    return bsim, csim, ysim, cons_mean, cons_var, debt_mean, debt_var
```

(continues on next page)

(continued from previous page)

```
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")
    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")
```

(continues on next page)

(continued from previous page)

```
    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 **co-integrating relationship**

$$(1-\beta)b_t + c_t = (1-\beta)E_t \sum_{j=0}^{\infty} \beta^j y_{t+j} \tag{66.6}$$

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 (66.6), 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()
```

66.5.3 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 [Bew86] 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
102
101
100
99
98
97
0
20
40
60
80
100
120
140
t

CHAPTER

SIXTYSEVEN

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
```

67.1 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 [HMMS60], 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

$$
\begin{aligned}
\pi_t =& p_t S_t - c\left(Q_t\right) - d\left(I_t, S_t\right) \\
=& \left(a_0 - a_1 S_t + v_t\right) S_t - c_1 Q_t - c_2 Q_t^2 - d_1 I_t - d_2 \left(S_t - I_t\right)^2 \\
=& a_0 S_t - a_1 S_t^2 + G z_t S_t - c_1 Q_t - c_2 Q_t^2 - d_1 I_t - d_2 S_t^2 - d_2 I_t^2 + 2 d_2 S_t I_t \\
=& -\left(\underbrace{d_1 I_t + d_2 I_t^2}_{x_t' R x_t} \underbrace{+ a_1 S_t^2 + d_2 S_t^2 + c_2 Q_t^2}_{u_t' Q u_t} \underbrace{- a_0 S_t - G z_t S_t + c_1 Q_t - 2 d_2 S_t I_t}_{2 u_t' N x_t}\right) \\
=& -\left(\begin{bmatrix} I_t & z_t' \end{bmatrix} \underbrace{\begin{bmatrix} d_2 & \frac{d_1}{2} S_c \\ \frac{d_1}{2} S_c' & 0 \end{bmatrix}}_{\equiv R} \begin{bmatrix} I_t \\ z_t \end{bmatrix} + \begin{bmatrix} Q_t & S_t \end{bmatrix} \underbrace{\begin{bmatrix} c_2 & 0 \\ 0 & a_1 + d_2 \end{bmatrix}}_{\equiv Q} \begin{bmatrix} Q_t \\ S_t \end{bmatrix} + 2 \begin{bmatrix} Q_t & S_t \end{bmatrix} \underbrace{\begin{bmatrix} 0 & \frac{c_1}{2} S_c \\ -d_2 & -\frac{a_0}{2} S_c - \frac{G}{2} \end{bmatrix}}_{\equiv N} \begin{bmatrix} I \\ z \end{bmatrix}\right.
\end{aligned}
$$

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
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
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
```

(continues on next page)

(continued from previous page)

```
                 a1=1,
                 A22=[[1,   0],     # z process
                      [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()
```

(continues on next page)

(continued from previous page)

```
    def simulate(self, x0, T=100):

        c1, c2 = self.c1, self.c2
        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]
        ?_path = (G @ z_path)[0, :]

        Q_path = u_path[0, :]
        S_path = u_path[1, :]

        revenue = (a0 - a1 * S_path + ?_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 + ?_path - c1) / (2 * (a1 + c2))
        Q_hardwired = (a0 + ?_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), ?_path, color='r')
        ax1_.set_ylabel("demand shock", color='r')
        span = max(abs(?_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')
```

(continues on next page)

(continued from previous page)

```
        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.

67.2 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.

67.3 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

$$Q_t^{ni} = \frac{a_0 + \nu_t - c_1}{c_2 + a_1}.$$

67.4 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)
```

67.5 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]])
```

(continues on next page)

(continued from previous page)

```
x0 = [0, 1, 0]
ex2.simulate(x0)
```

67.6 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)
```

67.7 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)
```

67.8 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)
```

67.9 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)
```

67.10 Exercises

Please try to analyze some inventory sales smoothing problems using the `SmoothingExample` class.

Exercise 67.10.1

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 to Exercise 67.10.1

```
# set parameters
α = 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 = SmoothingEmpty(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 67.10.2

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 to Exercise 67.10.2

```
x0 = [0, 1, 0]
```

```
SmoothingExample(c2=5).simulate(x0)
```

```
SmoothingExample(d2=5).simulate(x0)
```

Part X

Multiple Agent Models

CHAPTER

SIXTYEIGHT

SCHELLING'S SEGREGATION MODEL

Contents

- *Schelling's Segregation Model*
  - *Outline*
  - *The Model*
  - *Results*
  - *Exercises*

68.1 Outline

In 1969, Thomas C. Schelling developed a simple but striking model of racial segregation [Sch69].

His model studies the dynamics of racially mixed neighborhoods.

Like much of Schelling's work, the model shows how local interactions can lead to surprising aggregate structure.

In particular, it shows that relatively mild preference for neighbors of similar race can lead in aggregate to the collapse of mixed neighborhoods, and high levels of segregation.

In recognition of this and other research, Schelling was awarded the 2005 Nobel Prize in Economic Sciences (joint with Robert Aumann).

In this lecture, we (in fact you) will build and run a version of Schelling's model.

Let's start with some imports:

```
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
from random import uniform, seed
from math import sqrt
```

68.2 The Model

We will cover a variation of Schelling's model that is easy to program and captures the main idea.

68.2.1 Set-Up

Suppose we have two types of people: orange people and green people.

For the purpose of this lecture, we will assume there are 250 of each type.

These agents all live on a single unit square.

The location of an agent is just a point (x, y), where $0 < x, y < 1$.

68.2.2 Preferences

We will say that an agent is *happy* if half or more of her 10 nearest neighbors are of the same type.

Here 'nearest' is in terms of Euclidean distance.

An agent who is not happy is called *unhappy*.

An important point here is that agents are not averse to living in mixed areas.

They are perfectly happy if half their neighbors are of the other color.

68.2.3 Behavior

Initially, agents are mixed together (integrated).

In particular, the initial location of each agent is an independent draw from a bivariate uniform distribution on $S = (0, 1)^2$.

Now, cycling through the set of all agents, each agent is now given the chance to stay or move.

We assume that each agent will stay put if they are happy and move if unhappy.

The algorithm for moving is as follows

1. Draw a random location in S
2. If happy at new location, move there
3. Else, go to step 1

In this way, we cycle continuously through the agents, moving as required.

We continue to cycle until no one wishes to move.

68.3 Results

Let's have a look at the results we got when we coded and ran this model.

As discussed above, agents are initially mixed randomly together.

But after several cycles, they become segregated into distinct regions.

In this instance, the program terminated after 4 cycles through the set of agents, indicating that all agents had reached a state of happiness.

What is striking about the pictures is how rapidly racial integration breaks down.

This is despite the fact that people in the model don't actually mind living mixed with the other type.

Even with these preferences, the outcome is a high degree of segregation.

Cycle 1
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0

Cycle 2
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0

Cycle 3
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0

68.4 Exercises

Exercise 68.4.1

Implement and run this simulation for yourself.

Consider the following structure for your program.

Agents can be modeled as objects.

Here's an indication of how they might look

```
* Data:

    * type (green or orange)
    * location

* Methods:

    * determine whether happy or not given locations of other agents

    * If not happy, move

        * find a new location where happy
```

And here's some pseudocode for the main loop

```
while agents are still moving
    for agent in agents
        give agent the opportunity to move
```

Use 250 agents of each type.

Solution to Exercise 68.4.1

Here's one solution that does the job we want.

If you feel like a further exercise, you can probably speed up some of the computations and then increase the number of agents.

```
seed(10)  # For reproducible random numbers

class Agent:

    def __init__(self, type):
        self.type = type
        self.draw_location()

    def draw_location(self):
        self.location = uniform(0, 1), uniform(0, 1)

    def get_distance(self, other):
        "Computes the euclidean distance between self and other agent."
        a = (self.location[0] - other.location[0])**2
        b = (self.location[1] - other.location[1])**2
```

(continues on next page)

(continued from previous page)

```
        return sqrt(a + b)

    def happy(self, agents):
        "True if sufficient number of nearest neighbors are of the same type."
        distances = []
        # distances is a list of pairs (d, agent), where d is distance from
        # agent to self
        for agent in agents:
            if self != agent:
                distance = self.get_distance(agent)
                distances.append((distance, agent))
        # == Sort from smallest to largest, according to distance == #
        distances.sort()
        # == Extract the neighboring agents == #
        neighbors = [agent for d, agent in distances[:num_neighbors]]
        # == Count how many neighbors have the same type as self == #
        num_same_type = sum(self.type == agent.type for agent in neighbors)
        return num_same_type >= require_same_type

    def update(self, agents):
        "If not happy, then randomly choose new locations until happy."
        while not self.happy(agents):
            self.draw_location()


def plot_distribution(agents, cycle_num):
    "Plot the distribution of agents after cycle_num rounds of the loop."
    x_values_0, y_values_0 = [], []
    x_values_1, y_values_1 = [], []
    # == Obtain locations of each type == #
    for agent in agents:
        x, y = agent.location
        if agent.type == 0:
            x_values_0.append(x)
            y_values_0.append(y)
        else:
            x_values_1.append(x)
            y_values_1.append(y)
    fig, ax = plt.subplots(figsize=(8, 8))
    plot_args = {'markersize': 8, 'alpha': 0.6}
    ax.set_facecolor('azure')
    ax.plot(x_values_0, y_values_0, 'o', markerfacecolor='orange', **plot_args)
    ax.plot(x_values_1, y_values_1, 'o', markerfacecolor='green', **plot_args)
    ax.set_title(f'Cycle {cycle_num-1}')
    plt.show()

# == Main == #

num_of_type_0 = 250
num_of_type_1 = 250
num_neighbors = 10       # Number of agents regarded as neighbors
require_same_type = 5    # Want at least this many neighbors to be same type

# == Create a list of agents == #
agents = [Agent(0) for i in range(num_of_type_0)]
agents.extend(Agent(1) for i in range(num_of_type_1))
```

(continues on next page)

(continued from previous page)

```
count = 1
# ==  Loop until none wishes to move == #
while True:
    print('Entering loop ', count)
    plot_distribution(agents, count)
    count += 1
    no_one_moved = True
    for agent in agents:
        old_location = agent.location
        agent.update(agents)
        if agent.location != old_location:
            no_one_moved = False
    if no_one_moved:
        break

print('Converged, terminating.')
```

```
Entering loop  1
```

```
Entering loop  2
```

```
Entering loop  3
```

```
Entering loop  4
```

```
Converged, terminating.
```

CHAPTER

SIXTYNINE

A LAKE MODEL OF EMPLOYMENT AND UNEMPLOYMENT

Contents

- *A Lake Model of Employment and Unemployment*
  - *Overview*
  - *The Model*
  - *Implementation*
  - *Dynamics of an Individual Worker*
  - *Endogenous Job Finding Rate*
  - *Exercises*

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install quantecon
```

69.1 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 jobs 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
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
from quantecon import MarkovChain
from scipy.stats import norm
from scipy.optimize import brentq
from quantecon.distributions import BetaBinomial
from numba import jit
```

69.1.1 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.

69.2 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 $g = b - d$.

69.2.1 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

We also want to know 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)

69.2.2 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

$$X_{t+1} = AX_t \quad \text{where} \quad A := \begin{pmatrix} (1-d)(1-\lambda)+b & (1-d)\alpha + b \\ (1-d)\lambda & (1-d)(1-\alpha) \end{pmatrix}$$

This law tells us how total employment and unemployment evolve over time.

69.2.3 Laws of Motion for Rates

Now let's derive the law of motion for rates.

To get these we can divide both sides of $X_{t+1} = AX_t$ by N_{t+1} to get

$$\begin{pmatrix} U_{t+1}/N_{t+1} \\ E_{t+1}/N_{t+1} \end{pmatrix} = \frac{1}{1+g} A \begin{pmatrix} U_t/N_t \\ E_t/N_t \end{pmatrix}$$

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

$$x_{t+1} = \hat{A} x_t \quad \text{where} \quad \hat{A} := \frac{1}{1+g} A$$

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 $\hat{A}$ sum to 1.

69.3 Implementation

Let's code up these equations.

To do this we're going to use a class that we'll call `LakeModel`.

This class will

1. store the primitives α, λ, b, d
2. compute and store the implied objects $g, A, \hat{A}$
3. provide methods to simulate dynamics of the stocks and rates
4. provide a method to compute the steady state vector $\bar{x}$ of employment and unemployment rates using *a technique* we previously introduced for computing stationary distributions of Markov chains

Please be careful because the implied objects $g, A, \hat{A}$ will not change if you only change the primitives.

For example, if you would like to update a primitive like $\alpha = 0.03$, you need to create an instance and update it by `lm = LakeModel(α=0.03)`.

In the exercises, we show how to avoid this issue by using getter and setter methods.

```
class LakeModel:
    """
    Solves the lake model and computes dynamics of unemployment stocks and
    rates.

    Parameters:
    ------------
    λ : scalar
        The job finding rate for currently unemployed workers
    α : scalar
        The dismissal rate for currently employed workers
    b : scalar
        Entry rate into the labor force
    d : scalar
        Exit rate from the labor force

    """
    def __init__(self, λ=0.283, α=0.013, b=0.0124, d=0.00822):
        self.λ, self.α, self.b, self.d = λ, α, b, d

        λ, α, b, d = self.λ, self.α, self.b, self.d
        self.g = b - d
        self.A = np.array([[(1-d) * (1-λ) + b,       (1 - d) * α + b],
                           [         (1-d) * λ,       (1 - d) * (1 - α)]])

        self.A_hat = self.A / (1 + self.g)


    def rate_steady_state(self, tol=1e-6):
        """
        Finds the steady state of the system :math:`x_{t+1} = \hat A x_{t}`

        Returns
        --------
        xbar : steady state vector of employment and unemployment rates
        """
```

(continues on next page)

(continued from previous page)

```
        x = np.array([self.A_hat[0, 1], self.A_hat[1, 0]])
        x /= x.sum()
        return x

    def simulate_stock_path(self, X0, T):
        """
        Simulates the sequence of Employment and Unemployment stocks

        Parameters
        ------------
        X0 : array
            Contains initial values (E0, U0)
        T : int
            Number of periods to simulate

        Returns
        ---------
        X : iterator
            Contains sequence of employment and unemployment stocks
        """

        X = np.atleast_1d(X0)  # Recast as array just in case
        for t in range(T):
            yield X
            X = self.A @ X

    def simulate_rate_path(self, x0, T):
        """
        Simulates the sequence of employment and unemployment rates

        Parameters
        ------------
        x0 : array
            Contains initial values (e0,u0)
        T : int
            Number of periods to simulate

        Returns
        ---------
        x : iterator
            Contains sequence of employment and unemployment rates

        """
        x = np.atleast_1d(x0)  # Recast as array just in case
        for t in range(T):
            yield x
            x = self.A_hat @ x
```

As explained, if we create an instance and update it by `lm = LakeModel(α=0.03)`, derived objects like A will also change.

```
lm = LakeModel()
lm.α
```

```
0.013
```

```
lm.A
```

```
array([[0.72350626, 0.02529314],
       [0.28067374, 0.97888686]])
```

```
lm = LakeModel(α = 0.03)
lm.A
```

```
array([[0.72350626, 0.0421534 ],
       [0.28067374, 0.9620266 ]])
```

69.3.1 Aggregate Dynamics

Let's run a simulation under the default parameters (see above) starting from $X_0 = (12, 138)$

```
lm = LakeModel()
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

fig, axes = plt.subplots(3, 1, figsize=(10, 8))
X_0 = (U_0, E_0)
X_path = np.vstack(tuple(lm.simulate_stock_path(X_0, T)))

axes[0].plot(X_path[:, 0], lw=2)
axes[0].set_title('Unemployment')

axes[1].plot(X_path[:, 1], lw=2)
axes[1].set_title('Employment')

axes[2].plot(X_path.sum(1), lw=2)
axes[2].set_title('Labor force')

for ax in axes:
    ax.grid()

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.

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

- $\bar{x} = \hat{A}\bar{x}$
- the components satisfy $\bar{e} + \bar{u} = 1$

This equation tells us that a steady state level $\bar{x}$ is an eigenvector of $\hat{A}$ associated with a unit eigenvalue.

We also have $x_t \to \bar{x}$ as $t \to \infty$ provided that the remaining eigenvalue of $\hat{A}$ has modulus less than 1.

This is the case for our default parameters:

```
lm = LakeModel()
e, f = np.linalg.eigvals(lm.A_hat)
abs(e), abs(f)
```

```
(0.6953067378358462, 1.0)
```

Let's look at the convergence of the unemployment and employment rate to steady state levels (dashed red line)

```
lm = LakeModel()
e_0 = 0.92     # Initial employment rate
```

(continues on next page)

(continued from previous page)

```
u_0 = 1 - e_0  # Initial unemployment rate
T = 50         # Simulation length

xbar = lm.rate_steady_state()

fig, axes = plt.subplots(2, 1, figsize=(10, 8))
x_0 = (u_0, e_0)
x_path = np.vstack(tuple(lm.simulate_rate_path(x_0, T)))

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, 'r', '--')
    axes[i].set_title(title)
    axes[i].grid()

plt.tight_layout()
plt.show()
```

69.4 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.

69.4.1 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 $\hat{A}$ 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.

69.4.2 Convergence Rate

How long does it take for time series sample averages to converge to cross-sectional averages?

We can use QuantEcon.py's MarkovChain class to investigate this.

Let's plot the path of the sample averages over 5,000 periods

```
lm = LakeModel(d=0, b=0)
T = 5000  # Simulation length

α, λ = lm.α, lm.λ

P = [[1 - λ,         λ],
     [    α,     1 - α]]

mc = MarkovChain(P)

xbar = lm.rate_steady_state()

fig, axes = plt.subplots(2, 1, figsize=(10, 8))
s_path = mc.simulate(T, init=1)
s_bar_e = s_path.cumsum() / range(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, 'r', '--')
    axes[i].set_title(titles[i])
    axes[i].grid()

plt.tight_layout()
plt.show()
```

The stationary probabilities are given by the dashed red 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.

69.5 Endogenous Job Finding Rate

We now make the hiring rate endogenous.

The transition rate from unemployment to employment will be determined by the McCall search model [McC70].

All details relevant to the following discussion can be found in *our treatment* of that model.

69.5.1 Reservation Wage

The most important thing to remember about the model is that optimal decisions are 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.

As we saw in *our discussion of the model*, 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

69.5.2 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

$$\lambda = \gamma \mathbb{P}\{w_t \geq \bar{w}\} = \gamma \sum_{w' \geq \bar{w}} p(w') \tag{69.1}$$

69.5.3 Fiscal Policy

We can use the McCall search version of the Lake Model 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.

Thus, 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 (69.1) 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 in the *McCall search model lecture*.

The wage offer distribution will be a discretized version of the lognormal distribution $LN(\log(20), 1)$, as shown in the next figure

We take a period to be a month.

We set b and d to match monthly birth and death rates, respectively, in the U.S. population

- $b = 0.0124$
- $d = 0.00822$

Following [DFH06], we set α, the hazard rate of leaving employment, to

- $\alpha = 0.013$

69.5.4 Fiscal Policy Code

We will make use of techniques from the *McCall model lecture*

The first piece of code implements value function iteration

```
# A default utility function

@jit
def u(c, σ):
    if c > 0:
        return (c**(1 - σ) - 1) / (1 - σ)
    else:
```

(continues on next page)

(continued from previous page)

```
        return -10e6


class McCallModel:
    """
    Stores the parameters and functions associated with a given model.
    """

    def __init__(self,
                 α=0.2,       # Job separation rate
                 β=0.98,      # Discount rate
                 γ=0.7,       # Job offer rate
                 c=6.0,       # Unemployment compensation
                 σ=2.0,       # Utility parameter
                 w_vec=None,  # Possible wage values
                 p_vec=None): # Probabilities over w_vec

        self.α, self.β, self.γ, self.c = α, β, γ, c
        self.σ = σ

        # Add a default wage vector and probabilities over the vector using
        # the beta-binomial distribution
        if w_vec is None:
            n = 60  # Number of possible outcomes for wage
            # Wages between 10 and 20
            self.w_vec = np.linspace(10, 20, n)
            a, b = 600, 400  # Shape parameters
            dist = BetaBinomial(n-1, a, b)
            self.p_vec = dist.pdf()
        else:
            self.w_vec = w_vec
            self.p_vec = p_vec

@jit
def _update_bellman(α, β, γ, c, σ, w_vec, p_vec, V, V_new, U):
    """
    A jitted function to update the Bellman equations.  Note that V_new is
    modified in place (i.e, modified by this function).  The new value of U
    is returned.

    """
    for w_idx, w in enumerate(w_vec):
        # w_idx indexes the vector of possible wages
        V_new[w_idx] = u(w, σ) + β * ((1 - α) * V[w_idx] + α * U)

    U_new = u(c, σ) + β * (1 - γ) * U + \
                    β * γ * np.sum(np.maximum(U, V) * p_vec)

    return U_new


def solve_mccall_model(mcm, tol=1e-5, max_iter=2000):
    """
    Iterates to convergence on the Bellman equations

    Parameters
```

(continues on next page)

(continued from previous page)

```
    ----------
    mcm : an instance of McCallModel
    tol : float
        error tolerance
    max_iter : int
        the maximum number of iterations
    """

    V = np.ones(len(mcm.w_vec))  # Initial guess of V
    V_new = np.empty_like(V)     # To store updates to V
    U = 1                        # Initial guess of U
    i = 0
    error = tol + 1

    while error > tol and i < max_iter:
        U_new = _update_bellman(mcm.α, mcm.β, mcm.γ,
                mcm.c, mcm.σ, mcm.w_vec, mcm.p_vec, V, V_new, U)
        error_1 = np.max(np.abs(V_new - V))
        error_2 = np.abs(U_new - U)
        error = max(error_1, error_2)
        V[:] = V_new
        U = U_new
        i += 1

    return V, U
```

```
/tmp/ipykernel_2962/1748977302.py:3: NumbaDeprecationWarning: The 'nopython'␣
↪keyword argument was not supplied to the 'numba.jit' decorator. The implicit␣
↪default value for this argument is currently False, but it will be changed to␣
↪True in Numba 0.59.0. See https://numba.readthedocs.io/en/stable/reference/
↪deprecation.html#deprecation-of-object-mode-fall-back-behaviour-when-using-jit␣
↪for details.
  @jit
/tmp/ipykernel_2962/1748977302.py:41: NumbaDeprecationWarning: The 'nopython'␣
↪keyword argument was not supplied to the 'numba.jit' decorator. The implicit␣
↪default value for this argument is currently False, but it will be changed to␣
↪True in Numba 0.59.0. See https://numba.readthedocs.io/en/stable/reference/
↪deprecation.html#deprecation-of-object-mode-fall-back-behaviour-when-using-jit␣
↪for details.
  @jit
```

The second piece of code is used to complete the reservation wage:

```
def compute_reservation_wage(mcm, return_values=False):
    """
    Computes the reservation wage of an instance of the McCall model
    by finding the smallest w such that V(w) > U.

    If V(w) > U for all w, then the reservation wage w_bar is set to
    the lowest wage in mcm.w_vec.

    If v(w) < U for all w, then w_bar is set to np.inf.

    Parameters
    ----------
```

(continues on next page)

(continued from previous page)

```
    mcm : an instance of McCallModel
    return_values : bool (optional, default=False)
        Return the value functions as well

    Returns
    -------
    w_bar : scalar
        The reservation wage

    """

    V, U = solve_mccall_model(mcm)
    w_idx = np.searchsorted(V - U, 0)

    if w_idx == len(V):
        w_bar = np.inf
    else:
        w_bar = mcm.w_vec[w_idx]

    if return_values == False:
        return w_bar
    else:
        return w_bar, V, U
```

Now let's compute and plot welfare, employment, unemployment, and tax revenue as a function of the unemployment compensation rate

```
# Some global variables that will stay constant
α = 0.013
α_q = (1-(1-α)**3)    # Quarterly (α is monthly)
b = 0.0124
d = 0.00822
β = 0.98
γ = 1.0
σ = 2.0

# The default wage distribution --- a discretized lognormal
log_wage_mean, wage_grid_size, max_wage = 20, 200, 170
logw_dist = norm(np.log(log_wage_mean), 1)
w_vec = np.linspace(1e-8, max_wage, wage_grid_size + 1)
cdf = logw_dist.cdf(np.log(w_vec))
pdf = cdf[1:] - cdf[:-1]
p_vec = pdf / pdf.sum()
w_vec = (w_vec[1:] + w_vec[:-1]) / 2


def compute_optimal_quantities(c, τ):
    """
    Compute the reservation wage, job finding rate and value functions
    of the workers given c and τ.

    """

    mcm = McCallModel(α=α_q,
                      β=β,
                      γ=γ,
```

(continues on next page)

(continued from previous page)

```
                    c=c-τ,          # Post tax compensation
                    σ=σ,
                    w_vec=w_vec-τ,  # Post tax wages
                    p_vec=p_vec)

    w_bar, V, U = compute_reservation_wage(mcm, return_values=True)
    λ = γ * np.sum(p_vec[w_vec - τ > w_bar])
    return w_bar, λ, V, U

def compute_steady_state_quantities(c, τ):
    """
    Compute the steady state unemployment rate given c and τ using optimal
    quantities from the McCall model and computing corresponding steady
    state quantities

    """
    w_bar, λ, V, U = compute_optimal_quantities(c, τ)

    # Compute steady state employment and unemployment rates
    lm = LakeModel(α=α_q, λ=λ, b=b, d=d)
    x = lm.rate_steady_state()
    u, e = x

    # Compute steady state welfare
    w = np.sum(V * p_vec * (w_vec - τ > w_bar)) / np.sum(p_vec * (w_vec -
            τ > w_bar))
    welfare = e * w + u * U

    return e, u, welfare


def find_balanced_budget_tax(c):
    """
    Find the tax level that will induce a balanced budget.

    """
    def steady_state_budget(t):
        e, u, w = compute_steady_state_quantities(c, t)
        return t - u * c

    τ = brentq(steady_state_budget, 0.0, 0.9 * c)
    return τ


# Levels of unemployment insurance we wish to study
c_vec = np.linspace(5, 140, 60)

tax_vec = []
unempl_vec = []
empl_vec = []
welfare_vec = []

for c in c_vec:
    t = find_balanced_budget_tax(c)
    e_rate, u_rate, welfare = compute_steady_state_quantities(c, t)
    tax_vec.append(t)
```

(continues on next page)

(continued from previous page)

```
    unempl_vec.append(u_rate)
    empl_vec.append(e_rate)
    welfare_vec.append(welfare)

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)
    ax.grid()

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.

69.6 Exercises

Exercise 69.6.1

In the Lake Model, there is derived data such as A which depends on primitives like α and λ.

So, when a user alters these primitives, we need the derived data to update automatically.

(For example, if a user changes the value of b for a given instance of the class, we would like $g = b - d$ to update automatically)

In the code above, we took care of this issue by creating new instances every time we wanted to change parameters.

That way the derived data is always matched to current parameter values.

However, we can use descriptors instead, so that derived data is updated whenever parameters are changed.

This is safer and means we don't need to create a fresh instance for every new parameterization.

(On the other hand, the code becomes denser, which is why we don't always use the descriptor approach in our lectures.)

In this exercise, your task is to arrange the `LakeModel` class by using descriptors and decorators such as `@property`.

(If you need to refresh your understanding of how these work, consult this lecture.)

Solution to Exercise 69.6.1

Here is one solution

```
class LakeModelModified:
    """
    Solves the lake model and computes dynamics of unemployment stocks and
    rates.

    Parameters:
    ------------
    λ : scalar
        The job finding rate for currently unemployed workers
    α : scalar
        The dismissal rate for currently employed workers
    b : scalar
        Entry rate into the labor force
    d : scalar
        Exit rate from the labor force

    """
    def __init__(self, λ=0.283, α=0.013, b=0.0124, d=0.00822):
        self._λ, self._α, self._b, self._d = λ, α, b, d
        self.compute_derived_values()

    def compute_derived_values(self):
        # Unpack names to simplify expression
        λ, α, b, d = self._λ, self._α, self._b, self._d

        self._g = b - d
        self._A = np.array([[(1-d) * (1-λ) + b,       (1 - d) * α + b],
                            [         (1-d) * λ,    (1 - d) * (1 - α)]])
```

(continues on next page)

(continued from previous page)

```
        self._A_hat = self._A / (1 + self._g)

    @property
    def g(self):
        return self._g

    @property
    def A(self):
        return self._A

    @property
    def A_hat(self):
        return self._A_hat

    @property
    def λ(self):
        return self._λ

    @λ.setter
    def λ(self, new_value):
        self._λ = new_value
        self.compute_derived_values()

    @property
    def α(self):
        return self._α

    @α.setter
    def α(self, new_value):
        self._α = new_value
        self.compute_derived_values()

    @property
    def b(self):
        return self._b

    @b.setter
    def b(self, new_value):
        self._b = new_value
        self.compute_derived_values()

    @property
    def d(self):
        return self._d

    @d.setter
    def d(self, new_value):
        self._d = new_value
        self.compute_derived_values()


    def rate_steady_state(self, tol=1e-6):
        """
        Finds the steady state of the system :math:`x_{t+1} = \hat A x_{t}`

        Returns
```

(continues on next page)

(continued from previous page)

```
        --------
        xbar : steady state vector of employment and unemployment rates
        """
        x = np.array([self.A_hat[0, 1], self.A_hat[1, 0]])
        x /= x.sum()
        return x

    def simulate_stock_path(self, X0, T):
        """
        Simulates the sequence of Employment and Unemployment stocks

        Parameters
        ------------
        X0 : array
            Contains initial values (E0, U0)
        T : int
            Number of periods to simulate

        Returns
        ---------
        X : iterator
            Contains sequence of employment and unemployment stocks
        """

        X = np.atleast_1d(X0)  # Recast as array just in case
        for t in range(T):
            yield X
            X = self.A @ X

    def simulate_rate_path(self, x0, T):
        """
        Simulates the sequence of employment and unemployment rates

        Parameters
        ------------
        x0 : array
            Contains initial values (e0,u0)
        T : int
            Number of periods to simulate

        Returns
        ---------
        x : iterator
            Contains sequence of employment and unemployment rates

        """
        x = np.atleast_1d(x0)  # Recast as array just in case
        for t in range(T):
            yield x
            x = self.A_hat @ x
```

Exercise 69.6.2

Consider an economy with an initial stock of workers $N_0 = 100$ at the steady state level of employment in the baseline parameterization

- $\alpha = 0.013$
- $\lambda = 0.283$
- $b = 0.0124$
- $d = 0.00822$

(The values for α and λ follow [DFH06])

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?

Note: It may be easier to use the class created in exercise 1 to help with changing variables.

Solution to Exercise 69.6.2

We begin by constructing the class containing the default parameters and assigning the steady state values to `x0`

```
lm = LakeModelModified()
x0 = lm.rate_steady_state()
print(f"Initial Steady State: {x0}")
```

```
Initial Steady State: [0.08266627 0.91733373]
```

Initialize the simulation values

```
N0 = 100
T = 50
```

New legislation changes λ to 0.2

```
lm.λ = 0.2

xbar = lm.rate_steady_state()  # new steady state
X_path = np.vstack(tuple(lm.simulate_stock_path(x0 * N0, T)))
x_path = np.vstack(tuple(lm.simulate_rate_path(x0, T)))
print(f"New Steady State: {xbar}")
```

```
New Steady State: [0.11309295 0.88690705]
```

Now plot stocks

```
fig, axes = plt.subplots(3, 1, figsize=[10, 9])

axes[0].plot(X_path[:, 0])
axes[0].set_title('Unemployment')
```

(continues on next page)

(continued from previous page)

```
axes[1].plot(X_path[:, 1])
axes[1].set_title('Employment')

axes[2].plot(X_path.sum(1))
axes[2].set_title('Labor force')

for ax in axes:
    ax.grid()

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):
```

(continues on next page)

(continued from previous page)

```
    axes[i].plot(x_path[:, i])
    axes[i].hlines(xbar[i], 0, T, 'r', '--')
    axes[i].set_title(title)
    axes[i].grid()

plt.tight_layout()
plt.show()
```

We see that it takes 20 periods for the economy to converge to its new steady state levels.

Exercise 69.6.3

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 to Exercise 69.6.3

This next 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

```
lm = LakeModelModified()
x0 = lm.rate_steady_state()
```

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

```
lm.b = b_hat
# Simulate stocks
X_path1 = np.vstack(tuple(lm.simulate_stock_path(x0 * N0, T_hat)))
# Simulate rates
x_path1 = np.vstack(tuple(lm.simulate_rate_path(x0, T_hat)))
```

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

```
lm.b = 0.0124
# Simulate stocks
X_path2 = np.vstack(tuple(lm.simulate_stock_path(X_path1[-1, :2], T-T_hat+1)))
# Simulate rates
x_path2 = np.vstack(tuple(lm.simulate_rate_path(x_path1[-1, :2], T-T_hat+1)))
```

Finally, we combine these two paths and plot

```
# note [1:] to avoid doubling period 20
x_path = np.vstack([x_path1, x_path2[1:]])
X_path = np.vstack([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(1))
axes[2].set_title('Labor force')

for ax in axes:
    ax.grid()

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, 'r', '--')
    axes[i].set_title(title)
    axes[i].grid()

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

SEVENTY

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
```

70.1 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 [LP71].

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
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
```

We'll also use the LQ class from `QuantEcon.py`.

```
from quantecon import LQ
```

70.1.1 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

$$p = a_0 - a_1 Y \tag{70.1}$$

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 (70.1), we can express the problem of the representative firm as

$$\max_y \Big[(a_0 - a_1 Y) y - c_1 y - 0.5 c_2 y^2 \Big] \tag{70.2}$$

In posing problem (70.2), 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 (70.2).

This assures that the firm is a price taker.

The first-order condition for problem (70.2) is

$$a_0 - a_1 Y - c_1 - c_2 y = 0 \tag{70.3}$$

At this point, *but not before*, we substitute $Y = y$ into (70.3) to obtain the following linear equation

$$a_0 - c_1 - (a_1 + c_2) Y = 0 \tag{70.4}$$

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 (70.1).

70.1.2 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 (70.4).

Thus, a Y that solves (70.4) 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.

70.1.3 Further Reading

References for this lecture include

- [LP71]
- [Sar87], chapter XIV
- [LS18], chapter 7

70.2 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.

70.2.1 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

$$p_t = a_0 - a_1 Y_t \tag{70.5}$$

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

$$\sum_{t=0}^{\infty} \beta^t r_t \tag{70.6}$$

where

$$r_t := p_t y_t - \frac{\gamma(y_{t+1} - y_t)^2}{2}, \qquad y_0 \text{ given} \tag{70.7}$$

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 (70.5), 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

$$Y_{t+1} = H(Y_t) \tag{70.8}$$

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 (70.8) 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

$$v(y, Y) = \max_{y'} \left\{ a_0 y - a_1 y Y - \frac{\gamma(y' - y)^2}{2} + \beta v(y', H(Y)) \right\} \tag{70.9}$$

Let's denote the firm's optimal policy function by h, so that

$$y_{t+1} = h(y_t, Y_t) \tag{70.10}$$

where

$$h(y, Y) := \text{argmax}_{y'} \left\{ a_0 y - a_1 y Y - \frac{\gamma(y' - y)^2}{2} + \beta v(y', H(Y)) \right\} \tag{70.11}$$

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

$$-\gamma(y' - y) + \beta v_y(y', H(Y)) = 0 \tag{70.12}$$

An important useful envelope result of Benveniste-Scheinkman [BS79] implies that to differentiate v with respect to y we can naively differentiate the right side of (70.9), giving

$$v_y(y, Y) = a_0 - a_1 Y + \gamma(y' - y)$$

Substituting this equation into (70.12) gives the *Euler equation*

$$-\gamma(y_{t+1} - y_t) + \beta[a_0 - a_1 Y_{t+1} + \gamma(y_{t+2} - y_{t+1})] = 0 \tag{70.13}$$

The firm optimally sets an output path that satisfies (70.13), taking (70.8) 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 (70.13) subject to the given initial condition y_0 and the transversality condition.

Note that solving the Bellman equation (70.9) for v and then h in (70.11) yields a decision rule that automatically imposes both the Euler equation (70.13) 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

$$Y_{t+1} = h(Y_t, Y_t) \tag{70.14}$$

Thus, when firms believe that the law of motion for market-wide output is (70.8), their optimizing behavior makes the actual law of motion be (70.14).

70.2.2 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 (70.8) and (70.14).

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 (70.9)–(70.11), the second of which maps a decision rule into an actual law via (70.14).

The H component of a rational expectations equilibrium is a fixed point of Φ.

70.3 Computing an Equilibrium

Now let's compute a rational expectations equilibrium.

70.3.1 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, [MCWG95]).

Lucas and Prescott [LP71] used this method to construct a rational expectations equilibrium.

Some details follow.

70.3.2 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

$$s(Y_t, Y_{t+1}) := \int_0^{Y_t} (a_0 - a_1 x)\, dx - \frac{\gamma(Y_{t+1} - Y_t)^2}{2} \tag{70.15}$$

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 [MS89] and [EH01] 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.

70.3.3 Solution of Planning Problem

Evaluating the integral in (70.15) 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

$$V(Y) = \max_{Y'} \left\{ a_0 Y - \frac{a_1}{2} Y^2 - \frac{\gamma (Y' - Y)^2}{2} + \beta V(Y') \right\} \tag{70.16}$$

The associated first-order condition is

$$-\gamma (Y' - Y) + \beta V'(Y') = 0 \tag{70.17}$$

Applying the same Benveniste-Scheinkman formula gives

$$V'(Y) = a_0 - a_1 Y + \gamma (Y' - Y)$$

Substituting this into equation (70.17) and rearranging leads to the Euler equation

$$\beta a_0 + \gamma Y_t - [\beta a_1 + \gamma (1 + \beta)] Y_{t+1} + \gamma \beta Y_{t+2} = 0 \tag{70.18}$$

70.3.4 Key Insight

Return to equation (70.13) and set $y_t = Y_t$ for all t.

A small amount of algebra will convince you that when $y_t = Y_t$, equations (70.18) and (70.13) 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 (70.16).

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

$$Y_{t+1} = \kappa_0 + \kappa_1 Y_t \tag{70.19}$$

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 (70.9) 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

$$y_{t+1} = h_0 + h_1 y_t + h_2 Y_t \tag{70.20}$$

Hence a rational expectations equilibrium will be defined by the parameters $(\kappa_0, \kappa_1, h_0, h_1, h_2)$ in (70.19)–(70.20).

70.4 Exercises

Exercise 70.4.1

Consider the firm problem *described above*.

Let the firm's belief function H be as given in (70.19).

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 (70.20) and give the values for each h_j.

If there were a unit measure of identical competitive firms all behaving according to (70.20), what would (70.20) imply for the *actual* law of motion (70.8) for market supply.

Solution to Exercise 70.4.1

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 70.4.2

Consider the following κ_0, κ_1 pairs as candidates for the aggregate law of motion component of a rational expectations equilibrium (see (70.19)).

Extending the program that you wrote for Exercise 70.4.1, 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 70.4.1 to compute a rational expectations equilibrium.

(You are not being asked actually to use the algorithm you are suggesting)

Solution to Exercise 70.4.2

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 70.4.3

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 to Exercise 70.4.3

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

- $x_t' R x_t + u_t' Q u_t = -s(Y_t, Y_{t+1})$
- $x_{t+1} = A x_t + B u_t$

By obtaining the optimal policy and using $u_t = -F x_t$ 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
```

(continues on next page)

(continued from previous page)

```
# 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]
print(κ0, κ1)
```

```
95.08187459215002 0.9524590627039248
```

The output yields the same (κ_0, κ_1) pair obtained as an equilibrium from the previous exercise.

Exercise 70.4.4

A monopolist faces the industry demand curve (70.5) 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 70.4.2.

In particular, solve for the parameters in

$$Y_{t+1} = m_0 + m_1 Y_t$$

Compare your results with Exercise 70.4.2 – comment.

Solution to Exercise 70.4.4

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)
```

```
73.47294403502818 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

SEVENTYONE

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
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
import quantecon as qe
from sympy import init_printing, symbols, Matrix
init_printing()
```

71.1 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 [Cag56] linking the price level to the money supply.

Cagan did not use a rational expectations version of his model, but Sargent [Sar77] 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) [BK80] and Whiteman (1983) [Whi83] 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.

71.2 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.

71.2.1 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

$$(1 - \lambda L)y_t = u_t, \forall t \tag{71.1}$$

has solutions

$$y_t = (1 - \lambda L)^{-1}u_t + k\lambda^t \tag{71.2}$$

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 (71.2) 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 (71.1) as

$$y_{t-1} = \lambda^{-1}y_t - \lambda^{-1}u_t, \forall t \tag{71.3}$$

or

$$(1 - \lambda^{-1}L^{-1})y_t = -\lambda^{-1}u_{t+1}. \tag{71.4}$$

A solution is

$$y_t = -\lambda^{-1}\left(\frac{1}{1 - \lambda^{-1}L^{-1}}\right)u_{t+1} + k\lambda^t \tag{71.5}$$

for any k.

To verify that this is a solution, check the consequences of operating on both sides of equation (71.5) by $(1 - \lambda L)$ and compare to equation (71.1).

For any bounded $\{u_t\}$ sequence, solution (71.2) exists for $|\lambda| < 1$ because the **distributed lag** in u converges.

Solution (71.5) exists when $|\lambda| > 1$ because the **distributed lead** in u converges.

When $|\lambda| > 1$, the distributed lag in u in (71.2) may diverge, in which case a solution of this form does not exist.

The distributed lead in u in (71.5) need not converge when $|\lambda| < 1$.

71.2.2 Second Order

Now consider the second order difference equation

$$(1 - \lambda_1 L)(1 - \lambda_2 L)y_{t+1} = u_t \tag{71.6}$$

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 (71.6). Using insights from our analysis of the first-order equation, operate on both sides of (71.6) by the forward inverse of $(1 - \lambda_2 L)$ to rewrite equation (71.6) as

$$(1 - \lambda_1 L) y_{t+1} = -\frac{\lambda_2^{-1}}{1 - \lambda_2^{-1} L^{-1}} u_{t+1}$$

or

$$y_{t+1} = \lambda_1 y_t - \lambda_2^{-1} \sum_{j=0}^{\infty} \lambda_2^{-j} u_{t+j+1}. \tag{71.7}$$

Thus, we obtained equation (71.7) by solving a stable root (in this case λ_1) **backward**, and an unstable root (in this case λ_2) **forward**.

Equation (71.7) 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**

71.3 Illustration: Cagan's Model

Now let's use linear difference equations to represent and solve Sargent's [Sar77] rational expectations version of Cagan's model [Cag56] 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 [Sar77]

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

$$p_t = (1 - \lambda) m_t + \lambda p_{t+1} \tag{71.8}$$

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 (71.8) forward gives

$$p_t = (1 - \lambda) \sum_{j=0}^{\infty} \lambda^j m_{t+j}, \tag{71.9}$$

which is the unique **stable** solution of difference equation (71.8) among a class of more general solutions

$$p_t = (1-\lambda)\sum_{j=0}^{\infty}\lambda^j m_{t+j} + c\lambda^{-t} \tag{71.10}$$

that is indexed by the real number $c \in \mathbf{R}$.

Because we want to focus on stable solutions, we set $c = 0$.

Equation (71.10) 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

$$\begin{aligned} m_t &= Gx_t \\ x_{t+1} &= Ax_t \end{aligned} \tag{71.11}$$

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 (71.11) 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 (71.8) that obeys the system comprised of (71.8)-(71.11).

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 (71.10) above by setting $c = 0$

The solution we are after is

$$p_t = Fx_t \tag{71.12}$$

where

$$F = (1-\lambda)G(I - \lambda A)^{-1} \tag{71.13}$$

Note: As mentioned above, an *explosive solution* of difference equation (71.8) can be constructed by adding to the right hand of (71.12) a sequence $c\lambda^{-t}$ where c is an arbitrary positive constant.

71.4 Some Python Code

We'll construct examples that illustrate (71.11).

Our first example takes as the law of motion for the log money supply the second order difference equation

$$m_{t+1} = \alpha + \rho_1 m_t + \rho_2 m_{t-1} \tag{71.14}$$

that is parameterized by ρ_1, ρ_2, α

To capture this parameterization with system (71.9) 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()
```

```
True
```

Now let's compute F in formulas (71.12) and (71.13).

```
# 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='$m_t$')
plt.plot(range(T+1), p_seq, label='$p_t$')
plt.xlabel('t')
plt.title(f'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 (71.9), 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

71.5 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
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()
```

71.5.1 Special Case

To simplify our presentation in ways that will let focus on an important idea, in the above second-order difference equation (71.14) 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

$$m_{t+1} = \rho m_t \tag{71.15}$$

and the state x_t becomes

$$x_t = m_t.$$

Consequently, we can set $G = 1, A = \rho$ making our formula (71.13) for F become

$$F = (1 - \lambda)(1 - \lambda\rho)^{-1}.$$

so that the log the log price level satisfies

$$p_t = F m_t.$$

Please keep these formulas in mind as we investigate an alternative route to and interpretation of our formula for F.

71.6 Another Perspective

Above, we imposed stability or non-explosiveness on the solution of the key difference equation (71.8) 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 (71.8) and (71.15) to form the system

$$\begin{bmatrix} m_{t+1} \\ p_{t+1} \end{bmatrix} = \begin{bmatrix} \rho & 0 \\ -(1-\lambda)/\lambda & \lambda^{-1} \end{bmatrix} \begin{bmatrix} m_t \\ p_t \end{bmatrix} \tag{71.16}$$

or

$$y_{t+1} = H y_t, \quad t \geq 0 \tag{71.17}$$

where

$$H = \begin{bmatrix} \rho & 0 \\ -(1-\lambda)/\lambda & \lambda^{-1} \end{bmatrix}. \tag{71.18}$$

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 (71.17) 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

$$y_0^* = \begin{bmatrix} y_{1,0}^* \\ 0 \end{bmatrix} \tag{71.19}$$

the path of y_t^* and therefore the paths of both components of $y_t = Qy_t^*$ will diverge in absolute value as $t \to +\infty$. (We say that the paths *explode*)

Equation (71.19) 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

$$Qy_0 = y_0^* = \begin{bmatrix} y_{1,0}^* \\ 0 \end{bmatrix}.$$

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 (71.19) 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

$$Q^{21} m_0 + Q^{22} p_0 = 0 \tag{71.20}$$

where Q^{ij} denotes the (i, j) component of Q^{-1}.

Solving this equation for p_0, we find

$$p_0 = -(Q^{22})^{-1} Q^{21} m_0. \tag{71.21}$$

This is the unique **stabilizing value** of p_0 expressed as a function of m_0.

71.6.1 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

$$p_0 = Q_{21}Q_{11}^{-1}m_0. \tag{71.22}$$

It can be verified that this formula replicates itself over time in the sense that

$$p_t = Q_{21}Q_{11}^{-1}m_t. \tag{71.23}$$

To implement formula (71.23), 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

$$Q_1 = \begin{bmatrix} Q_{11} \\ Q_{21} \end{bmatrix}.$$

71.6.2 Remarks about Feedback

We have expressed (71.16) 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}, $j = 0, 1, 2, \ldots$.

A tell-tale sign that we should look beyond its superficial "feedback" form is that $\lambda^{-1} > 1$ so that the matrix H in (71.16) 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.

71.7 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

$$m_{t+1} = \rho m_t + \delta p_t \tag{71.24}$$

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 (71.8) continues to describe equality between the demand and supply of money.

We assume that equations (71.8) and (71.24) 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:
```

(continues on next page)

(continued from previous page)

```
        print("both eigenvalues exceed unity in modulus")

        return None

    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("$m_0$")
plt.ylabel("$p_0$")
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('$\delta$')
plt.ylabel('$p_0$')
plt.title(f'$m_0$={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
```

71.8 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 (71.9) 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

$$x_{t+1} = A x_t$$

where $x_t = \begin{bmatrix} m_t \\ P_t \end{bmatrix}$.

Apply formula (71.13) 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 (71.11), (71.12), and (71.13) 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 (71.13) 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
```

$$0.950124378879109$$

```
# 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
```

$$(0.95012437887911,\ 0.950124378879109)$$

71.9 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()
```

$$\left\{ \frac{\lambda\rho+1}{2\lambda} - \frac{\sqrt{4\delta\lambda^2-4\delta\lambda+\lambda^2\rho^2-2\lambda\rho+1}}{2\lambda} : 1,\ \frac{\lambda\rho+1}{2\lambda} + \frac{\sqrt{4\delta\lambda^2-4\delta\lambda+\lambda^2\rho^2-2\lambda\rho+1}}{2\lambda} : 1 \right\}$$

```
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()
```

$$\left\{ \frac{1}{\lambda} : 1,\ \rho : 1 \right\}$$

```
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

SEVENTYTWO

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
```

72.1 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 [LS18].

Let's start with some standard imports:

```
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
import quantecon as qe
```

72.2 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., [EP95], [Rya12], [DS10]).
- Rate of extraction from a shared natural resource, such as a fishery (e.g., [LM80], [VL11]).

Let's examine a model of the first type.

72.2.1 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

$$p = a_0 - a_1(q_1 + q_2) \tag{72.1}$$

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 (72.1) 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:

$$\pi_i = pq_i - \gamma(\hat{q}_i - q_i)^2, \quad \gamma > 0, \tag{72.2}$$

Substituting the inverse demand curve (72.1) into (72.2) lets us express the one-period payoff as

$$\pi_i(q_i, q_{-i}, \hat{q}_i) = a_0 q_i - a_1 q_i^2 - a_1 q_i q_{-i} - \gamma(\hat{q}_i - q_i)^2, \tag{72.3}$$

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

$$v_i(q_i, q_{-i}) = \max_{\hat{q}_i} \{\pi_i(q_i, q_{-i}, \hat{q}_i) + \beta v_i(\hat{q}_i, f_{-i}(q_{-i}, q_i))\} \tag{72.4}$$

Definition 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 (72.4).
- The maximizer on the right side of (72.4) 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.

72.2.2 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

$$v_i^{j+1}(q_i, q_{-i}) = \max_{\hat{q}_i} \left\{ \pi_i(q_i, q_{-i}, \hat{q}_i) + \beta v_i^j(\hat{q}_i, f_{-i}(q_{-i}, q_i)) \right\} \tag{72.5}$$

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.

72.3 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.

72.3.1 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

$$\sum_{t=t_0}^{t_1-1} \beta^{t-t_0} \{x_t' R_i x_t + u_{it}' Q_i u_{it} + u_{-it}' S_i u_{-it} + 2x_t' W_i u_{it} + 2u_{-it}' M_i u_{it}\} \tag{72.6}$$

while the state evolves according to

$$x_{t+1} = Ax_t + B_1 u_{1t} + B_2 u_{2t} \tag{72.7}$$

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 $n \times k_i$

72.3.2 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 (72.6) and (72.7), then player 1's problem becomes minimization of

$$\sum_{t=t_0}^{t_1-1} \beta^{t-t_0} \left\{ x_t' \Pi_{1t} x_t + u_{1t}' Q_1 u_{1t} + 2 u_{1t}' \Gamma_{1t} x_t \right\} \tag{72.8}$$

subject to

$$x_{t+1} = \Lambda_{1t} x_t + B_1 u_{1t}, \tag{72.9}$$

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

$$F_{1t} = (Q_1 + \beta B_1' P_{1t+1} B_1)^{-1} (\beta B_1' P_{1t+1} \Lambda_{1t} + \Gamma_{1t}) \tag{72.10}$$

where P_{1t} solves the matrix Riccati difference equation

$$P_{1t} = \Pi_{1t} - (\beta B_1' P_{1t+1} \Lambda_{1t} + \Gamma_{1t})' (Q_1 + \beta B_1' P_{1t+1} B_1)^{-1} (\beta B_1' P_{1t+1} \Lambda_{1t} + \Gamma_{1t}) + \beta \Lambda_{1t}' P_{1t+1} \Lambda_{1t} \tag{72.11}$$

Similarly, decision rules that solve player 2's problem are

$$F_{2t} = (Q_2 + \beta B_2' P_{2t+1} B_2)^{-1} (\beta B_2' P_{2t+1} \Lambda_{2t} + \Gamma_{2t}) \tag{72.12}$$

where P_{2t} solves

$$P_{2t} = \Pi_{2t} - (\beta B_2' P_{2t+1} \Lambda_{2t} + \Gamma_{2t})' (Q_2 + \beta B_2' P_{2t+1} B_2)^{-1} (\beta B_2' P_{2t+1} \Lambda_{2t} + \Gamma_{2t}) + \beta \Lambda_{2t}' P_{2t+1} \Lambda_{2t} \tag{72.13}$$

Here, in all cases $t = t_0, \dots, t_1 - 1$ and the terminal conditions are $P_{it_1} = 0$.

The solution procedure is to use equations (72.10), (72.11), (72.12), and (72.13), 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 (72.10) contain F_{2t}
- some terms on the right-hand side of (72.12) contain F_{1t}

we need to solve these $k_1 + k_2$ equations simultaneously.

Key Insight

A key insight is that equations (72.10) and (72.12) 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 (72.11) and (72.13).

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.

72.3.3 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.

72.4 Application

Let's use these procedures to treat some applications, starting with the duopoly model.

72.4.1 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 (72.3).

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:

$$x_{t+1} = (A - B_1 F_1 - B_1 F_2) x_t \tag{72.14}$$

72.4.2 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
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)
```

(continues on next page)

(continued from previous page)

```
# 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 (72.8) and (72.9) to get firm 1's problem and solve it using LQ.

We hope that the resulting policy will agree with F1 as computed above

```
Λ1 = A - B2 @ F2
lq1 = qe.LQ(Q1, R1, Λ1, B1, beta=β)
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
```

72.4.3 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 (72.14).

The following program

- imports $F1$ and $F2$ from the previous program along with all parameters.
- computes the evolution of x_t using (72.14).
- 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]
```

(continues on next page)

(continued from previous page)

```
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.

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

72.5 Exercises

Exercise 72.5.1

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 to Exercise 72.5.1

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 $x_{t+1} = A x_t + B u_t$.

We solve for the optimal policy $u_t = -F x_t$ 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) + 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 72.5.2

In this exercise, we consider a slightly more sophisticated duopoly problem.

It takes the form of infinite horizon linear-quadratic game proposed by Judd [Jud90].

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 to Exercise 72.5.2

We treat the case $\delta = 0.02$

$\delta = 0.02$
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

$\delta = 0.05$
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

```
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,
```

(continues on next page)

(continued from previous page)

```
                            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

SEVENTYTHREE

UNCERTAINTY TRAPS

Contents

- *Uncertainty Traps*
  - *Overview*
  - *The Model*
  - *Implementation*
  - *Results*
  - *Exercises*

73.1 Overview

In this lecture, we study a simplified version of an uncertainty traps model of Fajgelbaum, Schaal and Taschereau-Dumouchel [FSTD15].

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
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
```

73.2 The Model

The original model described in [FSTD15] has many interesting moving parts.

Here we examine a simplified version that nonetheless captures many of the key ideas.

73.2.1 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.

73.2.2 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

$$x_m = \theta + \epsilon_m \quad \text{where} \quad \epsilon_m \sim N\left(0, \gamma_x^{-1}\right) \tag{73.1}$$

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.

73.2.3 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

- $\mathbb{M} \subset \{1, \dots, \bar{M}\}$ 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

$$\mu' = \rho \frac{\gamma\mu + M\gamma_x X}{\gamma + M\gamma_x} \tag{73.2}$$

$$\gamma' = \left(\frac{\rho^2}{\gamma + M\gamma_x} + \sigma_\theta^2\right)^{-1} \tag{73.3}$$

These are standard Kalman filtering results applied to the current setting.

Exercise 1 provides more details on how (73.2) and (73.3) are derived and then asks you to fill in remaining steps.

The next figure plots the law of motion for the precision in (73.3) as a 45 degree diagram, with one curve for each $M \in \{0, \dots, 6\}$.

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.

73.2.4 Participation

Omitting time subscripts once more, entrepreneurs enter the market in the current period if

$$\mathbb{E}[u(x_m - F_m)] > c \tag{73.4}$$

Here

- the mathematical expectation of x_m is based on (73.1) 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

3.0
2.5
2.0
1.5
1.0
0.5
0.0
γ'
0.0
0.5
1.0
1.5
2.0
2.5
3.0
γ
$M = 0$
$M = 1$
$M = 2$
$M = 3$
$M = 4$
$M = 5$
$M = 6$

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 (73.4).

The utility function has the constant absolute risk aversion form

$$u(x) = \frac{1}{a}\left(1 - \exp(-ax)\right) \tag{73.5}$$

where a is a positive parameter.

Combining (73.4) and (73.5), 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

$$\psi(\mu, \gamma, F_m) := \frac{1}{a}\left(1 - \exp\left(-a\mu + aF_m + \frac{a^2\left(\frac{1}{\gamma} + \frac{1}{\gamma_x}\right)}{2}\right)\right) - c > 0 \tag{73.6}$$

73.3 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 $F_1, \dots, F_{\bar{M}}$ and tests condition (73.6) 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)
```

(continues on next page)

(continued from previous page)

```
        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.
        """
        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.

73.4 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.

73.5 Exercises

Exercise 73.5.1

Fill in the details behind (73.2) and (73.3) based on the following standard result (see, e.g., p. 24 of [YS05]).

Fact Let $\mathbf{x} = (x_1, \dots, x_M)$ 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 to Exercise 73.5.1

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

$$N(\mu_0, 1/\gamma_0)$$

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

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 73.5.2

Modulo randomness, replicate the simulation figures shown above.

- Use the parameter values listed as defaults in the **init** method of the UncertaintyTrapEcon class.

Solution to Exercise 73.5.2

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
```

(continues on next page)

(continued from previous page)

```
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
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
0
250
500
750
1000
1250
1500
1750
2000

time series for μ
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
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

SEVENTYFOUR

THE AIYAGARI MODEL

Contents

- *The Aiyagari Model*
  - *Overview*
  - *The Economy*
  - *Firms*
  - *Code*

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install quantecon
```

74.1 Overview

In this lecture, we describe the structure of a class of models that build on work by Truman Bewley [Bew77].

We begin by discussing an example of a Bewley model due to Rao Aiyagari [Aiy94].

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 [Aiy94]
- risk sharing and asset pricing [HL96]
- the shape of the wealth distribution [BBZ15]
- etc., etc., etc.

Let's start with some imports:

```
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
from quantecon.markov import DiscreteDP
from numba import jit
```

74.1.1 References

The primary reference for this lecture is [Aiy94].

A textbook treatment is available in chapter 18 of [LS18].

A continuous time version of the model by SeHyoun Ahn and Benjamin Moll can be found here.

74.2 The Economy

74.2.1 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

$$a_{t+1} + c_t \leq w z_t + (1+r) a_t \quad c_t \geq 0, \quad \text{and} \quad a_t \geq -B$$

where

- c_t is current consumption
- a_t is assets
- z_t is an exogenous component of labor income 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.

74.3 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

$$Y_t = AK_t^\alpha N^{1-\alpha}$$

where

- A and α are parameters with $A > 0$ and $\alpha \in (0, 1)$
- K_t is aggregate capital
- N is total labor supply (which is constant in this simple version of the model)

The firm's problem is

$$max_{K,N} \left\{ AK_t^\alpha N^{1-\alpha} - (r+\delta)K - wN \right\}$$

The parameter δ is the depreciation rate.

From the first-order condition with respect to capital, the firm's inverse demand for capital is

$$r = A\alpha \left(\frac{N}{K}\right)^{1-\alpha} - \delta \tag{74.1}$$

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

$$w(r) = A(1-\alpha)(A\alpha/(r+\delta))^{\alpha/(1-\alpha)} \tag{74.2}$$

74.3.1 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

In practice, once parameter values are set, we can check for an SREE by the following steps

1. pick a proposed quantity K for aggregate capital

2. determine corresponding prices, with interest rate r determined by (74.1) and a wage rate $w(r)$ as given in (74.2)
3. determine the common optimal savings policy of the households given these prices
4. compute aggregate capital as the mean of steady state capital given this savings policy

If this final quantity agrees with K then we have a SREE.

74.4 Code

Let's look at how we might compute such an equilibrium in practice.

To solve the household's dynamic programming problem we'll use the DiscreteDP class from QuantEcon.py.

Our first task is the least exciting one: write code that maps parameters for a household problem into the `R` and `Q` matrices needed to generate an instance of `DiscreteDP`.

Below is a piece of boilerplate code that does just this.

In reading the code, the following information will be helpful

- `R` needs to be a matrix where `R[s, a]` is the reward at state `s` under action `a`.
- `Q` needs to be a three-dimensional array where `Q[s, a, s']` is the probability of transitioning to state `s'` when the current state is `s` and the current action is `a`.

(A more detailed discussion of `DiscreteDP` is available in the Discrete State Dynamic Programming lecture in the Advanced Quantitative Economics with Python lecture series.)

Here we take the state to be $s_t := (a_t, z_t)$, where a_t is assets and z_t is the shock.

The action is the choice of next period asset level a_{t+1}.

We use Numba to speed up the loops so we can update the matrices efficiently when the parameters change.

The class also includes a default set of parameters that we'll adopt unless otherwise specified.

```
class Household:
    """
    This class takes the parameters that define a household asset accumulation
    problem and computes the corresponding reward and transition matrices R
    and Q required to generate an instance of DiscreteDP, and thereby solve
    for the optimal policy.

    Comments on indexing: We need to enumerate the state space S as a sequence
    S = {0, ..., n}.  To this end, (a_i, z_i) index pairs are mapped to s_i
    indices according to the rule

        s_i = a_i * z_size + z_i

    To invert this map, use

        a_i = s_i // z_size  (integer division)
        z_i = s_i % z_size

    """


    def __init__(self,
                 r=0.01,                        # Interest rate
```

(continues on next page)

(continued from previous page)

```
                w=1.0,                          # Wages
                β=0.96,                         # Discount factor
                a_min=1e-10,
                Π=[[0.9, 0.1], [0.1, 0.9]],     # Markov chain
                z_vals=[0.1, 1.0],              # Exogenous states
                a_max=18,
                a_size=200):

        # Store values, set up grids over a and z
        self.r, self.w, self.β = r, w, β
        self.a_min, self.a_max, self.a_size = a_min, a_max, a_size

        self.Π = np.asarray(Π)
        self.z_vals = np.asarray(z_vals)
        self.z_size = len(z_vals)

        self.a_vals = np.linspace(a_min, a_max, a_size)
        self.n = a_size * self.z_size

        # Build the array Q
        self.Q = np.zeros((self.n, a_size, self.n))
        self.build_Q()

        # Build the array R
        self.R = np.empty((self.n, a_size))
        self.build_R()

    def set_prices(self, r, w):
        """
        Use this method to reset prices. Calling the method will trigger a
        re-build of R.
        """
        self.r, self.w = r, w
        self.build_R()

    def build_Q(self):
        populate_Q(self.Q, self.a_size, self.z_size, self.Π)

    def build_R(self):
        self.R.fill(-np.inf)
        populate_R(self.R,
                self.a_size,
                self.z_size,
                self.a_vals,
                self.z_vals,
                self.r,
                self.w)


# Do the hard work using JIT-ed functions

@jit(nopython=True)
def populate_R(R, a_size, z_size, a_vals, z_vals, r, w):
    n = a_size * z_size
    for s_i in range(n):
        a_i = s_i // z_size
```

(continues on next page)

(continued from previous page)

```
        z_i = s_i % z_size
        a = a_vals[a_i]
        z = z_vals[z_i]
        for new_a_i in range(a_size):
            a_new = a_vals[new_a_i]
            c = w * z + (1 + r) * a - a_new
            if c > 0:
                R[s_i, new_a_i] = np.log(c)  # Utility

@jit(nopython=True)
def populate_Q(Q, a_size, z_size, Π):
    n = a_size * z_size
    for s_i in range(n):
        z_i = s_i % z_size
        for a_i in range(a_size):
            for next_z_i in range(z_size):
                Q[s_i, a_i, a_i*z_size + next_z_i] = Π[z_i, next_z_i]


@jit(nopython=True)
def asset_marginal(s_probs, a_size, z_size):
    a_probs = np.zeros(a_size)
    for a_i in range(a_size):
        for z_i in range(z_size):
            a_probs[a_i] += s_probs[a_i*z_size + z_i]
    return a_probs
```

As a first example of what we can do, let's compute and plot an optimal accumulation policy at fixed prices.

```
# Example prices
r = 0.03
w = 0.956

# Create an instance of Household
am = Household(a_max=20, r=r, w=w)

# Use the instance to build a discrete dynamic program
am_ddp = DiscreteDP(am.R, am.Q, am.β)

# Solve using policy function iteration
results = am_ddp.solve(method='policy_iteration')

# Simplify names
z_size, a_size = am.z_size, am.a_size
z_vals, a_vals = am.z_vals, am.a_vals
n = a_size * z_size

# Get all optimal actions across the set of a indices with z fixed in each row
a_star = np.empty((z_size, a_size))
for s_i in range(n):
    a_i = s_i // z_size
    z_i = s_i % z_size
    a_star[z_i, a_i] = a_vals[results.sigma[s_i]]

fig, ax = plt.subplots(figsize=(9, 9))
ax.plot(a_vals, a_vals, 'k--')  # 45 degrees
```

(continues on next page)

(continued from previous page)

```
for i in range(z_size):
    lb = f'$z = {z_vals[i]:.2}$'
    ax.plot(a_vals, a_star[i, :], 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.

Now we want to calculate the equilibrium.

Let's do this visually as a first pass.

The following code draws aggregate supply and demand curves.

The intersection gives equilibrium interest rates and capital.

```
A = 1.0
N = 1.0
α = 0.33
β = 0.96
δ = 0.05


def r_to_w(r):
    """
    Equilibrium wages associated with a given interest rate r.
    """
    return A * (1 - α) * (A * α / (r + δ))**(α / (1 - α))

def rd(K):
    """
    Inverse demand curve for capital.  The interest rate associated with a
    given demand for capital K.
    """
    return A * α * (N / K)**(1 - α) - δ


def prices_to_capital_stock(am, r):
    """
    Map prices to the induced level of capital stock.

    Parameters:
    ----------

    am : Household
        An instance of an aiyagari_household.Household
    r : float
        The interest rate
    """
    w = r_to_w(r)
    am.set_prices(r, w)
    aiyagari_ddp = DiscreteDP(am.R, am.Q, β)
    # Compute the optimal policy
    results = aiyagari_ddp.solve(method='policy_iteration')
    # Compute the stationary distribution
    stationary_probs = results.mc.stationary_distributions[0]
    # Extract the marginal distribution for assets
    asset_probs = asset_marginal(stationary_probs, am.a_size, am.z_size)
    # Return K
    return np.sum(asset_probs * am.a_vals)


# Create an instance of Household
am = Household(a_max=20)

# Use the instance to build a discrete dynamic program
am_ddp = DiscreteDP(am.R, am.Q, am.β)

# Create a grid of r values at which to compute demand and supply of capital
num_points = 20
r_vals = np.linspace(0.005, 0.04, num_points)

# Compute supply of capital
```

(continues on next page)

(continued from previous page)

```
k_vals = np.empty(num_points)
for i, r in enumerate(r_vals):
    k_vals[i] = prices_to_capital_stock(am, r)

# 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, rd(k_vals), lw=2, alpha=0.6, label='demand for capital')
ax.grid()
ax.set_xlabel('capital')
ax.set_ylabel('interest rate')
ax.legend(loc='upper right')

plt.show()
```

Part XI

Asset Pricing and Finance

CHAPTER

SEVENTYFIVE

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
```

75.1 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
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
import quantecon as qe
from numpy.linalg import eigvals, solve
```

75.2 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 $d_t, d_{t+1}, \ldots$.
- A time-t **ex-dividend** asset is a claim to the stream $d_{t+1}, d_{t+2}, \ldots$.

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).

75.2.1 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

$$p_t = \beta \mathbb{E}_t[d_{t+1} + p_{t+1}] \tag{75.1}$$

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.

75.2.2 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 [HK79] and Lars Peter Hansen and Scott Richard [HR87] showed that in quite general settings the price of an ex-dividend asset obeys

$$p_t = \mathbb{E}_t \left[m_{t+1}(d_{t+1} + p_{t+1}) \right] \tag{75.2}$$

for some **stochastic discount factor** m_{t+1}.

Here the fixed discount factor β in (75.1) 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.

75.2.3 Asset Pricing and Covariances

Recall that, from the definition of a conditional covariance $\text{cov}_t(x_{t+1}, y_{t+1})$, we have

$$\mathbb{E}_t(x_{t+1}y_{t+1}) = \text{cov}_t(x_{t+1}, y_{t+1}) + \mathbb{E}_t x_{t+1}\mathbb{E}_t y_{t+1} \tag{75.3}$$

If we apply this definition to the asset pricing equation (75.2) we obtain

$$p_t = \mathbb{E}_t m_{t+1}\mathbb{E}_t(d_{t+1} + p_{t+1}) + \text{cov}_t(m_{t+1}, d_{t+1} + p_{t+1}) \tag{75.4}$$

It is useful to regard equation (75.4) as a generalization of equation (75.1)

- In equation (75.1), the stochastic discount factor $m_{t+1} = \beta$, a constant.
- In equation (75.1), the covariance term $\text{cov}_t(m_{t+1}, d_{t+1} + p_{t+1})$ is zero because $m_{t+1} = \beta$.
- In equation (75.1), $\mathbb{E}_t m_{t+1}$ 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 (75.4) 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.

75.2.4 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 (75.2) by d_t to get

$$v_t = \mathbb{E}_t \left[m_{t+1} \frac{d_{t+1}}{d_t}(1 + v_{t+1}) \right] \tag{75.5}$$

Below we'll discuss the implication of this equation.

75.3 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.

75.3.1 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 (75.1) 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

$$\bar{p} := \frac{\beta d}{1 - \beta} \tag{75.6}$$

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 $p_t = \beta(d + p_{t+1})$.

75.3.2 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 (75.5) 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*.

75.3.3 Example 3: Markov Growth, Risk-Neutral Pricing

Next, we consider a dividend process

$$d_{t+1} = g_{t+1} d_t \tag{75.7}$$

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 (75.5) 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

$$v(x) = \beta \sum_{y \in S} K(x, y)(1 + v(y)) \quad \text{where} \quad K(x, y) := g(y)P(x, y) \tag{75.8}$$

Suppose that there are n possible states $x_1, \dots, x_n$.

We can then think of (75.8) as n stacked equations, one for each state, and write it in matrix form as

$$v = \beta K(\mathbb{1} + v) \tag{75.9}$$

Here

- v is understood to be the column vector $(v(x_1), \dots, v(x_n))'$.
- 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 (75.9) have a unique solution?

From the *Neumann series lemma* and Gelfand's formula, equation (75.9) 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

$$v = (I - \beta K)^{-1} \beta K \mathbb{1} \tag{75.10}$$

75.3.4 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)
```

(continues on next page)

(continued from previous page)

```
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.

75.4 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

75.4.1 Pricing a Lucas Tree

Let's start with a version of the celebrated asset pricing model of Robert E. Lucas, Jr. [Luc78].

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 [Luc78], we suppose that the stochastic discount factor takes the form

$$m_{t+1} = \beta \frac{u'(c_{t+1})}{u'(c_t)} \tag{75.11}$$

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 (75.7).

The asset being priced is a claim on the endowment process, i.e., the *Lucas tree* described above.

Following [Luc78], 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

$$u(c) = \frac{c^{1-\gamma}}{1-\gamma} \text{ with } \gamma > 0 \tag{75.12}$$

When $\gamma = 1$ we let $u(c) = \ln c$.

Inserting the CRRA specification into (75.11) and using $c_t = d_t$ gives

$$m_{t+1} = \beta \left(\frac{c_{t+1}}{c_t} \right)^{-\gamma} = \beta g_{t+1}^{-\gamma} \tag{75.13}$$

Substituting this into (75.5) gives the price-dividend ratio formula

$$v(X_t) = \beta \mathbb{E}_t \left[g(X_{t+1})^{1-\gamma} (1 + v(X_{t+1})) \right] \tag{75.14}$$

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 (75.14) 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

$$v = (I - \beta J)^{-1} \beta J \mathbb{1} \tag{75.15}$$

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 $g(x) = \exp(x)$

```
γs = [1.2, 1.4, 1.6, 1.8, 2.0]
ap = AssetPriceModel()
states = ap.mc.state_values

fig, ax = plt.subplots()

for γ in γs:
    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 (75.13), 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 (75.10).

This is as expected, since $\gamma = 0$ implies $u(c) = c$ (and hence agents are risk-neutral).

75.4.2 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 (75.2) with $d_t = \zeta$, or

$$p_t = \mathbb{E}_t \left[m_{t+1}(\zeta + p_{t+1}) \right]$$

With the stochastic discount factor (75.13), this becomes

$$p_t = \mathbb{E}_t \left[\beta g_{t+1}^{-\gamma} (\zeta + p_{t+1}) \right] \tag{75.16}$$

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

$$p = (I - \beta M)^{-1} \beta M \zeta \mathbb{1} \tag{75.17}$$

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)
    Ones = np.ones(ap.n)
    p = solve(I - β * M, β * ζ * M @ Ones)

    return p
```

75.4.3 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

$$w(x, p_S) = \max \left\{ \beta \sum_{y \in S} P(x,y) g(y)^{-\gamma} w(y, p_S), \; p(x) - p_S \right\} \tag{75.18}$$

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 (75.18) as the nonlinear vector equation

$$w = \max\{\beta M w, \; p - p_S \mathbb{1}\} \tag{75.19}$$

To solve (75.19), 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
```

(continues on next page)

(continued from previous page)

```
    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')
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.

75.4.4 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 $j \geq 1$.

75.5 Exercises

Exercise 75.5.1

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 (75.1), 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 $d_t = gd_t$ where $0 < g\beta < 1$, what is the equilibrium price of a cum-dividend asset?

Solution to Exercise 75.5.1

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 75.5.2

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 to Exercise 75.5.2

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()
```

Exercise 75.5.3

Let's consider finite horizon call options, which are more common than infinite horizon ones.

Finite horizon options obey functional equations closely related to (75.18).

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 75.5.1.

Is one higher than the other? Can you give intuition?

Solution to Exercise 75.5.3

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

SEVENTYSIX

COMPETITIVE EQUILIBRIA WITH ARROW SECURITIES

76.1 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 across 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 equilbrium prices before determining an equilibrium distribution of wealth

Diverse endowments across agents provide motivations for individuals to 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 describe 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

76.2 The setting

In effect, this lecture implements a Python version of the model presented in section 9.3.3 of Ljungqvist and Sargent [LS18].

76.2.1 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 $s^t = [s_0, s_1, \dots, s_{t-1}, s_t]$.

(Sometimes we inadvertently reverse the recording order and denote a history as $s^t = [s_t, s_{t-1}, \dots, s_1, s_0]$.)

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 $k = 1, \dots, K$.

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 i purchases a history-dependent consumption plan $c^k = \{c_t^k(s^t)\}_{t=0}^\infty$

Consumer i 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 pf 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.

76.3 Recursive Formulation

Following descriptions in section 9.3.3 of Ljungqvist and Sargent [LS18] 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 i'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 i's problem starting from state (a,s).

- $v^k(a,s)$ is the maximum expected discounted utility that consumer i 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 $\{v^k(a,s)\}_{i=1}^K$, and decision rules $\{c^k(s), a^k(s)\}_{i=1}^K$ 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 i, 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 $\{\{c_t^k, \{\hat{a}_{t+1}^k(s')\}_{s'}\}_i\}_t$ satisfy $\sum_i c_t^k = \sum_i y^k(s_t)$ and $\sum_i \hat{a}_{t+1}^k(s') = 0$ for all t and s'.
- The initial financial wealth vector $\vec{a}_0$ satisfies $\sum_{i=1}^K a_0^k = 0$.

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.

76.4 State Variable Degeneracy

Please see Ljungqvist and Sargent [LS18] 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 $k = 1, \dots, K$.

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 can 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*

76.5 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: $s \in S = [\bar{s}_1, \dots, \bar{s}_n]$ governed by an n-state Markov chain with transition probability

$$P_{ij} = \text{Pr}\left\{s_{t+1} = \bar{s}_j \mid s_t = \bar{s}_k\right\}$$

- A collection $h = 1, \dots, H$ 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 $R_i = (\sum_j Q_{i,j})^{-1}$

76.5.1 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

$$p^h = (I - Q)^{-1} Q d^h$$

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.

76.5.2 Multi-Step-Forward Transition Probabilities and Pricing Kernels

The (i, j) component of the k-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.

76.5.3 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:

$$\begin{aligned}
V\left[V(d(s_{t+j})|s_{t+1})\right]|s_t &= \sum_{s_{t+1}}\left[\sum_{s_{t+j}} d(s_{t+j})Q_{j-1}(s_{t+j}|s_{t+1})\right]Q(s_{t+1}|s_t) \\
&= \sum_{s_{t+j}} d(s_{t+j})\left[\sum_{s_{t+1}} Q_{j-1}(s_{t+j}|s_{t+1})Q(s_{t+1}|s_t)\right] \\
&= \sum_{s_{t+j}} d(s_{t+j})Q_j(s_{t+j}|s_t) \\
&= EV(d(s_{t+j}))|s_t
\end{aligned}$$

76.6 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.

76.6.1 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

$$u'(c) = c^{-\gamma}$$

76.6.2 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: $c^k(s), k = 1, \ldots, K$

76.6.3 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** 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

$$Q_{ij} = \beta \left(\frac{y_j}{y_i} \right)^{-\gamma} P_{ij} \tag{76.1}$$

Note that Q_{ij} is independent of vector α.

Key finding: We can compute competitive equilibrium **prices** prior to computing a **distribution of wealth**.

76.6.4 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

$$A(s) = \begin{bmatrix} A^1(s) \\ \vdots \\ A^K(s) \end{bmatrix}, \quad s \in [\bar{s}_1, \dots, \bar{s}_n]$$

and an $n \times 1$ vector of continuation endowment values for each individual k as

$$A^k = \begin{bmatrix} A^k(\bar{s}_1) \\ \vdots \\ A^k(\bar{s}_n) \end{bmatrix}, \quad k \in [1, \dots, K]$$

A^k of consumer k satisfies

$$A^k = [I - Q]^{-1} [y^k]$$

where

$$y^k = \begin{bmatrix} y^k(\bar{s}_1) \\ \vdots \\ y^k(\bar{s}_n) \end{bmatrix} \equiv \begin{bmatrix} y_1^k \\ \vdots \\ v_n^k \end{bmatrix}$$

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 below on a Finite Horizon Economy.

76.6.5 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

$$\psi(s) = \begin{bmatrix} \psi^1(s) \\ \vdots \\ \psi^K(s) \end{bmatrix}, \quad s \in [\bar{s}_1, \dots, \bar{s}_n]$$

and an $n \times 1$ vector of continuation wealths for each individual k as

$$\psi^k = \begin{bmatrix} \psi^k(\bar{s}_1) \\ \vdots \\ \psi^k(\bar{s}_n) \end{bmatrix}, \quad k \in [1, \dots, K]$$

Continuation wealth ψ^k of consumer k satisfies

$$\psi^k = [I - Q]^{-1} [\alpha_k y - y^k] \tag{76.2}$$

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 $s_0 \in [\bar{s}_1, \dots, \bar{s}_n]$, the continuation wealth $\psi^k(s_0) = 0$ for all agents $k = 1, \dots, K$. 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.

76.6.6 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

$$a_k(s) = \psi^k(s), \quad s \in [\bar{s}_1, \dots, \bar{s}_n] \tag{76.3}$$

76.6.7 Equilibrium Wealth Distribution α

With the initial state being a particular state $s_0 \in [\bar{s}_1, \dots, \bar{s}_n]$, we must have

$$\psi^k(s_0) = 0, \quad k = 1, \dots, K$$

which means the equilibrium distribution of wealth satisfies

$$\alpha_k = \frac{V_z y^k}{V_z y} \tag{76.4}$$

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 (76.1)
- compute the distribution of wealth α from the formula (76.4)
- Using α assign each consumer k the share α_k of the aggregate endowment at each state
- return to the α-dependent formula (76.2) and compute continuation wealths

- via formula (76.3) 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.

76.7 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 handly infinite-horizon economies, the code is set up to handle finite-horizon economies indexed by horizon T.

We'll study some finite horizon economies after we 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)
```

(continues on next page)

(continued from previous page)

```
        # 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"

        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
```

(continues on next page)

(continued from previous page)

```
        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"

        PRF = np.sum(self.Q, 0)
        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, 0)
        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
```

(continues on next page)

(continued from previous page)

```
        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(α_1 y), ..., u(α_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
```

76.7.1 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]]]
```

76.7.2 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 =  [0.93477529 1.10604104]
```

```
# pricing kernal
ex2.Q
```

```
array([[0.49      , 0.41412558],
       [0.57977582, 0.49      ]])
```

```
# Risk free rate R
ex2.R
```

```
array([0.93477529, 1.10604104])
```

```
# 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]]]
```

76.7.3 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
```

(continues on next page)

(continued from previous page)

```
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 =  [10.20408163  0.53705693]
```

```
# 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.        ]]]
```

(continues on next page)

(continued from previous page)

```
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, 1, 100)

# prepare containers
αs0_seq = np.empty((len(λ_seq), 2))
αs1_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)
    αs0_seq[i, :] = α

    # initial state s0 = 2
    α = ex3.wealth_distribution(s0_idx=1)
    αs1_seq[i, :] = α
```

```
/tmp/ipykernel_2253/2194301487.py:126: RuntimeWarning: divide by zero encountered␣
↪in reciprocal
  R = np.reciprocal(R)
```

```
fig, axs = plt.subplots(1, 2, figsize=(12, 4))

for i, αs_seq in enumerate([αs0_seq, αs1_seq]):
    for j in range(2):
        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()
```

76.7.4 Example 4

```
# dimensions
K, n = 2, 3

# states
s = np.array([1, 2, 3])

# transition
λ = .9
μ = .9
δ = .05

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
print("R = ", ex4.R)

# natural debt limit, A = [A1, A2, ..., AI]
print("A = \n", ex4.A)

print('')

for i in range(1, 4):
```

(continues on next page)

(continued from previous page)

```
    # 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 =  [1.43172499 0.44313807 1.33997564]
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.        ]]]
J =
[[[-2.37756442 -1.92325926]
  [-4.59067883 -3.71349163]
  [-2.37674158 -1.92259365]]]
```

76.8 Finite Horizon

The Python class **RecurCompetitive** provided above also can be used to compute competitive equilibrium allocations and Arrow securities prices for finite horizon economies.

The setting is a finite-horizon version of the one above except that time now runs for $T+1$ periods $t \in \mathbf{T} = \{0, 1, \dots, T\}$.

Consequently, we 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.

76.8.1 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

$$\begin{aligned} \psi_T^k &= [\alpha_k y - y^k] \\ \psi_{T-1}^k &= [I + Q][\alpha_k y - y^k] \\ \vdots \quad & \quad \vdots \\ \psi_0^k &= [I + Q + Q^2 + \cdots + Q^T][\alpha_k y - y^k] \end{aligned} \tag{76.5}$$

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 $s_0 \in [\bar{s}_1, \dots, \bar{s}_n]$, we must have

$$\psi_0^k(s_0) = 0, \quad k = 1, \dots, K$$

which means the equilibrium distribution of wealth satisfies

$$\alpha_k = \frac{V_z y^k}{V_z y} \tag{76.6}$$

where now in our finite-horizon economy

$$V = [I + Q + Q^2 + \cdots + Q^T] \tag{76.7}$$

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 (76.1)
- compute the distribution of wealth α from formulas (76.6) and (76.7)
- using α, assign each consumer k the share α_k of the aggregate endowment at each state
- return to the α-dependent formula (76.5) 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.

76.8.2 Finite Horizon Example

Below we 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 J 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]]
```

(continues on next page)

(continued from previous page)

```
 [[-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 ]
  [ 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]]
```

(continues on next page)

(continued from previous page)

```
 [[-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.        ]]]

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]]
```

(continues on next page)

(continued from previous page)

```
[[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.],
        [ 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

SEVENTYSEVEN

HETEROGENEOUS BELIEFS AND BUBBLES

Contents

- *Heterogeneous Beliefs and Bubbles*
  - *Overview*
  - *Structure of the Model*
  - *Solving the Model*
  - *Exercises*

In addition to what's in Anaconda, this lecture uses following libraries:

```
!pip install quantecon
```

77.1 Overview

This lecture describes a version of a model of Harrison and Kreps [HK78].

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
```

77.1.1 References

Prior to reading the following, you might like to review our lectures on

- *Markov chains*
- *Asset pricing with finite state space*

77.1.2 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.*

77.2 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.

77.2.1 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 [HK78] 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.

77.2.2 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.

77.2.3 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.

77.2.4 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.

77.3 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.

77.3.1 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 purchse 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.

77.3.2 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:

$$p_h(s) = \beta \left(P_h(s,0)(0 + p_h(0)) + P_h(s,1)(1 + p_h(1)) \right), \quad s = 0, 1 \tag{77.1}$$

These equations imply that the equilibrium price vector is

$$\begin{bmatrix} p_h(0) \\ p_h(1) \end{bmatrix} = \beta [I - \beta P_h]^{-1} P_h \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{77.2}$$

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.

77.3.3 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:

$$\bar{p}(s) = \beta \max \left\{ P_a(s,0)\bar{p}(0) + P_a(s,1)(1+\bar{p}(1)),\ P_b(s,0)\bar{p}(0) + P_b(s,1)(1+\bar{p}(1)) \right\} \tag{77.3}$$

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 (77.3) 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 (77.3), namely

$$\bar{p}^{j+1}(s) = \beta \max \left\{ P_a(s,0)\bar{p}^j(0) + P_a(s,1)(1+\bar{p}^j(1)),\ P_b(s,0)\bar{p}^j(0) + P_b(s,1)(1+\bar{p}^j(1)) \right\} \tag{77.4}$$

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],
                            1)

        # 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],
                           1)

    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
```

77.3.4 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 (77.3), the equilibrium price satisfies

$$\check{p}(s) = \beta \min \left\{ P_a(s,1)\check{p}(0) + P_a(s,1)(1+\check{p}(1)),\ P_b(s,1)\check{p}(0) + P_b(s,1)(1+\check{p}(1)) \right\} \tag{77.5}$$

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 (77.5), 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 (77.3), 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],
                           1)

        # 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
```

77.3.5 Further Interpretation

[Sch14] 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 (77.3) 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.

77.4 Exercises

Exercise 77.4.1

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 to Exercise 77.4.1

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)
```

(continues on next page)

(continued from previous page)

```
    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.

Part XII

Data and Empirics

CHAPTER

SEVENTYEIGHT

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*

78.1 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.

78.2 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://raw.githubusercontent.com/QuantEcon/lecture-python/master/source/_
↪static/lecture_specific/pandas_panel/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
Time
```

(continues on next page)

(continued from previous page)

```
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.core.indexes.datetimes.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.core.indexes.multi.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().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  ...
```

(continues on next page)

(continued from previous page)

```
Time       Pay period                                          ...
2006-01-01 Annual                                     21,042.28  ...
           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').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
```

(continues on next page)

(continued from previous page)

```
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)).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
```

(continues on next page)

(continued from previous page)

```
2010-01-01           7.88

[5 rows x 32 columns]
```

78.3 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://raw.githubusercontent.com/QuantEcon/lecture-python/master/source/_
↪static/lecture_specific/pandas_panel/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
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
```

(continues on next page)

(continued from previous page)

```
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: object
```

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'].replace(to_replace=country,
                                value='America',
                                inplace=True)
```

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  2007-01-01  2008-01-01  ...  2014-01-01  \
Continent Country                                          ...
America   Brazil            0.87        0.92        0.96  ...        1.21
          Canada            6.89        6.96        7.24  ...        8.22
          Chile             1.42        1.45        1.44  ...        1.76
          Colombia          1.01        1.02        1.01  ...        1.13
          Costa Rica         NaN         NaN         NaN  ...        2.41

                      2015-01-01  2016-01-01
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[ns]', 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]
```

78.4 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")

# 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
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.groupby(level='Continent', axis=1).mean().head()
```

```
Continent   America  Asia  Australia  Europe
Time
```

(continues on next page)

(continued from previous page)

```
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.groupby(level='Continent', axis=1).mean().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.groupby(level='Continent', axis=1).mean().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().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.groupby(level='Continent', axis=1)
grouped
```

```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x7f51e4726c90>
```

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).loc['2015'].unstack(), label=continent,␣
 ↪fill=True)

plt.title('Real minimum wages in 2015')
plt.xlabel('US dollars')
plt.legend()
plt.show()
```

78.5 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.

78.6 Exercises

Exercise 78.6.1

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://raw.githubusercontent.com/QuantEcon/lecture-python/master/source/_
↪static/lecture_specific/pandas_panel/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 to Exercise 78.6.1

```
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='object',␣
↪name='UNIT')
AGE Index(['From 15 to 24 years', 'From 25 to 54 years', 'From 55 to 64 years'],␣
↪dtype='object', name='AGE')
SEX Index(['Females', 'Males', 'Total'], dtype='object', name='SEX')
INDIC_EM Index(['Active population', 'Total employment (resident population␣
↪concept - LFS)'], dtype='object', 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='object', name='GEO')
```

Exercise 78.6.2

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 Exercise 78.6.2

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='object', 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

CHAPTER

SEVENTYNINE

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
```

79.1 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 [AJR01].

- 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
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
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()
```

79.1.1 Prerequisites

This lecture assumes you are familiar with basic econometrics.

For an introductory text covering these topics, see, for example, [Woo15].

79.2 Simple Linear Regression

[AJR01] 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/blob/master/source/_
↪static/lecture_specific/ols/maketable1.dta?raw=true')
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 [AJR01])

```
# 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']
labels = df1_subset['shortnam']

# Replace markers with country labels
fig, ax = plt.subplots()
```

(continues on next page)

(continued from previous page)

```
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, 15 Jan 2024   Prob (F-statistic):           4.16e-24
Time:                        07:26:17   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
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
```

```
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
```

(continues on next page)

(continued from previous page)

```
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()
```

79.3 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 $logpgp95_i$.

[AJR01] 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/blob/master/source/_
↪static/lecture_specific/ols/maketable2.dta?raw=true')
```

(continues on next page)

(continued from previous page)

```
# 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
R-squared        0.61    0.62    0.72
```

(continues on next page)

(continued from previous page)

```
No. observations 111     111     111
=========================================
Standard errors in parentheses.
* p<.1, ** p<.05, ***p<.01
```

79.4 Endogeneity

As [AJR01] 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 [AJR01] 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 [AJR01]), 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)),
```

(continues on next page)

(continued from previous page)

```
        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.

[AJR01] 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/blob/master/source/_
↪static/lecture_specific/ols/maketable4.dta?raw=true')
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, 15 Jan 2024   Prob (F-statistic):           1.08e-05
Time:                        07:26:19   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
==============================================================================
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'],
```

(continues on next page)

(continued from previous page)

```
                    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, 15 Jan 2024   Prob (F-statistic):           2.66e-10
Time:                        07:26:19   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
==============================================================================
```

(continues on next page)

(continued from previous page)

```
Dep. Variable:                logpgp95   R-squared:                       0.1870
Estimator:                     IV-2SLS   Adj. R-squared:                  0.1739
No. Observations:                   64   F-statistic:                     37.568
Date:                 Mon, Jan 15 2024   P-value (F-stat)                 0.0000
Time:                         07:26:19   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.

[AJR01] 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.

79.5 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.

79.6 Exercises

Exercise 79.6.1

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 to Exercise 79.6.1

```
# Load in data
df4 = pd.read_stata('https://github.com/QuantEcon/lecture-python/blob/master/source/_
↪static/lecture_specific/ols/maketable4.dta?raw=true')

# 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, 15 Jan 2024   Prob (F-statistic):           1.07e-17
Time:                        07:26:19   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
==============================================================================
```

(continues on next page)

(continued from previous page)

```
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 79.6.2

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 to Exercise 79.6.2

```
# Load in data
df1 = pd.read_stata('https://github.com/QuantEcon/lecture-python/blob/master/source/_
↪static/lecture_specific/ols/maketable1.dta?raw=true')
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
```

(continues on next page)

(continued from previous page)

```
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

EIGHTY

MAXIMUM LIKELIHOOD ESTIMATION

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*

80.1 Overview

In a *previous lecture*, 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 matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
from numpy import exp
from scipy.special import factorial
```

(continues on next page)

(continued from previous page)

```
import pandas as pd
from mpl_toolkits.mplot3d import Axes3D
import statsmodels.api as sm
from statsmodels.api import Poisson
from scipy.stats import norm
from statsmodels.iolib.summary2 import summary_col
```

80.1.1 Prerequisites

We assume familiarity with basic probability and multivariate calculus.

80.2 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.

80.2.1 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**.

80.2.2 Counting Billionaires

Treisman [Tre16] 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

```
poisson_pmf = lambda y, μ: μ**y / factorial(y) * 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=f'$\mu$={μ}',
            alpha=0.5,
            marker='o',
            markersize=8)

ax.grid()
ax.set_xlabel('$y$', fontsize=14)
ax.set_ylabel('$f(y \mid \mu)$', 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.

```
pd.options.display.max_columns = 10

# Load in data and view
df = pd.read_stata('https://github.com/QuantEcon/lecture-python/blob/master/source/_
↪static/lecture_specific/mle/fp.dta?raw=true')
df.head()
```

```
         country  ccode    year    cyear  numbil  ...   topint08     rintr  \
0  United States    2.0  1990.0  21990.0     NaN  ...  39.799999  4.988405
1  United States    2.0  1991.0  21991.0     NaN  ...  39.799999  4.988405
2  United States    2.0  1992.0  21992.0     NaN  ...  39.799999  4.988405
3  United States    2.0  1993.0  21993.0     NaN  ...  39.799999  4.988405
4  United States    2.0  1994.0  21994.0     NaN  ...  39.799999  4.988405

   noyrs  roflaw  nrrents
0   20.0    1.61      NaN
1   20.0    1.61      NaN
2   20.0    1.61      NaN
3   20.0    1.61      NaN
4   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).

80.3 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:

$$f(y_i \mid \mathbf{x}_i) = \frac{\mu_i^{y_i}}{y_i!} e^{-\mu_i}; \qquad y_i = 0, 1, 2, \dots, \infty. \tag{80.1}$$

$$\text{where } \mu_i = \exp(\mathbf{x}_i' \beta) = \exp(\beta_0 + \beta_1 x_{i1} + \dots + \beta_k x_{ik})$$

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 = np.array([0.26, 0.18, 0.25, -0.1, -0.22])

# Create some observations X
datasets = [np.array([0, 1, 1, 1, 2]),
```

(continues on next page)

(continued from previous page)

```
            np.array([2, 3, 2, 4, 0]),
            np.array([3, 4, 5, 3, 2]),
            np.array([6, 5, 4, 4, 7])]


fig, ax = plt.subplots(figsize=(12, 8))

for X in datasets:
    μ = exp(X @ β)
    distribution = []
    for y_i in y_values:
        distribution.append(poisson_pmf(y_i, μ))
    ax.plot(y_values,
            distribution,
            label=f'$\mu_i$={μ:.1}',
            marker='o',
            markersize=8,
            alpha=0.5)

ax.grid()
ax.legend()
ax.set_xlabel('$y \mid x_i$')
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).

80.4 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 = np.arange(0, y_n, 1)

    # Create coordinate points of X and Y
    X, Y = np.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='$y_1$', ylabel='$y_2$')
    ax.set_zlabel('$f(y_1, y_2)$', 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

$$f(y_1, y_2, \dots, y_n \mid \mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n; \beta) = \prod_{i=1}^{n} \frac{\mu_i^{y_i}}{y_i!} e^{-\mu_i}$$

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

$$\begin{aligned}\mathcal{L}(\beta \mid y_1, y_2, \dots, y_n \; ; \; \mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n) =& \prod_{i=1}^{n} \frac{\mu_i^{y_i}}{y_i!} e^{-\mu_i} \\ =& f(y_1, y_2, \dots, y_n \mid \; \mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n; \beta)\end{aligned}$$

Now that we have our likelihood function, we want to find the $\hat{\beta}$ 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

$$\begin{aligned}
\log \mathcal{L}(\beta) &= \log \Big(f(y_1; \beta) \cdot f(y_2; \beta) \cdot \ldots \cdot f(y_n; \beta) \Big) \\
&= \sum_{i=1}^{n} \log f(y_i; \beta) \\
&= \sum_{i=1}^{n} \log \Big(\frac{\mu_i^{y_i}}{y_i!} e^{-\mu_i} \Big) \\
&= \sum_{i=1}^{n} y_i \log \mu_i - \sum_{i=1}^{n} \mu_i - \sum_{i=1}^{n} \log y!
\end{aligned}$$

The MLE of the Poisson to the Poisson for $\hat{\beta}$ can be obtained by solving

$$\max_{\beta} \Big(\sum_{i=1}^{n} y_i \log \mu_i - \sum_{i=1}^{n} \mu_i - \sum_{i=1}^{n} \log y! \Big)$$

However, no analytical solution exists to the above problem – to find the MLE we need to use numerical methods.

80.5 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 $\hat{\beta}$.

At $\hat{\beta}$, the first derivative of the log-likelihood function will be equal to 0.

Let's illustrate this by supposing

$$\log \mathcal{L}(\beta) = -(\beta - 10)^2 - 10$$

```
β = np.linspace(1, 20)
logL = -(β - 10) ** 2 - 10
dlogL = -2 * β + 20

fig, (ax1, ax2) = plt.subplots(2, sharex=True, figsize=(12, 8))

ax1.plot(β, logL, lw=2)
ax2.plot(β, dlogL, 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,
```

(continues on next page)

(continued from previous page)

```
                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 $\hat{\beta} = \beta_{(k+1)}$
- If false, then update $\beta_{(k+1)}$

As can be seen from the updating equation, $\beta_{(k+1)} = \beta_{(k)}$ only when $G(\beta_{(k)}) = 0$ ie. 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:

    def __init__(self, y, X, β):
        self.X = X
        self.n, self.k = X.shape
        # Reshape y as a n_by_1 column vector
        self.y = y.reshape(self.n,1)
        # Reshape β as a k_by_1 column vector
        self.β = β.reshape(self.k,1)

    def μ(self):
        return np.exp(self.X @ self.β)

    def logL(self):
        y = self.y
        μ = self.μ()
        return np.sum(y * np.log(μ) - μ - np.log(factorial(y)))

    def G(self):
        y = self.y
        μ = self.μ()
        return X.T @ (y - μ)

    def H(self):
        X = self.X
        μ = self.μ()
        return -(X.T @ (μ * X))
```

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, tol=1e-3, max_iter=1000, display=True):

    i = 0
```

(continues on next page)

(continued from previous page)

```
    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 np.any(error > tol) and i < max_iter:
        H, G = model.H(), model.G()
        β_new = model.β - (np.linalg.inv(H) @ G)
        error = np.abs(β_new - model.β)
        model.β = β_new

        # Print iterations
        if display:
            β_list = [f'{t:.3}' for t in list(model.β.flatten())]
            update = f'{i:<13}{model.logL():<16.8}{β_list}'
            print(update)

        i += 1

    print(f'Number of iterations: {i}')
    print(f'β_hat = {model.β.flatten()}')

    # Return a flat array for β (instead of a k_by_1 column vector)
    return model.β.flatten()
```

Let's try out our algorithm with a small dataset of 5 observations and 3 variables in **X**.

```
X = np.array([[1, 2, 5],
              [1, 1, 3],
              [1, 4, 2],
              [1, 5, 2],
              [1, 3, 1]])

y = np.array([1, 0, 1, 1, 0])

# Take a guess at initial βs
init_β = np.array([0.1, 0.1, 0.1])

# Create an object with Poisson model values
poi = PoissonRegression(y, X, β=init_β)

# Use newton_raphson to find the MLE
β_hat = newton_raphson(poi, display=True)
```

```
Iteration_k  Log-likelihood  θ
--------------------------------------------------------------------------------
 ↪------
0            -4.3447622      ['-1.49', '0.265', '0.244']
1            -3.5742413      ['-3.38', '0.528', '0.474']
2            -3.3999526      ['-5.06', '0.782', '0.702']
```

(continues on next page)

(continued from previous page)

```
3            -3.3788646      ['-5.92', '0.909', '0.82']
4            -3.3783559      ['-6.07', '0.933', '0.843']
5            -3.3783555      ['-6.08', '0.933', '0.843']
6            -3.3783555      ['-6.08', '0.933', '0.843']
Number of iterations: 7
β_hat = [-6.07848573  0.9334028   0.84329677]
```

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 $\hat{\beta}$

```
poi.G()
```

```
array([[-2.54574140e-13],
       [-6.44040377e-13],
       [-4.99100761e-13]])
```

The iterative process can be visualized in the following diagram, where the maximum is found at $\beta = 10$

```
logL = lambda x: -(x - 10) ** 2 - 10

def find_tangent(β, a=0.01):
    y1 = logL(β)
    y2 = logL(β+a)
    x = np.array([[β, 1], [β+a, 1]])
    m, c = np.linalg.lstsq(x, np.array([y1, y2]), rcond=None)[0]
    return m, c

β = np.linspace(2, 18)
fig, ax = plt.subplots(figsize=(12, 8))
ax.plot(β, logL(β), lw=2, c='black')

for β in [7, 8.5, 9.5, 10]:
    β_line = np.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], f'$G({β}) = {abs(m):.0f}$', 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)
```

(continues on next page)

(continued from previous page)

```
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.

80.6 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.

```
X = np.array([[1, 2, 5],
              [1, 1, 3],
              [1, 4, 2],
              [1, 5, 2],
              [1, 3, 1]])

y = np.array([1, 0, 1, 1, 0])

stats_poisson = Poisson(y, X).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, 15 Jan 2024   Pseudo R-squ.:                  0.2546
Time:                        07:22:21   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 (80.1), 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 = sm.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, 15 Jan 2024   Pseudo R-squ.:                  0.8574
Time:                        07:22:21   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
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 = sm.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)
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)
Pseudo R-squared 0.86       0.90       0.90
No. observations 197        131        131
==================================================
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 an 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.

80.7 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.

80.8 Exercises

Exercise 80.8.1

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 `scipy` module `stats.norm` contains the functions needed to compute the cmf and pmf of the normal distribution.

Solution to Exercise 80.8.1

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

$$\frac{\partial \log \mathcal{L}}{\partial \beta} = \sum_{i=1}^{n} \Big[y_i \frac{\phi(\mathbf{x}_i'\beta)}{\Phi(\mathbf{x}_i'\beta)} - (1-y_i)\frac{\phi(\mathbf{x}_i'\beta)}{1-\Phi(\mathbf{x}_i'\beta)} \Big] \mathbf{x}_i$$

The Hessian of the Probit model is

$$\frac{\partial^2 \log \mathcal{L}}{\partial \beta \partial \beta'} = -\sum_{i=1}^{n} \phi(\mathbf{x}_i'\beta) \Big[y_i \frac{\phi(\mathbf{x}_i'\beta) + \mathbf{x}_i'\beta\Phi(\mathbf{x}_i'\beta)}{[\Phi(\mathbf{x}_i'\beta)]^2} + (1-y_i)\frac{\phi(\mathbf{x}_i'\beta) - \mathbf{x}_i'\beta(1-\Phi(\mathbf{x}_i'\beta))}{[1-\Phi(\mathbf{x}_i'\beta)]^2} \Big] \mathbf{x}_i\mathbf{x}_i'$$

Using these results, we can write a class for the Probit model as follows

```
class ProbitRegression:

    def __init__(self, y, X, β):
        self.X, self.y, self.β = X, y, β
        self.n, self.k = X.shape

    def μ(self):
        return norm.cdf(self.X @ self.β.T)
```

(continues on next page)

(continued from previous page)

```
    def ϕ(self):
        return norm.pdf(self.X @ self.β.T)

    def logL(self):
        μ = self.μ()
        return np.sum(y * np.log(μ) + (1 - y) * np.log(1 - μ))

    def G(self):
        μ = self.μ()
        ϕ = self.ϕ()
        return np.sum((X.T * y * ϕ / μ - X.T * (1 - y) * ϕ / (1 - μ)),
                      axis=1)

    def H(self):
        X = self.X
        β = self.β
        μ = self.μ()
        ϕ = self.ϕ()
        a = (ϕ + (X @ β.T) * μ) / μ**2
        b = (ϕ - (X @ β.T) * (1 - μ)) / (1 - μ)**2
        return -(ϕ * (y * a + (1 - y) * b) * X.T) @ X
```

Exercise 80.8.2

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 to Exercise 80.8.2

Here is one solution

```
X = np.array([[1, 2, 4],
              [1, 1, 1],
              [1, 4, 3],
              [1, 5, 6],
              [1, 3, 5]])

y = np.array([1, 0, 1, 1, 0])
```

(continues on next page)

(continued from previous page)

```
# Take a guess at initial βs
β = np.array([0.1, 0.1, 0.1])

# Create instance of Probit regression class
prob = ProbitRegression(y, X, β)

# Run Newton-Raphson algorithm
newton_raphson(prob)
```

```
Iteration_k  Log-likelihood  θ
-------------------------------------------------------------------------------
 ↪------
0            -2.3796884      ['-1.34', '0.775', '-0.157']
1            -2.3687526      ['-1.53', '0.775', '-0.0981']
2            -2.3687294      ['-1.55', '0.778', '-0.0971']
3            -2.3687294      ['-1.55', '0.778', '-0.0971']
Number of iterations: 4
β_hat = [-1.54625858  0.77778952 -0.09709757]
```

```
array([-1.54625858,  0.77778952, -0.09709757])
```

```
# Use statsmodels to verify results

print(Probit(y, X).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, 15 Jan 2024   Pseudo R-squ.:                  0.2961
Time:                        07:22:22   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
==============================================================================
```

Part XIII

Auctions

CHAPTER

EIGHTYONE

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 [Vic61]

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.

81.1 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 $i = 1, 2, \dots, n$.

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 $b - v_i > 0$.
- 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.

81.1.1 Characterization of FPSB Auction

A FPSB auction has a unique symmetric Bayesian Nash Equilibrium.

The optimal bid of buyer i is

$$\mathbf{E}[y_i | y_i < v_i] \tag{81.1}$$

where v_i is the valuation of bidder i and y_i is the maximum valuation of all other bidders:

$$y_i = \max_{j \neq i} v_j \tag{81.2}$$

A proof for this assertion is available at the Wikepedia page about Vickrey auctions

81.2 Second-Price Sealed-Bid Auction (SPSB)

Protocols: In a second-price sealed-bid (SPSB) auction, the winner pays the second-highest bid.

81.3 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 Wikepedia page about Vickrey auctions

81.4 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

81.5 Setup

There are n bidders.

Each bidder knows that there are $n-1$ other bidders.

81.6 First price sealed bid auction

An optimal bid for bidder i in a **FPSB** is described by equations (81.1) and (81.2).

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:

$$\begin{aligned}\mathbf{E}(y_i | y_i < v_i) &= \frac{\int_0^{v_i} y_i \tilde{f}_{n-1}(y_i) dy_i}{\int_0^{v_i} \tilde{f}_{n-1}(y_i) dy_i} \\ &= \frac{\int_0^{v_i} (n-1) y_i^{n-1} dy_i}{\int_0^{v_i} (n-1) y_i^{n-2} dy_i} \\ &= \frac{n-1}{n} y_i \Big|_0^{v_i} \\ &= \frac{n-1}{n} v_i\end{aligned}$$

81.7 Second Price Sealed Bid Auction

In a **SPSB**, it is optimal for bidder i to bid v_i.

81.8 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 generate the same randomess
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.

```
idx = np.argsort(v, axis=0)  # Biders' 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.

v = np.take_along_axis(v, idx, axis=0)  # same as np.sort(v, axis=0), except now we␣
↪retain the idx
b = np.take_along_axis(b, idx, axis=0)

ii = np.repeat(np.arange(1,N+1)[:,None], R, axis=1)  # the id for the bidders is␣
↪created.
ii = np.take_along_axis(ii, idx, axis=0)  # the id is sorted according to bid price␣
↪as well.

winning_player = ii[-1,:] # In FPSB and SPSB, winners are those with highest values.

winner_pays_fpsb = b[-1,:]  # highest bid
winner_pays_spsb = v[-2,:]  # 2nd-highest valuation
```

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()
```

81.9 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 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 $b(v_i) = \mathbf{E}_{y_i}[y_i | y_i < v_i]$.

81.10 Calculation of Bid Price in FPSB

In equations (81.1) and (81.1), 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
    array_residual=array[1:,:].copy()  # drop the first row because we assume first␣
↪row is the winner's bid

    index=(array_residual<v_hat).all(axis=0)
    array_conditional=array_residual[:,index].copy()

    array_conditional=np.sort(array_conditional, axis=0)
    return array_conditional[-order,:].mean()
```

We can check the accuracy of our `evaluate_largest` method by comparing it with an analytical solution.

We find that despite small discrepancy, the evaluate_largest method functions well.

Furthermore, if we take a very large number of auctions, say 1 million, the discrepancy disappears.

```
v_grid = np.linspace(0.3,1,8)
bid_analytical = b_star(v_grid,N)
bid_simulated = [evaluate_largest(ii, v) for ii in v_grid]
```

(continues on next page)

(continued from previous page)

```
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()
```

81.11 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)

EV=[evaluate_largest(ii, v) for ii in v_grid]
# nan values are returned for some low quantiles due to lack of observations
```

```
/tmp/ipykernel_4341/521884726.py:25: RuntimeWarning: Mean of empty slice.
  return array_conditional[-order,:].mean()
/opt/conda/envs/quantecon/lib/python3.11/site-packages/numpy/core/_methods.py:192:␣
↪RuntimeWarning: invalid value encountered in scalar divide
  ret = ret.dtype.type(ret / rcount)
```

```
# 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)

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)
v = np.take_along_axis(v, idx, axis=0)  # same as np.sort(v, axis=0), except now we␣
↪retain the idx
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)

winning_player = ii[-1,:]

winner_pays_fpsb = b[-1,:]  # highest bid
winner_pays_spsb = v[-2,:]  # 2nd-highest valuation
```

```
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
```

81.12 5 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
```

(continues on next page)

(continued from previous page)

```
        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
        array_residual = self.value_mat[1:,:].copy()
        # drop the first row because we assume first row is the winner's bid

        index=(array_residual<v_hat).all(axis=0)
        array_conditional=array_residual[:,index].copy()

        array_conditional=np.sort(array_conditional, axis=0)

        return array_conditional[-order,:].mean()

    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)

        EV=[self.evaluate_largest(ii) for ii in v_grid]
        # nan values are returned for some low quantiles due to lack of observations

        # 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')
```

(continues on next page)

(continued from previous page)

```
        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)
        self.v = np.take_along_axis(self.value_mat, idx, axis=0)  # same as np.sort(v,
↪ axis=0), except now we retain the idx
        self.b = np.take_along_axis(self.b, idx, axis=0)

        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,:]

        winner_pays_fpsb = self.b[-1,:]  # highest bid
        winner_pays_spsb = self.v[-2,:]  # 2nd-highest valuation

        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()

        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()
```

```
/tmp/ipykernel_4341/919518230.py:37: RuntimeWarning: Mean of empty slice.
  return array_conditional[-order,:].mean()
/opt/conda/envs/quantecon/lib/python3.11/site-packages/numpy/core/_methods.py:192:␣
↪RuntimeWarning: invalid value encountered in scalar divide
  ret = ret.dtype.type(ret / rcount)
```

```
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
```

81.13 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: http://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

EIGHTYTWO

MULTIPLE GOOD ALLOCATION MECHANISMS

```
!pip install prettytable
```

82.1 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 [Gro73], [Cla71] 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.

82.2 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}

82.3 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.

82.4 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.

82.5 Ascending Bid Auction

82.5.1 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 $j = [1, 2, \dots, m]$

  - 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 $v_{ij}, i = 1, \dots, n$, 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 $p = [p_1, \dots, p_n]$ 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.

82.6 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)
```

82.7 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)
    result=[]
```

(continues on next page)

(continued from previous page)

```
    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)
```

```
True
```

82.7.1 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   |
+--------------+-----+-----+-------+
|    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)
```

(continues on next page)

(continued from previous page)

```
    if loser_num == 0:
        print('The auction ends because every buyer gets one house.')
        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
```

```
[2, 1, 0]
```

```
loser_list
```

```
[3]
```

82.7.2 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:
    ----------
```

(continues on next page)

(continued from previous page)

```
    loser_list: a list that includes the indexes of losers

    p: np.array. price array of houses

    ϵ: minimum increment of bid price

    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 |
+--------------+---+---+---+
```

82.7.3 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 |
+--------------+---+---+---+
```

82.7.4 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 |
+--------------+---+---+---+
```

82.7.5 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
```

```
10
```

82.8 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:
        ----------
```

(continues on next page)

(continued from previous page)

```
        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 = []
        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
```

(continues on next page)

(continued from previous page)

```
        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)

        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
```

(continues on next page)

(continued from previous page)

```
        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␣
↪== 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')
```

(continues on next page)

(continued from previous page)

```
    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)
        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)
```

(continues on next page)

(continued from previous page)

```
                    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␣
↪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
+--------------+---+----+---+---+
```

(continues on next page)

(continued from previous page)

```
| 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   |
+--------------+-----+-----+-------+
|    Buyer     | [2] | [1] | [0 3] |
+--------------+-----+-----+-------+
The bid prices for houses are
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Price     | 3 | 2 | 1 |
+--------------+---+---+---+
The winners are
[2, 1, 0]
The losers are
[3]


Round 2
The bid information is
+--------------+-----+-----+-----+
| House Number |  0  |  1  |  2  |
+--------------+-----+-----+-----+
|    Buyer     | [2] | [1] | [3] |
+--------------+-----+-----+-----+
The bid prices for houses are
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Price     | 3 | 2 | 2 |
+--------------+---+---+---+
The winners are
[2, 1, 3]
The losers are
[0]
```

(continues on next page)

(continued from previous page)

```
Round 3
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 | 3 |
+--------------+---+---+---+
The winners are
[2, 1, 0]
The losers are
[3]


Round 4
The bid information is
+--------------+-----+-----+-----+
| House Number |  0  |  1  |  2  |
+--------------+-----+-----+-----+
|    Buyer     | [2] | [3] | [0] |
+--------------+-----+-----+-----+
The bid prices for houses are
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Price     | 3 | 3 | 3 |
+--------------+---+---+---+
The winners are
[2, 3, 0]
The losers are
[1]


Round 5
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 | 4 | 3 |
+--------------+---+---+---+
The winners are
[2, 1, 0]
The losers are
[3]
```

(continues on next page)

(continued from previous page)

```
Auction ends


The final result is as follows


The allocation plan is
+--------------+-----+-----+-----+
| House Number |  0  |  1  |  2  |
+--------------+-----+-----+-----+
|    Buyer     | [2] | [1] | [0] |
+--------------+-----+-----+-----+
The bid prices for houses are
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Price     | 3 | 4 | 3 |
+--------------+---+---+---+
The winners are
[2, 1, 0]
The losers are
[3]
The houses unsold are
[]
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.]])
```

82.9 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


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
[2, 1, 0]
The losers are
[]


The auction ends because every buyer gets one house.
```

(continues on next page)

(continued from previous page)

```
Auction ends


The final result is as follows


The allocation plan 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
[2, 1, 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)

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
```

(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 4] | [0 3] |
+--------------+-----+-------+-------+
The bid prices for houses are
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Price     | 3 | 2 | 1 |
+--------------+---+---+---+
The winners are
[2, 4, 3]
The losers are
[0, 1]


Round 2
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 | 3 | 2 |
+--------------+---+---+---+
The winners are
[2, 1, 0]
The losers are
[3, 4]


Round 3
The bid information is
+--------------+-----+-----+-----+
| House Number |  0  |  1  |  2  |
+--------------+-----+-----+-----+
|    Buyer     | [2] | [4] | [3] |
+--------------+-----+-----+-----+
The bid prices for houses are
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Price     | 3 | 4 | 3 |
+--------------+---+---+---+
```

(continues on next page)

(continued from previous page)

```
The winners are
[2, 4, 3]
The losers are
[0, 1]


Round 4
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 | 5 | 4 |
+--------------+---+---+---+
The winners are
[2, 1, 0]
The losers are
[3, 4]


Auction ends


The final result is as follows


The allocation plan is
+--------------+-----+-----+-----+
| House Number |  0  |  1  |  2  |
+--------------+-----+-----+-----+
|    Buyer     | [2] | [1] | [0] |
+--------------+-----+-----+-----+
The bid prices for houses are
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Price     | 3 | 5 | 4 |
+--------------+---+---+---+
The winners are
[2, 1, 0]
The losers are
[3, 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])
```

(continues on next page)

(continued from previous page)

```
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] |
+--------------+-----+-------+
The bid prices for houses are
+--------------+---+---+---+---+---+
| House Number | 0 | 1 | 2 | 3 | 4 |
+--------------+---+---+---+---+---+
|    Price     | 2 | 2 | 1 | 1 | 1 |
+--------------+---+---+---+---+---+
The winners are
[1, 2]
The losers are
[0]


Round 2
The bid information is
+--------------+-----+-----+
| House Number |  1  |  2  |
+--------------+-----+-----+
|    Buyer     | [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 | 2 | 1 | 1 |
+--------------+---+---+---+---+---+
The winners are
[1, 0]
The losers are
[2]


Round 3
The bid information is
+--------------+-----+-----+
| House Number |  1  |  2  |
+--------------+-----+-----+
|    Buyer     | [1] | [2] |
+--------------+-----+-----+
The bid prices for houses are
+--------------+---+---+---+---+---+
| House Number | 0 | 1 | 2 | 3 | 4 |
+--------------+---+---+---+---+---+
|    Price     | 2 | 2 | 3 | 1 | 1 |
+--------------+---+---+---+---+---+
The winners are
[1, 2]
The losers are
[0]


Round 4
The bid information is
+--------------+-----+-----+
| House Number |  1  |  2  |
+--------------+-----+-----+
|    Buyer     | [1] | [0] |
+--------------+-----+-----+
The bid prices for houses are
+--------------+---+---+---+---+---+
| House Number | 0 | 1 | 2 | 3 | 4 |
+--------------+---+---+---+---+---+
|    Price     | 2 | 2 | 4 | 1 | 1 |
+--------------+---+---+---+---+---+
The winners are
[1, 0]
The losers are
[2]


Round 5
The bid information is
+--------------+-----+-----+
| House Number |  1  |  2  |
+--------------+-----+-----+
```

(continues on next page)

(continued from previous page)

```
|    Buyer     | [2] | [0] |
+--------------+-----+-----+
The bid prices for houses are
+--------------+---+---+---+---+---+
| House Number | 0 | 1 | 2 | 3 | 4 |
+--------------+---+---+---+---+---+
|    Price     | 2 | 3 | 4 | 1 | 1 |
+--------------+---+---+---+---+---+
The winners are
[2, 0]
The losers are
[1]


Round 6
The bid information is
+--------------+-----+-----+
| House Number |  1  |  2  |
+--------------+-----+-----+
|    Buyer     | [1] | [0] |
+--------------+-----+-----+
The bid prices for houses are
+--------------+---+---+---+---+---+
| House Number | 0 | 1 | 2 | 3 | 4 |
+--------------+---+---+---+---+---+
|    Price     | 2 | 4 | 4 | 1 | 1 |
+--------------+---+---+---+---+---+
The winners are
[1, 0]
The losers are
[2]


Round 7
The bid information is
+--------------+-----+-----+
| House Number |  1  |  2  |
+--------------+-----+-----+
|    Buyer     | [1] | [2] |
+--------------+-----+-----+
The bid prices for houses are
+--------------+---+---+---+---+---+
| House Number | 0 | 1 | 2 | 3 | 4 |
+--------------+---+---+---+---+---+
|    Price     | 2 | 4 | 5 | 1 | 1 |
+--------------+---+---+---+---+---+
The winners are
[1, 2]
The losers are
[0]


Round 8
The bid information is
+--------------+-----+-----+-----+
| House Number |  1  |  2  |  0  |
```

(continues on next page)

(continued from previous page)

```
+--------------+-----+-----+-----+
|    Buyer     | [1] | [2] | [0] |
+--------------+-----+-----+-----+
The bid prices for houses are
+--------------+---+---+---+---+---+
| House Number | 0 | 1 | 2 | 3 | 4 |
+--------------+---+---+---+---+---+
|    Price     | 3 | 4 | 5 | 1 | 1 |
+--------------+---+---+---+---+---+
The winners are
[1, 2, 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     | [1] | [2] | [0] |
+--------------+-----+-----+-----+
The bid prices for houses are
+--------------+---+---+---+---+---+
| House Number | 0 | 1 | 2 | 3 | 4 |
+--------------+---+---+---+---+---+
|    Price     | 3 | 4 | 5 | 1 | 1 |
+--------------+---+---+---+---+---+
The winners are
[1, 2, 0]
The losers are
[]
The houses unsold are
[3, 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
```

(continues on next page)

(continued from previous page)

```
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
[1, 0]
The losers are
[2]


Round 2
The bid information is
+--------------+-----+-----+
| House Number |  1  |  2  |
+--------------+-----+-----+
|    Buyer     | [1] | [2] |
+--------------+-----+-----+
The bid prices for houses are
+--------------+----+---+---+---+---+
| House Number | 0  | 1 | 2 | 3 | 4 |
+--------------+----+---+---+---+---+
|    Price     | 10 | 2 | 2 | 1 | 1 |
+--------------+----+---+---+---+---+
```

(continues on next page)

(continued from previous page)

```
The winners are
[1, 2]
The losers are
[0]


Round 3
The bid information is
+--------------+-----+-----+
| House Number |  1  |  2  |
+--------------+-----+-----+
|    Buyer     | [1] | [0] |
+--------------+-----+-----+
The bid prices for houses are
+--------------+----+---+---+---+---+
| House Number | 0  | 1 | 2 | 3 | 4 |
+--------------+----+---+---+---+---+
|    Price     | 10 | 2 | 3 | 1 | 1 |
+--------------+----+---+---+---+---+
The winners are
[1, 0]
The losers are
[2]


Round 4
The bid information is
+--------------+-----+-----+
| House Number |  1  |  2  |
+--------------+-----+-----+
|    Buyer     | [1] | [2] |
+--------------+-----+-----+
The bid prices for houses are
+--------------+----+---+---+---+---+
| House Number | 0  | 1 | 2 | 3 | 4 |
+--------------+----+---+---+---+---+
|    Price     | 10 | 2 | 4 | 1 | 1 |
+--------------+----+---+---+---+---+
The winners are
[1, 2]
The losers are
[0]


Round 5
The bid information is
+--------------+-----+-----+
| House Number |  1  |  2  |
+--------------+-----+-----+
|    Buyer     | [1] | [0] |
+--------------+-----+-----+
The bid prices for houses are
+--------------+----+---+---+---+---+
| House Number | 0  | 1 | 2 | 3 | 4 |
+--------------+----+---+---+---+---+
|    Price     | 10 | 2 | 5 | 1 | 1 |
```

(continues on next page)

(continued from previous page)

```
+--------------+----+---+---+---+---+
The winners are
[1, 0]
The losers are
[2]


Round 6
The bid information is
+--------------+-----+-----+
| House Number |  1  |  2  |
+--------------+-----+-----+
|    Buyer     | [2] | [0] |
+--------------+-----+-----+
The bid prices for houses are
+--------------+----+---+---+---+---+
| House Number | 0  | 1 | 2 | 3 | 4 |
+--------------+----+---+---+---+---+
|    Price     | 10 | 3 | 5 | 1 | 1 |
+--------------+----+---+---+---+---+
The winners are
[2, 0]
The losers are
[1]


Round 7
The bid information is
+--------------+-----+-----+
| House Number |  1  |  2  |
+--------------+-----+-----+
|    Buyer     | [1] | [0] |
+--------------+-----+-----+
The bid prices for houses are
+--------------+----+---+---+---+---+
| House Number | 0  | 1 | 2 | 3 | 4 |
+--------------+----+---+---+---+---+
|    Price     | 10 | 4 | 5 | 1 | 1 |
+--------------+----+---+---+---+---+
The winners are
[1, 0]
The losers are
[2]


Round 8
The bid information is
+--------------+-----+-----+
| House Number |  1  |  2  |
+--------------+-----+-----+
|    Buyer     | [1] | [2] |
+--------------+-----+-----+
The bid prices for houses are
+--------------+----+---+---+---+---+
| House Number | 0  | 1 | 2 | 3 | 4 |
+--------------+----+---+---+---+---+
```

(continues on next page)

(continued from previous page)

```
|    Price     | 10 | 4 | 6 | 1 | 1 |
+--------------+----+---+---+---+---+
The winners are
[1, 2]
The losers are
[0]


Round 9
The bid information is
+--------------+-----+-----+-----+
| House Number |  1  |  2  |  3  |
+--------------+-----+-----+-----+
|    Buyer     | [1] | [2] | [0] |
+--------------+-----+-----+-----+
The bid prices for houses are
+--------------+----+---+---+---+---+
| House Number | 0  | 1 | 2 | 3 | 4 |
+--------------+----+---+---+---+---+
|    Price     | 10 | 4 | 6 | 2 | 1 |
+--------------+----+---+---+---+---+
The winners are
[1, 2, 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     | [1] | [2] | [0] |
+--------------+-----+-----+-----+
The bid prices for houses are
+--------------+----+---+---+---+---+
| House Number | 0  | 1 | 2 | 3 | 4 |
+--------------+----+---+---+---+---+
|    Price     | 10 | 4 | 6 | 2 | 1 |
+--------------+----+---+---+---+---+
The winners are
[1, 2, 0]
The losers are
[]
The houses unsold are
[0, 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
```

82.10 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 [Gro73], [Cla71].

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 $i = 1, \dots, n$.
- 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.

82.11 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.],
        [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
```

```
(2, 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.]]))
```

82.12 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)

    def find_argmax(self):
        """
        Find the house-buyer pair with the highest value.
```

(continues on next page)

(continued from previous page)

```
        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)
        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.]]
```

82.12.1 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 [Gro73], [Cla71], and [Vic61].

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

$$X(v) = \text{argmax}_{x \in X} \sum_{j=1}^{m} v_j(x) \tag{82.1}$$

and charges person j a "social cost"

$$\check{t}_j(v) = \max_{x \in X} \sum_{k \neq j} v_k(x) - \sum_{k \neq j} v_k(X(v)) \tag{82.2}$$

In our setting, equation (82.1) says that the VCG allocation allocates houses to maximize the total value of the successful prospective buyers.

In our setting, equation (82.2) 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 (82.2):

- 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 $j = 1, \dots, m$ and compare with p_j from the second price auction.

82.12.2 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


Prices of house:
```

(continues on next page)

(continued from previous page)

```
 [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


House 2 is sold to buyer 3 at price 4
```

(continues on next page)

(continued from previous page)

```
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 XIV

Other

CHAPTER

EIGHTYTHREE

TROUBLESHOOTING

Contents

- *Troubleshooting*
  - *Fixing Your Local Environment*
  - *Reporting an Issue*

This page is for readers experiencing errors when running the code from the lectures.

83.1 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 conda install -y quantecon on the command line, or
- execute !conda install -y 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.

83.2 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.

Another feedback option is to use our discourse forum.

Finally, you can provide direct feedback to contact@quantecon.org

CHAPTER

EIGHTYFOUR

REFERENCES

CHAPTER

EIGHTYFIVE

EXECUTION STATISTICS

This table contains the latest execution statistics.

Document	Modified	Method	Run Time (s)	Status
aiyagari	2024-01-15 05:40	cache	26.56	✅
ar1_bayes	2024-01-15 05:49	cache	584.2	✅
ar1_processes	2024-01-15 05:49	cache	5.59	✅
ar1_turningpts	2024-01-15 05:50	cache	62.84	✅
back_prop	2024-01-15 05:53	cache	139.85	✅
bayes_nonconj	2024-01-15 07:03	cache	4195.28	✅
cake_eating_numerical	2024-01-15 07:03	cache	29.94	✅
cake_eating_problem	2024-01-15 07:03	cache	2.25	✅
career	2024-01-15 07:04	cache	18.33	✅
cass_koopmans_1	2024-01-15 07:04	cache	10.88	✅
cass_koopmans_2	2024-01-15 07:04	cache	9.31	✅
coleman_policy_iter	2024-01-15 07:04	cache	23.6	✅
complex_and_trig	2024-01-15 07:04	cache	4.05	✅
cross_product_trick	2024-01-15 07:04	cache	1.0	✅
egm_policy_iter	2024-01-15 07:05	cache	11.98	✅
eig_circulant	2024-01-15 07:05	cache	4.96	✅
exchangeable	2024-01-15 07:05	cache	10.48	✅
finite_markov	2024-01-15 07:05	cache	11.55	✅
ge_arrow	2024-01-15 07:05	cache	2.47	✅
geom_series	2024-01-15 07:05	cache	4.61	✅
harrison_kreps	2024-01-15 07:05	cache	8.95	✅
heavy_tails	2024-01-15 07:05	cache	1.27	✅
hoist_failure	2024-01-15 07:07	cache	76.11	✅
house_auction	2024-01-15 07:07	cache	7.84	✅
ifp	2024-01-15 07:07	cache	42.79	✅
ifp_advanced	2024-01-15 07:08	cache	31.18	✅
imp_sample	2024-01-15 07:13	cache	275.86	✅
intro	2024-01-15 07:05	cache	1.27	✅
inventory_dynamics	2024-01-15 07:13	cache	12.2	✅
jv	2024-01-15 07:13	cache	20.52	✅
kalman	2024-01-15 07:13	cache	12.8	✅
kalman_2	2024-01-15 07:14	cache	30.19	✅
kesten_processes	2024-01-15 07:15	cache	70.61	✅
lagrangian_lqdp	2024-01-15 07:15	cache	23.36	✅
lake_model	2024-01-15 07:16	cache	22.31	✅

continues on next page

Table 85.1 – continued from previous page

Document	Modified	Method	Run Time (s)	Status
likelihood_bayes	2024-01-15 07:17	cache	48.95	✅
likelihood_ratio_process	2024-01-15 07:17	cache	10.67	✅
linear_algebra	2024-01-15 07:17	cache	3.05	✅
linear_models	2024-01-15 07:17	cache	12.12	✅
lln_clt	2024-01-15 07:17	cache	14.44	✅
lp_intro	2024-01-15 07:17	cache	2.18	✅
lq_inventories	2024-01-15 07:18	cache	23.62	✅
lqcontrol	2024-01-15 07:18	cache	10.72	✅
markov_asset	2024-01-15 07:18	cache	11.1	✅
markov_perf	2024-01-15 07:18	cache	9.85	✅
mccall_correlated	2024-01-15 07:20	cache	92.96	✅
mccall_fitted_vfi	2024-01-15 07:20	cache	19.22	✅
mccall_model	2024-01-15 07:20	cache	20.69	✅
mccall_model_with_separation	2024-01-15 07:21	cache	12.31	✅
mccall_q	2024-01-15 07:21	cache	24.14	✅
mix_model	2024-01-15 07:22	cache	39.6	✅
mle	2024-01-15 07:22	cache	6.01	✅
multi_hyper	2024-01-15 07:22	cache	26.08	✅
multivariate_normal	2024-01-15 07:22	cache	5.66	✅
navy_captain	2024-01-15 07:23	cache	38.81	✅
newton_method	2024-01-15 07:25	cache	90.15	✅
odu	2024-01-15 07:26	cache	58.12	✅
ols	2024-01-15 07:26	cache	17.24	✅
opt_transport	2024-01-15 07:26	cache	26.95	✅
optgrowth	2024-01-15 07:28	cache	86.67	✅
optgrowth_fast	2024-01-15 07:29	cache	51.68	✅
pandas_panel	2024-01-15 07:29	cache	6.01	✅
perm_income	2024-01-15 07:29	cache	4.81	✅
perm_income_cons	2024-01-15 07:29	cache	12.16	✅
prob_matrix	2024-01-15 07:29	cache	18.15	✅
prob_meaning	2024-01-15 07:31	cache	76.07	✅
qr_decomp	2024-01-15 07:31	cache	1.51	✅
rand_resp	2024-01-15 07:31	cache	3.27	✅
rational_expectations	2024-01-15 07:31	cache	8.93	✅
re_with_feedback	2024-01-15 07:31	cache	13.27	✅
samuelson	2024-01-15 07:31	cache	18.6	✅
scalar_dynam	2024-01-15 07:31	cache	4.63	✅
schelling	2024-01-15 07:31	cache	3.15	✅
short_path	2024-01-15 07:32	cache	1.25	✅
sir_model	2024-01-15 07:32	cache	4.06	✅
status	2024-01-15 07:05	cache	1.27	✅
svd_intro	2024-01-15 07:32	cache	1.65	✅
troubleshooting	2024-01-15 07:05	cache	1.27	✅
two_auctions	2024-01-15 07:32	cache	17.44	✅
uncertainty_traps	2024-01-15 07:32	cache	3.17	✅
util_rand_resp	2024-01-15 07:32	cache	3.78	✅
var_dmd	2024-01-15 07:05	cache	1.27	✅
von_neumann_model	2024-01-15 07:32	cache	2.81	✅
wald_friedman	2024-01-15 07:32	cache	20.21	✅

continues on next page

Table 85.1 – continued from previous page

Document	Modified	Method	Run Time (s)	Status
wealth_dynamics	2024-01-15 07:33	cache	43.69	✅
zreferences	2024-01-15 07:05	cache	1.27	✅

These lectures are built on `linux` instances through `github actions` and `amazon web services (aws)` to enable access to a `gpu`. These lectures are built on a p3.2xlarge that has access to `8 vcpu's`, a `V100 NVIDIA Tesla GPU`, and `61 Gb` of memory.

BIBLIOGRAPHY

[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.

[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.

[And76] Harald Anderson. Estimation of a proportion through randomized response. *International Statistical Review/Revue Internationale de Statistique*, pages 213–217, 1976.

[Apo90] George Apostolakis. The concept of probability in safety assessments of technological systems. *Science*, 250(4986):1359–1364, 1990.

[Axt01] Robert L Axtell. Zipf distribution of us firm sizes. *science*, 293(5536):1818–1820, 2001.

[Bar79] Robert J Barro. On the Determination of the Public Debt. *Journal of Political Economy*, 87(5):940–971, 1979.

[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.

[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.

[Ber75] Dmitri Bertsekas. *Dynamic Programming and Stochastic Control*. Academic Press, New York, 1975.

[Ber97] J. N. Bertsimas, D. & Tsitsiklis. *Introduction to linear optimization*. Athena Scientific, 1997.

[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.

[BK80] Olivier Jean Blanchard and Charles M Kahn. The Solution of Linear Difference Models under Rational Expectations. *Econometrica*, 48(5):1305–1311, July 1980.

[BK22] Steven L. Brunton and J. Nathan Kutz. *Data-Driven Science and Engineering, Second Edition*. Cambridge University Press, New York, 2022.

[BDM+16] Dariusz Buraczewski, Ewa Damek, Thomas Mikosch, and others. *Stochastic models with power-law tails*. Springer, 2016.

[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.

[Cap85] Andrew S Caplin. The variability of aggregate demand with (s, s) inventory policies. *Econometrica*, pages 1395–1409, 1985.

[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.

[Cas65] David Cass. Optimum growth in an aggregative model of capital accumulation. *Review of Economic Studies*, 32(3):233–240, 1965.

[CM88] A Chadhuri and R Mukerjee. *Randomized Response: Theory and Technique*. Marcel Dekker, New York, 1988.

[Cla71] E. Clarke. Multipart pricing of public goods. *Public Choice*, 8:19–33, 1971.

[Col90] Wilbur John Coleman. Solving the Stochastic Growth Model by Policy-Function Iteration. *Journal of Business & Economic Statistics*, 8(1):27–29, 1990.

[DFH06] Steven J Davis, R Jason Faberman, and John Haltiwanger. The flow approach to labor markets: new data sources, micro-macro links and the recent downturn. *Journal of Economic Perspectives*, 2006.

[dF37] Bruno de Finetti. La prevision: ses lois logiques, ses sources subjectives. *Annales de l'Institute Henri Poincare'*, 7:1 – 68, 1937. English translation in Kyburg and Smokler (eds.), \it Studies in Subjective Probability, Wiley, New York, 1964.

[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.

[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.

[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.

[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.

[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.

[EG87] Robert F Engle and Clive W J Granger. Co-integration and Error Correction: Representation, Estimation, and Testing. *Econometrica*, 55(2):251–276, 1987.

[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.

[FSTD15] Pablo Fajgelbaum, Edouard Schaal, and Mathieu Taschereau-Dumouchel. Uncertainty traps. Technical Report, National Bureau of Economic Research, 2015.

[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.

[Fri56] M. Friedman. *A Theory of the Consumption Function*. Princeton University Press, 1956.

[FF98] Milton Friedman and Rose D Friedman. *Two Lucky People*. University of Chicago Press, 1998.

[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.

[Gor95] Geoffrey J Gordon. Stable function approximation in dynamic programming. In *Machine Learning Proceedings 1995*, pages 261–268. Elsevier, 1995.

[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 hoise accident at the wipp. Environmental Evaluation Group, Albuquerque, New Mexico, June 1993. URL: http://www.tomsargent.com/research/EEG-53.pdf.

[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.

[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. *National Bureau of Economic Research Working Paper Series*, 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.

[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.

[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.

[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.

[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.

[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.

[HP92] Hugo A Hopenhayn and Edward C Prescott. Stochastic Monotonicity and Stationary Distributions for Dynamic Economies. *Econometrica*, 60(6):1387–1406, 1992.

[Hu18] Y. Hu, Y. & Guo. *Operations research*. Tsinghua University Press, 5th edition, 2018.

[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.

[Haggstrom02] Olle Häggström. *Finite Markov chains and algorithmic applications*. Volume 52. Cambridge University Press, 2002.

[Janich94] K Jänich. *Linear Algebra*. Springer Undergraduate Texts in Mathematics and Technology. Springer, 1994.

[JYC88] Robert J. Shiller John Y. Campbell. The Dividend-Price Ratio and Expectations of Future Dividends and Discount Factors. *Review of Financial Studies*, 1(3):195–228, 1988.

[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.

[Kam12] Takashi Kamihigashi. Elementary results on solutions to the bellman equation of dynamic programming: existence, uniqueness, and convergence. Technical Report, Kobe University, 2012.

[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.

[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. Chicago, 1965.

[Kre88] David M. Kreps. *Notes on the Theory of Choice*. Westview Press, Boulder, Colorado, 1988.

[Kuh13] Moritz Kuhn. Recursive Equilibria In An Aiyagari-Style Economy With Permanent Income Shocks. *International Economic Review*, 54:807–835, 2013.

[KBBWP16] J. N. Kutz, S. L. Brunton, Brunton B. W, and J. L. Proctor. *Dynamic mode decomposition: data-driven modeling of complex systems*. SIAM, 2016.

[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.

[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.

[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.

[Luc78] Robert E Lucas, Jr. Asset prices in an exchange economy. *Econometrica: Journal of the Econometric Society*, 46(6):1429–1445, 1978.

[LP71] Robert E Lucas, Jr. and Edward C Prescott. Investment under uncertainty. *Econometrica: Journal of the Econometric Society*, pages 659–681, 1971.

[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.

[MS89] 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.

[MdRV10] V Filipe Martins-da-Rocha and Yiannis Vailakis. Existence and Uniqueness of a Fixed Point for Local Contractions. *Econometrica*, 78(3):1127–1141, 2010.

[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.

[MT09] S P Meyn and R L Tweedie. *Markov Chains and Stochastic Stability*. Cambridge University Press, 2009.

[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.

[Nea99] Derek Neal. The Complexity of Job Mobility among Young Men. *Journal of Labor Economics*, 17(2):237–261, 1999.

[NP33] J. Neyman and E. S Pearson. On the problem of the most efficient tests of statistical hypotheses. *Phil. Trans. R. Soc. Lond. A. 231 (694–706)*, pages 289–337, 1933.

[OW69] Guy H. Orcutt and Herbert S. Winokur. First order autoregression: inference, estimation, and prediction. *Econometrica*, 37(1):1–14, 1969.

[Par99] Jonathan A Parker. The Reaction of Household Consumption to Predictable Changes in Social Security Taxes. *American Economic Review*, 89(4):959–973, 1999.

[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.

[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.

[Rya12] Stephen P Ryan. The costs of environmental regulation in a concentrated industry. *Econometrica*, 80(3):1019–1061, 2012.

[Sam39] Paul A. Samuelson. Interactions between the multiplier analysis and the principle of acceleration. *Review of Economic Studies*, 21(2):75–78, 1939.

[Sar77] Thomas J Sargent. The Demand for Money During Hyperinflations under Rational Expectations: I. *International Economic Review*, 18(1):59–82, February 1977.

[Sar87] Thomas J Sargent. *Macroeconomic Theory*. Academic Press, New York, 2nd edition, 1987.

[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.

[Sch69] Thomas C Schelling. Models of Segregation. *American Economic Review*, 59(2):488–493, 1969.

[Sch10] Peter J Schmid. Dynamic mode decomposition of numerical and experimental data. *Journal of fluid mechanics*, 656:5–28, 2010.

[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.

[SLP89] N L Stokey, R E Lucas, and E C Prescott. *Recursive Methods in Economic Dynamics*. Harvard University Press, 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.

[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.

[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.

[Whi83] Charles Whiteman. *Linear Rational Expectations Models: A User's Guide*. University of Minnesota Press, Minneapolis, Minnesota, 1983.

[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.

INDEX

A

A Problem that Stumped Milton Friedman, 943
An Introduction to Job Search, 601
Asset Pricing: Finite State Models, 1263
Autoregressive processes, 409

C

Central Limit Theorem, 193, 198
    Intuition, 198
    Multivariate Case, 203
CLT, 193

D

Dynamic Programming
    Computation, 742, 755
    Shortest Paths, 573
    Theory, 742
    Unbounded Utility, 742
Dynamics in One Dimension, 389

E

Eigenvalues, 35, 51
Eigenvectors, 35, 51
Ergodicity, 421, 435

F

Finite Markov Asset Pricing
    Lucas Tree, 1270
Finite Markov Chains, 421, 422
    Stochastic Matrices, 422

I

Irreducibility and Aperiodicity, 421, 429

J

Job Search VI: On-the-Job Search, 645
Job Search VI: On-the-Job Search, 645

K

Kalman Filter, 541
    Programming Implementation, 550
    Recursive Procedure, 549
Kalman Filter 2, 559
Kesten processes
    heavy tails, 516

L

Lake Model, 1157
Law of Large Numbers, 193, 194
    Illustration, 196
    Multivariate Case, 203
    Proof, 195
Linear Algebra, 35
    Differentiating Linear and Quadratic Forms, 55
    Eigenvalues, 51
    Eigenvectors, 51
    Matrices, 44
    Matrix Norms, 54
    Neumann's Theorem, 55
    Positive Definite Matrices, 55
    SciPy, 51
    Series Expansions, 54
    Spectral Radius, 55
    Vectors, 36
Linear Markov Perfect Equilibria, 1221
Linear State Space Models, 455, 515
    Distributions, 461, 462
    Ergodicity, 468
    Martingale Difference Shocks, 457
    Moments, 461
    Moving Average Representations, 461
    Prediction, 473
    Seasonals, 460
    Stationarity, 468
    Time Trends, 460
    Univariate Autoregressive Processes, 458
    Vector Autoregressions, 459
LLN, 193
LQ Control, 1037
    Infinite Horizon, 1049

Optimality (*Finite Horizon*), 1040

M

Marginal Distributions, 421, 427
Markov Asset Pricing
Overview, 1263
Markov Chains, 422
Calculating Stationary Distributions, 433
Convergence to Stationarity, 434
Cross-Sectional Distributions, 428
Ergodicity, 435
Forecasting Future Values, 436
Future Probabilities, 428
Irreducibility, Aperiodicity, 429
Marginal Distributions, 427
Simulation, 424
Stationary Distributions, 432
Markov Perfect Equilibrium, 1219
Applications, 1223
Background, 1220
Overview, 1219
Markov process, inventory, 445
Matrix
Determinants, 49
Inverse, 49
Maps, 46
Numpy, 45
Operations, 44
Solving Systems of Equations, 46
Modeling
Career Choice, 631
Modeling COVID 19, 25
Models
Harrison Kreps, 1309
Linear State Space, 456
Markov Asset Pricing, 1263
McCall, 586
On-the-Job Search, 645
Permanent Income, 1085, 1101
Pricing, 1264
Schelling's Segregation Model, 1143
Sequential analysis, 943

N

Neumann's Theorem, 55

O

On-the-Job Search
Model, 646
Model Features, 646
Parameterization, 646
Programming Implementation, 647
Solving for Policies, 650
Optimal Growth
Model, 738, 754
Policy Function, 749
Policy Function Approach, 739
Optimal Growth I: The Stochastic Optimal Growth Model, 737
Optimal Growth II: Accelerating the Code with Numba, 753
Optimal Growth III: Time Iteration, 765
Optimal Growth IV: The Endogenous Grid Method, 777
Optimal Savings
Computation, 788, 803
Problem, 786
Programming Implementation, 789

P

Pandas for Panel Data, 1323
Permanent Income II: LQ Techniques, 1101
Permanent Income Model
Hall's Representation, 1093
Savings Problem, 1086
Positive Definite Matrices, 55
Pricing Models, 1263, 1264
Risk Aversion, 1264
Risk-Neutral, 1264
Python
Pandas, 1323
python, 7, 67, 123, 370

R

Rational Expectations Equilibrium, 1185
Competitive Equilbrium (*w. Adjustment Costs*), 1188
Computation, 1191
Definition, 1188
Planning Problem Approach, 1191

S

Schelling Segregation Model, 1143
Spectral Radius, 55
Stability in Linear Rational Expectations Models, 1199
Stationary Distributions, 421, 432
Stochastic Matrices, 422

T

The Income Fluctuation Problem I: Basic Model, 785
The Income Fluctuation Problem II: Stochastic Returns on Assets, 801
The Permanent Income Model, 1085

U

Unbounded Utility, 742

V

Vectors, 35, 36
    Inner Product, 40
    Linear Independence, 43
    Norm, 40
    Operations, 37
    Span, 40