Advanced Quantitative Economics with Python

Thomas J. Sargent & John Stachurski

Aug 04, 2025

CONTENTS

I Tools and Techniques **3**

1 Orthogonal Projections and Their Applications **5**
- 1.1 Overview . . . 5
- 1.2 Key Definitions . . . 6
- 1.3 The Orthogonal Projection Theorem . . . 9
- 1.4 Orthonormal Basis . . . 12
- 1.5 Projection Via Matrix Algebra . . . 13
- 1.6 Least Squares Regression . . . 16
- 1.7 Orthogonalization and Decomposition . . . 17
- 1.8 Exercises . . . 19

2 Continuous State Markov Chains **23**
- 2.1 Overview . . . 23
- 2.2 The Density Case . . . 24
- 2.3 Beyond Densities . . . 30
- 2.4 Stability . . . 31
- 2.5 Exercises . . . 34
- 2.6 Appendix . . . 43

3 Reverse Engineering a la Muth **45**
- 3.1 Friedman (1956) and Muth (1960) . . . 45
- 3.2 A Process for Which Adaptive Expectations are Optimal . . . 46
- 3.3 Some Useful State-Space Math . . . 47
- 3.4 Estimates of Unobservables . . . 48
- 3.5 Relationship of Unobservables to Observables . . . 49
- 3.6 MA and AR Representations . . . 51

4 Discrete State Dynamic Programming **53**
- 4.1 Overview . . . 53
- 4.2 Discrete DPs . . . 54
- 4.3 Solving Discrete DPs . . . 57
- 4.4 Example: A Growth Model . . . 58
- 4.5 Exercises . . . 63
- 4.6 Solutions . . . 64
- 4.7 Appendix: Algorithms . . . 73

II LQ Control **75**

5 Information and Consumption Smoothing **77**

5.1 Overview . . . 77
5.2 Two Representations of One Nonfinancial Income Process . . . 78
5.3 Application of Kalman filter . . . 79
5.4 News Shocks and Less Informative Shocks . . . 80
5.5 Representation of ϵ_t Shock in Terms of Future y_t . . . 81
5.6 Representation in Terms of a_t Shocks . . . 81
5.7 Permanent Income Consumption-Smoothing Model . . . 82
5.8 State Space Representations . . . 82
5.9 Computations . . . 83
5.10 Simulating Income Process and Two Associated Shock Processes . . . 89
5.11 Calculating Innovations in Another Way . . . 90
5.12 Another Invertibility Issue . . . 90

6 Consumption Smoothing with Complete and Incomplete Markets 91
6.1 Overview . . . 91
6.2 Background . . . 92
6.3 Linear State Space Version of Complete Markets Model . . . 92
6.4 Model 1 (Complete Markets) . . . 97
6.5 Model 2 (One-Period Risk-Free Debt Only) . . . 102

7 Tax Smoothing with Complete and Incomplete Markets 105
7.1 Overview . . . 105
7.2 Tax Smoothing with Complete Markets . . . 110
7.3 Returns on State-Contingent Debt . . . 111
7.4 More Finite Markov Chain Tax-Smoothing Examples . . . 115

8 Markov Jump Linear Quadratic Dynamic Programming 143
8.1 Overview . . . 143
8.2 Review of useful LQ dynamic programming formulas . . . 144
8.3 Linked Riccati equations for Markov LQ dynamic programming . . . 144
8.4 Applications . . . 145
8.5 Example 1 . . . 146
8.6 Example 2 . . . 164
8.7 More examples . . . 195

9 How to Pay for a War: Part 1 197
9.1 Overview . . . 197
9.2 Public Finance Questions . . . 198
9.3 Barro (1979) Model . . . 199
9.4 Python Class to Solve Markov Jump Linear Quadratic Control Problems . . . 203
9.5 Barro Model with a Time-varying Interest Rate . . . 204

10 How to Pay for a War: Part 2 207
10.1 Overview . . . 207
10.2 Two example specifications . . . 208
10.3 One- and Two-period Bonds but No Restructuring . . . 208
10.4 Mapping into an LQ Markov Jump Problem . . . 209
10.5 Penalty on Different Issues Across Maturities . . . 211
10.6 A Model with Restructuring . . . 214
10.7 Restructuring as a Markov Jump Linear Quadratic Control Problem . . . 216

11 How to Pay for a War: Part 3 221
11.1 Overview . . . 221
11.2 Roll-Over Risk . . . 221
11.3 A Dead End . . . 222

11.4 Better Representation of Roll-Over Risk . . . 222

12 Optimal Taxation in an LQ Economy **227**
12.1 Overview . . . 227
12.2 The Ramsey Problem . . . 228
12.3 Implementation . . . 234
12.4 Examples . . . 239
12.5 Exercises . . . 245

III Multiple Agent Models 249

13 Default Risk and Income Fluctuations **251**
13.1 Overview . . . 251
13.2 Structure . . . 252
13.3 Equilibrium . . . 254
13.4 Computation . . . 255
13.5 Results . . . 260
13.6 Exercises . . . 264

14 Globalization and Cycles **271**
14.1 Overview . . . 271
14.2 Key Ideas . . . 272
14.3 Model . . . 272
14.4 Simulation . . . 275
14.5 Exercises . . . 285

15 Coase's Theory of the Firm **289**
15.1 Overview . . . 289
15.2 The Model . . . 291
15.3 Equilibrium . . . 293
15.4 Existence, Uniqueness and Computation of Equilibria . . . 294
15.5 Implementation . . . 296
15.6 Exercises . . . 299

16 Composite Sorting **303**
16.1 Overview . . . 303
16.2 Setup . . . 304
16.3 Characterization of primal solution . . . 308
16.4 Solving primal problem . . . 325
16.5 Examples . . . 329
16.6 Dual Solution . . . 338
16.7 Application . . . 347

IV Dynamic Linear Economies 357

17 Recursive Models of Dynamic Linear Economies **359**
17.1 A Suite of Models . . . 359
17.2 Econometrics . . . 376
17.3 Dynamic Demand Curves and Canonical Household Technologies . . . 378
17.4 Gorman Aggregation and Engel Curves . . . 379
17.5 Partial Equilibrium . . . 380
17.6 Equilibrium Investment Under Uncertainty . . . 381
17.7 A Rosen-Topel Housing Model . . . 382

17.8 Cattle Cycles . . . 382
17.9 Models of Occupational Choice and Pay . . . 383
17.10 Permanent Income Models . . . 384
17.11 Gorman Heterogeneous Households . . . 385
17.12 Non-Gorman Heterogeneous Households . . . 386

18 Growth in Dynamic Linear Economies **389**
18.1 Common Structure . . . 389
18.2 A Planning Problem . . . 390
18.3 Example Economies . . . 391

19 Lucas Asset Pricing Using DLE **401**
19.1 Asset Pricing Equations . . . 402
19.2 Asset Pricing Simulations . . . 402

20 IRFs in Hall Models **409**
20.1 Example 1: Hall (1978) . . . 409
20.2 Example 2: Higher Adjustment Costs . . . 412
20.3 Example 3: Durable Consumption Goods . . . 415

21 Permanent Income Model using the DLE Class **419**
21.1 The Permanent Income Model . . . 419

22 Rosen Schooling Model **423**
22.1 A One-Occupation Model . . . 423
22.2 Mapping into HS2013 Framework . . . 424

23 Cattle Cycles **429**
23.1 The Model . . . 429
23.2 Mapping into HS2013 Framework . . . 430

24 Shock Non Invertibility **437**
24.1 Overview . . . 437
24.2 Model . . . 438
24.3 Code . . . 440

V Risk, Model Uncertainty, and Robustness **443**

25 Risk and Model Uncertainty **445**
25.1 Overview . . . 445
25.2 Basic objects . . . 446
25.3 Five preference specifications . . . 449
25.4 Expected utility . . . 450
25.5 Constraint preferences . . . 450
25.6 Multiplier preferences . . . 452
25.7 Risk-sensitive preferences . . . 452
25.8 Ex post Bayesian preferences . . . 455
25.9 Comparing preferences . . . 455
25.10 Risk aversion and misspecification aversion . . . 457
25.11 Indifference curves . . . 458
25.12 State price deflators . . . 461
25.13 Iso-utility and iso-entropy curves and expansion paths . . . 464
25.14 Bounds on expected utility . . . 465
25.15 Why entropy? . . . 467

26 Etymology of Entropy **469**
26.1 Information Theory . . . 469
26.2 A Measure of Unpredictability . . . 470
26.3 Mathematical Properties of Entropy . . . 470
26.4 Conditional Entropy . . . 471
26.5 Independence as Maximum Conditional Entropy . . . 471
26.6 Thermodynamics . . . 472
26.7 Statistical Divergence . . . 472
26.8 Continuous distributions . . . 472
26.9 Relative entropy and Gaussian distributions . . . 473
26.10 Von Neumann Entropy . . . 473
26.11 Backus-Chernov-Zin Entropy . . . 474
26.12 Wiener-Kolmogorov Prediction Error Formula as Entropy . . . 474
26.13 Multivariate Processes . . . 475
26.14 Frequency Domain Robust Control . . . 475
26.15 Relative Entropy for a Continuous Random Variable . . . 476

27 Robustness **479**
27.1 Overview . . . 479
27.2 The Model . . . 482
27.3 Constructing More Robust Policies . . . 483
27.4 Robustness as Outcome of a Two-Person Zero-Sum Game . . . 485
27.5 The Stochastic Case . . . 489
27.6 Implementation . . . 491
27.7 Application . . . 491
27.8 Appendix . . . 496

28 Robust Markov Perfect Equilibrium **499**
28.1 Overview . . . 499
28.2 Linear Markov Perfect Equilibria with Robust Agents . . . 500
28.3 Application . . . 503

VI Time Series Models **519**

29 Covariance Stationary Processes **521**
29.1 Overview . . . 521
29.2 Introduction . . . 522
29.3 Spectral Analysis . . . 526
29.4 Implementation . . . 534

30 Estimation of Spectra **543**
30.1 Overview . . . 543
30.2 Periodograms . . . 543
30.3 Smoothing . . . 546
30.4 Exercises . . . 552

31 Additive and Multiplicative Functionals **559**
31.1 Overview . . . 559
31.2 A Particular Additive Functional . . . 560
31.3 Dynamics . . . 561
31.4 Code . . . 573
31.5 More About the Multiplicative Martingale . . . 576

32 Classical Control with Linear Algebra **583**

32.1 Overview . . . 583
32.2 A Control Problem . . . 584
32.3 Finite Horizon Theory . . . 585
32.4 Infinite Horizon Limit . . . 589
32.5 Undiscounted Problems . . . 591
32.6 Implementation . . . 593
32.7 Exercises . . . 601

33 Classical Prediction and Filtering With Linear Algebra 603
33.1 Overview . . . 603
33.2 Finite Dimensional Prediction . . . 604
33.3 Combined Finite Dimensional Control and Prediction . . . 615
33.4 Infinite Horizon Prediction and Filtering Problems . . . 616
33.5 Exercises . . . 620

34 Knowing the Forecasts of Others 623
34.1 Introduction . . . 623
34.2 The Setting . . . 624
34.3 Tactics . . . 625
34.4 Equilibrium Conditions . . . 626
34.5 Equilibrium with θ_t stochastic but observed at t . . . 628
34.6 Guess-and-Verify Tactic . . . 631
34.7 Equilibrium with One Noisy Signal on θ_t . . . 631
34.8 Equilibrium with Two Noisy Signals on θ_t . . . 637
34.9 Key Step . . . 641
34.10 An observed common shock benchmark . . . 641
34.11 Comparison of All Signal Structures . . . 643
34.12 Notes on History of the Problem . . . 645

VII Asset Pricing and Finance 647

35 Asset Pricing II: The Lucas Asset Pricing Model 649
35.1 Overview . . . 649
35.2 The Lucas Model . . . 649
35.3 Exercises . . . 656

36 Elementary Asset Pricing Theory 659
36.1 Overview . . . 659
36.2 Key Equation . . . 660
36.3 Implications of Key Equation . . . 660
36.4 Expected Return - Beta Representation . . . 661
36.5 Mean-Variance Frontier . . . 663
36.6 Sharpe Ratios and the Price of Risk . . . 666
36.7 Mathematical Structure of Frontier . . . 666
36.8 Multi-factor Models . . . 666
36.9 Empirical Implementations . . . 667
36.10 Exercises . . . 668

37 Two Modifications of Mean-Variance Portfolio Theory 673
37.1 Overview . . . 673
37.2 Mean-Variance Portfolio Choice . . . 674
37.3 Estimating Mean and Variance . . . 674
37.4 Black-Litterman Starting Point . . . 675
37.5 Details . . . 676

37.6 Adding Views . . . 678
37.7 Bayesian Interpretation . . . 680
37.8 Curve Decolletage . . . 681
37.9 Black-Litterman Recommendation as Regularization . . . 685
37.10 A Robust Control Operator . . . 687
37.11 A Robust Mean-Variance Portfolio Model . . . 688
37.12 Appendix . . . 689
37.13 Special Case – IID Sample . . . 690
37.14 Dependence and Sampling Frequency . . . 690
37.15 Frequency and the Mean Estimator . . . 692

38 Irrelevance of Capital Structures with Complete Markets **697**
38.1 Introduction . . . 697
38.2 Competitive equilibrium . . . 701
38.3 Code . . . 707

39 Equilibrium Capital Structures with Incomplete Markets **717**
39.1 Introduction . . . 717
39.2 Asset Markets . . . 720
39.3 Equilibrium verification . . . 724
39.4 Pseudo Code . . . 724
39.5 Code . . . 726
39.6 Examples . . . 736
39.7 A picture worth a thousand words . . . 753

VIII Dynamic Programming Squared **755**

40 Optimal Unemployment Insurance **757**
40.1 Overview . . . 757
40.2 Shavell and Weiss's Model . . . 757
40.3 Private Information . . . 760
40.4 Outcomes . . . 767

41 Stackelberg Plans **771**
41.1 Overview . . . 771
41.2 Duopoly . . . 771
41.3 Stackelberg Problem . . . 774
41.4 Two Bellman Equations . . . 776
41.5 Stackelberg Plan for Duopoly . . . 777
41.6 Recursive Representation of Stackelberg Plan . . . 779
41.7 Dynamic Programming and Time Consistency of Follower's Problem . . . 780
41.8 Computing Stackelberg Plan . . . 782
41.9 Time Series for Price and Quantities . . . 783
41.10 Time Inconsistency of Stackelberg Plan . . . 785
41.11 Recursive Formulation of Follower's Problem . . . 786
41.12 Markov Perfect Equilibrium . . . 791
41.13 Comparing Markov Perfect Equilibrium and Stackelberg Outcome . . . 793

42 Machine Learning a Ramsey Plan **795**
42.1 Introduction . . . 795
42.2 The Model . . . 796
42.3 Model Components . . . 796
42.4 Parameters and Variables . . . 798
42.5 Approximation and Truncation parameter T . . . 799

42.6 A Gradient Descent Algorithm . . . 800
42.7 A More Structured ML Algorithm . . . 804
42.8 Continuation Values . . . 811
42.9 Adding Some Human Intelligence . . . 814
42.10 What has Machine Learning Taught Us? . . . 820

43 Time Inconsistency of Ramsey Plans **823**
43.1 Overview . . . 823
43.2 Model Components . . . 824
43.3 Friedman's Optimal Rate of Deflation . . . 825
43.4 Calvo's Distortion . . . 826
43.5 Structure . . . 827
43.6 Three Timing Protocols . . . 828
43.7 Note on Dynamic Programming Squared . . . 828
43.8 A Ramsey Planner . . . 829
43.9 Representation of Ramsey Plan . . . 831
43.10 Multiple roles of θ_t . . . 832
43.11 Time inconsistency . . . 832
43.12 Constrained-to-Constant-Growth-Rate Ramsey Plan . . . 832
43.13 Markov Perfect Governments . . . 833
43.14 Outcomes under Three Timing Protocols . . . 834
43.15 Ramsey Planner's Value Function . . . 839
43.16 Perturbing Model Parameters . . . 841

44 Sustainable Plans for a Calvo Model **847**
44.1 Overview . . . 847
44.2 Model Components . . . 847
44.3 Another Timing Protocol . . . 848
44.4 Sustainable or Credible Plan . . . 850
44.5 Whose Plan is It? . . . 857

45 Optimal Taxation with State-Contingent Debt **859**
45.1 Overview . . . 859
45.2 A Competitive Equilibrium with Distorting Taxes . . . 860
45.3 Recursive Formulation of the Ramsey Problem . . . 871
45.4 Examples . . . 878

46 Optimal Taxation without State-Contingent Debt **891**
46.1 Overview . . . 891
46.2 Competitive Equilibrium with Distorting Taxes . . . 892
46.3 Recursive Version of AMSS Model . . . 900
46.4 Examples . . . 907

47 Fluctuating Interest Rates Deliver Fiscal Insurance **919**
47.1 Overview . . . 919
47.2 Forces at Work . . . 920
47.3 Logical Flow of Lecture . . . 921
47.4 Example Economy . . . 923
47.5 Reverse Engineering Strategy . . . 933
47.6 Code for Reverse Engineering . . . 934
47.7 Short Simulation for Reverse-engineered: Initial Debt . . . 935
47.8 Long Simulation . . . 944
47.9 BEGS Approximations of Limiting Debt and Convergence Rate . . . 946

48 Fiscal Risk and Government Debt **951**

48.1 Overview . . . 951
48.2 The Economy . . . 952
48.3 Long Simulation . . . 953
48.4 Asymptotic Mean and Rate of Convergence . . . 978

49 Competitive Equilibria of a Model of Chang **987**
49.1 Overview . . . 987
49.2 Decisions . . . 988
49.3 Competitive Equilibrium . . . 991
49.4 Inventory of Objects in Play . . . 992
49.5 Analysis . . . 993
49.6 Calculating all Promise-Value Pairs in CE . . . 996
49.7 Solving a Continuation Ramsey Planner's Bellman Equation . . . 1012

50 Credible Government Policies in a Model of Chang **1019**
50.1 Overview . . . 1019
50.2 The Setting . . . 1020
50.3 Calculating the Set of Sustainable Promise-Value Pairs . . . 1026

IX Other **1043**

51 Troubleshooting **1045**
51.1 Fixing Your Local Environment . . . 1045
51.2 Reporting an Issue . . . 1046

52 References **1047**

53 Execution Statistics **1049**

Bibliography **1051**

Proof Index **1057**

Index **1059**

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

- Tools and Techniques
  - *Orthogonal Projections and Their Applications*
  - *Continuous State Markov Chains*
  - *Reverse Engineering a la Muth*
  - *Discrete State Dynamic Programming*
- LQ Control
  - *Information and Consumption Smoothing*
  - *Consumption Smoothing with Complete and Incomplete Markets*
  - *Tax Smoothing with Complete and Incomplete Markets*
  - *Markov Jump Linear Quadratic Dynamic Programming*
  - *How to Pay for a War: Part 1*
  - *How to Pay for a War: Part 2*
  - *How to Pay for a War: Part 3*
  - *Optimal Taxation in an LQ Economy*
- Multiple Agent Models
  - *Default Risk and Income Fluctuations*
  - *Globalization and Cycles*
  - *Coase's Theory of the Firm*
  - *Composite Sorting*
- Dynamic Linear Economies
  - *Recursive Models of Dynamic Linear Economies*
  - *Growth in Dynamic Linear Economies*
  - *Lucas Asset Pricing Using DLE*
  - *IRFs in Hall Models*
  - *Permanent Income Model using the DLE Class*
  - *Rosen Schooling Model*
  - *Cattle Cycles*
  - *Shock Non Invertibility*
- Risk, Model Uncertainty, and Robustness
  - *Risk and Model Uncertainty*
  - *Etymology of Entropy*
  - *Robustness*
  - *Robust Markov Perfect Equilibrium*
- Time Series Models
  - *Covariance Stationary Processes*

  - *Estimation of Spectra*
  - *Additive and Multiplicative Functionals*
  - *Classical Control with Linear Algebra*
  - *Classical Prediction and Filtering With Linear Algebra*
  - *Knowing the Forecasts of Others*
- Asset Pricing and Finance
  - *Asset Pricing II: The Lucas Asset Pricing Model*
  - *Elementary Asset Pricing Theory*
  - *Two Modifications of Mean-Variance Portfolio Theory*
  - *Irrelevance of Capital Structures with Complete Markets*
  - *Equilibrium Capital Structures with Incomplete Markets*
- Dynamic Programming Squared
  - *Optimal Unemployment Insurance*
  - *Stackelberg Plans*
  - *Machine Learning a Ramsey Plan*
  - *Time Inconsistency of Ramsey Plans*
  - *Sustainable Plans for a Calvo Model*
  - *Optimal Taxation with State-Contingent Debt*
  - *Optimal Taxation without State-Contingent Debt*
  - *Fluctuating Interest Rates Deliver Fiscal Insurance*
  - *Fiscal Risk and Government Debt*
  - *Competitive Equilibria of a Model of Chang*
  - *Credible Government Policies in a Model of Chang*
- Other
  - *Troubleshooting*
  - *References*
  - *Execution Statistics*

Part I

Tools and Techniques

CHAPTER

ONE

ORTHOGONAL PROJECTIONS AND THEIR APPLICATIONS

1.1 Overview

Orthogonal projection is a cornerstone of vector space methods, with many diverse applications.

These include

- Least squares projection, also known as linear regression
- Conditional expectations for multivariate normal (Gaussian) distributions
- Gram–Schmidt orthogonalization
- QR decomposition
- Orthogonal polynomials
- etc

In this lecture, we focus on

- key ideas
- least squares regression

We'll require the following imports:

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

1.1.1 Further Reading

For background and foundational concepts, see our lecture on linear algebra.

For more proofs and greater theoretical detail, see A Primer in Econometric Theory.

For a complete set of proofs in a general setting, see, for example, [Roman, 2005].

For an advanced treatment of projection in the context of least squares prediction, see this book chapter.

1.2 Key Definitions

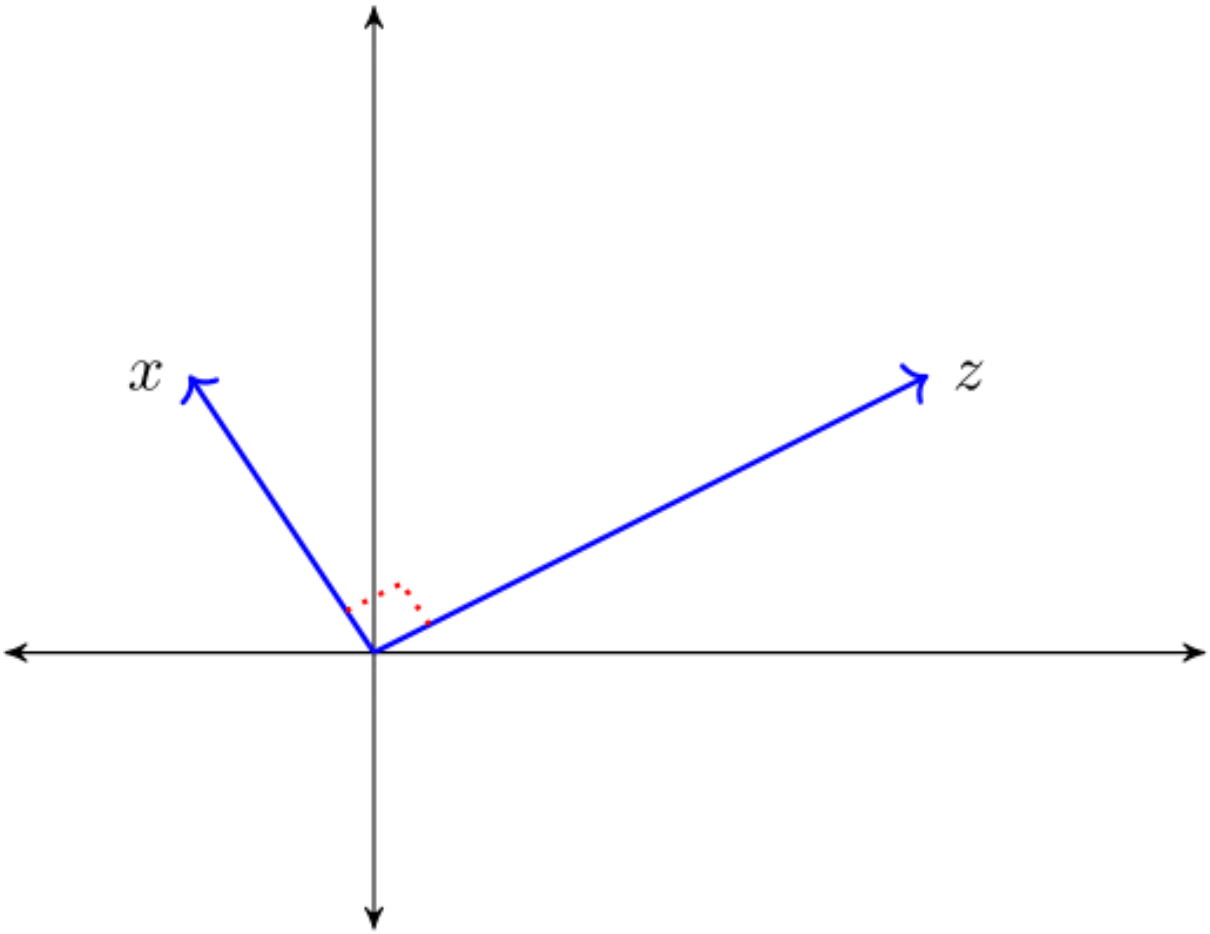
Assume $x, z \in \mathbb{R}^n$.

Define $\langle x, z \rangle = \sum_i x_i z_i$.

Recall $\|x\|^2 = \langle x, x \rangle$.

The **law of cosines** states that $\langle x, z \rangle = \|x\| \|z\| \cos(\theta)$ where θ is the angle between the vectors x and z.

When $\langle x, z \rangle = 0$, then $\cos(\theta) = 0$ and x and z are said to be **orthogonal** and we write $x \perp z$.



For a linear subspace $S \subset \mathbb{R}^n$, we call $x \in \mathbb{R}^n$ **orthogonal to** S if $x \perp z$ for all $z \in S$, and write $x \perp S$.

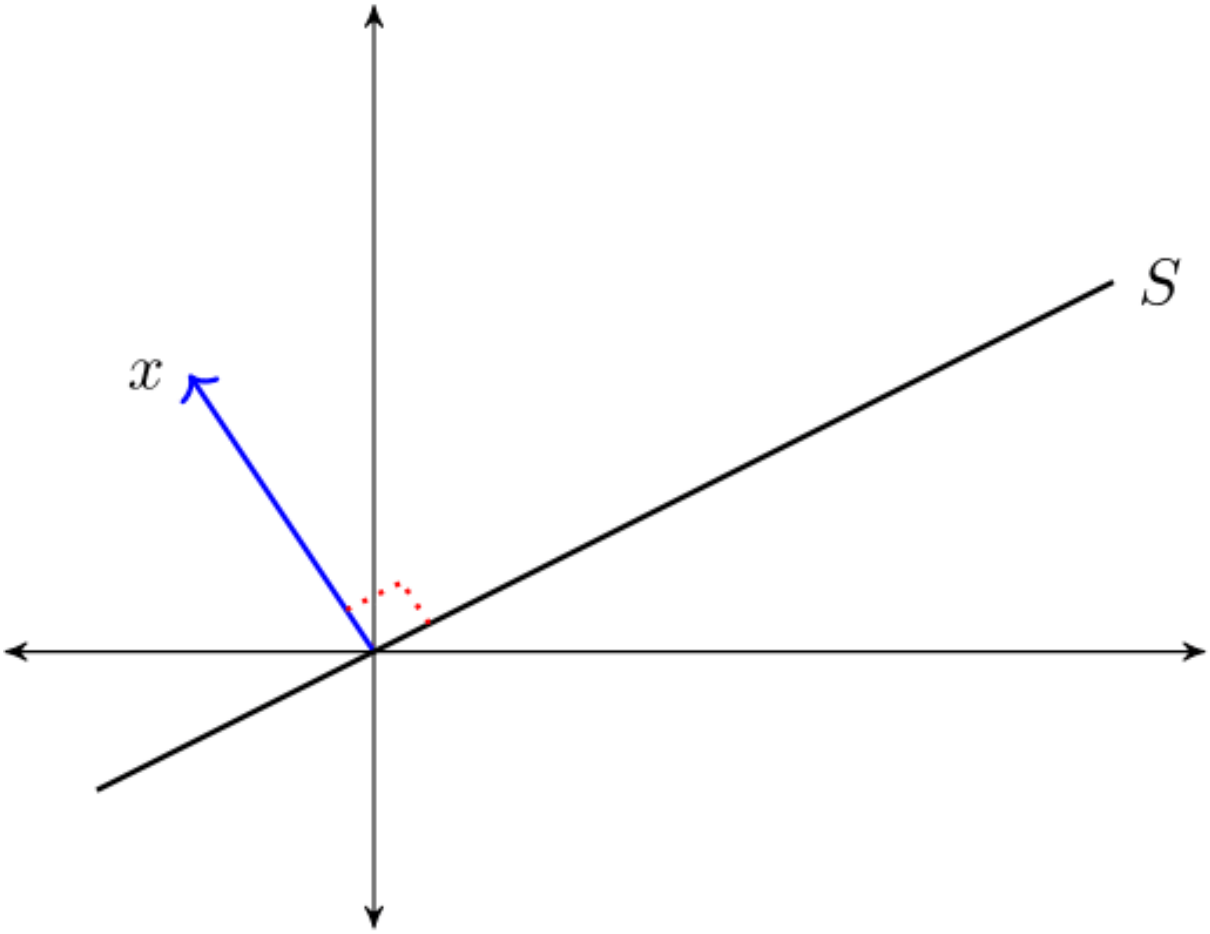
The **orthogonal complement** of linear subspace $S \subset \mathbb{R}^n$ is the set $S^\perp := \{x \in \mathbb{R}^n \,:\, x \perp S\}$.

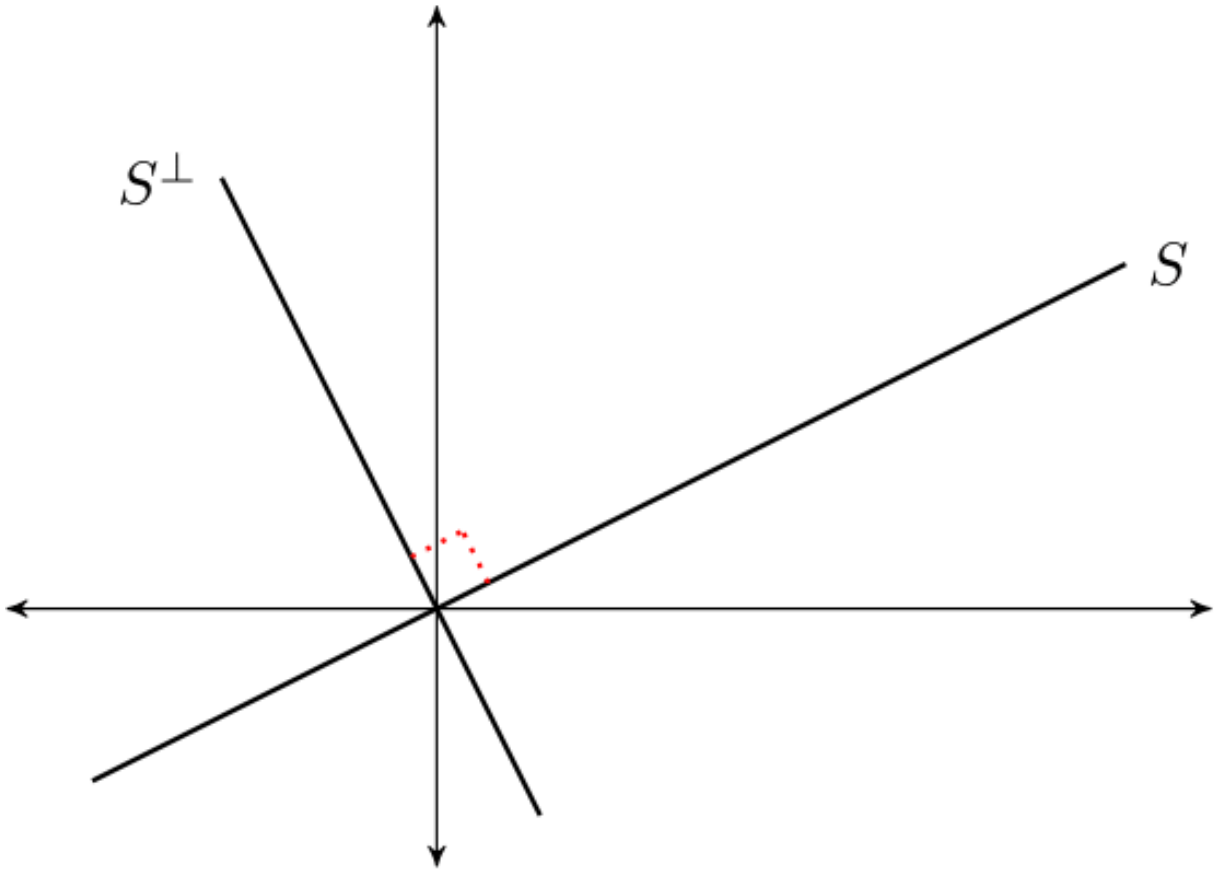
$S^\perp$ is a linear subspace of $\mathbb{R}^n$

- To see this, fix $x, y \in S^\perp$ and $\alpha, \beta \in \mathbb{R}$.
- Observe that if $z \in S$, then

$$\langle \alpha x + \beta y, z \rangle = \alpha \langle x, z \rangle + \beta \langle y, z \rangle = \alpha \times 0 + \beta \times 0 = 0$$

- Hence $\alpha x + \beta y \in S^\perp$, as was to be shown


S
x


$S^\perp$
S

A set of vectors $\{x_1, \ldots, x_k\} \subset \mathbb{R}^n$ is called an **orthogonal set** if $x_i \perp x_j$ whenever $i \neq j$.

If $\{x_1, \ldots, x_k\}$ is an orthogonal set, then the **Pythagorean Law** states that

$$\|x_1 + \cdots + x_k\|^2 = \|x_1\|^2 + \cdots + \|x_k\|^2$$

For example, when $k = 2$, $x_1 \perp x_2$ implies

$$\|x_1 + x_2\|^2 = \langle x_1 + x_2, x_1 + x_2 \rangle = \langle x_1, x_1 \rangle + 2\langle x_2, x_1 \rangle + \langle x_2, x_2 \rangle = \|x_1\|^2 + \|x_2\|^2$$

1.2.1 Linear Independence vs Orthogonality

If $X \subset \mathbb{R}^n$ is an orthogonal set and $0 \notin X$, then X is linearly independent.

Proving this is a nice exercise.

While the converse is not true, a kind of partial converse holds, as we'll *see below*.

1.3 The Orthogonal Projection Theorem

What vector within a linear subspace of $\mathbb{R}^n$ best approximates a given vector in $\mathbb{R}^n$?

The next theorem answers this question.

Theorem 1.3.1 (Orthogonal Projection Theorem)

Given $y \in \mathbb{R}^n$ and linear subspace $S \subset \mathbb{R}^n$, there exists a unique solution to the minimization problem

$$\hat{y} := \operatorname*{arg\,min}_{z \in S} \|y - z\|$$

The minimizer $\hat{y}$ is the unique vector in $\mathbb{R}^n$ that satisfies

- $\hat{y} \in S$
- $y - \hat{y} \perp S$

The vector $\hat{y}$ is called the **orthogonal projection** of y onto S.

The next figure provides some intuition

1.3.1 Proof of Sufficiency

We'll omit the full proof.

But we will prove sufficiency of the asserted conditions.

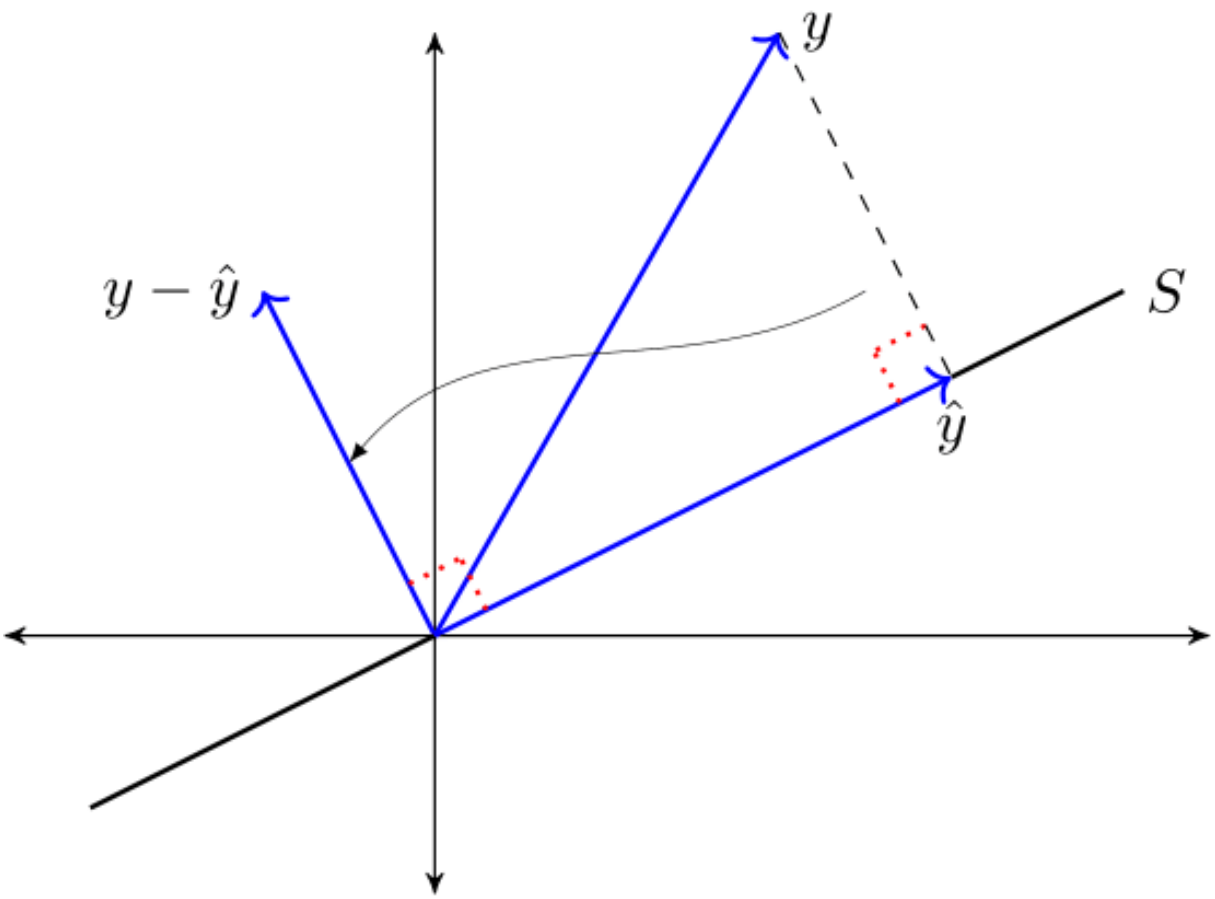
To this end, let $y \in \mathbb{R}^n$ and let S be a linear subspace of $\mathbb{R}^n$.

Let $\hat{y}$ be a vector in $\mathbb{R}^n$ such that $\hat{y} \in S$ and $y - \hat{y} \perp S$.

Let z be any other point in S and use the fact that S is a linear subspace to deduce

$$\|y - z\|^2 = \|(y - \hat{y}) + (\hat{y} - z)\|^2 = \|y - \hat{y}\|^2 + \|\hat{y} - z\|^2$$

Hence $\|y - z\| \geq \|y - \hat{y}\|$, which completes the proof.


y
$y - \hat{y}$
S
$\hat{y}$

1.3.2 Orthogonal Projection as a Mapping

For a linear space Y and a fixed linear subspace S, we have a functional relationship

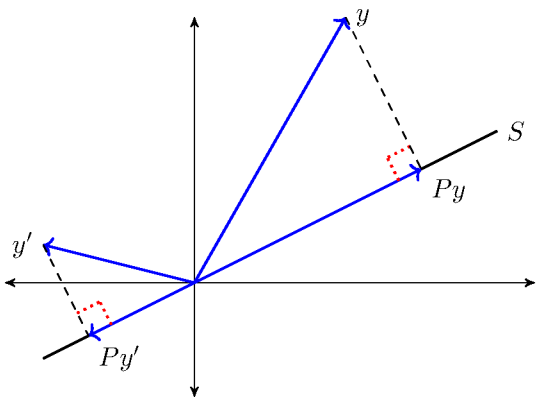
$$y \in Y \;\mapsto \text{ its orthogonal projection } \hat{y} \in S$$

By the *Theorem 1.3.1*, this is a well-defined mapping or *operator* from $\mathbb{R}^n$ to $\mathbb{R}^n$.

In what follows we denote this operator by a matrix P

- Py represents the projection $\hat{y}$.
- This is sometimes expressed as $\hat{E}_S y = Py$, where $\hat{E}$ denotes a **wide-sense expectations operator** and the subscript S indicates that we are projecting y onto the linear subspace S.

The operator P is called the **orthogonal projection mapping onto** S.



It is immediate from the *Theorem 1.3.1* that for any $y \in \mathbb{R}^n$

1. $Py \in S$ and
2. $y - Py \perp S$

From this, we can deduce additional useful properties, such as

1. $\|y\|^2 = \|Py\|^2 + \|y - Py\|^2$ and
2. $\|Py\| \leq \|y\|$

For example, to prove 1, observe that $y = Py + y - Py$ and apply the Pythagorean law.

Orthogonal Complement

Let $S \subset \mathbb{R}^n$.

The **orthogonal complement** of S is the linear subspace $S^\perp$ that satisfies $x_1 \perp x_2$ for every $x_1 \in S$ and $x_2 \in S^\perp$.

Let Y be a linear space with linear subspace S and its orthogonal complement $S^\perp$.

We write

$$Y = S \oplus S^\perp$$

to indicate that for every $y \in Y$ there is unique $x_1 \in S$ and a unique $x_2 \in S^\perp$ such that $y = x_1 + x_2$.

Moreover, $x_1 = \hat{E}_S y$ and $x_2 = y - \hat{E}_S y$.

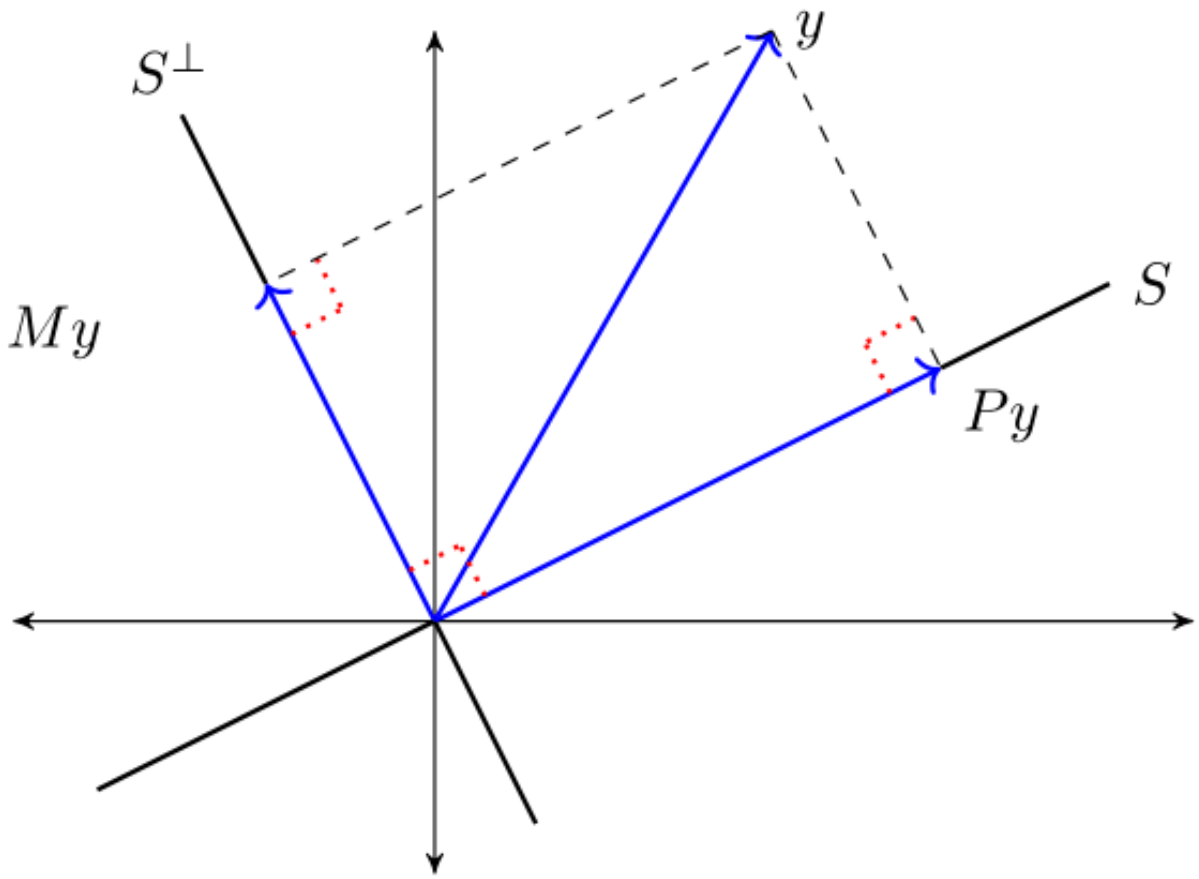
This amounts to another version of the *Theorem 1.3.1*:

Theorem 1.3.2 (Orthogonal Projection Theorem (another version))

If S is a linear subspace of $\mathbb{R}^n$, $\hat{E}_S y = Py$ and $\hat{E}_{S^\perp} y = My$, then

$$Py \perp My \quad \text{and} \quad y = Py + My \quad \text{for all } y \in \mathbb{R}^n$$

The next figure illustrates



1.4 Orthonormal Basis

An orthogonal set of vectors $O \subset \mathbb{R}^n$ is called an **orthonormal set** if $\|u\| = 1$ for all $u \in O$.

Let S be a linear subspace of $\mathbb{R}^n$ and let $O \subset S$.

If O is orthonormal and span $O = S$, then O is called an **orthonormal basis** of S.

O is necessarily a basis of S (being independent by orthogonality and the fact that no element is the zero vector).

One example of an orthonormal set is the canonical basis $\{e_1, \ldots, e_n\}$ that forms an orthonormal basis of $\mathbb{R}^n$, where e_i is the i th unit vector.

If $\{u_1, \ldots, u_k\}$ is an orthonormal basis of linear subspace S, then

$$x = \sum_{i=1}^{k} \langle x, u_i \rangle u_i \quad \text{for all} \quad x \in S$$

To see this, observe that since $x \in \text{span}\{u_1, \ldots, u_k\}$, we can find scalars $\alpha_1, \ldots, \alpha_k$ that verify

$$x = \sum_{j=1}^{k} \alpha_j u_j \tag{1.1}$$

Taking the inner product with respect to u_i gives

$$\langle x, u_i \rangle = \sum_{j=1}^{k} \alpha_j \langle u_j, u_i \rangle = \alpha_i$$

Combining this result with (1.1) verifies the claim.

1.4.1 Projection onto an Orthonormal Basis

When a subspace onto which we project is orthonormal, computing the projection simplifies:

Theorem 1.4.1

If $\{u_1, \ldots, u_k\}$ is an orthonormal basis for S, then

$$Py = \sum_{i=1}^{k} \langle y, u_i \rangle u_i, \quad \forall \; y \in \mathbb{R}^n \tag{1.2}$$

```
```{prf:proof}  Fix $y \in \mathbb{R}^n$ and let $P y$ be  defined as in {eq}`exp_for_
↪op`.

Clearly, $P y \in S$.

We claim that $y - P y \perp S$ also holds.

It suffices to show that $y - P y \perp u_i$ for any basis vector $u_i$.

This is true because

$$
\left\langle y - \sum_{i=1}^k \langle y, u_i \rangle u_i, u_j \right\rangle
= \langle y, u_j \rangle  - \sum_{i=1}^k \langle y, u_i \rangle
\langle u_i, u_j  \rangle = 0
$$
```

(Why is this sufficient to establish the claim that $y - Py \perp S$?)

1.5 Projection Via Matrix Algebra

Let S be a linear subspace of $\mathbb{R}^n$ and let $y \in \mathbb{R}^n$.

We want to compute the matrix P that verifies

$$\hat{E}_S y = Py$$

Evidently Py is a linear function from $y \in \mathbb{R}^n$ to $Py \in \mathbb{R}^n$.
```

This reference is useful.

Theorem 1.5.1

Let the columns of $n \times k$ matrix X form a basis of S. Then

$$P = X(X'X)^{-1}X'$$

Proof. Given arbitrary $y \in \mathbb{R}^n$ and $P = X(X'X)^{-1}X'$, our claim is that

1. $Py \in S$, and
2. $y - Py \perp S$

Claim 1 is true because

$$Py = X(X'X)^{-1}X'y = Xa \quad \text{when} \quad a := (X'X)^{-1}X'y$$

An expression of the form Xa is precisely a linear combination of the columns of X and hence an element of S.

Claim 2 is equivalent to the statement

$$y - X(X'X)^{-1}X'y \perp Xb \quad \text{for all} \quad b \in \mathbb{R}^K$$

To verify this, notice that if $b \in \mathbb{R}^K$, then

$$(Xb)'[y - X(X'X)^{-1}X'y] = b'[X'y - X'y] = 0$$

The proof is now complete.

1.5.1 Starting with the Basis

It is common in applications to start with $n \times k$ matrix X with linearly independent columns and let

$$S := \operatorname{span} X := \operatorname{span}\{\operatorname*{col}_1 X, \ldots, \operatorname*{col}_k X\}$$

Then the columns of X form a basis of S.

From the *Theorem 1.5.1*, $P = X(X'X)^{-1}X'y$ projects y onto S.

In this context, P is often called the **projection matrix**

- The matrix $M = I - P$ satisfies $My = \hat{E}_{S^\perp} y$ and is sometimes called the **annihilator matrix**.

1.5.2 The Orthonormal Case

Suppose that U is $n \times k$ with orthonormal columns.

Let $u_i := \operatorname{col}_i U$ for each i, let $S := \operatorname{span} U$ and let $y \in \mathbb{R}^n$.

We know that the projection of y onto S is

$$Py = U(U'U)^{-1}U'y$$

Since U has orthonormal columns, we have $U'U = I$.

Hence

$$Py = UU'y = \sum_{i=1}^{k} \langle u_i, y \rangle u_i$$

We have recovered our earlier result about projecting onto the span of an orthonormal basis.

1.5.3 Application: Overdetermined Systems of Equations

Let $y \in \mathbb{R}^n$ and let X be $n \times k$ with linearly independent columns.

Given X and y, we seek $b \in \mathbb{R}^k$ that satisfies the system of linear equations $Xb = y$.

If $n > k$ (more equations than unknowns), then the system is said to be **overdetermined**.

Intuitively, we may not be able to find a b that satisfies all n equations.

The best approach here is to

- Accept that an exact solution may not exist.
- Look instead for an approximate solution.

By approximate solution, we mean a $b \in \mathbb{R}^k$ such that Xb is close to y.

The next theorem shows that a best approximation is well defined and unique.

The proof uses the *Theorem 1.3.1*.

Theorem 1.5.2

The unique minimizer of $\|y - Xb\|$ over $b \in \mathbb{R}^k$ is

$$\hat{\beta} := (X'X)^{-1}X'y$$

Proof. Note that

$$X\hat{\beta} = X(X'X)^{-1}X'y = Py$$

Since Py is the orthogonal projection onto span(X) we have

$$\|y - Py\| \leq \|y - z\| \text{ for any } z \in \text{span}(X)$$

Because $Xb \in \text{span}(X)$

$$\|y - X\hat{\beta}\| \leq \|y - Xb\| \text{ for any } b \in \mathbb{R}^k$$

This is what we aimed to show.

1.6 Least Squares Regression

Let's apply the theory of orthogonal projection to least squares regression.

This approach provides insights about many geometric properties of linear regression.

We treat only some examples.

1.6.1 Squared Risk Measures

Given pairs $(x, y) \in \mathbb{R}^K \times \mathbb{R}$, consider choosing $f\colon \mathbb{R}^K \to \mathbb{R}$ to minimize the **risk**

$$R(f) := \mathbb{E}\left[(y - f(x))^2\right]$$

If probabilities and hence $\mathbb{E}$ are unknown, we cannot solve this problem directly.

However, if a sample is available, we can estimate the risk with the **empirical risk**:

$$\min_{f \in \mathcal{F}} \frac{1}{N} \sum_{n=1}^{N} (y_n - f(x_n))^2$$

Minimizing this expression is called **empirical risk minimization**.

The set $\mathcal{F}$ is sometimes called the hypothesis space.

The theory of statistical learning tells us that to prevent overfitting we should take the set $\mathcal{F}$ to be relatively simple.

If we let $\mathcal{F}$ be the class of linear functions, the problem is

$$\min_{b \in \mathbb{R}^K} \sum_{n=1}^{N} (y_n - b' x_n)^2$$

This is the sample **linear least squares problem**.

1.6.2 Solution

Define the matrices

$$y := \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{pmatrix}, \quad x_n := \begin{pmatrix} x_{n1} \\ x_{n2} \\ \vdots \\ x_{nK} \end{pmatrix} = n\text{-th obs on all regressors}$$

and

$$X := \begin{pmatrix} x_1' \\ x_2' \\ \vdots \\ x_N' \end{pmatrix} =: \begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1K} \\ x_{21} & x_{22} & \cdots & x_{2K} \\ \vdots & \vdots & & \vdots \\ x_{N1} & x_{N2} & \cdots & x_{NK} \end{pmatrix}$$

We assume throughout that $N > K$ and X is full column rank.

If you work through the algebra, you will be able to verify that $\|y - Xb\|^2 = \sum_{n=1}^{N} (y_n - b' x_n)^2$.

Since monotone transforms don't affect minimizers, we have

$$\operatorname*{arg\,min}_{b \in \mathbb{R}^K} \sum_{n=1}^{N} (y_n - b' x_n)^2 = \operatorname*{arg\,min}_{b \in \mathbb{R}^K} \|y - Xb\|$$

By our results about overdetermined linear systems of equations, the solution is

$$\hat{\beta} := (X'X)^{-1}X'y$$

Let P and M be the projection and annihilator associated with X:

$$P := X(X'X)^{-1}X' \quad \text{and} \quad M := I - P$$

The **vector of fitted values** is

$$\hat{y} := X\hat{\beta} = Py$$

The **vector of residuals** is

$$\hat{u} := y - \hat{y} = y - Py = My$$

Here are some more standard definitions:

- The **total sum of squares** is $:= \|y\|^2$.
- The **sum of squared residuals** is $:= \|\hat{u}\|^2$.
- The **explained sum of squares** is $:= \|\hat{y}\|^2$.

  TSS = ESS + SSR

We can prove this easily using the *Theorem 1.3.1*.

From the *Theorem 1.3.1* we have $y = \hat{y} + \hat{u}$ and $\hat{u} \perp \hat{y}$.

Applying the Pythagorean law completes the proof.

1.7 Orthogonalization and Decomposition

Let's return to the connection between linear independence and orthogonality touched on above.

A result of much interest is a famous algorithm for constructing orthonormal sets from linearly independent sets.

The next section gives details.

1.7.1 Gram-Schmidt Orthogonalization

Theorem 1.7.1

For each linearly independent set $\{x_1, \ldots, x_k\} \subset \mathbb{R}^n$, there exists an orthonormal set $\{u_1, \ldots, u_k\}$ with

$$\text{span}\{x_1, \ldots, x_i\} = \text{span}\{u_1, \ldots, u_i\} \quad \text{for} \quad i = 1, \ldots, k$$

The **Gram-Schmidt orthogonalization** procedure constructs an orthogonal set $\{u_1, u_2, \ldots, u_n\}$.

One description of this procedure is as follows:

- For $i = 1, \ldots, k$, form $S_i := \text{span}\{x_1, \ldots, x_i\}$ and $S_i^\perp$
- Set $v_1 = x_1$
- For $i \geq 2$ set $v_i := \hat{E}_{S_{i-1}^\perp} x_i$ and $u_i := v_i / \|v_i\|$

The sequence $u_1, \ldots, u_k$ has the stated properties.

A Gram-Schmidt orthogonalization construction is a key idea behind the Kalman filter described in A First Look at the Kalman filter.

In some exercises below, you are asked to implement this algorithm and test it using projection.

1.7.2 QR Decomposition

The following result uses the preceding algorithm to produce a useful decomposition.

Theorem 1.7.2

If X is $n \times k$ with linearly independent columns, then there exists a factorization $X = QR$ where

- R is $k \times k$, upper triangular, and nonsingular
- Q is $n \times k$ with orthonormal columns

Proof. Let

- $x_j := \text{col}_j(X)$
- $\{u_1, \ldots, u_k\}$ be orthonormal with the same span as $\{x_1, \ldots, x_k\}$ (to be constructed using Gram–Schmidt)
- Q be formed from columns u_i

Since $x_j \in \text{span}\{u_1, \ldots, u_j\}$, we have

$$x_j = \sum_{i=1}^{j} \langle u_i, x_j \rangle u_i \quad \text{for } j = 1, \ldots, k$$

Some rearranging gives $X = QR$.

1.7.3 Linear Regression via QR Decomposition

For matrices X and y that overdetermine β in the linear equation system $y = X\beta$, we found the least squares approximator $\hat{\beta} = (X'X)^{-1}X'y$.

Using the QR decomposition $X = QR$ gives

$$\begin{aligned} \hat{\beta} &= (R'Q'QR)^{-1}R'Q'y \\ &= (R'R)^{-1}R'Q'y \\ &= R^{-1}Q'y \end{aligned}$$

where the last step uses the fact that $(R'R)^{-1}R' = R^{-1}$ since R is nonsingular.

Numerical routines would in this case use the alternative form $R\hat{\beta} = Q'y$ and back substitution.

1.8 Exercises

Exercise 1.8.1

Show that, for any linear subspace $S \subset \mathbb{R}^n$, $S \cap S^\perp = \{0\}$.

Solution to Exercise 1.8.1

If $x \in S$ and $x \in S^\perp$, then we have in particular that $\langle x, x \rangle = 0$, but then $x = 0$.

Exercise 1.8.2

Let $P = X(X'X)^{-1}X'$ and let $M = I - P$. Show that P and M are both idempotent and symmetric. Can you give any intuition as to why they should be idempotent?

Solution to Exercise 1.8.2

Symmetry and idempotence of M and P can be established using standard rules for matrix algebra. The intuition behind idempotence of M and P is that both are orthogonal projections. After a point is projected into a given subspace, applying the projection again makes no difference (A point inside the subspace is not shifted by orthogonal projection onto that space because it is already the closest point in the subspace to itself).

Exercise 1.8.3

Using Gram-Schmidt orthogonalization, produce a linear projection of y onto the column space of X and verify this using the projection matrix $P := X(X'X)^{-1}X'$ and also using QR decomposition, where:

$$y := \begin{pmatrix} 1 \\ 3 \\ -3 \end{pmatrix},$$

and

$$X := \begin{pmatrix} 1 & 0 \\ 0 & -6 \\ 2 & 2 \end{pmatrix}$$

Solution to Exercise 1.8.3

Here's a function that computes the orthonormal vectors using the GS algorithm given in the lecture

```
def gram_schmidt(X):
    """
    Implements Gram-Schmidt orthogonalization.

    Parameters
    ----------
    X : an n x k array with linearly independent columns
```

(continues on next page)

(continued from previous page)

```
    Returns
    -------
    U : an n x k array with orthonormal columns

    """

    # Set up
    n, k = X.shape
    U = np.empty((n, k))
    I = np.eye(n)

    # The first column of U is just the normalized first column of X
    v1 = X[:, 0]
    U[:, 0] = v1 / np.sqrt(np.sum(v1 * v1))

    for i in range(1, k):
        # Set up
        b = X[:, i]       # The vector we're going to project
        Z = X[:, :i]      # First i-1 columns of X

        # Project onto the orthogonal complement of the columns span of Z
        M = I - Z @ np.linalg.inv(Z.T @ Z) @ Z.T
        u = M @ b

        # Normalize
        U[:, i] = u / np.sqrt(np.sum(u * u))

    return U
```

Here are the arrays we'll work with

```
y = [1, 3, -3]

X = [[1,  0],
     [0, -6],
     [2,  2]]

X, y = [np.asarray(z) for z in (X, y)]
```

First, let's try projection of y onto the column space of X using the ordinary matrix expression:

```
Py1 = X @ np.linalg.inv(X.T @ X) @ X.T @ y
Py1
```

```
array([-0.56521739,  3.26086957, -2.2173913 ])
```

Now let's do the same using an orthonormal basis created from our `gram_schmidt` function

```
U = gram_schmidt(X)
U
```

```
array([[ 0.4472136 , -0.13187609],
       [ 0.        , -0.98907071],
```

(continues on next page)

(continued from previous page)

```
       [ 0.89442719,  0.06593805]])
```

```
Py2 = U @ U.T @ y
Py2
```

```
array([-0.56521739,  3.26086957, -2.2173913 ])
```

This is the same answer. So far so good. Finally, let's try the same thing but with the basis obtained via QR decomposition:

```
Q, R = qr(X, mode='economic')
Q
```

```
array([[-0.4472136 , -0.13187609],
       [-0.        , -0.98907071],
       [-0.89442719,  0.06593805]])
```

```
Py3 = Q @ Q.T @ y
Py3
```

```
array([-0.56521739,  3.26086957, -2.2173913 ])
```

Again, we obtain the same answer.

CHAPTER

TWO

CONTINUOUS STATE MARKOV CHAINS

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install --upgrade quantecon
```

2.1 Overview

In a previous lecture, we learned about finite Markov chains, a relatively elementary class of stochastic dynamic models.

The present lecture extends this analysis to continuous (i.e., uncountable) state Markov chains.

Most stochastic dynamic models studied by economists either fit directly into this class or can be represented as continuous state Markov chains after minor modifications.

In this lecture, our focus will be on continuous Markov models that

- evolve in discrete-time
- are often nonlinear

The fact that we accommodate nonlinear models here is significant, because linear stochastic models have their own highly developed toolset, as we'll see *later on*.

The question that interests us most is: Given a particular stochastic dynamic model, how will the state of the system evolve over time?

In particular,

- What happens to the distribution of the state variables?
- Is there anything we can say about the "average behavior" of these variables?
- Is there a notion of "steady state" or "long-run equilibrium" that's applicable to the model?
  - If so, how can we compute it?

Answering these questions will lead us to revisit many of the topics that occupied us in the finite state case, such as simulation, distribution dynamics, stability, ergodicity, etc.

Note: For some people, the term "Markov chain" always refers to a process with a finite or discrete state space. We follow the mainstream mathematical literature (e.g., [Meyn and Tweedie, 2009]) in using the term to refer to any discrete **time** Markov process.

Let's begin with some imports:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import lognorm, beta
from quantecon import LAE
from scipy.stats import norm, gaussian_kde
```

2.2 The Density Case

You are probably aware that some distributions can be represented by densities and some cannot.

(For example, distributions on the real numbers $\mathbb{R}$ that put positive probability on individual points have no density representation)

We are going to start our analysis by looking at Markov chains where the one-step transition probabilities have density representations.

The benefit is that the density case offers a very direct parallel to the finite case in terms of notation and intuition.

Once we've built some intuition we'll cover the general case.

2.2.1 Definitions and Basic Properties

In our lecture on finite Markov chains, we studied discrete-time Markov chains that evolve on a finite state space S.

In this setting, the dynamics of the model are described by a stochastic matrix — a nonnegative square matrix $P = P[i, j]$ such that each row $P[i, \cdot]$ sums to one.

The interpretation of P is that $P[i, j]$ represents the probability of transitioning from state i to state j in one unit of time.

In symbols,

$$\mathbb{P}\{X_{t+1} = j \,|\, X_t = i\} = P[i, j]$$

Equivalently,

- P can be thought of as a family of distributions $P[i, \cdot]$, one for each $i \in S$
- $P[i, \cdot]$ is the distribution of X_{t+1} given $X_t = i$

(As you probably recall, when using NumPy arrays, $P[i, \cdot]$ is expressed as `P[i, :]`)

In this section, we'll allow S to be a subset of $\mathbb{R}$, such as

- $\mathbb{R}$ itself
- the positive reals $(0, \infty)$
- a bounded interval (a, b)

The family of discrete distributions $P[i, \cdot]$ will be replaced by a family of densities $p(x, \cdot)$, one for each $x \in S$.

Analogous to the finite state case, $p(x, \cdot)$ is to be understood as the distribution (density) of X_{t+1} given $X_t = x$.

More formally, a *stochastic kernel on* S is a function $p \colon S \times S \to \mathbb{R}$ with the property that

1. $p(x, y) \geq 0$ for all $x, y \in S$
2. $\int p(x, y) dy = 1$ for all $x \in S$

(Integrals are over the whole space unless otherwise specified)

For example, let $S = \mathbb{R}$ and consider the particular stochastic kernel p_w defined by

$$p_w(x, y) := \frac{1}{\sqrt{2\pi}} \exp\left\{-\frac{(y-x)^2}{2}\right\} \tag{2.1}$$

What kind of model does p_w represent?

The answer is, the (normally distributed) random walk

$$X_{t+1} = X_t + \xi_{t+1} \quad \text{where} \quad \{\xi_t\} \overset{\text{IID}}{\sim} N(0,1) \tag{2.2}$$

To see this, let's find the stochastic kernel p corresponding to (2.2).

Recall that $p(x, \cdot)$ represents the distribution of X_{t+1} given $X_t = x$.

Letting $X_t = x$ in (2.2) and considering the distribution of X_{t+1}, we see that $p(x, \cdot) = N(x, 1)$.

In other words, p is exactly p_w, as defined in (2.1).

2.2.2 Connection to Stochastic Difference Equations

In the previous section, we made the connection between stochastic difference equation (2.2) and stochastic kernel (2.1).

In economics and time-series analysis we meet stochastic difference equations of all different shapes and sizes.

It will be useful for us if we have some systematic methods for converting stochastic difference equations into stochastic kernels.

To this end, consider the generic (scalar) stochastic difference equation given by

$$X_{t+1} = \mu(X_t) + \sigma(X_t)\,\xi_{t+1} \tag{2.3}$$

Here we assume that

- $\{\xi_t\} \overset{\text{IID}}{\sim} \phi$, where ϕ is a given density on $\mathbb{R}$
- μ and σ are given functions on S, with $\sigma(x) > 0$ for all x

Example 1: The random walk (2.2) is a special case of (2.3), with $\mu(x) = x$ and $\sigma(x) = 1$.

Example 2: Consider the ARCH model

$$X_{t+1} = \alpha X_t + \sigma_t\,\xi_{t+1}, \qquad \sigma_t^2 = \beta + \gamma X_t^2, \qquad \beta, \gamma > 0$$

Alternatively, we can write the model as

$$X_{t+1} = \alpha X_t + (\beta + \gamma X_t^2)^{1/2} \xi_{t+1} \tag{2.4}$$

This is a special case of (2.3) with $\mu(x) = \alpha x$ and $\sigma(x) = (\beta + \gamma x^2)^{1/2}$.

Example 3: With stochastic production and a constant savings rate, the one-sector neoclassical growth model leads to a law of motion for capital per worker such as

$$k_{t+1} = sA_{t+1} f(k_t) + (1-\delta)k_t \tag{2.5}$$

Here

- s is the rate of savings
- A_{t+1} is a production shock

  - The $t+1$ subscript indicates that A_{t+1} is not visible at time t
- δ is a depreciation rate
- $f\colon \mathbb{R}_+ \to \mathbb{R}_+$ is a production function satisfying $f(k) > 0$ whenever $k > 0$

(The fixed savings rate can be rationalized as the optimal policy for a particular set of technologies and preferences (see [Ljungqvist and Sargent, 2018], section 3.1.2), although we omit the details here).

Equation (2.5) is a special case of (2.3) with $\mu(x) = (1-\delta)x$ and $\sigma(x) = sf(x)$.

Now let's obtain the stochastic kernel corresponding to the generic model (2.3).

To find it, note first that if U is a random variable with density f_U, and $V = a + bU$ for some constants a, b with $b > 0$, then the density of V is given by

$$f_V(v) = \frac{1}{b} f_U\left(\frac{v-a}{b}\right) \tag{2.6}$$

(The proof is *below*. For a multidimensional version see EDTC, theorem 8.1.3).

Taking (2.6) as given for the moment, we can obtain the stochastic kernel p for (2.3) by recalling that $p(x, \cdot)$ is the conditional density of X_{t+1} given $X_t = x$.

In the present case, this is equivalent to stating that $p(x, \cdot)$ is the density of $Y := \mu(x) + \sigma(x)\,\xi_{t+1}$ when $\xi_{t+1} \sim \phi$.

Hence, by (2.6),

$$p(x, y) = \frac{1}{\sigma(x)} \phi\left(\frac{y - \mu(x)}{\sigma(x)}\right) \tag{2.7}$$

For example, the growth model in (2.5) has stochastic kernel

$$p(x, y) = \frac{1}{sf(x)} \phi\left(\frac{y - (1-\delta)x}{sf(x)}\right) \tag{2.8}$$

where ϕ is the density of A_{t+1}.

(Regarding the state space S for this model, a natural choice is $(0, \infty)$ — in which case $\sigma(x) = sf(x)$ is strictly positive for all s as required)

2.2.3 Distribution Dynamics

In this section of our lecture on **finite** Markov chains, we asked the following question: If

1. $\{X_t\}$ is a Markov chain with stochastic matrix P
2. the distribution of X_t is known to be ψ_t

then what is the distribution of X_{t+1}?

Letting ψ_{t+1} denote the distribution of X_{t+1}, the answer we gave was that

$$\psi_{t+1}[j] = \sum_{i \in S} P[i, j] \psi_t[i]$$

This intuitive equality states that the probability of being at j tomorrow is the probability of visiting i today and then going on to j, summed over all possible i.

In the density case, we just replace the sum with an integral and probability mass functions with densities, yielding

$$\psi_{t+1}(y) = \int p(x, y) \psi_t(x)\, dx, \qquad \forall y \in S \tag{2.9}$$

It is convenient to think of this updating process in terms of an operator.

(An operator is just a function, but the term is usually reserved for a function that sends functions into functions)

Let $\mathcal{D}$ be the set of all densities on S, and let P be the operator from $\mathcal{D}$ to itself that takes density ψ and sends it into new density ψP, where the latter is defined by

$$(\psi P)(y) = \int p(x, y)\psi(x)dx \tag{2.10}$$

This operator is usually called the *Markov operator* corresponding to p

---

Note: Unlike most operators, we write P to the right of its argument, instead of to the left (i.e., ψP instead of $P\psi$). This is a common convention, with the intention being to maintain the parallel with the finite case — see here

---

With this notation, we can write (2.9) more succinctly as $\psi_{t+1}(y) = (\psi_t P)(y)$ for all y, or, dropping the y and letting "=" indicate equality of functions,

$$\psi_{t+1} = \psi_t P \tag{2.11}$$

Equation (2.11) tells us that if we specify a distribution for ψ_0, then the entire sequence of future distributions can be obtained by iterating with P.

It's interesting to note that (2.11) is a deterministic difference equation.

Thus, by converting a stochastic difference equation such as (2.3) into a stochastic kernel p and hence an operator P, we convert a stochastic difference equation into a deterministic one (albeit in a much higher dimensional space).

Note: Some people might be aware that discrete Markov chains are in fact a special case of the continuous Markov chains we have just described. The reason is that probability mass functions are densities with respect to the counting measure.

2.2.4 Computation

To learn about the dynamics of a given process, it's useful to compute and study the sequences of densities generated by the model.

One way to do this is to try to implement the iteration described by (2.10) and (2.11) using numerical integration.

However, to produce ψP from ψ via (2.10), you would need to integrate at every y, and there is a continuum of such y.

Another possibility is to discretize the model, but this introduces errors of unknown size.

A nicer alternative in the present setting is to combine simulation with an elegant estimator called the *look-ahead* estimator.

Let's go over the ideas with reference to the growth model *discussed above*, the dynamics of which we repeat here for convenience:

$$k_{t+1} = sA_{t+1}f(k_t) + (1 - \delta)k_t \tag{2.12}$$

Our aim is to compute the sequence $\{\psi_t\}$ associated with this model and fixed initial condition ψ_0.

To approximate ψ_t by simulation, recall that, by definition, ψ_t is the density of k_t given $k_0 \sim \psi_0$.

If we wish to generate observations of this random variable, all we need to do is

1. draw k_0 from the specified initial condition ψ_0

2. draw the shocks $A_1, \ldots, A_t$ from their specified density ϕ
3. compute k_t iteratively via (2.12)

If we repeat this n times, we get n independent observations $k_t^1, \ldots, k_t^n$.

With these draws in hand, the next step is to generate some kind of representation of their distribution ψ_t.

A naive approach would be to use a histogram, or perhaps a smoothed histogram using SciPy's `gaussian_kde` function.

However, in the present setting, there is a much better way to do this, based on the look-ahead estimator.

With this estimator, to construct an estimate of ψ_t, we actually generate n observations of k_{t-1}, rather than k_t.

Now we take these n observations $k_{t-1}^1, \ldots, k_{t-1}^n$ and form the estimate

$$\psi_t^n(y) = \frac{1}{n} \sum_{i=1}^n p(k_{t-1}^i, y) \tag{2.13}$$

where p is the growth model stochastic kernel in (2.8).

What is the justification for this slightly surprising estimator?

The idea is that, by the strong law of large numbers,

$$\frac{1}{n} \sum_{i=1}^n p(k_{t-1}^i, y) \to \mathbb{E} p(k_{t-1}^i, y) = \int p(x, y) \psi_{t-1}(x) \, dx = \psi_t(y)$$

with probability one as $n \to \infty$.

Here the first equality is by the definition of ψ_{t-1}, and the second is by (2.9).

We have just shown that our estimator $\psi_t^n(y)$ in (2.13) converges almost surely to $\psi_t(y)$, which is just what we want to compute.

In fact, much stronger convergence results are true (see, for example, this paper).

2.2.5 Implementation

A class called `LAE` for estimating densities by this technique can be found in lae.py.

Given our use of the `__call__` method, an instance of `LAE` acts as a callable object, which is essentially a function that can store its own data (see this discussion).

This function returns the right-hand side of (2.13) using

- the data and stochastic kernel that it stores as its instance data
- the value y as its argument

The function is vectorized, in the sense that if `psi` is such an instance and `y` is an array, then the call `psi(y)` acts elementwise.

(This is the reason that we reshaped `X` and `y` inside the class — to make vectorization work)

Because the implementation is fully vectorized, it is about as efficient as it would be in C or Fortran.

2.2.6 Example

The following code is an example of usage for the stochastic growth model *described above*

```
# == Define parameters == #
s = 0.2
δ = 0.1
a_σ = 0.4                    # A = exp(B) where B ~ N(0, a_σ)
α = 0.4                      # We set f(k) = k**α
ψ_0 = beta(5, 5, scale=0.5)  # Initial distribution
ϕ = lognorm(a_σ)

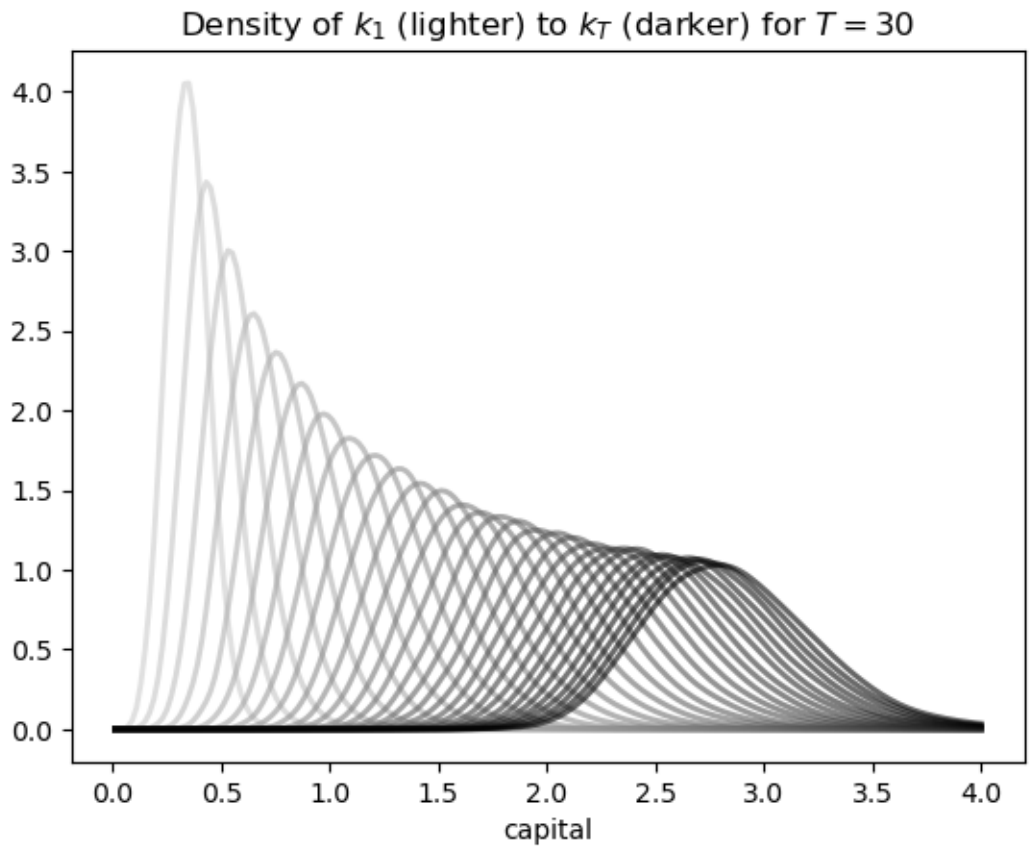

def p(x, y):
    """
    Stochastic kernel for the growth model with Cobb-Douglas production.
    Both x and y must be strictly positive.
    """
    d = s * x**α
    return ϕ.pdf((y - (1 - δ) * x) / d) / d

n = 10000    # Number of observations at each date t
T = 30       # Compute density of k_t at 1,...,T+1

# == Generate matrix s.t. t-th column is n observations of k_t == #
k = np.empty((n, T))
A = ϕ.rvs((n, T))
k[:, 0] = ψ_0.rvs(n)  # Draw first column from initial distribution
for t in range(T-1):
    k[:, t+1] = s * A[:, t] * k[:, t]**α + (1 - δ) * k[:, t]

# == Generate T instances of LAE using this data, one for each date t == #
laes = [LAE(p, k[:, t]) for t in range(T)]

# == Plot == #
fig, ax = plt.subplots()
ygrid = np.linspace(0.01, 4.0, 200)
greys = [str(g) for g in np.linspace(0.0, 0.8, T)]
greys.reverse()
for ψ, g in zip(laes, greys):
    ax.plot(ygrid, ψ(ygrid), color=g, lw=2, alpha=0.6)
ax.set_xlabel('capital')
ax.set_title(f'Density of $k_1$ (lighter) to $k_T$ (darker) for $T={T}$')
plt.show()
```



The figure shows part of the density sequence $\{\psi_t\}$, with each density computed via the look-ahead estimator.

Notice that the sequence of densities shown in the figure seems to be converging — more on this in just a moment.

Another quick comment is that each of these distributions could be interpreted as a cross-sectional distribution (recall this discussion).

2.3 Beyond Densities

Up until now, we have focused exclusively on continuous state Markov chains where all conditional distributions $p(x, \cdot)$ are densities.

As discussed above, not all distributions can be represented as densities.

If the conditional distribution of X_{t+1} given $X_t = x$ **cannot** be represented as a density for some $x \in S$, then we need a slightly different theory.

The ultimate option is to switch from densities to probability measures, but not all readers will be familiar with measure theory.

We can, however, construct a fairly general theory using distribution functions.

2.3.1 Example and Definitions

To illustrate the issues, recall that Hopenhayn and Rogerson [Hopenhayn and Rogerson, 1993] study a model of firm dynamics where individual firm productivity follows the exogenous process

$$X_{t+1} = a + \rho X_t + \xi_{t+1}, \quad \text{where} \quad \{\xi_t\} \overset{\text{IID}}{\sim} N(0, \sigma^2)$$

As is, this fits into the density case we treated above.

However, the authors wanted this process to take values in $[0, 1]$, so they added boundaries at the endpoints 0 and 1.

One way to write this is

$$X_{t+1} = h(a + \rho X_t + \xi_{t+1}) \quad \text{where} \quad h(x) := x \mathbf{1}\{0 \leq x \leq 1\} + \mathbf{1}\{x > 1\}$$

If you think about it, you will see that for any given $x \in [0, 1]$, the conditional distribution of X_{t+1} given $X_t = x$ puts positive probability mass on 0 and 1.

Hence it cannot be represented as a density.

What we can do instead is use cumulative distribution functions (cdfs).

To this end, set

$$G(x, y) := \mathbb{P}\{h(a + \rho x + \xi_{t+1}) \leq y\} \qquad (0 \leq x, y \leq 1)$$

This family of cdfs $G(x, \cdot)$ plays a role analogous to the stochastic kernel in the density case.

The distribution dynamics in (2.9) are then replaced by

$$F_{t+1}(y) = \int G(x, y) F_t(dx) \tag{2.14}$$

Here F_t and F_{t+1} are cdfs representing the distribution of the current state and next period state.

The intuition behind (2.14) is essentially the same as for (2.9).

2.3.2 Computation

If you wish to compute these cdfs, you cannot use the look-ahead estimator as before.

Indeed, you should not use any density estimator, since the objects you are estimating/computing are not densities.

One good option is simulation as before, combined with the empirical distribution function.

2.4 Stability

In our lecture on finite Markov chains, we also studied stationarity, stability and ergodicity.

Here we will cover the same topics for the continuous case.

We will, however, treat only the density case (as in *this section*), where the stochastic kernel is a family of densities.

The general case is relatively similar — references are given below.

2.4.1 Theoretical Results

Analogous to the finite case, given a stochastic kernel p and corresponding Markov operator as defined in (2.10), a density ψ^* on S is called *stationary* for P if it is a fixed point of the operator P.

In other words,

$$\psi^*(y) = \int p(x, y)\psi^*(x)\, dx, \qquad \forall y \in S \tag{2.15}$$

As with the finite case, if ψ^* is stationary for P, and the distribution of X_0 is ψ^*, then, in view of (2.11), X_t will have this same distribution for all t.

Hence ψ^* is the stochastic equivalent of a steady state.

In the finite case, we learned that at least one stationary distribution exists, although there may be many.

When the state space is infinite, the situation is more complicated.

Even existence can fail very easily.

For example, the random walk model has no stationary density (see, e.g., EDTC, p. 210).

However, there are well-known conditions under which a stationary density ψ^* exists.

With additional conditions, we can also get a unique stationary density ($\psi \in \mathcal{D}$ and $\psi = \psi P \implies \psi = \psi^*$), and also global convergence in the sense that

$$\forall\, \psi \in \mathcal{D}, \quad \psi P^t \to \psi^* \quad \text{as} \quad t \to \infty \tag{2.16}$$

This combination of existence, uniqueness and global convergence in the sense of (2.16) is often referred to as *global stability*.

Under very similar conditions, we get *ergodicity*, which means that

$$\frac{1}{n}\sum_{t=1}^{n} h(X_t) \to \int h(x)\psi^*(x)dx \quad \text{as } n \to \infty \tag{2.17}$$

for any (measurable) function $h : S \to \mathbb{R}$ such that the right-hand side is finite.

Note that the convergence in (2.17) does not depend on the distribution (or value) of X_0.

This is actually very important for simulation — it means we can learn about ψ^* (i.e., approximate the right-hand side of (2.17) via the left-hand side) without requiring any special knowledge about what to do with X_0.

So what are these conditions we require to get global stability and ergodicity?

In essence, it must be the case that

1. Probability mass does not drift off to the "edges" of the state space.
2. Sufficient "mixing" obtains.

For one such set of conditions see theorem 8.2.14 of EDTC.

In addition

- [Stokey *et al.*, 1989] contains a classic (but slightly outdated) treatment of these topics.
- From the mathematical literature, [Lasota and MacKey, 1994] and [Meyn and Tweedie, 2009] give outstanding in-depth treatments.
- Section 8.1.2 of EDTC provides detailed intuition, and section 8.3 gives additional references.
- EDTC, section 11.3.4 provides a specific treatment for the growth model we considered in this lecture.

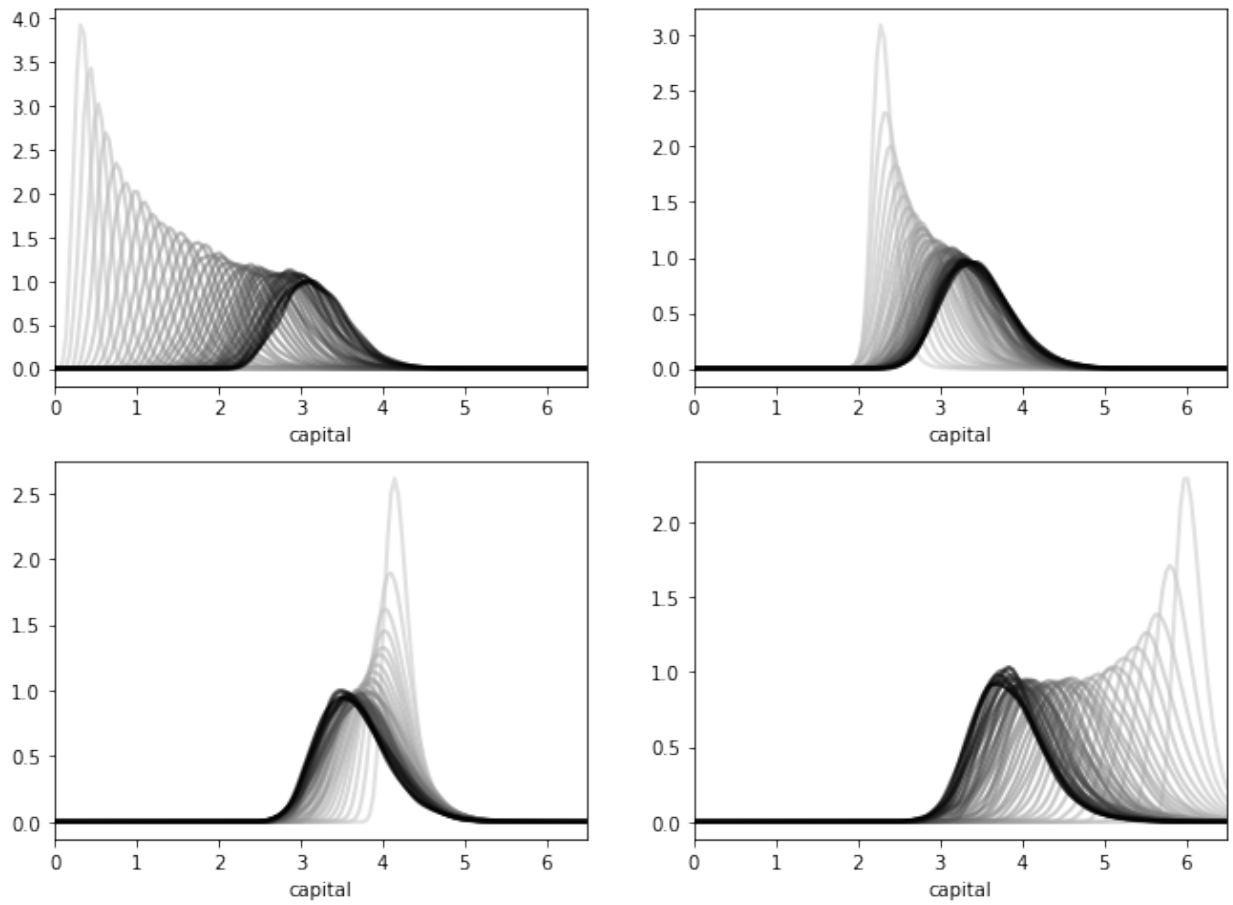
2.4.2 An Example of Stability

As stated above, the *growth model treated here* is stable under mild conditions on the primitives.

- See EDTC, section 11.3.4 for more details.

We can see this stability in action — in particular, the convergence in (2.16) — by simulating the path of densities from various initial conditions.

Here is such a figure.



All sequences are converging towards the same limit, regardless of their initial condition.

The details regarding initial conditions and so on are given in *this exercise*, where you are asked to replicate the figure.

2.4.3 Computing Stationary Densities

In the preceding figure, each sequence of densities is converging towards the unique stationary density ψ^*.

Even from this figure, we can get a fair idea what ψ^* looks like, and where its mass is located.

However, there is a much more direct way to estimate the stationary density, and it involves only a slight modification of the look-ahead estimator.

Let's say that we have a model of the form (2.3) that is stable and ergodic.

Let p be the corresponding stochastic kernel, as given in (2.7).

To approximate the stationary density ψ^*, we can simply generate a long time-series $X_0, X_1, \ldots, X_n$ and estimate ψ^* via

$$\psi_n^*(y) = \frac{1}{n} \sum_{t=1}^{n} p(X_t, y) \tag{2.18}$$

This is essentially the same as the look-ahead estimator (2.13), except that now the observations we generate are a single time-series, rather than a cross-section.

The justification for (2.18) is that, with probability one as $n \to \infty$,

$$\frac{1}{n} \sum_{t=1}^{n} p(X_t, y) \to \int p(x, y) \psi^*(x) \, dx = \psi^*(y)$$

where the convergence is by (2.17) and the equality on the right is by (2.15).

The right-hand side is exactly what we want to compute.

On top of this asymptotic result, it turns out that the rate of convergence for the look-ahead estimator is very good.

The first exercise helps illustrate this point.

2.5 Exercises

Exercise 2.5.1

Consider the simple threshold autoregressive model

$$X_{t+1} = \theta |X_t| + (1 - \theta^2)^{1/2} \xi_{t+1} \qquad \text{where} \quad \{\xi_t\} \overset{\text{IID}}{\sim} N(0, 1) \tag{2.19}$$

This is one of those rare nonlinear stochastic models where an analytical expression for the stationary density is available.

In particular, provided that $|\theta| < 1$, there is a unique stationary density ψ^* given by

$$\psi^*(y) = 2 \, \phi(y) \, \Phi \left[\frac{\theta y}{(1 - \theta^2)^{1/2}} \right] \tag{2.20}$$

Here ϕ is the standard normal density and Φ is the standard normal cdf.

As an exercise, compute the look-ahead estimate of ψ^*, as defined in (2.18), and compare it with ψ^* in (2.20) to see whether they are indeed close for large n.

In doing so, set $\theta = 0.8$ and $n = 500$.

The next figure shows the result of such a computation

The additional density (black line) is a nonparametric kernel density estimate, added to the solution for illustration.

(You can try to replicate it before looking at the solution if you want to)

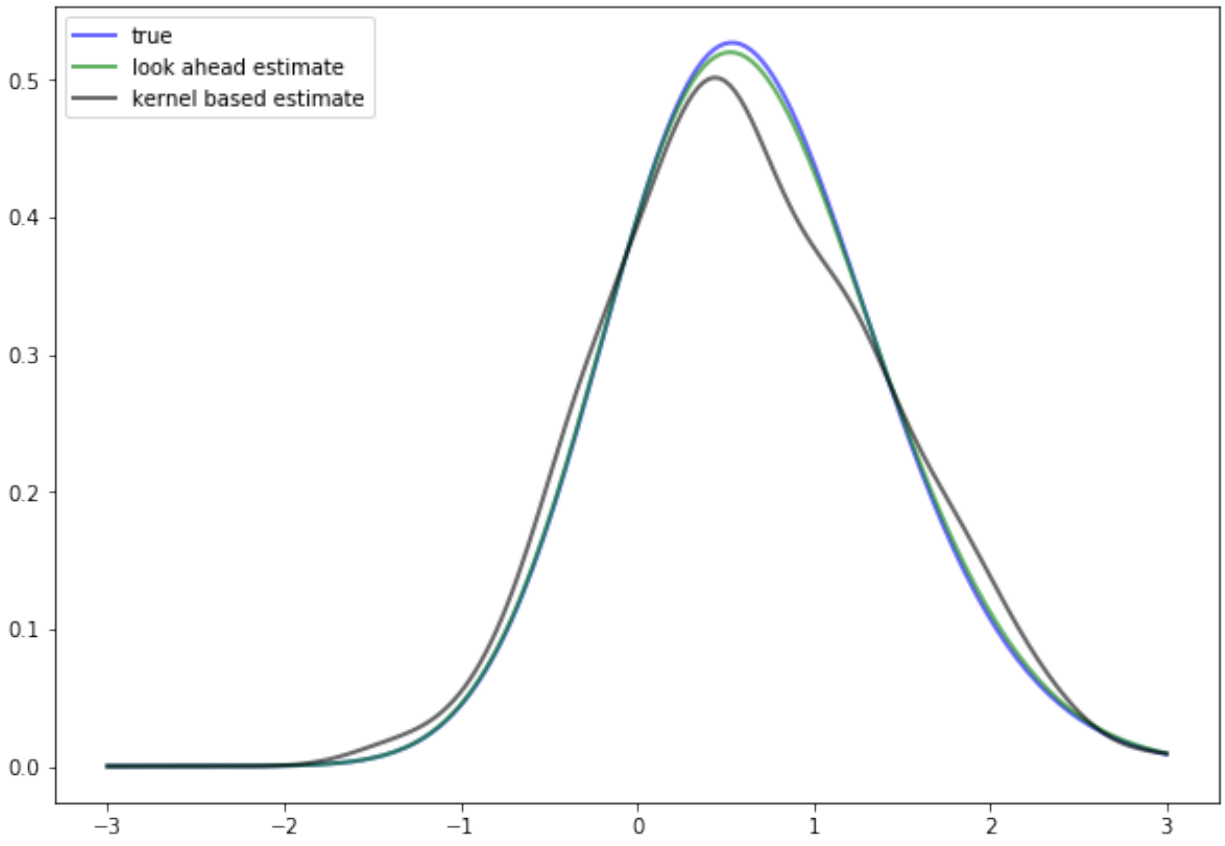
As you can see, the look-ahead estimator is a much tighter fit than the kernel density estimator.

If you repeat the simulation you will see that this is consistently the case.

Solution to Exercise 2.5.1

Look-ahead estimation of a TAR stationary density, where the TAR model is

$$X_{t+1} = \theta |X_t| + (1 - \theta^2)^{1/2} \xi_{t+1}$$


true
look ahead estimate
kernel based estimate
0.5
0.4
0.3
0.2
0.1
0.0
−3
−2
−1
0
1
2
3

and $\xi_t \sim N(0,1)$.

Try running at `n = 10, 100, 1000, 10000` to get an idea of the speed of convergence

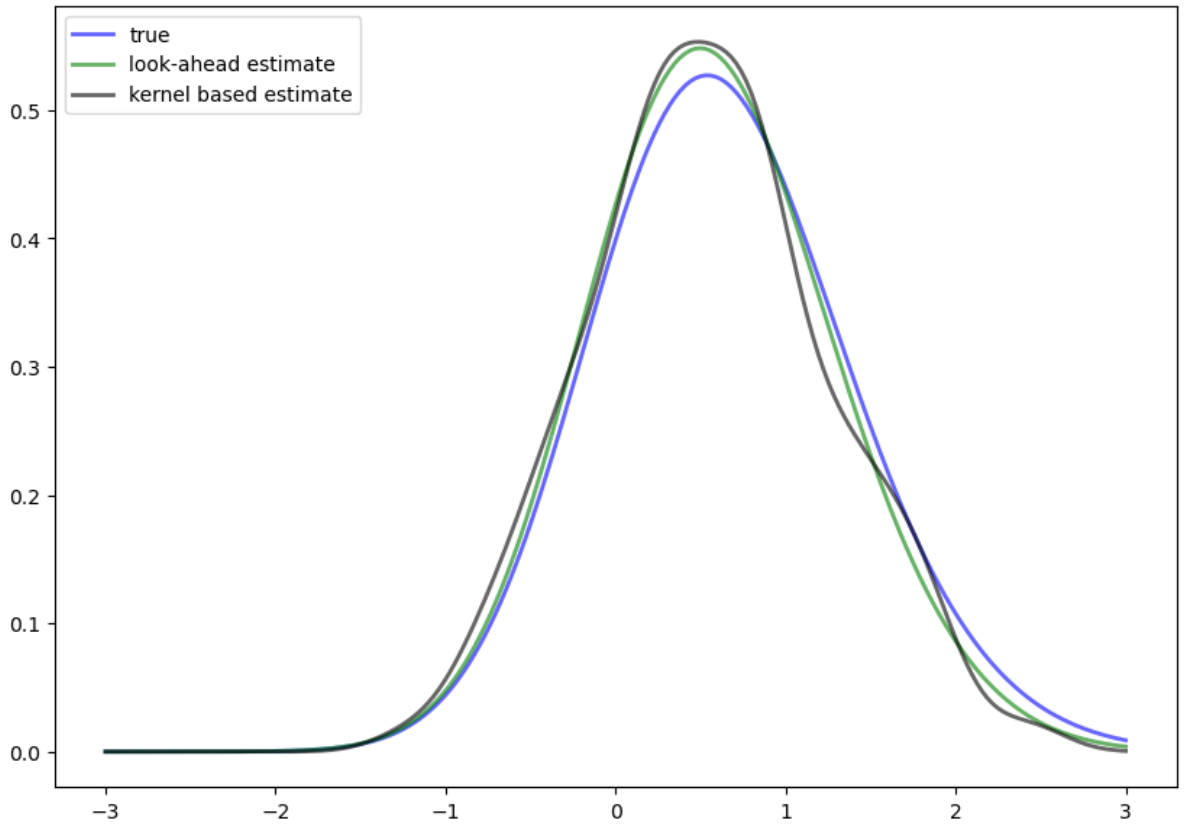
```
ϕ = norm()
n = 500
θ = 0.8
# == Frequently used constants == #
d = np.sqrt(1 - θ**2)
δ = θ / d

def ψ_star(y):
    "True stationary density of the TAR Model"
    return 2 * norm.pdf(y) * norm.cdf(δ * y)

def p(x, y):
    "Stochastic kernel for the TAR model."
    return ϕ.pdf((y - θ * np.abs(x)) / d) / d

Z = ϕ.rvs(n)
X = np.empty(n)
for t in range(n-1):
    X[t+1] = θ * np.abs(X[t]) + d * Z[t]
ψ_est = LAE(p, X)
k_est = gaussian_kde(X)

fig, ax = plt.subplots(figsize=(10, 7))
ys = np.linspace(-3, 3, 200)
ax.plot(ys, ψ_star(ys), 'b-', lw=2, alpha=0.6, label='true')
ax.plot(ys, ψ_est(ys), 'g-', lw=2, alpha=0.6, label='look-ahead estimate')
ax.plot(ys, k_est(ys), 'k-', lw=2, alpha=0.6, label='kernel based estimate')
ax.legend(loc='upper left')
plt.show()
```



Exercise 2.5.2

Replicate the figure on global convergence *shown above*.

The densities come from the stochastic growth model treated *at the start of the lecture*.

Begin with the code found *above*.

Use the same parameters.

For the four initial distributions, use the shifted beta distributions

```
ψ_0 = beta(5, 5, scale=0.5, loc=i*2)
```

Solution to Exercise 2.5.2

Here's one program that does the job

```
# == Define parameters == #
s = 0.2
δ = 0.1
a_σ = 0.4                                # A = exp(B) where B ~ N(0, a_σ)
α = 0.4                                  # f(k) = k**α

ϕ = lognorm(a_σ)
```

(continues on next page)

(continued from previous page)

```
def p(x, y):
    "Stochastic kernel, vectorized in x.  Both x and y must be positive."
    d = s * x**α
    return ϕ.pdf((y - (1 - δ) * x) / d) / d

n = 1000                             # Number of observations at each date t
T = 40                               # Compute density of k_t at 1,...,T

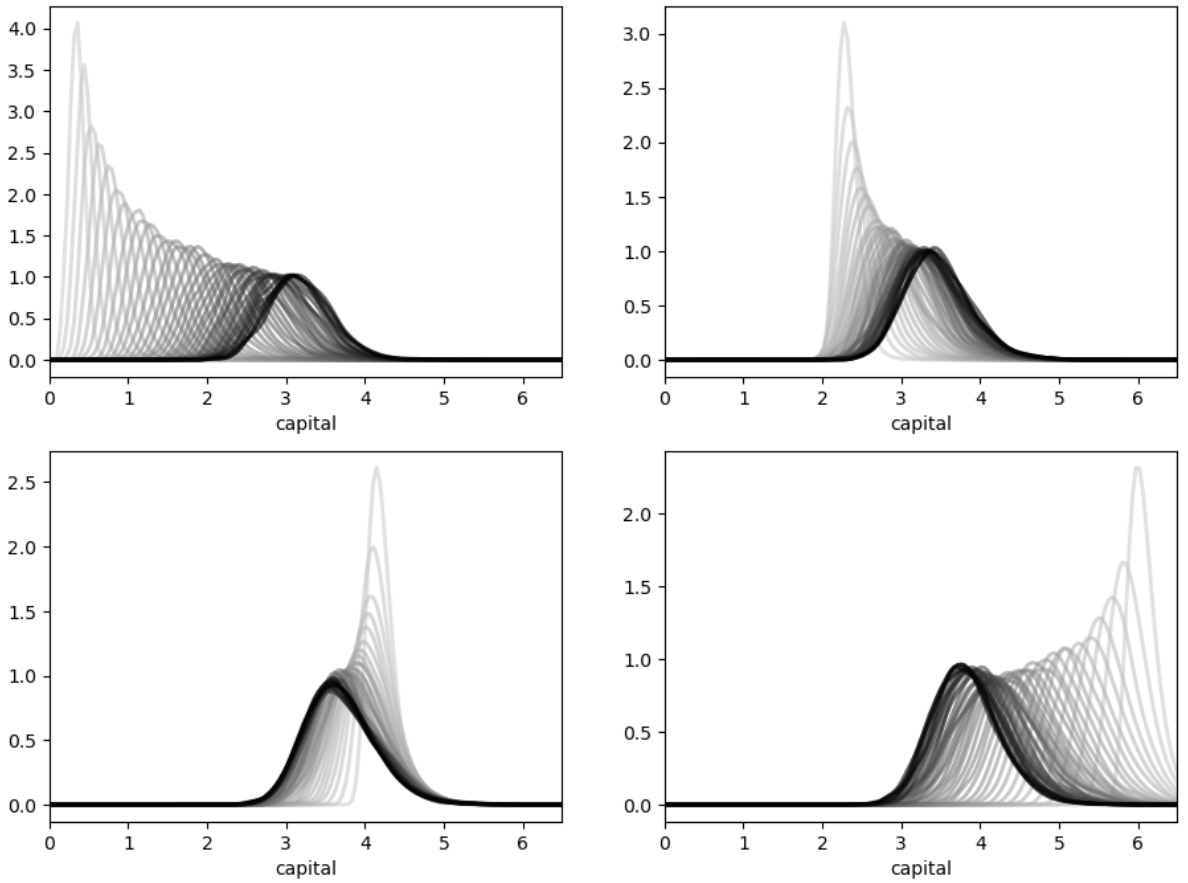
fig, axes = plt.subplots(2, 2, figsize=(11, 8))
axes = axes.flatten()
xmax = 6.5

for i in range(4):
    ax = axes[i]
    ax.set_xlim(0, xmax)
    ψ_0 = beta(5, 5, scale=0.5, loc=i*2)  # Initial distribution

    # == Generate matrix s.t. t-th column is n observations of k_t == #
    k = np.empty((n, T))
    A = ϕ.rvs((n, T))
    k[:, 0] = ψ_0.rvs(n)
    for t in range(T-1):
        k[:, t+1] = s * A[:,t] * k[:, t]**α + (1 - δ) * k[:, t]

    # == Generate T instances of lae using this data, one for each t == #
    laes = [LAE(p, k[:, t]) for t in range(T)]

    ygrid = np.linspace(0.01, xmax, 150)
    greys = [str(g) for g in np.linspace(0.0, 0.8, T)]
    greys.reverse()
    for ψ, g in zip(laes, greys):
        ax.plot(ygrid, ψ(ygrid), color=g, lw=2, alpha=0.6)
    ax.set_xlabel('capital')
plt.show()
```



Exercise 2.5.3

A common way to compare distributions visually is with boxplots.

To illustrate, let's generate three artificial data sets and compare them with a boxplot.

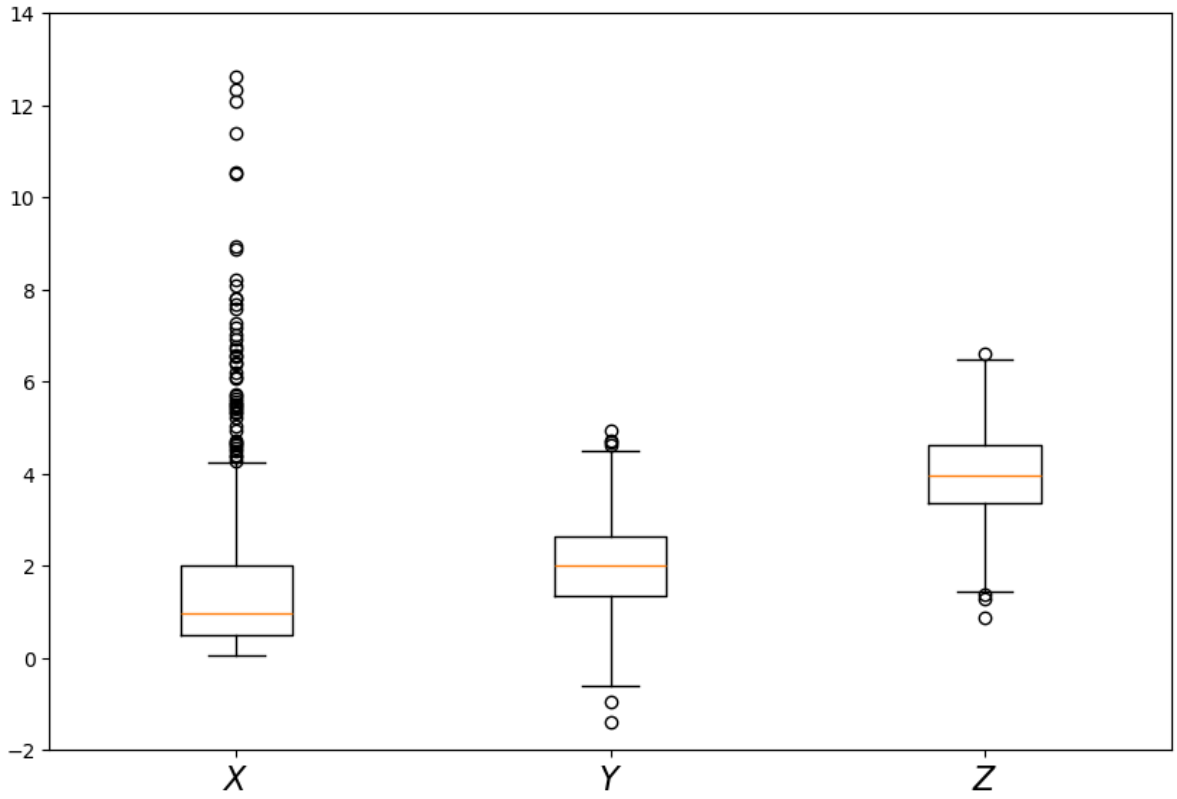
The three data sets we will use are:

$$\{X_1, \ldots, X_n\} \sim LN(0, 1), \quad \{Y_1, \ldots, Y_n\} \sim N(2, 1), \quad \text{and} \quad \{Z_1, \ldots, Z_n\} \sim N(4, 1),$$

Here is the code and figure:

```
n = 500
x = np.random.randn(n)          # N(0, 1)
x = np.exp(x)                   # Map x to lognormal
y = np.random.randn(n) + 2.0    # N(2, 1)
z = np.random.randn(n) + 4.0    # N(4, 1)

fig, ax = plt.subplots(figsize=(10, 6.6))
ax.boxplot([x, y, z])
ax.set_xticks((1, 2, 3))
ax.set_ylim(-2, 14)
ax.set_xticklabels(('$X$', '$Y$', '$Z$'), fontsize=16)
plt.show()
```



Each data set is represented by a box, where the top and bottom of the box are the third and first quartiles of the data, and the red line in the center is the median.

The boxes give some indication as to

- the location of probability mass for each sample
- whether the distribution is right-skewed (as is the lognormal distribution), etc

Now let's put these ideas to use in a simulation.

Consider the threshold autoregressive model in (2.19).

We know that the distribution of X_t will converge to (2.20) whenever $|\theta| < 1$.

Let's observe this convergence from different initial conditions using boxplots.

In particular, the exercise is to generate J boxplot figures, one for each initial condition X_0 in

```
initial_conditions = np.linspace(8, 0, J)
```

For each X_0 in this set,

1. Generate k time-series of length n, each starting at X_0 and obeying (2.19).
2. Create a boxplot representing n distributions, where the t-th distribution shows the k observations of X_t.

Use $\theta = 0.9, n = 20, k = 5000, J = 8$

Solution to Exercise 2.5.3

Here's a possible solution.

Note the way we use vectorized code to simulate the k time series for one boxplot all at once

```
n = 20
k = 5000
J = 8

θ = 0.9
d = np.sqrt(1 - θ**2)
δ = θ / d

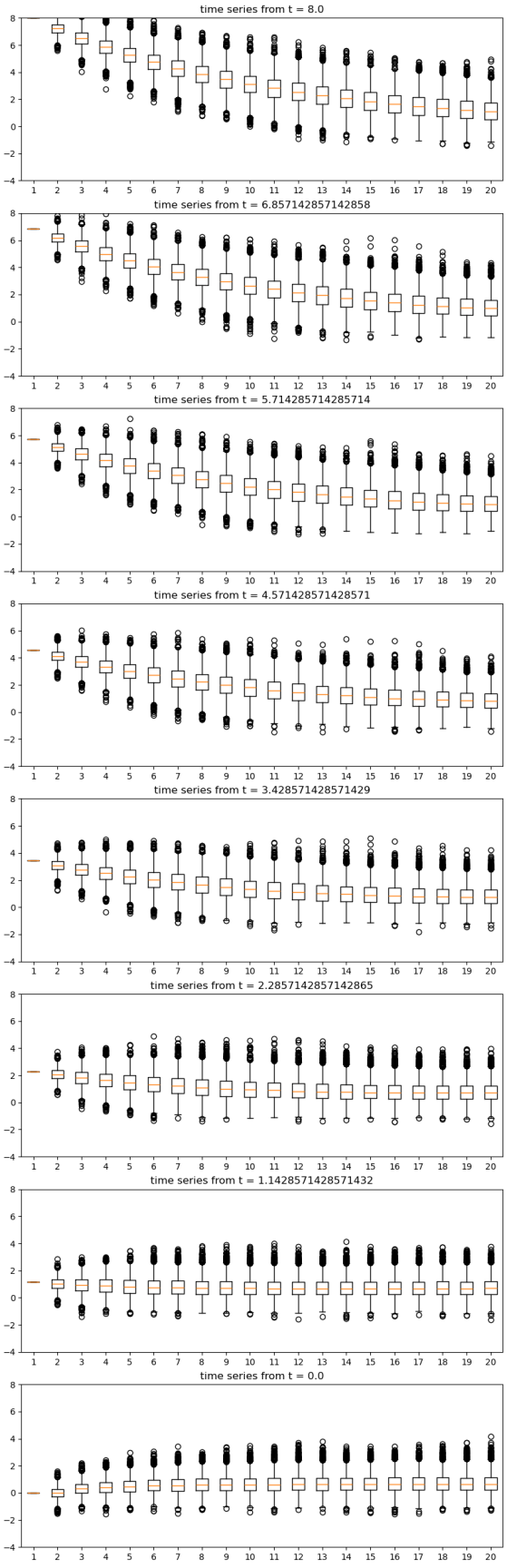
fig, axes = plt.subplots(J, 1, figsize=(10, 4*J))
initial_conditions = np.linspace(8, 0, J)
X = np.empty((k, n))

for j in range(J):

    axes[j].set_ylim(-4, 8)
    axes[j].set_title(f'time series from t = {initial_conditions[j]}')

    Z = np.random.randn(k, n)
    X[:, 0] = initial_conditions[j]
    for t in range(1, n):
        X[:, t] = θ * np.abs(X[:, t-1]) + d * Z[:, t]
    axes[j].boxplot(X)

plt.show()
```


time series from t = 8.0
time series from t = 6.857142857142858
time series from t = 5.714285714285714
time series from t = 4.571428571428571
time series from t = 3.428571428571429
time series from t = 2.2857142857142865
time series from t = 1.1428571428571432
time series from t = 0.0

2.6 Appendix

Here's the proof of (2.6).

Let F_U and F_V be the cumulative distributions of U and V respectively.

By the definition of V, we have $F_V(v) = \mathbb{P}\{a + bU \leq v\} = \mathbb{P}\{U \leq (v-a)/b\}$.

In other words, $F_V(v) = F_U((v-a)/b)$.

Differentiating with respect to v yields (2.6).

CHAPTER

THREE

REVERSE ENGINEERING A LA MUTH

In addition to what's in Anaconda, this lecture uses the quantecon library.

```
!pip install --upgrade quantecon
```

We'll also need the following imports:

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

from quantecon import Kalman
from quantecon import LinearStateSpace
np.set_printoptions(linewidth=120, precision=4, suppress=True)
```

This lecture uses the Kalman filter to reformulate John F. Muth's first paper [Muth, 1960] about rational expectations.

Muth used *classical* prediction methods to reverse engineer a stochastic process that renders optimal Milton Friedman's [Friedman, 1956] "adaptive expectations" scheme.

3.1 Friedman (1956) and Muth (1960)

Milton Friedman [Friedman, 1956] (1956) posited that consumer's forecast their future disposable income with the adaptive expectations scheme

$$y^*_{t+i,t} = K \sum_{j=0}^{\infty} (1-K)^j y_{t-j} \tag{3.1}$$

where $K \in (0,1)$ and $y^*_{t+i,t}$ is a forecast of future y over horizon i.

Milton Friedman justified the **exponential smoothing** forecasting scheme (3.1) informally, noting that it seemed a plausible way to use past income to forecast future income.

In his first paper about rational expectations, John F. Muth [Muth, 1960] reverse-engineered a univariate stochastic process $\{y_t\}_{t=-\infty}^{\infty}$ for which Milton Friedman's adaptive expectations scheme gives linear least forecasts of y_{t+j} for any horizon i.

Muth sought a setting and a sense in which Friedman's forecasting scheme is optimal.

That is, Muth asked for what optimal forecasting **question** is Milton Friedman's adaptive expectation scheme the **answer**.

Muth (1960) used classical prediction methods based on lag-operators and z-transforms to find the answer to his question.

Please see lectures *Classical Control with Linear Algebra* and *Classical Filtering and Prediction with Linear Algebra* for an introduction to the classical tools that Muth used.

Rather than using those classical tools, in this lecture we apply the Kalman filter to express the heart of Muth's analysis concisely.

The lecture First Look at Kalman Filter describes the Kalman filter.

We'll use limiting versions of the Kalman filter corresponding to what are called **stationary values** in that lecture.

3.2 A Process for Which Adaptive Expectations are Optimal

Suppose that an observable y_t is the sum of an unobserved random walk x_t and an IID shock $\epsilon_{2,t}$:

$$\begin{aligned} x_{t+1} &= x_t + \sigma_x \epsilon_{1,t+1} \\ y_t &= x_t + \sigma_y \epsilon_{2,t} \end{aligned} \tag{3.2}$$

where

$$\begin{bmatrix} \epsilon_{1,t+1} \\ \epsilon_{2,t} \end{bmatrix} \sim \mathcal{N}(0, I)$$

is an IID process.

Note: A property of the state-space representation (3.2) is that in general neither $\epsilon_{1,t}$ nor $\epsilon_{2,t}$ is in the space spanned by square-summable linear combinations of $y_t, y_{t-1}, \ldots$.

In general $\begin{bmatrix} \epsilon_{1,t} \\ \epsilon_{2t} \end{bmatrix}$ has more information about future y_{t+j}'s than is contained in $y_t, y_{t-1}, \ldots$.

We can use the asymptotic or stationary values of the Kalman gain and the one-step-ahead conditional state covariance matrix to compute a time-invariant *innovations representation*

$$\begin{aligned} \hat{x}_{t+1} &= \hat{x}_t + K a_t \\ y_t &= \hat{x}_t + a_t \end{aligned} \tag{3.3}$$

where $\hat{x}_t = E[x_t | y_{t-1}, y_{t-2}, \ldots]$ and $a_t = y_t - E[y_t | y_{t-1}, y_{t-2}, \ldots]$.

Note: A key property about an *innovations representation* is that a_t is in the space spanned by square summable linear combinations of $y_t, y_{t-1}, \ldots$.

For more ramifications of this property, see the lectures *Shock Non-Invertibility* and *Recursive Models of Dynamic Linear Economies*.

Later we'll stack these state-space systems (3.2) and (3.3) to display some classic findings of Muth.

But first, let's create an instance of the state-space system (3.2) then apply the quantecon `Kalman` class, then uses it to construct the associated "innovations representation"

```
# Make some parameter choices
# sigx/sigy are state noise std err and measurement noise std err
μ_0, σ_x, σ_y = 10, 1, 5

# Create a LinearStateSpace object
A, C, G, H = 1, σ_x, 1, σ_y
ss = LinearStateSpace(A, C, G, H, mu_0=μ_0)
```

(continues on next page)

(continued from previous page)

```
# Set prior and initialize the Kalman type
x_hat_0, Σ_0 = 10, 1
kmuth = Kalman(ss, x_hat_0, Σ_0)

# Computes stationary values which we need for the innovation
# representation
S1, K1 = kmuth.stationary_values()

# Extract scalars from nested arrays
S1, K1 = S1.item(), K1.item()

# Form innovation representation state-space
Ak, Ck, Gk, Hk = A, K1, G, 1

ssk = LinearStateSpace(Ak, Ck, Gk, Hk, mu_0=x_hat_0)
```

3.3 Some Useful State-Space Math

Now we want to map the time-invariant innovations representation (3.3) and the original state-space system (3.2) into a convenient form for deducing the impulse responses from the original shocks to the x_t and $\widehat{x}_t$.

Putting both of these representations into a single state-space system is yet another application of the insight that "finding the state is an art".

We'll define a state vector and appropriate state-space matrices that allow us to represent both systems in one fell swoop.

Note that

$$a_t = x_t + \sigma_y \epsilon_{2,t} - \widehat{x}_t$$

so that

$$\begin{aligned} \widehat{x}_{t+1} &= \widehat{x}_t + K(x_t + \sigma_y \epsilon_{2,t} - \widehat{x}_t) \\ &= (1-K)\widehat{x}_t + Kx_t + K\sigma_y \epsilon_{2,t} \end{aligned}$$

The stacked system

$$\begin{bmatrix} x_{t+1} \\ \widehat{x}_{t+1} \\ \epsilon_{2,t+1} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ K & (1-K) & K\sigma_y \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_t \\ \widehat{x}_t \\ \epsilon_{2,t} \end{bmatrix} + \begin{bmatrix} \sigma_x & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \epsilon_{1,t+1} \\ \epsilon_{2,t+1} \end{bmatrix}$$

$$\begin{bmatrix} y_t \\ a_t \end{bmatrix} = \begin{bmatrix} 1 & 0 & \sigma_y \\ 1 & -1 & \sigma_y \end{bmatrix} \begin{bmatrix} x_t \\ \widehat{x}_t \\ \epsilon_{2,t} \end{bmatrix}$$

is a state-space system that tells us how the shocks $\begin{bmatrix} \epsilon_{1,t+1} \\ \epsilon_{2,t+1} \end{bmatrix}$ affect states $\widehat{x}_{t+1}, x_t$, the observable y_t, and the innovation a_t.

With this tool at our disposal, let's form the composite system and simulate it

```
# Create grand state-space for y_t, a_t as observed vars -- Use
# stacking trick above
Af = np.array([[ 1,      0,        0],
               [K1, 1 - K1, K1 * σ_y],
               [ 0,      0,        0]])
Cf = np.array([[σ_x,        0],
               [  0, K1 * σ_y],
               [  0,        1]])
Gf = np.array([[1,  0, σ_y],
               [1, -1, σ_y]])

μ_true, μ_prior = 10, 10
μ_f = np.array([μ_true, μ_prior, 0]).reshape(3, 1)

# Create the state-space
ssf = LinearStateSpace(Af, Cf, Gf, mu_0=μ_f)

# Draw observations of y from the state-space model
N = 50
xf, yf = ssf.simulate(N)

print(f"Kalman gain = {K1}")
print(f"Conditional variance = {S1}")
```

```
Kalman gain = 0.1809975124224177
Conditional variance = 5.524937810560442
```

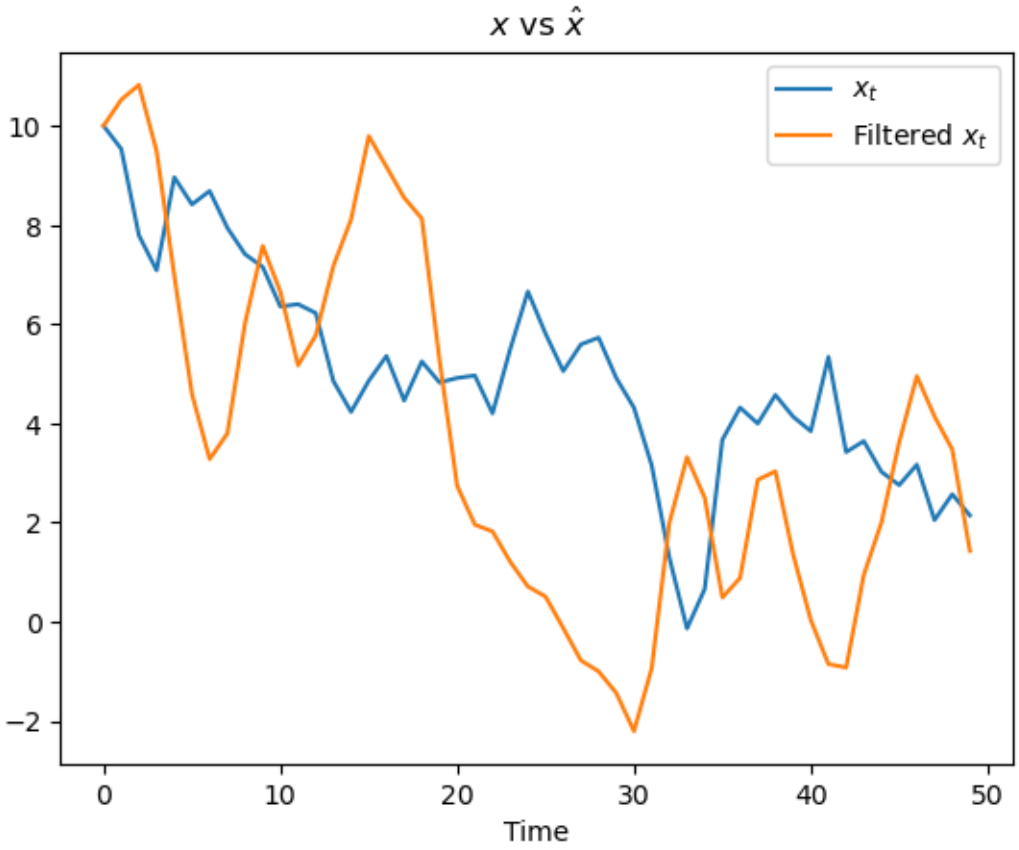
Now that we have simulated our joint system, we have x_t, $\hat{x}_t$, and y_t.

We can now investigate how these variables are related by plotting some key objects.

3.4 Estimates of Unobservables

First, let's plot the hidden state x_t and the filtered version $\hat{x}_t$ that is linear-least squares projection of x_t on the history $y_{t-1}, y_{t-2}, \cdots$

```
fig, ax = plt.subplots()
ax.plot(xf[0, :], label="$x_t$")
ax.plot(xf[1, :], label="Filtered $x_t$")
ax.legend()
ax.set_xlabel("Time")
ax.set_title(r"$x$ vs $\hat{x}$")
plt.show()
```



Note how x_t and $\hat{x_t}$ differ.

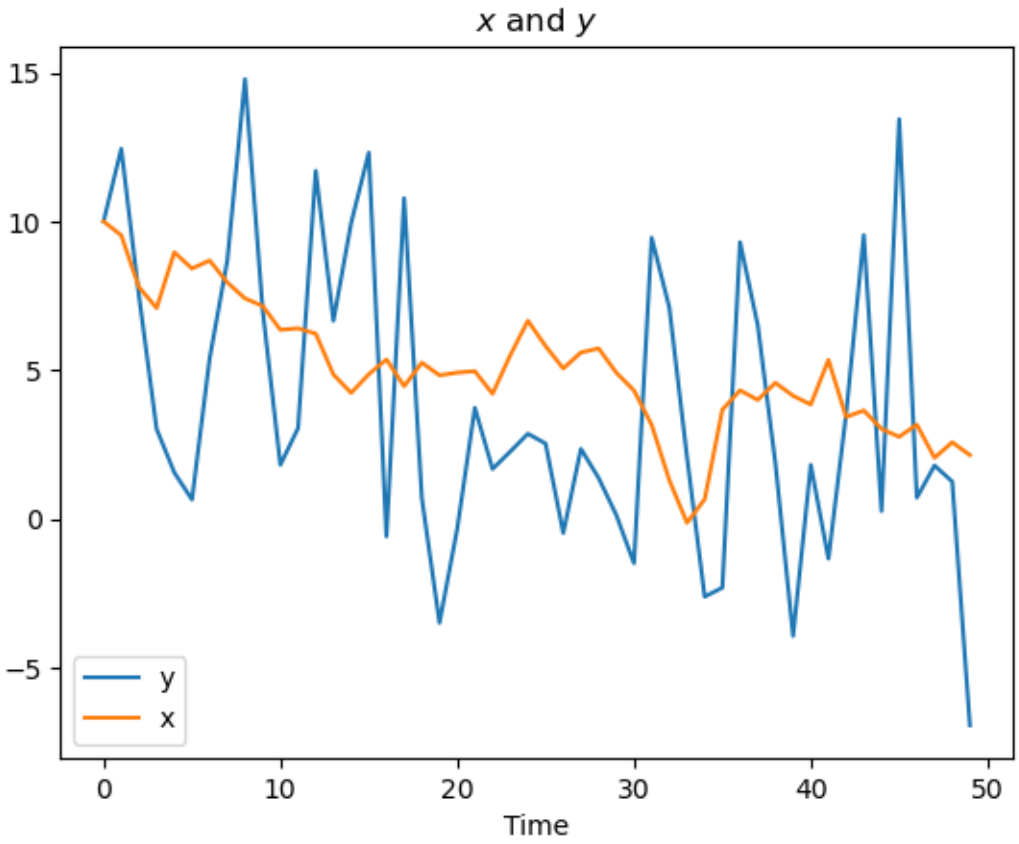
For Friedman, $\hat{x}_t$ and not x_t is the consumer's idea about her/his *permanent income*.

3.5 Relationship of Unobservables to Observables

Now let's plot x_t and y_t.

Recall that y_t is just x_t plus white noise

```
fig, ax = plt.subplots()
ax.plot(yf[0, :], label="y")
ax.plot(xf[0, :], label="x")
ax.legend()
ax.set_title(r"$x$ and $y$")
ax.set_xlabel("Time")
plt.show()
```

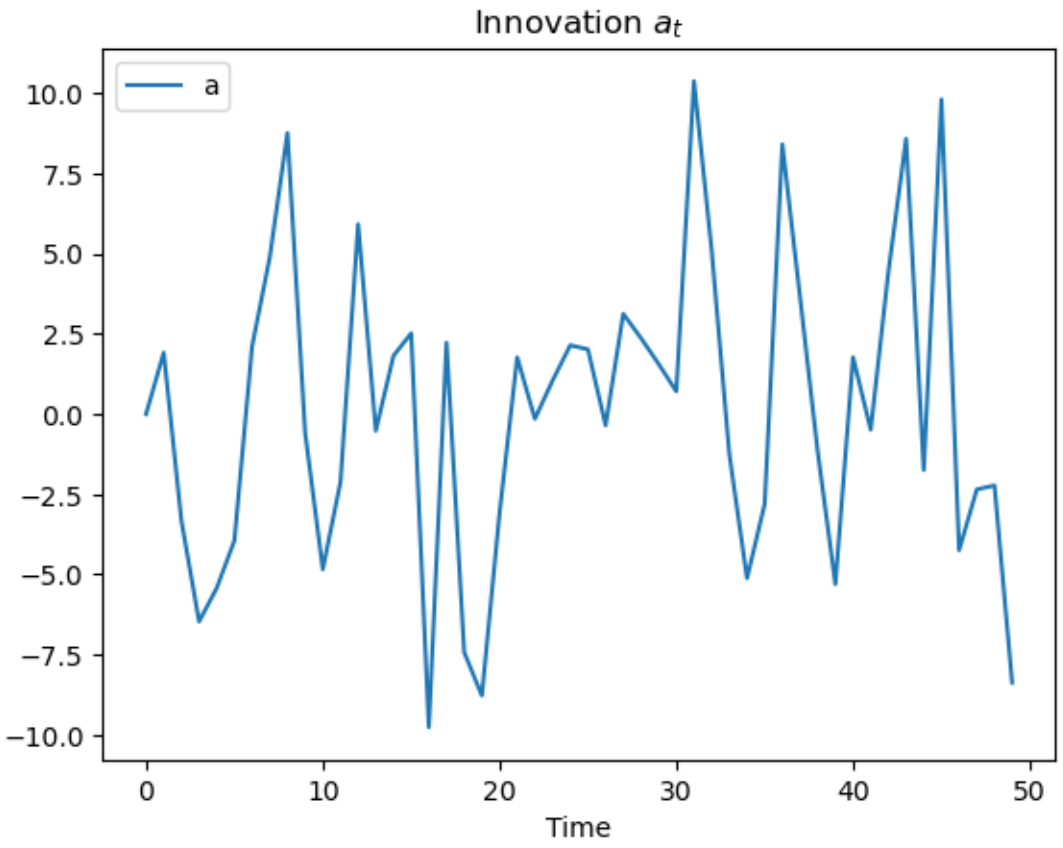


We see above that y seems to look like white noise around the values of x.

3.5.1 Innovations

Recall that we wrote down the innovation representation that depended on a_t. We now plot the innovations $\{a_t\}$:

```
fig, ax = plt.subplots()
ax.plot(yf[1, :], label="a")
ax.legend()
ax.set_title(r"Innovation $a_t$")
ax.set_xlabel("Time")
plt.show()
```



3.6 MA and AR Representations

Now we shall extract from the `Kalman` instance `kmuth` coefficients of

- a fundamental moving average representation that represents y_t as a one-sided moving sum of current and past a_ts that are square summable linear combinations of $y_t, y_{t-1}, \ldots$.
- a univariate autoregression representation that depicts the coefficients in a linear least square projection of y_t on the semi-infinite history $y_{t-1}, y_{t-2}, \ldots$.

Then we'll plot each of them

```
# Kalman Methods for MA and VAR
coefs_ma = kmuth.stationary_coefficients(5, "ma")
coefs_var = kmuth.stationary_coefficients(5, "var")

# Coefficients come in a list of arrays, but we
# want to plot them and so need to stack into an array
coefs_ma_array = np.vstack(coefs_ma)
coefs_var_array = np.vstack(coefs_var)

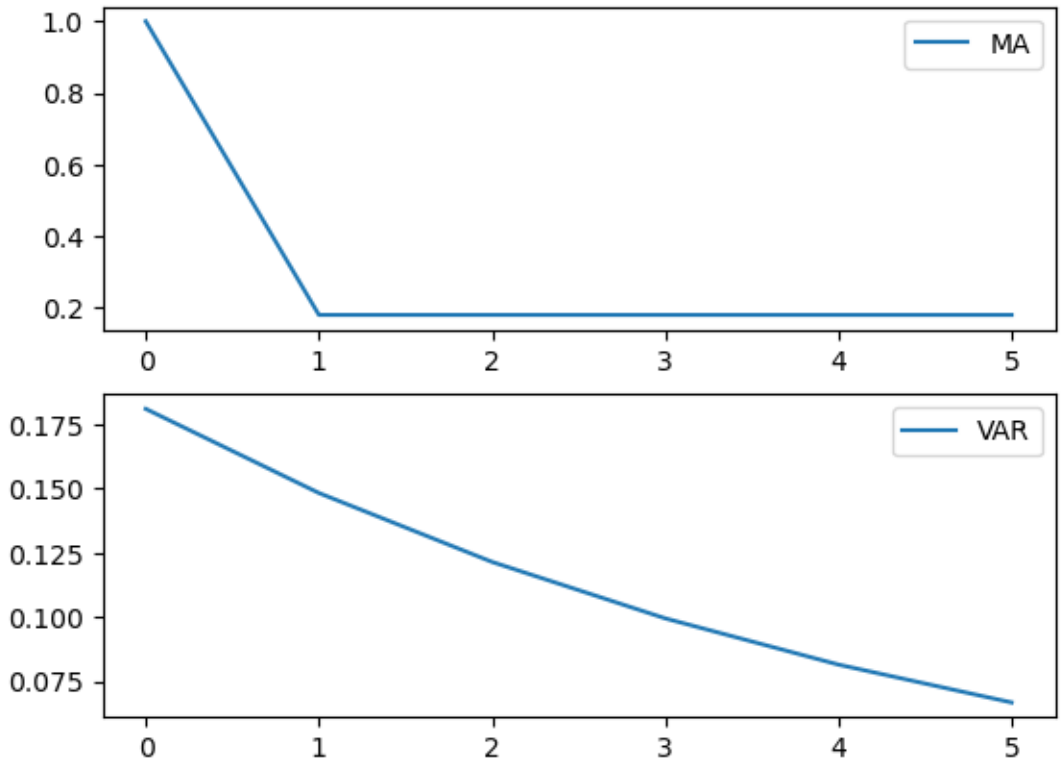
fig, ax = plt.subplots(2)
ax[0].plot(coefs_ma_array, label="MA")
ax[0].legend()
ax[1].plot(coefs_var_array, label="VAR")
```

(continues on next page)

(continued from previous page)

```
ax[1].legend()

plt.show()
```



The **moving average** coefficients in the top panel show tell-tale signs of y_t being a process whose first difference is a first-order autoregression.

The **autoregressive coefficients** decline geometrically with decay rate $(1 - K)$.

These are exactly the target outcomes that Muth (1960) aimed to reverse engineer

```
print(f'decay parameter 1 - K1 = {1 - K1}')
```

```
decay parameter 1 - K1 = 0.8190024875775823
```

CHAPTER

FOUR

DISCRETE STATE DYNAMIC PROGRAMMING

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install --upgrade quantecon
```

4.1 Overview

In this lecture we discuss a family of dynamic programming problems with the following features:

1. a discrete state space and discrete choices (actions)
2. an infinite horizon
3. discounted rewards
4. Markov state transitions

We call such problems discrete dynamic programs or discrete DPs.

Discrete DPs are the workhorses in much of modern quantitative economics, including

- monetary economics
- search and labor economics
- household savings and consumption theory
- investment theory
- asset pricing
- industrial organization, etc.

When a given model is not inherently discrete, it is common to replace it with a discretized version in order to use discrete DP techniques.

This lecture covers

- the theory of dynamic programming in a discrete setting, plus examples and applications
- a powerful set of routines for solving discrete DPs from the QuantEcon code library

Let's start with some imports:

```
import numpy as np
import matplotlib.pyplot as plt
import quantecon as qe
import scipy.sparse as sparse
```

(continues on next page)

(continued from previous page)

```
from quantecon import compute_fixed_point
from quantecon.markov import DiscreteDP
```

4.1.1 How to Read this Lecture

We use dynamic programming many applied lectures, such as

- The shortest path lecture
- The McCall search model lecture

The objective of this lecture is to provide a more systematic and theoretical treatment, including algorithms and implementation while focusing on the discrete case.

4.1.2 Code

Among other things, it offers

- a flexible, well-designed interface
- multiple solution methods, including value function and policy function iteration
- high-speed operations via carefully optimized JIT-compiled functions
- the ability to scale to large problems by minimizing vectorized operators and allowing operations on sparse matrices

JIT compilation relies on Numba, which should work seamlessly if you are using Anaconda as suggested.

4.1.3 References

For background reading on dynamic programming and additional applications, see, for example,

- [Ljungqvist and Sargent, 2018]
- [Hernandez-Lerma and Lasserre, 1996], section 3.5
- [Puterman, 2005]
- [Stokey *et al.*, 1989]
- [Rust, 1996]
- [Miranda and Fackler, 2002]
- EDTC, chapter 5

4.2 Discrete DPs

Loosely speaking, a discrete DP is a maximization problem with an objective function of the form

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t r(s_t, a_t) \tag{4.1}$$

where

- s_t is the state variable

- a_t is the action
- β is a discount factor
- $r(s_t, a_t)$ is interpreted as a current reward when the state is s_t and the action chosen is a_t

Each pair (s_t, a_t) pins down transition probabilities $Q(s_t, a_t, s_{t+1})$ for the next period state s_{t+1}.

Thus, actions influence not only current rewards but also the future time path of the state.

The essence of dynamic programming problems is to trade off current rewards vs favorable positioning of the future state (modulo randomness).

Examples:

- consuming today vs saving and accumulating assets
- accepting a job offer today vs seeking a better one in the future
- exercising an option now vs waiting

4.2.1 Policies

The most fruitful way to think about solutions to discrete DP problems is to compare *policies*.

In general, a policy is a randomized map from past actions and states to current action.

In the setting formalized below, it suffices to consider so-called *stationary Markov policies*, which consider only the current state.

In particular, a stationary Markov policy is a map σ from states to actions

- $a_t = \sigma(s_t)$ indicates that a_t is the action to be taken in state s_t

It is known that, for any arbitrary policy, there exists a stationary Markov policy that dominates it at least weakly.

- See section 5.5 of [Puterman, 2005] for discussion and proofs.

In what follows, stationary Markov policies are referred to simply as policies.

The aim is to find an optimal policy, in the sense of one that maximizes (4.1).

Let's now step through these ideas more carefully.

4.2.2 Formal Definition

Formally, a discrete dynamic program consists of the following components:

1. A finite set of *states* $S = \{0, \ldots, n-1\}$.
2. A finite set of *feasible actions* $A(s)$ for each state $s \in S$, and a corresponding set of *feasible state-action pairs*.

$$SA := \{(s, a) \mid s \in S,\ a \in A(s)\}$$

3. A *reward function* $r \colon SA \to \mathbb{R}$.
4. A *transition probability function* $Q \colon SA \to \Delta(S)$, where $\Delta(S)$ is the set of probability distributions over S.
5. A *discount factor* $\beta \in [0, 1)$.

We also use the notation $A := \bigcup_{s \in S} A(s) = \{0, \ldots, m-1\}$ and call this set the *action space*.

A *policy* is a function $\sigma \colon S \to A$.

A policy is called *feasible* if it satisfies $\sigma(s) \in A(s)$ for all $s \in S$.

Denote the set of all feasible policies by Σ.

If a decision-maker uses a policy $\sigma \in \Sigma$, then

- the current reward at time t is $r(s_t, \sigma(s_t))$
- the probability that $s_{t+1} = s'$ is $Q(s_t, \sigma(s_t), s')$

For each $\sigma \in \Sigma$, define

- r_σ by $r_\sigma(s) := r(s, \sigma(s)))$
- Q_σ by $Q_\sigma(s, s') := Q(s, \sigma(s), s')$

Notice that Q_σ is a stochastic matrix on S.

It gives transition probabilities of the *controlled chain* when we follow policy σ.

If we think of r_σ as a column vector, then so is $Q_\sigma^t r_\sigma$, and the s-th row of the latter has the interpretation

$$(Q_\sigma^t r_\sigma)(s) = \mathbb{E}[r(s_t, \sigma(s_t)) \mid s_0 = s] \quad \text{when } \{s_t\} \sim Q_\sigma \tag{4.2}$$

Comments

- $\{s_t\} \sim Q_\sigma$ means that the state is generated by stochastic matrix Q_σ.
- See this discussion on computing expectations of Markov chains for an explanation of the expression in (4.2).

Notice that we're not really distinguishing between functions from S to $\mathbb{R}$ and vectors in $\mathbb{R}^n$.

This is natural because they are in one to one correspondence.

4.2.3 Value and Optimality

Let $v_\sigma(s)$ denote the discounted sum of expected reward flows from policy σ when the initial state is s.

To calculate this quantity we pass the expectation through the sum in (4.1) and use (4.2) to get

$$v_\sigma(s) = \sum_{t=0}^{\infty} \beta^t (Q_\sigma^t r_\sigma)(s) \qquad (s \in S)$$

This function is called the *policy value function* for the policy σ.

The *optimal value function*, or simply *value function*, is the function $v^* \colon S \to \mathbb{R}$ defined by

$$v^*(s) = \max_{\sigma \in \Sigma} v_\sigma(s) \qquad (s \in S)$$

(We can use max rather than sup here because the domain is a finite set)

A policy $\sigma \in \Sigma$ is called *optimal* if $v_\sigma(s) = v^*(s)$ for all $s \in S$.

Given any $w \colon S \to \mathbb{R}$, a policy $\sigma \in \Sigma$ is called w-greedy if

$$\sigma(s) \in \operatorname*{arg\,max}_{a \in A(s)} \left\{ r(s, a) + \beta \sum_{s' \in S} w(s') Q(s, a, s') \right\} \qquad (s \in S)$$

As discussed in detail below, optimal policies are precisely those that are v^*-greedy.

4.2.4 Two Operators

It is useful to define the following operators:

- The *Bellman operator* $T\colon \mathbb{R}^S \to \mathbb{R}^S$ is defined by

$$(Tv)(s) = \max_{a \in A(s)} \left\{ r(s,a) + \beta \sum_{s' \in S} v(s') Q(s,a,s') \right\} \qquad (s \in S)$$

- For any policy function $\sigma \in \Sigma$, the operator $T_\sigma\colon \mathbb{R}^S \to \mathbb{R}^S$ is defined by

$$(T_\sigma v)(s) = r(s,\sigma(s)) + \beta \sum_{s' \in S} v(s') Q(s,\sigma(s),s') \qquad (s \in S)$$

This can be written more succinctly in operator notation as

$$T_\sigma v = r_\sigma + \beta Q_\sigma v$$

The two operators are both monotone

- $v \leq w$ implies $Tv \leq Tw$ pointwise on S, and similarly for T_σ

They are also contraction mappings with modulus β

- $\|Tv - Tw\| \leq \beta \|v - w\|$ and similarly for T_σ, where $\|\cdot\|$ is the max norm

For any policy σ, its value v_σ is the unique fixed point of T_σ.

For proofs of these results and those in the next section, see, for example, EDTC, chapter 10.

4.2.5 The Bellman Equation and the Principle of Optimality

The main principle of the theory of dynamic programming is that

- the optimal value function v^* is a unique solution to the *Bellman equation*

$$v(s) = \max_{a \in A(s)} \left\{ r(s,a) + \beta \sum_{s' \in S} v(s') Q(s,a,s') \right\} \qquad (s \in S)$$

or in other words, v^* is the unique fixed point of T, and

- σ^* is an optimal policy function if and only if it is v^*-greedy

By the definition of greedy policies given above, this means that

$$\sigma^*(s) \in \operatorname*{arg\,max}_{a \in A(s)} \left\{ r(s,a) + \beta \sum_{s' \in S} v^*(s') Q(s,a,s') \right\} \qquad (s \in S)$$

4.3 Solving Discrete DPs

Now that the theory has been set out, let's turn to solution methods.

The code for solving discrete DPs is available in ddp.py from the QuantEcon.py code library.

It implements the three most important solution methods for discrete dynamic programs, namely

- value function iteration
- policy function iteration
- modified policy function iteration

Let's briefly review these algorithms and their implementation.

4.3.1 Value Function Iteration

Perhaps the most familiar method for solving all manner of dynamic programs is value function iteration.

This algorithm uses the fact that the Bellman operator T is a contraction mapping with fixed point v^*.

Hence, iterative application of T to any initial function $v^0 : S \to \mathbb{R}$ converges to v^*.

The details of the algorithm can be found in *the appendix*.

4.3.2 Policy Function Iteration

This routine, also known as Howard's policy improvement algorithm, exploits more closely the particular structure of a discrete DP problem.

Each iteration consists of

1. A policy evaluation step that computes the value v_σ of a policy σ by solving the linear equation $v = T_\sigma v$.
2. A policy improvement step that computes a v_σ-greedy policy.

In the current setting, policy iteration computes an exact optimal policy in finitely many iterations.

- See theorem 10.2.6 of EDTC for a proof.

The details of the algorithm can be found in *the appendix*.

4.3.3 Modified Policy Function Iteration

Modified policy iteration replaces the policy evaluation step in policy iteration with "partial policy evaluation".

The latter computes an approximation to the value of a policy σ by iterating T_σ for a specified number of times.

This approach can be useful when the state space is very large and the linear system in the policy evaluation step of policy iteration is correspondingly difficult to solve.

The details of the algorithm can be found in *the appendix*.

4.4 Example: A Growth Model

Let's consider a simple consumption-saving model.

A single household either consumes or stores its own output of a single consumption good.

The household starts each period with current stock s.

Next, the household chooses a quantity a to store and consumes $c = s - a$

- Storage is limited by a global upper bound M.
- Flow utility is $u(c) = c^\alpha$.

Output is drawn from a discrete uniform distribution on $\{0, \dots, B\}$.

The next period stock is therefore

$$s' = a + U \quad \text{where} \quad U \sim U[0, \dots, B]$$

The discount factor is $\beta \in [0, 1)$.

4.4.1 Discrete DP Representation

We want to represent this model in the format of a discrete dynamic program.

To this end, we take

- the state variable to be the stock s
- the state space to be $S = \{0, \dots, M + B\}$
  - hence $n = M + B + 1$
- the action to be the storage quantity a
- the set of feasible actions at s to be $A(s) = \{0, \dots, \min\{s, M\}\}$
  - hence $A = \{0, \dots, M\}$ and $m = M + 1$
- the reward function to be $r(s, a) = u(s - a)$
- the transition probabilities to be

$$Q(s, a, s') := \begin{cases} \frac{1}{B+1} & \text{if } a \leq s' \leq a + B \\ 0 & \text{otherwise} \end{cases} \tag{4.3}$$

4.4.2 Defining a DiscreteDP Instance

This information will be used to create an instance of DiscreteDP by passing the following information

1. An $n \times m$ reward array R.
2. An $n \times m \times n$ transition probability array Q.
3. A discount factor β.

For R we set $R[s, a] = u(s - a)$ if $a \leq s$ and $-\infty$ otherwise.

For Q we follow the rule in (4.3).

Note:

- The feasibility constraint is embedded into R by setting $R[s, a] = -\infty$ for $a \notin A(s)$.
- Probability distributions for (s, a) with $a \notin A(s)$ can be arbitrary.

The following code sets up these objects for us

```
class SimpleOG:

    def __init__(self, B=10, M=5, α=0.5, β=0.9):
        """
        Set up R, Q and β, the three elements that define an instance of
        the DiscreteDP class.
        """

        self.B, self.M, self.α, self.β  = B, M, α, β
        self.n = B + M + 1
        self.m = M + 1

        self.R = np.empty((self.n, self.m))
```

(continues on next page)

(continued from previous page)

```
        self.Q = np.zeros((self.n, self.m, self.n))

        self.populate_Q()
        self.populate_R()

    def u(self, c):
        return c**self.α

    def populate_R(self):
        """
        Populate the R matrix, with R[s, a] = -np.inf for infeasible
        state-action pairs.
        """
        for s in range(self.n):
            for a in range(self.m):
                self.R[s, a] = self.u(s - a) if a <= s else -np.inf

    def populate_Q(self):
        """
        Populate the Q matrix by setting

            Q[s, a, s'] = 1 / (1 + B) if a <= s' <= a + B

        and zero otherwise.
        """

        for a in range(self.m):
            self.Q[:, a, a:(a + self.B + 1)] = 1.0 / (self.B + 1)
```

Let's run this code and create an instance of `SimpleOG`.

```
g = SimpleOG()  # Use default parameters
```

Instances of `DiscreteDP` are created using the signature `DiscreteDP(R, Q, β)`.

Let's create an instance using the objects stored in `g`

```
ddp = qe.markov.DiscreteDP(g.R, g.Q, g.β)
```

Now that we have an instance `ddp` of `DiscreteDP` we can solve it as follows

```
results = ddp.solve(method='policy_iteration')
```

Let's see what we've got here

```
dir(results)
```

```
['max_iter', 'mc', 'method', 'num_iter', 'sigma', 'v']
```

(In IPython version 4.0 and above you can also type `results.` and hit the tab key)

The most important attributes are `v`, the value function, and `σ`, the optimal policy

```
results.v
```

```
array([19.01740222, 20.01740222, 20.43161578, 20.74945302, 21.04078099,
       21.30873018, 21.54479816, 21.76928181, 21.98270358, 22.18824323,
       22.3845048 , 22.57807736, 22.76109127, 22.94376708, 23.11533996,
       23.27761762])
```

```
results.sigma
```

```
array([0, 0, 0, 0, 1, 1, 1, 2, 2, 3, 3, 4, 5, 5, 5, 5])
```

Since we've used policy iteration, these results will be exact unless we hit the iteration bound `max_iter`.

Let's make sure this didn't happen

```
results.max_iter
```

```
250
```

```
results.num_iter
```

```
3
```

Another interesting object is `results.mc`, which is the controlled chain defined by Q_{σ^*}, where σ^* is the optimal policy.

In other words, it gives the dynamics of the state when the agent follows the optimal policy.

Since this object is an instance of MarkovChain from QuantEcon.py (see this lecture for more discussion), we can easily simulate it, compute its stationary distribution and so on.

```
results.mc.stationary_distributions
```

```
array([[0.01732187, 0.04121063, 0.05773956, 0.07426848, 0.08095823,
        0.09090909, 0.09090909, 0.09090909, 0.09090909, 0.09090909,
        0.09090909, 0.07358722, 0.04969846, 0.03316953, 0.01664061,
        0.00995086]])
```

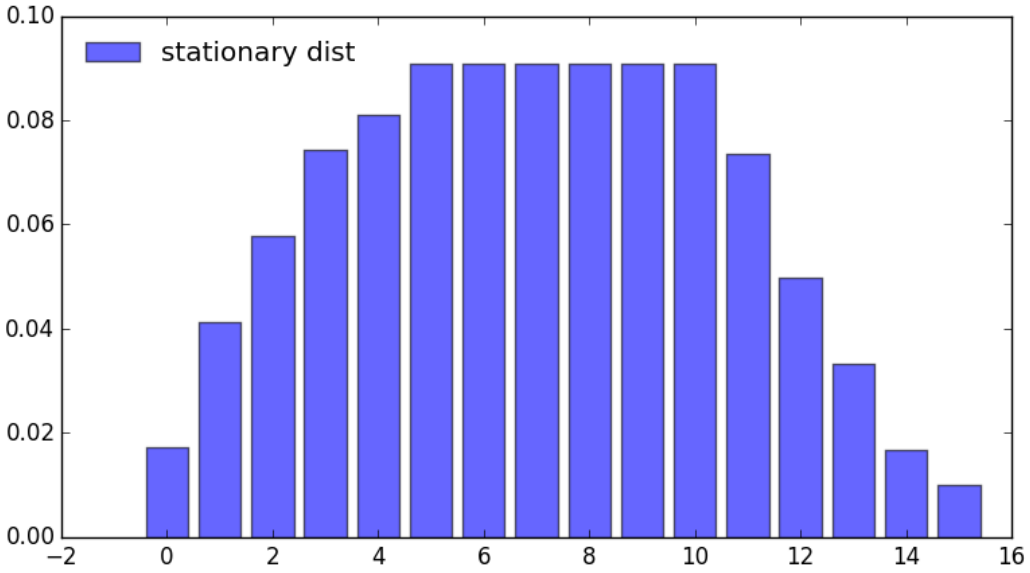
Here's the same information in a bar graph

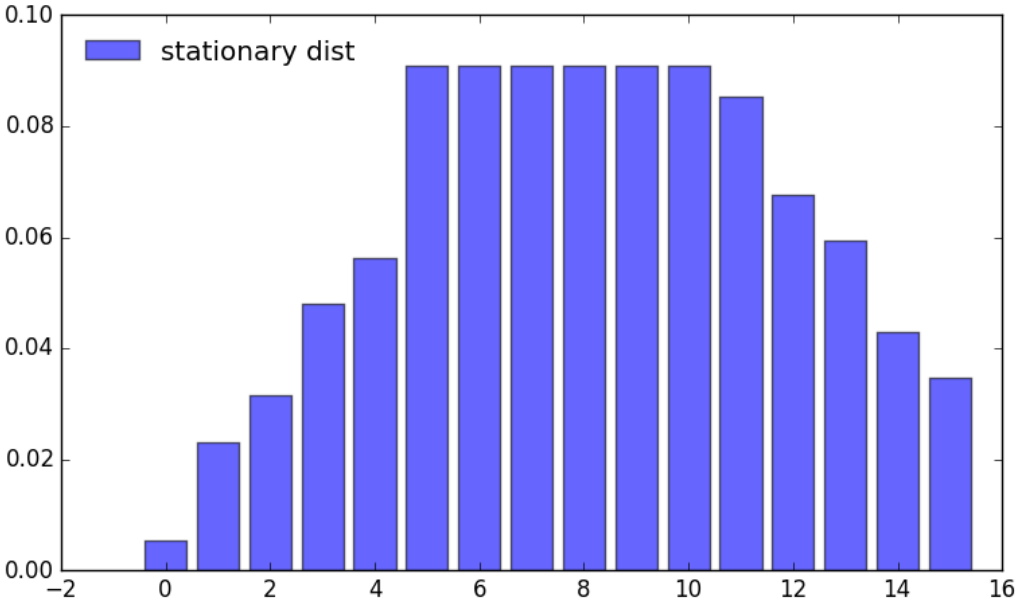
What happens if the agent is more patient?

```
ddp = qe.markov.DiscreteDP(g.R, g.Q, 0.99)  # Increase β to 0.99
results = ddp.solve(method='policy_iteration')
results.mc.stationary_distributions
```

```
array([[0.00546913, 0.02321342, 0.03147788, 0.04800681, 0.05627127,
        0.09090909, 0.09090909, 0.09090909, 0.09090909, 0.09090909,
        0.09090909, 0.08543996, 0.06769567, 0.05943121, 0.04290228,
        0.03463782]])
```

If we look at the bar graph we can see the rightward shift in probability mass


stationary dist
0.10
0.08
0.06
0.04
0.02
0.00
−2
0
2
4
6
8
10
12
14
16


stationary dist
0.10
0.08
0.06
0.04
0.02
0.00
−2
0
2
4
6
8
10
12
14
16

4.4.3 State-Action Pair Formulation

The `DiscreteDP` class in fact, provides a second interface to set up an instance.

One of the advantages of this alternative set up is that it permits the use of a sparse matrix for `Q`.

(An example of using sparse matrices is given in the exercises below)

The call signature of the second formulation is `DiscreteDP(R, Q, β, s_indices, a_indices)` where

- `s_indices` and `a_indices` are arrays of equal length `L` enumerating all feasible state-action pairs
- `R` is an array of length `L` giving corresponding rewards
- `Q` is an `L x n` transition probability array

Here's how we could set up these objects for the preceding example

```
B, M, α, β = 10, 5, 0.5, 0.9
n = B + M + 1
m = M + 1

def u(c):
    return c**α

s_indices = []
a_indices = []
Q = []
R = []
b = 1.0 / (B + 1)

for s in range(n):
    for a in range(min(M, s) + 1):  # All feasible a at this s
        s_indices.append(s)
        a_indices.append(a)
        q = np.zeros(n)
        q[a:(a + B + 1)] = b        # b on these values, otherwise 0
        Q.append(q)
        R.append(u(s - a))

ddp = qe.markov.DiscreteDP(R, Q, β, s_indices, a_indices)
```

For larger problems, you might need to write this code more efficiently by vectorizing or using Numba.

4.5 Exercises

In the stochastic optimal growth lecture from our introductory lecture series, we solve a benchmark model that has an analytical solution.

The exercise is to replicate this solution using `DiscreteDP`.

4.6 Solutions

4.6.1 Setup

Details of the model can be found in the lecture on optimal growth.

We let $f(k) = k^\alpha$ with $\alpha = 0.65$, $u(c) = \log c$, and $\beta = 0.95$

```
α = 0.65
f = lambda k: k**α
u = np.log
β = 0.95
```

Here we want to solve a finite state version of the continuous state model above.

We discretize the state space into a grid of size `grid_size=500`, from 10^{-6} to `grid_max=2`

```
grid_max = 2
grid_size = 500
grid = np.linspace(1e-6, grid_max, grid_size)
```

We choose the action to be the amount of capital to save for the next period (the state is the capital stock at the beginning of the period).

Thus the state indices and the action indices are both `0`, ..., `grid_size-1`.

Action (indexed by) `a` is feasible at state (indexed by) `s` if and only if `grid[a] < f([grid[s])` (zero consumption is not allowed because of the log utility).

Thus the Bellman equation is:

$$v(k) = \max_{0<k'<f(k)} u(f(k) - k') + \beta v(k'),$$

where k' is the capital stock in the next period.

The transition probability array `Q` will be highly sparse (in fact it is degenerate as the model is deterministic), so we formulate the problem with state-action pairs, to represent `Q` in scipy sparse matrix format.

We first construct indices for state-action pairs:

```
# Consumption matrix, with nonpositive consumption included
C = f(grid).reshape(grid_size, 1) - grid.reshape(1, grid_size)

# State-action indices
s_indices, a_indices = np.where(C > 0)

# Number of state-action pairs
L = len(s_indices)

print(L)
print(s_indices)
print(a_indices)
```

```
118841
[  0   1   1 ... 499 499 499]
[  0   0   1 ... 389 390 391]
```

Reward vector `R` (of length `L`):

```
R = u(C[s_indices, a_indices])
```

(Degenerate) transition probability matrix `Q` (of shape `(L, grid_size)`), where we choose the scipy.sparse.lil_matrix format, while any format will do (internally it will be converted to the csr format):

```
Q = sparse.lil_matrix((L, grid_size))
Q[np.arange(L), a_indices] = 1
```

(If you are familiar with the data structure of scipy.sparse.csr_matrix, the following is the most efficient way to create the `Q` matrix in the current case)

```
# data = np.ones(L)
# indptr = np.arange(L+1)
# Q = sparse.csr_matrix((data, a_indices, indptr), shape=(L, grid_size))
```

Discrete growth model:

```
ddp = DiscreteDP(R, Q, β, s_indices, a_indices)
```

Notes

Here we intensively vectorized the operations on arrays to simplify the code.

As noted, however, vectorization is memory consumptive, and it can be prohibitively so for grids with large size.

4.6.2 Solving the Model

Solve the dynamic optimization problem:

```
res = ddp.solve(method='policy_iteration')
v, σ, num_iter = res.v, res.sigma, res.num_iter
num_iter
```

```
10
```

Note that `sigma` contains the *indices* of the optimal *capital stocks* to save for the next period. The following translates `sigma` to the corresponding consumption vector.

```
# Optimal consumption in the discrete version
c = f(grid) - grid[σ]

# Exact solution of the continuous version
ab = α * β
c1 = (np.log(1 - ab) + np.log(ab) * ab / (1 - ab)) / (1 - β)
c2 = α / (1 - ab)

def v_star(k):
    return c1 + c2 * np.log(k)

def c_star(k):
    return (1 - ab) * k**α
```

Let us compare the solution of the discrete model with that of the original continuous model

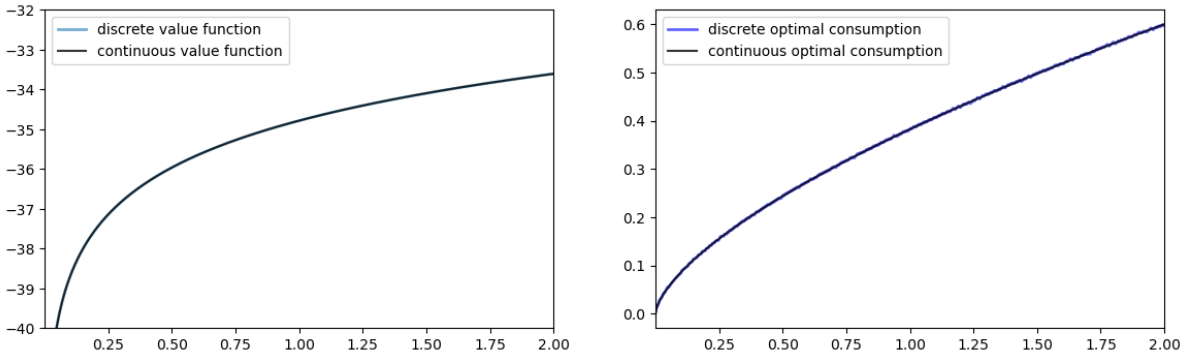
```
fig, ax = plt.subplots(1, 2, figsize=(14, 4))
ax[0].set_ylim(-40, -32)
ax[0].set_xlim(grid[0], grid[-1])
ax[1].set_xlim(grid[0], grid[-1])

lb0 = 'discrete value function'
ax[0].plot(grid, v, lw=2, alpha=0.6, label=lb0)

lb0 = 'continuous value function'
ax[0].plot(grid, v_star(grid), 'k-', lw=1.5, alpha=0.8, label=lb0)
ax[0].legend(loc='upper left')

lb1 = 'discrete optimal consumption'
ax[1].plot(grid, c, 'b-', lw=2, alpha=0.6, label=lb1)

lb1 = 'continuous optimal consumption'
ax[1].plot(grid, c_star(grid), 'k-', lw=1.5, alpha=0.8, label=lb1)
ax[1].legend(loc='upper left')
plt.show()
```



The outcomes appear very close to those of the continuous version.

Except for the "boundary" point, the value functions are very close:

```
np.abs(v - v_star(grid)).max()
```

```
121.49819147053378
```

```
np.abs(v - v_star(grid))[1:].max()
```

```
0.012681735127500815
```

The optimal consumption functions are close as well:

```
np.abs(c - c_star(grid)).max()
```

```
0.003826523100010082
```

In fact, the optimal consumption obtained in the discrete version is not really monotone, but the decrements are quite small:

```
diff = np.diff(c)
(diff >= 0).all()
```

```
False
```

```
dec_ind = np.where(diff < 0)[0]
len(dec_ind)
```

```
174
```

```
np.abs(diff[dec_ind]).max()
```

```
0.001961853339766839
```

The value function is monotone:

```
(np.diff(v) > 0).all()
```

```
True
```

4.6.3 Comparison of the Solution Methods

Let us solve the problem with the other two methods.

Value Iteration

```
ddp.epsilon = 1e-4
ddp.max_iter = 500
res1 = ddp.solve(method='value_iteration')
res1.num_iter
```

```
294
```

```
np.array_equal(σ, res1.sigma)
```

```
True
```

Modified Policy Iteration

```
res2 = ddp.solve(method='modified_policy_iteration')
res2.num_iter
```

```
16
```

```
np.array_equal(σ, res2.sigma)
```

```
True
```

Speed Comparison

```
%timeit ddp.solve(method='value_iteration')
%timeit ddp.solve(method='policy_iteration')
%timeit ddp.solve(method='modified_policy_iteration')
```

```
90.5 ms ± 586 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
9.32 ms ± 13.5 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
11.1 ms ± 40.8 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

As is often the case, policy iteration and modified policy iteration are much faster than value iteration.

4.6.4 Replication of the Figures

Using `DiscreteDP` we replicate the figures shown in the lecture.

Convergence of Value Iteration

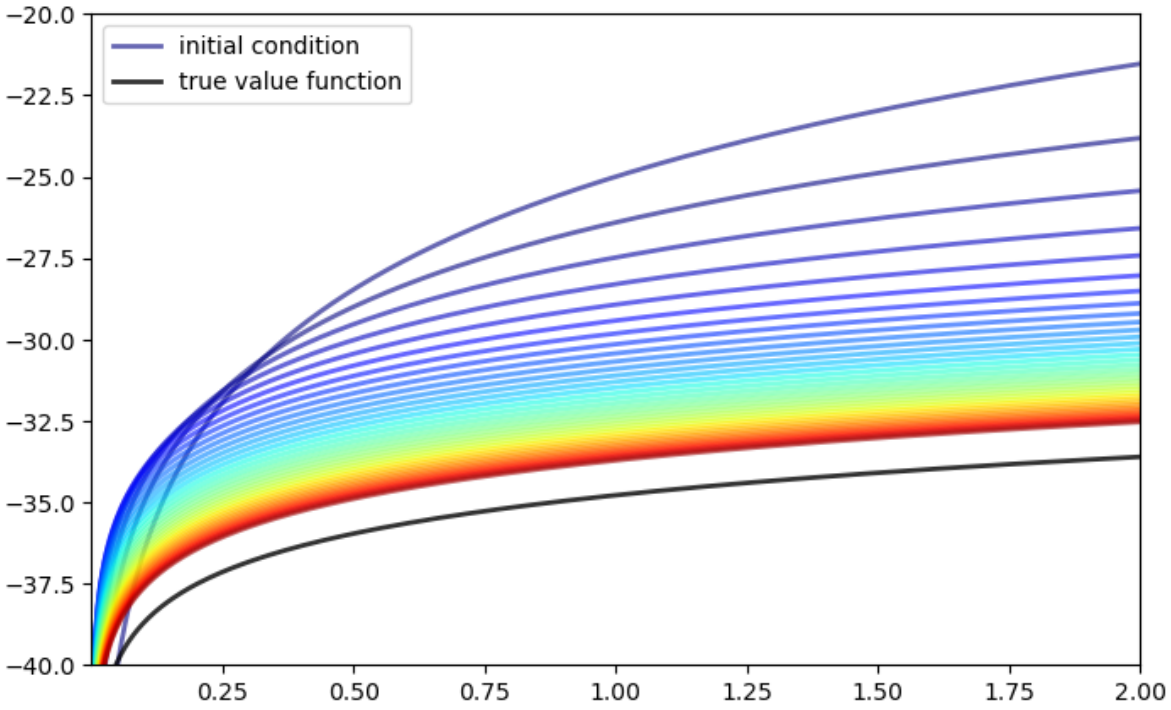
Let us first visualize the convergence of the value iteration algorithm as in the lecture, where we use `ddp.bellman_operator` implemented as a method of `DiscreteDP`

```
w = 5 * np.log(grid) - 25  # Initial condition
n = 35
fig, ax = plt.subplots(figsize=(8,5))
ax.set_ylim(-40, -20)
ax.set_xlim(np.min(grid), np.max(grid))
lb = 'initial condition'
ax.plot(grid, w, color=plt.cm.jet(0), lw=2, alpha=0.6, label=lb)
for i in range(n):
    w = ddp.bellman_operator(w)
    ax.plot(grid, w, color=plt.cm.jet(i / n), lw=2, alpha=0.6)
lb = 'true value function'
ax.plot(grid, v_star(grid), 'k-', lw=2, alpha=0.8, label=lb)
ax.legend(loc='upper left')
```

(continues on next page)

(continued from previous page)

```
plt.show()
```



We next plot the consumption policies along with the value iteration

```
w = 5 * u(grid) - 25           # Initial condition

fig, ax = plt.subplots(3, 1, figsize=(8, 10))
true_c = c_star(grid)

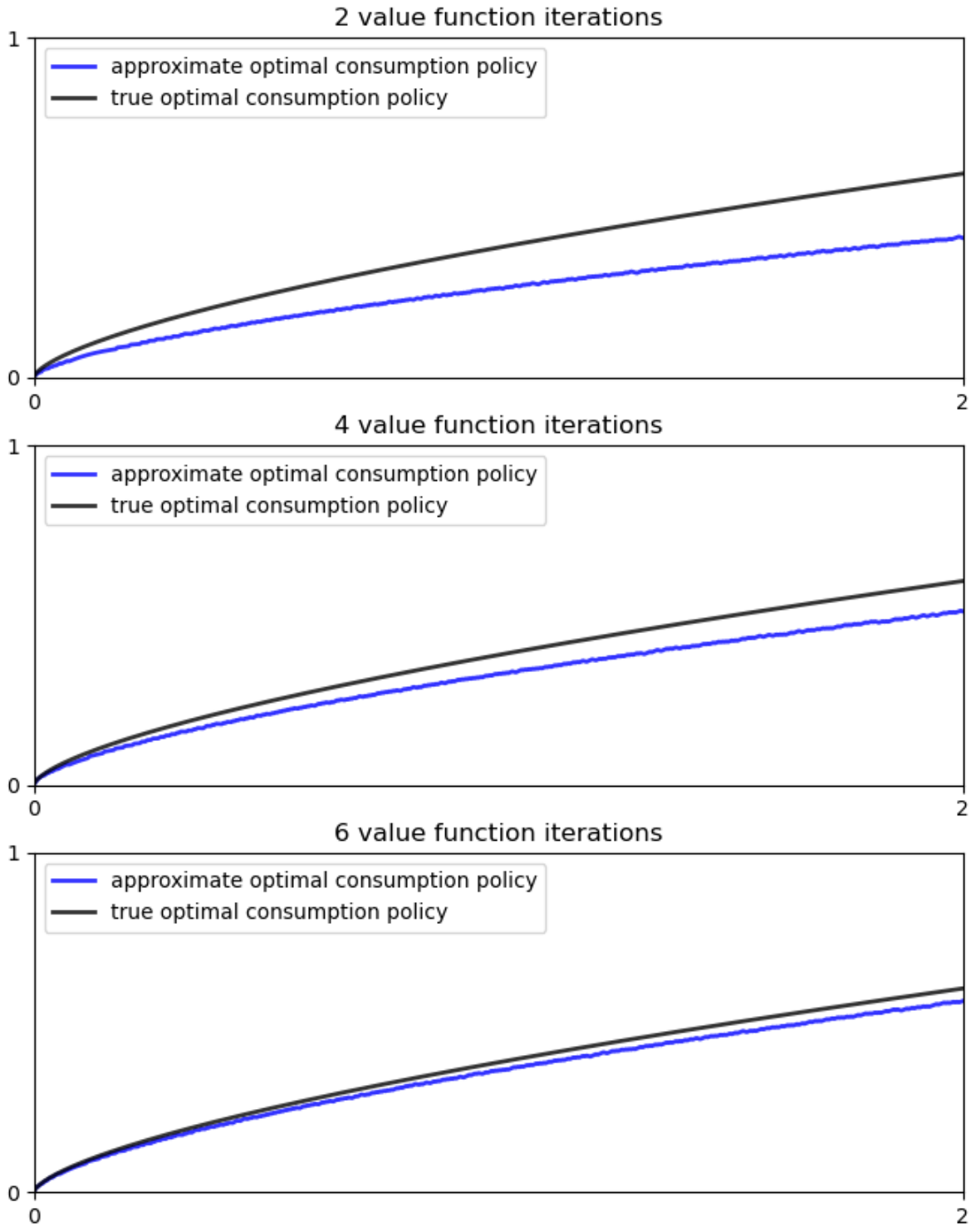
for i, n in enumerate((2, 4, 6)):
    ax[i].set_ylim(0, 1)
    ax[i].set_xlim(0, 2)
    ax[i].set_yticks((0, 1))
    ax[i].set_xticks((0, 2))

    w = 5 * u(grid) - 25       # Initial condition
    compute_fixed_point(ddp.bellman_operator, w, max_iter=n, print_skip=1)
    σ = ddp.compute_greedy(w)  # Policy indices
    c_policy = f(grid) - grid[σ]

    ax[i].plot(grid, c_policy, 'b-', lw=2, alpha=0.8,
               label='approximate optimal consumption policy')
    ax[i].plot(grid, true_c, 'k-', lw=2, alpha=0.8,
               label='true optimal consumption policy')
    ax[i].legend(loc='upper left')
    ax[i].set_title(f'{n} value function iterations')
plt.show()
```

```
Iteration    Distance       Elapsed (seconds)
--------------------------------------------
1            5.518e+00      6.368e-04
2            4.070e+00      1.045e-03
Iteration    Distance       Elapsed (seconds)
--------------------------------------------
1            5.518e+00      4.075e-04
2            4.070e+00      7.656e-04
3            3.866e+00      1.133e-03
4            3.673e+00      1.494e-03
Iteration    Distance       Elapsed (seconds)
--------------------------------------------
1            5.518e+00      3.934e-04
2            4.070e+00      7.384e-04
3            3.866e+00      1.101e-03
4            3.673e+00      1.435e-03
5            3.489e+00      1.773e-03
6            3.315e+00      2.129e-03
```

```
/home/runner/miniconda3/envs/quantecon/lib/python3.12/site-packages/quantecon/_
 ↪compute_fp.py:152: RuntimeWarning: max_iter attained before convergence in␣
 ↪compute_fixed_point
  warnings.warn(_non_convergence_msg, RuntimeWarning)
```


2 value function iterations
approximate optimal consumption policy
true optimal consumption policy
1
0
0
2
4 value function iterations
approximate optimal consumption policy
true optimal consumption policy
1
0
0
2
6 value function iterations
approximate optimal consumption policy
true optimal consumption policy
1
0
0
2

Dynamics of the Capital Stock

Finally, let us work on Exercise 2, where we plot the trajectories of the capital stock for three different discount factors, 0.9, 0.94, and 0.98, with initial condition $k_0 = 0.1$.

```
discount_factors = (0.9, 0.94, 0.98)
k_init = 0.1

# Search for the index corresponding to k_init
k_init_ind = np.searchsorted(grid, k_init)

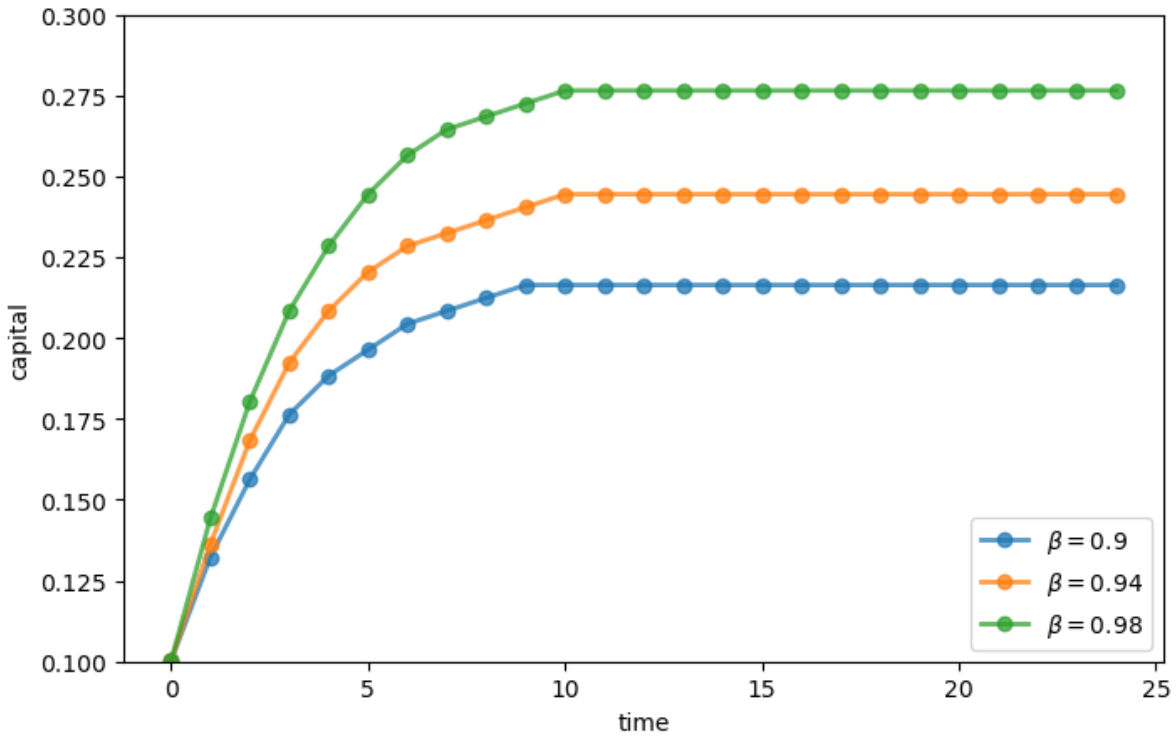
sample_size = 25

fig, ax = plt.subplots(figsize=(8,5))
ax.set_xlabel("time")
ax.set_ylabel("capital")
ax.set_ylim(0.10, 0.30)

# Create a new instance, not to modify the one used above
ddp0 = DiscreteDP(R, Q, β, s_indices, a_indices)

for beta in discount_factors:
    ddp0.beta = beta
    res0 = ddp0.solve()
    k_path_ind = res0.mc.simulate(init=k_init_ind, ts_length=sample_size)
    k_path = grid[k_path_ind]
    ax.plot(k_path, 'o-', lw=2, alpha=0.75, label=fr'$\beta = {beta}$')

ax.legend(loc='lower right')
plt.show()
```



4.7 Appendix: Algorithms

This appendix covers the details of the solution algorithms implemented for `DiscreteDP`.

We will make use of the following notions of approximate optimality:

- For $\varepsilon > 0$, v is called an ε-approximation of v^* if $\|v - v^*\| < \varepsilon$.
- A policy $\sigma \in \Sigma$ is called ε-optimal if v_σ is an ε-approximation of v^*.

4.7.1 Value Iteration

The `DiscreteDP` value iteration method implements value function iteration as follows

1. Choose any $v^0 \in \mathbb{R}^n$, and specify $\varepsilon > 0$; set $i = 0$.
2. Compute $v^{i+1} = Tv^i$.
3. If $\|v^{i+1} - v^i\| < [(1-\beta)/(2\beta)]\varepsilon$, then go to step 4; otherwise, set $i = i + 1$ and go to step 2.
4. Compute a v^{i+1}-greedy policy σ, and return v^{i+1} and σ.

Given $\varepsilon > 0$, the value iteration algorithm

- terminates in a finite number of iterations
- returns an $\varepsilon/2$-approximation of the optimal value function and an ε-optimal policy function (unless `iter_max` is reached)

(While not explicit, in the actual implementation each algorithm is terminated if the number of iterations reaches `iter_max`)

4.7.2 Policy Iteration

The `DiscreteDP` policy iteration method runs as follows

1. Choose any $v^0 \in \mathbb{R}^n$ and compute a v^0-greedy policy σ^0; set $i = 0$.
2. Compute the value v_{σ^i} by solving the equation $v = T_{\sigma^i} v$.
3. Compute a v_{σ^i}-greedy policy σ^{i+1}; let $\sigma^{i+1} = \sigma^i$ if possible.
4. If $\sigma^{i+1} = \sigma^i$, then return v_{σ^i} and σ^{i+1}; otherwise, set $i = i + 1$ and go to step 2.

The policy iteration algorithm terminates in a finite number of iterations.

It returns an optimal value function and an optimal policy function (unless `iter_max` is reached).

4.7.3 Modified Policy Iteration

The `DiscreteDP` modified policy iteration method runs as follows:

1. Choose any $v^0 \in \mathbb{R}^n$, and specify $\varepsilon > 0$ and $k \geq 0$; set $i = 0$.
2. Compute a v^i-greedy policy σ^{i+1}; let $\sigma^{i+1} = \sigma^i$ if possible (for $i \geq 1$).
3. Compute $u = Tv^i$ $(= T_{\sigma^{i+1}} v^i)$. If $\text{span}(u - v^i) < [(1-\beta)/\beta]\varepsilon$, then go to step 5; otherwise go to step 4.
    - Span is defined by $\text{span}(z) = \max(z) - \min(z)$.
4. Compute $v^{i+1} = (T_{\sigma^{i+1}})^k u$ $(= (T_{\sigma^{i+1}})^{k+1} v^i)$; set $i = i + 1$ and go to step 2.

5. Return $v = u + [\beta/(1-\beta)][(\min(u - v^i) + \max(u - v^i))/2]\mathbf{1}$ and σ_{i+1}.

Given $\varepsilon > 0$, provided that v^0 is such that $Tv^0 \geq v^0$, the modified policy iteration algorithm terminates in a finite number of iterations.

It returns an $\varepsilon/2$-approximation of the optimal value function and an ε-optimal policy function (unless `iter_max` is reached).

See also the documentation for `DiscreteDP`.

Part II

LQ Control

CHAPTER

FIVE

INFORMATION AND CONSUMPTION SMOOTHING

In addition to what's in Anaconda, this lecture employs the following libraries:

```
!pip install --upgrade quantecon
```

5.1 Overview

In the linear-quadratic permanent income of consumption smoothing model described in this quantecon lecture, a scalar parameter $\beta \in (0, 1)$ plays two roles:

- it is a **discount factor** that the consumer applies to future utilities from consumption
- it is the reciprocal of the gross **interest rate** on risk-free one-period loans

That β plays these two roles is essential in delivering the outcome that, **regardless** of the stochastic process that describes his non-financial income, the consumer chooses to make consumption follow a random walk (see [Hall, 1978]).

In this lecture, we assign a third role to β:

- it describes a **first-order moving average** process for the growth in non-financial income

5.1.1 Same non-financial incomes, different information

We study two consumers who have exactly the same nonfinancial income process and who both conform to the linear-quadratic permanent income of consumption smoothing model described here.

The two consumers have different information about their future nonfinancial incomes.

A better informed consumer each period receives **news** in the form of a shock that simultaneously affects both **today's** nonfinancial income and the present value of **future** nonfinancial incomes in a particular way.

A less informed consumer each period receives a shock that equals the part of today's nonfinancial income that could not be forecast from past values of nonfinancial income.

Even though they receive exactly the same nonfinancial incomes each period, our two consumers behave differently because they have different information about their future nonfinancial incomes.

The second consumer receives less information about future nonfinancial incomes in a sense that we shall make precise.

This difference in their information sets manifests itself in their responding differently to what they regard as time t **information shocks**.

Thus, although at each date they receive exactly the same histories of nonfinancial income, our two consumers receive different **shocks** or **news** about their **future** nonfinancial incomes.

We use the different behaviors of our consumers as a way to learn about

- operating characteristics of a linear-quadratic permanent income model
- how the Kalman filter introduced in this lecture and/or another representation of the theory of optimal forecasting introduced in *this lecture* embody lessons that can be applied to the **news** and **noise** literature
- ways of representing and computing optimal decision rules in the linear-quadratic permanent income model
- a **Ricardian equivalence** outcome that describes effects on optimal consumption of a tax cut at time t accompanied by a foreseen permanent increases in taxes that is just sufficient to cover the interest payments used to service the risk-free government bonds that are issued to finance the tax cut
- a simple application of alternative ways to factor a covariance generating function along lines described in *this lecture*

This lecture can be regarded as an introduction to **invertibility** issues that take center stage in the analysis of **fiscal foresight** by Eric Leeper, Todd Walker, and Susan Yang [Leeper *et al.*, 2013], as well as in chapter 4 of [Sargent *et al.*, 1991].

5.2 Two Representations of One Nonfinancial Income Process

We study consequences of endowing a consumer with one of two alternative representations for the change in the consumer's nonfinancial income $y_{t+1} - y_t$.

For both types of consumer, a parameter $\beta \in (0, 1)$ plays three roles.

It appears

- as a **discount factor** applied to future expected one-period utilities,
- as the **reciprocal of a gross interest rate** on one-period loans, and
- as a parameter in a first-order moving average that equals the increment in a consumer's non-financial income

The first representation, which we shall sometimes refer to as the **more informative representation**, is

$$y_{t+1} - y_t = \epsilon_{t+1} - \beta^{-1}\epsilon_t \tag{5.1}$$

where $\{\epsilon_t\}$ is an i.i.d. normally distributed scalar process with means of zero and contemporaneous variances σ_ϵ^2.

This representation of the process is used by a consumer who at time t knows both y_t and the shock ϵ_t and can use both of them to forecast future y_{t+j}'s.

As we'll see below, representation (5.1) has the peculiar property that a positive shock ϵ_{t+1} leaves the discounted present value of the consumer's financial income at time $t+1$ unaltered.

The second representation of the **same** $\{y_t\}$ process is

$$y_{t+1} - y_t = a_{t+1} - \beta a_t \tag{5.2}$$

where $\{a_t\}$ is another i.i.d. normally distributed scalar process, with means of zero and now variances $\sigma_a^2 > \sigma_\epsilon^2$.

The i.i.d. shock variances are related by

$$\sigma_a^2 = \beta^{-2}\sigma_\epsilon^2 > \sigma_\epsilon^2$$

so that the variance of the innovation exceeds the variance of the original shock by a multiplicative factor β^{-2}.

Representation (5.2) is the **innovations representation** of equation (5.1) associated with Kalman filtering theory.

To see how this works, note that equating representations (5.1) and (5.2) for $y_{t+1} - y_t$ implies $\epsilon_{t+1} - \beta^{-1}\epsilon_t = a_{t+1} - \beta a_t$, which in turn implies

$$a_{t+1} = \beta a_t + \epsilon_{t+1} - \beta^{-1}\epsilon_t.$$

Solving this difference equation backwards for a_{t+1} gives, after a few lines of algebra,

$$a_{t+1} = \epsilon_{t+1} + (\beta - \beta^{-1}) \sum_{j=0}^{\infty} \beta^j \epsilon_{t-j} \tag{5.3}$$

which we can also write as

$$a_{t+1} = \sum_{j=0}^{\infty} h_j \epsilon_{t+1-j} \equiv h(L)\epsilon_{t+1}$$

where L is the one-period lag operator, $h(L) = \sum_{j=0}^{\infty} h_j L^j$, I is the identity operator, and

$$h(L) = \frac{I - \beta^{-1}L}{I - \beta L}$$

Let $g_j \equiv E z_t z_{t-j}$ be the jth autocovariance of the $\{y_t - y_{t-1}\}$ process.

Using calculations in the *quantecon lecture*, where $z \in C$ is a complex variable, the **covariance generating function** $g(z) = \sum_{j=-\infty}^{\infty} g_j z^j$ of the $\{y_t - y_{t-1}\}$ process equals

$$g(z) = \sigma_\epsilon^2 h(z) h(z^{-1}) = \beta^{-2}\sigma_\epsilon^2 > \sigma_\epsilon^2,$$

which confirms that $\{a_t\}$ is a **serially uncorrelated** process with variance

$$\sigma_a^2 = \beta^{-1}\sigma_\epsilon^2.$$

To verify these claims, just notice that $g(z) = \beta^{-2}\sigma_\epsilon^2$ implies that

- $g_0 = \beta^{-2}\sigma_\epsilon^2$, and
- $g_j = 0$ for $j \neq 0$.

Alternatively, if you are uncomfortable with covariance generating functions, note that we can directly calculate σ_a^2 from formula (5.3) according to

$$\sigma_a^2 = \sigma_\epsilon^2 + [1 + (\beta - \beta^{-1})^2 \sum_{j=0}^{\infty} \beta^{2j}] = \beta^{-1}\sigma_\epsilon^2.$$

5.3 Application of Kalman filter

We can also use the the **Kalman filter** to obtain representation (5.2) from representation (5.1).

Thus, from equations associated with the **Kalman filter**, it can be verified that the steady-state Kalman gain $K = \beta^2$ and the steady state conditional covariance

$$\Sigma = E[(\epsilon_t - \hat{\epsilon}_t)^2 | y_{t-1}, y_{t-2}, ...] = (1 - \beta^2)\sigma_\epsilon^2$$

In a little more detail, let $z_t = y_t - y_{t-1}$ and form the state-space representation

$$\begin{aligned} \epsilon_{t+1} &= 0\epsilon_t + \epsilon_{t+1} \\ z_{t+1} &= -\beta^{-1}\epsilon_t + \epsilon_{t+1} \end{aligned}$$

and assume that $\sigma_\epsilon = 1$ for convenience

Let's compute the steady-state Kalman filter for this system.

Let K be the steady-state gain and a_{t+1} the one-step ahead innovation.

The steady-state innovations representation is

$$\begin{aligned}\hat{\epsilon}_{t+1} &= 0\hat{\epsilon}_t + Ka_{t+1} \\ z_{t+1} &= -\beta a_t + a_{t+1}\end{aligned}$$

By applying formulas for the steady-state Kalman filter, by hand it is possible to verify that $K = \beta^2, \sigma_a^2 = \beta^{-2}\sigma_\epsilon^2 = \beta^{-2}$, and $\Sigma = (1-\beta^2)\sigma_\epsilon^2$.

Alternatively, we can obtain these formulas via the classical filtering theory described in *this lecture*.

5.4 News Shocks and Less Informative Shocks

Representation (5.1) is cast in terms of a **news shock** ϵ_{t+1} that represents a shock to nonfinancial income coming from taxes, transfers, and other random sources of income changes known to a well-informed person who perhaps has all sorts of information about the income process.

Representation (5.2) for the **same** income process is driven by shocks a_t that contain less information than the news shock ϵ_t.

Representation (5.2) is called the **innovations** representation for the $\{y_t - y_{t-1}\}$ process.

It is cast in terms of what time series statisticians call the **innovation** or **fundamental** shock that emerges from applying the theory of optimally predicting nonfinancial income based solely on the information in **past** levels of growth in nonfinancial income.

Fundamental for the y_t process means that the shock a_t can be expressed as a square-summable linear combination of $y_t, y_{t-1}, \cdots$.

The shock ϵ_t is **not fundamental** because it has more information about the future of the $\{y_t - y_{t-1}\}$ process than is contained in a_t.

Representation (5.3) reveals the important fact that the **original shock** ϵ_t contains more information about future y's than is contained in the semi-infinite history $y^t = [y_t, y_{t-1}, ...]$.

Staring at representation (5.3) for a_{t+1} shows that it consists both of **new news** ϵ_{t+1} as well as a long moving average $(\beta - \beta^{-1})\sum_{j=0}^{\infty}\beta^j\epsilon_{t-j}$ of **old news**.

The **more information** representation (5.1) asserts that a shock ϵ_t results in an impulse response to nonfinancial income of ϵ_t times the sequence

$$1, 1-\beta^{-1}, 1-\beta^{-1}, ...$$

so that a shock that **increases** nonfinancial income y_t by ϵ_t at time t is followed by a change in future y of ϵ_t times $1-\beta^{-1} < 0$ in **all** subsequent periods.

Because $1-\beta^{-1} < 0$, this means that a positive shock of ϵ_t today raises income at time t by ϵ_t and then permanently **decreases all** future incomes by $(\beta^{-1}-1)\epsilon_t$.

This pattern precisely describes the following mental experiment:

- The consumer receives a government transfer of ϵ_t at time t.
- The government finances the transfer by issuing a one-period bond on which it pays a gross one-period risk-free interest rate equal to β^{-1}.

- In each future period, the government **rolls over** the one-period bond and so continues to borrow ϵ_t forever.
- The government imposes a lump-sum tax on the consumer in order to pay just the current interest on the original bond and its rolled over successors.
- Thus, in periods $t+1, t+2, ...$, the government levies a lump-sum tax on the consumer of $\beta^{-1} - 1$ that is just enough to pay the interest on the bond.

The **present value** of the impulse response or moving average coefficients equals $d_\epsilon(L) = \frac{0}{1-\beta} = 0$, a fact that we'll see again below.

Representation (5.2), i.e., the innovations representation, asserts that a shock a_t results in an impulse response to nonfinancial income of a_t times

$$1, 1-\beta, 1-\beta, ...$$

so that a shock that increases income y_t by a_t at time t can be expected to be followed by an **increase** in y_{t+j} of a_t times $1-\beta > 0$ in all future periods $j = 1, 2, ...$

The present value of the impulse response or moving average coefficients for representation (5.2) is $d_a(\beta) = \frac{1-\beta^2}{1-\beta} = (1+\beta)$, another fact that will be important below.

5.5 Representation of ϵ_t Shock in Terms of Future y_t

Notice that reprentation (5.1), namely, $y_{t+1} - y_t = -\beta^{-1}\epsilon_t + \epsilon_{t+1}$ implies the linear difference equation

$$\epsilon_t = \beta\epsilon_{t+1} - \beta(y_{t+1} - y_t).$$

Solving forward we obtain

$$\epsilon_t = \beta(y_t - (1-\beta)\sum_{j=0}^{\infty}\beta^j y_{t+j+1})$$

This equation shows that ϵ_t equals β times the one-step-backwards error in optimally **backcasting** y_t based on the semi-infinite **future** $y_+^t \equiv [y_{t+1}, y_{t+2}, ...]$ via the **optimal backcasting formula**

$$E[y_t | y_+^t] = (1-\beta)\sum_{j=0}^{\infty}\beta^j y_{t+j+1}$$

Thus, ϵ_t **exactly** reveals the gap between y_t and $E[y_t|y_+^t]$.

5.6 Representation in Terms of a_t Shocks

Next notice that representation (5.2), namely, $y_{t+1} - y_t = -\beta a_t + a_{t+1}$ implies the linear difference equation

$$a_{t+1} = \beta a_t + (y_{t+1} - y_t)$$

Solving this equation backward establishes that the one-step-prediction error a_{t+1} is

$$a_{t+1} = y_{t+1} - (1-\beta)\sum_{j=0}^{\infty}\beta^j y_{t-j}.$$

Here the information set is $y^t = [y_t, y_{t-1}, ...]$ and a one step-ahead optimal prediction is

$$E[y_{t+1}|y^t] = (1-\beta)\sum_{j=0}^{\infty}\beta^j y_{t-j}$$

5.7 Permanent Income Consumption-Smoothing Model

When we computed optimal consumption-saving policies for our two representations (5.1) and (5.2) by using formulas obtained with the difference equation approach described in quantecon lecture, we obtained:

for a consumer having the information assumed in the news representation (5.1):

$$
\begin{aligned}
c_{t+1} - c_t &= 0 \\
b_{t+1} - b_t &= -\beta^{-1}\epsilon_t
\end{aligned}
$$

for a consumer having the more limited information associated with the innovations representation (5.2):

$$
\begin{aligned}
c_{t+1} - c_t &= (1-\beta^2)a_{t+1} \\
b_{t+1} - b_t &= -\beta a_t
\end{aligned}
$$

These formulas agree with outcomes from Python programs below that deploy state-space representations and dynamic programming.

Evidently, although they receive exactly the same histories of nonfinancial incomethe two consumers behave differently.

The better informed consumer who has the information sets associated with representation (5.1) responds to each shock ϵ_{t+1} by leaving his consumption unaltered and **saving** all of ϵ_{t+1} in anticipation of the permanently increased taxes that he will bear in order to service the permanent interest payments on the risk-free bonds that the government has presumably issued to pay for the one-time addition ϵ_{t+1} to his time $t+1$ nonfinancial income.

The less well informed consumer who has information sets associated with representation (5.2) responds to a shock a_{t+1} by increasing his consumption by what he perceives to be the **permanent** part of the increase in consumption and by increasing his **saving** by what he perceives to be the temporary part.

The behavior of the better informed consumer sharply illustrates the behavior predicted in a classic Ricardian equivalence experiment.

5.8 State Space Representations

We now cast our representations (5.1) and (5.2), respectively, in terms of the following two state space systems:

$$
\begin{aligned}
\begin{bmatrix} y_{t+1} \\ \epsilon_{t+1} \end{bmatrix} &= \begin{bmatrix} 1 & -\beta^{-1} \\ 0 & 0 \end{bmatrix} \begin{bmatrix} y_t \\ \epsilon_t \end{bmatrix} + \begin{bmatrix} \sigma_\epsilon \\ \sigma_\epsilon \end{bmatrix} v_{t+1} \\
y_t &= \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} y_t \\ \epsilon_t \end{bmatrix}
\end{aligned}
\tag{5.4}
$$

and

$$
\begin{aligned}
\begin{bmatrix} y_{t+1} \\ a_{t+1} \end{bmatrix} &= \begin{bmatrix} 1 & -\beta \\ 0 & 0 \end{bmatrix} \begin{bmatrix} y_t \\ a_t \end{bmatrix} + \begin{bmatrix} \sigma_a \\ \sigma_a \end{bmatrix} u_{t+1} \\
y_t &= \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} y_t \\ a_t \end{bmatrix}
\end{aligned}
\tag{5.5}
$$

where $\{v_t\}$ and $\{u_t\}$ are both i.i.d. sequences of univariate standardized normal random variables.

These two alternative income processes are ready to be used in the framework presented in the section "Comparison with the Difference Equation Approach" in thid quantecon lecture.

All the code that we shall use below is presented in that lecture.

5.9 Computations

We shall use Python to form two state-space representations (5.4) and (5.5).

We set the following parameter values $\sigma_\epsilon = 1, \sigma_a = \beta^{-1}\sigma_\epsilon = \beta^{-1}$ where β is the **same** value as the discount factor in the household's problem in the LQ savings problem in the lecture.

For these two representations, we use the code in this lecture to

- compute optimal decision rules for c_t, b_t for the two types of consumers associated with our two representations of nonfinancial income
- use the value function objects P, d returned by the code to compute optimal values for the two representations when evaluated at the initial condition

$$x_0 = \begin{bmatrix} 10 \\ 0 \end{bmatrix}$$

for each representation.

- create instances of the LinearStateSpace class for the two representations of the $\{y_t\}$ process and use them to obtain impulse response functions of c_t and b_t to the respective shocks ϵ_t and a_t for the two representations.
- run simulations of $\{y_t, c_t, b_t\}$ of length T under both of the representations

We formulae the problem:

$$\min \sum_{t=0}^{\infty} \beta^t \left(c_t - \gamma\right)^2$$

subject to a sequence of constraints

$$c_t + b_t = \frac{1}{1+r} b_{t+1} + y_t, \quad t \geq 0$$

where y_t follows one of the representations defined above.

Define the control as $u_t \equiv c_t - \gamma$.

(For simplicity we can assume $\gamma = 0$ below because γ has no effect on the impulse response functions that interest us.)

The state transition equations under our two representations for the nonfinancial income process $\{y_t\}$ can be written as

$$\begin{bmatrix} y_{t+1} \\ \epsilon_{t+1} \\ b_{t+1} \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & -\beta^{-1} & 0 \\ 0 & 0 & 0 \\ -(1+r) & 0 & 1+r \end{bmatrix}}_{\equiv A_1} \begin{bmatrix} y_t \\ \epsilon_t \\ b_t \end{bmatrix} + \underbrace{\begin{bmatrix} 0 \\ 0 \\ 1+r \end{bmatrix}}_{\equiv B_1} \begin{bmatrix} c_t \end{bmatrix} + \underbrace{\begin{bmatrix} \sigma_\epsilon \\ \sigma_\epsilon \\ 0 \end{bmatrix}}_{\equiv C_1} \nu_{t+1},$$

and

$$\begin{bmatrix} y_{t+1} \\ a_{t+1} \\ b_{t+1} \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & -\beta & 0 \\ 0 & 0 & 0 \\ -(1+r) & 0 & 1+r \end{bmatrix}}_{\equiv A_2} \begin{bmatrix} y_t \\ a_t \\ b_t \end{bmatrix} + \underbrace{\begin{bmatrix} 0 \\ 0 \\ 1+r \end{bmatrix}}_{\equiv B_2} \begin{bmatrix} c_t \end{bmatrix} + \underbrace{\begin{bmatrix} \sigma_a \\ \sigma_a \\ 0 \end{bmatrix}}_{\equiv C_2} u_{t+1}.$$

As usual, we start by importing packages.

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

```
# Set parameters
β, σϵ = 0.95, 1
σa = σϵ / β

R = 1 / β

# Payoff matrices are the same for two representations
RLQ = np.array([[0, 0, 0],
                [0, 0, 0],
                [0, 0, 1e-12]]) # put penalty on debt
QLQ = np.array([1.])
```

```
# More informative representation state transition matrices
ALQ1 = np.array([[1, -R, 0],
                 [0, 0, 0],
                 [-R, 0, R]])
BLQ1 = np.array([[0, 0, R]]).T
CLQ1 = np.array([[σϵ, σϵ, 0]]).T

# Construct and solve the LQ problem
LQ1 = qe.LQ(QLQ, RLQ, ALQ1, BLQ1, C=CLQ1, beta=β)
P1, F1, d1 = LQ1.stationary_values()
```

```
# The optimal decision rule for c
-F1
```

```
array([[ 1.  , -1.  , -0.05]])
```

Evidently, optimal consumption and debt decision rules for the consumer having news representation (5.1) are

$$\begin{aligned} c_t^* &= y_t - \epsilon_t - (1-\beta)\, b_t, \\ b_{t+1}^* &= \beta^{-1} c_t^* + \beta^{-1} b_t - \beta^{-1} y_t \\ &= \beta^{-1} y_t - \beta^{-1}\epsilon_t - (\beta^{-1} - 1)\, b_t + \beta^{-1} b_t - \beta^{-1} y_t \\ &= b_t - \beta^{-1}\epsilon_t. \end{aligned}$$

```
# Innovations representation
ALQ2 = np.array([[1, -β, 0],
                 [0,  0, 0],
                 [-R, 0, R]])
BLQ2 = np.array([[0, 0, R]]).T
CLQ2 = np.array([[σa, σa, 0]]).T

LQ2 = qe.LQ(QLQ, RLQ, ALQ2, BLQ2, C=CLQ2, beta=β)
P2, F2, d2 = LQ2.stationary_values()
```

```
-F2
```

```
array([[ 1.    , -0.9025, -0.05  ]])
```

For a consumer having access only to the information associated with the innovations representation (5.2), the optimal

decision rules are

$$\begin{aligned} c_t^* &= y_t - \beta^2 a_t - (1-\beta)\, b_t, \\ b_{t+1}^* &= \beta^{-1} c_t^* + \beta^{-1} b_t - \beta^{-1} y_t \\ &= \beta^{-1} y_t - \beta a_t - (\beta^{-1} - 1)\, b_t + \beta^{-1} b_t - \beta^{-1} y_t \\ &= b_t - \beta a_t. \end{aligned}$$

Now we construct two Linear State Space models that emerge from using optimal policies of the form $u_t = -Fx_t$.

Take the more informative original representation (5.1) as an example:

$$\begin{bmatrix} y_{t+1} \\ \epsilon_{t+1} \\ b_{t+1} \end{bmatrix} = (A_1 - B_1 F_1) \begin{bmatrix} y_t \\ \epsilon_t \\ b_t \end{bmatrix} + C_1 \nu_{t+1}$$

$$\begin{bmatrix} c_t \\ b_t \end{bmatrix} = \begin{bmatrix} -F_1 \\ S_b \end{bmatrix} \begin{bmatrix} y_t \\ \epsilon_t \\ b_t \end{bmatrix}$$

To have the Linear State Space model be of an innovations representation form (5.2), we can simply replace the corresponding matrices.

```
# Construct two Linear State Space models
Sb = np.array([0, 0, 1])

ABF1 = ALQ1 - BLQ1 @ F1
G1 = np.vstack([-F1, Sb])
LSS1 = qe.LinearStateSpace(ABF1, CLQ1, G1)

ABF2 = ALQ2 - BLQ2 @ F2
G2 = np.vstack([-F2, Sb])
LSS2 = qe.LinearStateSpace(ABF2, CLQ2, G2)
```

The following code computes impulse response functions of c_t and b_t.

```
J = 5 # Number of coefficients that we want

x_res1, y_res1 = LSS1.impulse_response(j=J)
b_res1 = np.array([x_res1[i][2, 0] for i in range(J)])
c_res1 = np.array([y_res1[i][0, 0] for i in range(J)])

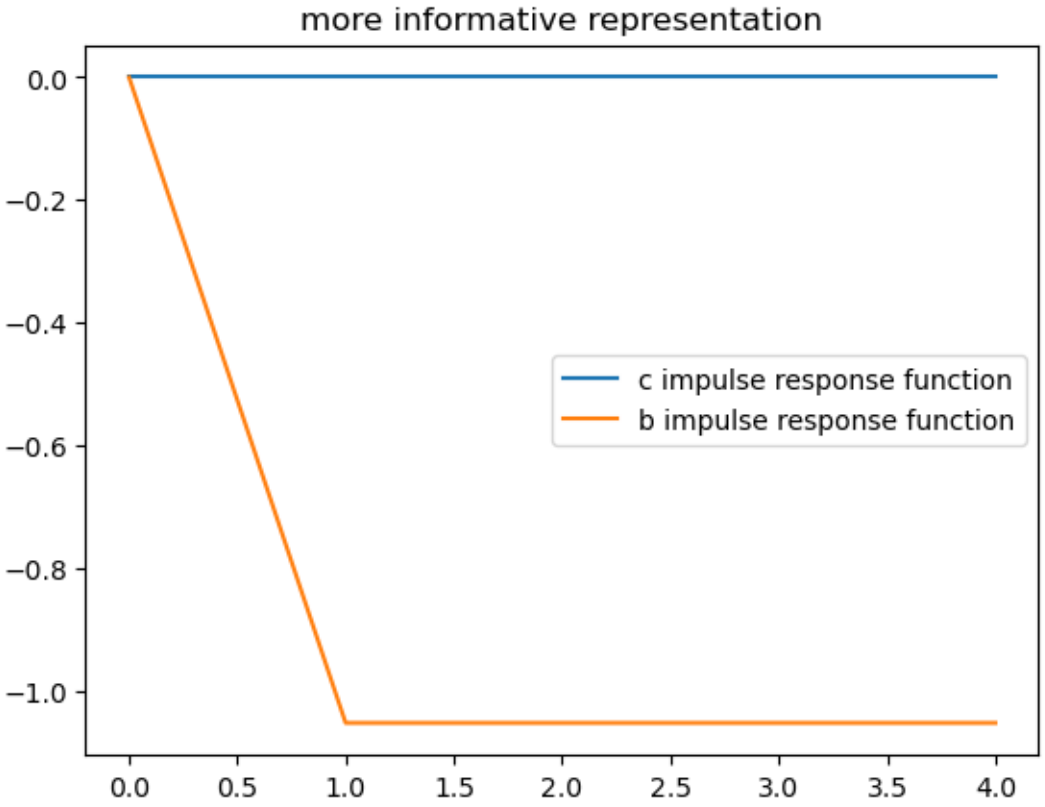
x_res2, y_res2 = LSS2.impulse_response(j=J)
b_res2 = np.array([x_res2[i][2, 0] for i in range(J)])
c_res2 = np.array([y_res2[i][0, 0] for i in range(J)])
```

```
c_res1 / σϵ, b_res1 / σϵ
```

```
(array([1.99998906e-11, 1.89473923e-11, 1.78947621e-11, 1.68421319e-11,
        1.57895017e-11]),
 array([ 0.        , -1.05263158, -1.05263158, -1.05263158, -1.05263158]))
```

```
plt.title("more informative representation")
plt.plot(range(J), c_res1 / σϵ, label="c impulse response function")
plt.plot(range(J), b_res1 / σϵ, label="b impulse response function")
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7fcb4e6d7d40>
```



The above two impulse response functions show that when the consumer has the information assumed in the more informative representation (5.1), his response to receiving a positive shock of ϵ_t is to leave his consumption unchanged and to save the entire amount of his extra income and then forever roll over the extra bonds that he holds.

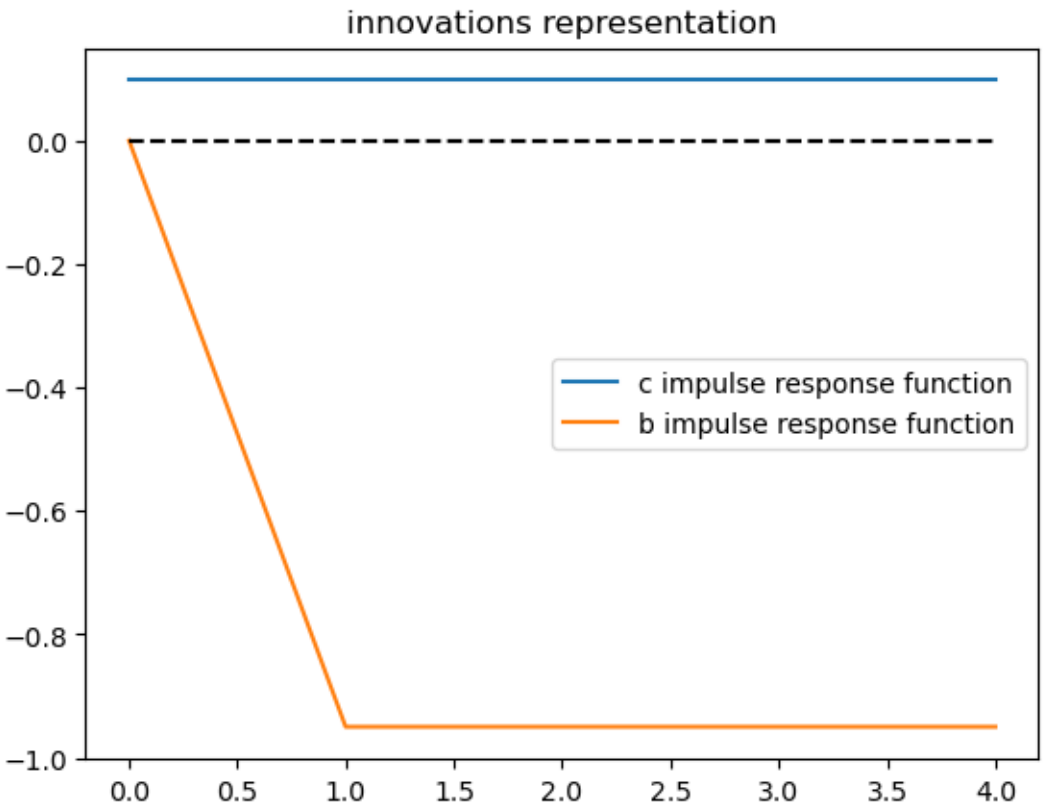
To see this notice, that starting from next period on, his debt permanently **decreases** by β^{-1}

```
c_res2 / σa, b_res2 / σa
```

```
(array([0.0975, 0.0975, 0.0975, 0.0975, 0.0975]),
 array([ 0.  , -0.95, -0.95, -0.95, -0.95]))
```

```
plt.title("innovations representation")
plt.plot(range(J), c_res2 / σa, label="c impulse response function")
plt.plot(range(J), b_res2 / σa, label="b impulse response function")
plt.plot([0, J-1], [0, 0], '--', color='k')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7fcb973d8560>
```



The above impulse responses show that when the consumer has only the information that is assumed to be available under the innovations representation (5.2) for $\{y_t - y_{t-1}\}$, he responds to a positive a_t by **permanently** increasing his consumption.

He accomplishes this by consuming a fraction $(1-\beta^2)$ of the increment a_t to his nonfinancial income and saving the rest, thereby lowering b_{t+1} in order to finance the permanent increment in his consumption.

The preceding computations confirm what we had derived earlier using paper and pencil.

Now let's simulate some paths of consumption and debt for our two types of consumers while always presenting both types with the same $\{y_t\}$ path.

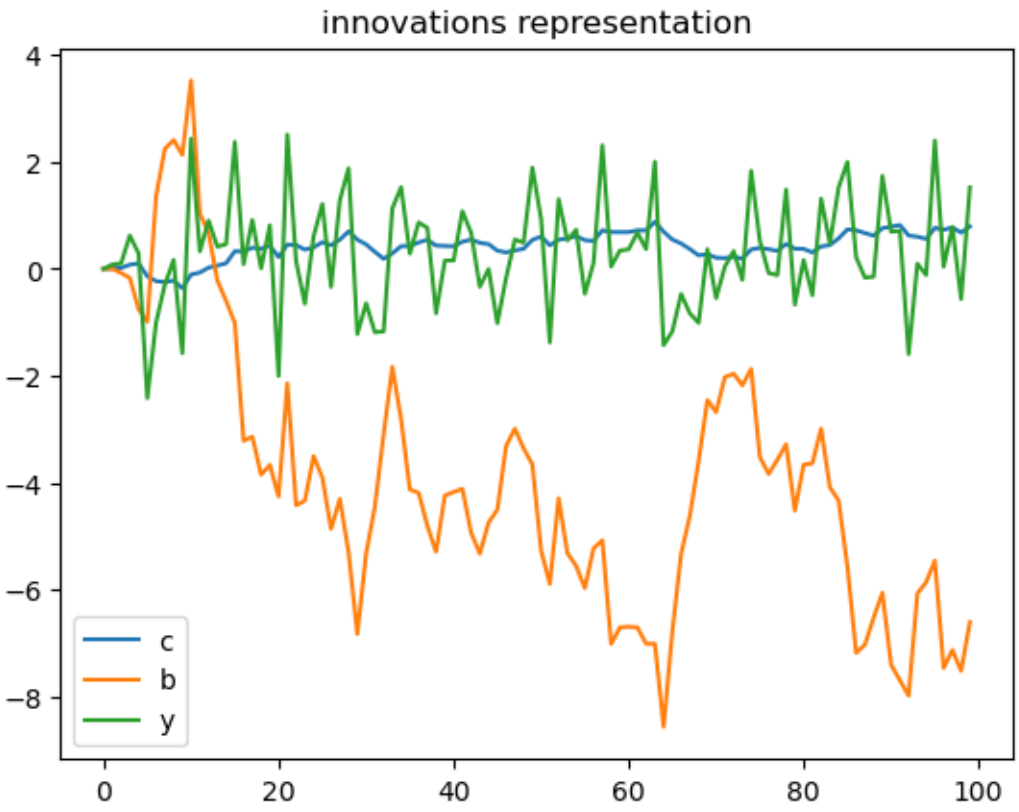
```
# Set time length for simulation
T = 100
```

```
x1, y1 = LSS1.simulate(ts_length=T)
plt.plot(range(T), y1[0, :], label="c")
plt.plot(range(T), x1[2, :], label="b")
plt.plot(range(T), x1[0, :], label="y")
plt.title("more informative representation")
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7fcb4e6d6e40>
```

```
x2, y2 = LSS2.simulate(ts_length=T)
plt.plot(range(T), y2[0, :], label="c")
plt.plot(range(T), x2[2, :], label="b")
plt.plot(range(T), x2[0, :], label="y")
plt.title("innovations representation")
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7fcb4dfd96a0>
```



5.10 Simulating Income Process and Two Associated Shock Processes

We now form a **single** $\{y_t\}_{t=0}^T$ realization that we will use to simulate decisions associated with our two types of consumer.

We accomplish this in the following steps.

1. We form a $\{y_t, \epsilon_t\}$ realization by drawing a long simulation of $\{\epsilon_t\}_{t=0}^T$, where T is a big integer, $\epsilon_t = \sigma_\epsilon v_t$, v_t is a standard normal scalar, $y_0 = 100$, and

$$y_{t+1} - y_t = -\beta^{-1}\epsilon_t + \epsilon_{t+1}.$$

2. We take the $\{y_t\}$ realization generated in step 1 and form an innovation process $\{a_t\}$ from the formulas

$$\begin{aligned} a_0 &= 0 \\ a_t &= \sum_{j=0}^{t-1} \beta^j (y_{t-j} - y_{t-j-1}) + \beta^t a_0, \quad t \geq 1 \end{aligned}$$

3. We throw away the first S observations and form a sample $\{y_t, \epsilon_t, a_t\}_{S+1}^T$ as the realization that we'll use in the following steps.

4. We use the step 3 realization to **evaluate** and **simulate** the decision rules for c_t, b_t that Python has computed for us above.

The above steps implement the experiment of comparing decisions made by two consumers having **identical** incomes at each date but at each date having **different** information about their future incomes.

5.11 Calculating Innovations in Another Way

Here we use formula (5.3) above to compute a_{t+1} as a function of the history $\epsilon_{t+1}, \epsilon_t, \epsilon_{t-1}, \ldots$

Thus, we compute

$$\begin{aligned}
a_{t+1} &= \beta a_t + \epsilon_{t+1} - \beta^{-1}\epsilon_t \\
&= \beta\left(\beta a_{t-1} + \epsilon_t - \beta^{-1}\epsilon_{t-1}\right) + \epsilon_{t+1} - \beta^{-1}\epsilon_t \\
&= \beta^2 a_{t-1} + \beta\left(\epsilon_t - \beta^{-1}\epsilon_{t-1}\right) + \epsilon_{t+1} - \beta^{-1}\epsilon_t \\
&= \qquad\quad \vdots \qquad\quad \vdots \\
&= \beta^{t+1} a_0 + \sum_{j=0}^{t} \beta^j \left(\epsilon_{t+1-j} - \beta^{-1}\epsilon_{t-j}\right) \\
&= \beta^{t+1} a_0 + \epsilon_{t+1} + \left(\beta - \beta^{-1}\right) \sum_{j=0}^{t-1} \beta^j \epsilon_{t-j} - \beta^{t-1}\epsilon_0 .
\end{aligned}$$

We can verify that we recover the same $\{a_t\}$ sequence computed earlier.

5.12 Another Invertibility Issue

This *quantecon lecture* contains another example of a shock-invertibility issue that is endemic to the LQ permanent income or consumption smoothing model.

The technical issue discussed there is ultimately the source of the shock-invertibility issues discussed by Eric Leeper, Todd Walker, and Susan Yang [Leeper *et al.*, 2013] in their analysis of **fiscal foresight**.

CHAPTER

SIX

CONSUMPTION SMOOTHING WITH COMPLETE AND INCOMPLETE MARKETS

In addition to what's in Anaconda, this lecture uses the library:

```
!pip install --upgrade quantecon
```

6.1 Overview

This lecture describes two types of consumption-smoothing models.

- one is in the **complete markets** tradition of Kenneth Arrow
- the other is in the **incomplete markets** tradition of Hall [Hall, 1978]

Complete markets allow a consumer to buy and sell claims contingent on all possible states of the world.

Incomplete markets allow a consumer to buy and sell a limited set of securities, often only a single risk-free security.

Hall [Hall, 1978] worked in an incomplete markets tradition by assuming that the only asset that can be traded is a risk-free one-period bond.

Hall assumed an exogenous stochastic process of nonfinancial income and an exogenous and time-invariant gross interest rate on one-period risk-free debt that equals β^{-1}, where $\beta \in (0, 1)$ is also a consumer's intertemporal discount factor.

This is equivalent to saying that it costs β of time t consumption to buy one unit of consumption at time $t + 1$ for sure.

So β is the price of a one-period risk-free claim to consumption next period.

We preserve Hall's assumption about the interest rate when we describe an incomplete markets version of our model.

In addition, we extend Hall's assumption about the risk-free interest rate to an appropriate counterpart when we create another model in which there are markets in a complete array of one-period Arrow state-contingent securities.

We'll consider two closely related alternative assumptions about the consumer's exogenous nonfinancial income process:

- that it is generated by a finite N state Markov chain (setting $N = 2$ most of the time in this lecture)
- that it is described by a linear state space model with a continuous state vector in $\mathbb{R}^n$ driven by a Gaussian vector IID shock process

We'll spend most of this lecture studying the finite-state Markov specification, but will begin by studying the linear state space specification because it is so closely linked to earlier lectures.

Let's start with some imports:

```
import numpy as np
import quantecon as qe
import matplotlib.pyplot as plt
import scipy.linalg as la
```

6.1.1 Relationship to Other Lectures

This lecture can be viewed as a followup to Optimal Savings II: LQ Techniques

This lecture is also a prologomenon to a lecture on tax-smoothing *Tax Smoothing with Complete and Incomplete Markets*

6.2 Background

Outcomes in consumption-smoothing models emerge from two sources:

- a consumer who wants to maximize an intertemporal objective function that expresses its preference for paths of consumption that are *smooth* in the sense of varying as little as possible both across time and across realized Markov states
- opportunities that allow the consumer to transform an erratic nonfinancial income process into a smoother consumption process by buying and selling one or more financial securities

In the **complete markets version**, each period the consumer can buy or sell a complete set of one-period ahead state-contingent securities whose payoffs depend on next period's realization of the Markov state.

- In the two-state Markov chain case, two such securities are traded each period.
- In an N state Markov state version, N such securities are traded each period.
- In a continuous state Markov state version, a continuum of such securities is traded each period.

These state-contingent securities are commonly called Arrow securities, after Kenneth Arrow.

In the **incomplete markets version**, the consumer can buy and sell only one security each period, a risk-free one-period bond with gross one-period return β^{-1}.

6.3 Linear State Space Version of Complete Markets Model

We'll study a complete markets model adapted to a setting with a continuous Markov state like that in the first lecture on the permanent income model.

In that model

- a consumer can trade only a single risk-free one-period bond bearing gross one-period risk-free interest rate equal to β^{-1}.
- a consumer's exogenous nonfinancial income is governed by a linear state space model driven by Gaussian shocks, the kind of model studied in an earlier lecture about linear state space models.

Let's write down a complete markets counterpart of that model.

Suppose that nonfinancial income is governed by the state space system

$$
\begin{aligned}
x_{t+1} &= Ax_t + Cw_{t+1} \\
y_t &= S_y x_t
\end{aligned}
$$

where x_t is an $n \times 1$ vector and $w_{t+1} \sim N(0, I)$ is IID over time.

We want a natural counterpart of the Hall assumption that the one-period risk-free gross interest rate is β^{-1}.

We make the good guess that prices of one-period ahead Arrow securities are described by the **pricing kernel**

$$q_{t+1}(x_{t+1} \mid x_t) = \beta \phi(x_{t+1} \mid Ax_t, CC') \tag{6.1}$$

where $\phi(\cdot \mid \mu, \Sigma)$ is a multivariate Gaussian distribution with mean vector μ and covariance matrix Σ.

With the pricing kernel $q_{t+1}(x_{t+1} \mid x_t)$ in hand, we can price claims to consumption at time $t+1$ consumption that pay off when $x_{t+1} \in S$ at time $t+1$:

$$\int_S q_{t+1}(x_{t+1} \mid x_t) dx_{t+1}$$

where S is a subset of $\mathbb{R}^n$.

The price $\int_S q_{t+1}(x_{t+1} \mid x_t) dx_{t+1}$ of such a claim depends on state x_t because the prices of the x_{t+1}-contingent securities depend on x_t through the pricing kernel $q(x_{t+1} \mid x_t)$.

Let $b(x_{t+1})$ be a vector of state-contingent debt due at $t+1$ as a function of the $t+1$ state x_{t+1}.

Using the pricing kernel assumed in (6.1), the value at t of $b(x_{t+1})$ is evidently

$$\beta \int b(x_{t+1}) \phi(x_{t+1} \mid Ax_t, CC') dx_{t+1} = \beta \mathbb{E}_t b_{t+1}$$

In our complete markets setting, the consumer faces a sequence of budget constraints

$$c_t + b_t = y_t + \beta \mathbb{E}_t b_{t+1}, \quad t \geq 0$$

Please note that

$$\beta E_t b_{t+1} = \beta \int \phi_{t+1}(x_{t+1} | Ax_t, CC') b_{t+1}(x_{t+1}) dx_{t+1}$$

or

$$\beta E_t b_{t+1} = \int q_{t+1}(x_{t+1} | x_t) b_{t+1}(x_{t+1}) dx_{t+1}$$

which verifies that $\beta E_t b_{t+1}$ is the **value** of time $t+1$ state-contingent claims on time $t+1$ consumption issued by the consumer at time t

We can solve the time t budget constraint forward to obtain

$$b_t = \mathbb{E}_t \sum_{j=0}^{\infty} \beta^j (y_{t+j} - c_{t+j})$$

The consumer cares about the expected value of

$$\sum_{t=0}^{\infty} \beta^t u(c_t), \quad 0 < \beta < 1$$

In the incomplete markets version of the model, we assumed that $u(c_t) = -(c_t - \gamma)^2$, so that the above utility functional became

$$-\sum_{t=0}^{\infty} \beta^t (c_t - \gamma)^2, \quad 0 < \beta < 1$$

But in the complete markets version, it is tractable to assume a more general utility function that satisfies $u' > 0$ and $u'' < 0$.

First-order conditions for the consumer's problem with complete markets and our assumption about Arrow securities prices are

$$u'(c_{t+1}) = u'(c_t) \quad \text{for all } t \geq 0$$

which implies $c_t = \bar{c}$ for some $\bar{c}$.

So it follows that

$$b_t = \mathbb{E}_t \sum_{j=0}^{\infty} \beta^j (y_{t+j} - \bar{c})$$

or

$$b_t = S_y (I - \beta A)^{-1} x_t - \frac{1}{1-\beta}\bar{c} \tag{6.2}$$

where $\bar{c}$ satisfies

$$\bar{b}_0 = S_y (I - \beta A)^{-1} x_0 - \frac{1}{1-\beta}\bar{c} \tag{6.3}$$

where $\bar{b}_0$ is an initial level of the consumer's debt due at time $t = 0$, specified as a parameter of the problem.

Thus, in the complete markets version of the consumption-smoothing model, $c_t = \bar{c}, \forall t \geq 0$ is determined by (6.3) and the consumer's debt is the fixed function of the state x_t described by (6.2).

Please recall that in the LQ permanent income model studied in permanent income model, the state is x_t, b_t, where b_t is a complicated function of past state vectors x_{t-j}.

Notice that in contrast to that incomplete markets model, at time t the state vector is x_t alone in our complete markets model.

Here's an example that shows how in this setting the availability of insurance against fluctuating nonfinancial income allows the consumer completely to smooth consumption across time and across states of the world

```
def complete_ss(β, b0, x0, A, C, S_y, T=12):
    """
    Computes the path of consumption and debt for the previously described
    complete markets model where exogenous income follows a linear
    state space
    """
    # Create a linear state space for simulation purposes
    # This adds "b" as a state to the linear state space system
    # so that setting the seed places shocks in same place for
    # both the complete and incomplete markets economy
    # Atilde = np.vstack([np.hstack([A, np.zeros((A.shape[0], 1))]),
    #                     np.zeros((1, A.shape[1] + 1))])
    # Ctilde = np.vstack([C, np.zeros((1, 1))])
    # S_ytilde = np.hstack([S_y, np.zeros((1, 1))])

    lss = qe.LinearStateSpace(A, C, S_y, mu_0=x0)

    # Add extra state to initial condition
    # x0 = np.hstack([x0, np.zeros(1)])

    # Compute the (I - β * A)^{-1}
```

(continues on next page)

(continued from previous page)

```
    rm = la.inv(np.eye(A.shape[0]) - β * A)

    # Constant level of consumption
    cbar = (1 - β) * (S_y @ rm @ x0 - b0)
    c_hist = np.full(T, cbar)

    # Debt
    x_hist, y_hist = lss.simulate(T)
    b_hist = np.squeeze(S_y @ rm @ x_hist - cbar / (1 - β))


    return c_hist, b_hist, np.squeeze(y_hist), x_hist


# Define parameters
N_simul = 80
α, ρ1, ρ2 = 10.0, 0.9, 0.0
σ = 1.0

A = np.array([[1., 0., 0.],
              [α,  ρ1, ρ2],
              [0., 1., 0.]])
C = np.array([[0.], [σ], [0.]])
S_y = np.array([[1,  1.0, 0.]])
β, b0 = 0.95, -10.0
x0 = np.array([1.0, α / (1 - ρ1), α / (1 - ρ1)])

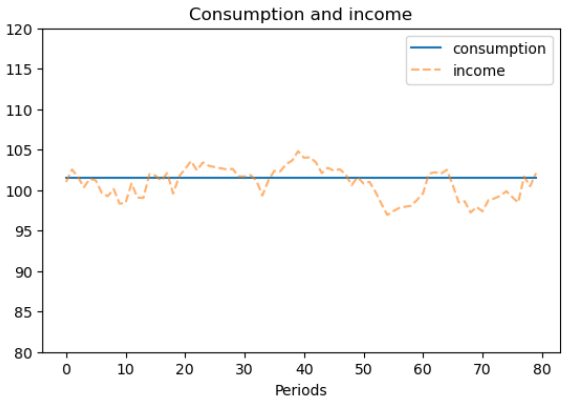
# Do simulation for complete markets
s = np.random.randint(0, 10000)
np.random.seed(s)  # Seeds get set the same for both economies
out = complete_ss(β, b0, x0, A, C, S_y, 80)
c_hist_com, b_hist_com, y_hist_com, x_hist_com = out

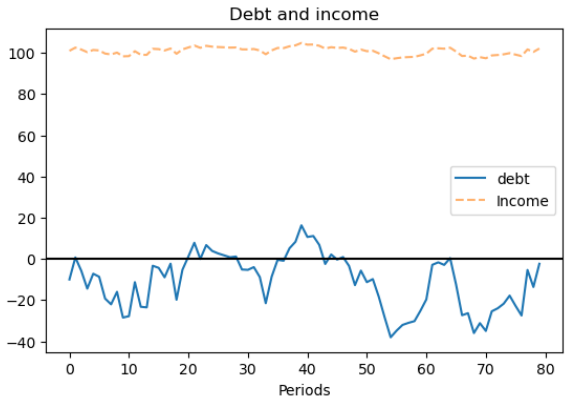
fig, ax = plt.subplots(1, 2, figsize=(14, 4))

# Consumption plots
ax[0].set_title('Consumption and income')
ax[0].plot(np.arange(N_simul), c_hist_com, label='consumption')
ax[0].plot(np.arange(N_simul), y_hist_com, label='income', alpha=.6, linestyle='--')
ax[0].legend()
ax[0].set_xlabel('Periods')
ax[0].set_ylim([80, 120])

# Debt plots
ax[1].set_title('Debt and income')
ax[1].plot(np.arange(N_simul), b_hist_com, label='debt')
ax[1].plot(np.arange(N_simul), y_hist_com, label='Income', alpha=.6, linestyle='--')
ax[1].legend()
ax[1].axhline(0, color='k')
ax[1].set_xlabel('Periods')

plt.show()
```





6.3.1 Interpretation of Graph

In the above graph, please note that:

- nonfinancial income fluctuates in a stationary manner.
- consumption is completely constant.
- the consumer's debt fluctuates in a stationary manner; in fact, in this case, because nonfinancial income is a first-order autoregressive process, the consumer's debt is an exact affine function (meaning linear plus a constant) of the consumer's nonfinancial income.

6.3.2 Incomplete Markets Version

The incomplete markets version of the model with nonfinancial income being governed by a linear state space system is described in permanent income model.

In that incomplete markerts setting, consumption follows a random walk and the consumer's debt follows a process with a unit root.

6.3.3 Finite State Markov Income Process

We now turn to a finite-state Markov version of the model in which the consumer's nonfinancial income is an exact function of a Markov state that takes one of N values.

We'll start with a setting in which in each version of our consumption-smoothing model, nonfinancial income is governed by a two-state Markov chain (it's easy to generalize this to an N state Markov chain).

In particular, the *state* $s_t \in \{1, 2\}$ follows a Markov chain with transition probability matrix

$$P_{ij} = \mathbb{P}\{s_{t+1} = j \,|\, s_t = i\}$$

where $\mathbb{P}$ means conditional probability

Nonfinancial income $\{y_t\}$ obeys

$$y_t = \begin{cases} \bar{y}_1 & \text{if } s_t = 1 \\ \bar{y}_2 & \text{if } s_t = 2 \end{cases}$$

A consumer wishes to maximize

$$\mathbb{E}\left[\sum_{t=0}^{\infty}\beta^t u(c_t)\right] \quad \text{where} \quad u(c_t)=-(c_t-\gamma)^2 \quad \text{and} \quad 0<\beta<1 \tag{6.4}$$

Here $\gamma > 0$ is a bliss level of consumption

6.3.4 Market Structure

Our complete and incomplete markets models differ in how thoroughly the market structure allows a consumer to transfer resources across time and Markov states, there being more transfer opportunities in the complete markets setting than in the incomplete markets setting.

Watch how these differences in opportunities affect

- how smooth consumption is across time and Markov states
- how the consumer chooses to make his levels of indebtedness behave over time and across Markov states

6.4 Model 1 (Complete Markets)

At each date $t \geq 0$, the consumer trades a full array of **one-period ahead Arrow securities**.

We assume that prices of these securities are exogenous to the consumer.

Exogenous means that they are unaffected by the consumer's decisions.

In Markov state s_t at time t, one unit of consumption in state s_{t+1} at time $t+1$ costs $q(s_{t+1} \mid s_t)$ units of the time t consumption good.

The prices $q(s_{t+1} \mid s_t)$ are given and can be organized into a matrix Q with $Q_{ij} = q(j|i)$

At time $t=0$, the consumer starts with an inherited level of debt due at time 0 of b_0 units of time 0 consumption goods.

The consumer's budget constraint at $t \geq 0$ in Markov state s_t is

$$c_t + b_t \leq y(s_t) + \sum_j q(j \mid s_t)\, b_{t+1}(j \mid s_t) \tag{6.5}$$

where b_t is the consumer's one-period debt that falls due at time t and $b_{t+1}(j \mid s_t)$ are the consumer's time t sales of the time $t+1$ consumption good in Markov state j.

Thus

- $q(j \mid s_t) b_{t+1}(j \mid s_t)$ is a source of time t **financial income** for the consumer in Markov state s_t
- $b_t \equiv b_t(j \mid s_{t-1})$ is a source of time t **expenditures** for the consumer when $s_t = j$

Remark: We are ignoring an important technicality here, namely, that the consumer's choice of $b_{t+1}(j| s_t)$ must respect so-called *natural debt limits* that assure that it is feasible for the consumer to repay debts due even if he consumers zero forevermore. We shall discuss such debt limits in another lecture.

A natural analog of Hall's assumption that the one-period risk-free gross interest rate is β^{-1} is

$$q(j \mid i) = \beta P_{ij} \tag{6.6}$$

To understand how this is a natural analogue, observe that in state i it costs $\sum_j q(j \mid i)$ to purchase one unit of consumption next period *for sure*, i.e., meaning no matter what Markov state j occurs at $t+1$.

Hence the **implied price** of a risk-free claim on one unit of consumption next period is

$$\sum_j q(j \,|\, i) = \sum_j \beta P_{ij} = \beta$$

This confirms the sense in which (6.6) is a natural counterpart to Hall's assumption that the risk-free one-period gross interest rate is $R = \beta^{-1}$.

It is timely please to recall that the gross one-period risk-free interest rate is the reciprocal of the price at time t of a risk-free claim on one unit of consumption tomorrow.

First-order necessary conditions for maximizing the consumer's expected utility subject to the sequence of budget constraints (6.5) are

$$\beta \frac{u'(c_{t+1})}{u'(c_t)} \mathbb{P}\{s_{t+1} \,|\, s_t\} = q(s_{t+1} \,|\, s_t)$$

for all s_t, s_{t+1} or, under our assumption (6.6) about Arrow security prices,

$$c_{t+1} = c_t \tag{6.7}$$

Thus, our consumer sets $c_t = \bar{c}$ for all $t \geq 0$ for some value $\bar{c}$ that it is our job now to determine along with values for $b_{t+1}(j|s_t = i)$ for $i = 1, 2$ and $j = 1, 2$.

We'll use a *guess and verify* method to determine these objects

Guess: We'll make the plausible guess that

$$b_{t+1}(s_{t+1} = j \,|\, s_t = i) = b(j), \quad i = 1, 2; \;\; j = 1, 2 \tag{6.8}$$

so that the amount borrowed today depends only on *tomorrow's* Markov state. (Why is this is a plausible guess?)

To determine $\bar{c}$, we shall deduce implications of the consumer's budget constraints in each Markov state today and our guess (6.8) about the consumer's debt level choices.

For $t \geq 1$, these imply

$$\begin{aligned} \bar{c} + b(1) &= y(1) + q(1 \,|\, 1) b(1) + q(2 \,|\, 1) b(2) \\ \bar{c} + b(2) &= y(2) + q(1 \,|\, 2) b(1) + q(2 \,|\, 2) b(2) \end{aligned} \tag{6.9}$$

or

$$\begin{bmatrix} b(1) \\ b(2) \end{bmatrix} + \begin{bmatrix} \bar{c} \\ \bar{c} \end{bmatrix} = \begin{bmatrix} y(1) \\ y(2) \end{bmatrix} + \beta \begin{bmatrix} P_{11} & P_{12} \\ P_{21} & P_{22} \end{bmatrix} \begin{bmatrix} b(1) \\ b(2) \end{bmatrix}$$

These are 2 equations in the 3 unknowns $\bar{c}, b(1), b(2)$

To get a third equation, we assume that at time $t = 0$, b_0 is debt due; and we assume that at time $t = 0$, the Markov state $s_0 = 1$

(We could instead have assumed that at time $t = 0$ the Markov state $s_0 = 2$, which would affect our answer as we shall see)

Since we have assumed that $s_0 = 1$, the budget constraint at time $t = 0$ is

$$\bar{c} + b_0 = y(1) + q(1 \,|\, 1) b(1) + q(2 \,|\, 1) b(2) \tag{6.10}$$

where b_0 is the (exogenous) debt the consumer is assumed to bring into period 0

If we substitute (6.10) into the first equation of (6.9) and rearrange, we discover that

$$b(1) = b_0 \tag{6.11}$$

We can then use the second equation of (6.9) to deduce the restriction

$$y(1) - y(2) + [q(1\,|\,1) - q(1\,|\,2) - 1]b_0 + [q(2\,|\,1) + 1 - q(2\,|\,2)]b(2) = 0, \tag{6.12}$$

an equation that we can solve for the unknown $b(2)$.

Knowing $b(1)$ and $b(2)$, we can solve equation (6.10) for the constant level of consumption $\bar{c}$.

6.4.1 Key Outcomes

The preceding calculations indicate that in the complete markets version of our model, we obtain the following striking results:

- The consumer chooses to make consumption perfectly constant across time and across Markov states.
- State-contingent debt purchases $b_{t+1}(s_{t+1} = j|s_t = i)$ depend only on j
- If the initial Markov state is $s_0 = j$ and initial consumer debt is b_0, then debt in Markov state j satisfies $b(j) = b_0$

To summarize what we have achieved up to now, we have computed the constant level of consumption $\bar{c}$ and indicated how that level depends on the underlying specifications of preferences, Arrow securities prices, the stochastic process of exogenous nonfinancial income, and the initial debt level b_0

- The consumer's debt neither accumulates, nor decumulates, nor drifts – instead, the debt level each period is an exact function of the Markov state, so in the two-state Markov case, it switches between two values.
- We have verified guess (6.8).
- When the state s_t returns to the initial state s_0, debt returns to the initial debt level.
- Debt levels in all other states depend on virtually all remaining parameters of the model.

6.4.2 Code

Here's some code that, among other things, contains a function called `consumption_complete()`.

This function computes $\{b(i)\}_{i=1}^N, \bar{c}$ as outcomes given a set of parameters for the general case with N Markov states under the assumption of complete markets

```
class ConsumptionProblem:
    """
    The data for a consumption problem, including some default values.
    """

    def __init__(self,
                 β=.96,
                 y=[2, 1.5],
                 b0=3,
                 P=[[.8, .2],
                    [.4, .6]],
                 init=0):
        """
        Parameters
        ----------

        β : discount factor
        y : list containing the two income levels
        b0 : debt in period 0 (= initial state debt level)
```

(continues on next page)

(continued from previous page)

```
        P : 2x2 transition matrix
        init : index of initial state s0
        """
        self.β = β
        self.y = np.asarray(y)
        self.b0 = b0
        self.P = np.asarray(P)
        self.init = init

    def simulate(self, N_simul=80, random_state=1):
        """
        Parameters
        ----------

        N_simul : number of periods for simulation
        random_state : random state for simulating Markov chain
        """
        # For the simulation define a quantecon MC class
        mc = qe.MarkovChain(self.P)
        s_path = mc.simulate(N_simul, init=self.init, random_state=random_state)

        return s_path


def consumption_complete(cp):
    """
    Computes endogenous values for the complete market case.

    Parameters
    ----------

    cp : instance of ConsumptionProblem

    Returns
    -------

        c_bar : constant consumption
        b : optimal debt in each state

    associated with the price system

        Q = β * P
    """
    β, P, y, b0, init = cp.β, cp.P, cp.y, cp.b0, cp.init    # Unpack

    Q = β * P                                  # assumed price system

    # construct matrices of augmented equation system
    n = P.shape[0] + 1

    y_aug = np.empty((n, 1))
    y_aug[0, 0] = y[init] - b0
    y_aug[1:, 0] = y

    Q_aug = np.zeros((n, n))
    Q_aug[0, 1:] = Q[init, :]
```

(continues on next page)

(continued from previous page)

```
    Q_aug[1:, 1:] = Q

    A = np.zeros((n, n))
    A[:, 0] = 1
    A[1:, 1:] = np.eye(n-1)

    x = np.linalg.inv(A - Q_aug) @ y_aug

    c_bar = x[0, 0]
    b = x[1:, 0]

    return c_bar, b


def consumption_incomplete(cp, s_path):
    """
    Computes endogenous values for the incomplete market case.

    Parameters
    ----------

    cp : instance of ConsumptionProblem
    s_path : the path of states
    """
    β, P, y, b0 = cp.β, cp.P, cp.y, cp.b0  # Unpack

    N_simul = len(s_path)

    # Useful variables
    n = len(y)
    y.shape = (n, 1)
    v = np.linalg.inv(np.eye(n) - β * P) @ y

    # Store consumption and debt path
    b_path, c_path = np.ones(N_simul+1), np.ones(N_simul)
    b_path[0] = b0

    # Optimal decisions from (12) and (13)
    db = ((1 - β) * v - y) / β

    for i, s in enumerate(s_path):
        c_path[i] = (1 - β) * (v - np.full((n, 1), b_path[i]))[s, 0]
        b_path[i + 1] = b_path[i] + db[s, 0]

    return c_path, b_path[:-1], y[s_path]
```

Let's test by checking that $\bar{c}$ and b_2 satisfy the budget constraint

```
cp = ConsumptionProblem()
c_bar, b = consumption_complete(cp)
np.isclose(c_bar + b[1] - cp.y[1] - (cp.β * cp.P)[1, :] @ b, 0)
```

```
True
```

Below, we'll take the outcomes produced by this code – in particular the implied consumption and debt paths – and compare them with outcomes from an incomplete markets model in the spirit of Hall [Hall, 1978]

6.5 Model 2 (One-Period Risk-Free Debt Only)

This is a version of the original model of Hall (1978) in which the consumer's ability to substitute intertemporally is constrained by his ability to buy or sell only one security, a risk-free one-period bond bearing a constant gross interest rate that equals β^{-1}.

Given an initial debt b_0 at time 0, the consumer faces a sequence of budget constraints

$$c_t + b_t = y_t + \beta b_{t+1}, \quad t \geq 0$$

where β is the price at time t of a risk-free claim on one unit of time consumption at time $t+1$.

First-order conditions for the consumer's problem are

$$\sum_j u'(c_{t+1,j}) P_{ij} = u'(c_{t,i})$$

For our assumed quadratic utility function this implies

$$\sum_j c_{t+1,j} P_{ij} = c_{t,i} \tag{6.13}$$

which for our finite-state Markov setting is Hall's (1978) conclusion that consumption follows a random walk.

As we saw in our first lecture on the permanent income model, this leads to

$$b_t = \mathbb{E}_t \sum_{j=0}^{\infty} \beta^j y_{t+j} - (1-\beta)^{-1} c_t \tag{6.14}$$

and

$$c_t = (1-\beta) \left[\mathbb{E}_t \sum_{j=0}^{\infty} \beta^j y_{t+j} - b_t \right] \tag{6.15}$$

Equation (6.15) expresses c_t as a net interest rate factor $1-\beta$ times the sum of the expected present value of nonfinancial income $\mathbb{E}_t \sum_{j=0}^{\infty} \beta^j y_{t+j}$ and financial wealth $-b_t$.

Substituting (6.15) into the one-period budget constraint and rearranging leads to

$$b_{t+1} - b_t = \beta^{-1} \left[(1-\beta) \mathbb{E}_t \sum_{j=0}^{\infty} \beta^j y_{t+j} - y_t \right] \tag{6.16}$$

Now let's calculate the key term $\mathbb{E}_t \sum_{j=0}^{\infty} \beta^j y_{t+j}$ in our finite Markov chain setting.

Define the expected discounted present value of non-financial income

$$v_t := \mathbb{E}_t \sum_{j=0}^{\infty} \beta^j y_{t+j}$$

which in the spirit of dynamic programming we can write as a *Bellman equation*

$$v_t := y_t + \beta \mathbb{E}_t v_{t+1}$$

In our two-state Markov chain setting, $v_t = v(1)$ when $s_t = 1$ and $v_t = v(2)$ when $s_t = 2$.

Therefore, we can write our Bellman equation as

$$\begin{aligned} v(1) &= y(1) + \beta P_{11} v(1) + \beta P_{12} v(2) \\ v(2) &= y(2) + \beta P_{21} v(1) + \beta P_{22} v(2) \end{aligned}$$

or

$$\vec{v} = \vec{y} + \beta P \vec{v}$$

where $\vec{v} = \begin{bmatrix} v(1) \\ v(2) \end{bmatrix}$ and $\vec{y} = \begin{bmatrix} y(1) \\ y(2) \end{bmatrix}$.

We can also write the last expression as

$$\vec{v} = (I - \beta P)^{-1} \vec{y}$$

In our finite Markov chain setting, from expression (6.15), consumption at date t when debt is b_t and the Markov state today is $s_t = i$ is evidently

$$c(b_t, i) = (1 - \beta) \left([(I - \beta P)^{-1} \vec{y}]_i - b_t \right) \tag{6.17}$$

and the increment to debt is

$$b_{t+1} - b_t = \beta^{-1} [(1 - \beta) v(i) - y(i)] \tag{6.18}$$

6.5.1 Summary of Outcomes

In contrast to outcomes in the complete markets model, in the incomplete markets model

- consumption drifts over time as a random walk; the level of consumption at time t depends on the level of debt that the consumer brings into the period as well as the expected discounted present value of nonfinancial income at t.
- the consumer's debt drifts upward over time in response to low realizations of nonfinancial income and drifts downward over time in response to high realizations of nonfinancial income.
- the drift over time in the consumer's debt and the dependence of current consumption on today's debt level account for the drift over time in consumption.

6.5.2 The Incomplete Markets Model

The code above also contains a function called `consumption_incomplete()` that uses (6.17) and (6.18) to

- simulate paths of y_t, c_t, b_{t+1}
- plot these against values of $\bar{c}, b(s_1), b(s_2)$ found in a corresponding complete markets economy

Let's try this, using the same parameters in both complete and incomplete markets economies

```
cp = ConsumptionProblem()
s_path = cp.simulate()
N_simul = len(s_path)

c_bar, debt_complete = consumption_complete(cp)

c_path, debt_path, y_path = consumption_incomplete(cp, s_path)

fig, ax = plt.subplots(1, 2, figsize=(14, 4))

ax[0].set_title('Consumption paths')
ax[0].plot(np.arange(N_simul), c_path, label='incomplete market')
ax[0].plot(np.arange(N_simul), np.full(N_simul, c_bar),
```

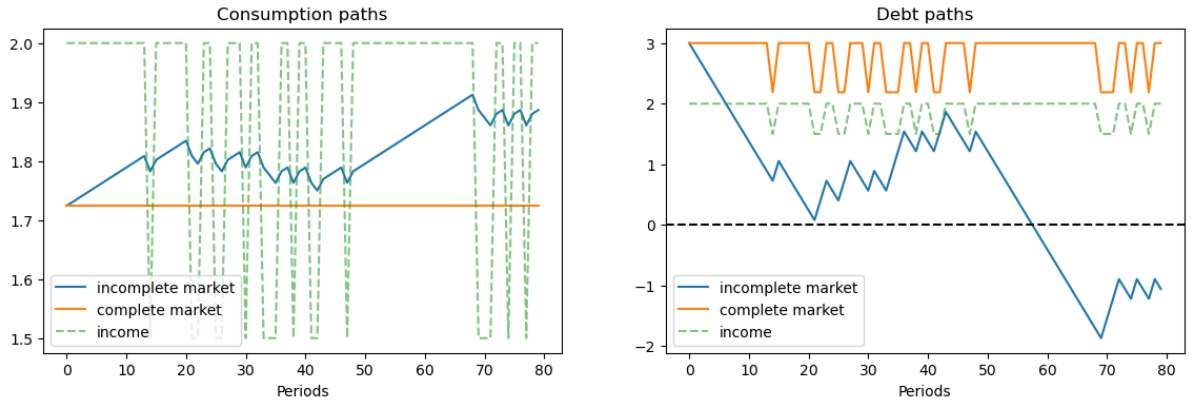
(continues on next page)

(continued from previous page)

```
                    label='complete market')
ax[0].plot(np.arange(N_simul), y_path, label='income', alpha=.6, ls='--')
ax[0].legend()
ax[0].set_xlabel('Periods')

ax[1].set_title('Debt paths')
ax[1].plot(np.arange(N_simul), debt_path, label='incomplete market')
ax[1].plot(np.arange(N_simul), debt_complete[s_path],
           label='complete market')
ax[1].plot(np.arange(N_simul), y_path, label='income', alpha=.6, ls='--')
ax[1].legend()
ax[1].axhline(0, color='k', ls='--')
ax[1].set_xlabel('Periods')

plt.show()
```



In the graph on the left, for the same sample path of nonfinancial income y_t, notice that

- consumption is constant when there are complete markets, but takes a random walk in the incomplete markets version of the model.
- the consumer's debt oscillates between two values that are functions of the Markov state in the complete markets model, while the consumer's debt drifts in a "unit root" fashion in the incomplete markets economy.

6.5.3 A sequel

In *tax smoothing with complete and incomplete markets*, we reinterpret the mathematics and Python code presented in this lecture in order to construct tax-smoothing models in the incomplete markets tradition of Barro [Barro, 1979] as well as in the complete markets tradition of Lucas and Stokey [Lucas and Stokey, 1983].

CHAPTER

SEVEN

TAX SMOOTHING WITH COMPLETE AND INCOMPLETE MARKETS

In addition to what's in Anaconda, this lecture uses the library:

```
!pip install --upgrade quantecon
```

7.1 Overview

This lecture describes tax-smoothing models that are counterparts to consumption-smoothing models in *Consumption Smoothing with Complete and Incomplete Markets*.

- one is in the **complete markets** tradition of Lucas and Stokey [Lucas and Stokey, 1983].
- the other is in the **incomplete markets** tradition of Barro [Barro, 1979].

Complete markets allow a government to buy or sell claims contingent on all possible Markov states.

Incomplete markets allow a government to buy or sell only a limited set of securities, often only a single risk-free security.

Barro [Barro, 1979] worked in an incomplete markets tradition by assuming that the only asset that can be traded is a risk-free one period bond.

In his consumption-smoothing model, Hall [Hall, 1978] had assumed an exogenous stochastic process of nonfinancial income and an exogenous gross interest rate on one period risk-free debt that equals β^{-1}, where $\beta \in (0, 1)$ is also a consumer's intertemporal discount factor.

Barro [Barro, 1979] made an analogous assumption about the risk-free interest rate in a tax-smoothing model that turns out to have the same mathematical structure as Hall's consumption-smoothing model.

To get Barro's model from Hall's, all we have to do is to rename variables.

We maintain Hall's and Barro's assumption about the interest rate when we describe an incomplete markets version of our model.

In addition, we extend their assumption about the interest rate to an appropriate counterpart to create a "complete markets" model in the style of Lucas and Stokey [Lucas and Stokey, 1983].

7.1.1 Isomorphism between Consumption and Tax Smoothing

For each version of a consumption-smoothing model, a tax-smoothing counterpart can be obtained simply by relabeling

- consumption as tax collections
- a consumer's one-period utility function as a government's one-period loss function from collecting taxes that impose deadweight welfare losses
- a consumer's nonfinancial income as a government's purchases
- a consumer's *debt* as a government's *assets*

Thus, we can convert the consumption-smoothing models in lecture *Consumption Smoothing with Complete and Incomplete Markets* into tax-smoothing models by setting $c_t = T_t$, $y_t = G_t$, and $-b_t = a_t$, where T_t is total tax collections, $\{G_t\}$ is an exogenous government expenditures process, and a_t is the government's holdings of one-period risk-free bonds coming maturing at the due at the beginning of time t.

For elaborations on this theme, please see Optimal Savings II: LQ Techniques and later parts of this lecture.

We'll spend most of this lecture studying acquire finite-state Markov specification, but will also treat the linear state space specification.

Link to History

For those who love history, President Thomas Jefferson's Secretary of Treasury Albert Gallatin (1807) [Gallatin, 1837] seems to have prescribed policies that come from Barro's model [Barro, 1979]

Let's start with some standard imports:

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

To exploit the isomorphism between consumption-smoothing and tax-smoothing models, we simply use code from *Consumption Smoothing with Complete and Incomplete Markets*

7.1.2 Code

Among other things, this code contains a function called `consumption_complete()`.

This function computes $\{b(i)\}_{i=1}^{N}, \bar{c}$ as outcomes given a set of parameters for the general case with N Markov states under the assumption of complete markets

```
class ConsumptionProblem:
    """
    The data for a consumption problem, including some default values.
    """

    def __init__(self,
                 β=.96,
                 y=[2, 1.5],
                 b0=3,
                 P=[[.8, .2],
                    [.4, .6]],
                 init=0):
        """
```

(continues on next page)

(continued from previous page)

```
        Parameters
        ----------

        β : discount factor
        y : list containing the two income levels
        b0 : debt in period 0 (= initial state debt level)
        P : 2x2 transition matrix
        init : index of initial state s0
        """
        self.β = β
        self.y = np.asarray(y)
        self.b0 = b0
        self.P = np.asarray(P)
        self.init = init

    def simulate(self, N_simul=80, random_state=1):
        """
        Parameters
        ----------

        N_simul : number of periods for simulation
        random_state : random state for simulating Markov chain
        """
        # For the simulation define a quantecon MC class
        mc = qe.MarkovChain(self.P)
        s_path = mc.simulate(N_simul, init=self.init, random_state=random_state)

        return s_path


def consumption_complete(cp):
    """
    Computes endogenous values for the complete market case.

    Parameters
    ----------

    cp : instance of ConsumptionProblem

    Returns
    -------

        c_bar : constant consumption
        b : optimal debt in each state

    associated with the price system

        Q = β * P
    """
    β, P, y, b0, init = cp.β, cp.P, cp.y, cp.b0, cp.init    # Unpack

    Q = β * P                                  # assumed price system

    # construct matrices of augmented equation system
    n = P.shape[0] + 1
```

(continues on next page)

(continued from previous page)

```
    y_aug = np.empty((n, 1))
    y_aug[0, 0] = y[init] - b0
    y_aug[1:, 0] = y

    Q_aug = np.zeros((n, n))
    Q_aug[0, 1:] = Q[init, :]
    Q_aug[1:, 1:] = Q

    A = np.zeros((n, n))
    A[:, 0] = 1
    A[1:, 1:] = np.eye(n-1)

    x = np.linalg.inv(A - Q_aug) @ y_aug

    c_bar = x[0, 0]
    b = x[1:, 0]

    return c_bar, b


def consumption_incomplete(cp, s_path):
    """
    Computes endogenous values for the incomplete market case.

    Parameters
    ----------

    cp : instance of ConsumptionProblem
    s_path : the path of states
    """
    β, P, y, b0 = cp.β, cp.P, cp.y, cp.b0  # Unpack

    N_simul = len(s_path)

    # Useful variables
    n = len(y)
    y.shape = (n, 1)
    v = np.linalg.inv(np.eye(n) - β * P) @ y

    # Store consumption and debt path
    b_path, c_path = np.ones(N_simul+1), np.ones(N_simul)
    b_path[0] = b0

    # Optimal decisions from (12) and (13)
    db = ((1 - β) * v - y) / β

    for i, s in enumerate(s_path):
        c_path[i] = (1 - β) * (v - np.full((n, 1), b_path[i]))[s, 0]
        b_path[i + 1] = b_path[i] + db[s, 0]

    return c_path, b_path[:-1], y[s_path]
```

7.1.3 Revisiting the consumption-smoothing model

The code above also contains a function called `consumption_incomplete()` that uses (6.17) and (6.18) to

- simulate paths of y_t, c_t, b_{t+1}
- plot these against values of $\bar{c}, b(s_1), b(s_2)$ found in a corresponding complete markets economy

Let's try this, using the same parameters in both complete and incomplete markets economies

```
cp = ConsumptionProblem()
s_path = cp.simulate()
N_simul = len(s_path)

c_bar, debt_complete = consumption_complete(cp)

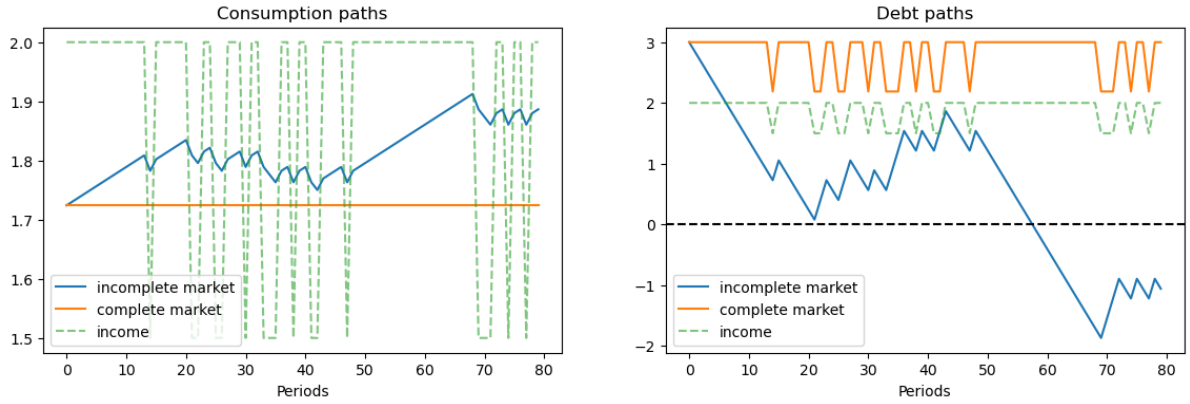
c_path, debt_path, y_path = consumption_incomplete(cp, s_path)

fig, ax = plt.subplots(1, 2, figsize=(14, 4))

ax[0].set_title('Consumption paths')
ax[0].plot(np.arange(N_simul), c_path, label='incomplete market')
ax[0].plot(np.arange(N_simul), np.full(N_simul, c_bar), label='complete market')
ax[0].plot(np.arange(N_simul), y_path, label='income', alpha=.6, ls='--')
ax[0].legend()
ax[0].set_xlabel('Periods')

ax[1].set_title('Debt paths')
ax[1].plot(np.arange(N_simul), debt_path, label='incomplete market')
ax[1].plot(np.arange(N_simul), debt_complete[s_path], label='complete market')
ax[1].plot(np.arange(N_simul), y_path, label='income', alpha=.6, ls='--')
ax[1].legend()
ax[1].axhline(0, color='k', ls='--')
ax[1].set_xlabel('Periods')

plt.show()
```



In the graph on the left, for the same sample path of nonfinancial income y_t, notice that

- consumption is constant when there are complete markets.
- consumption takes a random walk in the incomplete markets version of the model.
- the consumer's debt oscillates between two values that are functions of the Markov state in the complete markets model.

- the consumer's debt drifts because it contains a unit root in the incomplete markets economy.

Relabeling variables to create tax-smoothing models

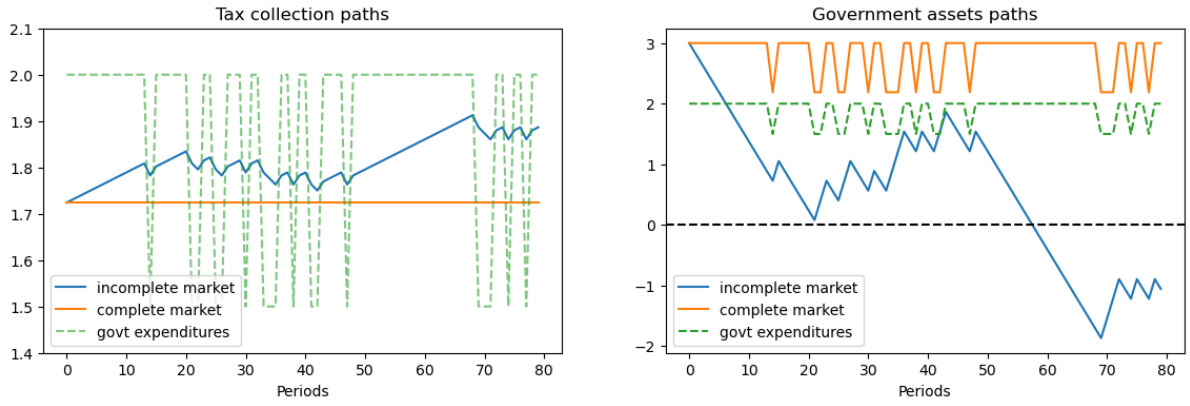
As indicated above, we relabel variables to acquire tax-smoothing interpretations of the complete markets and incomplete markets consumption-smoothing models.

```
fig, ax = plt.subplots(1, 2, figsize=(14, 4))

ax[0].set_title('Tax collection paths')
ax[0].plot(np.arange(N_simul), c_path, label='incomplete market')
ax[0].plot(np.arange(N_simul), np.full(N_simul, c_bar), label='complete market')
ax[0].plot(np.arange(N_simul), y_path, label='govt expenditures', alpha=.6, ls='--')
ax[0].legend()
ax[0].set_xlabel('Periods')
ax[0].set_ylim([1.4, 2.1])

ax[1].set_title('Government assets paths')
ax[1].plot(np.arange(N_simul), debt_path, label='incomplete market')
ax[1].plot(np.arange(N_simul), debt_complete[s_path], label='complete market')
ax[1].plot(np.arange(N_simul), y_path, label='govt expenditures', ls='--')
ax[1].legend()
ax[1].axhline(0, color='k', ls='--')
ax[1].set_xlabel('Periods')

plt.show()
```



7.2 Tax Smoothing with Complete Markets

It is instructive to focus on a simple tax-smoothing example with complete markets.

This example illustrates how, in a complete markets model like that of Lucas and Stokey [Lucas and Stokey, 1983], the government purchases insurance from the private sector.

Payouts from the insurance it had purchased allows the government to avoid raising taxes when emergencies make government expenditures surge.

We assume that government expenditures take one of two values $G_1 < G_2$, where Markov state 1 means "peace" and Markov state 2 means "war".

The government budget constraint in Markov state i is

$$T_i + b_i = G_i + \sum_j Q_{ij} b_j$$

where

$$Q_{ij} = \beta P_{ij}$$

is the price today of one unit of goods in Markov state j tomorrow when the Markov state is i today.

b_i is the government's level of *assets* when it arrives in Markov state i.

That is, b_i equals one-period state-contingent claims *owed to the government* that fall due at time t when the Markov state is i.

Thus, if $b_i < 0$, it means the government **is owed** b_i or **owes** $-b_i$ when the economy arrives in Markov state i at time t.

In our examples below, this happens when in a previous war-time period the government has sold an Arrow securities paying off $-b_i$ in peacetime Markov state i

It can be enlightening to express the government's budget constraint in Markov state i as

$$T_i = G_i + \left(\sum_j Q_{ij} b_j - b_i \right)$$

in which the term $(\sum_j Q_{ij} b_j - b_i)$ equals the net amount that the government spends to purchase one-period Arrow securities that will pay off next period in Markov states $j = 1, \ldots, N$ after it has received payments b_i this period.

7.3 Returns on State-Contingent Debt

Notice that $\sum_{j'=1}^N Q_{ij'} b(j')$ is the amount that the government spends in Markov state i at time t to purchase one-period state-contingent claims that will pay off in Markov state j' at time $t+1$.

Then the *ex post* one-period gross return on the portfolio of government assets held from state i at time t to state j at time $t+1$ is

$$R(j|i) = \frac{b(j)}{\sum_{j'=1}^N Q_{ij'} b(j')}$$

The cumulative return earned from putting 1 unit of time t goods into the government portfolio of state-contingent securities at time t and then rolling over the proceeds into the government portfolio each period thereafter is

$$R^T(s_{t+T}, s_{t+T-1}, \ldots, s_t) \equiv R(s_{t+1}|s_t) R(s_{t+2}|s_{t+1}) \cdots R(s_{t+T}|s_{t+T-1})$$

Here is some code that computes one-period and cumulative returns on the government portfolio in the finite-state Markov version of our complete markets model.

Convention: In this code, when $P_{ij} = 0$, we arbitrarily set $R(j|i)$ to be 0.

```
def ex_post_gross_return(b, cp):
    """
    calculate the ex post one-period gross return on the portfolio
    of government assets, given b and Q.
    """
    Q = cp.β * cp.P
```

(continues on next page)

(continued from previous page)

```
    values = Q @ b

    n = len(b)
    R = np.zeros((n, n))

    for i in range(n):
        ind = cp.P[i, :] != 0
        R[i, ind] = b[ind] / values[i]

    return R

def cumulative_return(s_path, R):
    """
    compute cumulative return from holding 1 unit market portfolio
    of government bonds, given some simulated state path.
    """
    T = len(s_path)

    RT_path = np.empty(T)
    RT_path[0] = 1
    RT_path[1:] = np.cumprod([R[s_path[t], s_path[t+1]] for t in range(T-1)])

    return RT_path
```

7.3.1 An Example of Tax Smoothing

We'll study a tax-smoothing model with two Markov states.

In Markov state 1, there is peace and government expenditures are low.

In Markov state 2, there is war and government expenditures are high.

We'll compute optimal policies in both complete and incomplete markets settings.

Then we'll feed in a **particular** assumed path of Markov states and study outcomes.

- We'll assume that the initial Markov state is state 1, which means we start from a state of peace.
- The government then experiences 3 time periods of war and come back to peace again.
- The history of Markov states is therefore $\{peace, war, war, war, peace\}$.

In addition, as indicated above, to simplify our example, we'll set the government's initial asset level to 1, so that $b_1 = 1$.

Here's code that itinitializes government assets to be unity in an initial peace time Markov state.

```
# Parameters
β = .96

# change notation y to g in the tax-smoothing example
g = [1, 2]
b0 = 1
P = np.array([[.8, .2],
              [.4, .6]])

cp = ConsumptionProblem(β, g, b0, P)
```

(continues on next page)

(continued from previous page)

```
Q = β * P

# change notation c_bar to T_bar in the tax-smoothing example
T_bar, b = consumption_complete(cp)
R = ex_post_gross_return(b, cp)
s_path = [0, 1, 1, 1, 0]
RT_path = cumulative_return(s_path, R)

print(f"P \n {P}")
print(f"Q \n {Q}")
print(f"Govt expenditures in peace and war = {g}")
print(f"Constant tax collections = {T_bar}")
print(f"Govt debts in two states = {-b}")

msg = """
Now let's check the government's budget constraint in peace and war.
Our assumptions imply that the government always purchases 0 units of the
Arrow peace security.
"""
print(msg)

AS1 = Q[0, :] @ b
# spending on Arrow security
# since the spending on Arrow peace security is not 0 anymore after we change b0 to 1
print(f"Spending on Arrow security in peace = {AS1}")
AS2 = Q[1, :] @ b
print(f"Spending on Arrow security in war = {AS2}")

print("")
# tax collections minus debt levels
print("Government tax collections minus debt levels in peace and war")
TB1 = T_bar + b[0]
print(f"T+b in peace = {TB1}")
TB2 = T_bar + b[1]
print(f"T+b in war = {TB2}")

print("")
print("Total government spending in peace and war")
G1 = g[0] + AS1
G2 = g[1] + AS2
print(f"Peace = {G1}")
print(f"War = {G2}")

print("")
print("Let's see ex-post and ex-ante returns on Arrow securities")

Π = np.reciprocal(Q)
exret = Π
print(f"Ex-post returns to purchase of Arrow securities = \n {exret}")
exant = Π * P
print(f"Ex-ante returns to purchase of Arrow securities \n {exant}")

print("")
print("The Ex-post one-period gross return on the portfolio of government assets")
print(R)
```

(continues on next page)

(continued from previous page)

```
print("")
print("The cumulative return earned from holding 1 unit market portfolio of␣
 ↪government bonds")
print(RT_path[-1])
```

```
P
 [[0.8 0.2]
 [0.4 0.6]]
Q
 [[0.768 0.192]
 [0.384 0.576]]
Govt expenditures in peace and war = [1, 2]
Constant tax collections = 1.2716883116883118
Govt debts in two states = [-1.         -2.62337662]

Now let's check the government's budget constraint in peace and war.
Our assumptions imply that the government always purchases 0 units of the
Arrow peace security.

Spending on Arrow security in peace = 1.2716883116883118
Spending on Arrow security in war = 1.895064935064935

Government tax collections minus debt levels in peace and war
T+b in peace = 2.2716883116883118
T+b in war = 3.895064935064935

Total government spending in peace and war
Peace = 2.2716883116883118
War = 3.895064935064935

Let's see ex-post and ex-ante returns on Arrow securities
Ex-post returns to purchase of Arrow securities =
 [[1.30208333 5.20833333]
 [2.60416667 1.73611111]]
Ex-ante returns to purchase of Arrow securities
 [[1.04166667 1.04166667]
 [1.04166667 1.04166667]]

The Ex-post one-period gross return on the portfolio of government assets
[[0.78635621 2.0629085 ]
 [0.5276864  1.38432018]]

The cumulative return earned from holding 1 unit market portfolio of government␣
 ↪bonds
2.0860704239993675
```

7.3.2 Explanation

In this example, the government always purchase 1 units of the Arrow security that pays off in peace time (Markov state 1).

And it purchases a higher amount of the security that pays off in war time (Markov state 2).

Thus, this is an example in which

- during peacetime, the government purchases *insurance* against the possibility that war breaks out next period
- during wartime, the government purchases *insurance* against the possibility that war continues another period
- so long as peace continues, the ex post return on insurance against war is low
- when war breaks out or continues, the ex post return on insurance against war is high
- given the history of states that we assumed, the value of one unit of the portfolio of government assets eventually doubles in the end because of high returns during wartime.

We recommend plugging the quantities computed above into the government budget constraints in the two Markov states and staring.

Exercise 7.3.1

Try changing the Markov transition matrix so that

$$P = \begin{bmatrix} 1 & 0 \\ .2 & .8 \end{bmatrix}$$

Also, start the system in Markov state 2 (war) with initial government assets -10, so that the government starts the war in debt and $b_2 = -10$.

7.4 More Finite Markov Chain Tax-Smoothing Examples

To interpret some episodes in the fiscal history of the United States, we find it interesting to study a few more examples.

We compute examples in an N state Markov setting under both complete and incomplete markets.

These examples differ in how Markov states are jumping between peace and war.

To wrap procedures for solving models, relabeling graphs so that we record government *debt* rather than government *assets*, and displaying results, we construct a Python class.

```
class TaxSmoothingExample:
    """
    construct a tax-smoothing example, by relabeling consumption problem class.
    """
    def __init__(self, g, P, b0, states, β=.96,
                 init=0, s_path=None, N_simul=80, random_state=1):

        self.states = states # state names

        # if the path of states is not specified
        if s_path is None:
            self.cp = ConsumptionProblem(β, g, b0, P, init=init)
            self.s_path = self.cp.simulate(N_simul=N_simul, random_state=random_state)
```

(continues on next page)

(continued from previous page)

```
        # if the path of states is specified
        else:
            self.cp = ConsumptionProblem(β, g, b0, P, init=s_path[0])
            self.s_path = s_path

        # solve for complete market case
        self.T_bar, self.b = consumption_complete(self.cp)
        self.debt_value = - (β * P @ self.b).T

        # solve for incomplete market case
        self.T_path, self.asset_path, self.g_path = \
            consumption_incomplete(self.cp, self.s_path)

        # calculate returns on state-contingent debt
        self.R = ex_post_gross_return(self.b, self.cp)
        self.RT_path = cumulative_return(self.s_path, self.R)

    def display(self):

        # plot graphs
        N = len(self.T_path)

        plt.figure()
        plt.title('Tax collection paths')
        plt.plot(np.arange(N), self.T_path, label='incomplete market')
        plt.plot(np.arange(N), np.full(N, self.T_bar), label='complete market')
        plt.plot(np.arange(N), self.g_path, label='govt expenditures', alpha=.6, ls='-
↪-')
        plt.legend()
        plt.xlabel('Periods')
        plt.show()

        plt.title('Government debt paths')
        plt.plot(np.arange(N), -self.asset_path, label='incomplete market')
        plt.plot(np.arange(N), -self.b[self.s_path], label='complete market')
        plt.plot(np.arange(N), self.g_path, label='govt expenditures', ls='--')
        plt.plot(np.arange(N), self.debt_value[self.s_path], label="value of debts␣
↪today")
        plt.legend()
        plt.axhline(0, color='k', ls='--')
        plt.xlabel('Periods')
        plt.show()

        fig, ax = plt.subplots()
        ax.set_title('Cumulative return path (complete markets)')
        line1 = ax.plot(np.arange(N), self.RT_path, color='blue')[0]
        c1 = line1.get_color()
        ax.set_xlabel('Periods')
        ax.set_ylabel('Cumulative return', color=c1)

        ax_ = ax.twinx()
        line2 = ax_.plot(np.arange(N), self.g_path, ls='--', color='green')[0]
        c2 = line2.get_color()
        ax_.set_ylabel('Government expenditures', color=c2)

        plt.show()
```

(continues on next page)

(continued from previous page)

```
        # plot detailed information
        Q = self.cp.β * self.cp.P

        print(f"P \n {self.cp.P}")
        print(f"Q \n {Q}")
        print(f"Govt expenditures in {', '.join(self.states)} = {self.cp.y.flatten()}
↪")
        print(f"Constant tax collections = {self.T_bar}")
        print(f"Govt debt in {len(self.states)} states = {-self.b}")

        print("")
        print(f"Government tax collections minus debt levels in {', '.join(self.
↪states)}")
        for i in range(len(self.states)):
            TB = self.T_bar + self.b[i]
            print(f"  T+b in {self.states[i]} = {TB}")

        print("")
        print(f"Total government spending in {', '.join(self.states)}")
        for i in range(len(self.states)):
            G = self.cp.y[i, 0] + Q[i, :] @ self.b
            print(f"  {self.states[i]} = {G}")

        print("")
        print("Let's see ex-post and ex-ante returns on Arrow securities \n")

        print(f"Ex-post returns to purchase of Arrow securities:")
        for i in range(len(self.states)):
            for j in range(len(self.states)):
                if Q[i, j] != 0.:
                    print(f"  π({self.states[j]}|{self.states[i]}) = {1/Q[i, j]}")

        print("")
        exant = 1 / self.cp.β
        print(f"Ex-ante returns to purchase of Arrow securities = {exant}")

        print("")
        print("The Ex-post one-period gross return on the portfolio of government␣
↪assets")
        print(self.R)

        print("")
        print("The cumulative return earned from holding 1 unit market portfolio of␣
↪government bonds")
        print(self.RT_path[-1])
```

7.4.1 Parameters

```
γ = .1
λ = .1
ϕ = .1
θ = .1
ψ = .1
g_L = .5
g_M = .8
g_H = 1.2
β = .96
```

7.4.2 Example 1

This example is designed to produce some stylized versions of tax, debt, and deficit paths followed by the United States during and after the Civil War and also during and after World War I.

We set the Markov chain to have three states

$$P = \begin{bmatrix} 1-\lambda & \lambda & 0 \\ 0 & 1-\phi & \phi \\ 0 & 0 & 1 \end{bmatrix}$$

where the government expenditure vector $g = \begin{bmatrix} g_L & g_H & g_M \end{bmatrix}$ where $g_L < g_M < g_H$.

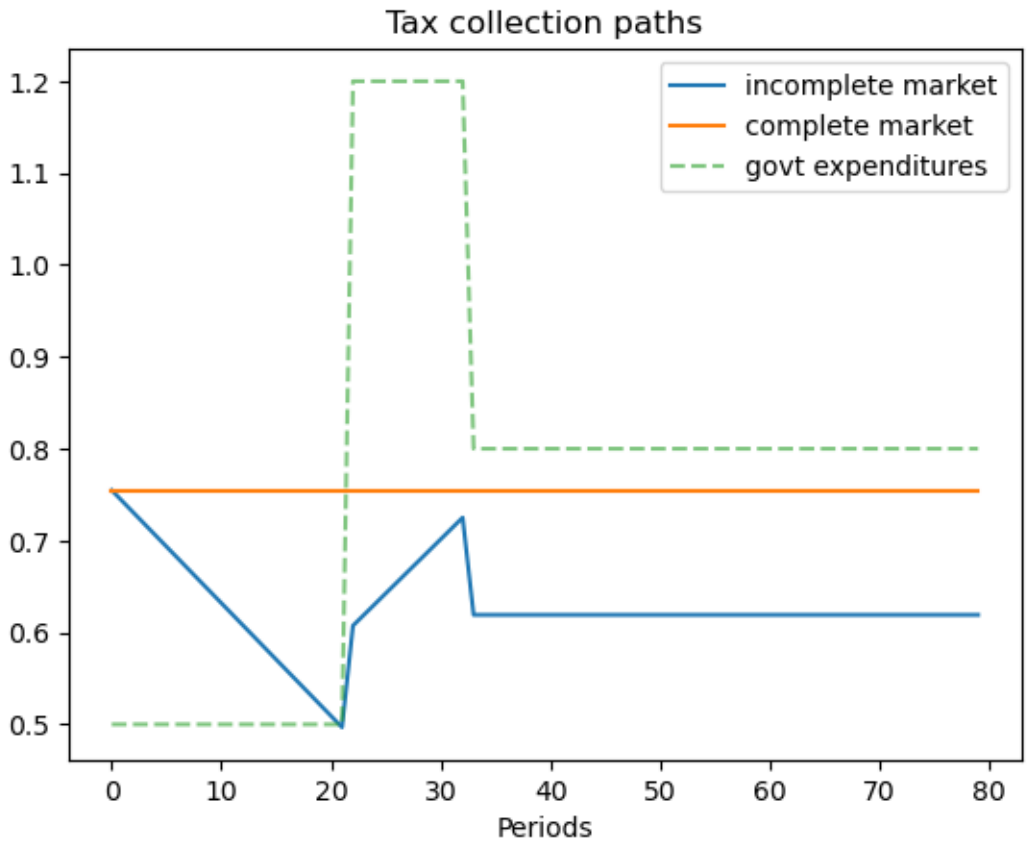
We set $b_0 = 1$ and assume that the initial Markov state is state 1 so that the system starts off in peace.

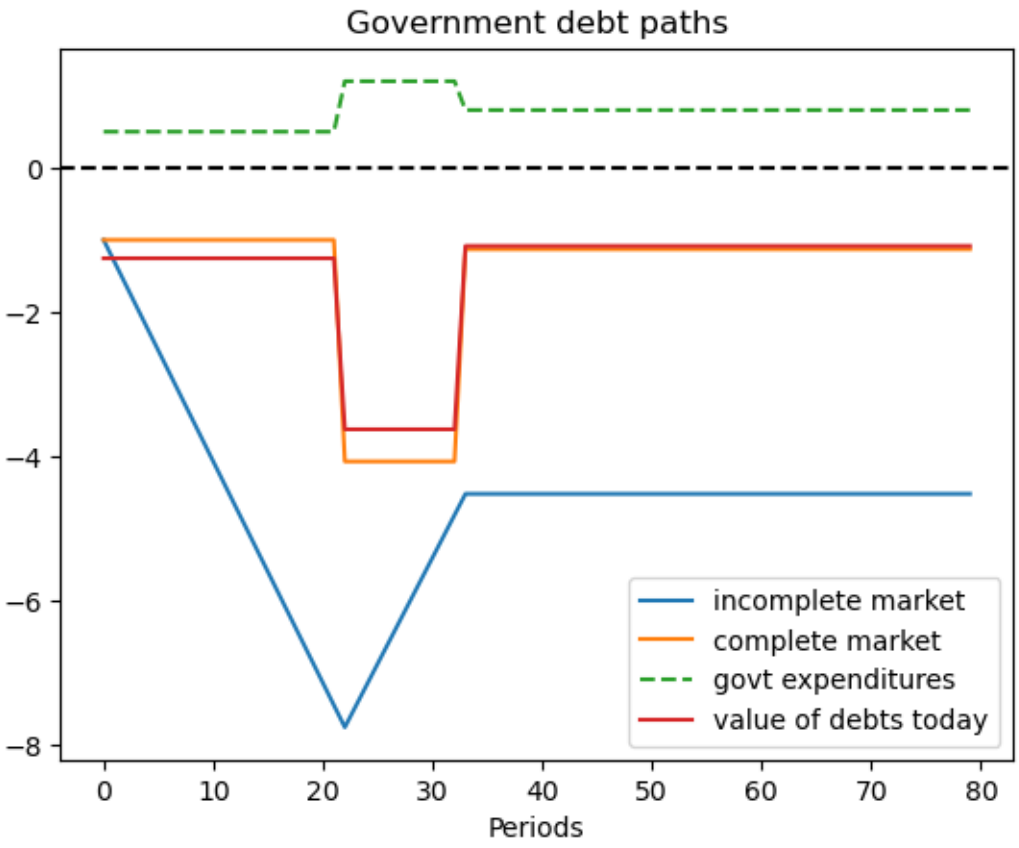
These parameters have government expenditure beginning at a low level, surging during the war, then decreasing after the war to a level that exceeds its prewar level.

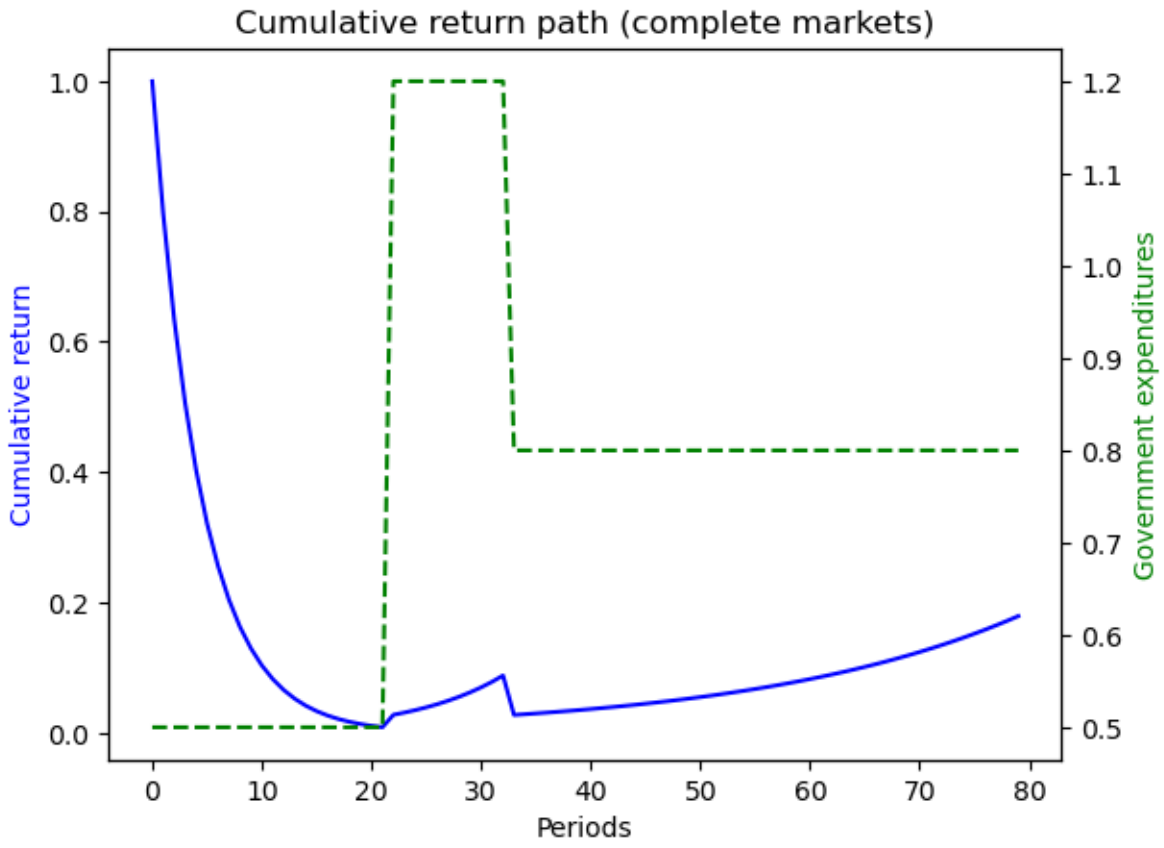
(This type of pattern occurred in the US Civil War and World War I experiences.)

```
g_ex1 = [g_L, g_H, g_M]
P_ex1 = np.array([[1-λ, λ,  0],
                  [0, 1-ϕ,  ϕ],
                  [0,   0,  1]])
b0_ex1 = 1
states_ex1 = ['peace', 'war', 'postwar']
```

```
ts_ex1 = TaxSmoothingExample(g_ex1, P_ex1, b0_ex1, states_ex1, random_state=1)
ts_ex1.display()
```


Tax collection paths
incomplete market
complete market
govt expenditures
1.2
1.1
1.0
0.9
0.8
0.7
0.6
0.5
0
10
20
30
40
50
60
70
80
Periods


Government debt paths
0
−2
−4
−6
−8
0
10
20
30
40
50
60
70
80
Periods
incomplete market
complete market
govt expenditures
value of debts today




```
P
 [[0.9 0.1 0. ]
 [0.  0.9 0.1]
 [0.  0.  1. ]]
Q
 [[0.864 0.096 0.   ]
 [0.    0.864 0.096]
 [0.    0.    0.96 ]]
Govt expenditures in peace, war, postwar = [0.5 1.2 0.8]
Constant tax collections = 0.7548096885813149
Govt debt in 3 states = [-1.         -4.07093426 -1.12975779]

Government tax collections minus debt levels in peace, war, postwar
  T+b in peace = 1.754809688581315
  T+b in war = 4.825743944636679
  T+b in postwar = 1.8845674740484442

Total government spending in peace, war, postwar
  peace = 1.754809688581315
  war = 4.825743944636679
  postwar = 1.8845674740484442

Let's see ex-post and ex-ante returns on Arrow securities

Ex-post returns to purchase of Arrow securities:
  π(peace|peace) = 1.1574074074074074
  π(war|peace) = 10.416666666666666
```

(continues on next page)

(continued from previous page)

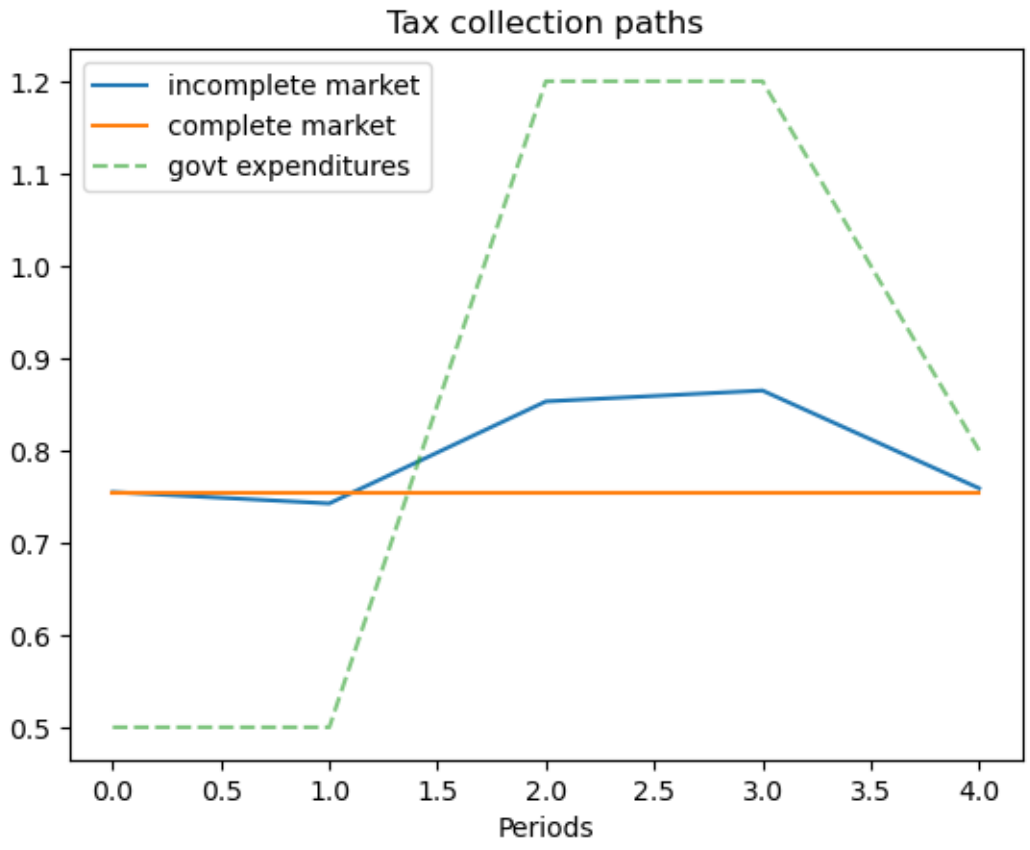
```
  π(war|war) = 1.1574074074074074
  π(postwar|war) = 10.416666666666666
  π(postwar|postwar) = 1.0416666666666667

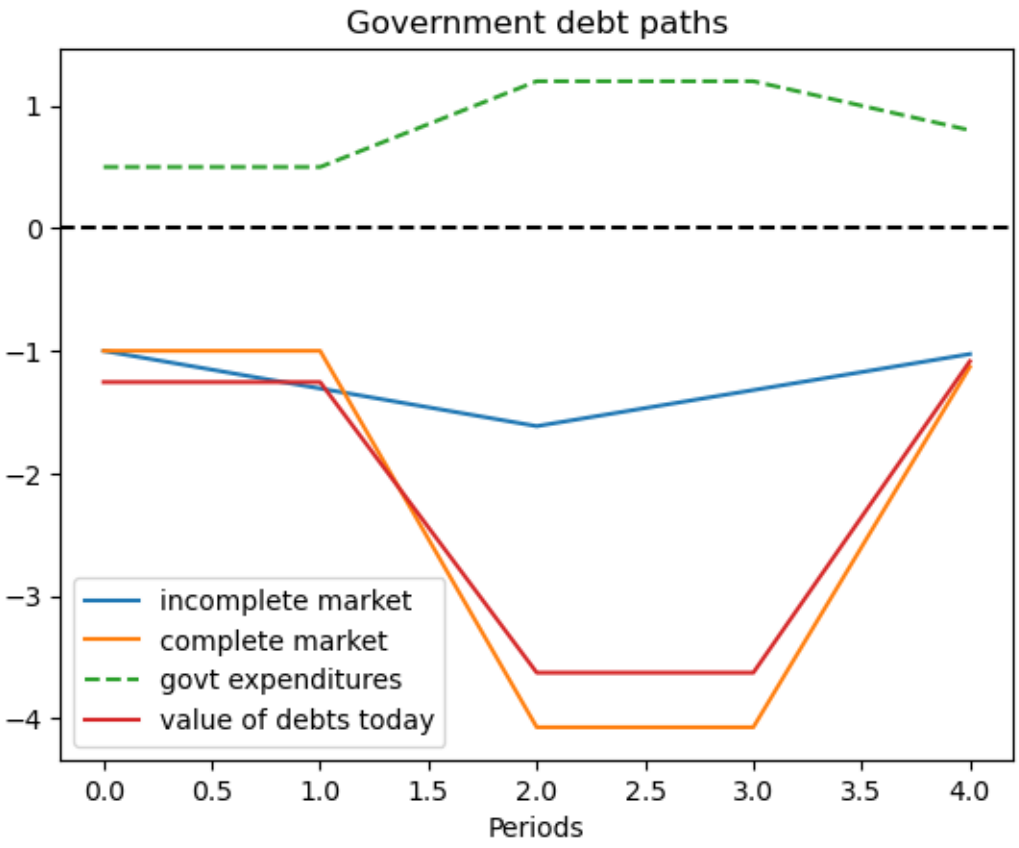
Ex-ante returns to purchase of Arrow securities = 1.0416666666666667

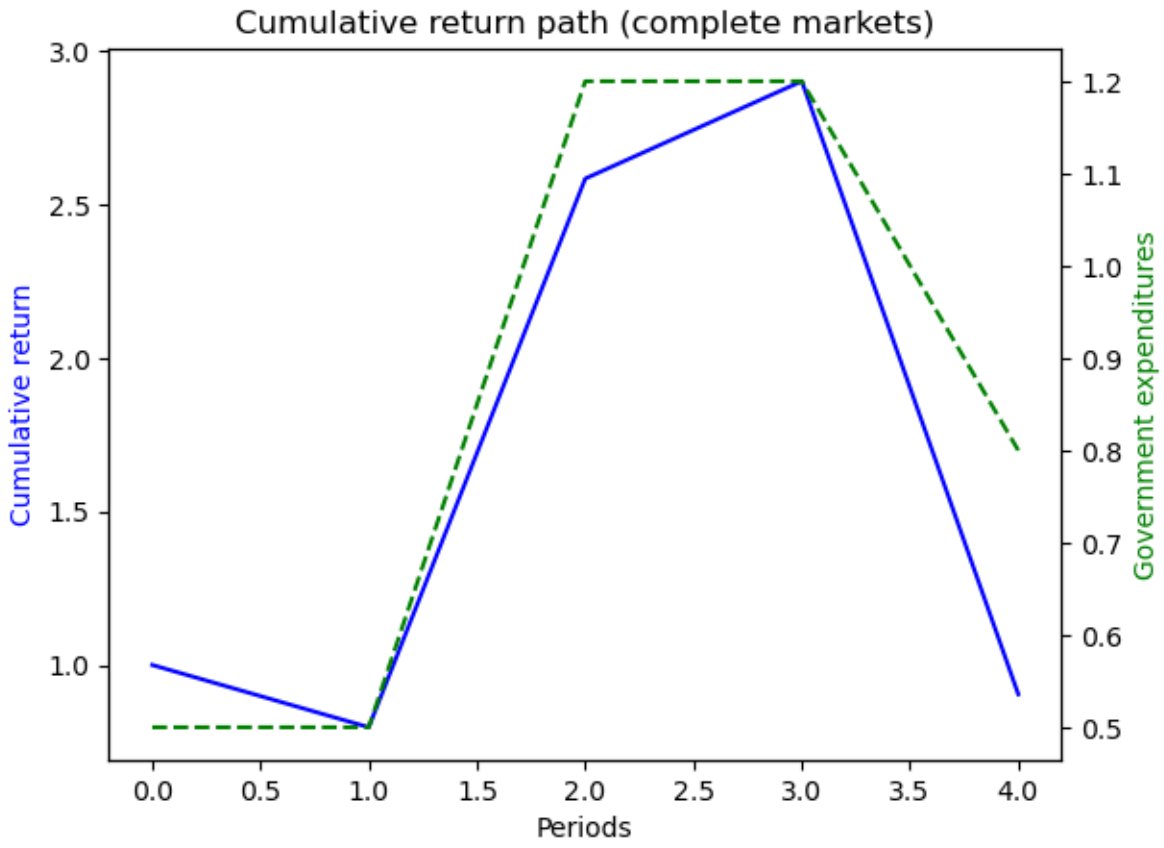
The Ex-post one-period gross return on the portfolio of government assets
[[0.7969336  3.24426428 0.        ]
 [0.         1.12278592 0.31159337]
 [0.         0.         1.04166667]]

The cumulative return earned from holding 1 unit market portfolio of government␣
 ↪bonds
0.17908622141460231
```

```
# The following shows the use of the wrapper class when a specific state path is given
s_path = [0, 0, 1, 1, 2]
ts_s_path = TaxSmoothingExample(g_ex1, P_ex1, b0_ex1, states_ex1, s_path=s_path)
ts_s_path.display()
```




Government debt paths
1
0
−1
−2
−3
−4
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
Periods
incomplete market
complete market
govt expenditures
value of debts today




```
P
 [[0.9 0.1 0. ]
 [0.  0.9 0.1]
 [0.  0.  1. ]]
Q
 [[0.864 0.096 0.   ]
 [0.    0.864 0.096]
 [0.    0.    0.96 ]]
Govt expenditures in peace, war, postwar = [0.5 1.2 0.8]
Constant tax collections = 0.7548096885813149
Govt debt in 3 states = [-1.         -4.07093426 -1.12975779]

Government tax collections minus debt levels in peace, war, postwar
  T+b in peace = 1.754809688581315
  T+b in war = 4.825743944636679
  T+b in postwar = 1.8845674740484442

Total government spending in peace, war, postwar
  peace = 1.754809688581315
  war = 4.825743944636679
  postwar = 1.8845674740484442

Let's see ex-post and ex-ante returns on Arrow securities

Ex-post returns to purchase of Arrow securities:
  π(peace|peace) = 1.1574074074074074
  π(war|peace) = 10.416666666666666
```

(continues on next page)

(continued from previous page)

```
  π(war|war) = 1.1574074074074074
  π(postwar|war) = 10.416666666666666
  π(postwar|postwar) = 1.0416666666666667

Ex-ante returns to purchase of Arrow securities = 1.0416666666666667

The Ex-post one-period gross return on the portfolio of government assets
[[0.7969336  3.24426428 0.        ]
 [0.         1.12278592 0.31159337]
 [0.         0.         1.04166667]]

The cumulative return earned from holding 1 unit market portfolio of government␣
 ↪bonds
0.9045311615620277
```

7.4.3 Example 2

This example captures a peace followed by a war, eventually followed by a permanent peace .

Here we set

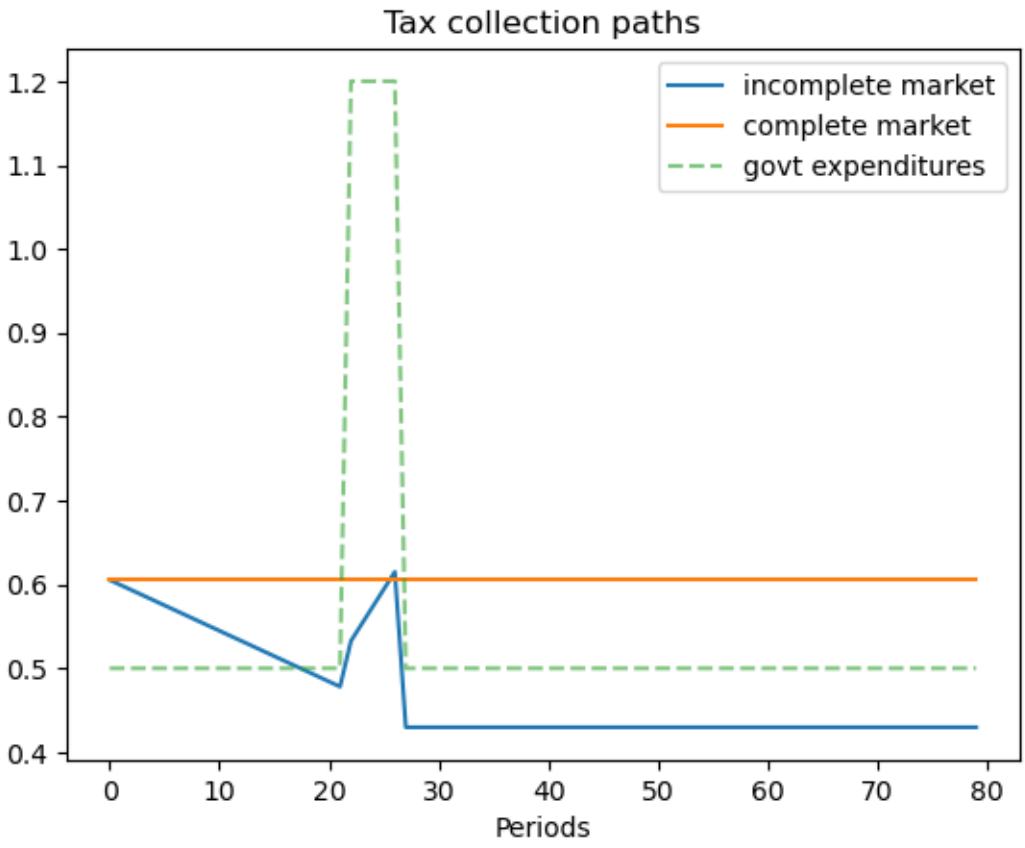
$$P = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1-\gamma & \gamma \\ \phi & 0 & 1-\phi \end{bmatrix}$$

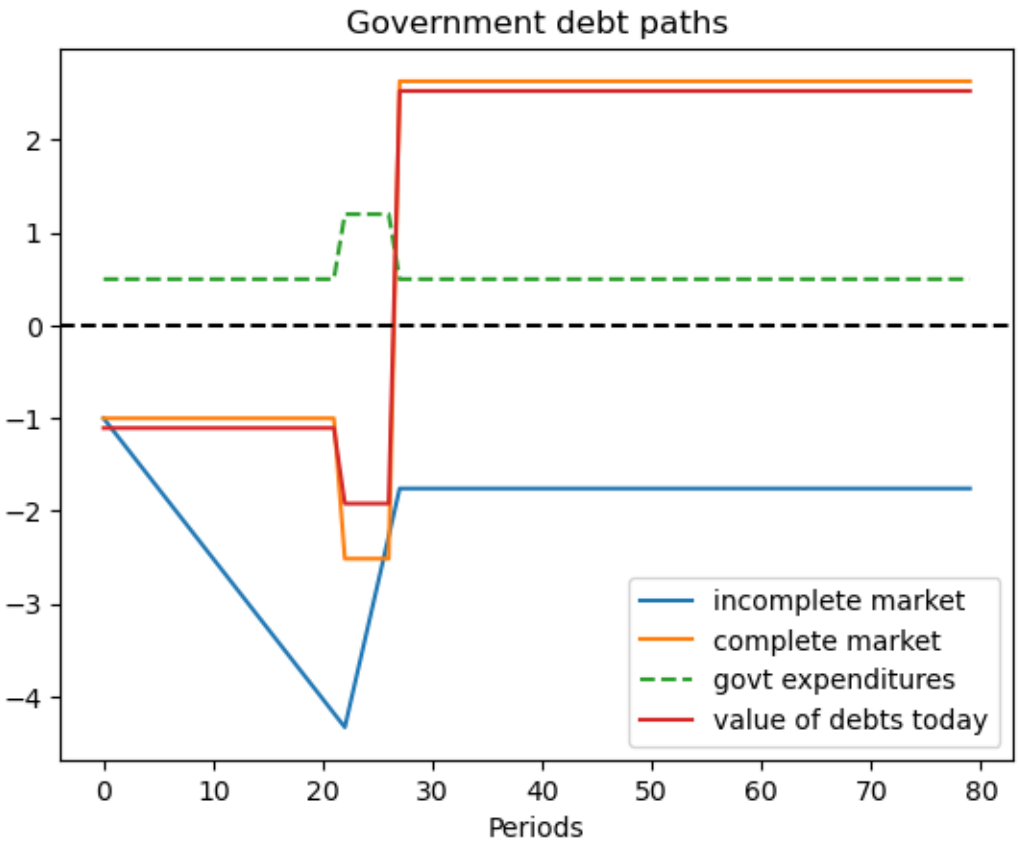
where the government expenditure vector $g = \begin{bmatrix} g_L & g_L & g_H \end{bmatrix}$ and where $g_L < g_H$.

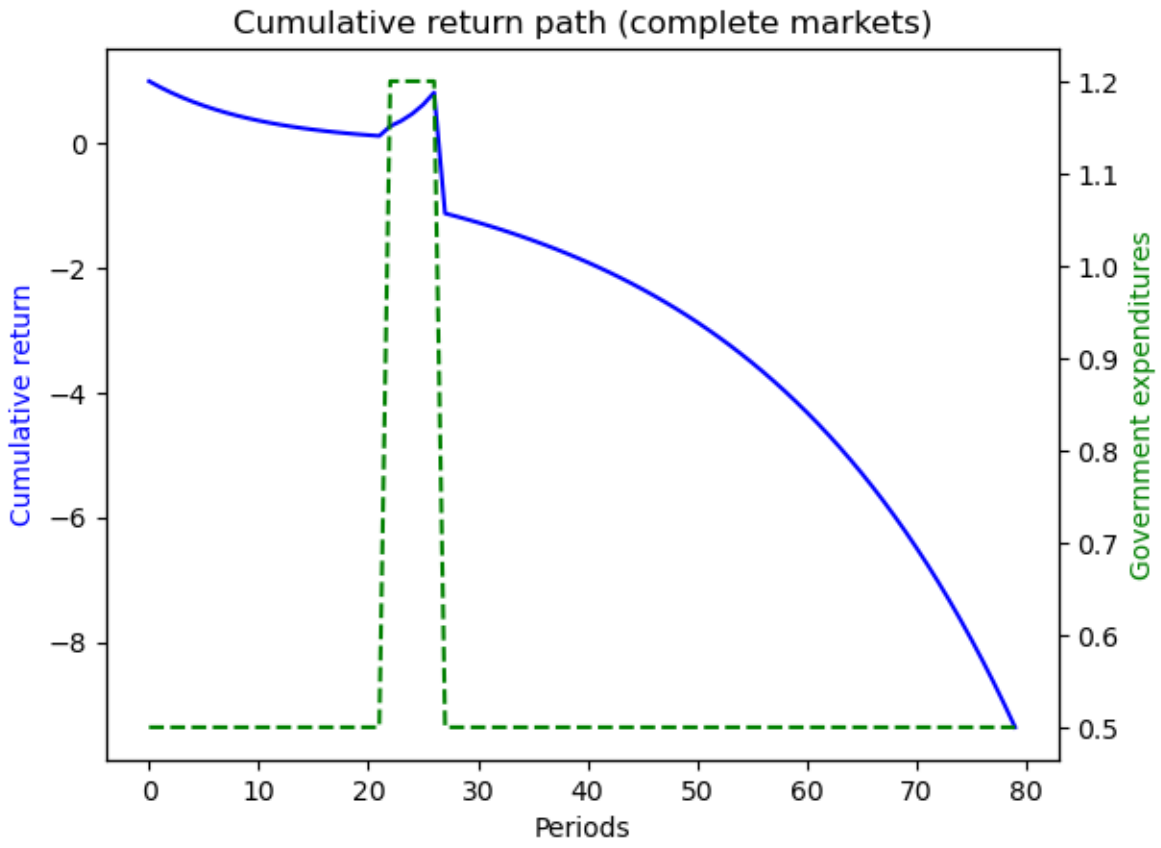
We assume $b_0 = 1$ and that the initial Markov state is state 2 so that the system starts off in a temporary peace.

```
g_ex2 = [g_L, g_L, g_H]
P_ex2 = np.array([[1,   0,    0],
                  [0, 1-γ,    γ],
                  [ϕ,   0, 1-ϕ]])
b0_ex2 = 1
states_ex2 = ['peace', 'temporary peace', 'war']
```

```
ts_ex2 = TaxSmoothingExample(g_ex2, P_ex2, b0_ex2, states_ex2, init=1, random_state=1)
ts_ex2.display()
```


Tax collection paths
incomplete market
complete market
govt expenditures
1.2
1.1
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0
10
20
30
40
50
60
70
80
Periods


Government debt paths
2
1
0
−1
−2
−3
−4
0
10
20
30
40
50
60
70
80
Periods
incomplete market
complete market
govt expenditures
value of debts today




```
P
 [[1.  0.  0. ]
 [0.  0.9 0.1]
 [0.1 0.  0.9]]
Q
 [[0.96  0.    0.   ]
 [0.    0.864 0.096]
 [0.096 0.    0.864]]
Govt expenditures in peace, temporary peace, war = [0.5 0.5 1.2]
Constant tax collections = 0.6053287197231834
Govt debt in 3 states = [ 2.63321799 -1.         -2.51384083]

Government tax collections minus debt levels in peace, temporary peace, war
  T+b in peace = -2.0278892733564
  T+b in temporary peace = 1.6053287197231834
  T+b in war = 3.1191695501730106

Total government spending in peace, temporary peace, war
  peace = -2.0278892733564
  temporary peace = 1.6053287197231834
  war = 3.1191695501730106

Let's see ex-post and ex-ante returns on Arrow securities

Ex-post returns to purchase of Arrow securities:
  π(peace|peace) = 1.0416666666666667
  π(temporary peace|temporary peace) = 1.1574074074074074
```

(continues on next page)

(continued from previous page)

```
  π(war|temporary peace) = 10.416666666666666
  π(peace|war) = 10.416666666666666
  π(war|war) = 1.1574074074074074

Ex-ante returns to purchase of Arrow securities = 1.0416666666666667

The Ex-post one-period gross return on the portfolio of government assets
[[ 1.04166667  0.          0.        ]
 [ 0.          0.90470824  2.27429251]
 [-1.37206116  0.          1.30985865]]

The cumulative return earned from holding 1 unit market portfolio of government␣
 ↪bonds
-9.368991732594216
```

7.4.4 Example 3

This example features a situation in which one of the states is a war state with no hope of peace next period, while another state is a war state with a positive probability of peace next period.

The Markov chain is:

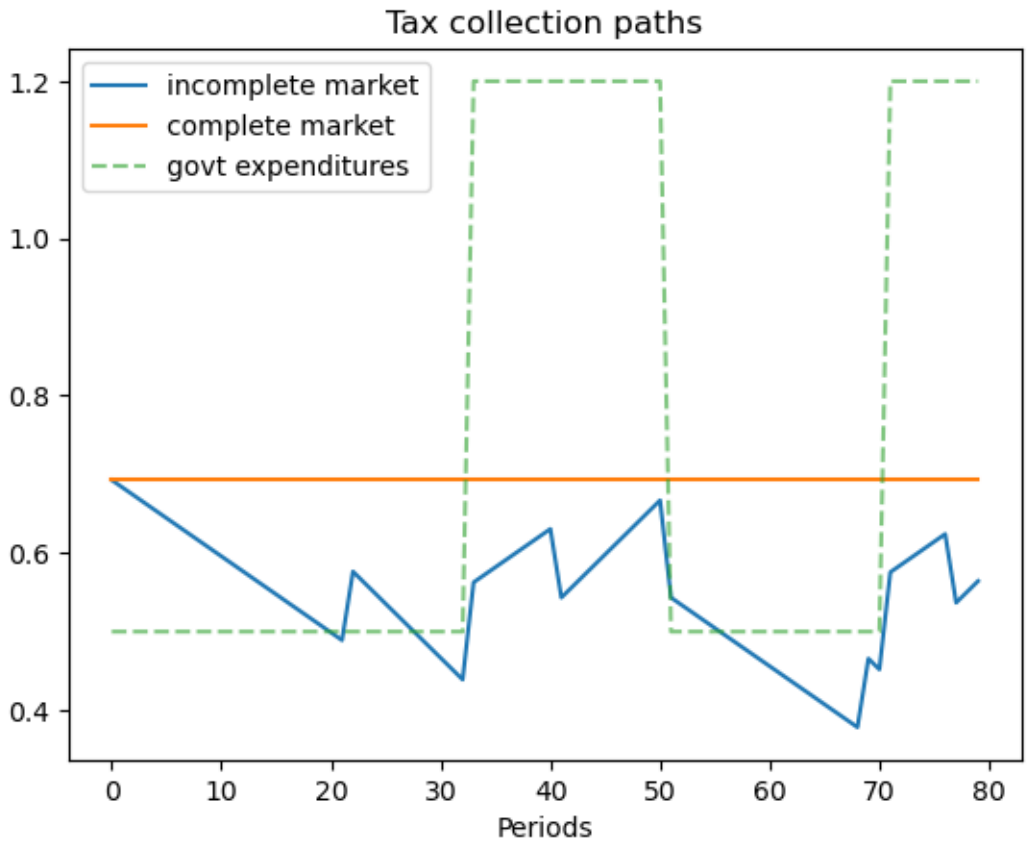
$$P = \begin{bmatrix} 1-\lambda & \lambda & 0 & 0 \\ 0 & 1-\phi & \phi & 0 \\ 0 & 0 & 1-\psi & \psi \\ \theta & 0 & 0 & 1-\theta \end{bmatrix}$$

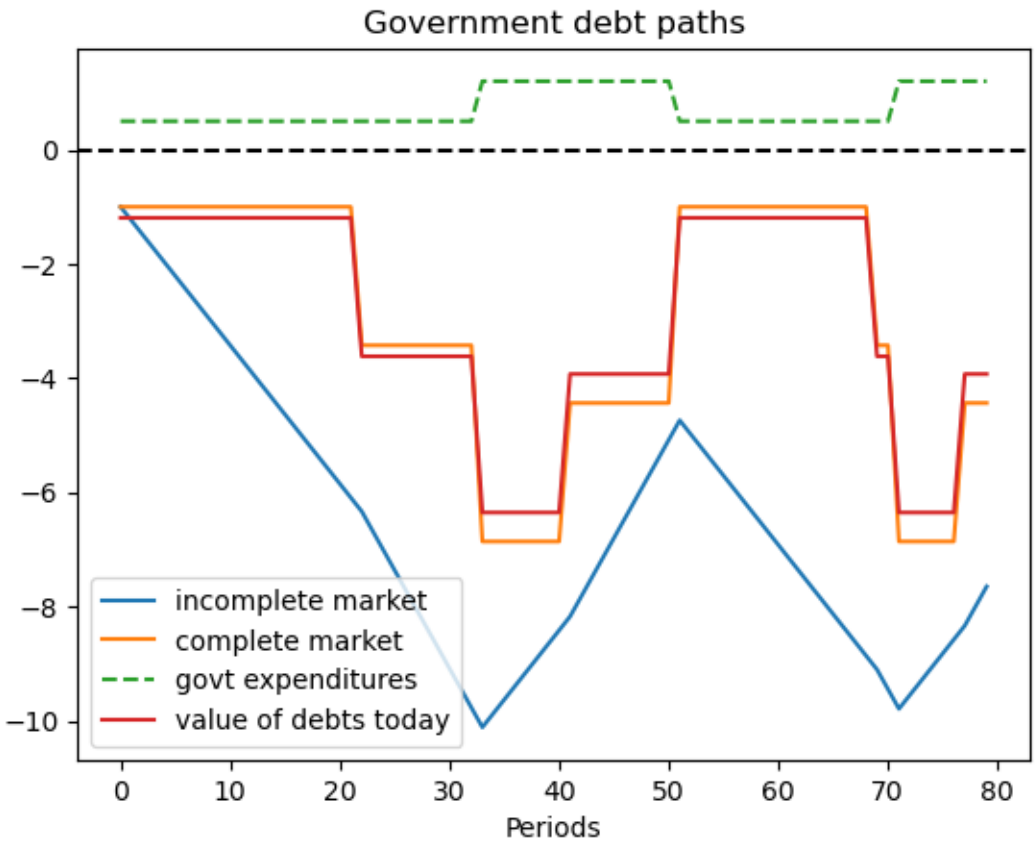
with government expenditure levels for the four states being $\begin{bmatrix} g_L & g_L & g_H & g_H \end{bmatrix}$ where $g_L < g_H$.

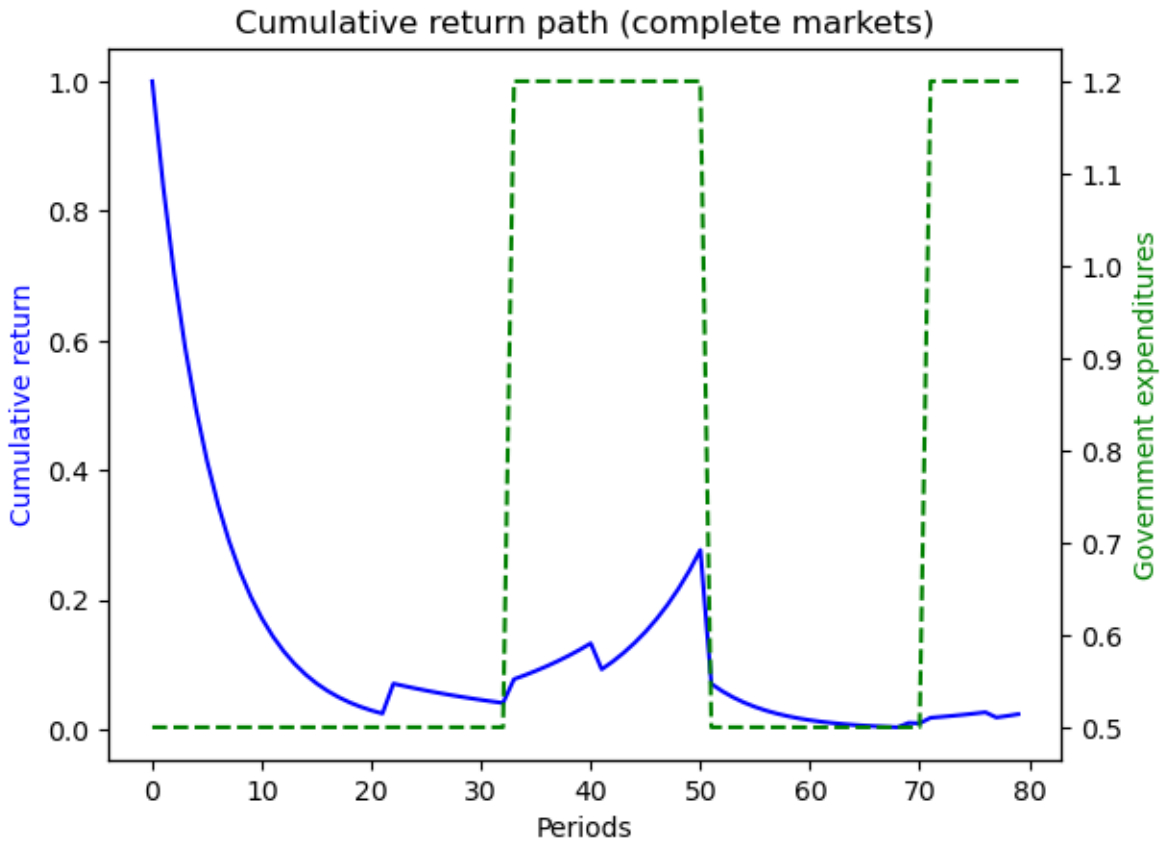
We start with $b_0 = 1$ and $s_0 = 1$.

```
g_ex3 = [g_L, g_L, g_H, g_H]
P_ex3 = np.array([[1-λ,  λ,    0,     0],
                  [0,  1-ϕ,    ϕ,     0],
                  [0,    0,  1-ψ,     ψ],
                  [θ,    0,    0,   1-θ ]])
b0_ex3 = 1
states_ex3 = ['peace1', 'peace2', 'war1', 'war2']
```

```
ts_ex3 = TaxSmoothingExample(g_ex3, P_ex3, b0_ex3, states_ex3, random_state=1)
ts_ex3.display()
```


Tax collection paths
incomplete market
complete market
govt expenditures
1.2
1.0
0.8
0.6
0.4
0
10
20
30
40
50
60
70
80
Periods


Government debt paths
0
−2
−4
−6
−8
−10
0
10
20
30
40
50
60
70
80
Periods
incomplete market
complete market
govt expenditures
value of debts today




```
P
 [[0.9 0.1 0.  0. ]
 [0.  0.9 0.1 0. ]
 [0.  0.  0.9 0.1]
 [0.1 0.  0.  0.9]]
Q
 [[0.864 0.096 0.    0.   ]
 [0.    0.864 0.096 0.   ]
 [0.    0.    0.864 0.096]
 [0.096 0.    0.    0.864]]
Govt expenditures in peace1, peace2, war1, war2 = [0.5 0.5 1.2 1.2]
Constant tax collections = 0.6927944572748268
Govt debt in 4 states = [-1.         -3.42494226 -6.86027714 -4.43533487]

Government tax collections minus debt levels in peace1, peace2, war1, war2
  T+b in peace1 = 1.6927944572748268
  T+b in peace2 = 4.117736720554273
  T+b in war1 = 7.553071593533488
  T+b in war2 = 5.128129330254041

Total government spending in peace1, peace2, war1, war2
  peace1 = 1.6927944572748268
  peace2 = 4.117736720554273
  war1 = 7.553071593533487
  war2 = 5.128129330254041

Let's see ex-post and ex-ante returns on Arrow securities
```

(continues on next page)

(continued from previous page)

```
Ex-post returns to purchase of Arrow securities:
  π(peace1|peace1) = 1.1574074074074074
  π(peace2|peace1) = 10.416666666666666
  π(peace2|peace2) = 1.1574074074074074
  π(war1|peace2) = 10.416666666666666
  π(war1|war1) = 1.1574074074074074
  π(war2|war1) = 10.416666666666666
  π(peace1|war2) = 10.416666666666666
  π(war2|war2) = 1.1574074074074074

Ex-ante returns to purchase of Arrow securities = 1.0416666666666667

The Ex-post one-period gross return on the portfolio of government assets
[[0.83836741 2.87135998 0.         0.        ]
 [0.         0.94670854 1.89628977 0.        ]
 [0.         0.         1.07983627 0.69814023]
 [0.2545741  0.         0.         1.1291214 ]]

The cumulative return earned from holding 1 unit market portfolio of government␣
 ↪bonds
0.02371440178864222
```

7.4.5 Example 4

Here the Markov chain is:

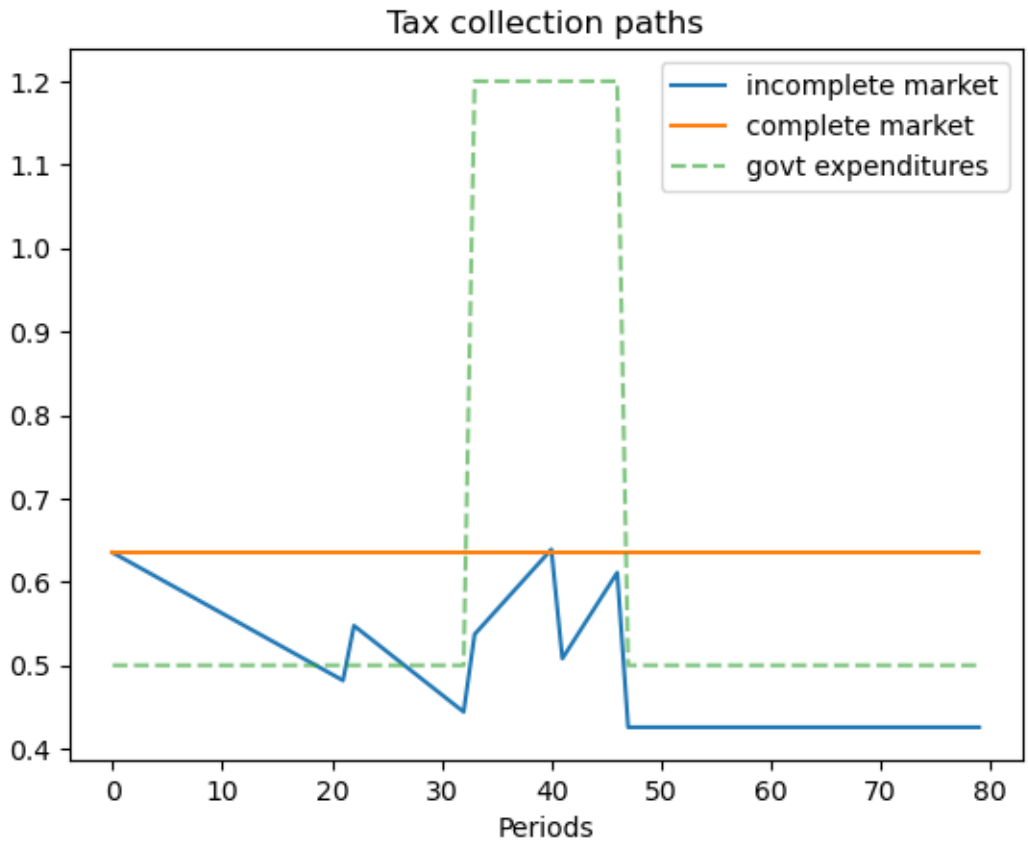
$$P = \begin{bmatrix} 1-\lambda & \lambda & 0 & 0 & 0 \\ 0 & 1-\phi & \phi & 0 & 0 \\ 0 & 0 & 1-\psi & \psi & 0 \\ 0 & 0 & 0 & 1-\theta & \theta \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

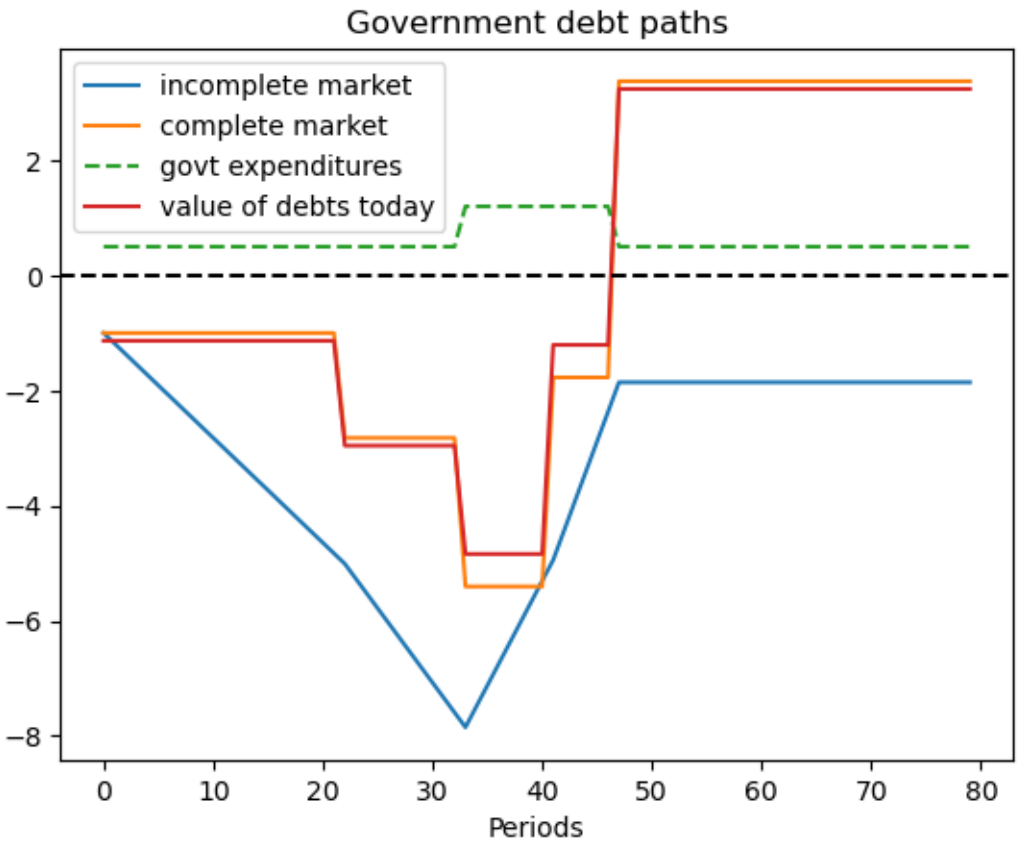
with government expenditure levels for the five states being $\begin{bmatrix} g_L & g_L & g_H & g_H & g_L \end{bmatrix}$ where $g_L < g_H$.

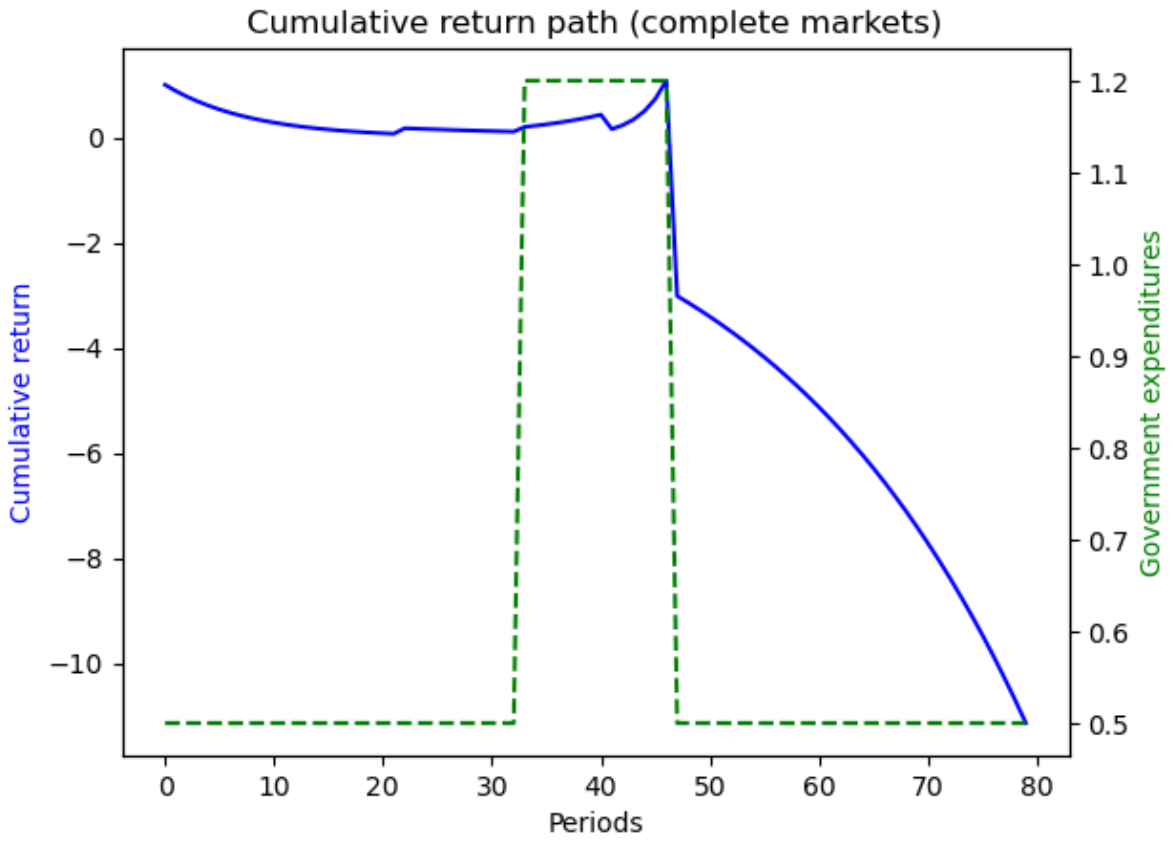
We ssume that $b_0 = 1$ and $s_0 = 1$.

```
g_ex4 = [g_L, g_L, g_H, g_H, g_L]
P_ex4 = np.array([[1-λ,  λ,    0,     0,    0],
                  [0,  1-ϕ,    ϕ,     0,    0],
                  [0,    0,  1-ψ,     ψ,    0],
                  [0,    0,    0,   1-θ,    θ],
                  [0,    0,    0,     0,    1]])
b0_ex4 = 1
states_ex4 = ['peace1', 'peace2', 'war1', 'war2', 'permanent peace']
```

```
ts_ex4 = TaxSmoothingExample(g_ex4, P_ex4, b0_ex4, states_ex4, random_state=1)
ts_ex4.display()
```


Tax collection paths
incomplete market
complete market
govt expenditures
1.2
1.1
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0
10
20
30
40
50
60
70
80
Periods


Government debt paths
incomplete market
complete market
govt expenditures
value of debts today
2
0
−2
−4
−6
−8
0
10
20
30
40
50
60
70
80
Periods




```
P
 [[0.9 0.1 0.  0.  0. ]
 [0.  0.9 0.1 0.  0. ]
 [0.  0.  0.9 0.1 0. ]
 [0.  0.  0.  0.9 0.1]
 [0.  0.  0.  0.  1. ]]
Q
 [[0.864 0.096 0.    0.    0.   ]
 [0.    0.864 0.096 0.    0.   ]
 [0.    0.    0.864 0.096 0.   ]
 [0.    0.    0.    0.864 0.096]
 [0.    0.    0.    0.    0.96 ]]
Govt expenditures in peace1, peace2, war1, war2, permanent peace = [0.5 0.5 1.2 1.
↪2 0.5]
Constant tax collections = 0.6349979047185738
Govt debt in 5 states = [-1.         -2.82289484 -5.4053292  -1.77211121  3.
↪37494762]

Government tax collections minus debt levels in peace1, peace2, war1, war2,␣
↪permanent peace
  T+b in peace1 = 1.6349979047185736
  T+b in peace2 = 3.4578927455370505
  T+b in war1 = 6.040327103363229
  T+b in war2 = 2.4071091102836433
  T+b in permanent peace = -2.7399497132457697

Total government spending in peace1, peace2, war1, war2, permanent peace
```

(continues on next page)

(continued from previous page)

```
  peace1 = 1.6349979047185736
  peace2 = 3.457892745537051
  war1 = 6.040327103363228
  war2 = 2.407109110283643
  permanent peace = -2.7399497132457697

Let's see ex-post and ex-ante returns on Arrow securities

Ex-post returns to purchase of Arrow securities:
  π(peace1|peace1) = 1.1574074074074074
  π(peace2|peace1) = 10.416666666666666
  π(peace2|peace2) = 1.1574074074074074
  π(war1|peace2) = 10.416666666666666
  π(war1|war1) = 1.1574074074074074
  π(war2|war1) = 10.416666666666666
  π(war2|war2) = 1.1574074074074074
  π(permanent peace|war2) = 10.416666666666666
  π(permanent peace|permanent peace) = 1.0416666666666667

Ex-ante returns to purchase of Arrow securities = 1.0416666666666667

The Ex-post one-period gross return on the portfolio of government assets
[[ 0.8810589   2.48713661  0.          0.          0.        ]
 [ 0.          0.95436011  1.82742569  0.          0.        ]
 [ 0.          0.          1.11672808  0.36611394  0.        ]
 [ 0.          0.          0.          1.46806216 -2.79589276]
 [ 0.          0.          0.          0.          1.04166667]]

The cumulative return earned from holding 1 unit market portfolio of government␣
↪bonds
-11.132109773063616
```

7.4.6 Example 5

The example captures a case when the system follows a deterministic path from peace to war, and back to peace again.

Since there is no randomness, the outcomes in complete markets setting should be the same as in incomplete markets setting.

The Markov chain is:

$$P = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

with government expenditure levels for the seven states being $\begin{bmatrix} g_L & g_L & g_H & g_H & g_H & g_H & g_L \end{bmatrix}$ where $g_L < g_H$. Assume $b_0 = 1$ and $s_0 = 1$.

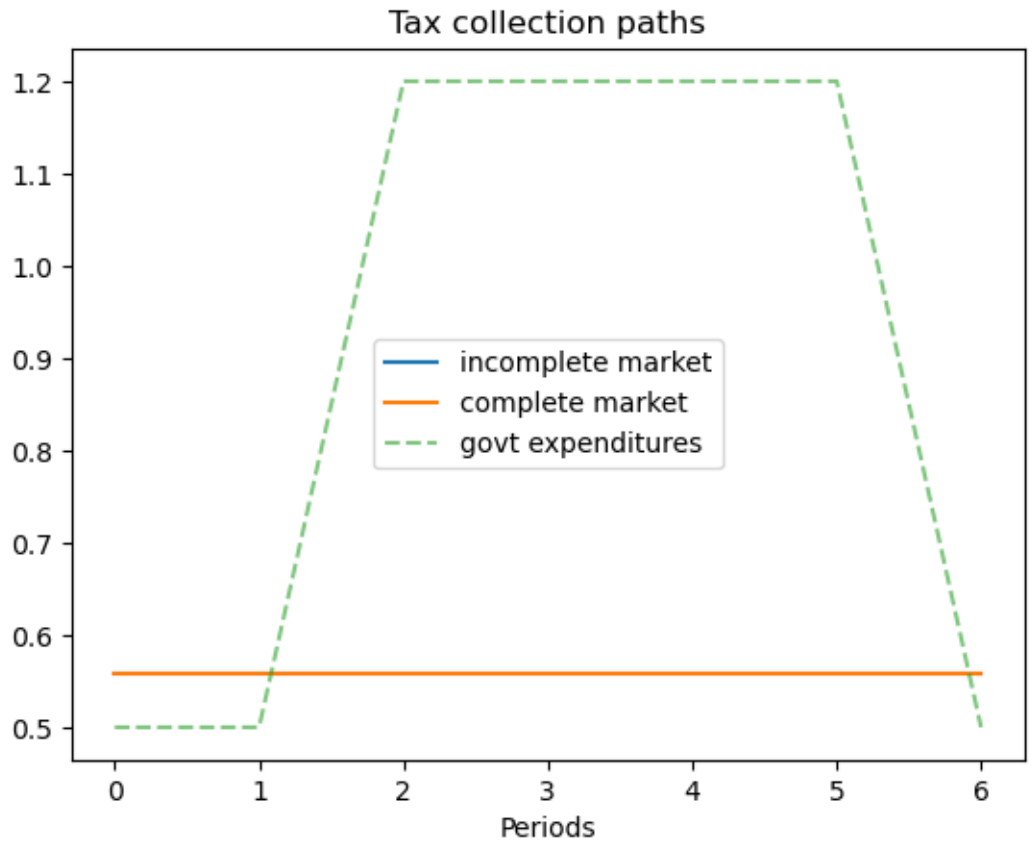
```
g_ex5 = [g_L, g_L, g_H, g_H, g_H, g_H, g_L]
P_ex5 = np.array([[0, 1, 0, 0, 0, 0, 0],
```

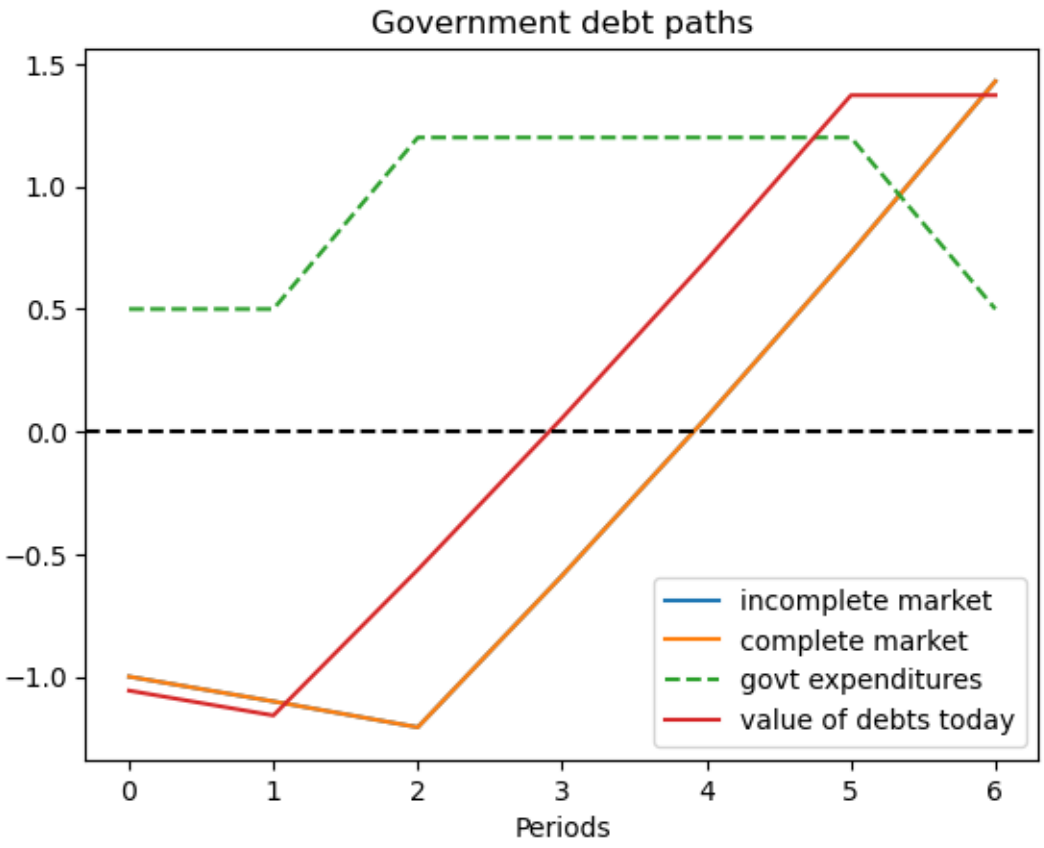
(continues on next page)

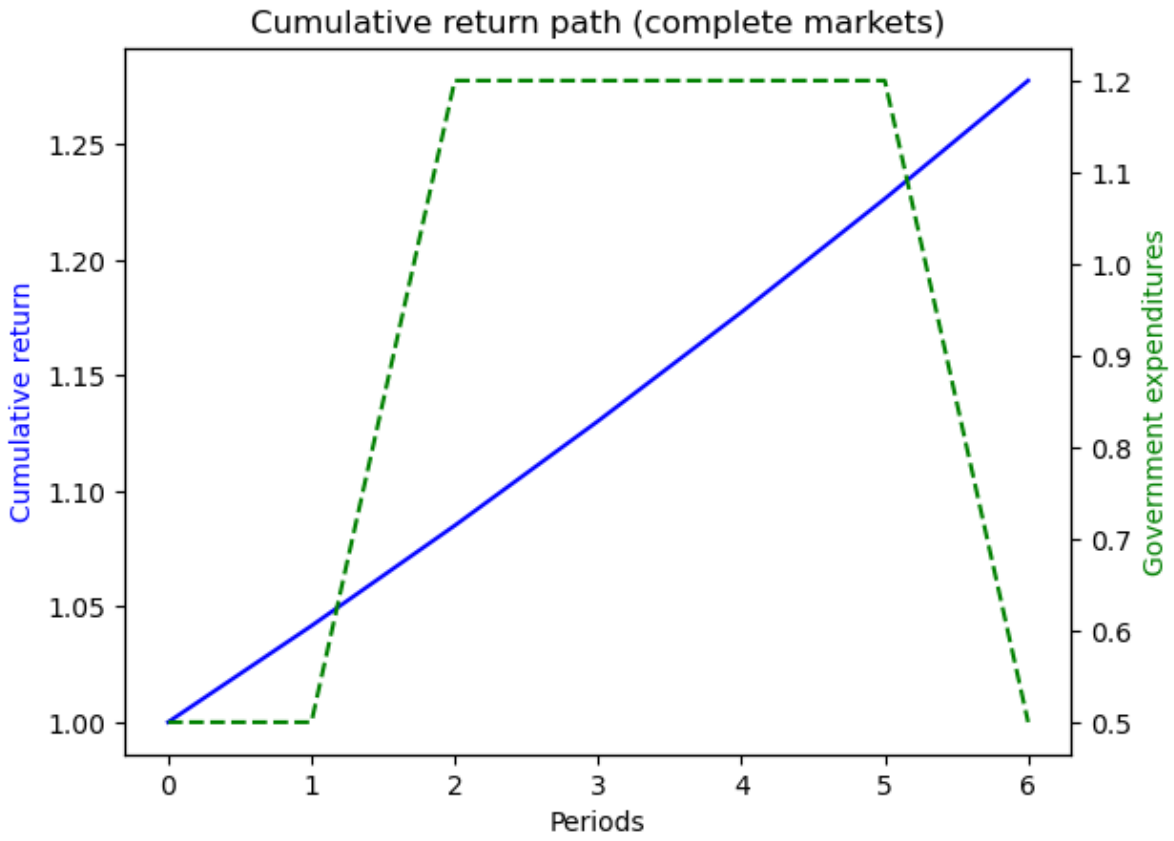
(continued from previous page)

```
                    [0, 0, 1, 0, 0, 0, 0],
                    [0, 0, 0, 1, 0, 0, 0],
                    [0, 0, 0, 0, 1, 0, 0],
                    [0, 0, 0, 0, 0, 1, 0],
                    [0, 0, 0, 0, 0, 0, 1],
                    [0, 0, 0, 0, 0, 0, 1]])
b0_ex5 = 1
states_ex5 = ['peace1', 'peace2', 'war1', 'war2', 'war3', 'permanent peace']
```

```
ts_ex5 = TaxSmoothingExample(g_ex5, P_ex5, b0_ex5, states_ex5, N_simul=7, random_
↪state=1)
ts_ex5.display()
```




Government debt paths
1.5
1.0
0.5
0.0
−0.5
−1.0
0
1
2
3
4
5
6
Periods
incomplete market
complete market
govt expenditures
value of debts today




```
P
 [[0 1 0 0 0 0 0]
 [0 0 1 0 0 0 0]
 [0 0 0 1 0 0 0]
 [0 0 0 0 1 0 0]
 [0 0 0 0 0 1 0]
 [0 0 0 0 0 0 1]
 [0 0 0 0 0 0 1]]
Q
 [[0.   0.96 0.   0.   0.   0.   ]
 [0.   0.   0.96 0.   0.   0.   0.  ]
 [0.   0.   0.   0.96 0.   0.   0.  ]
 [0.   0.   0.   0.   0.96 0.   0.  ]
 [0.   0.   0.   0.   0.   0.96 0.  ]
 [0.   0.   0.   0.   0.   0.   0.96]
 [0.   0.   0.   0.   0.   0.   0.96]]
Govt expenditures in peace1, peace2, war1, war2, war3, permanent peace = [0.5 0.5␣
↪1.2 1.2 1.2 1.2 0.5]
Constant tax collections = 0.5571895472128001
Govt debt in 6 states = [-1.         -1.10123911 -1.20669652 -0.58738132  0.
↪05773868  0.72973868
  1.42973868]

Government tax collections minus debt levels in peace1, peace2, war1, war2, war3,␣
↪permanent peace
  T+b in peace1 = 1.5571895472128001
  T+b in peace2 = 1.6584286588928001
```

(continues on next page)

(continued from previous page)

```
  T+b in war1 = 1.7638860668928005
  T+b in war2 = 1.1445708668928003
  T+b in war3 = 0.4994508668928004
  T+b in permanent peace = -0.17254913310719955

Total government spending in peace1, peace2, war1, war2, war3, permanent peace
  peace1 = 1.5571895472128
  peace2 = 1.6584286588928003
  war1 = 1.7638860668928
  war2 = 1.1445708668928003
  war3 = 0.49945086689280027
  permanent peace = -0.17254913310719933

Let's see ex-post and ex-ante returns on Arrow securities

Ex-post returns to purchase of Arrow securities:
  π(peace2|peace1) = 1.0416666666666667
  π(war1|peace2) = 1.0416666666666667
  π(war2|war1) = 1.0416666666666667
  π(war3|war2) = 1.0416666666666667
  π(permanent peace|war3) = 1.0416666666666667

Ex-ante returns to purchase of Arrow securities = 1.0416666666666667

The Ex-post one-period gross return on the portfolio of government assets
[[0.         1.04166667 0.         0.         0.         0.
  0.        ]
 [0.         0.         1.04166667 0.         0.         0.
  0.        ]
 [0.         0.         0.         1.04166667 0.         0.
  0.        ]
 [0.         0.         0.         0.         1.04166667 0.
  0.        ]
 [0.         0.         0.         0.         0.         1.04166667
  0.        ]
 [0.         0.         0.         0.         0.         0.
  1.04166667]
 [0.         0.         0.         0.         0.         0.
  1.04166667]]

The cumulative return earned from holding 1 unit market portfolio of government␣
 ↪bonds
1.2775343959060068
```

7.4.7 Continuous-State Gaussian Model

To construct a tax-smoothing version of the complete markets consumption-smoothing model with a continuous state space that we presented in the lecture *consumption smoothing with complete and incomplete markets*, we simply relabel variables.

Thus, a government faces a sequence of budget constraints

$$T_t + b_t = g_t + \beta \mathbb{E}_t b_{t+1}, \quad t \geq 0$$

where T_t is tax revenues, b_t are receipts at t from contingent claims that the government had *purchased* at time $t-1$, and

$$\beta \mathbb{E}_t b_{t+1} \equiv \int q_{t+1}(x_{t+1}|x_t) b_{t+1}(x_{t+1}) dx_{t+1}$$

is the value of time $t+1$ state-contingent claims purchased by the government at time t.

As above with the consumption-smoothing model, we can solve the time t budget constraint forward to obtain

$$b_t = \mathbb{E}_t \sum_{j=0}^{\infty} \beta^j (g_{t+j} - T_{t+j})$$

which can be rearranged to become

$$\mathbb{E}_t \sum_{j=0}^{\infty} \beta^j g_{t+j} = b_t + \mathbb{E}_t \sum_{j=0}^{\infty} \beta^j T_{t+j}$$

which states that the present value of government purchases equals the value of government assets at t plus the present value of tax receipts.

With these relabelings, examples presented in *consumption smoothing with complete and incomplete markets* can be interpreted as tax-smoothing models.

Returns: In the continuous state version of our incomplete markets model, the ex post one-period gross rate of return on the government portfolio equals

$$R(x_{t+1}|x_t) = \frac{b(x_{t+1})}{\beta E b(x_{t+1})|x_t}$$

Related Lectures

Throughout this lecture, we have taken one-period interest rates and Arrow security prices as exogenous objects determined outside the model and specified them in ways designed to align our models closely with the consumption smoothing model of Barro [Barro, 1979].

Other lectures make these objects endogenous and describe how a government optimally manipulates prices of government debt, albeit indirectly via effects distorting taxes have on equilibrium prices and allocations.

In *optimal taxation in an LQ economy* and *recursive optimal taxation*, we study **complete-markets** models in which the government recognizes that it can manipulate Arrow securities prices.

Linear-quadratic versions of the Lucas-Stokey tax-smoothing model are described in *Optimal Taxation in an LQ Economy*.

That lecture is a warm-up for the non-linear-quadratic model of tax smoothing described in *Optimal Taxation with State-Contingent Debt*.

In both *Optimal Taxation in an LQ Economy* and *Optimal Taxation with State-Contingent Debt*, the government recognizes that its decisions affect prices.

In *optimal taxation with incomplete markets*, we study an **incomplete-markets** model in which the government also manipulates prices of government debt.

CHAPTER

EIGHT

MARKOV JUMP LINEAR QUADRATIC DYNAMIC PROGRAMMING

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install --upgrade quantecon
```

8.1 Overview

This lecture describes **Markov jump linear quadratic dynamic programming**, an extension of the method described in the first LQ control lecture.

Markov jump linear quadratic dynamic programming is described and analyzed in [Do Val *et al.*, 1999] and the references cited there.

The method has been applied to problems in macroeconomics and monetary economics by [Svensson *et al.*, 2008] and [Svensson and Williams, 2009].

The periodic models of seasonality described in chapter 14 of [Hansen and Sargent, 2013] are a special case of Markov jump linear quadratic problems.

Markov jump linear quadratic dynamic programming combines advantages of

- the computational simplicity of **linear quadratic dynamic programming**, with
- the ability of **finite state Markov chains** to represent interesting patterns of random variation.

The idea is to replace the constant matrices that define a **linear quadratic dynamic programming problem** with N sets of matrices that are fixed functions of the state of an N state Markov chain.

The state of the Markov chain together with the continuous $n \times 1$ state vector x_t form the state of the system.

For the class of infinite horizon problems being studied in this lecture, we obtain N interrelated matrix Riccati equations that determine N optimal value functions and N linear decision rules.

One of these value functions and one of these decision rules apply in each of the N Markov states.

That is, when the Markov state is in state j, the value function and the decision rule for state j prevails.

8.2 Review of useful LQ dynamic programming formulas

To begin, it is handy to have the following reminder in mind.

A **linear quadratic dynamic programming problem** consists of a scalar discount factor $\beta \in (0,1)$, an $n \times 1$ state vector x_t, an initial condition for x_0, a $k \times 1$ control vector u_t, a $p \times 1$ random shock vector w_{t+1} and the following two triples of matrices:

- A triple of matrices (R, Q, W) defining a loss function

$$r(x_t, u_t) = x_t' R x_t + u_t' Q u_t + 2 u_t' W x_t$$

- a triple of matrices (A, B, C) defining a state-transition law

$$x_{t+1} = A x_t + B u_t + C w_{t+1}$$

The problem is

$$-x_0' P x_0 - \rho = \min_{\{u_t\}_{t=0}^{\infty}} E \sum_{t=0}^{\infty} \beta^t r(x_t, u_t)$$

subject to the transition law for the state.

The optimal decision rule has the form

$$u_t = -F x_t$$

and the optimal value function is of the form

$$-(x_t' P x_t + \rho)$$

where P solves the algebraic matrix Riccati equation

$$P = R + \beta A' P A - (\beta B' P A + W)'(Q + \beta B P B)^{-1}(\beta B P A + W)$$

and the constant ρ satisfies

$$\rho = \beta \left(\rho + \text{trace}(P C C')\right)$$

and the matrix F in the decision rule for u_t satisfies

$$F = (Q + \beta B' P B)^{-1}(\beta (B' P A) + W)$$

With the preceding formulas in mind, we are ready to approach Markov Jump linear quadratic dynamic programming.

8.3 Linked Riccati equations for Markov LQ dynamic programming

The key idea is to make the matrices A, B, C, R, Q, W fixed functions of a finite state s that is governed by an N state Markov chain.

This makes decision rules depend on the Markov state, and so fluctuate through time in limited ways.

In particular, we use the following extension of a discrete-time linear quadratic dynamic programming problem.

We let $s_t \in [1, 2, \ldots, N]$ be a time t realization of an N-state Markov chain with transition matrix Π having typical element Π_{ij}.

Here i denotes today and j denotes tomorrow and

$$\Pi_{ij} = \text{Prob}(s_{t+1} = j | s_t = i)$$

We'll switch between labeling today's state as s_t and i and between labeling tomorrow's state as s_{t+1} or j.

The decision-maker solves the minimization problem:

$$\min_{\{u_t\}_{t=0}^{\infty}} E \sum_{t=0}^{\infty} \beta^t r(x_t, s_t, u_t)$$

with

$$r(x_t, s_t, u_t) = x_t' R_{s_t} x_t + u_t' Q_{s_t} u_t + 2 u_t' W_{s_t} x_t$$

subject to linear laws of motion with matrices (A, B, C) each possibly dependent on the Markov-state-s_t:

$$x_{t+1} = A_{s_t} x_t + B_{s_t} u_t + C_{s_t} w_{t+1}$$

where $\{w_{t+1}\}$ is an i.i.d. stochastic process with $w_{t+1} \sim N(0, I)$.

The optimal decision rule for this problem has the form

$$u_t = -F_{s_t} x_t$$

and the optimal value functions are of the form

$$-\left(x_t' P_{s_t} x_t + \rho_{s_t}\right)$$

or equivalently

$$-x_t' P_i x_t - \rho_i$$

The optimal value functions $-x' P_i x - \rho_i$ for $i = 1, \ldots, n$ satisfy the N interrelated Bellman equations

$$-x' P_i x - \rho_i = \max_u -x' R_i x \quad + u' Q_i u + 2 u' W_i x + \beta \sum_j \Pi_{ij} E((A_i x + B_i u + C_i w)' P_j (A_i x + B_i u + C_i w) x + \rho_j)$$

The matrices $P_{s_t} = P_i$ and the scalars $\rho_{s_t} = \rho_i, i = 1, ...,$ n satisfy the following stacked system of **algebraic matrix Riccati** equations:

$$P_i = R_i + \beta \sum_j A_i' P_j A_i \Pi_{ij} - \sum_j \Pi_{ij} [(\beta B_i' P_j A_i + W_i)' (Q + \beta B_i' P_j B_i)^{-1} (\beta B_i' P_j A_i + W_i)]$$

$$\rho_i = \beta \sum_j \Pi_{ij} (\rho_j + \text{trace}(P_j C_i C_i'))$$

and the F_i matrices in the optimal decision rules are

$$F_i = (Q_i + \beta \sum_j \Pi_{ij} B_i' P_j B_i)^{-1} (\beta \sum_j \Pi_{ij} (B_i' P_j A_i) + W_i)$$

8.4 Applications

We now describe Python code and some examples.

To begin, we import these Python modules

```
import numpy as np
import quantecon as qe
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

```
# Set discount factor
β = 0.95
```

8.5 Example 1

This example is a version of a classic problem of optimally adjusting a variable k_t to a target level in the face of costly adjustment.

This provides a model of gradual adjustment.

Given k_0, the objective function is

$$\max_{\{k_t\}_{t=1}^{\infty}} E_0 \sum_{t=0}^{\infty} \beta^t r\left(s_t, k_t\right)$$

where the one-period payoff function is

$$r(s_t, k_t) = f_{1,s_t} k_t - f_{2,s_t} k_t^2 - d_{s_t} (k_{t+1} - k_t)^2,$$

E_0 is a mathematical expectation conditioned on time 0 information x_0, s_0 and the transition law for continuous state variable k_t is

$$k_{t+1} - k_t = u_t$$

We can think of k_t as the decision-maker's capital and u_t as costs of adjusting the level of capital.

We assume that $f_1\left(s_t\right) > 0$, $f_2\left(s_t\right) > 0$, and $d\left(s_t\right) > 0$.

Denote the state transition matrix for Markov state $s_t \in \{1, 2\}$ as Π:

$$\Pr\left(s_{t+1} = j \mid s_t = i\right) = \Pi_{ij}$$

Let $x_t = \begin{bmatrix} k_t \\ 1 \end{bmatrix}$

We can represent the one-period payoff function $r\left(s_t, k_t\right)$ as

$$\begin{aligned} r\left(s_t, k_t\right) &= f_{1,s_t} k_t - f_{2,s_t} k_t^2 - d_{s_t} {u_t}^2 \\ &= -x_t' \underbrace{\begin{bmatrix} f_{2,s_t} & -\frac{f_{1,s_t}}{2} \\ -\frac{f_{1,s_t}}{2} & 0 \end{bmatrix}}_{\equiv R(s_t)} x_t + \underbrace{d_{s_t}}_{\equiv Q(s_t)} {u_t}^2 \end{aligned}$$

and the state-transition law as

$$x_{t+1} = \begin{bmatrix} k_{t+1} \\ 1 \end{bmatrix} = \underbrace{I_2}_{\equiv A(s_t)} x_t + \underbrace{\begin{bmatrix} 1 \\ 0 \end{bmatrix}}_{\equiv B(s_t)} u_t$$

```
def construct_arrays1(f1_vals=[1. ,1.],
                      f2_vals=[1., 1.],
                      d_vals=[1., 1.]):
    """
    Construct matrices that map the problem described in example 1
    into a Markov jump linear quadratic dynamic programming problem
    """

    # Number of Markov states
    m = len(f1_vals)
    # Number of state and control variables
    n, k = 2, 1

    # Construct sets of matrices for each state
    As = [np.eye(n) for i in range(m)]
    Bs = [np.array([[1, 0]]).T for i in range(m)]

    Rs = np.zeros((m, n, n))
    Qs = np.zeros((m, k, k))

    for i in range(m):
        Rs[i, 0, 0] = f2_vals[i]
        Rs[i, 1, 0] = - f1_vals[i] / 2
        Rs[i, 0, 1] = - f1_vals[i] / 2

        Qs[i, 0, 0] = d_vals[i]

    Cs, Ns = None, None

    # Compute the optimal k level of the payoff function in each state
    k_star = np.empty(m)
    for i in range(m):
        k_star[i] = f1_vals[i] / (2 * f2_vals[i])

    return Qs, Rs, Ns, As, Bs, Cs, k_star
```

The continuous part of the state x_t consists of two variables, namely, k_t and a constant term.

```
state_vec1 = ["k", "constant term"]
```

We start with a Markov transition matrix that makes the Markov state be strictly periodic:

$$\Pi_1 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix},$$

We set f_{1,s_t} and f_{2,s_t} to be independent of the Markov state s_t

$$f_{1,1} = f_{1,2} = 1,$$

$$f_{2,1} = f_{2,2} = 1$$

In contrast to f_{1,s_t} and f_{2,s_t}, we make the adjustment cost d_{s_t} vary across Markov states s_t.

We set the adjustment cost to be lower in Markov state 2

$$d_1 = 1, d_2 = 0.5$$

The following code forms a Markov switching LQ problem and computes the optimal value functions and optimal decision rules for each Markov state

```
# Construct Markov transition matrix
Π1 = np.array([[0., 1.],
               [1., 0.]])
```

```
# Construct matrices
Qs, Rs, Ns, As, Bs, Cs, k_star = construct_arrays1(d_vals=[1., 0.5])
```

```
# Construct a Markov Jump LQ problem
ex1_a = qe.LQMarkov(Π1, Qs, Rs, As, Bs, Cs=Cs, Ns=Ns, beta=β)
# Solve for optimal value functions and decision rules
ex1_a.stationary_values();
```

Let's look at the value function matrices and the decision rules for each Markov state

```
# P(s)
ex1_a.Ps
```

```
array([[[ 1.56626026, -0.78313013],
        [-0.78313013, -4.60843493]],

       [[ 1.37424214, -0.68712107],
        [-0.68712107, -4.65643947]]])
```

```
# d(s) = 0, since there is no randomness
ex1_a.ds
```

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

```
# F(s)
ex1_a.Fs
```

```
array([[[ 0.56626026, -0.28313013]],

       [[ 0.74848427, -0.37424214]]])
```

Now we'll plot the decision rules and see if they make sense

```
# Plot the optimal decision rules
k_grid = np.linspace(0., 1., 100)
# Optimal choice in state s1
u1_star = - ex1_a.Fs[0, 0, 1] - ex1_a.Fs[0, 0, 0] * k_grid
# Optimal choice in state s2
u2_star = - ex1_a.Fs[1, 0, 1] - ex1_a.Fs[1, 0, 0] * k_grid

fig, ax = plt.subplots()
ax.plot(k_grid, k_grid + u1_star, label=r"$\overline{s}_1$ (high)")
ax.plot(k_grid, k_grid + u2_star, label=r"$\overline{s}_2$ (low)")

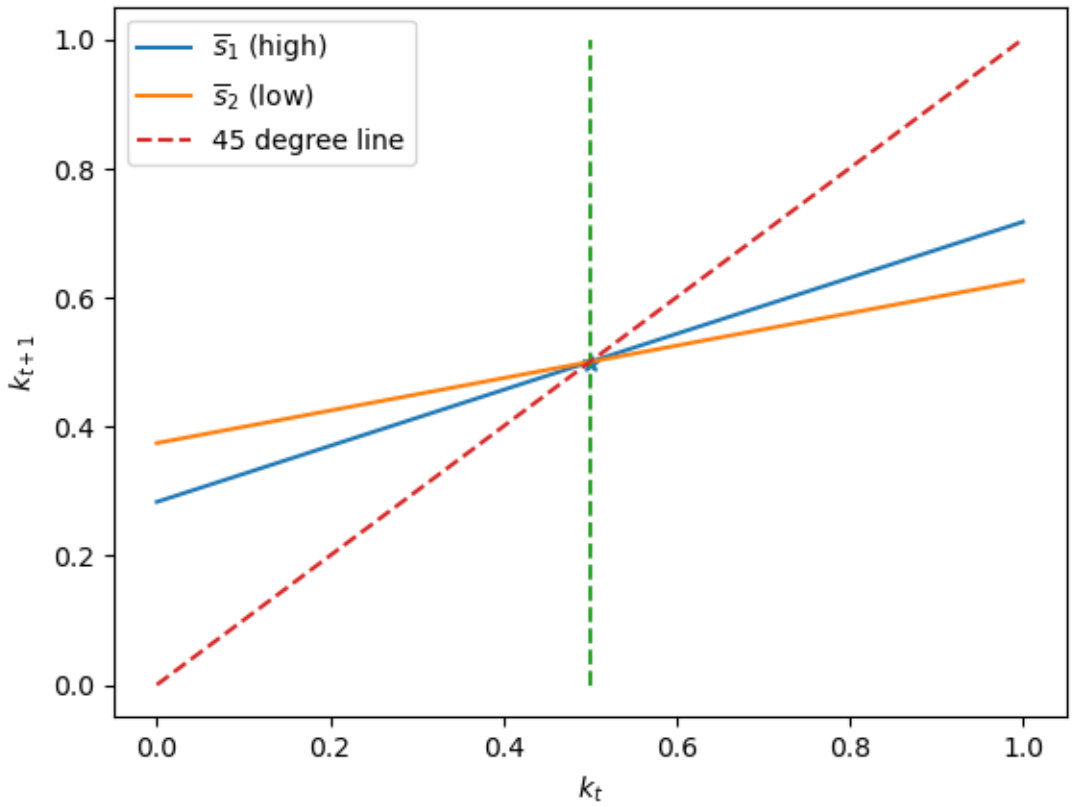
# The optimal k*
ax.scatter([0.5, 0.5], [0.5, 0.5], marker="*")
ax.plot([k_star[0], k_star[0]], [0., 1.0], '--')
```

(continues on next page)

(continued from previous page)

```
# 45 degree line
ax.plot([0., 1.], [0., 1.], '--', label="45 degree line")

ax.set_xlabel("$k_t$")
ax.set_ylabel("$k_{t+1}$")
ax.legend()
plt.show()
```



The above graph plots $k_{t+1} = k_t + u_t = k_t - Fx_t$ as an affine (i.e., linear in k_t plus a constant) function of k_t for both Markov states s_t.

It also plots the 45 degree line.

Notice that the two s_t-dependent *closed loop* functions that determine k_{t+1} as functions of k_t share the same rest point (also called a fixed point) at $k_t = 0.5$.

Evidently, the optimal decision rule in Markov state 2, in which the adjustment cost is lower, makes k_{t+1} a flatter function of k_t in Markov state 2.

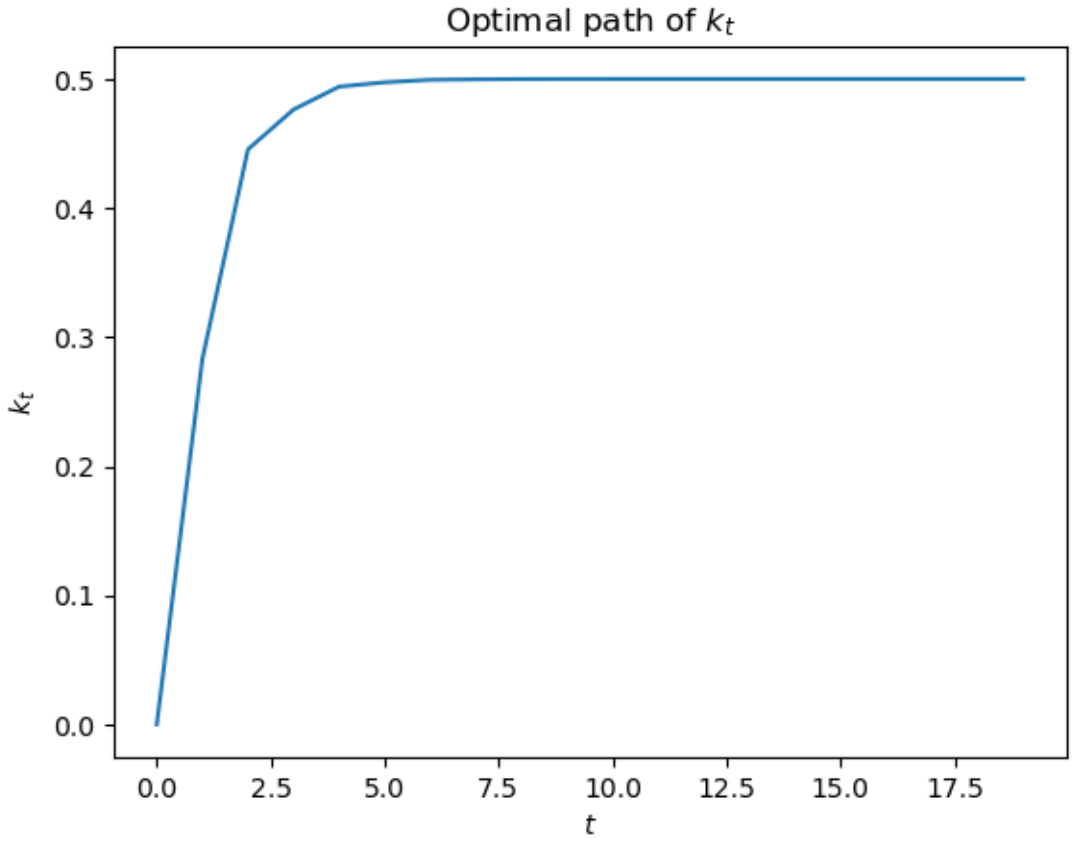
This happens because when k_t is not at its fixed point, $|u_{t,2}| > |u_{t,2}|$, so that the decision-maker adjusts toward the fixed point faster when the Markov state s_t takes a value that makes it cheaper.

```
# Compute time series
T = 20
x0 = np.array([[0., 1.]]).T
x_path = ex1_a.compute_sequence(x0, ts_length=T)[0]
```

(continues on next page)

(continued from previous page)

```
fig, ax = plt.subplots()
ax.plot(range(T), x_path[0, :-1])
ax.set_xlabel("$t$")
ax.set_ylabel("$k_t$")
ax.set_title("Optimal path of $k_t$")
plt.show()
```



Now we'll depart from the preceding transition matrix that made the Markov state be strictly periodic.

We'll begin with symmetric transition matrices of the form

$$\Pi_2 = \begin{bmatrix} 1-\lambda & \lambda \\ \lambda & 1-\lambda \end{bmatrix}.$$

```
λ = 0.8 # high λ
Π2 = np.array([[1-λ, λ],
               [λ, 1-λ]])

ex1_b = qe.LQMarkov(Π2, Qs, Rs, As, Bs, Cs=Cs, Ns=Ns, beta=β)
ex1_b.stationary_values();
ex1_b.Fs
```

```
array([[[ 0.57291724, -0.28645862]],

       [[ 0.74434525, -0.37217263]]])
```

```
λ = 0.2 # low λ
Π2 = np.array([[1-λ, λ],
               [λ, 1-λ]])

ex1_b = qe.LQMarkov(Π2, Qs, Rs, As, Bs, Cs=Cs, Ns=Ns, beta=β)
ex1_b.stationary_values();
ex1_b.Fs
```

```
array([[[ 0.59533259, -0.2976663 ]],

       [[ 0.72818728, -0.36409364]]])
```

We can plot optimal decision rules associated with different λ values.

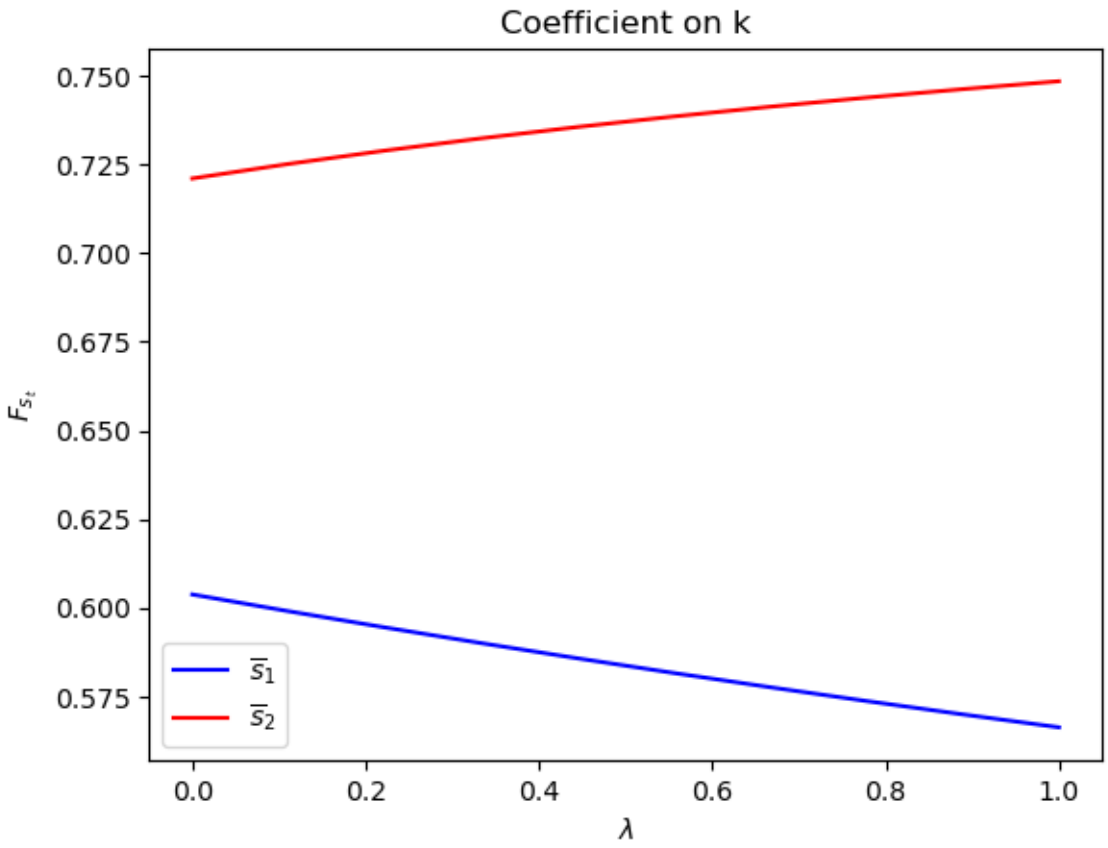
```
λ_vals = np.linspace(0., 1., 10)
F1 = np.empty((λ_vals.size, 2))
F2 = np.empty((λ_vals.size, 2))

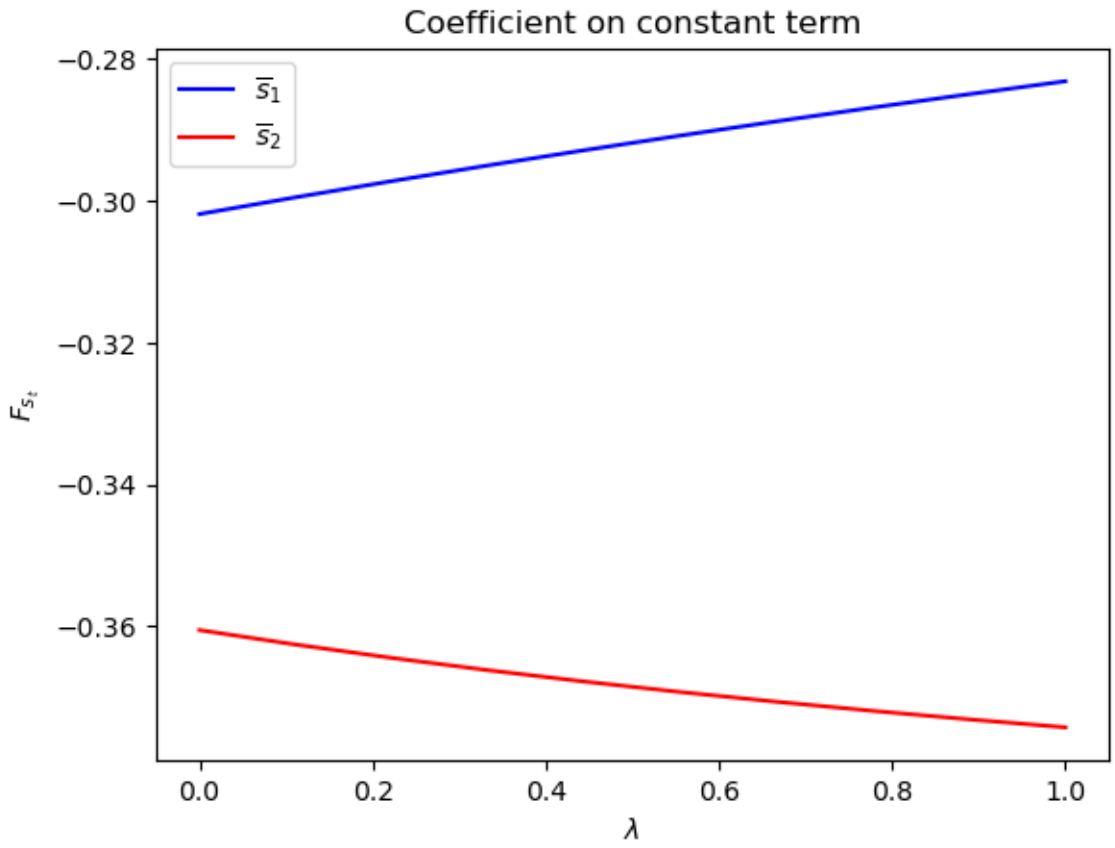
for i, λ in enumerate(λ_vals):
    Π2 = np.array([[1-λ, λ],
                   [λ, 1-λ]])

    ex1_b = qe.LQMarkov(Π2, Qs, Rs, As, Bs, Cs=Cs, Ns=Ns, beta=β)
    ex1_b.stationary_values();
    F1[i, :] = ex1_b.Fs[0, 0, :]
    F2[i, :] = ex1_b.Fs[1, 0, :]
```

```
for i, state_var in enumerate(state_vec1):
    fig, ax = plt.subplots()
    ax.plot(λ_vals, F1[:, i], label=r"$\overline{s}_1$", color="b")
    ax.plot(λ_vals, F2[:, i], label=r"$\overline{s}_2$", color="r")

    ax.set_xlabel(r"$\lambda$")
    ax.set_ylabel("$F_{s_t}$")
    ax.set_title(f"Coefficient on {state_var}")
    ax.legend()
    plt.show()
```


Coefficient on k
0.750
0.725
0.700
0.675
0.650
0.625
0.600
0.575
F_{s_t}
$\overline{s}_1$
$\overline{s}_2$
0.0
0.2
0.4
0.6
0.8
1.0
λ



Notice how the decision rules' constants and slopes behave as functions of λ.

Evidently, as the Markov chain becomes *more nearly periodic* (i.e., as $\lambda \to 1$), the dynamic program adjusts capital faster in the low adjustment cost Markov state to take advantage of what is only temporarily a more favorable time to invest.

Now let's study situations in which the Markov transition matrix Π is asymmetric

$$\Pi_3 = \begin{bmatrix} 1-\lambda & \lambda \\ \delta & 1-\delta \end{bmatrix}.$$

```
λ, δ = 0.8, 0.2
Π3 = np.array([[1-λ, λ],
               [δ, 1-δ]])

ex1_b = qe.LQMarkov(Π3, Qs, Rs, As, Bs, Cs=Cs, Ns=Ns, beta=β)
ex1_b.stationary_values();
ex1_b.Fs
```

```
array([[[ 0.57169781, -0.2858489 ]],

       [[ 0.72749075, -0.36374537]]])
```

We can plot optimal decision rules for different λ and δ values.

```
λ_vals = np.linspace(0., 1., 10)
δ_vals = np.linspace(0., 1., 10)
```

(continues on next page)

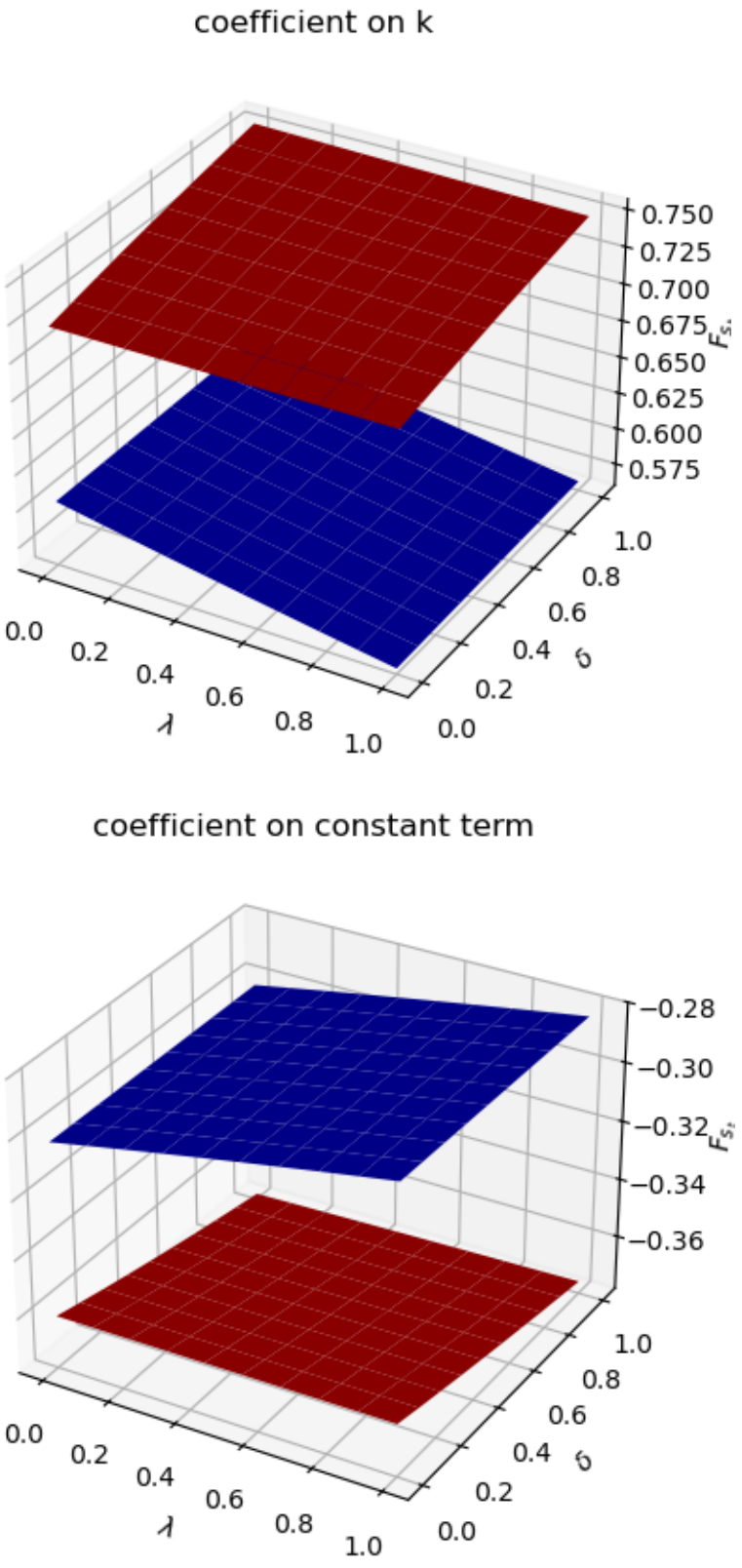
(continued from previous page)

```
λ_grid = np.empty((λ_vals.size, δ_vals.size))
δ_grid = np.empty((λ_vals.size, δ_vals.size))
F1_grid = np.empty((λ_vals.size, δ_vals.size, len(state_vec1)))
F2_grid = np.empty((λ_vals.size, δ_vals.size, len(state_vec1)))

for i, λ in enumerate(λ_vals):
    λ_grid[i, :] = λ
    δ_grid[i, :] = δ_vals
    for j, δ in enumerate(δ_vals):
        Π3 = np.array([[1-λ, λ],
                       [δ, 1-δ]])

        ex1_b = qe.LQMarkov(Π3, Qs, Rs, As, Bs, Cs=Cs, Ns=Ns, beta=β)
        ex1_b.stationary_values();
        F1_grid[i, j, :] = ex1_b.Fs[0, 0, :]
        F2_grid[i, j, :] = ex1_b.Fs[1, 0, :]
```

```
for i, state_var in enumerate(state_vec1):
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    # high adjustment cost, blue surface
    ax.plot_surface(λ_grid, δ_grid, F1_grid[:, :, i], color="b")
    # low adjustment cost, red surface
    ax.plot_surface(λ_grid, δ_grid, F2_grid[:, :, i], color="r")
    ax.set_xlabel(r"$\lambda$")
    ax.set_ylabel(r"$\delta$")
    ax.set_zlabel("$F_{s_t}$")
    ax.set_title(f"coefficient on {state_var}")
    plt.show()
```



The following code defines a wrapper function that computes optimal decision rules for cases with different Markov transition matrices

```
def run(construct_func, vals_dict, state_vec):
    """
    A Wrapper function that repeats the computation above
    for different cases
    """

    Qs, Rs, Ns, As, Bs, Cs, k_star = construct_func(**vals_dict)

    # Symmetric Π
    # Notice that pure periodic transition is a special case
    # when λ=1
    print("symmetric Π case:\n")
    λ_vals = np.linspace(0., 1., 10)
    F1 = np.empty((λ_vals.size, len(state_vec)))
    F2 = np.empty((λ_vals.size, len(state_vec)))

    for i, λ in enumerate(λ_vals):
        Π2 = np.array([[1-λ, λ],
                       [λ, 1-λ]])

        mplq = qe.LQMarkov(Π2, Qs, Rs, As, Bs, Cs=Cs, Ns=Ns, beta=β)
        mplq.stationary_values();
        F1[i, :] = mplq.Fs[0, 0, :]
        F2[i, :] = mplq.Fs[1, 0, :]

    for i, state_var in enumerate(state_vec):
        fig = plt.figure()
        ax = fig.add_subplot(111)
        ax.plot(λ_vals, F1[:, i], label=r"$\overline{s}_1$", color="b")
        ax.plot(λ_vals, F2[:, i], label=r"$\overline{s}_2$", color="r")

        ax.set_xlabel(r"$\lambda$")
        ax.set_ylabel(r"$F(\overline{s}_t)$")
        ax.set_title(f"coefficient on {state_var}")
        ax.legend()
        plt.show()

    # Plot optimal k*_{s_t} and k that optimal policies are targeting
    # only for example 1
    if state_vec == ["k", "constant term"]:
        fig = plt.figure()
        ax = fig.add_subplot(111)
        for i in range(2):
            F = [F1, F2][i]
            c = ["b", "r"][i]
            ax.plot([0, 1], [k_star[i], k_star[i]], "--",
                    color=c, label=r"$k^*(\overline{s}_"+str(i+1)+")$")
            ax.plot(λ_vals, - F[:, 1] / F[:, 0], color=c,
                    label=r"$k^{target}(\overline{s}_"+str(i+1)+")$")

        # Plot a vertical line at λ=0.5
        ax.plot([0.5, 0.5], [min(k_star), max(k_star)], "-.")

        ax.set_xlabel(r"$\lambda$")
        ax.set_ylabel("$k$")
        ax.set_title("Optimal k levels and k targets")
        ax.text(0.5, min(k_star)+(max(k_star)-min(k_star))/20, r"$\lambda=0.5$")
```

(continues on next page)

(continued from previous page)

```
        ax.legend(bbox_to_anchor=(1., 1.))
        plt.show()

    # Asymmetric Π
    print("asymmetric Π case:\n")
    δ_vals = np.linspace(0., 1., 10)

    λ_grid = np.empty((λ_vals.size, δ_vals.size))
    δ_grid = np.empty((λ_vals.size, δ_vals.size))
    F1_grid = np.empty((λ_vals.size, δ_vals.size, len(state_vec)))
    F2_grid = np.empty((λ_vals.size, δ_vals.size, len(state_vec)))

    for i, λ in enumerate(λ_vals):
        λ_grid[i, :] = λ
        δ_grid[i, :] = δ_vals
        for j, δ in enumerate(δ_vals):
            Π3 = np.array([[1-λ, λ],
                           [δ, 1-δ]])

            mplq = qe.LQMarkov(Π3, Qs, Rs, As, Bs, Cs=Cs, Ns=Ns, beta=β)
            mplq.stationary_values();
            F1_grid[i, j, :] = mplq.Fs[0, 0, :]
            F2_grid[i, j, :] = mplq.Fs[1, 0, :]

    for i, state_var in enumerate(state_vec):
        fig = plt.figure()
        ax = fig.add_subplot(111, projection='3d')
        ax.plot_surface(λ_grid, δ_grid, F1_grid[:, :, i], color="b")
        ax.plot_surface(λ_grid, δ_grid, F2_grid[:, :, i], color="r")
        ax.set_xlabel(r"$\lambda$")
        ax.set_ylabel(r"$\delta$")
        ax.set_zlabel(r"$F(\overline{s}_t)$")
        ax.set_title(f"coefficient on {state_var}")
        plt.show()
```

To illustrate the code with another example, we shall set f_{2,s_t} and d_{s_t} as constant functions and

$$f_{1,1} = 0.5, f_{1,2} = 1$$

Thus, the sole role of the Markov jump state s_t is to identify times in which capital is very productive and other times in which it is less productive.

The example below reveals much about the structure of the optimum problem and optimal policies.

Only f_{1,s_t} varies with s_t.

So there are different s_t-dependent optimal static k level in different states $k^*_{s_t} = \frac{f_{1,s_t}}{2f_{2,s_t}}$, values of k that maximize one-period payoff functions in each state.

We denote a target k level as $k^{target}_{s_t}$, the fixed point of the optimal policies in each state, given the value of λ.

We call $k^{target}_{s_t}$ a "target" because in each Markov state s_t, optimal policies are contraction mappings and will push k_t towards a fixed point $k^{target}_{s_t}$.

When $\lambda \to 0$, each Markov state becomes close to absorbing state and consequently $k^{target}_{s_t} \to k^*_{s_t}$.

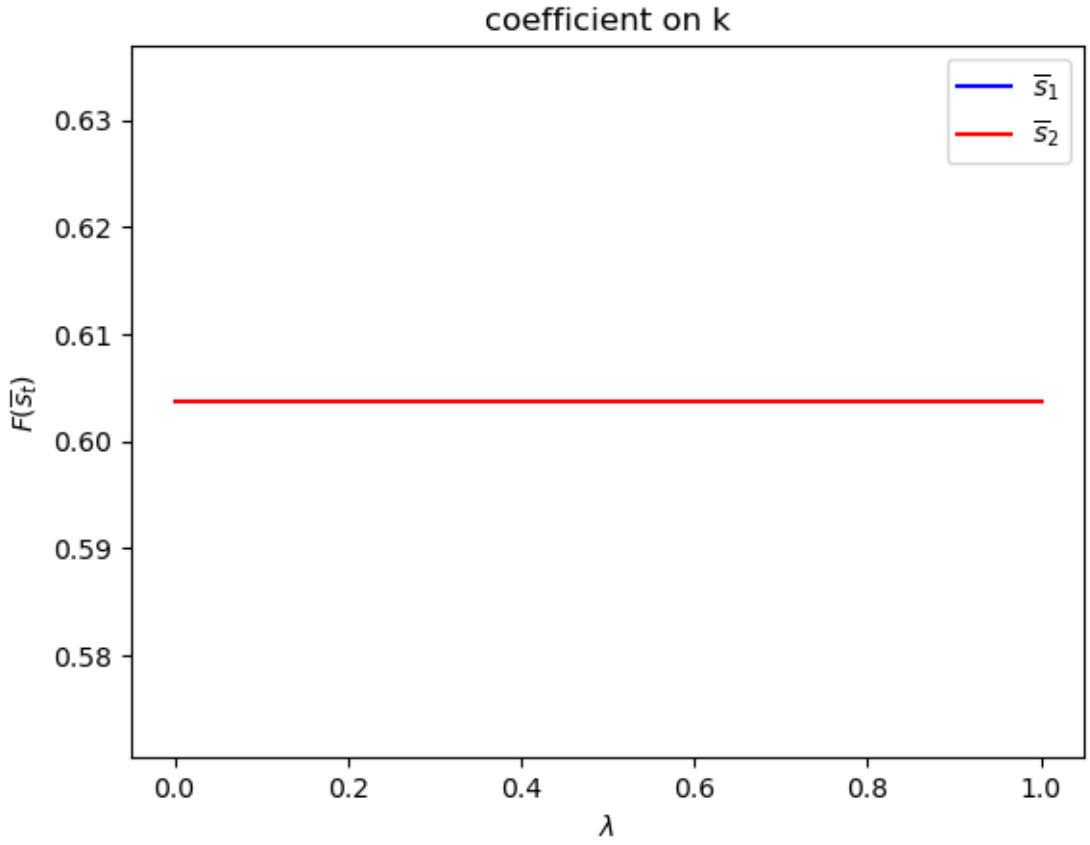
But when $\lambda \to 1$, the Markov transition matrix becomes more nearly periodic, so the optimum decision rules target more at the optimal k level in the other state in order to enjoy higher expected payoff in the next period.

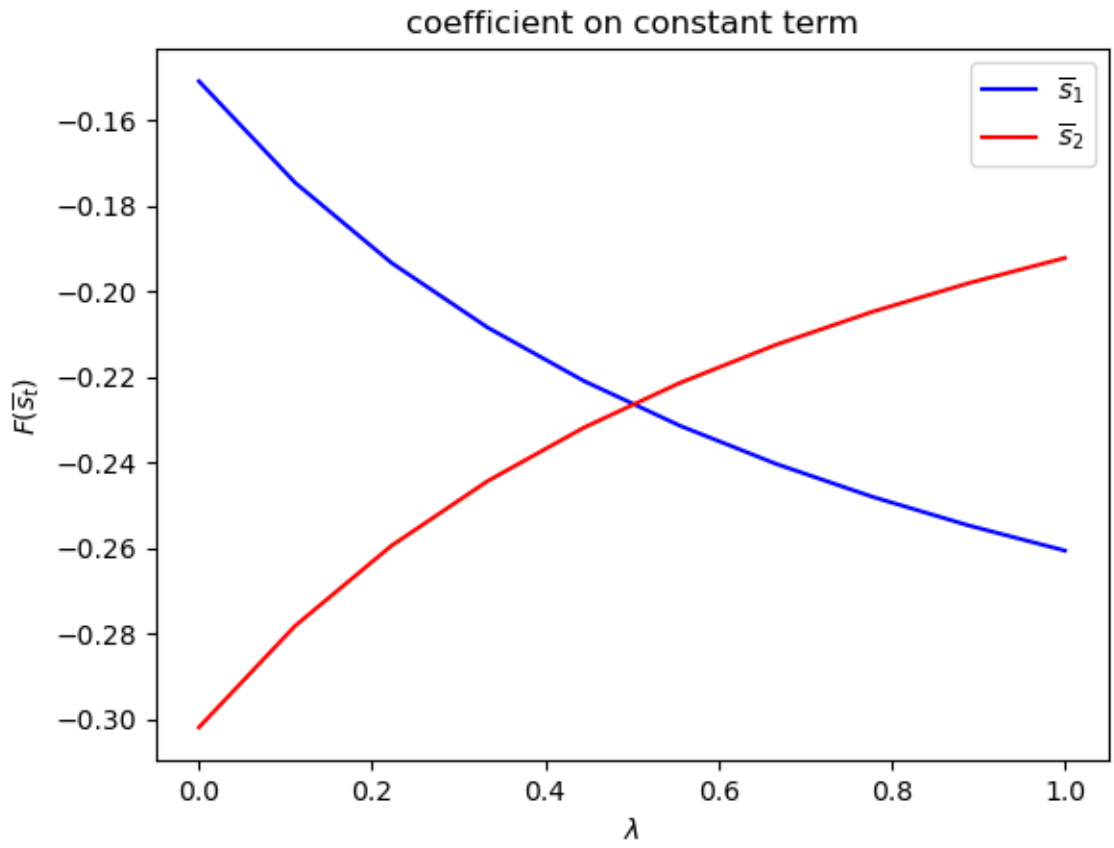
The switch happens at $\lambda = 0.5$ when both states are equally likely to be reached.

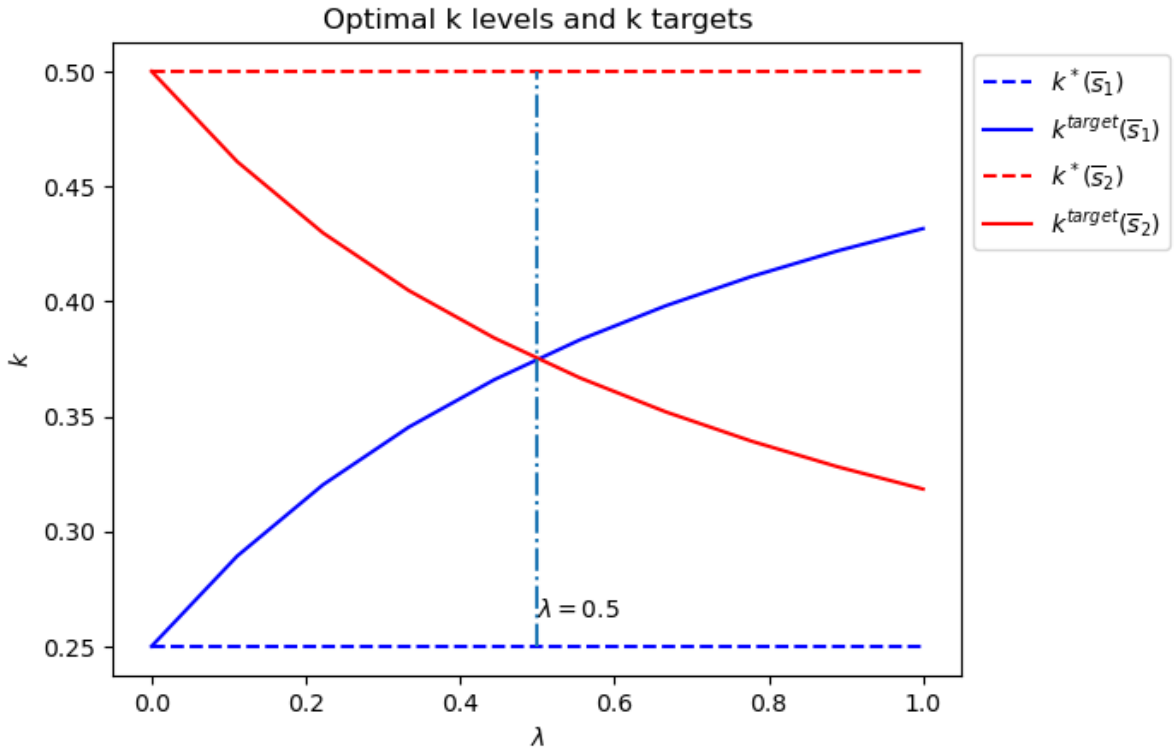
Below we plot an additional figure that shows optimal k levels in the two states Markov jump state and also how the targeted k levels change as λ changes.

```
run(construct_arrays1, {"f1_vals":[0.5, 1.]}, state_vec1)
```

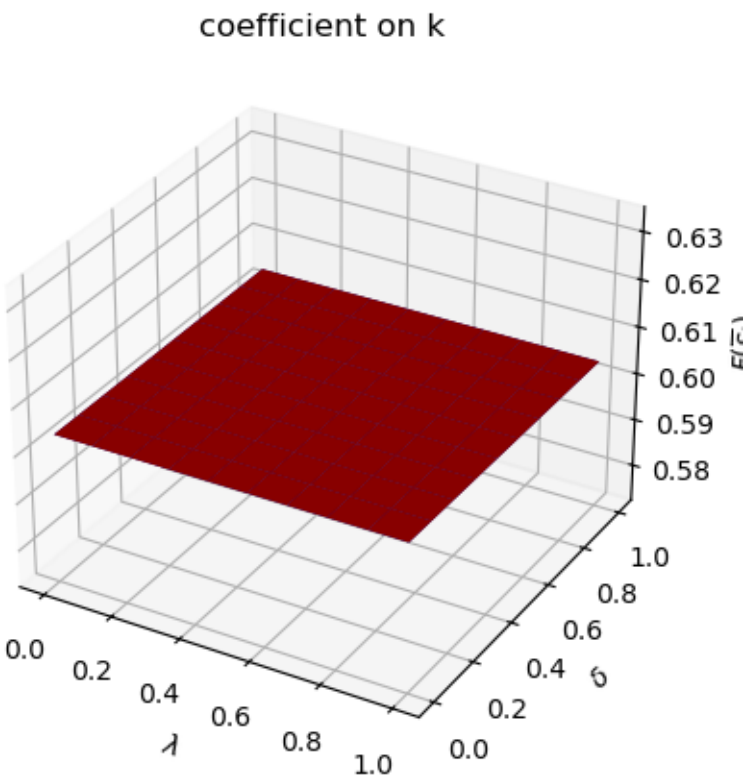
```
symmetric Π case:
```

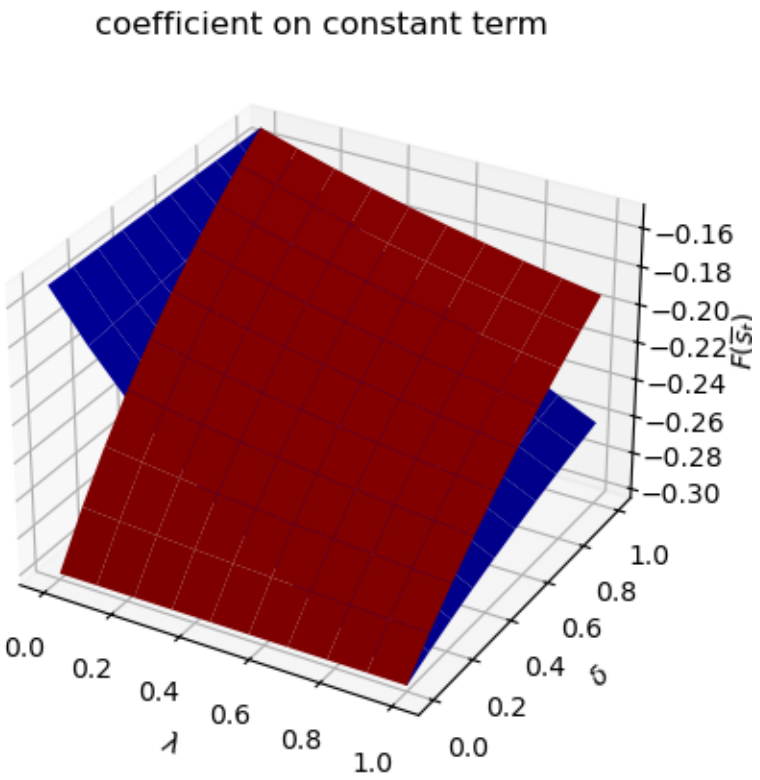



coefficient on constant term
$\overline{s}_1$
$\overline{s}_2$
−0.16
−0.18
−0.20
−0.22
−0.24
−0.26
−0.28
−0.30
$F(\overline{s}_t)$
0.0
0.2
0.4
0.6
0.8
1.0
λ


Optimal k levels and k targets
$k^*(\overline{s}_1)$
$k^{target}(\overline{s}_1)$
$k^*(\overline{s}_2)$
$k^{target}(\overline{s}_2)$
0.50
0.45
0.40
0.35
0.30
0.25
k
$\lambda = 0.5$
0.0
0.2
0.4
0.6
0.8
1.0
λ

```
asymmetric Π case:
```



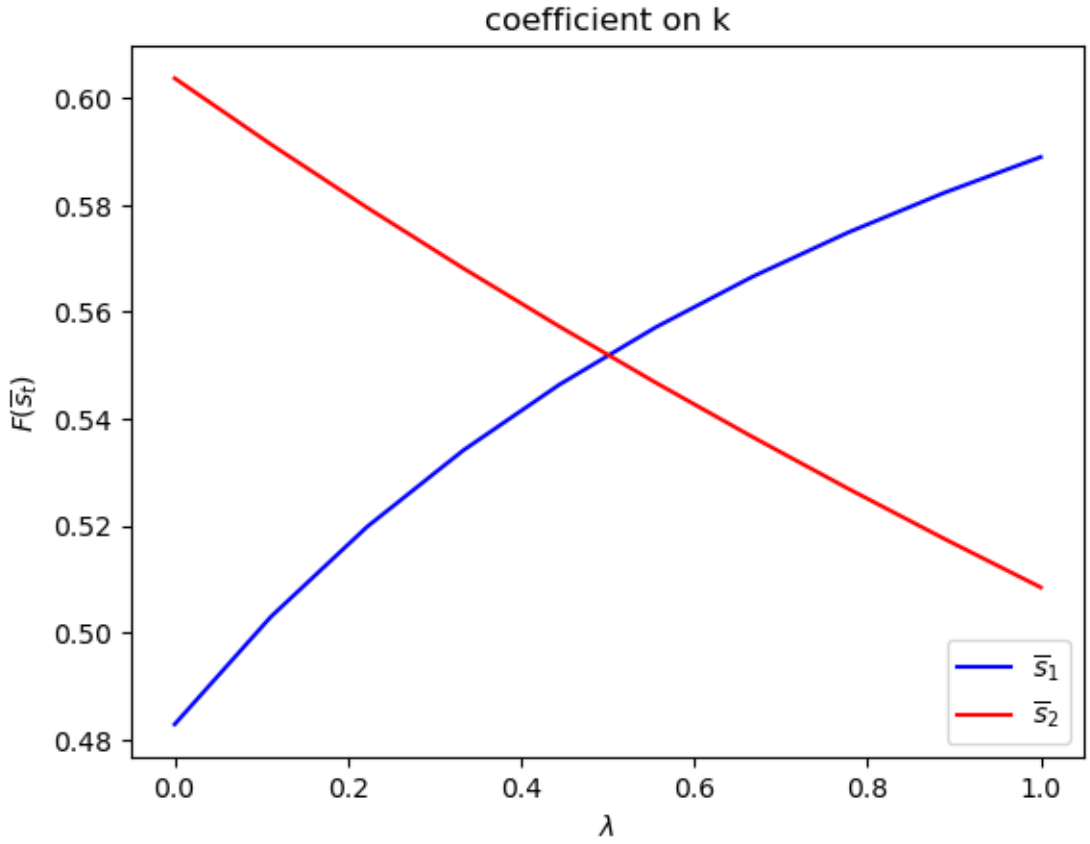


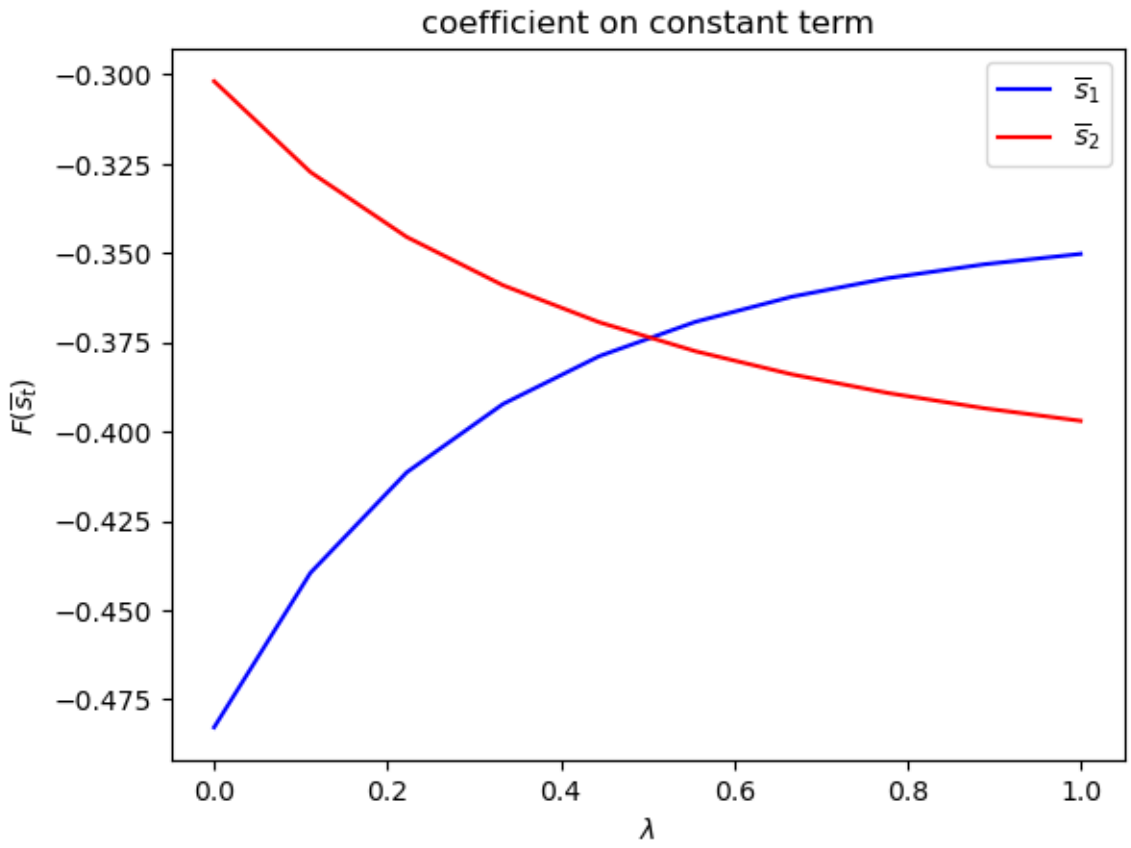
Set f_{1,s_t} and d_{s_t} as constant functions and

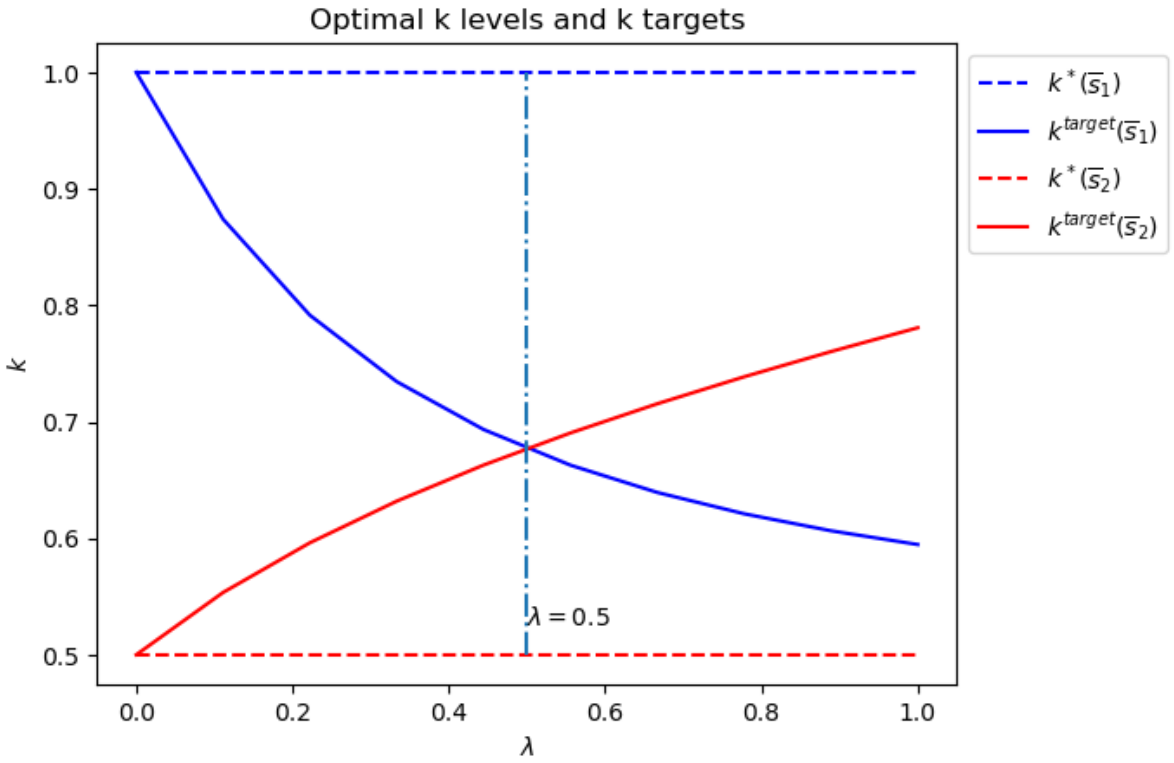
$$f_{2,1} = 0.5, f_{2,2} = 1$$

```
run(construct_arrays1, {"f2_vals":[0.5, 1.]}, state_vec1)
```

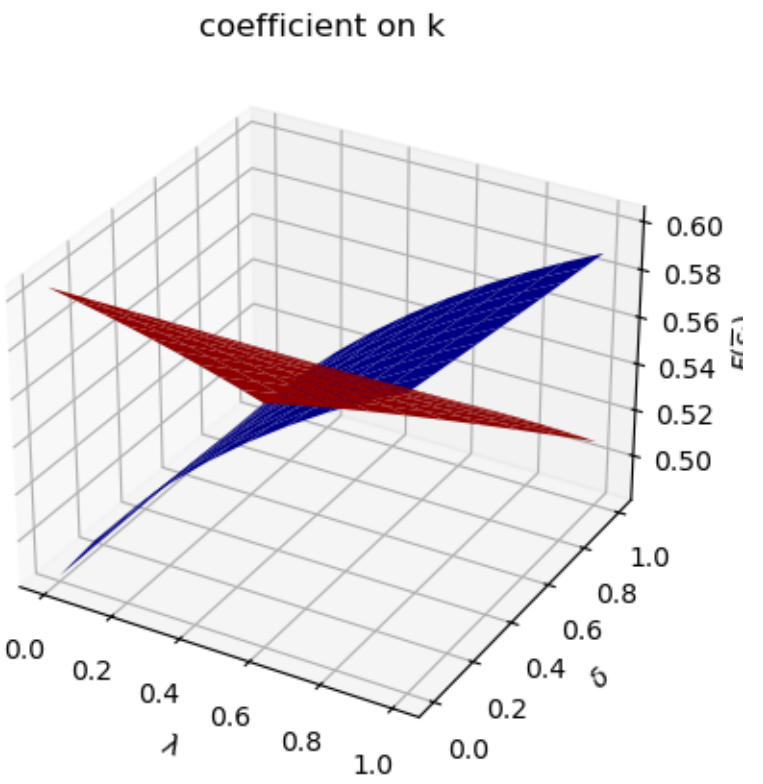
```
symmetric Π case:
```

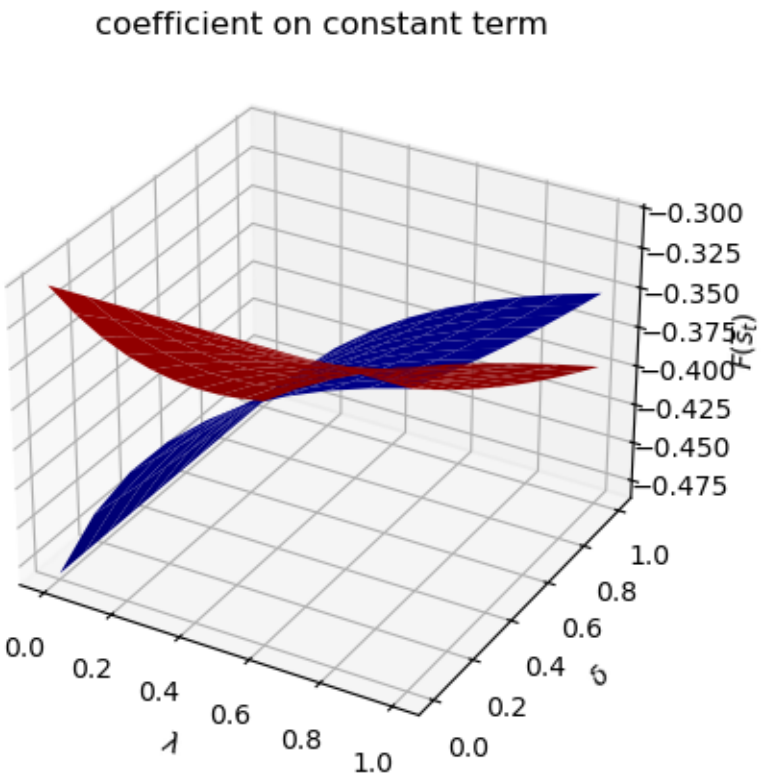



coefficient on constant term
$\overline{s}_1$
$\overline{s}_2$
−0.300
−0.325
−0.350
−0.375
−0.400
−0.425
−0.450
−0.475
$F(\overline{s}_t)$
0.0
0.2
0.4
0.6
0.8
1.0
λ


Optimal k levels and k targets
$k^*(\overline{s}_1)$
$k^{target}(\overline{s}_1)$
$k^*(\overline{s}_2)$
$k^{target}(\overline{s}_2)$
1.0
0.9
0.8
0.7
0.6
0.5
k
$\lambda = 0.5$
0.0
0.2
0.4
0.6
0.8
1.0
λ

```
asymmetric Π case:
```





8.6 Example 2

We now add to the example 1 setup another state variable w_t that follows the evolution law

$$w_{t+1} = \alpha_0(s_t) + \rho(s_t) w_t + \sigma(s_t)\epsilon_{t+1}, \quad \epsilon_{t+1} \sim N(0,1)$$

We think of w_t as a rental rate or tax rate that the decision maker pays each period for k_t.

To capture this idea, we add to the decision-maker's one-period payoff function the product of w_t and k_t

$$r(s_t, k_t, w_t) = f_{1,s_t} k_t - f_{2,s_t} k_t^2 - d_{s_t}(k_{t+1} - k_t)^2 - w_t k_t,$$

We now let the continuous part of the state at time t be $x_t = \begin{bmatrix} k_t \\ 1 \\ w_t \end{bmatrix}$ and continue to set the control $u_t = k_{t+1} - k_t$.

We can write the one-period payoff function $r(s_t, k_t, w_t)$ as

$$\begin{aligned} r(s_t, k_t, w_t) &= f_1(s_t) k_t - f_2(s_t) k_t^2 - d(s_t)(k_{t+1} - k_t)^2 - w_t k_t \\ &= -\left(x_t' \underbrace{\begin{bmatrix} f_2(s_t) & -\frac{f_1(s_t)}{2} & \frac{1}{2} \\ -\frac{f_1(s_t)}{2} & 0 & 0 \\ \frac{1}{2} & 0 & 0 \end{bmatrix}}_{\equiv R(s_t)} x_t + \underbrace{d(s_t)}_{\equiv Q(s_t)} u_t^2 \right), \end{aligned}$$

and the state-transition law as

$$x_{t+1} = \begin{bmatrix} k_{t+1} \\ 1 \\ w_{t+1} \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & \alpha_0(s_t) & \rho(s_t) \end{bmatrix}}_{\equiv A(s_t)} x_t + \underbrace{\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}}_{\equiv B(s_t)} u_t + \underbrace{\begin{bmatrix} 0 \\ 0 \\ \sigma(s_t) \end{bmatrix}}_{\equiv C(s_t)} \epsilon_{t+1}$$

```
def construct_arrays2(f1_vals=[1. ,1.],
                      f2_vals=[1., 1.],
                      d_vals=[1., 1.],
                      α0_vals=[1., 1.],
                      ρ_vals=[0.9, 0.9],
                      σ_vals=[1., 1.]):
    """
    Construct matrices that maps the problem described in example 2
    into a Markov jump linear quadratic dynamic programming problem.
    """

    m = len(f1_vals)
    n, k, j = 3, 1, 1

    Rs = np.zeros((m, n, n))
    Qs = np.zeros((m, k, k))
    As = np.zeros((m, n, n))
    Bs = np.zeros((m, n, k))
    Cs = np.zeros((m, n, j))

    for i in range(m):
        Rs[i, 0, 0] = f2_vals[i]
        Rs[i, 1, 0] = - f1_vals[i] / 2
```

(continues on next page)

(continued from previous page)

```
        Rs[i, 0, 1] = - f1_vals[i] / 2
        Rs[i, 0, 2] = 1/2
        Rs[i, 2, 0] = 1/2

        Qs[i, 0, 0] = d_vals[i]

        As[i, 0, 0] = 1
        As[i, 1, 1] = 1
        As[i, 2, 1] = α0_vals[i]
        As[i, 2, 2] = ρ_vals[i]

        Bs[i, :, :] = np.array([[1, 0, 0]]).T

        Cs[i, :, :] = np.array([[0, 0, σ_vals[i]]]).T

    Ns = None
    k_star = None

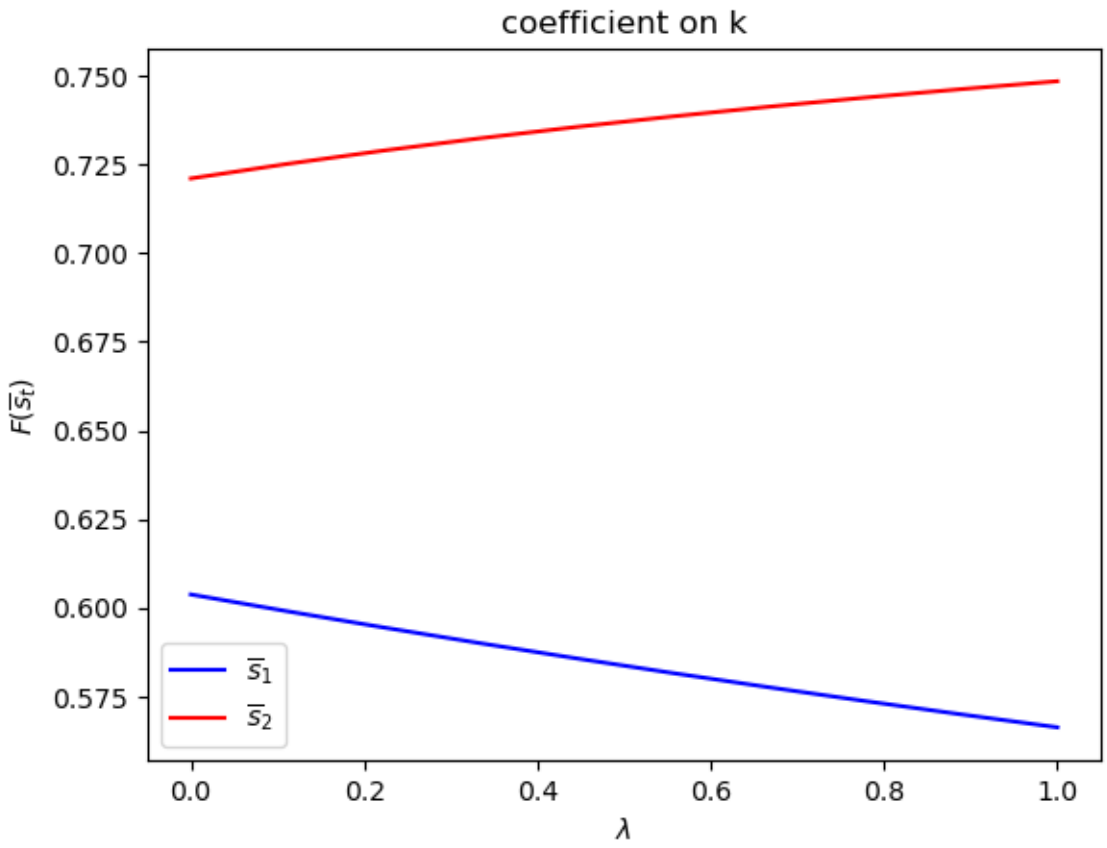
    return Qs, Rs, Ns, As, Bs, Cs, k_star
```

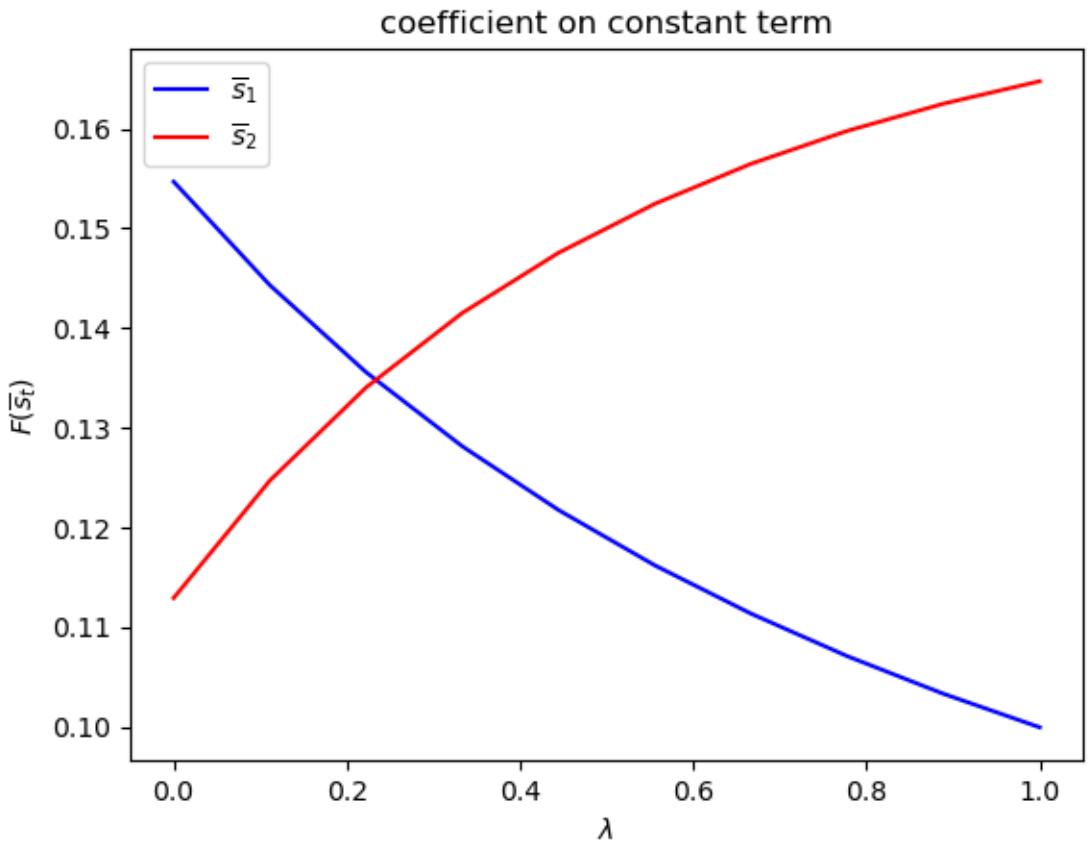
```
state_vec2 = ["k", "constant term", "w"]
```

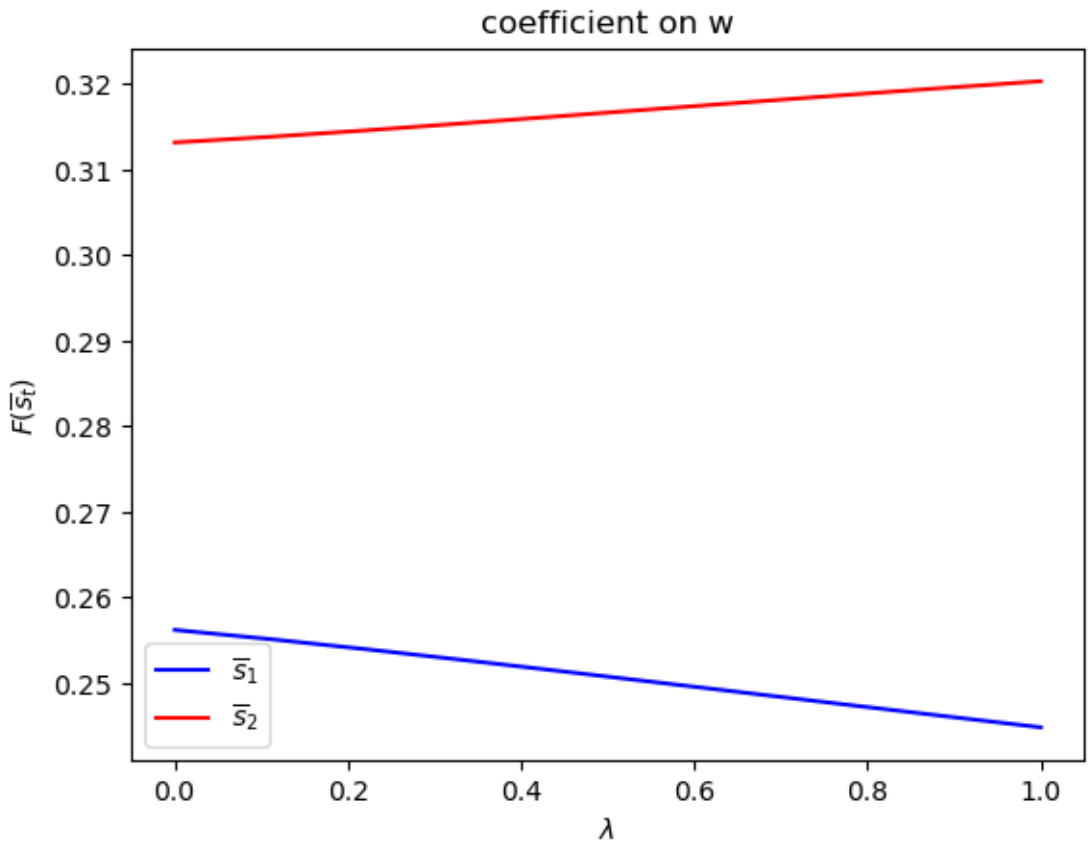
Only d_{s_t} depends on s_t.

```
run(construct_arrays2, {"d_vals":[1., 0.5]}, state_vec2)
```

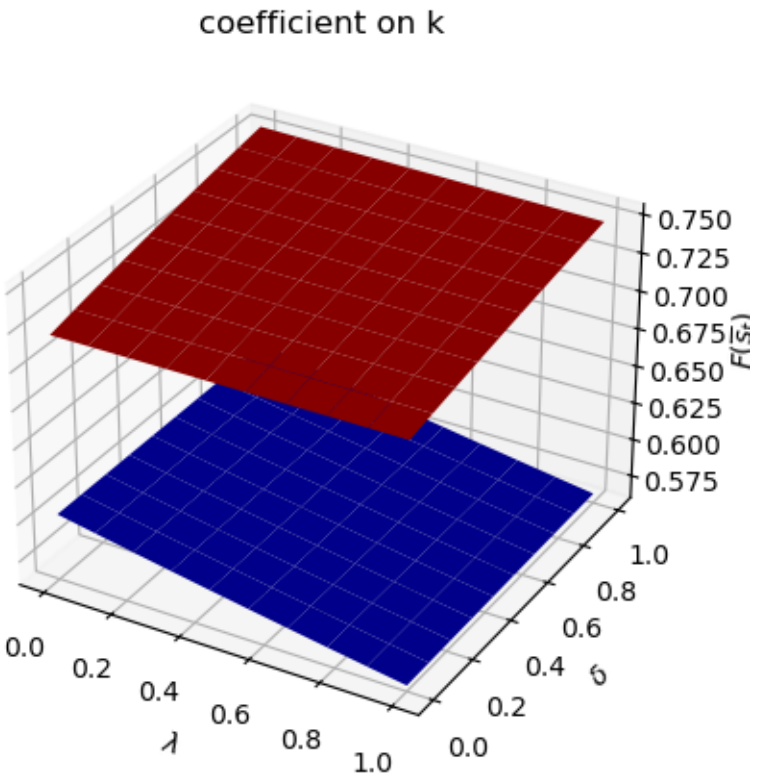
```
symmetric Π case:
```

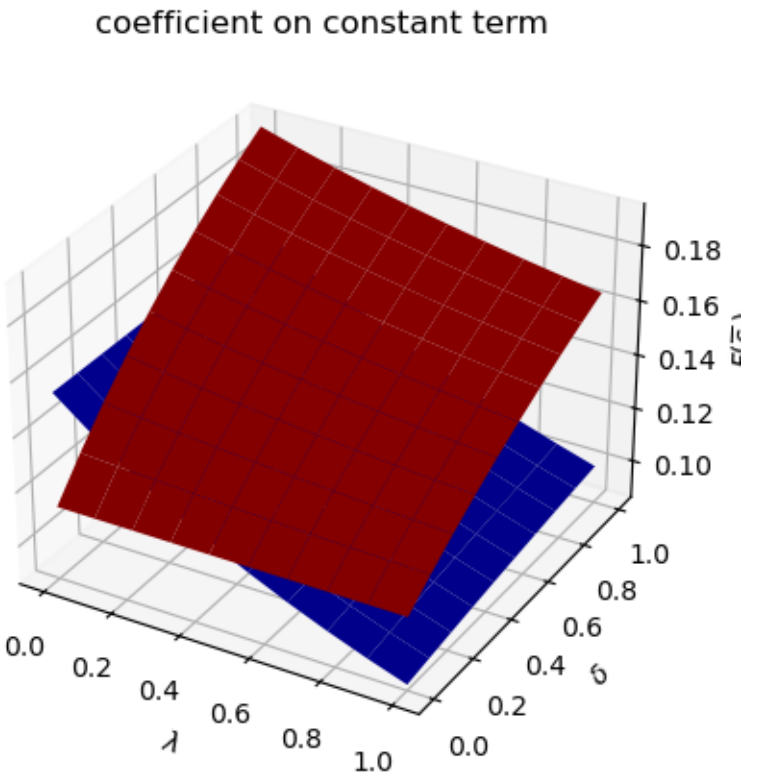

coefficient on k
0.750
0.725
0.700
0.675
0.650
0.625
0.600
0.575
$F(\bar{s}_t)$
$\bar{s}_1$
$\bar{s}_2$
0.0
0.2
0.4
0.6
0.8
1.0
λ

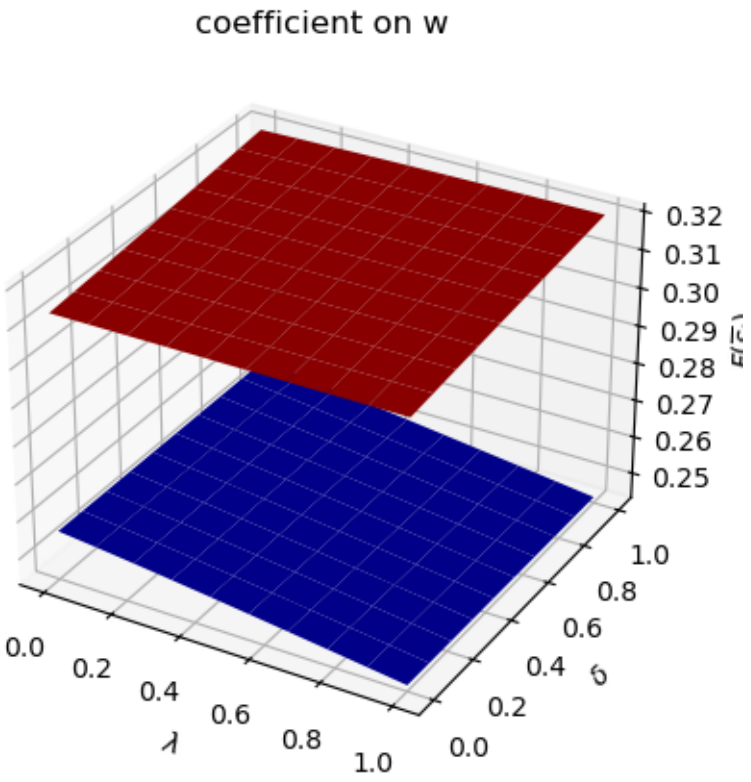

coefficient on constant term
$\overline{s}_1$
$\overline{s}_2$
0.16
0.15
0.14
0.13
0.12
0.11
0.10
$F(\overline{s}_t)$
0.0
0.2
0.4
0.6
0.8
1.0
λ




```
asymmetric Π case:
```


coefficient on k
0.750
0.725
0.700
0.675
0.650
0.625
0.600
0.575
$F(\widetilde{s}_t)$
0.0
0.2
0.4
0.6
0.8
1.0
λ
0.0
0.2
0.4
0.6
0.8
1.0
ϕ

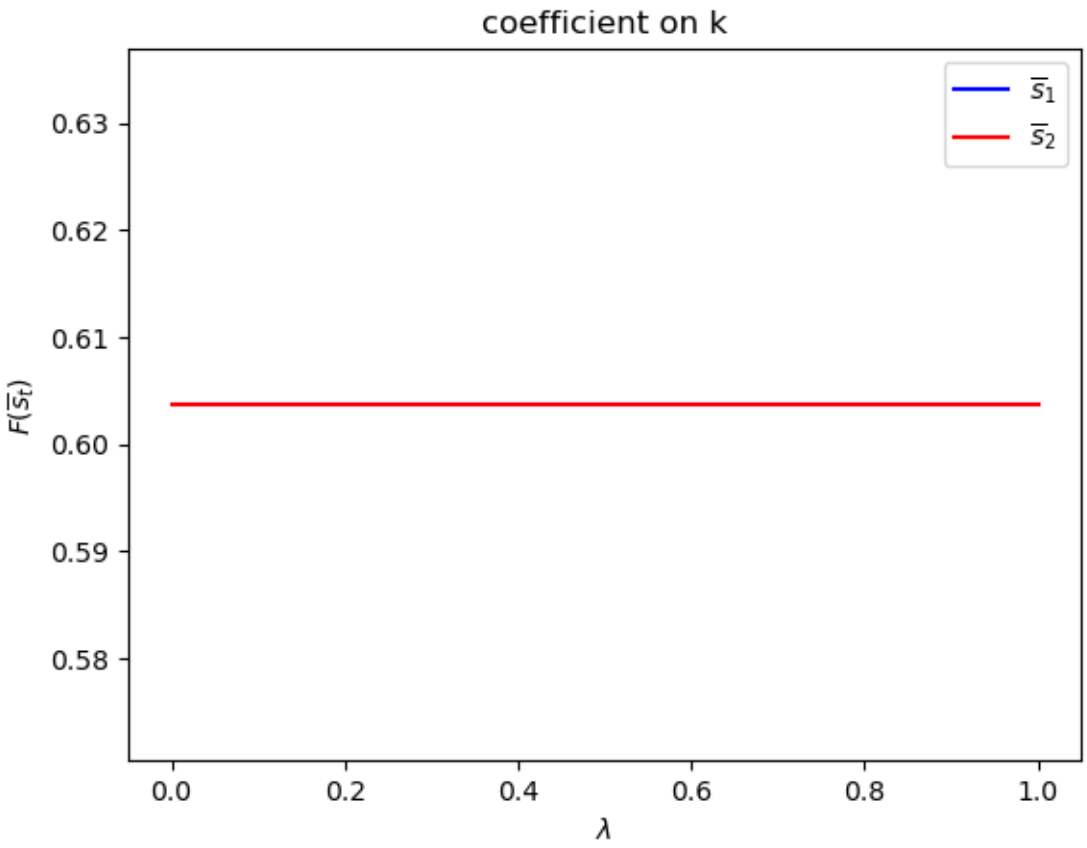

coefficient on constant term
0.18
0.16
0.14
0.12
0.10
0.0
0.2
0.4
0.6
0.8
1.0
λ
0.0
0.2
0.4
0.6
0.8
1.0
ϕ

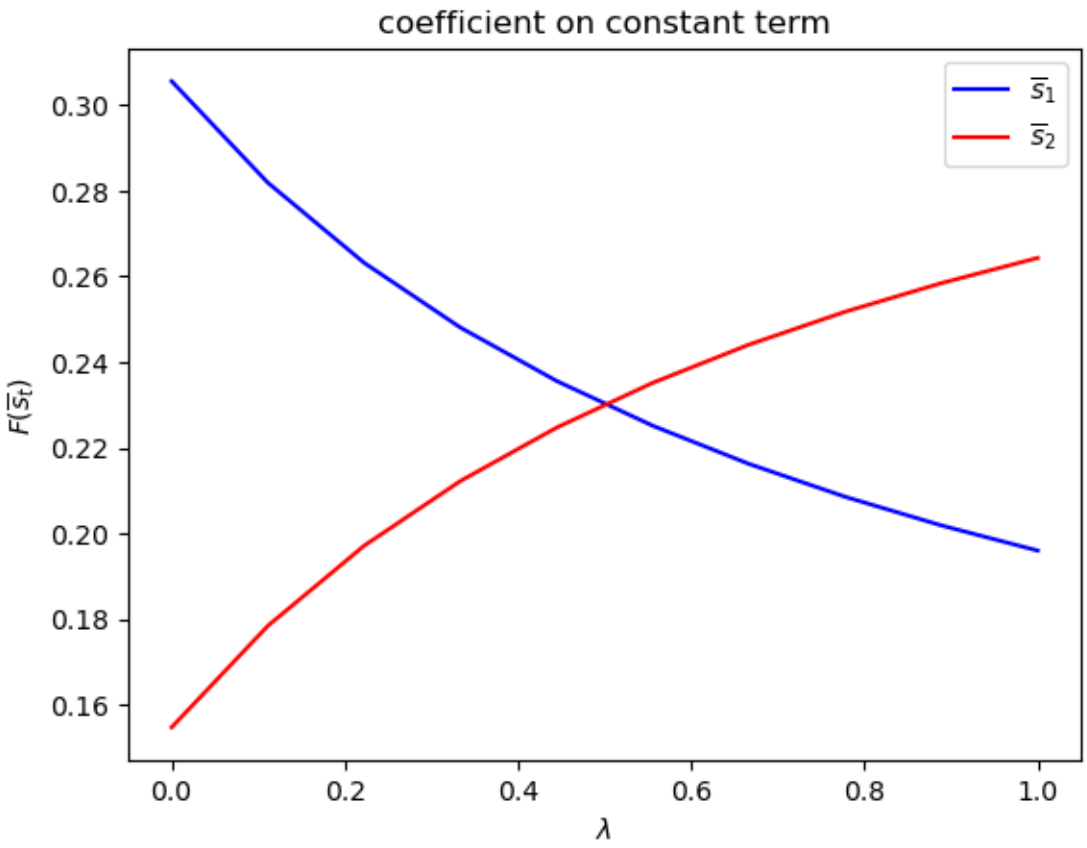


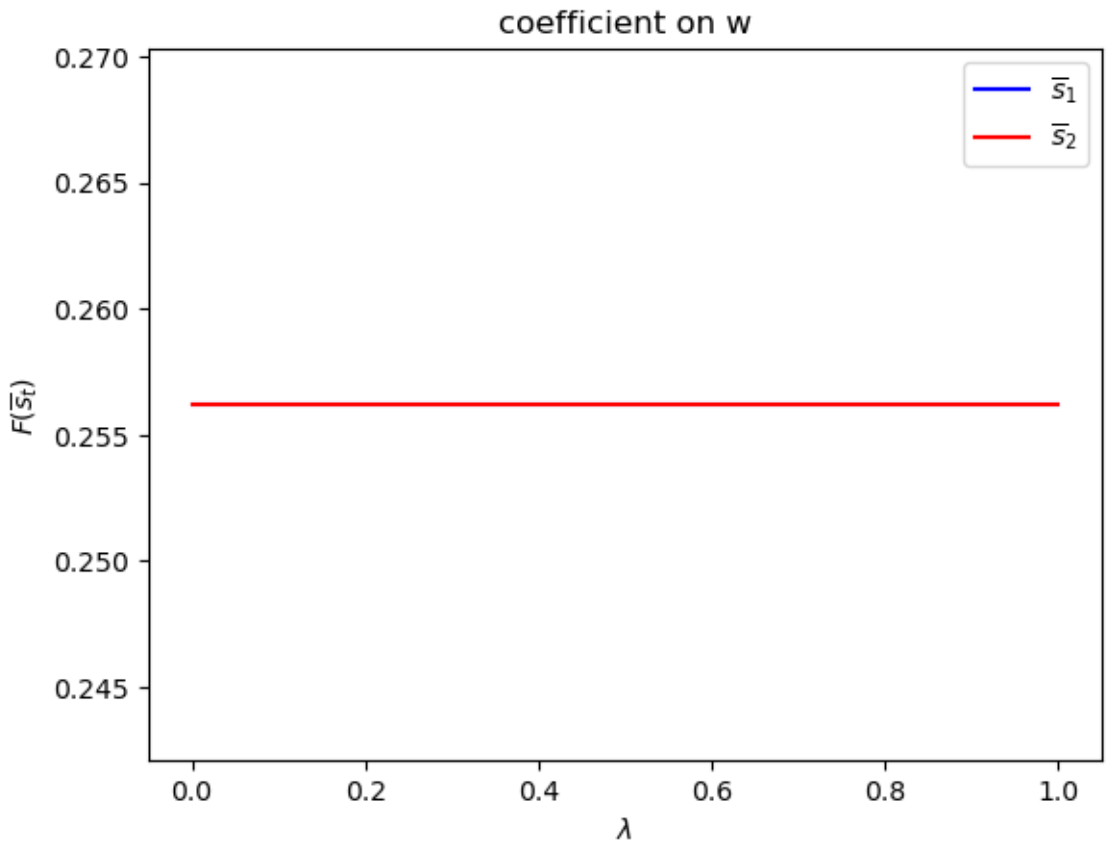
Only f_{1,s_t} depends on s_t.

```
run(construct_arrays2, {"f1_vals":[0.5, 1.]}, state_vec2)
```

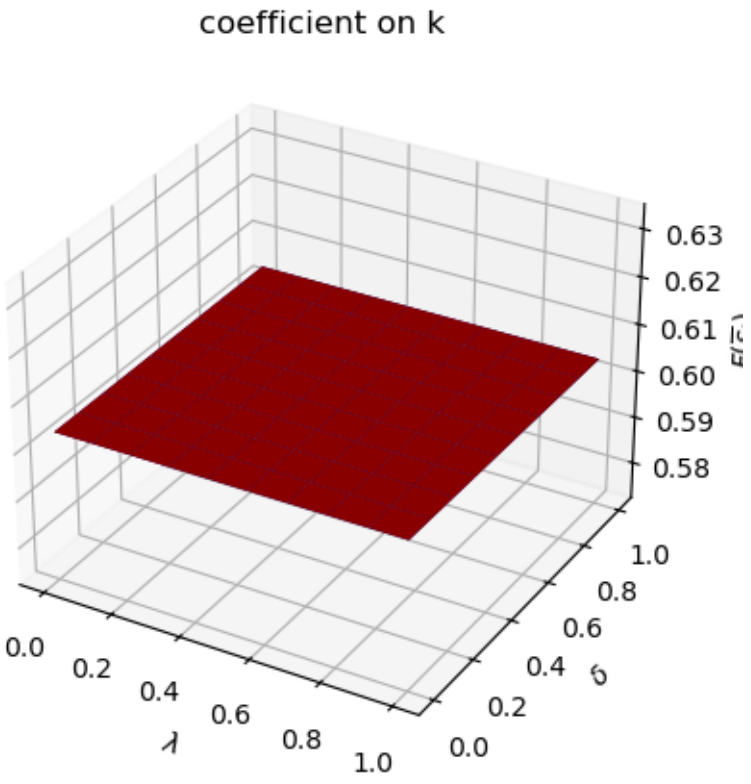
```
symmetric Π case:
```

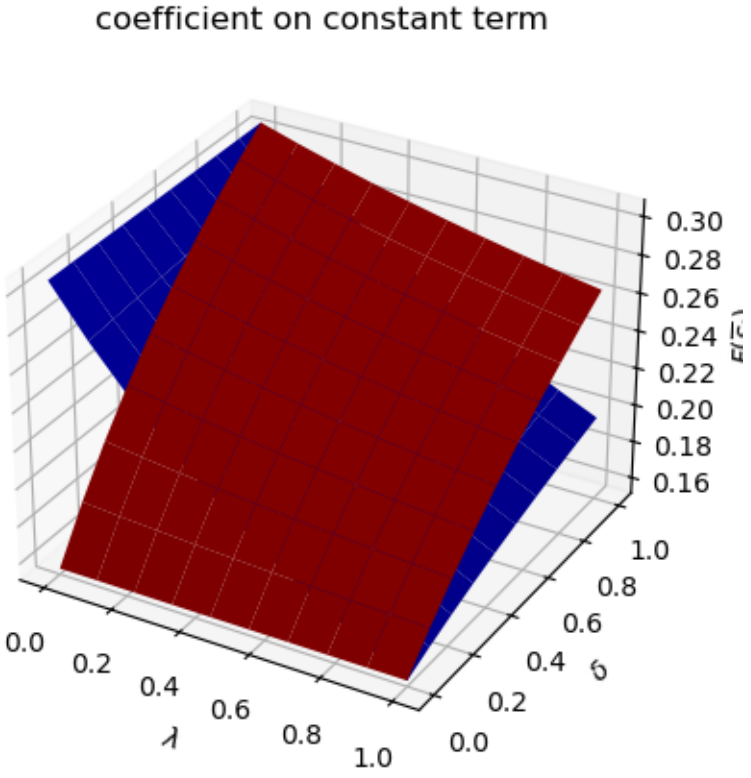

coefficient on k
$\overline{s}_1$
$\overline{s}_2$
0.63
0.62
0.61
0.60
0.59
0.58
$F(\overline{s}_t)$
0.0
0.2
0.4
0.6
0.8
1.0
λ

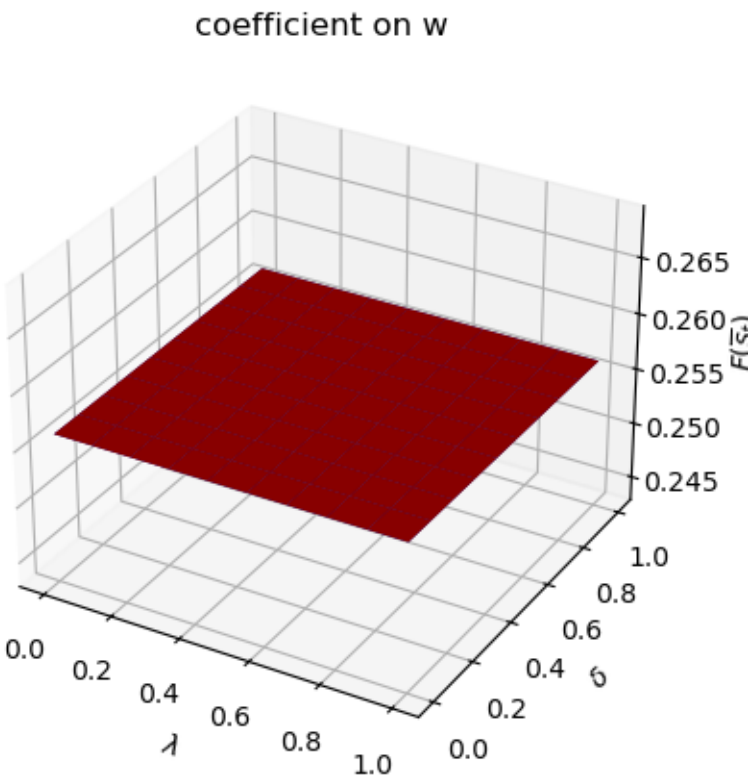

coefficient on constant term
$\overline{s}_1$
$\overline{s}_2$
0.30
0.28
0.26
0.24
0.22
0.20
0.18
0.16
$F(\overline{s}_t)$
0.0
0.2
0.4
0.6
0.8
1.0
λ




```
asymmetric Π case:
```


coefficient on k
0.63
0.62
0.61
0.60
0.59
0.58
0.0
0.2
0.4
0.6
0.8
1.0
λ
1.0
0.8
0.6
0.4
0.2
0.0
ρ

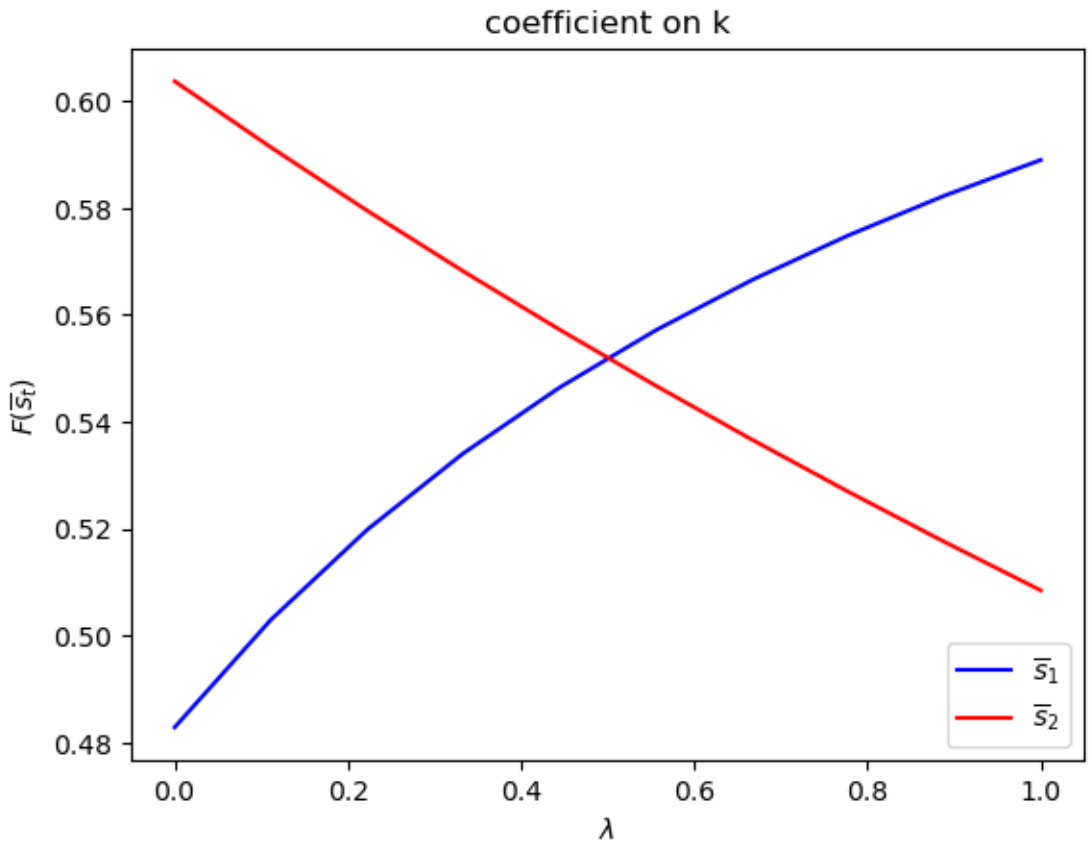

coefficient on constant term
0.30
0.28
0.26
0.24
0.22
0.20
0.18
0.16
0.0
0.2
0.4
0.6
0.8
1.0
λ
1.0
0.8
0.6
0.4
0.2
0.0
ρ

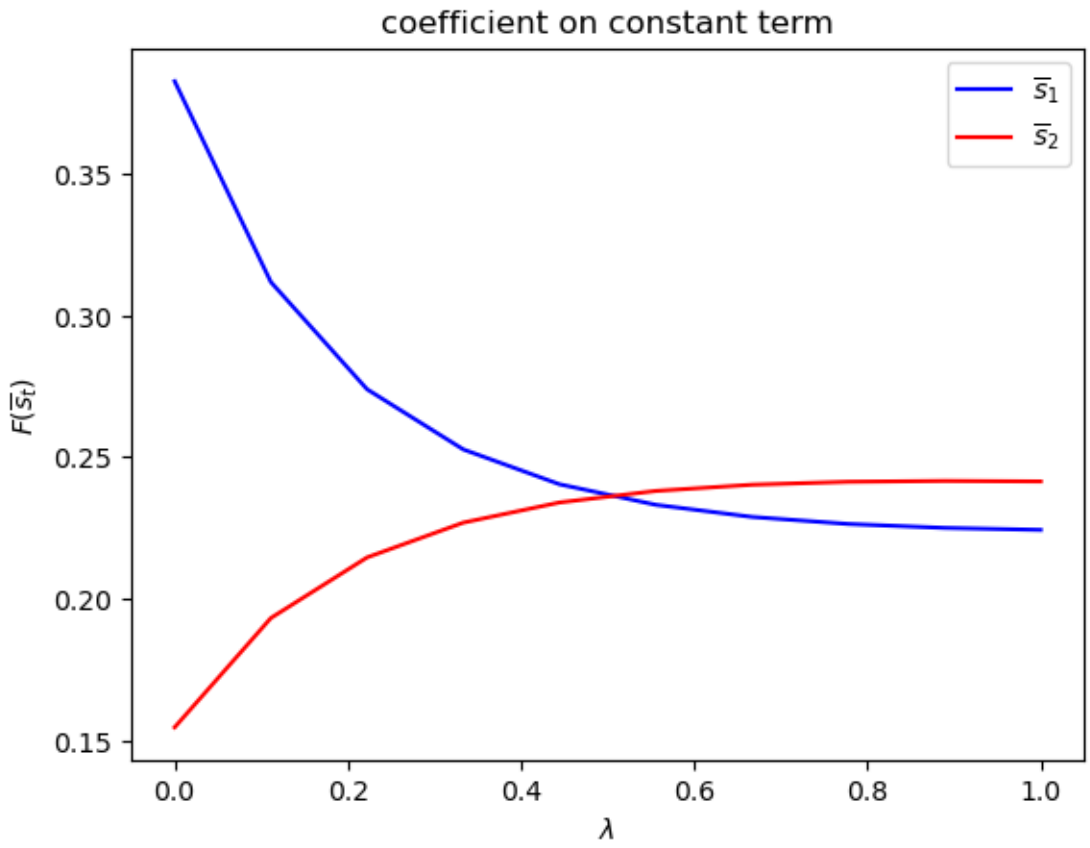


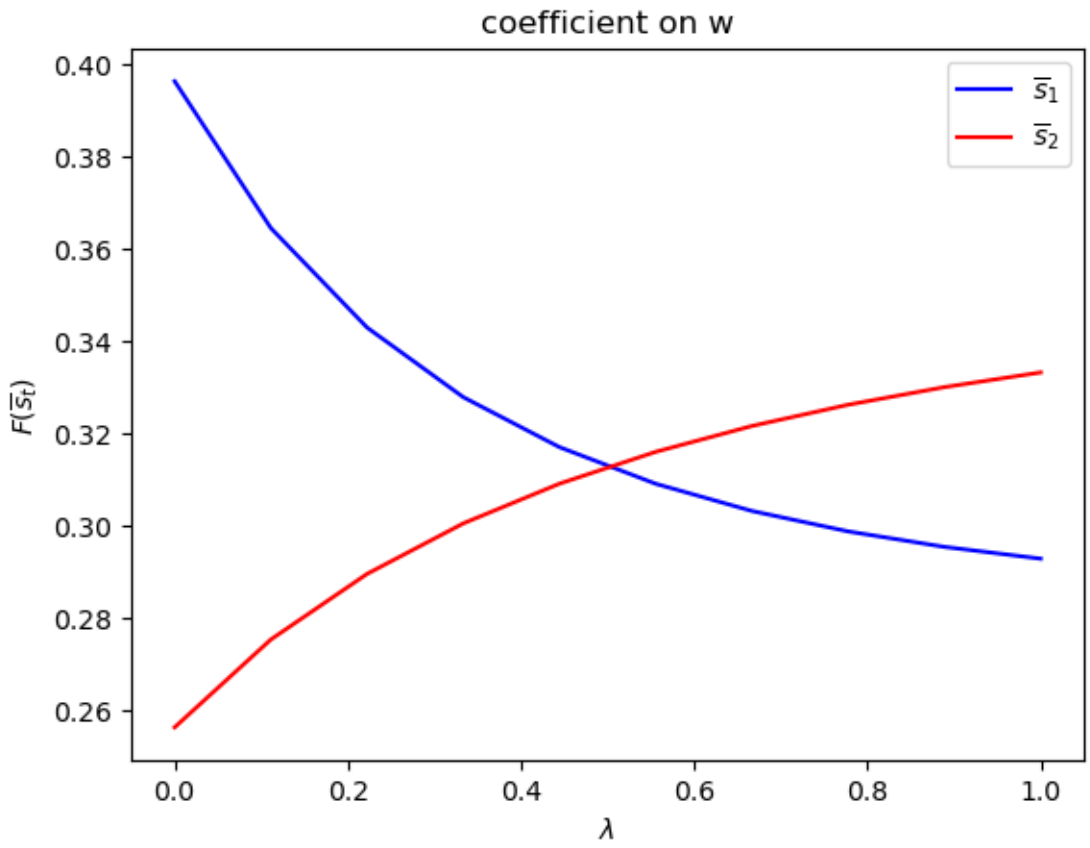
Only f_{2,s_t} depends on s_t.

```
run(construct_arrays2, {"f2_vals":[0.5, 1.]}, state_vec2)
```

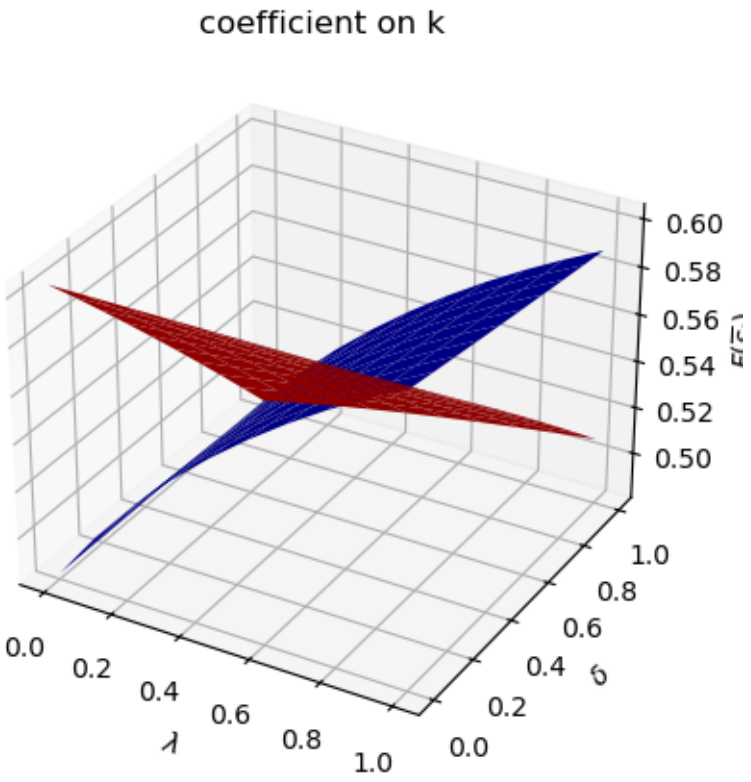
```
symmetric Π case:
```

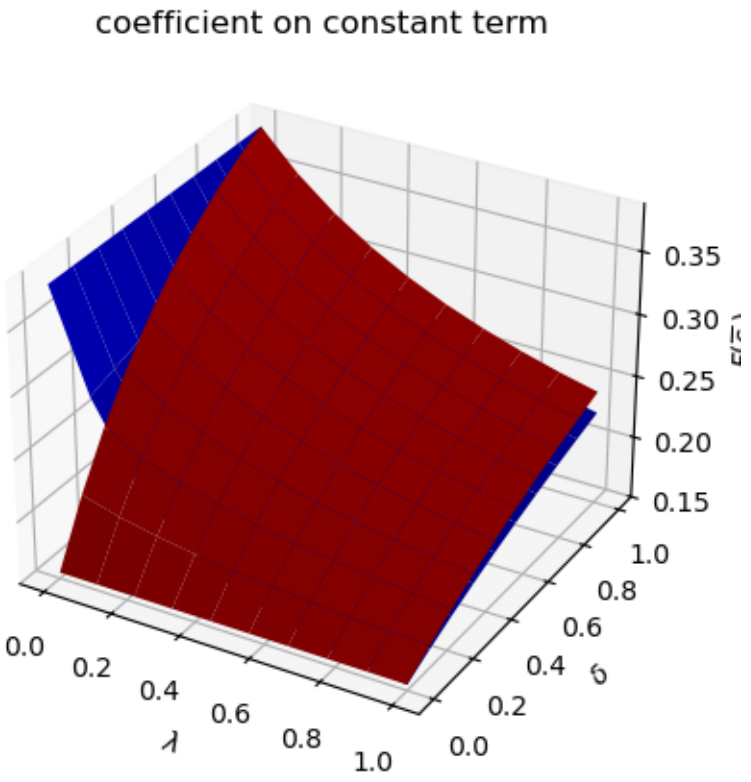

coefficient on k
0.60
0.58
0.56
0.54
0.52
0.50
0.48
$F(\bar{s}_t)$
0.0
0.2
0.4
0.6
0.8
1.0
λ
$\bar{s}_1$
$\bar{s}_2$

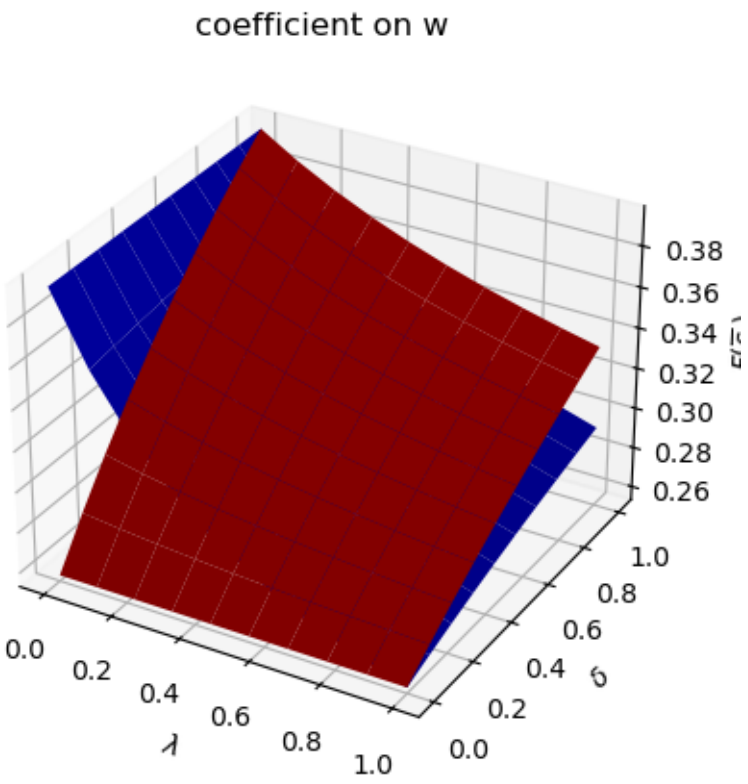

coefficient on constant term
$\overline{s}_1$
$\overline{s}_2$
0.35
0.30
0.25
0.20
0.15
$F(\overline{s}_t)$
0.0
0.2
0.4
0.6
0.8
1.0
λ




```
asymmetric Π case:
```


coefficient on k
0.60
0.58
0.56
0.54
0.52
0.50
0.0
0.2
0.4
0.6
0.8
1.0
λ
1.0
0.8
0.6
0.4
0.2
0.0
σ

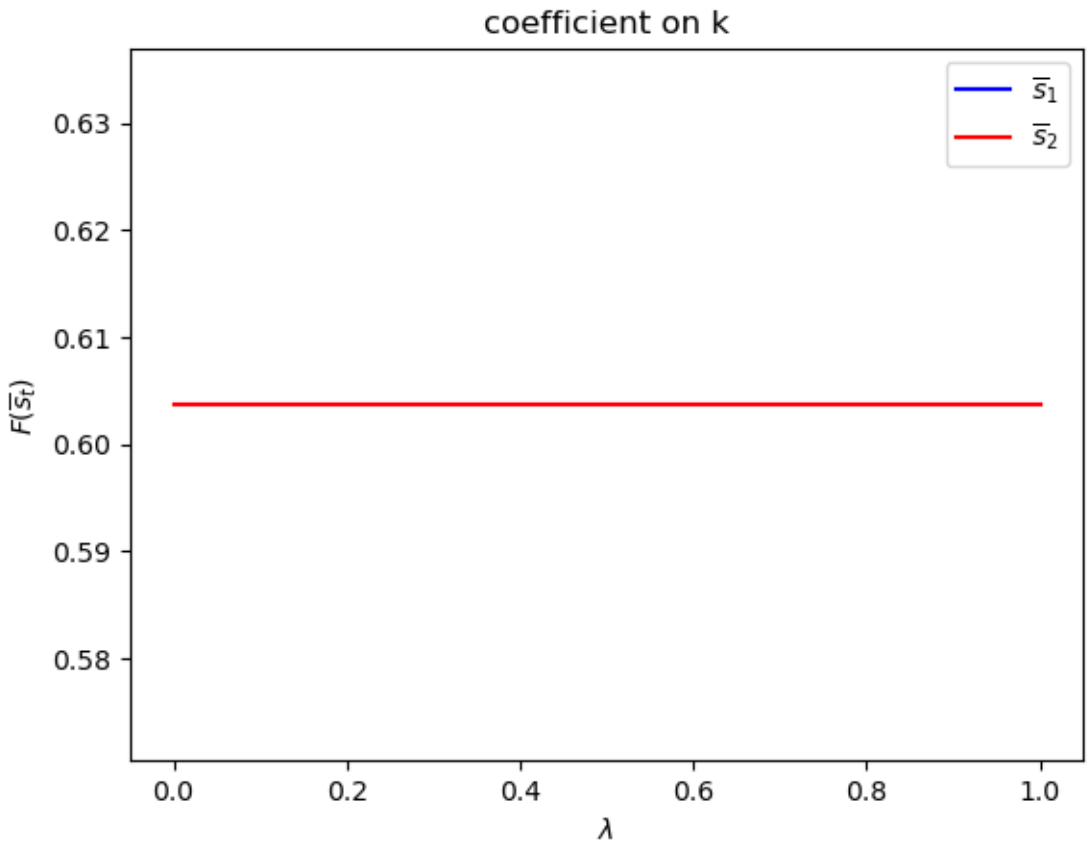

coefficient on constant term
0.35
0.30
0.25
0.20
0.15
0.0
0.2
0.4
0.6
0.8
1.0
λ
1.0
0.8
0.6
0.4
0.2
0.0
σ

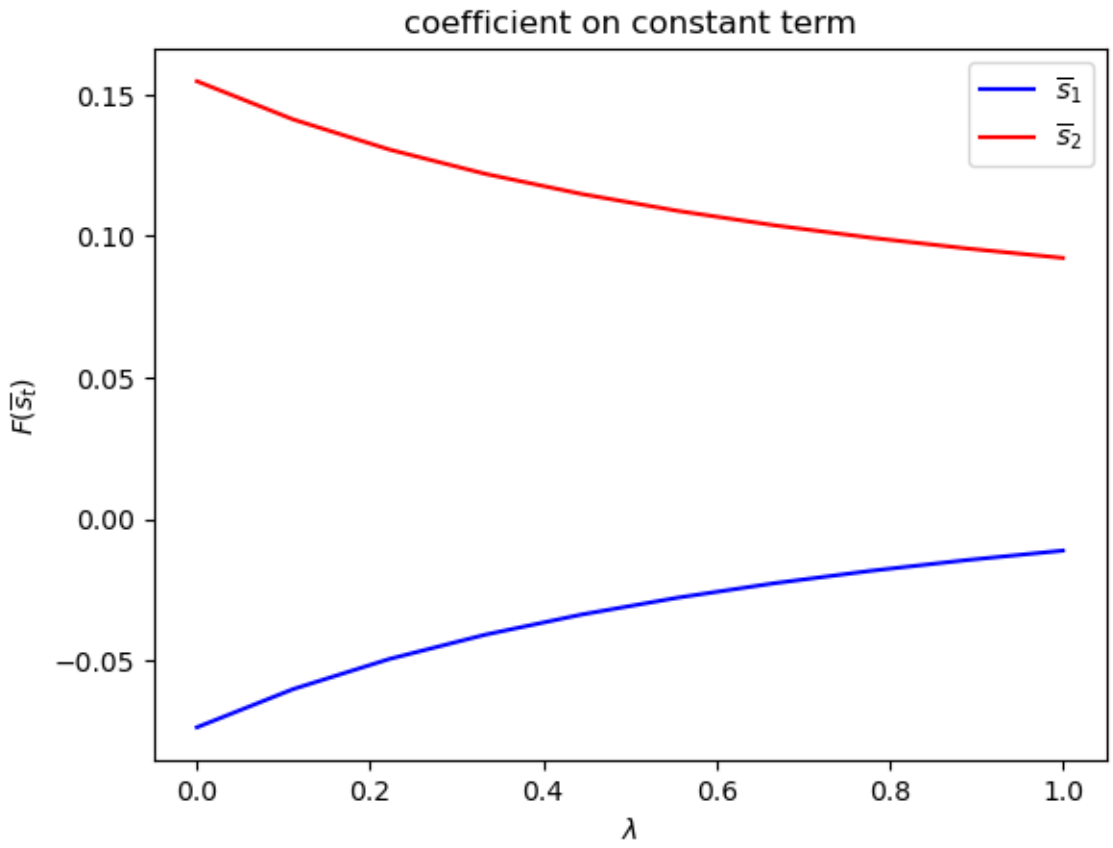


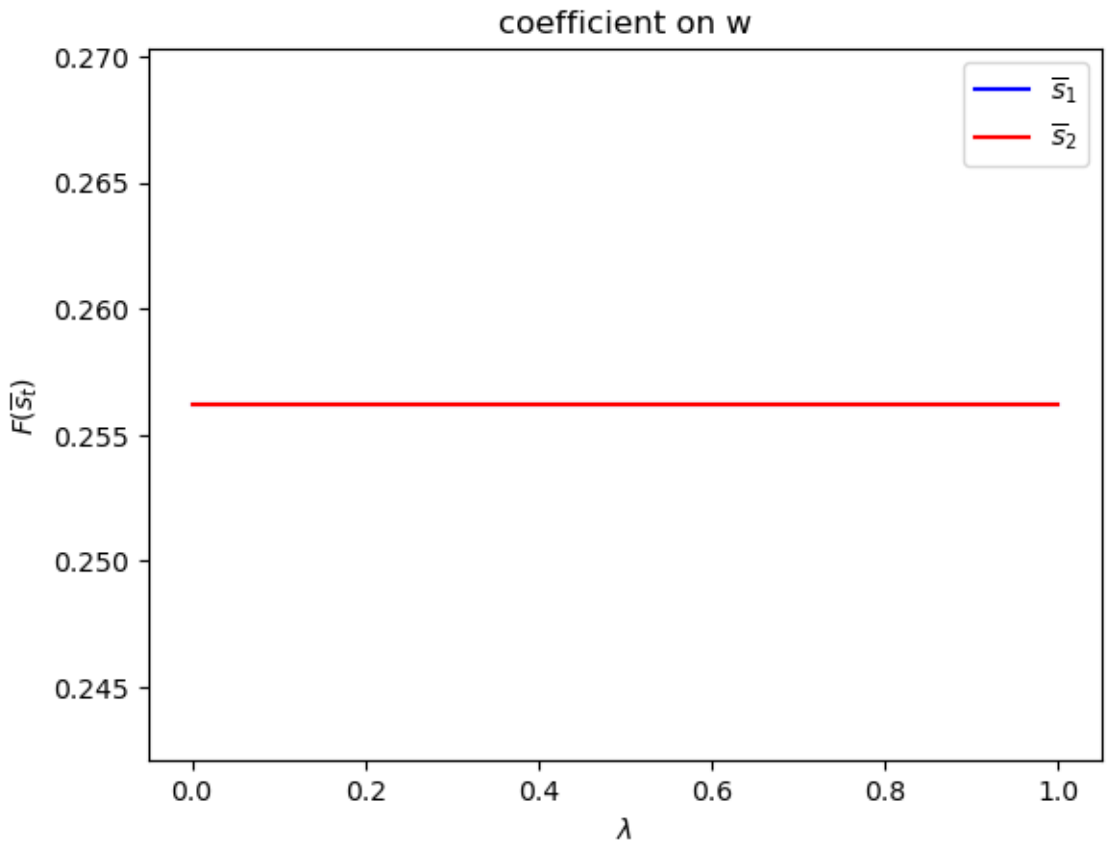
Only $\alpha_0(s_t)$ depends on s_t.

```
run(construct_arrays2, {"α0_vals":[0.5, 1.]}, state_vec2)
```

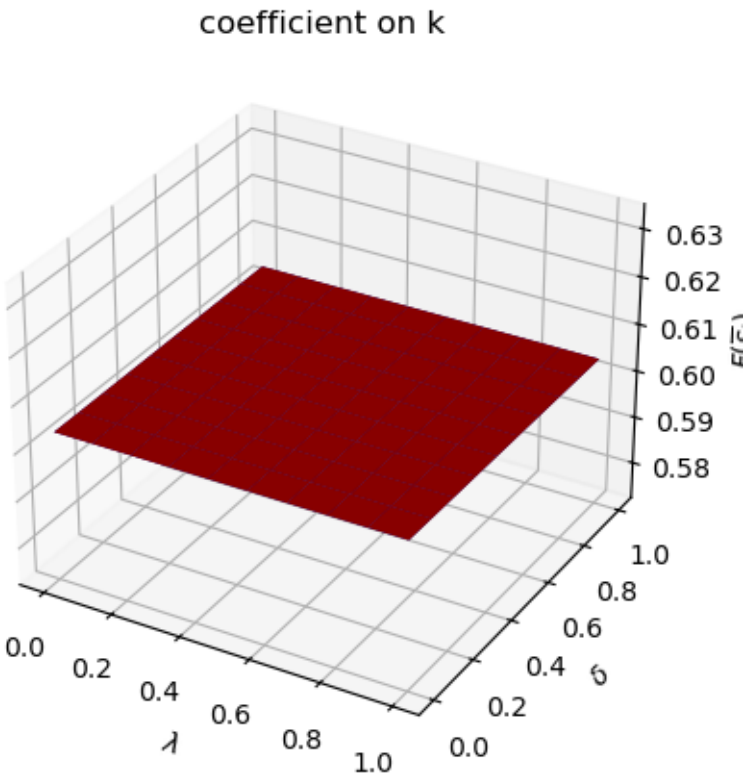
```
symmetric Π case:
```

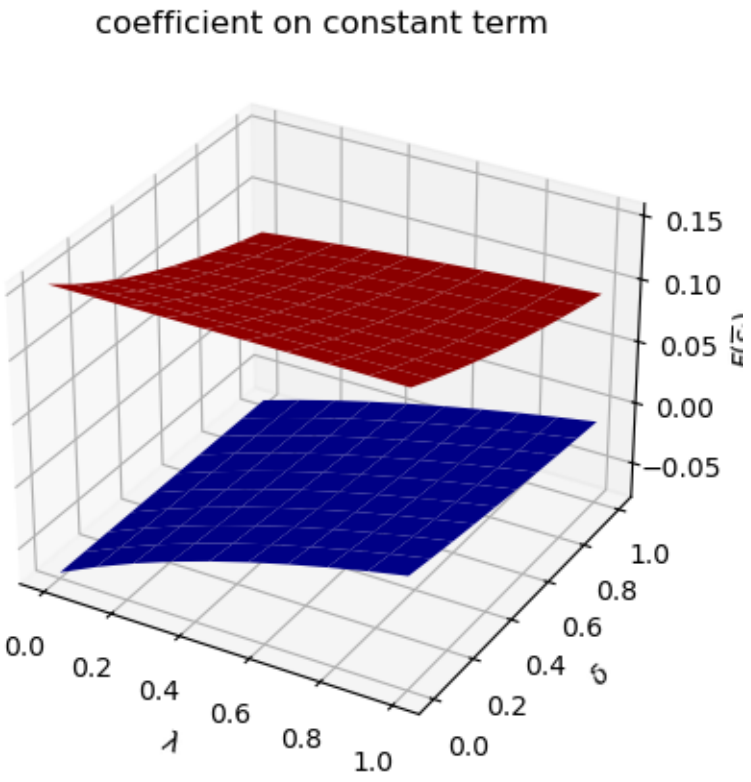

coefficient on k
$\overline{s}_1$
$\overline{s}_2$
0.63
0.62
0.61
0.60
0.59
0.58
$F(\overline{s}_t)$
0.0
0.2
0.4
0.6
0.8
1.0
λ

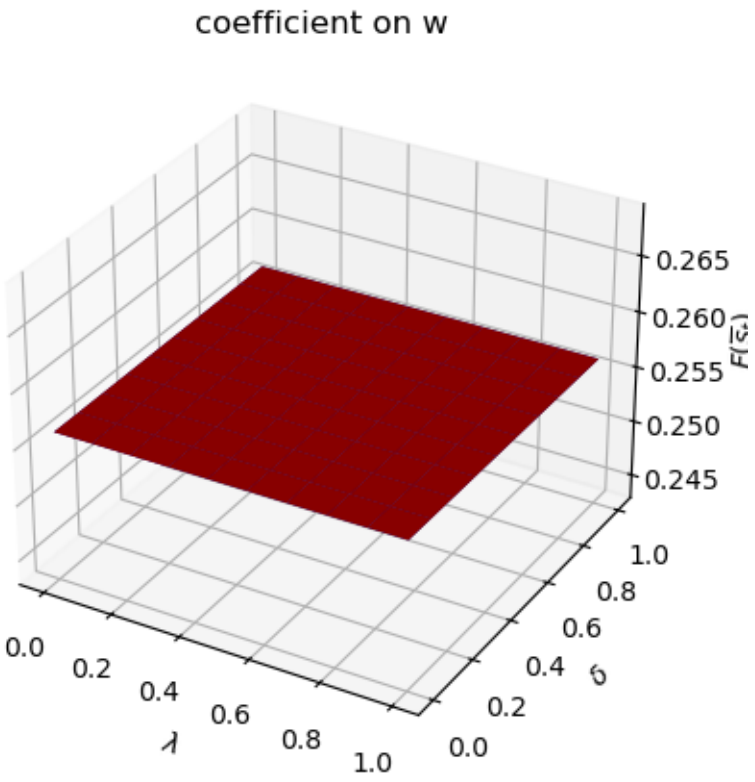

coefficient on constant term
0.15
0.10
0.05
0.00
−0.05
$F(\overline{s}_t)$
0.0
0.2
0.4
0.6
0.8
1.0
λ
$\overline{s}_1$
$\overline{s}_2$




```
asymmetric Π case:
```


coefficient on k
0.63
0.62
0.61
0.60
0.59
0.58
0.0
0.2
0.4
0.6
0.8
1.0
λ
1.0
0.8
0.6
0.4
0.2
0.0
ρ

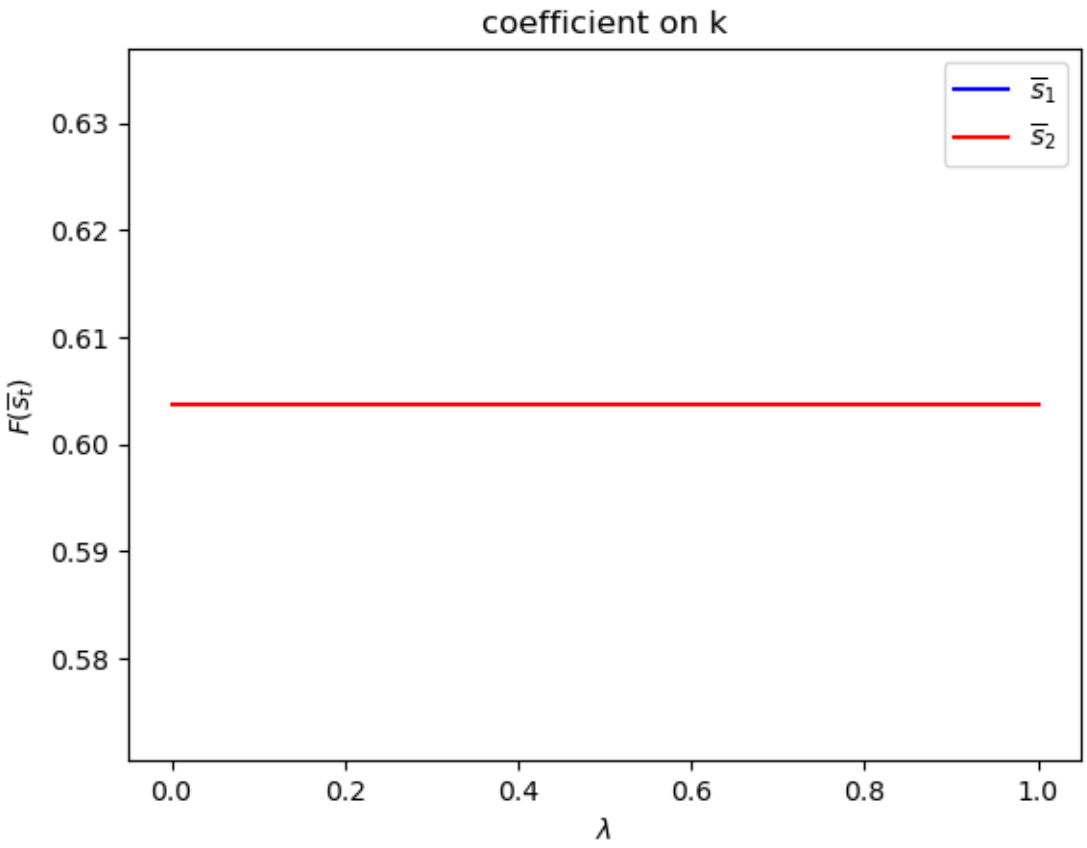

coefficient on constant term
0.15
0.10
0.05
0.00
−0.05
0.0
0.2
0.4
0.6
0.8
1.0
λ
1.0
0.8
0.6
0.4
0.2
0.0
ρ

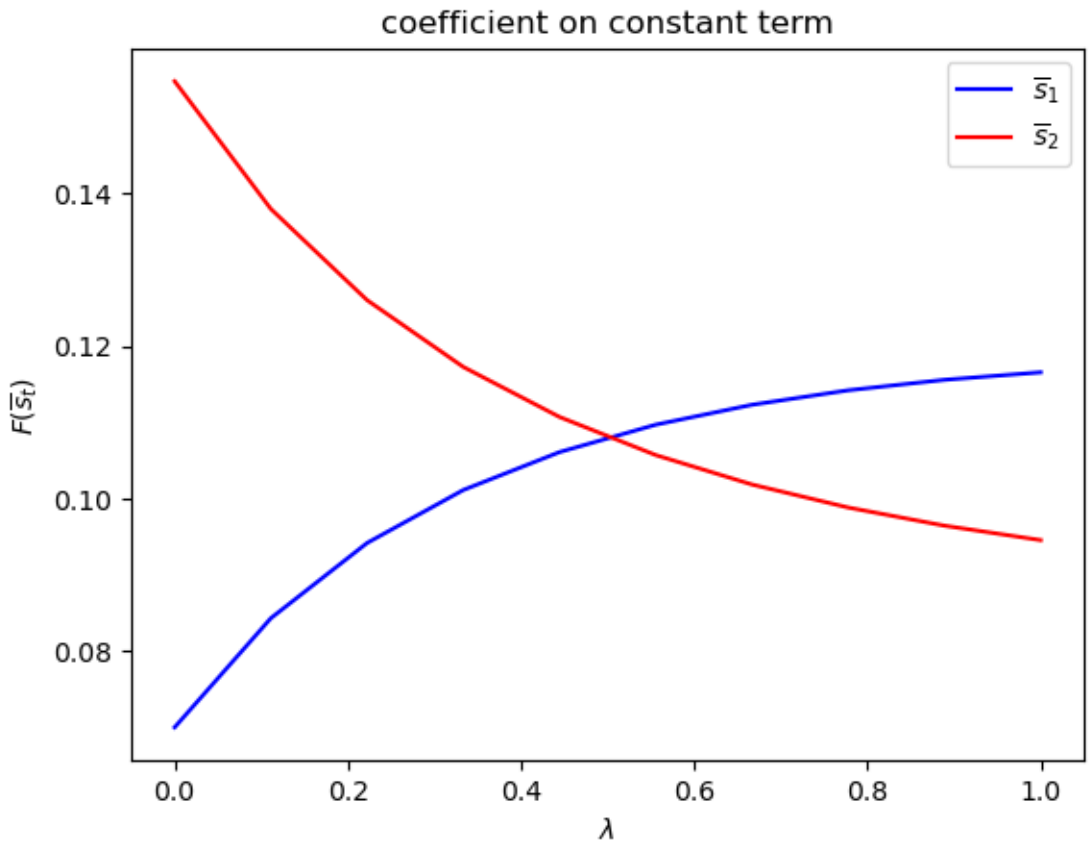


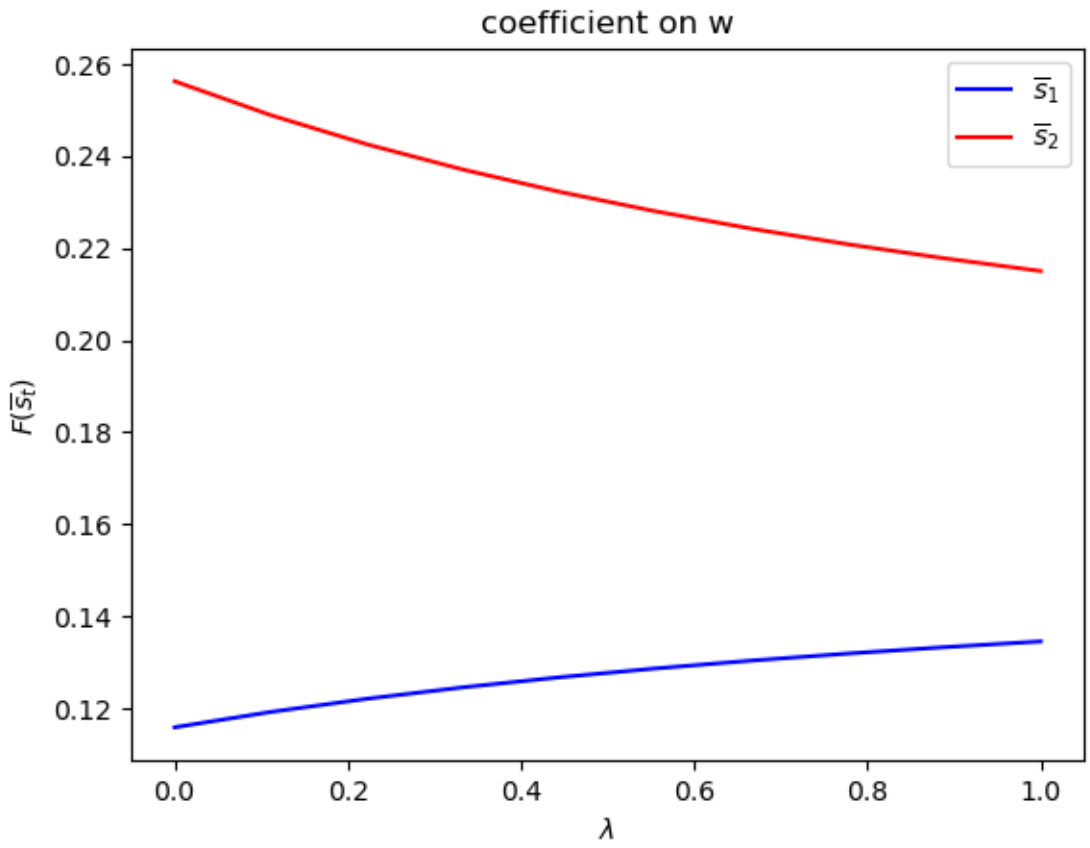
Only ρ_{s_t} depends on s_t.

```
run(construct_arrays2, {"ρ_vals":[0.5, 0.9]}, state_vec2)
```

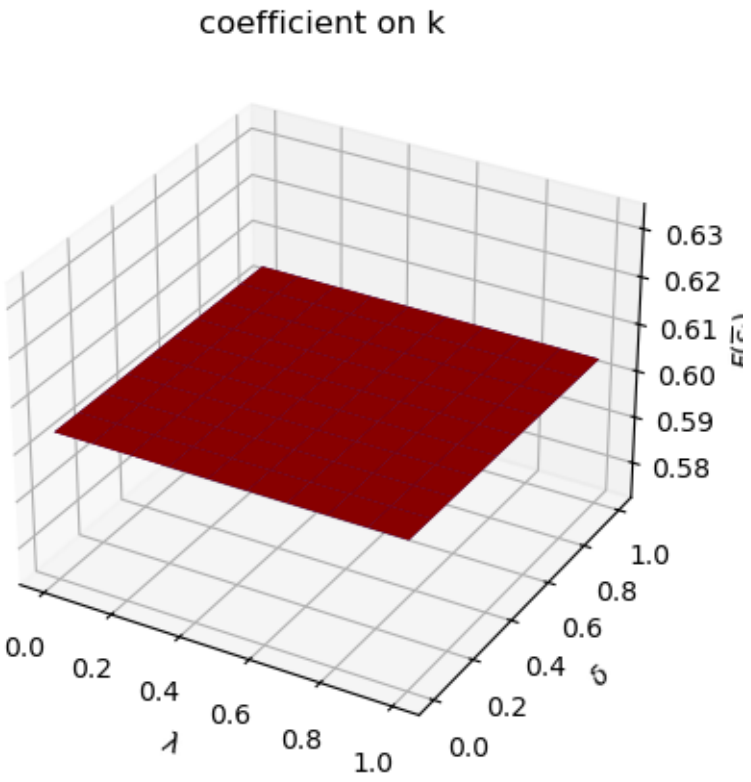
```
symmetric Π case:
```

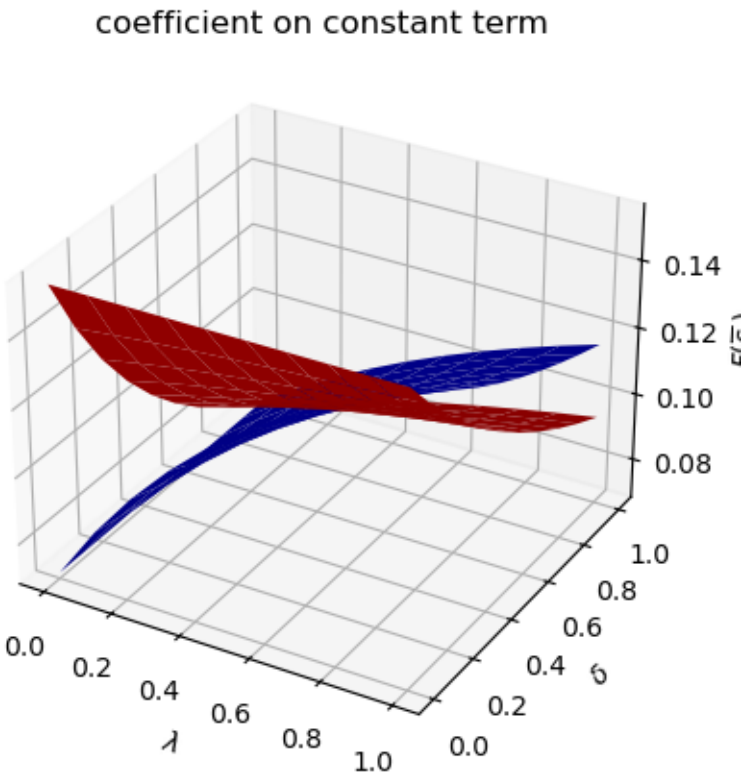

coefficient on k
$\overline{s}_1$
$\overline{s}_2$
0.63
0.62
0.61
0.60
0.59
0.58
$F(\overline{s}_t)$
0.0
0.2
0.4
0.6
0.8
1.0
λ

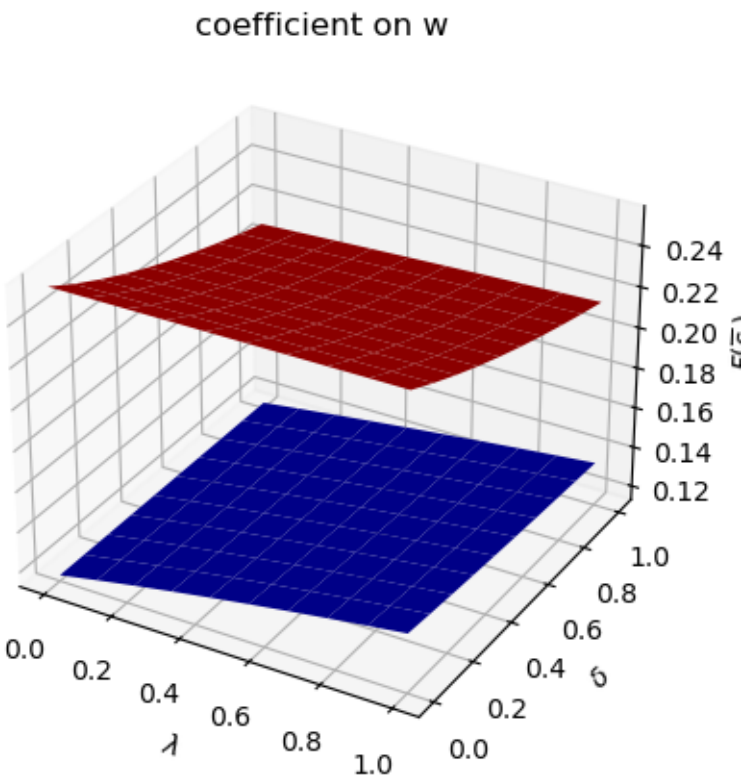

coefficient on constant term
$\overline{s}_1$
$\overline{s}_2$
0.14
0.12
0.10
0.08
$F(\overline{s}_t)$
0.0
0.2
0.4
0.6
0.8
1.0
λ




```
asymmetric Π case:
```


coefficient on k
0.63
0.62
0.61
0.60
0.59
0.58
0.0
0.2
0.4
0.6
0.8
1.0
λ
0.0
0.2
0.4
0.6
0.8
1.0
b

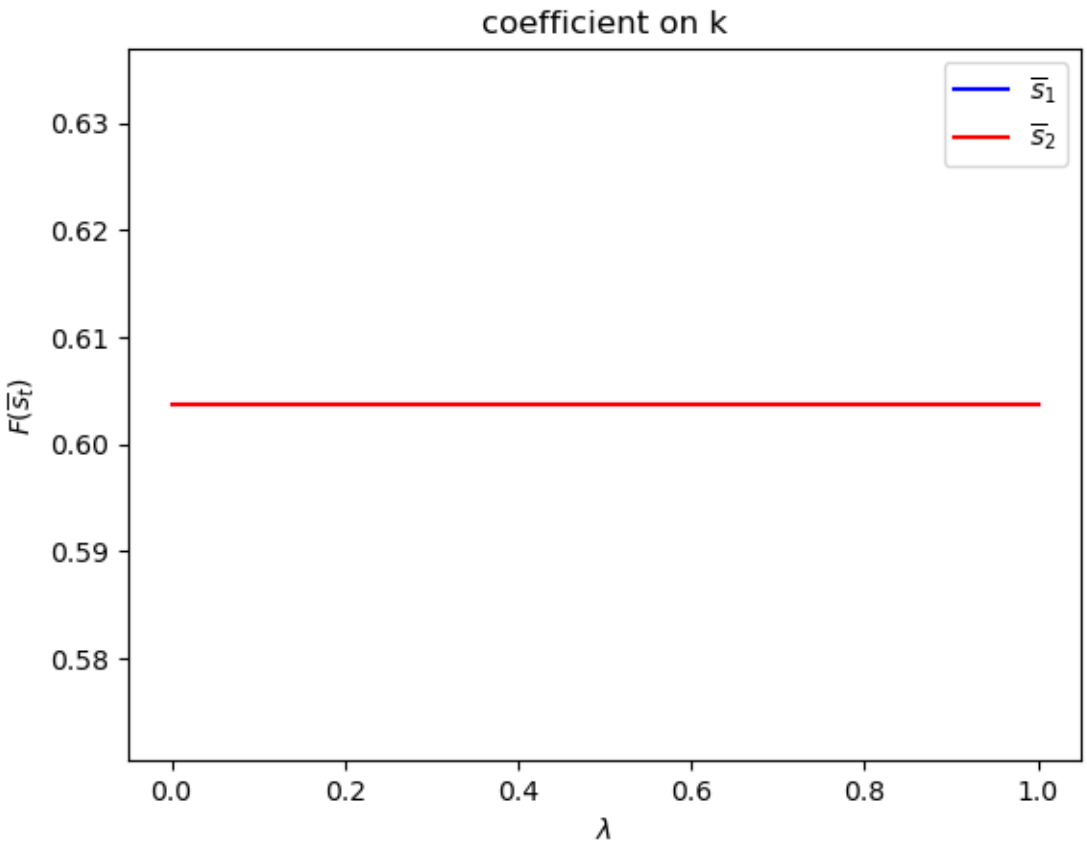

coefficient on constant term
0.14
0.12
0.10
0.08
0.0
0.2
0.4
0.6
0.8
1.0
λ
0.0
0.2
0.4
0.6
0.8
1.0
b

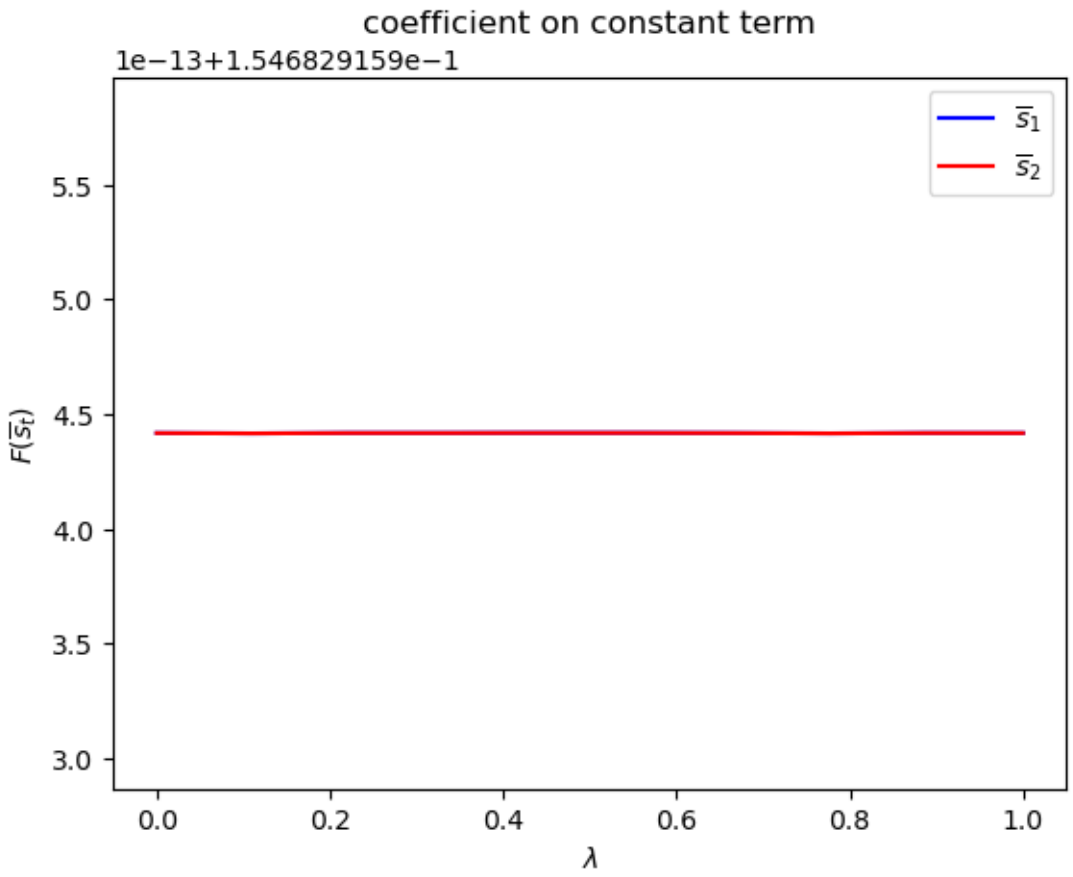


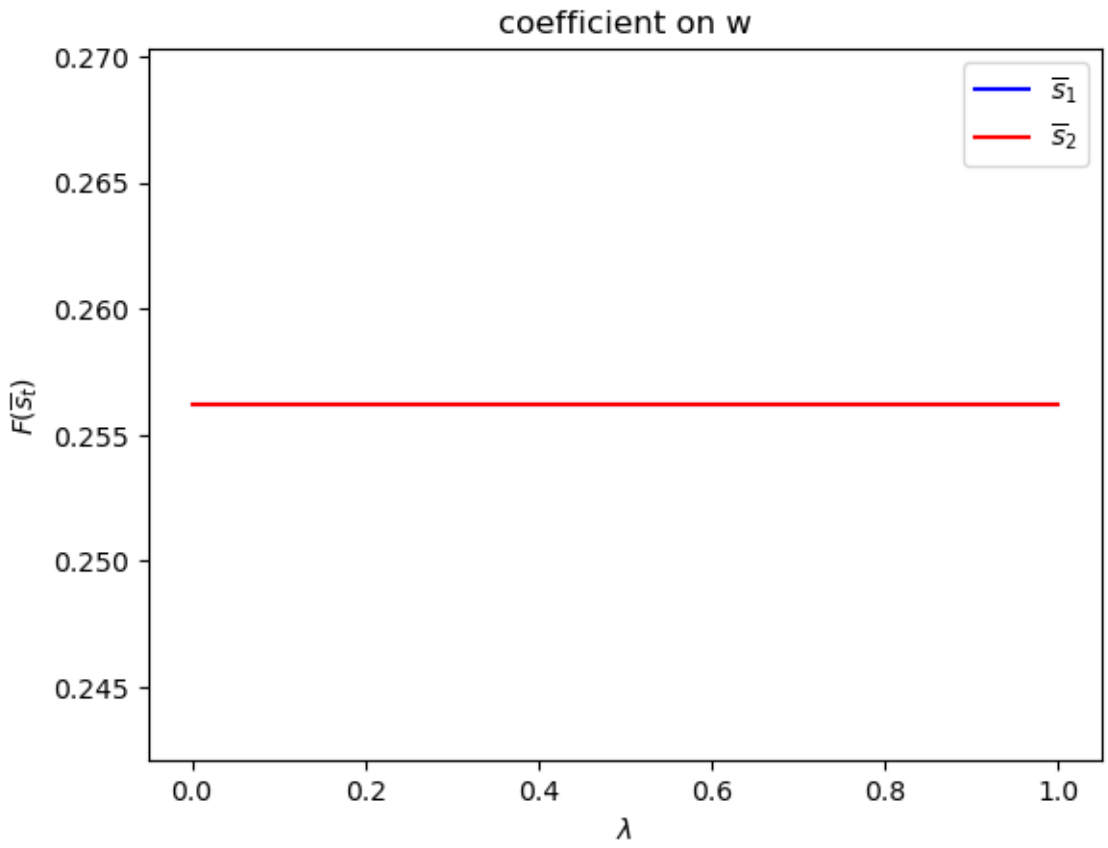
Only σ_{s_t} depends on s_t.

```
run(construct_arrays2, {"σ_vals":[0.5, 1.]}, state_vec2)
```

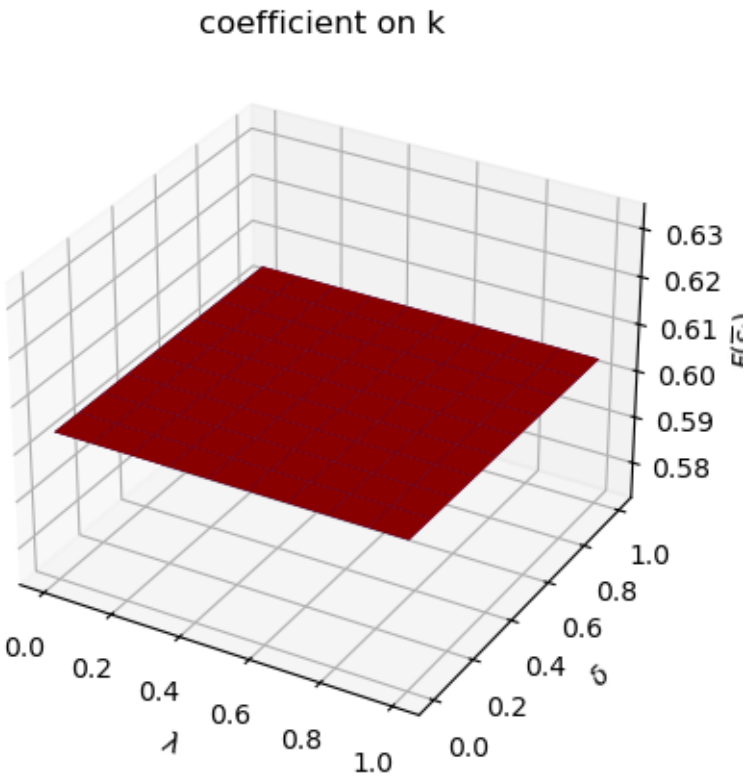
```
symmetric Π case:
```

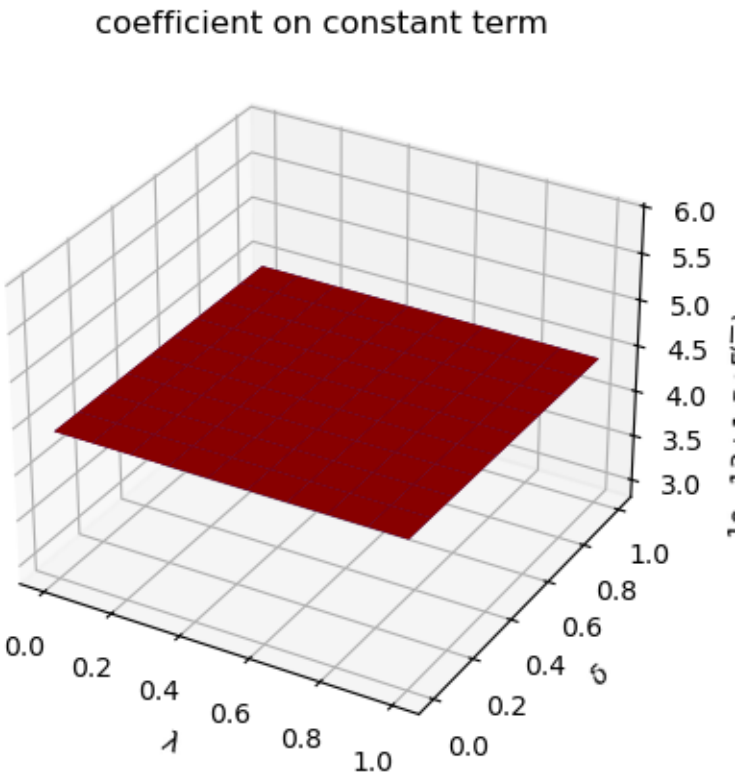

coefficient on k
$\overline{s}_1$
$\overline{s}_2$
0.63
0.62
0.61
0.60
0.59
0.58
$F(\overline{s}_t)$
0.0
0.2
0.4
0.6
0.8
1.0
λ

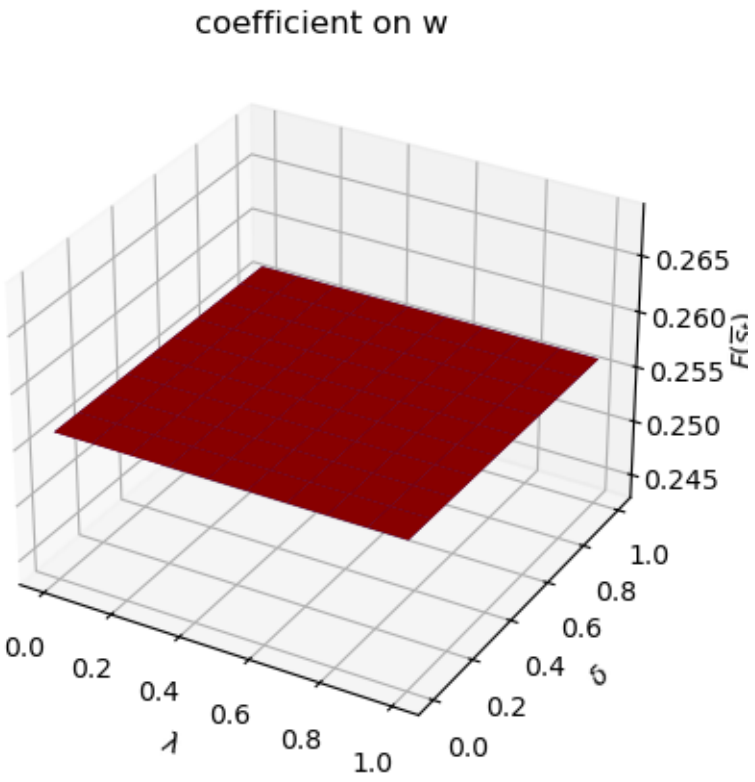

coefficient on constant term
1e−13+1.546829159e−1
$\overline{s}_1$
$\overline{s}_2$
5.5
5.0
4.5
4.0
3.5
3.0
$F(\overline{s}_t)$
0.0
0.2
0.4
0.6
0.8
1.0
λ




```
asymmetric Π case:
```


coefficient on k
0.63
0.62
0.61
0.60
0.59
0.58
0.0
0.2
0.4
0.6
0.8
1.0
λ
0.0
0.2
0.4
0.6
0.8
1.0
ρ


coefficient on constant term
6.0
5.5
5.0
4.5
4.0
3.5
3.0
0.0
0.2
0.4
0.6
0.8
1.0
λ
0.0
0.2
0.4
0.6
0.8
1.0
ρ



8.7 More examples

The following lectures describe how Markov jump linear quadratic dynamic programming can be used to extend the [Barro, 1979] model of optimal tax-smoothing and government debt in several interesting directions

1. *How to Pay for a War: Part 1*
2. *How to Pay for a War: Part 2*
3. *How to Pay for a War: Part 3*

CHAPTER

NINE

HOW TO PAY FOR A WAR: PART 1

9.1 Overview

This lecture uses the method of **Markov jump linear quadratic dynamic programming** that is described in lecture *Markov Jump LQ dynamic programming* to extend the [Barro, 1979] model of optimal tax-smoothing and government debt in a particular direction.

This lecture has two sequels that offer further extensions of the Barro model

1. *How to Pay for a War: Part 2*
2. *How to Pay for a War: Part 3*

The extensions are modified versions of his 1979 model suggested by [Barro, 1999] and [Barro and McCleary, 2003]).

[Barro, 1979] m is about a government that borrows and lends in order to minimize an intertemporal measure of distortions caused by taxes.

Technical tractability induced [Barro, 1979] to assume that

- the government trades only one-period risk-free debt, and
- the one-period risk-free interest rate is constant

By using *Markov jump linear quadratic dynamic programming* we can allow interest rates to move over time in empirically interesting ways.

Also, by expanding the dimension of the state, we can add a maturity composition decision to the government's problem.

By doing these two things we extend [Barro, 1979] along lines he suggested in [Barro, 1999] and [Barro and McCleary, 2003]).

[Barro, 1979] assumed

- that a government faces an **exogenous sequence** of expenditures that it must finance by a tax collection sequence whose expected present value equals the initial debt it owes plus the expected present value of those expenditures.
- that the government wants to minimize a measure of tax distortions that is proportional to $E_0 \sum_{t=0}^{\infty} \beta^t T_t^2$, where T_t are total tax collections and E_0 is a mathematical expectation conditioned on time 0 information.
- that the government trades only one asset, a risk-free one-period bond.
- that the gross interest rate on the one-period bond is constant and equal to β^{-1}, the reciprocal of the factor β at which the government discounts future tax distortions.

Barro's model can be mapped into a discounted linear quadratic dynamic programming problem.

Partly inspired by [Barro, 1999] and [Barro and McCleary, 2003], our generalizations of [Barro, 1979], assume

- that the government borrows or saves in the form of risk-free bonds of maturities $1, 2, \ldots, H$.

- that interest rates on those bonds are time-varying and in particular, governed by a jointly stationary stochastic process.

Our generalizations are designed to fit within a generalization of an ordinary linear quadratic dynamic programming problem in which matrices that define the quadratic objective function and the state transition function are **time-varying** and **stochastic**.

This generalization, known as a **Markov jump linear quadratic dynamic program**, combines

- the computational simplicity of **linear quadratic dynamic programming**, and
- the ability of **finite state Markov chains** to represent interesting patterns of random variation.

We want the stochastic time variation in the matrices defining the dynamic programming problem to represent variation over time in

- interest rates
- default rates
- roll over risks

As described in *Markov Jump LQ dynamic programming*, the idea underlying **Markov jump linear quadratic dynamic programming** is to replace the constant matrices defining a **linear quadratic dynamic programming problem** with matrices that are fixed functions of an N state Markov chain.

For infinite horizon problems, this leads to N interrelated matrix Riccati equations that pin down N value functions and N linear decision rules, applying to the N Markov states.

9.2 Public Finance Questions

[Barro, 1979] is designed to answer questions such as

- Should a government finance an exogenous surge in government expenditures by raising taxes or borrowing?
- How does the answer to that first question depend on the exogenous stochastic process for government expenditures, for example, on whether the surge in government expenditures can be expected to be temporary or permanent?

[Barro, 1999] and [Barro and McCleary, 2003] are designed to answer more fine-grained questions such as

- What determines whether a government wants to issue short-term or long-term debt?
- How do roll-over risks affect that decision?
- How does the government's long-short *portfolio management* decision depend on features of the exogenous stochastic process for government expenditures?

Thus, both the simple and the more fine-grained versions of Barro's models are ways of precisely formulating the classic issue of *How to pay for a war*.

This lecture describes:

- An application of Markov jump LQ dynamic programming to a model in which a government faces exogenous time-varying interest rates for issuing one-period risk-free debt.

A *sequel to this lecture* describes applies Markov LQ control to settings in which a government issues risk-free debt of different maturities.

```
!pip install --upgrade quantecon
```

Let's start with some standard imports:

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

9.3 Barro (1979) Model

We begin by solving a version of [Barro, 1979] by mapping it into the original LQ framework.

As mentioned in this lecture, the Barro model is mathematically isomorphic with the LQ permanent income model.

Let

- T_t denote tax collections
- β be a discount factor
- $b_{t,t+1}$ be time $t+1$ goods that at t the government promises to deliver to time t buyers of one-period government bonds
- G_t be government purchases
- $p_{t,t+1}$ the number of time t goods received per time $t+1$ goods promised to one-period bond purchasers.

Evidently, $p_{t,t+1}$ is inversely related to appropriate corresponding gross interest rates on government debt.

In the spirit of [Barro, 1979], the stochastic process of government expenditures is exogenous.

The government's problem is to choose a plan for taxation and borrowing $\{b_{t+1}, T_t\}_{t=0}^{\infty}$ to minimize

$$E_0 \sum_{t=0}^{\infty} \beta^t T_t^2$$

subject to the constraints

$$T_t + p_{t,t+1} b_{t,t+1} = G_t + b_{t-1,t}$$

$$G_t = U_g z_t$$

$$z_{t+1} = A_{22} z_t + C_2 w_{t+1}$$

where $w_{t+1} \sim N(0, I)$

The variables $T_t, b_{t,t+1}$ are *control* variables chosen at t, while $b_{t-1,t}$ is an endogenous state variable inherited from the past at time t and $p_{t,t+1}$ is an exogenous state variable at time t.

To begin, we assume that $p_{t,t+1}$ is constant (and equal to β)

- later we will extend the model to allow $p_{t,t+1}$ to vary over time

To map into the LQ framework, we use $x_t = \begin{bmatrix} b_{t-1,t} \\ z_t \end{bmatrix}$ as the state vector, and $u_t = b_{t,t+1}$ as the control variable.

Therefore, the (A, B, C) matrices are defined by the state-transition law:

$$x_{t+1} = \begin{bmatrix} 0 & 0 \\ 0 & A_{22} \end{bmatrix} x_t + \begin{bmatrix} 1 \\ 0 \end{bmatrix} u_t + \begin{bmatrix} 0 \\ C_2 \end{bmatrix} w_{t+1}$$

To find the appropriate (R, Q, W) matrices, we note that G_t and $b_{t-1,t}$ can be written as appropriately defined functions of the current state:

$$G_t = S_G x_t \quad , \quad b_{t-1,t} = S_1 x_t$$

If we define $M_t = -p_{t,t+1}$, and let $S = S_G + S_1$, then we can write taxation as a function of the states and control using the government's budget constraint:

$$T_t = Sx_t + M_t u_t$$

It follows that the (R, Q, W) matrices are implicitly defined by:

$$T_t^2 = x_t' S' S x_t + u_t' M_t' M_t u_t + 2u_t' M_t' S x_t$$

If we assume that $p_{t,t+1} = \beta$, then $M_t \equiv M = -\beta$.

In this case, none of the LQ matrices are time varying, and we can use the original LQ framework.

We will implement this constant interest-rate version first, assuming that G_t follows an AR(1) process:

$$G_{t+1} = \bar{G} + \rho G_t + \sigma w_{t+1}$$

To do this, we set $z_t = \begin{bmatrix} 1 \\ G_t \end{bmatrix}$, and consequently:

$$A_{22} = \begin{bmatrix} 1 & 0 \\ \bar{G} & \rho \end{bmatrix} \quad , \quad C_2 = \begin{bmatrix} 0 \\ \sigma \end{bmatrix}$$

```
# Model parameters
β, Gbar, ρ, σ = 0.95, 5, 0.8, 1

# Basic model matrices
A22 = np.array([[1,    0],
                [Gbar, ρ],])

C2 = np.array([[0],
               [σ]])

Ug = np.array([[0, 1]])

# LQ framework matrices
A_t = np.zeros((1, 3))
A_b = np.hstack((np.zeros((2, 1)), A22))
A = np.vstack((A_t, A_b))

B = np.zeros((3, 1))
B[0, 0] = 1

C = np.vstack((np.zeros((1, 1)), C2))

Sg = np.hstack((np.zeros((1, 1)), Ug))
S1 = np.zeros((1, 3))
S1[0, 0] = 1
S = S1 + Sg

M = np.array([[-β]])

R = S.T @ S
Q = M.T @ M
W = M.T @ S

# Small penalty on the debt required to implement the no-Ponzi scheme
R[0, 0] = R[0, 0] + 1e-9
```

We can now create an instance of `LQ`:

```
LQBarro = qe.LQ(Q, R, A, B, C=C, N=W, beta=β)
P, F, d = LQBarro.stationary_values()
x0 = np.array([[100, 1, 25]])
```

We can see the isomorphism by noting that consumption is a martingale in the permanent income model and that taxation is a martingale in Barro's model.

We can check this using the F matrix of the LQ model.

Because $u_t = -Fx_t$, we have

$$T_t = Sx_t + Mu_t = (S - MF)x_t$$

and

$$T_{t+1} = (S - MF)x_{t+1} = (S - MF)(Ax_t + Bu_t + Cw_{t+1}) = (S - MF)((A - BF)x_t + Cw_{t+1})$$

Therefore, the mathematical expectation of T_{t+1} conditional on time t information is

$$E_t T_{t+1} = (S - MF)(A - BF)x_t$$

Consequently, taxation is a martingale ($E_t T_{t+1} = T_t$) if

$$(S - MF)(A - BF) = (S - MF),$$

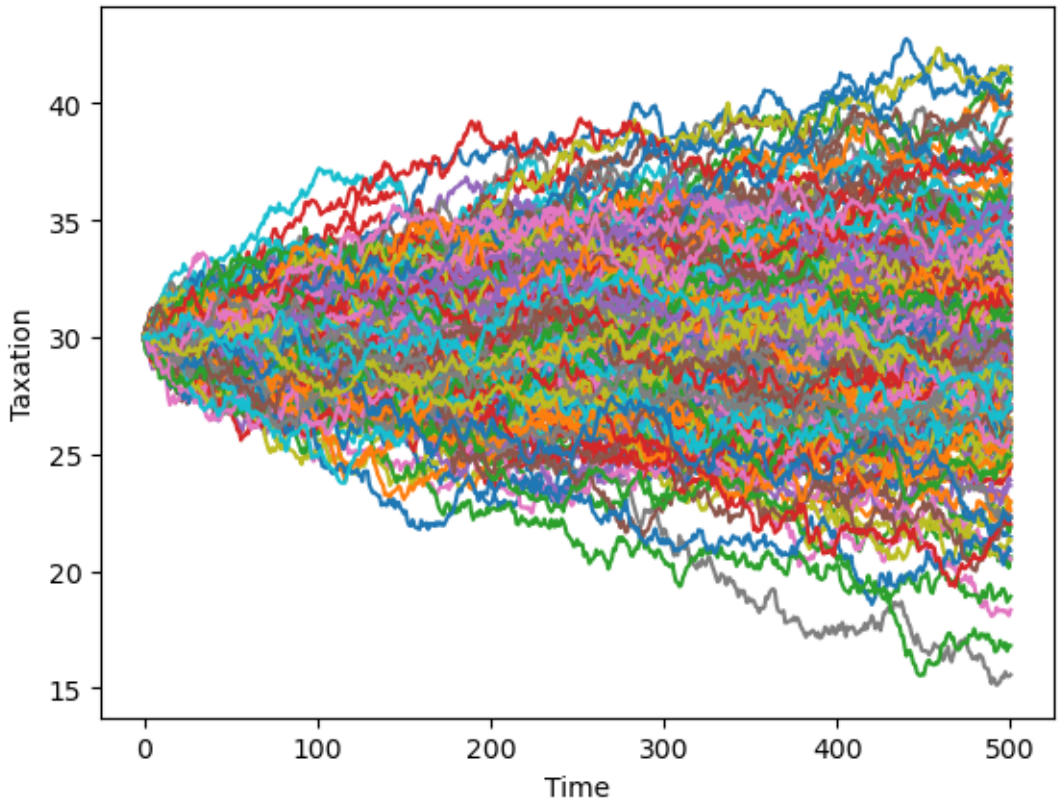
which holds in this case:

```
S - M @ F, (S - M @ F) @ (A - B @ F)
```

```
(array([[ 0.05000002, 19.79166502,  0.2083334 ]]),
 array([[ 0.05000002, 19.79166504,  0.2083334 ]]))
```

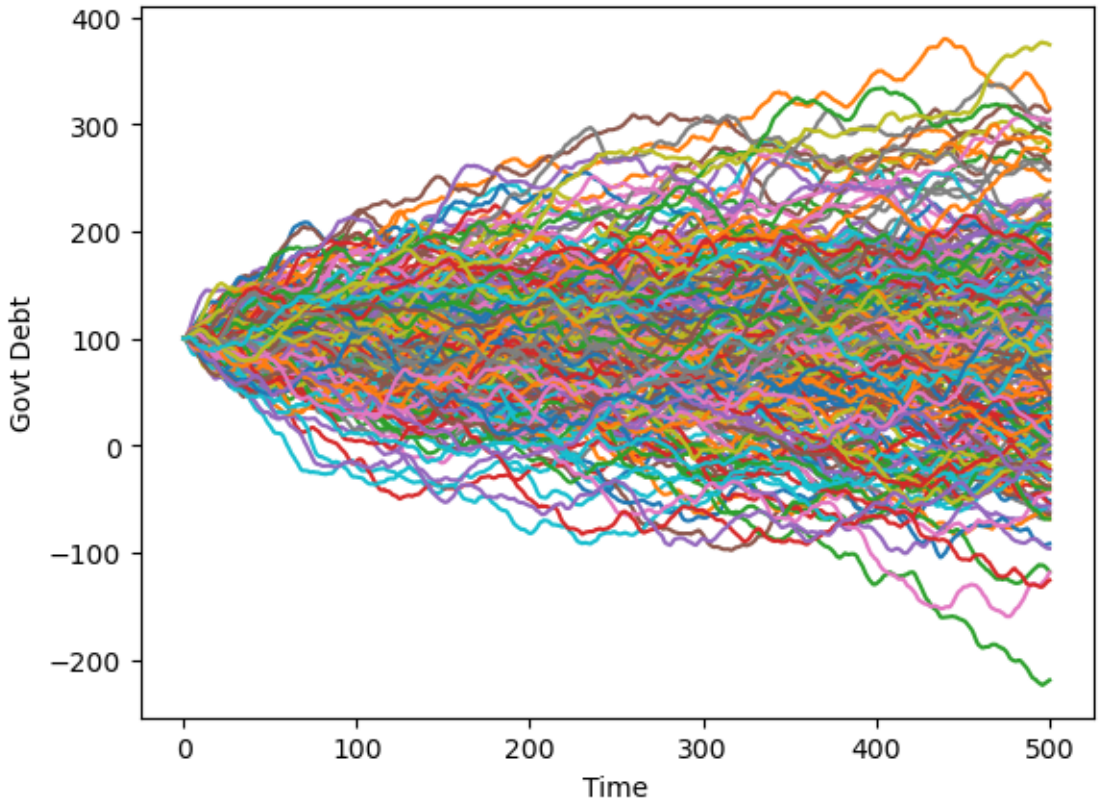
This explains the fanning out of the conditional empirical distribution of taxation across time, computed by simulating the Barro model many times and averaging over simulated paths:

```
T = 500
for i in range(250):
    x, u, w = LQBarro.compute_sequence(x0, ts_length=T)
    plt.plot(list(range(T+1)), ((S - M @ F) @ x)[0, :])
plt.xlabel('Time')
plt.ylabel('Taxation')
plt.show()
```



We can see a similar, but a smoother pattern, if we plot government debt over time.

```
T = 500
for i in range(250):
    x, u, w = LQBarro.compute_sequence(x0, ts_length=T)
    plt.plot(list(range(T+1)), x[0, :])
plt.xlabel('Time')
plt.ylabel('Govt Debt')
plt.show()
```



9.4 Python Class to Solve Markov Jump Linear Quadratic Control Problems

To implement the extension to the Barro model in which $p_{t,t+1}$ varies over time, we must allow the M matrix to be time-varying.

Our Q and W matrices must also vary over time.

We can solve such a model using the `LQMarkov` class that solves Markov jump linear quandratic control problems as described above.

The code for the class can be viewed here.

The class takes lists of matrices that corresponds to N Markov states.

The value and policy functions are then found by iterating on a coupled system of matrix Riccati difference equations.

Optimal P_s, F_s, d_s are stored as attributes.

The class also contains a method that simulates a model.

9.5 Barro Model with a Time-varying Interest Rate

We can use the above class to implement a version of the Barro model with a time-varying interest rate.

A simple way to extend the model is to allow the interest rate to take two possible values.

We set:

$$p^1_{t,t+1} = \beta + 0.02 = 0.97$$

$$p^2_{t,t+1} = \beta - 0.017 = 0.933$$

Thus, the first Markov state has a low interest rate and the second Markov state has a high interest rate.

We must also specify a transition matrix for the Markov state.

We use:

$$\Pi = \begin{bmatrix} 0.8 & 0.2 \\ 0.2 & 0.8 \end{bmatrix}$$

Here, each Markov state is persistent, and there is are equal chances of moving from one state to the other.

The choice of parameters means that the unconditional expectation of $p_{t,t+1}$ is 0.9515, higher than $\beta(= 0.95)$.

If we were to set $p_{t,t+1} = 0.9515$ in the version of the model with a constant interest rate, government debt would explode.

```
# Create list of matrices that corresponds to each Markov state
Π = np.array([[0.8, 0.2],
              [0.2, 0.8]])

As = [A, A]
Bs = [B, B]
Cs = [C, C]
Rs = [R, R]

M1 = np.array([[-β - 0.02]])
M2 = np.array([[-β + 0.017]])

Q1 = M1.T @ M1
Q2 = M2.T @ M2
Qs = [Q1, Q2]
W1 = M1.T @ S
W2 = M2.T @ S
Ws = [W1, W2]

# create Markov Jump LQ DP problem instance
lqm = qe.LQMarkov(Π, Qs, Rs, As, Bs, Cs=Cs, Ns=Ws, beta=β)
lqm.stationary_values();
```

The decision rules are now dependent on the Markov state:

```
lqm.Fs[0]
```

```
array([[-0.98437712, 19.20516427, -0.8314215 ]])
```

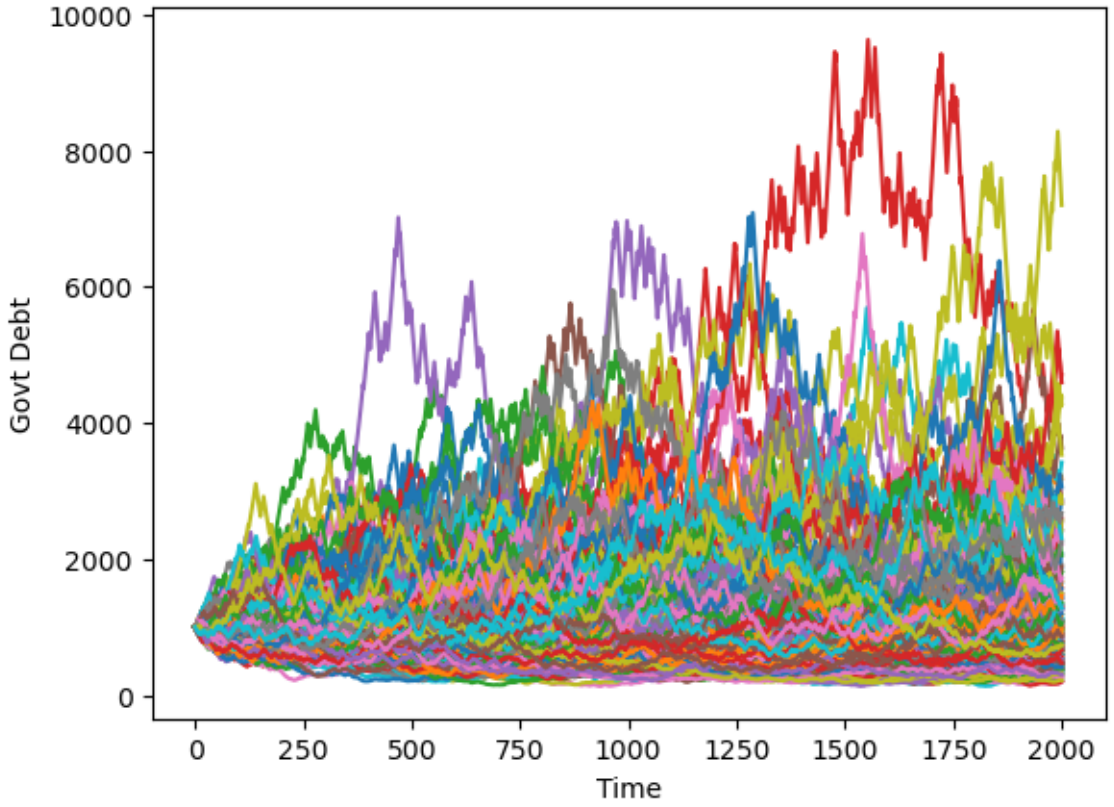
```
lqm.Fs[1]
```

```
array([[-1.01434301, 21.5847983 , -0.83851116]])
```

Simulating a large number of such economies over time reveals interesting dynamics.

Debt tends to stay low and stable but recurrently surges.

```
T = 2000
x0 = np.array([[1000, 1, 25]])
for i in range(250):
    x, u, w, s = lqm.compute_sequence(x0, ts_length=T)
    plt.plot(list(range(T+1)), x[0, :])
plt.xlabel('Time')
plt.ylabel('Govt Debt')
plt.show()
```



CHAPTER

TEN

HOW TO PAY FOR A WAR: PART 2

10.1 Overview

This lecture presents another application of Markov jump linear quadratic dynamic programming and constitutes a *sequel to an earlier lecture*.

We use a method introduced in lecture *Markov Jump LQ dynamic programming* toimplement suggestions by [Barro, 1999] and [Barro and McCleary, 2003]) for extending his classic 1979 model of tax smoothing.

[Barro, 1979] model is about a government that borrows and lends in order to help it minimize an intertemporal measure of distortions caused by taxes.

Technically, [Barro, 1979] model looks a lot like a consumption-smoothing model.

Our generalizations of [Barro, 1979] will also look like souped-up consumption-smoothing models.

Wanting tractability induced [Barro, 1979] to assume that

- the government trades only one-period risk-free debt, and
- the one-period risk-free interest rate is constant

In our *earlier lecture*, we relaxed the second of these assumptions but not the first.

In particular, we used *Markov jump linear quadratic dynamic programming* to allow the exogenous interest rate to vary over time.

In this lecture, we add a maturity composition decision to the government's problem by expanding the dimension of the state.

We assume

- that the government borrows or saves in the form of risk-free bonds of maturities $1, 2, \ldots, H$.
- that interest rates on those bonds are time-varying and in particular are governed by a jointly stationary stochastic process.

In addition to what's in Anaconda, this lecture deploys the quantecon library:

```
!pip install --upgrade quantecon
```

Let's start with some standard imports:

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

10.2 Two example specifications

We'll describe two possible specifications

- In one, each period the government issues zero-coupon bonds of one- and two-period maturities and redeems them only when they mature – in this version, the maturity structure of government debt at each date is partly inherited from the past.
- In the second, the government redesigns the maturity structure of the debt each period.

10.3 One- and Two-period Bonds but No Restructuring

Let

- T_t denote tax collections
- β be a discount factor
- $b_{t,t+1}$ be time $t+1$ goods that the government promises to pay at t
- $b_{t,t+2}$ betime $t+2$ goods that the government promises to pay at time t
- G_t be government purchases
- $p_{t,t+1}$ be the number of time t goods received per time $t+1$ goods promised
- $p_{t,t+2}$ be the number of time t goods received per time $t+2$ goods promised.

Evidently, $p_{t,t+1}, p_{t,t+2}$ are inversely related to appropriate corresponding gross interest rates on government debt.

In the spirit of [Barro, 1979], government expenditures are governed by an exogenous stochastic process.

Given initial conditions $b_{-2,0}, b_{-1,0}, z_0, i_0$, where i_0 is the initial Markov state, the government chooses a contingency plan for $\{b_{t,t+1}, b_{t,t+2}, T_t\}_{t=0}^{\infty}$ to maximize.

$$-E_0 \sum_{t=0}^{\infty} \beta^t \left[T_t^2 + c_1 (b_{t,t+1} - b_{t,t+2})^2\right]$$

subject to the constraints

$$\begin{aligned} T_t &= G_t + b_{t-2,t} + b_{t-1,t} - p_{t,t+2} b_{t,t+2} - p_{t,t+1} b_{t,t+1} \\ G_t &= U_{g,s_t} z_t \\ z_{t+1} &= A_{22,s_t} z_t + C_{2,s_t} w_{t+1} \\ \begin{bmatrix} p_{t,t+1} \\ p_{t,t+2} \\ U_{g,s_t} \\ A_{22,s_t} \\ C_{2,s_t} \end{bmatrix} &\sim \text{functions of Markov state with transition matrix } \Pi \end{aligned}$$

Here

- $w_{t+1} \sim N(0, I)$ and Π_{ij} is the probability that the Markov state moves from state i to state j in one period
- $T_t, b_{t,t+1}, b_{t,t+2}$ are *control* variables chosen at time t
- variables $b_{t-1,t}, b_{t-2,t}$ are endogenous state variables inherited from the past at time t
- $p_{t,t+1}, p_{t,t+2}$ are exogenous state variables at time t

The parameter c_1 imposes a penalty on the government's issuing different quantities of one and two-period debt.

This penalty deters the government from taking large "long-short" positions in debt of different maturities.

An example below will show the penalty in action.

As well as extending the model to allow for a maturity decision for government debt, we can also in principle allow the matrices $U_{g,s_t}, A_{22,s_t}, C_{2,s_t}$ to depend on the Markov state s_t.

Below, we will often adopt the convention that for matrices appearing in a linear state space, $A_t \equiv A_{s_t}, C_t \equiv C_{s_t}$ and so on, so that dependence on t is always intermediated through the Markov state s_t.

10.4 Mapping into an LQ Markov Jump Problem

First, define

$$\hat{b}_t = b_{t-1,t} + b_{t-2,t},$$

which is debt due at time t.

Then define the endogenous part of the state:

$$\bar{b}_t = \begin{bmatrix} \hat{b}_t \\ b_{t-1,t+1} \end{bmatrix}$$

and the complete state vector

$$x_t = \begin{bmatrix} \bar{b}_t \\ z_t \end{bmatrix}$$

and the control vector

$$u_t = \begin{bmatrix} b_{t,t+1} \\ b_{t,t+2} \end{bmatrix}$$

The endogenous part of state vector follows the law of motion:

$$\begin{bmatrix} \hat{b}_{t+1} \\ b_{t,t+2} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \hat{b}_t \\ b_{t-1,t+1} \end{bmatrix} + \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} b_{t,t+1} \\ b_{t,t+2} \end{bmatrix}$$

or

$$\bar{b}_{t+1} = A_{11}\bar{b}_t + B_1 u_t$$

Define the following functions of the state

$$G_t = S_{G,t} x_t, \quad \hat{b}_t = S_1 x_t$$

and

$$M_t = \begin{bmatrix} -p_{t,t+1} & -p_{t,t+2} \end{bmatrix}$$

where $p_{t,t+1}$ is the discount on one period loans in the discrete Markov state at time t and $p_{t,t+2}$ is the discount on two-period loans in the discrete Markov state.

Define

$$S_t = S_{G,t} + S_1$$

Note that in discrete Markov state i

$$T_t = M_t u_t + S_t x_t$$

It follows that

$$T_t^2 = x_t' S_t' S_t x_t + u_t' M_t' M_t u_t + 2 u_t' M_t' S_t x_t$$

or

$$T_t^2 = x_t' R_t x_t + u_t' Q_t u_t + 2 u_t' W_t x_t$$

where

$$R_t = S_t' S_t, \quad Q_t = M_t' M_t, \quad W_t = M_t' S_t$$

Because the payoff function also includes the penalty parameter on issuing debt of different maturities, we have:

$$T_t^2 + c_1 (b_{t,t+1} - b_{t,t+2})^2 = x_t' R_t x_t + u_t' Q_t u_t + 2 u_t' W_t x_t + c_1 u_t' Q^c u_t$$

where $Q^c = \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}$.

Therefore, the appropriate Q matrix in the Markov jump LQ problem is:

$$Q_t^c = Q_t + c_1 Q^c$$

The law of motion of the state in all discrete Markov states i is

$$x_{t+1} = A_t x_t + B u_t + C_t w_{t+1}$$

where

$$A_t = \begin{bmatrix} A_{11} & 0 \\ 0 & A_{22,t} \end{bmatrix}, \quad B = \begin{bmatrix} B_1 \\ 0 \end{bmatrix}, \quad C_t = \begin{bmatrix} 0 \\ C_{2,t} \end{bmatrix}$$

Thus, in this problem all the matrices apart from B may depend on the Markov state at time t.

As shown in the *previous lecture*, when provided with appropriate A, B, C, R, Q, W matrices for each Markov state the `LQMarkov` class can solve Markov jump LQ problems.

The function below maps the primitive matrices and parameters from the above two-period model into the matrices that the `LQMarkov` class requires:

```
def LQ_markov_mapping(A22, C2, Ug, p1, p2, c1=0):

    """
    Function which takes A22, C2, Ug, p_{t, t+1}, p_{t, t+2} and penalty
    parameter c1, and returns the required matrices for the LQMarkov
    model: A, B, C, R, Q, W.
    This version uses the condensed version of the endogenous state.
    """

    # Make sure all matrices can be treated as 2D arrays
    A22 = np.atleast_2d(A22)
    C2 = np.atleast_2d(C2)
    Ug = np.atleast_2d(Ug)
    p1 = np.atleast_2d(p1)
```

(continues on next page)

(continued from previous page)

```
    p2 = np.atleast_2d(p2)

    # Find the number of states (z) and shocks (w)
    nz, nw = C2.shape

    # Create A11, B1, S1, S2, Sg, S matrices
    A11 = np.zeros((2, 2))
    A11[0, 1] = 1

    B1 = np.eye(2)

    S1 = np.hstack((np.eye(1), np.zeros((1, nz+1))))
    Sg = np.hstack((np.zeros((1, 2)), Ug))
    S = S1 + Sg

    # Create M matrix
    M = np.hstack((-p1, -p2))

    # Create A, B, C matrices
    A_T = np.hstack((A11, np.zeros((2, nz))))
    A_B = np.hstack((np.zeros((nz, 2)), A22))
    A = np.vstack((A_T, A_B))

    B = np.vstack((B1, np.zeros((nz, 2))))

    C = np.vstack((np.zeros((2, nw)), C2))

    # Create Q^c matrix
    Qc = np.array([[1, -1], [-1, 1]])

    # Create R, Q, W matrices

    R = S.T @ S
    Q = M.T @ M + c1 * Qc
    W = M.T @ S

    return A, B, C, R, Q, W
```

With the above function, we can proceed to solve the model in two steps:

1. Use `LQ_markov_mapping` to map $U_{g,t}, A_{22,t}, C_{2,t}, p_{t,t+1}, p_{t,t+2}$ into the A, B, C, R, Q, W matrices for each of the n Markov states.
2. Use the `LQMarkov` class to solve the resulting n-state Markov jump LQ problem.

10.5 Penalty on Different Issues Across Maturities

To implement a simple example of the two-period model, we assume that G_t follows an AR(1) process:

$$G_{t+1} = \bar{G} + \rho G_t + \sigma w_{t+1}$$

To do this, we set $z_t = \begin{bmatrix} 1 \\ G_t \end{bmatrix}$, and consequently:

$$A_{22} = \begin{bmatrix} 1 & 0 \\ \bar{G} & \rho \end{bmatrix} \quad , \quad C_2 = \begin{bmatrix} 0 \\ \sigma \end{bmatrix} \quad , \quad U_g = \begin{bmatrix} 0 & 1 \end{bmatrix}$$

Therefore, in this example, A_{22}, C_2 and U_g are not time-varying.

We will assume that there are two Markov states, one with a flatter yield curve, and one with a steeper yield curve.

In state 1, prices are:

$$p^1_{t,t+1} = \beta \ , \ p^1_{t,t+2} = \beta^2 - 0.02$$

and in state 2, prices are:

$$p^2_{t,t+1} = \beta \ , \ p^2_{t,t+2} = \beta^2 + 0.02$$

We first solve the model with no penalty parameter on different issuance across maturities, i.e. $c_1 = 0$.

We specify that the transition matrix for the Markov state is

$$\Pi = \begin{bmatrix} 0.9 & 0.1 \\ 0.1 & 0.9 \end{bmatrix}$$

Thus, each Markov state is persistent, and there is an equal chance of moving from one to the other.

```
# Model parameters
β, Gbar, ρ, σ, c1 = 0.95, 5, 0.8, 1, 0
p1, p2, p3, p4 = β, β**2 - 0.02, β, β**2 + 0.02

# Basic model matrices
A22 = np.array([[1, 0], [Gbar, ρ] ,])
C_2 = np.array([[0], [σ]])
Ug = np.array([[0, 1]])

A1, B1, C1, R1, Q1, W1 = LQ_markov_mapping(A22, C_2, Ug, p1, p2, c1)
A2, B2, C2, R2, Q2, W2 = LQ_markov_mapping(A22, C_2, Ug, p3, p4, c1)

# Small penalties on debt required to implement no-Ponzi scheme
R1[0, 0] = R1[0, 0] + 1e-9
R2[0, 0] = R2[0, 0] + 1e-9

# Construct lists of matrices correspond to each state
As = [A1, A2]
Bs = [B1, B2]
Cs = [C1, C2]
Rs = [R1, R2]
Qs = [Q1, Q2]
Ws = [W1, W2]

Π = np.array([[0.9, 0.1],
              [0.1, 0.9]])

# Construct and solve the model using the LQMarkov class
lqm = qe.LQMarkov(Π, Qs, Rs, As, Bs, Cs=Cs, Ns=Ws, beta=β)
lqm.stationary_values()

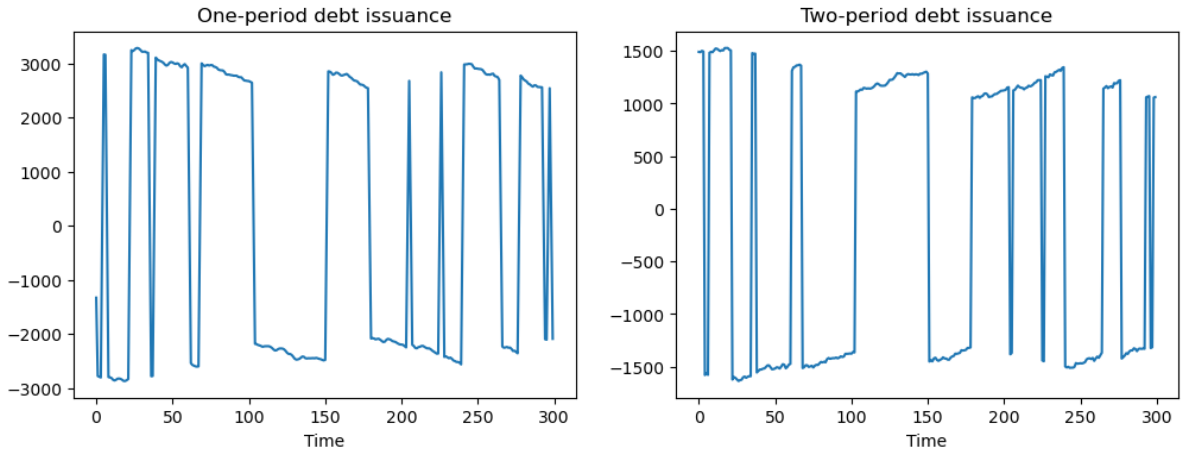
# Simulate the model
x0 = np.array([[100, 50, 1, 10]])
x, u, w, t = lqm.compute_sequence(x0, ts_length=300)

# Plot of one and two-period debt issuance
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))
ax1.plot(u[0, :])
```

(continues on next page)

(continued from previous page)

```
ax1.set_title('One-period debt issuance')
ax1.set_xlabel('Time')
ax2.plot(u[1, :])
ax2.set_title('Two-period debt issuance')
ax2.set_xlabel('Time')
plt.show()
```



The above simulations show that when no penalty is imposed on different issuances across maturities, the government has an incentive to take large "long-short" positions in debt of different maturities.

To prevent such outcomes, we set $c_1 = 0.01$.

This penalty is big enough to motivate the government to issue positive quantities of both one- and two-period debt:

```
# Put small penalty on different issuance across maturities
c1 = 0.01

A1, B1, C1, R1, Q1, W1 = LQ_markov_mapping(A22, C_2, Ug, p1, p2, c1)
A2, B2, C2, R2, Q2, W2 = LQ_markov_mapping(A22, C_2, Ug, p3, p4, c1)

# Small penalties on debt required to implement no-Ponzi scheme
R1[0, 0] = R1[0, 0] + 1e-9
R2[0, 0] = R2[0, 0] + 1e-9

# Construct lists of matrices
As = [A1, A2]
Bs = [B1, B2]
Cs = [C1, C2]
Rs = [R1, R2]
Qs = [Q1, Q2]
Ws = [W1, W2]

# Construct and solve the model using the LQMarkov class
lqm2 = qe.LQMarkov(Π, Qs, Rs, As, Bs, Cs=Cs, Ns=Ws, beta=β)
lqm2.stationary_values()

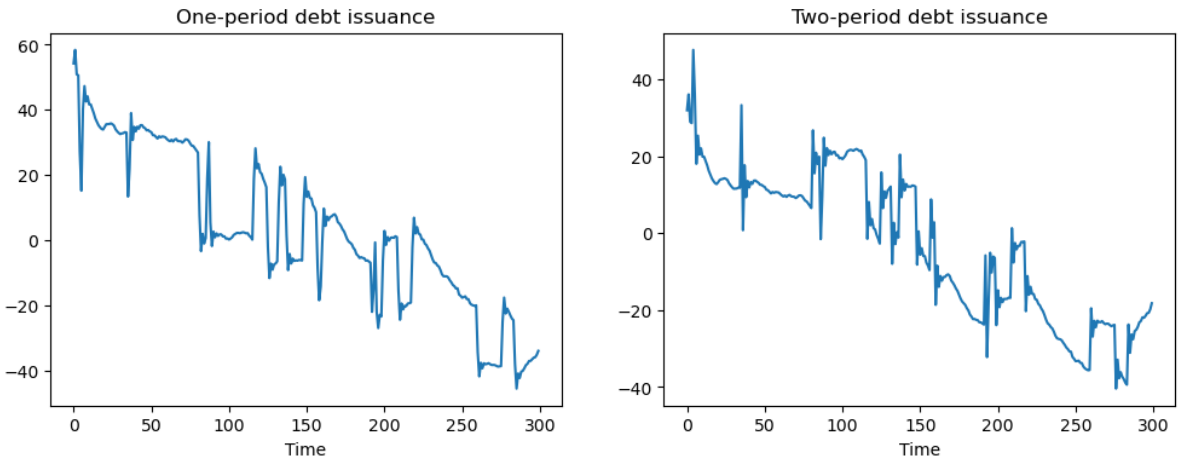
# Simulate the model
x, u, w, t = lqm2.compute_sequence(x0, ts_length=300)

# Plot of one and two-period debt issuance
```

(continues on next page)

(continued from previous page)

```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))
ax1.plot(u[0, :])
ax1.set_title('One-period debt issuance')
ax1.set_xlabel('Time')
ax2.plot(u[1, :])
ax2.set_title('Two-period debt issuance')
ax2.set_xlabel('Time')
plt.show()
```



10.6 A Model with Restructuring

We now alter two features of the previous model:

1. The maximum horizon of government debt is now extended to a general H periods.
2. The government is able to redesign the maturity structure of debt every period.

We impose a cost on adjusting issuance of each maturity by amending the payoff function to become:

$$T_t^2 + \sum_{j=0}^{H-1} c_2 (b_{t+j}^{t-1} - b_{t+j+1}^t)^2$$

The government's budget constraint is now:

$$T_t + \sum_{j=1}^{H} p_{t,t+j} b_{t+j}^t = b_t^{t-1} + \sum_{j=1}^{H-1} p_{t,t+j} b_{t+j}^{t-1} + G_t$$

To map this into the Markov Jump LQ framework, we define state and control variables.

Let:

$$\bar{b}_t = \begin{bmatrix} b_t^{t-1} \\ b_{t+1}^{t-1} \\ \vdots \\ b_{t+H-1}^{t-1} \end{bmatrix} \quad , \quad u_t = \begin{bmatrix} b_{t+1}^t \\ b_{t+2}^t \\ \vdots \\ b_{t+H}^t \end{bmatrix}$$

Thus, $\bar{b}_t$ is the endogenous state (debt issued last period) and u_t is the control (debt issued today).

As before, we will also have the exogenous state z_t, which determines government spending.

Therefore, the full state is:

$$x_t = \begin{bmatrix} \bar{b}_t \\ z_t \end{bmatrix}$$

We also define a vector p_t that contains the time t price of goods in period $t+j$:

$$p_t = \begin{bmatrix} p_{t,t+1} \\ p_{t,t+2} \\ \vdots \\ p_{t,t+H} \end{bmatrix}$$

Finally, we define three useful matrices $S_s, S_x, \tilde{S}_x$:

$$\begin{bmatrix} p_{t,t+1} \\ p_{t,t+2} \\ \vdots \\ p_{t,t+H-1} \end{bmatrix} = S_s p_t \text{ where } S_s = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ \vdots & & \ddots & & \\ 0 & 0 & \cdots & 1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} b_{t+1}^{t-1} \\ b_{t+2}^{t-1} \\ \vdots \\ b_{t+T-1}^{t-1} \end{bmatrix} = S_x \bar{b}_t \text{ where } S_x = \begin{bmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & & & \ddots & \\ 0 & 0 & \cdots & 0 & 1 \end{bmatrix}$$

$$b_t^{t-1} = \tilde{S}_x \bar{b}_t \text{ where } \tilde{S}_x = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \end{bmatrix}$$

In terms of dimensions, the first two matrices defined above are $(H-1) \times H$.

The last is $1 \times H$

We can now write the government's budget constraint in matrix notation.

We can rearrange the government budget constraint to become

$$T_t = b_t^{t-1} + \sum_{j=1}^{H-1} p_{t+j}^t b_{t+j}^{t-1} + G_t - \sum_{j=1}^{H} p_{t+j}^t b_{t+j}^t$$

or

$$T_t = \tilde{S}_x \bar{b}_t + (S_s p_t) \cdot (S_x \bar{b}_t) + U_g z_t - p_t \cdot u_t$$

To express T_t as a function of the full state, let

$$T_t = \begin{bmatrix} (\tilde{S}_x + p_t' S_s' S_x) & Ug \end{bmatrix} x_t - p_t' u_t$$

To simplify the notation, let $S_t = \begin{bmatrix} (\tilde{S}_x + p_t\text{'}S_s\text{'}S_x) & Ug \end{bmatrix}$.

Then

$$T_t = S_t x_t - p_t' u_t$$

Therefore

$$T_t^2 = x_t' R_t x_t + u_t' Q_t u_t + 2 u_t' W_t x_t$$

where

$$R_t = S_t' S_t, \quad Q_t = p_t p_t', \quad W_t = -p_t S_t$$

where to economize on notation we adopt the convention that for the linear state matrices $R_t \equiv R_{s_t}, Q_t \equiv W_{s_t}$ and so on.

We'll use this convention for the linear state matrices A, B, W and so on below.

Because the payoff function also includes the penalty parameter for rescheduling, we have:

$$T_t^2 + \sum_{j=0}^{H-1} c_2 (b_{t+j}^{t-1} - b_{t+j+1}^t)^2 = T_t^2 + c_2 (\bar{b}_t - u_t)'(\bar{b}_t - u_t)$$

Because the complete state is x_t and not $\bar{b}_t$, we rewrite this as:

$$T_t^2 + c_2 (S_c x_t - u_t)'(S_c x_t - u_t)$$

where $S_c = \begin{bmatrix} I & 0 \end{bmatrix}$

Multiplying this out gives:

$$T_t^2 + c_2 x_t' S_c' S_c x_t - 2 c_2 u_t' S_c x_t + c_2 u_t' u_t$$

Therefore, with the cost term, we must amend our R, Q, W matrices as follows:

$$R_t^c = R_t + c_2 S_c' S_c$$

$$Q_t^c = Q_t + c_2 I$$

$$W_t^c = W_t - c_2 S_c$$

To finish mapping into the Markov jump LQ setup, we need to construct the law of motion for the full state.

This is simpler than in the previous setup, as we now have $\bar{b}_{t+1} = u_t$.

Therefore:

$$x_{t+1} \equiv \begin{bmatrix} \bar{b}_{t+1} \\ z_{t+1} \end{bmatrix} = A_t x_t + B u_t + C_t w_{t+1}$$

where

$$A_t = \begin{bmatrix} 0 & 0 \\ 0 & A_{22,t} \end{bmatrix}, \quad B = \begin{bmatrix} I \\ 0 \end{bmatrix}, \quad C = \begin{bmatrix} 0 \\ C_{2,t} \end{bmatrix}$$

This completes the mapping into a Markov jump LQ problem.

10.7 Restructuring as a Markov Jump Linear Quadratic Control Problem

We can define a function that maps the primitives of the model with restructuring into the matrices required by the `LQMarkov` class:

```
def LQ_markov_mapping_restruct(A22, C2, Ug, T, p_t, c=0):

    """
    Function which takes A22, C2, T, p_t, c and returns the
    required matrices for the LQMarkov model: A, B, C, R, Q, W
    Note, p_t should be a T by 1 matrix
```

(continues on next page)

(continued from previous page)

```
    c is the rescheduling cost (a scalar)
    This version uses the condensed version of the endogenous state
    """

    # Make sure all matrices can be treated as 2D arrays
    A22 = np.atleast_2d(A22)
    C2 = np.atleast_2d(C2)
    Ug = np.atleast_2d(Ug)
    p_t = np.atleast_2d(p_t)

    # Find the number of states (z) and shocks (w)
    nz, nw = C2.shape

    # Create Sx, tSx, Ss, S_t matrices (tSx stands for \tilde S_x)
    Ss = np.hstack((np.eye(T-1), np.zeros((T-1, 1))))
    Sx = np.hstack((np.zeros((T-1, 1)), np.eye(T-1)))
    tSx = np.zeros((1, T))
    tSx[0, 0] = 1

    S_t = np.hstack((tSx + p_t.T @ Ss.T @ Sx, Ug))

    # Create A, B, C matrices
    A_T = np.hstack((np.zeros((T, T)), np.zeros((T, nz))))
    A_B = np.hstack((np.zeros((nz, T)), A22))
    A = np.vstack((A_T, A_B))

    B = np.vstack((np.eye(T), np.zeros((nz, T))))
    C = np.vstack((np.zeros((T, nw)), C2))

    # Create cost matrix Sc
    Sc = np.hstack((np.eye(T), np.zeros((T, nz))))

    # Create R_t, Q_t, W_t matrices

    R_c = S_t.T @ S_t + c * Sc.T @ Sc
    Q_c = p_t @ p_t.T + c * np.eye(T)
    W_c = -p_t @ S_t - c * Sc

    return A, B, C, R_c, Q_c, W_c
```

10.7.1 Example with Restructuring

As an example let $H = 3$.

Assume that there are two Markov states, one with a flatter yield curve, the other with a steeper yield curve.

In state 1, prices are:

$$p^1_{t,t+1} = 0.9695 \ , \ p^1_{t,t+2} = 0.902 \ , \ p^1_{t,t+3} = 0.8369$$

and in state 2, prices are:

$$p^2_{t,t+1} = 0.9295 \ , \ p^2_{t,t+2} = 0.902 \ , \ p^2_{t,t+3} = 0.8769$$

We specify the same transition matrix and G_t process that we used earlier.

```
# New model parameters
H = 3
p1 = np.array([[0.9695], [0.902], [0.8369]])
p2 = np.array([[0.9295], [0.902], [0.8769]])
Pi = np.array([[0.9, 0.1], [0.1, 0.9]])

# Put penalty on different issuance across maturities
c2 = 0.5

A1, B1, C1, R1, Q1, W1 = LQ_markov_mapping_restruct(A22, C_2, Ug, H, p1, c2)
A2, B2, C2, R2, Q2, W2 = LQ_markov_mapping_restruct(A22, C_2, Ug, H, p2, c2)

# Small penalties on debt required to implement no-Ponzi scheme
R1[0, 0] = R1[0, 0] + 1e-9
R1[1, 1] = R1[1, 1] + 1e-9
R1[2, 2] = R1[2, 2] + 1e-9
R2[0, 0] = R2[0, 0] + 1e-9
R2[1, 1] = R2[1, 1] + 1e-9
R2[2, 2] = R2[2, 2] + 1e-9

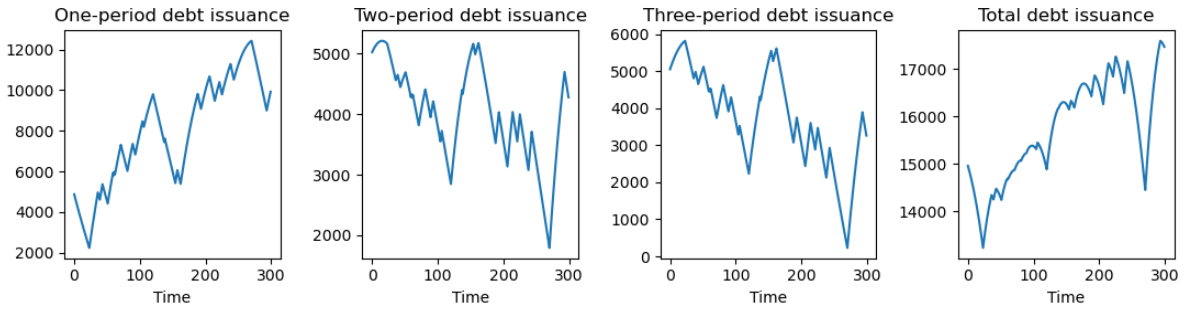
# Construct lists of matrices
As = [A1, A2]
Bs = [B1, B2]
Cs = [C1, C2]
Rs = [R1, R2]
Qs = [Q1, Q2]
Ws = [W1, W2]

# Construct and solve the model using the LQMarkov class
lqm3 = qe.LQMarkov(Π, Qs, Rs, As, Bs, Cs=Cs, Ns=Ws, beta=β)
lqm3.stationary_values()

x0 = np.array([[5000, 5000, 5000, 1, 10]])
x, u, w, t = lqm3.compute_sequence(x0, ts_length=300)
```

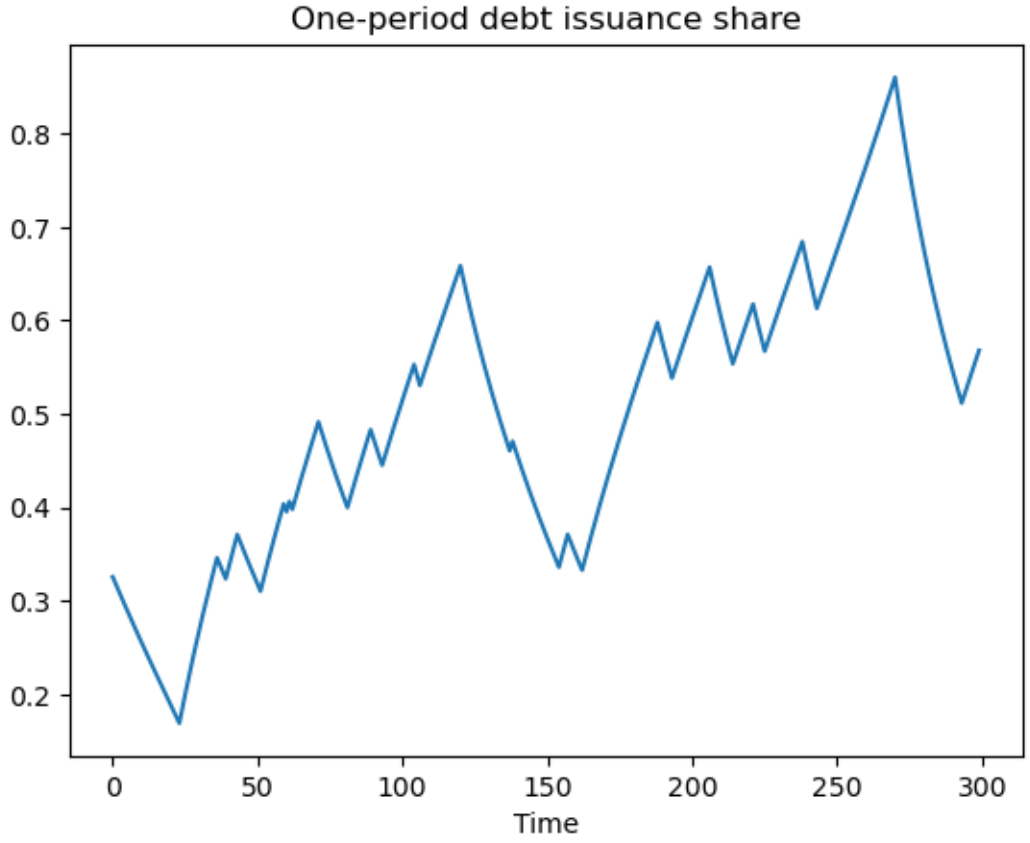
```
# Plots of different maturities debt issuance

fig, (ax1, ax2, ax3, ax4) = plt.subplots(1, 4, figsize=(11, 3))
ax1.plot(u[0, :])
ax1.set_title('One-period debt issuance')
ax1.set_xlabel('Time')
ax2.plot(u[1, :])
ax2.set_title('Two-period debt issuance')
ax2.set_xlabel('Time')
ax3.plot(u[2, :])
ax3.set_title('Three-period debt issuance')
ax3.set_xlabel('Time')
ax4.plot(u[0, :] + u[1, :] + u[2, :])
ax4.set_title('Total debt issuance')
ax4.set_xlabel('Time')
plt.tight_layout()
plt.show()
```




```
# Plot share of debt issuance that is short-term

fig, ax = plt.subplots()
ax.plot((u[0, :] / (u[0, :] + u[1, :] + u[2, :])))
ax.set_title('One-period debt issuance share')
ax.set_xlabel('Time')
plt.show()
```



CHAPTER

ELEVEN

HOW TO PAY FOR A WAR: PART 3

11.1 Overview

This lecture presents another application of Markov jump linear quadratic dynamic programming and constitutes a *sequel to an earlier lecture*.

We again use a method introduced in lecture *Markov Jump LQ dynamic programming* to implement some ideas of [Barro, 1999] and [Barro and McCleary, 2003]) that extend the classic [Barro, 1979] model of tax smoothing.

[Barro, 1979] is about a government that borrows and lends in order to help it minimize an intertemporal measure of distortions caused by taxes.

Technically, [Barro, 1979] looks a lot like a consumption-smoothing model.

Our generalization will also look like a souped-up consumption-smoothing model.

In this lecture, we describe a tax-smoothing problem of a government that faces **roll-over risk**.

In addition to what's in Anaconda, this lecture deploys the quantecon library:

```
!pip install --upgrade quantecon
```

Let's start with some standard imports:

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

11.2 Roll-Over Risk

Let T_t denote tax collections, β a discount factor, $b_{t,t+1}$ time $t+1$ goods that the government promises to pay at t, G_t government purchases, p^t_{t+1} the number of time t goods received per time $t+1$ goods promised.

The stochastic process of government expenditures is exogenous.

The government's problem is to choose a plan for borrowing and tax collections $\{b_{t+1}, T_t\}_{t=0}^{\infty}$ to minimize

$$E_0 \sum_{t=0}^{\infty} \beta^t T_t^2$$

subject to the constraints

$$T_t + p^t_{t+1} b_{t,t+1} = G_t + b_{t-1,t}$$

$$G_t = U_{g,t} z_t$$

$$z_{t+1} = A_{22,t} z_t + C_{2,t} w_{t+1}$$

where $w_{t+1} \sim N(0, I)$.

Let

- $T_t, b_{t,t+1}$ be *controls* chosen at t
- $b_{t-1,t}$ be an endogenous state variable inherited from the past at time t
- p^t_{t+1} be an exogenous price at time t.

This is the same set-up as used *in this lecture*.

We will consider a situation in which the government faces "roll-over risk".

Specifically, we shut down the government's ability to borrow in one of the Markov states.

11.3 A Dead End

A first thought for how to implement this might be to allow p^t_{t+1} to vary over time with:

$$p^t_{t+1} = \beta$$

in Markov state 1 and

$$p^t_{t+1} = 0$$

in Markov state 2.

Consequently, in the second Markov state, the government is unable to borrow, and the budget constraint becomes $T_t = G_t + b_{t-1,t}$.

However, if this is the only adjustment we make in our linear-quadratic model, the government will not set $b_{t,t+1} = 0$, which is the outcome we want to express *roll-over* risk in period t.

Instead, the government would have an incentive to set $b_{t,t+1}$ to a large negative number in state 2 – it would accumulate large amounts of *assets* to bring into period $t+1$ because that is cheap

- Riccati equations will tell us this

Thus, we must represent "roll-over risk" some other way.

11.4 Better Representation of Roll-Over Risk

To force the government to set $b_{t,t+1} = 0$, we can instead extend the model to have four Markov states:

1. Good today, good yesterday
2. Good today, bad yesterday
3. Bad today, good yesterday
4. Bad today, bad yesterday

where good is a state in which effectively the government can issue debt and bad is a state in which effectively the government can't issue debt.

We'll explain what *effectively* means shortly.

We now set

$$p_{t+1}^t = \beta$$

in all states.

In addition – and this is important because it defines what we mean by *effectively* – we put a large penalty on the $b_{t-1,t}$ element of the state vector in states 2 and 4.

This will prevent the government from wishing to issue any debt in states 3 or 4 because it would experience a large penalty from doing so in the next period.

The transition matrix for this formulation is:

$$\Pi = \begin{bmatrix} 0.95 & 0 & 0.05 & 0 \\ 0.95 & 0 & 0.05 & 0 \\ 0 & 0.9 & 0 & 0.1 \\ 0 & 0.9 & 0 & 0.1 \end{bmatrix}$$

This transition matrix ensures that the Markov state cannot move, for example, from state 3 to state 1.

Because state 3 is "bad today", the next period cannot have "good yesterday".

```
# Model parameters
β, Gbar, ρ, σ = 0.95, 5, 0.8, 1

# Basic model matrices
A22 = np.array([[1, 0], [Gbar, ρ], ])
C2 = np.array([[0], [σ]])
Ug = np.array([[0, 1]])

# LQ framework matrices
A_t = np.zeros((1, 3))
A_b = np.hstack((np.zeros((2, 1)), A22))
A = np.vstack((A_t, A_b))

B = np.zeros((3, 1))
B[0, 0] = 1

C = np.vstack((np.zeros((1, 1)), C2))

Sg = np.hstack((np.zeros((1, 1)), Ug))
S1 = np.zeros((1, 3))
S1[0, 0] = 1
S = S1 + Sg

R = S.T @ S

# Large penalty on debt in R2 to prevent borrowing in a bad state
R1 = np.copy(R)
R2 = np.copy(R)
R1[0, 0] = R[0, 0] + 1e-9
R2[0, 0] = R[0, 0] + 1e12

M = np.array([[-β]])
```

(continues on next page)

(continued from previous page)

```
Q = M.T @ M
W = M.T @ S

Π = np.array([[0.95,   0, 0.05,   0],
              [0.95,   0, 0.05,   0],
              [0,    0.9,    0, 0.1],
              [0,    0.9,    0, 0.1]])

# Construct lists of matrices that correspond to each state
As = [A, A, A, A]
Bs = [B, B, B, B]
Cs = [C, C, C, C]
Rs = [R1, R2, R1, R2]
Qs = [Q, Q, Q, Q]
Ws = [W, W, W, W]

lqm = qe.LQMarkov(Π, Qs, Rs, As, Bs, Cs=Cs, Ns=Ws, beta=β)
lqm.stationary_values();
```

Using the same process for G_t as in *this lecture*, we shall simulate our model with roll-over risk.

When $p_{t+1}^t = \beta$ government debt fluctuates around zero.

The spikes in the tax collection series indicate periods when the government is unable to access financial markets:

- positive spikes occur when debt is positive and the government must urgently raise tax revenues now

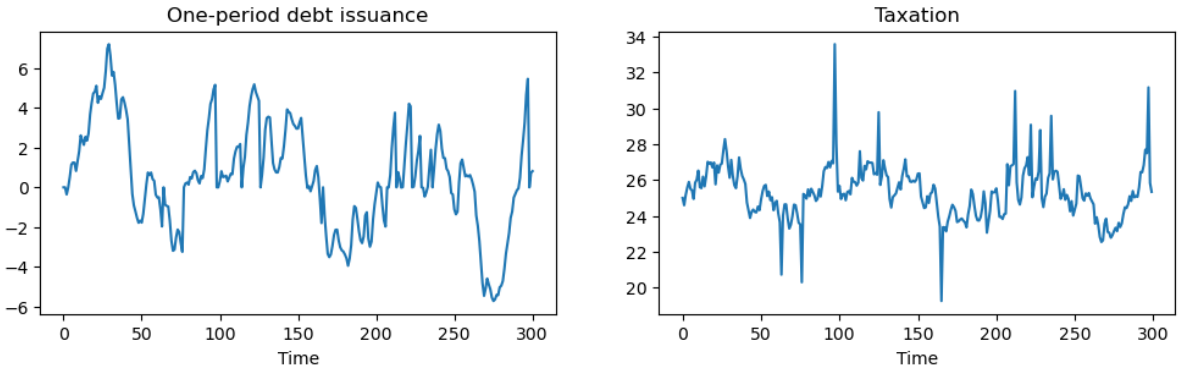
Negative spikes occur when the government has positive asset holdings.

An inability to use financial markets in the next period means that the government uses those assets to lower taxation today.

```
x0 = np.array([[0, 1, 25]])
T = 300
x, u, w, state = lqm.compute_sequence(x0, ts_length=T)

# Calculate taxation each period from the budget constraint and the Markov state
tax = np.zeros([T, 1])
for i in range(T):
    tax[i, :] = S @ x[:, i] + M @ u[:, i]

# Plot of debt issuance and taxation
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 3))
ax1.plot(x[0, :])
ax1.set_title('One-period debt issuance')
ax1.set_xlabel('Time')
ax2.plot(tax)
ax2.set_title('Taxation')
ax2.set_xlabel('Time')
plt.show()
```



We can adjust parameters so that, rather than debt fluctuating around zero, the government is a debtor in every period that it can borrow.

To accomplish this, we simply raise p^t_{t+1} to $\beta + 0.02 = 0.97$.

```
M = np.array([[-β - 0.02]])

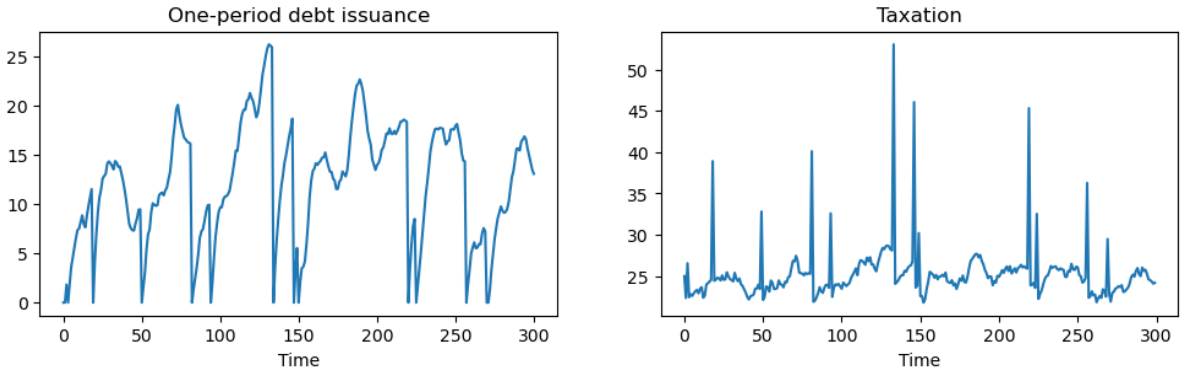
Q = M.T @ M
W = M.T @ S

# Construct lists of matrices
As = [A, A, A, A]
Bs = [B, B, B, B]
Cs = [C, C, C, C]
Rs = [R1, R2, R1, R2]
Qs = [Q, Q, Q, Q]
Ws = [W, W, W, W]

lqm2 = qe.LQMarkov(Π, Qs, Rs, As, Bs, Cs=Cs, Ns=Ws, beta=β)
x, u, w, state = lqm2.compute_sequence(x0, ts_length=T)

# Calculate taxation each period from the budget constraint and the
# Markov state
tax = np.zeros([T, 1])
for i in range(T):
    tax[i, :] = S @ x[:, i] + M @ u[:, i]

# Plot of debt issuance and taxation
fig, (ax1, ax2) =  plt.subplots(1, 2, figsize=(12, 3))
ax1.plot(x[0, :])
ax1.set_title('One-period debt issuance')
ax1.set_xlabel('Time')
ax2.plot(tax)
ax2.set_title('Taxation')
ax2.set_xlabel('Time')
plt.show()
```



With a lower interest rate, the government has an incentive to increase debt over time.

However, with "roll-over risk", debt is recurrently reset to zero and tax collections spike up.

In this model, high costs of a "sudden stop" make the government wary about letting its debt get too high.

CHAPTER

TWELVE

OPTIMAL TAXATION IN AN LQ ECONOMY

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install --upgrade quantecon
```

12.1 Overview

In this lecture, we study optimal fiscal policy in a linear quadratic setting.

We modify a model of Robert Lucas and Nancy Stokey [Lucas and Stokey, 1983] so that convenient formulas for solving linear-quadratic models can be applied.

The economy consists of a representative household and a benevolent government.

The government finances an exogenous stream of government purchases with state-contingent loans and a linear tax on labor income.

A linear tax is sometimes called a flat-rate tax.

The household maximizes utility by choosing paths for consumption and labor, taking prices and the government's tax rate and borrowing plans as given.

Maximum attainable utility for the household depends on the government's tax and borrowing plans.

The *Ramsey problem* [Ramsey, 1927] is to choose tax and borrowing plans that maximize the household's welfare, taking the household's optimizing behavior as given.

There is a large number of competitive equilibria indexed by different government fiscal policies.

The Ramsey planner chooses the best competitive equilibrium.

We want to study the dynamics of tax rates, tax revenues, government debt under a Ramsey plan.

Because the Lucas and Stokey model features state-contingent government debt, the government debt dynamics differ substantially from those in a model of Robert Barro [Barro, 1979].

The treatment given here closely follows this manuscript, prepared by Thomas J. Sargent and Francois R. Velde.

We cover only the key features of the problem in this lecture, leaving you to refer to that source for additional results and intuition.

We'll need the following imports:

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

(continues on next page)

(continued from previous page)

```
from numpy import sqrt, eye, zeros, cumsum
from numpy.random import randn
import scipy.linalg
from collections import namedtuple
from quantecon import nullspace, mc_sample_path, var_quadratic_sum
```

12.1.1 Model Features

- Linear quadratic (LQ) model
- Representative household
- Stochastic dynamic programming over an infinite horizon
- Distortionary taxation

12.2 The Ramsey Problem

We begin by outlining the key assumptions regarding technology, households and the government sector.

12.2.1 Technology

Labor can be converted one-for-one into a single, non-storable consumption good.

In the usual spirit of the LQ model, the amount of labor supplied in each period is unrestricted.

This is unrealistic, but helpful when it comes to solving the model.

Realistic labor supply can be induced by suitable parameter values.

12.2.2 Households

Consider a representative household who chooses a path $\{\ell_t, c_t\}$ for labor and consumption to maximize

$$-\mathbb{E}\frac{1}{2}\sum_{t=0}^{\infty}\beta^t\left[(c_t - b_t)^2 + \ell_t^2\right] \tag{12.1}$$

subject to the budget constraint

$$\mathbb{E}\sum_{t=0}^{\infty}\beta^t p_t^0\left[d_t + (1-\tau_t)\ell_t + s_t - c_t\right] = 0 \tag{12.2}$$

Here

- β is a discount factor in $(0, 1)$.
- p_t^0 is a scaled Arrow-Debreu price at time 0 of history contingent goods at time $t + j$.
- b_t is a stochastic preference parameter.
- d_t is an endowment process.
- τ_t is a flat tax rate on labor income.

- s_t is a promised time-t coupon payment on debt issued by the government.

The scaled Arrow-Debreu price p_t^0 is related to the unscaled Arrow-Debreu price as follows.

If we let $\pi_t^0(x^t)$ denote the probability (density) of a history $x^t = [x_t, x_{t-1}, \dots, x_0]$ of the state x^t, then the Arrow-Debreu time 0 price of a claim on one unit of consumption at date t, history x^t would be

$$\frac{\beta^t p_t^0}{\pi_t^0(x^t)}$$

Thus, our scaled Arrow-Debreu price is the ordinary Arrow-Debreu price multiplied by the discount factor β^t and divided by an appropriate probability.

The budget constraint (12.2) requires that the present value of consumption be restricted to equal the present value of endowments, labor income and coupon payments on bond holdings.

12.2.3 Government

The government imposes a linear tax on labor income, fully committing to a stochastic path of tax rates at time zero.

The government also issues state-contingent debt.

Given government tax and borrowing plans, we can construct a competitive equilibrium with distorting government taxes.

Among all such competitive equilibria, the Ramsey plan is the one that maximizes the welfare of the representative consumer.

12.2.4 Exogenous Variables

Endowments, government expenditure, the preference shock process b_t, and promised coupon payments on initial government debt s_t are all exogenous, and given by

- $d_t = S_d x_t$
- $g_t = S_g x_t$
- $b_t = S_b x_t$
- $s_t = S_s x_t$

The matrices S_d, S_g, S_b, S_s are primitives and $\{x_t\}$ is an exogenous stochastic process taking values in $\mathbb{R}^k$.

We consider two specifications for $\{x_t\}$.

1. Discrete case: $\{x_t\}$ is a discrete state Markov chain with transition matrix P.
2. VAR case: $\{x_t\}$ obeys $x_{t+1} = Ax_t + Cw_{t+1}$ where $\{w_t\}$ is independent zero-mean Gaussian with identify covariance matrix.

12.2.5 Feasibility

The period-by-period feasibility restriction for this economy is

$$c_t + g_t = d_t + \ell_t \tag{12.3}$$

A labor-consumption process $\{\ell_t, c_t\}$ is called *feasible* if (12.3) holds for all t.

12.2.6 Government Budget Constraint

Where p_t^0 is again a scaled Arrow-Debreu price, the time zero government budget constraint is

$$\mathbb{E}\sum_{t=0}^{\infty}\beta^t p_t^0(s_t + g_t - \tau_t\ell_t) = 0 \tag{12.4}$$

12.2.7 Equilibrium

An *equilibrium* is a feasible allocation $\{\ell_t, c_t\}$, a sequence of prices $\{p_t^0\}$, and a tax system $\{\tau_t\}$ such that

1. The allocation $\{\ell_t, c_t\}$ is optimal for the household given $\{p_t^0\}$ and $\{\tau_t\}$.
2. The government's budget constraint (12.4) is satisfied.

The *Ramsey problem* is to choose the equilibrium $\{\ell_t, c_t, \tau_t, p_t^0\}$ that maximizes the household's welfare.

If $\{\ell_t, c_t, \tau_t, p_t^0\}$ solves the Ramsey problem, then $\{\tau_t\}$ is called the *Ramsey plan*.

The solution procedure we adopt is

1. Use the first-order conditions from the household problem to pin down prices and allocations given $\{\tau_t\}$.
2. Use these expressions to rewrite the government budget constraint (12.4) in terms of exogenous variables and allocations.
3. Maximize the household's objective function (12.1) subject to the constraint constructed in step 2 and the feasibility constraint (12.3).

The solution to this maximization problem pins down all quantities of interest.

12.2.8 Solution

Step one is to obtain the first-conditions for the household's problem, taking taxes and prices as given.

Letting μ be the Lagrange multiplier on (12.2), the first-order conditions are $p_t^0 = (c_t - b_t)/\mu$ and $\ell_t = (c_t - b_t)(1-\tau_t)$.

Rearranging and normalizing at $\mu = b_0 - c_0$, we can write these conditions as

$$p_t^0 = \frac{b_t - c_t}{b_0 - c_0} \quad \text{and} \quad \tau_t = 1 - \frac{\ell_t}{b_t - c_t} \tag{12.5}$$

Substituting (12.5) into the government's budget constraint (12.4) yields

$$\mathbb{E}\sum_{t=0}^{\infty}\beta^t\left[(b_t - c_t)(s_t + g_t - \ell_t) + \ell_t^2\right] = 0 \tag{12.6}$$

The Ramsey problem now amounts to maximizing (12.1) subject to (12.6) and (12.3).

The associated Lagrangian is

$$\mathcal{L} = \mathbb{E}\sum_{t=0}^{\infty}\beta^t\left\{-\frac{1}{2}\left[(c_t - b_t)^2 + \ell_t^2\right] + \lambda\left[(b_t - c_t)(\ell_t - s_t - g_t) - \ell_t^2\right] + \mu_t[d_t + \ell_t - c_t - g_t]\right\} \tag{12.7}$$

The first-order conditions associated with c_t and ℓ_t are

$$-(c_t - b_t) + \lambda[-\ell_t + (g_t + s_t)] = \mu_t$$

and

$$\ell_t - \lambda[(b_t - c_t) - 2\ell_t] = \mu_t$$

Combining these last two equalities with (12.3) and working through the algebra, one can show that

$$\ell_t = \bar{\ell}_t - \nu m_t \quad \text{and} \quad c_t = \bar{c}_t - \nu m_t \tag{12.8}$$

where

- $\nu := \lambda/(1 + 2\lambda)$
- $\bar{\ell}_t := (b_t - d_t + g_t)/2$
- $\bar{c}_t := (b_t + d_t - g_t)/2$
- $m_t := (b_t - d_t - s_t)/2$

Apart from ν, all of these quantities are expressed in terms of exogenous variables.

To solve for ν, we can use the government's budget constraint again.

The term inside the brackets in (12.6) is $(b_t - c_t)(s_t + g_t) - (b_t - c_t)\ell_t + \ell_t^2$.

Using (12.8), the definitions above and the fact that $\bar{\ell} = b - \bar{c}$, this term can be rewritten as

$$(b_t - \bar{c}_t)(g_t + s_t) + 2m_t^2(\nu^2 - \nu)$$

Reinserting into (12.6), we get

$$\mathbb{E}\left\{\sum_{t=0}^{\infty} \beta^t (b_t - \bar{c}_t)(g_t + s_t)\right\} + (\nu^2 - \nu)\mathbb{E}\left\{\sum_{t=0}^{\infty} \beta^t 2m_t^2\right\} = 0 \tag{12.9}$$

Although it might not be clear yet, we are nearly there because:

- The two expectations terms in (12.9) can be solved for in terms of model primitives.
- This in turn allows us to solve for the Lagrange multiplier ν.
- With ν in hand, we can go back and solve for the allocations via (12.8).
- Once we have the allocations, prices and the tax system can be derived from (12.5).

12.2.9 Computing the Quadratic Term

Let's consider how to obtain the term ν in (12.9).

If we can compute the two expected geometric sums

$$b_0 := \mathbb{E}\left\{\sum_{t=0}^{\infty} \beta^t (b_t - \bar{c}_t)(g_t + s_t)\right\} \quad \text{and} \quad a_0 := \mathbb{E}\left\{\sum_{t=0}^{\infty} \beta^t 2m_t^2\right\} \tag{12.10}$$

then the problem reduces to solving

$$b_0 + a_0(\nu^2 - \nu) = 0$$

for ν.

Provided that $4b_0 < a_0$, there is a unique solution $\nu \in (0, 1/2)$, and a unique corresponding $\lambda > 0$.

Let's work out how to compute mathematical expectations in (12.10).

For the first one, the random variable $(b_t - \bar{c}_t)(g_t + s_t)$ inside the summation can be expressed as

$$\frac{1}{2} x_t'(S_b - S_d + S_g)'(S_g + S_s)x_t$$

For the second expectation in (12.10), the random variable $2m_t^2$ can be written as

$$\frac{1}{2} x_t'(S_b - S_d - S_s)'(S_b - S_d - S_s)x_t$$

It follows that both objects of interest are special cases of the expression

$$q(x_0) = \mathbb{E} \sum_{t=0}^{\infty} \beta^t x_t' H x_t \tag{12.11}$$

where H is a matrix conformable to x_t and x_t' is the transpose of column vector x_t.

Suppose first that $\{x_t\}$ is the Gaussian VAR described *above*.

In this case, the formula for computing $q(x_0)$ is known to be $q(x_0) = x_0' Q x_0 + v$, where

- Q is the solution to $Q = H + \beta A' Q A$, and
- $v = \text{trace}\,(C' Q C)\beta/(1-\beta)$

The first equation is known as a discrete Lyapunov equation and can be solved using this function.

12.2.10 Finite State Markov Case

Next, suppose that $\{x_t\}$ is the discrete Markov process described *above*.

Suppose further that each x_t takes values in the state space $\{x^1, \ldots, x^N\} \subset \mathbb{R}^k$.

Let $h \colon \mathbb{R}^k \to \mathbb{R}$ be a given function, and suppose that we wish to evaluate

$$q(x_0) = \mathbb{E} \sum_{t=0}^{\infty} \beta^t h(x_t) \quad \text{given} \quad x_0 = x^j$$

For example, in the discussion above, $h(x_t) = x_t' H x_t$.

It is legitimate to pass the expectation through the sum, leading to

$$q(x_0) = \sum_{t=0}^{\infty} \beta^t (P^t h)[j] \tag{12.12}$$

Here

- P^t is the t-th power of the transition matrix P.
- h is, with some abuse of notation, the vector $(h(x^1), \ldots, h(x^N))$.
- $(P^t h)[j]$ indicates the j-th element of $P^t h$.

It can be shown that (12.12) is in fact equal to the j-th element of the vector $(I - \beta P)^{-1} h$.

This last fact is applied in the calculations below.

12.2.11 Other Variables

We are interested in tracking several other variables besides the ones described above.

To prepare the way for this, we define

$$p_{t+j}^t = \frac{b_{t+j} - c_{t+j}}{b_t - c_t}$$

as the scaled Arrow-Debreu time t price of a history contingent claim on one unit of consumption at time $t+j$.

These are prices that would prevail at time t if markets were reopened at time t.

These prices are constituents of the present value of government obligations outstanding at time t, which can be expressed as

$$B_t := \mathbb{E}_t \sum_{j=0}^{\infty} \beta^j p_{t+j}^t (\tau_{t+j} \ell_{t+j} - g_{t+j}) \tag{12.13}$$

Using our expression for prices and the Ramsey plan, we can also write B_t as

$$B_t = \mathbb{E}_t \sum_{j=0}^{\infty} \beta^j \frac{(b_{t+j} - c_{t+j})(\ell_{t+j} - g_{t+j}) - \ell_{t+j}^2}{b_t - c_t}$$

This version is more convenient for computation.

Using the equation

$$p_{t+j}^t = p_{t+1}^t p_{t+j}^{t+1}$$

it is possible to verify that (12.13) implies that

$$B_t = (\tau_t \ell_t - g_t) + E_t \sum_{j=1}^{\infty} p_{t+j}^t (\tau_{t+j} \ell_{t+j} - g_{t+j})$$

and

$$B_t = (\tau_t \ell_t - g_t) + \beta E_t p_{t+1}^t B_{t+1} \tag{12.14}$$

Define

$$R_t^{-1} := \mathbb{E}_t \beta^j p_{t+1}^t \tag{12.15}$$

R_t is the gross 1-period risk-free rate for loans between t and $t+1$.

12.2.12 A Martingale

We now want to study the following two objects, namely,

$$\pi_{t+1} := B_{t+1} - R_t [B_t - (\tau_t \ell_t - g_t)]$$

and the cumulation of π_t

$$\Pi_t := \sum_{s=0}^{t} \pi_t$$

The term π_{t+1} is the difference between two quantities:

- B_{t+1}, the value of government debt at the start of period $t+1$.
- $R_t[B_t + g_t - \tau_t]$, which is what the government would have owed at the beginning of period $t+1$ if it had simply borrowed at the one-period risk-free rate rather than selling state-contingent securities.

Thus, π_{t+1} is the excess payout on the actual portfolio of state-contingent government debt relative to an alternative portfolio sufficient to finance $B_t + g_t - \tau_t \ell_t$ and consisting entirely of risk-free one-period bonds.

Use expressions (12.14) and (12.15) to obtain

$$\pi_{t+1} = B_{t+1} - \frac{1}{\beta E_t p_{t+1}^t} \left[\beta E_t p_{t+1}^t B_{t+1}\right]$$

or

$$\pi_{t+1} = B_{t+1} - \tilde{E}_t B_{t+1} \tag{12.16}$$

where $\tilde{E}_t$ is the conditional mathematical expectation taken with respect to a one-step transition density that has been formed by multiplying the original transition density with the likelihood ratio

$$m_{t+1}^t = \frac{p_{t+1}^t}{E_t p_{t+1}^t}$$

It follows from equation (12.16) that

$$\tilde{E}_t \pi_{t+1} = \tilde{E}_t B_{t+1} - \tilde{E}_t B_{t+1} = 0$$

which asserts that $\{\pi_{t+1}\}$ is a martingale difference sequence under the distorted probability measure, and that $\{\Pi_t\}$ is a martingale under the distorted probability measure.

In the tax-smoothing model of Robert Barro [Barro, 1979], government debt is a random walk.

In the current model, government debt $\{B_t\}$ is not a random walk, but the `excess payoff` $\{\Pi_t\}$ on it is.

12.3 Implementation

The following code provides functions for

1. Solving for the Ramsey plan given a specification of the economy.
2. Simulating the dynamics of the major variables.

Description and clarifications are given below

```
# Set up a namedtuple to store data on the model economy
Economy = namedtuple('economy',
                     ('β',          # Discount factor
                     'Sg',          # Govt spending selector matrix
                     'Sd',          # Exogenous endowment selector matrix
                     'Sb',          # Utility parameter selector matrix
                     'Ss',          # Coupon payments selector matrix
                     'discrete',    # Discrete or continuous -- boolean
                     'proc'))       # Stochastic process parameters

# Set up a namedtuple to store return values for compute_paths()
Path = namedtuple('path',
                  ('g',           # Govt spending
                  'd',            # Endowment
```

(continues on next page)

(continued from previous page)

```
                'b',            # Utility shift parameter
                's',            # Coupon payment on existing debt
                'c',            # Consumption
                'l',            # Labor
                'p',            # Price
                'τ',            # Tax rate
                'rvn',          # Revenue
                'B',            # Govt debt
                'R',            # Risk-free gross return
                'π',            # One-period risk-free interest rate
                'Π',            # Cumulative rate of return, adjusted
                'ξ'))           # Adjustment factor for Π


def compute_paths(T, econ):
    """
    Compute simulated time paths for exogenous and endogenous variables.

    Parameters
    ==========
    T: int
        Length of the simulation

    econ: a namedtuple of type 'Economy', containing
        β           - Discount factor
        Sg          - Govt spending selector matrix
        Sd          - Exogenous endowment selector matrix
        Sb          - Utility parameter selector matrix
        Ss          - Coupon payments selector matrix
        discrete    - Discrete exogenous process (True or False)
        proc        - Stochastic process parameters

    Returns
    ========
    path: a namedtuple of type 'Path', containing
        g             - Govt spending
        d             - Endowment
        b             - Utility shift parameter
        s             - Coupon payment on existing debt
        c             - Consumption
        l             - Labor
        p             - Price
        τ             - Tax rate
        rvn           - Revenue
        B             - Govt debt
        R             - Risk-free gross return
        π             - One-period risk-free interest rate
        Π             - Cumulative rate of return, adjusted
        ξ             - Adjustment factor for Π

        The corresponding values are flat numpy ndarrays.

    """

    # Simplify names
    β, Sg, Sd, Sb, Ss = econ.β, econ.Sg, econ.Sd, econ.Sb, econ.Ss
```

(continues on next page)

(continued from previous page)

```
    if econ.discrete:
        P, x_vals = econ.proc
    else:
        A, C = econ.proc

    # Simulate the exogenous process x
    if econ.discrete:
        state = mc_sample_path(P, init=0, sample_size=T)
        x = x_vals[:, state]
    else:
        # Generate an initial condition x0 satisfying x0 = A x0
        nx, nx = A.shape
        x0 = nullspace((eye(nx) - A))
        x0 = -x0 if (x0[nx-1] < 0) else x0
        x0 = x0 / x0[nx-1]

        # Generate a time series x of length T starting from x0
        nx, nw = C.shape
        x = zeros((nx, T))
        w = randn(nw, T)
        x[:, 0] = x0.T
        for t in range(1, T):
            x[:, t] = A @ x[:, t-1] + C @ w[:, t]

    # Compute exogenous variable sequences
    g, d, b, s = ((S @ x).flatten() for S in (Sg, Sd, Sb, Ss))

    # Solve for Lagrange multiplier in the govt budget constraint
    # In fact we solve for v = lambda / (1 + 2*lambda).  Here v is the
    # solution to a quadratic equation a(v**2 - v) + b = 0 where
    # a and b are expected discounted sums of quadratic forms of the state.
    Sm = Sb - Sd - Ss
    # Compute a and b
    if econ.discrete:
        ns = P.shape[0]
        F = scipy.linalg.inv(eye(ns) - β * P)
        a0 = 0.5 * (F @ (x_vals.T @ Sm.T)**2)[0]
        H = ((Sb - Sd + Sg) @ x_vals) * ((Sg - Ss) @ x_vals)
        b0 = 0.5 * (F @ H.T)[0]
        a0, b0 = float(a0[0]), float(b0[0])
    else:
        H = Sm.T @ Sm
        a0 = 0.5 * var_quadratic_sum(A, C, H, β, x0)
        H = (Sb - Sd + Sg).T @ (Sg + Ss)
        b0 = 0.5 * var_quadratic_sum(A, C, H, β, x0)

    # Test that v has a real solution before assigning
    warning_msg = """
    Hint: you probably set government spending too {}.  Elect a {}
    Congress and start over.
    """
    disc = a0**2 - 4 * a0 * b0
    if disc >= 0:
        v = 0.5 * (a0 - sqrt(disc)) / a0
    else:
```

(continues on next page)

(continued from previous page)

```
        print("There is no Ramsey equilibrium for these parameters.")
        print(warning_msg.format('high', 'Republican'))
        sys.exit(0)

    # Test that the Lagrange multiplier has the right sign
    if ν * (0.5 - ν) < 0:
        print("Negative multiplier on the government budget constraint.")
        print(warning_msg.format('low', 'Democratic'))
        sys.exit(0)

    # Solve for the allocation given ν and x
    Sc = 0.5 * (Sb + Sd - Sg - ν * Sm)
    Sl = 0.5 * (Sb - Sd + Sg - ν * Sm)
    c = (Sc @ x).flatten()
    l = (Sl @ x).flatten()
    p = ((Sb - Sc) @ x).flatten()  # Price without normalization
    τ = 1 - l / (b - c)
    rvn = l * τ

    # Compute remaining variables
    if econ.discrete:
        H = ((Sb - Sc) @ x_vals) * ((Sl - Sg) @ x_vals) - (Sl @ x_vals)**2
        temp = (F @ H.T).flatten()
        B = temp[state] / p
        H = (P[state, :] @ x_vals.T @ (Sb - Sc).T).flatten()
        R = p / (β * H)
        temp = ((P[state, :] @ x_vals.T @ (Sb - Sc).T)).flatten()
        ξ = p[1:] / temp[:T-1]
    else:
        H = Sl.T @ Sl - (Sb - Sc).T @ (Sl - Sg)
        L = np.empty(T)
        for t in range(T):
            L[t] = var_quadratic_sum(A, C, H, β, x[:, t])
        B = L / p
        Rinv = (β * ((Sb - Sc) @ A @ x)).flatten() / p
        R = 1 / Rinv
        AF1 = (Sb - Sc) @ x[:, 1:]
        AF2 = (Sb - Sc) @ A @ x[:, :T-1]
        ξ = AF1 / AF2
        ξ = ξ.flatten()

    π = B[1:] - R[:T-1] * B[:T-1] - rvn[:T-1] + g[:T-1]
    Π = cumsum(π * ξ)

    # Prepare return values
    path = Path(g=g, d=d, b=b, s=s, c=c, l=l, p=p,
                τ=τ, rvn=rvn, B=B, R=R, π=π, Π=Π, ξ=ξ)

    return path


def gen_fig_1(path):
    """
    The parameter is the path namedtuple returned by compute_paths().  See
    the docstring of that function for details.
    """
```

(continues on next page)

(continued from previous page)

```
    T = len(path.c)

    # Prepare axes
    num_rows, num_cols = 2, 2
    fig, axes = plt.subplots(num_rows, num_cols, figsize=(14, 10))
    plt.subplots_adjust(hspace=0.4)
    for i in range(num_rows):
        for j in range(num_cols):
            axes[i, j].grid()
            axes[i, j].set_xlabel('Time')
    bbox = (0., 1.02, 1., .102)
    legend_args = {'bbox_to_anchor': bbox, 'loc': 3, 'mode': 'expand'}
    p_args = {'lw': 2, 'alpha': 0.7}

    # Plot consumption, govt expenditure and revenue
    ax = axes[0, 0]
    ax.plot(path.rvn, label=r'$\tau_t \ell_t$', **p_args)
    ax.plot(path.g, label='$g_t$', **p_args)
    ax.plot(path.c, label='$c_t$', **p_args)
    ax.legend(ncol=3, **legend_args)

    # Plot govt expenditure and debt
    ax = axes[0, 1]
    ax.plot(list(range(1, T+1)), path.rvn, label=r'$\tau_t \ell_t$', **p_args)
    ax.plot(list(range(1, T+1)), path.g, label='$g_t$', **p_args)
    ax.plot(list(range(1, T)), path.B[1:T], label='$B_{t+1}$', **p_args)
    ax.legend(ncol=3, **legend_args)

    # Plot risk-free return
    ax = axes[1, 0]
    ax.plot(list(range(1, T+1)), path.R - 1, label='$R_t - 1$', **p_args)
    ax.legend(ncol=1, **legend_args)

    # Plot revenue, expenditure and risk free rate
    ax = axes[1, 1]
    ax.plot(list(range(1, T+1)), path.rvn, label=r'$\tau_t \ell_t$', **p_args)
    ax.plot(list(range(1, T+1)), path.g, label='$g_t$', **p_args)
    axes[1, 1].plot(list(range(1, T)), path.π, label=r'$\pi_{t+1}$', **p_args)
    ax.legend(ncol=3, **legend_args)

    plt.show()


def gen_fig_2(path):
    """
    The parameter is the path namedtuple returned by compute_paths(). See
    the docstring of that function for details.
    """

    T = len(path.c)

    # Prepare axes
    num_rows, num_cols = 2, 1
    fig, axes = plt.subplots(num_rows, num_cols, figsize=(10, 10))
    plt.subplots_adjust(hspace=0.5)
```

(continues on next page)

(continued from previous page)

```
    bbox = (0., 1.02, 1., .102)
    bbox = (0., 1.02, 1., .102)
    legend_args = {'bbox_to_anchor': bbox, 'loc': 3, 'mode': 'expand'}
    p_args = {'lw': 2, 'alpha': 0.7}

    # Plot adjustment factor
    ax = axes[0]
    ax.plot(list(range(2, T+1)), path.ξ, label=r'$\xi_t$', **p_args)
    ax.grid()
    ax.set_xlabel('Time')
    ax.legend(ncol=1, **legend_args)

    # Plot adjusted cumulative return
    ax = axes[1]
    ax.plot(list(range(2, T+1)), path.Π, label=r'$\Pi_t$', **p_args)
    ax.grid()
    ax.set_xlabel('Time')
    ax.legend(ncol=1, **legend_args)

    plt.show()
```

12.3.1 Comments on the Code

The function `var_quadratic_sum` imported from `quadsums` is for computing the value of (12.11) when the exogenous process $\{x_t\}$ is of the VAR type described *above*.

Below the definition of the function, you will see definitions of two `namedtuple` objects, `Economy` and `Path`.

The first is used to collect all the parameters and primitives of a given LQ economy, while the second collects output of the computations.

In Python, a `namedtuple` is a popular data type from the `collections` module of the standard library that replicates the functionality of a tuple, but also allows you to assign a name to each tuple element.

These elements can then be references via dotted attribute notation — see for example the use of `path` in the functions `gen_fig_1()` and `gen_fig_2()`.

The benefits of using `namedtuples`:

- Keeps content organized by meaning.
- Helps reduce the number of global variables.

Other than that, our code is long but relatively straightforward.

12.4 Examples

Let's look at two examples of usage.

12.4.1 The Continuous Case

Our first example adopts the VAR specification described *above*.

Regarding the primitives, we set

- $\beta = 1/1.05$
- $b_t = 2.135$ and $s_t = d_t = 0$ for all t

Government spending evolves according to

$$g_{t+1} - \mu_g = \rho(g_t - \mu_g) + C_g w_{g,t+1}$$

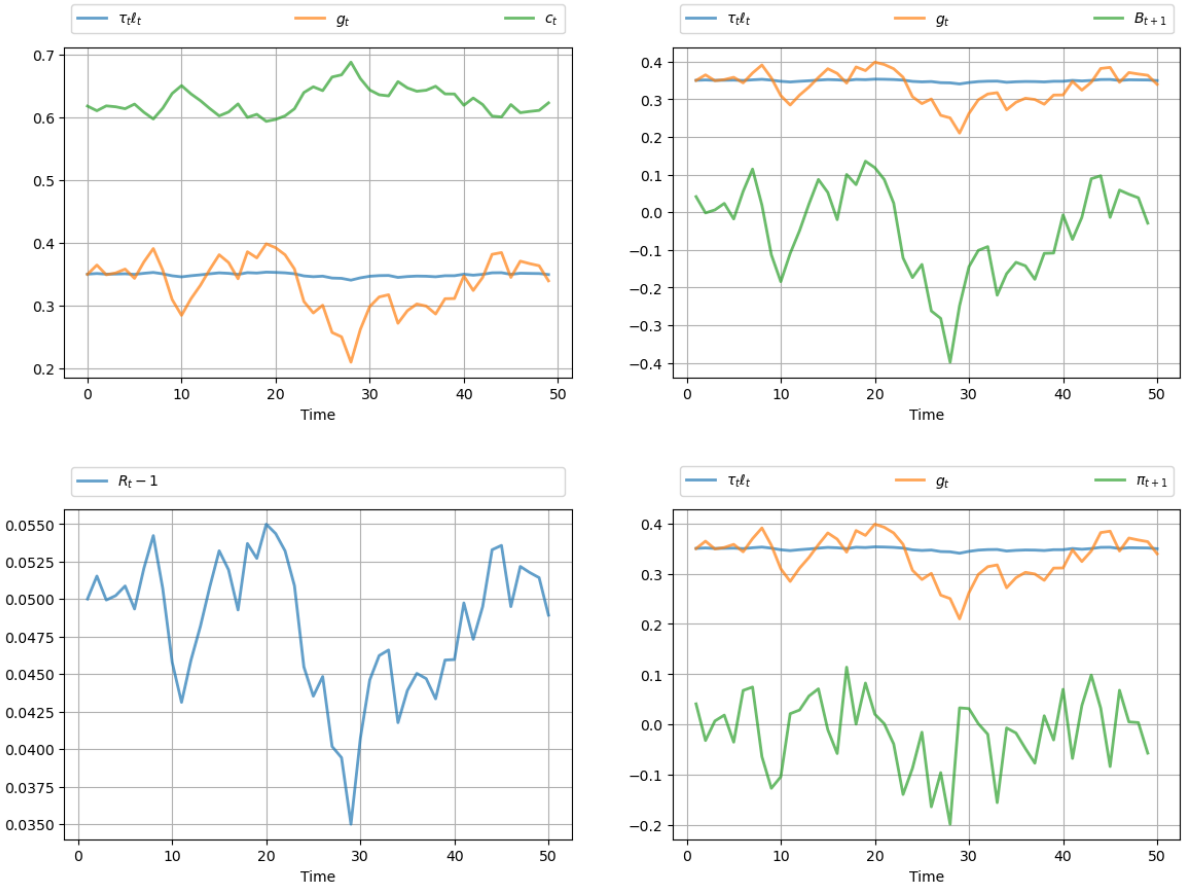
with $\rho = 0.7$, $\mu_g = 0.35$ and $C_g = \mu_g\sqrt{1-\rho^2}/10$.

Here's the code

```
# == Parameters == #
β = 1 / 1.05
ρ, mg = .7, .35
A = eye(2)
A[0, :] = ρ, mg * (1-ρ)
C = np.zeros((2, 1))
C[0, 0] = np.sqrt(1 - ρ**2) * mg / 10
Sg = np.array((1, 0)).reshape(1, 2)
Sd = np.array((0, 0)).reshape(1, 2)
Sb = np.array((0, 2.135)).reshape(1, 2)
Ss = np.array((0, 0)).reshape(1, 2)

economy = Economy(β=β, Sg=Sg, Sd=Sd, Sb=Sb, Ss=Ss,
                  discrete=False, proc=(A, C))

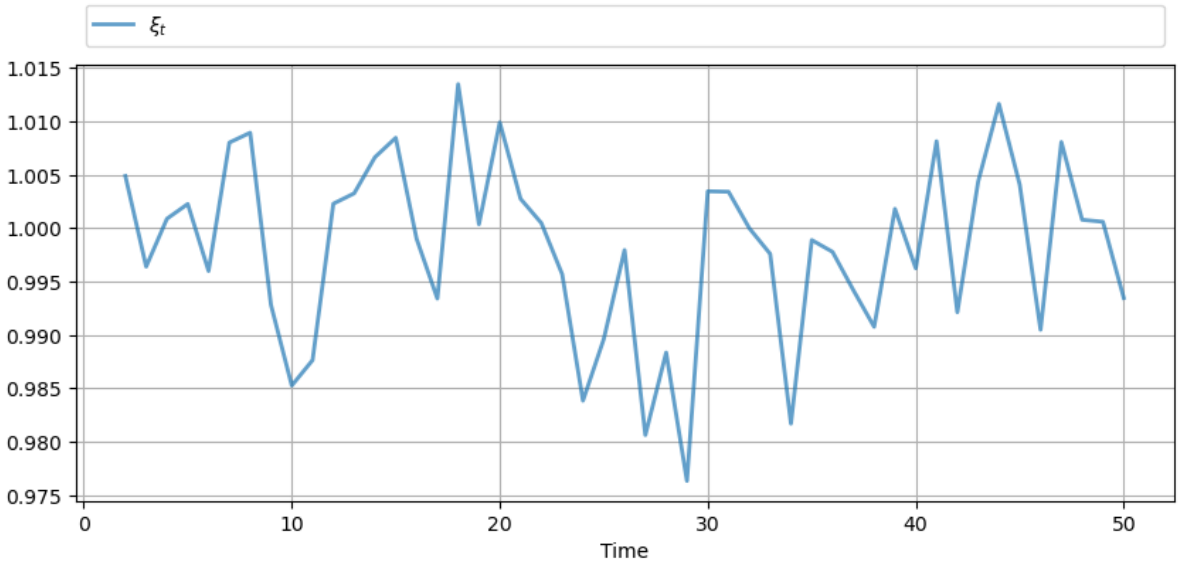
T = 50
path = compute_paths(T, economy)
gen_fig_1(path)
```

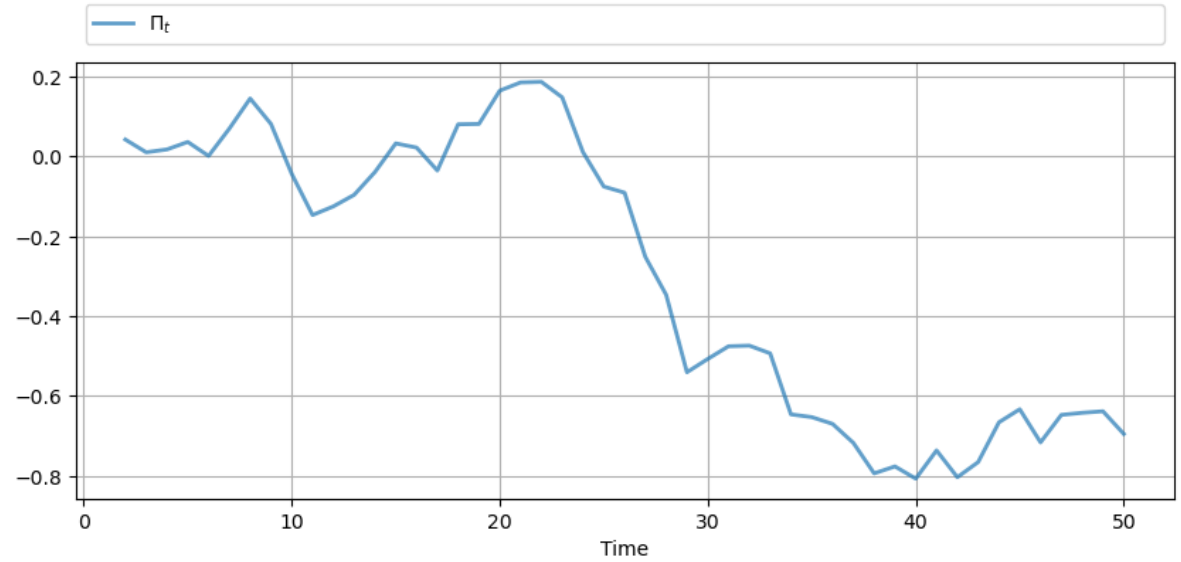


The legends on the figures indicate the variables being tracked.

Most obvious from the figure is tax smoothing in the sense that tax revenue is much less variable than government expenditure.

```
gen_fig_2(path)
```





See the original manuscript for comments and interpretation.

12.4.2 The Discrete Case

Our second example adopts a discrete Markov specification for the exogenous process

```
# == Parameters == #
β = 1 / 1.05
P = np.array([[0.8, 0.2, 0.0],
              [0.0, 0.5, 0.5],
              [0.0, 0.0, 1.0]])

# Possible states of the world
```

(continues on next page)

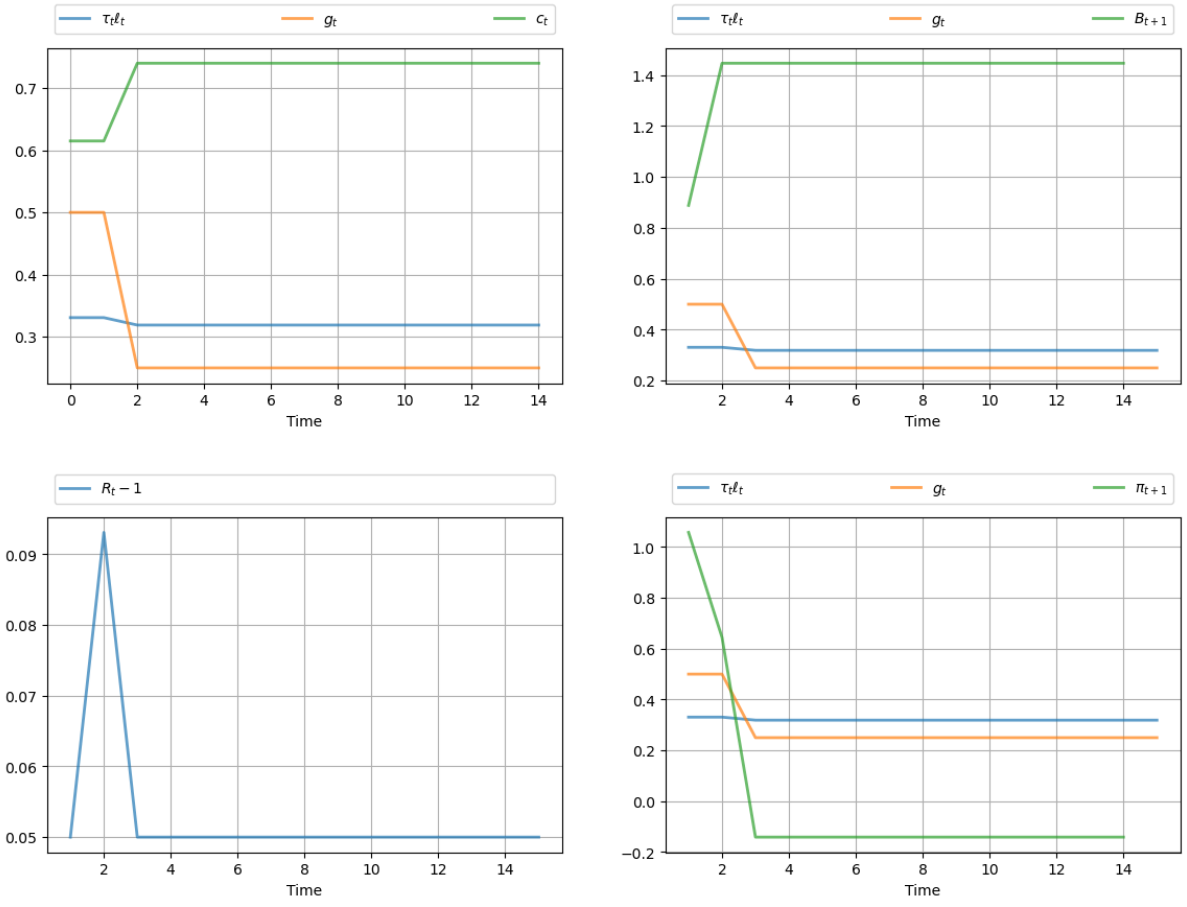
(continued from previous page)

```
# Each column is a state of the world. The rows are [g d b s 1]
x_vals = np.array([[0.5, 0.5, 0.25],
                   [0.0, 0.0,  0.0],
                   [2.2, 2.2,  2.2],
                   [0.0, 0.0,  0.0],
                   [1.0, 1.0,  1.0]])

Sg = np.array((1, 0, 0, 0, 0)).reshape(1, 5)
Sd = np.array((0, 1, 0, 0, 0)).reshape(1, 5)
Sb = np.array((0, 0, 1, 0, 0)).reshape(1, 5)
Ss = np.array((0, 0, 0, 1, 0)).reshape(1, 5)

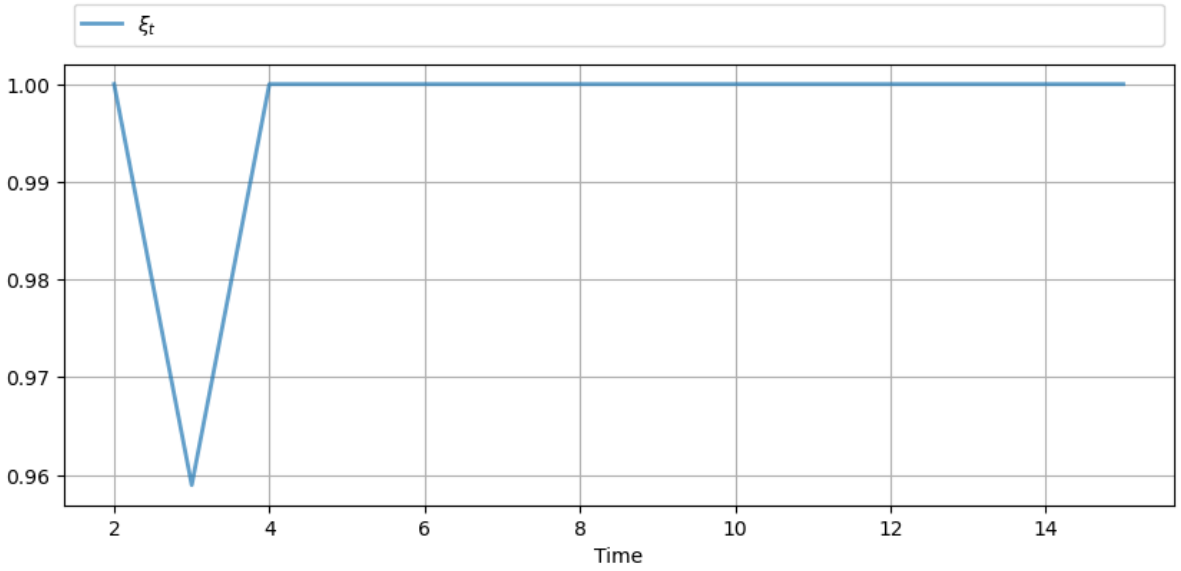
economy = Economy(β=β, Sg=Sg, Sd=Sd, Sb=Sb, Ss=Ss,
                  discrete=True, proc=(P, x_vals))

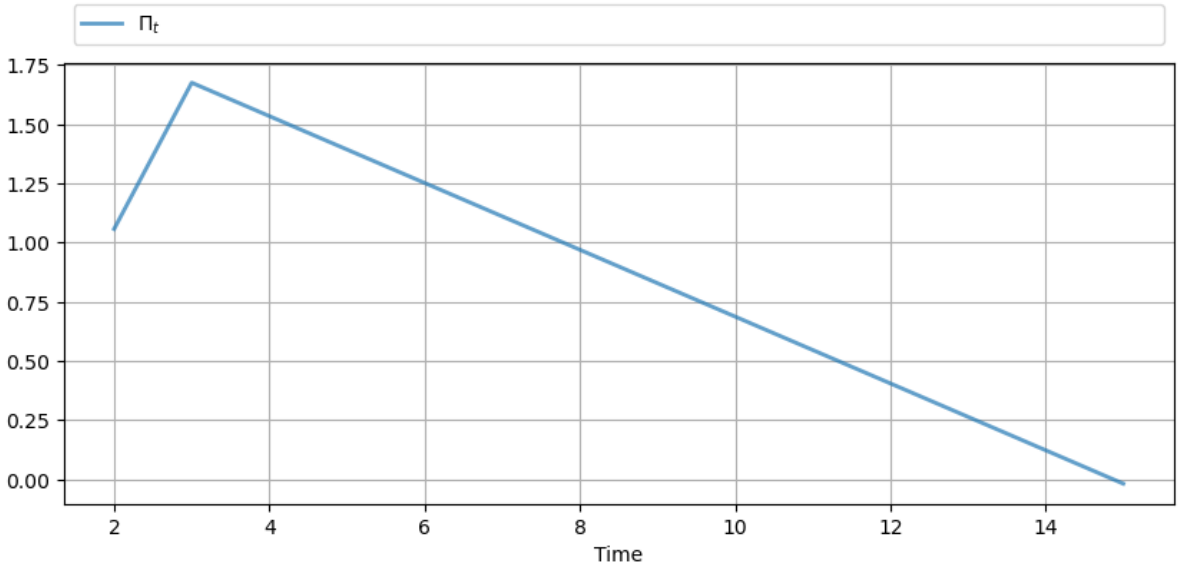
T = 15
path = compute_paths(T, economy)
gen_fig_1(path)
```



The call `gen_fig_2(path)` generates

```
gen_fig_2(path)
```





See the original manuscript for comments and interpretation.

12.5 Exercises

Exercise 12.5.1

Modify the VAR example *given above*, setting

$$g_{t+1} - \mu_g = \rho(g_{t-3} - \mu_g) + C_g w_{g,t+1}$$

with $\rho = 0.95$ and $C_g = 0.7\sqrt{1-\rho^2}$.

Produce the corresponding figures.

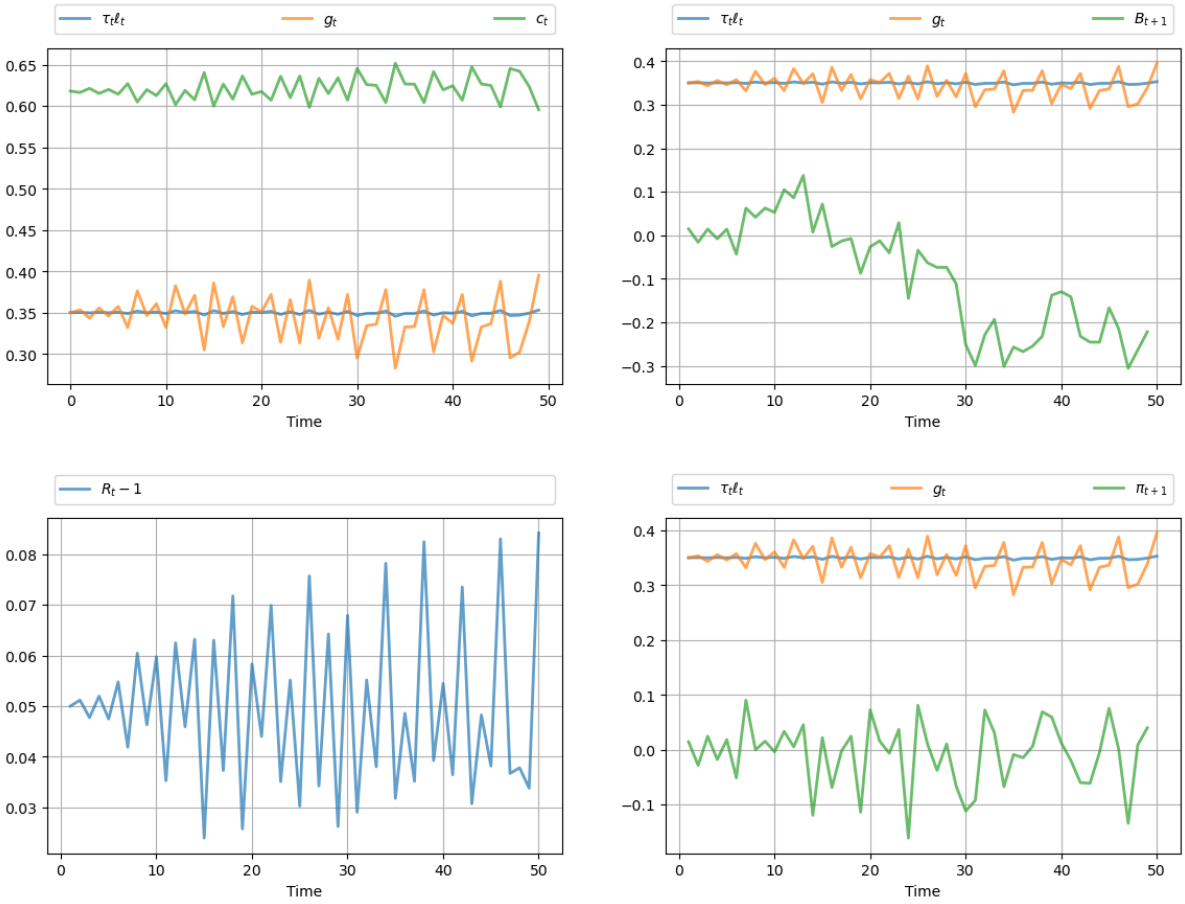
Solution to Exercise 12.5.1

```
# == Parameters == #
β = 1 / 1.05
ρ, mg = .95, .35
A = np.array([[0, 0, 0, ρ, mg*(1-ρ)],
              [1, 0, 0, 0,         0],
              [0, 1, 0, 0,         0],
              [0, 0, 1, 0,         0],
              [0, 0, 0, 0,         1]])
C = np.zeros((5, 1))
C[0, 0] = np.sqrt(1 - ρ**2) * mg / 8
Sg = np.array((1, 0, 0, 0, 0)).reshape(1, 5)
Sd = np.array((0, 0, 0, 0, 0)).reshape(1, 5)
# Chosen st. (Sc + Sg) * x0 = 1
Sb = np.array((0, 0, 0, 0, 2.135)).reshape(1, 5)
Ss = np.array((0, 0, 0, 0, 0)).reshape(1, 5)

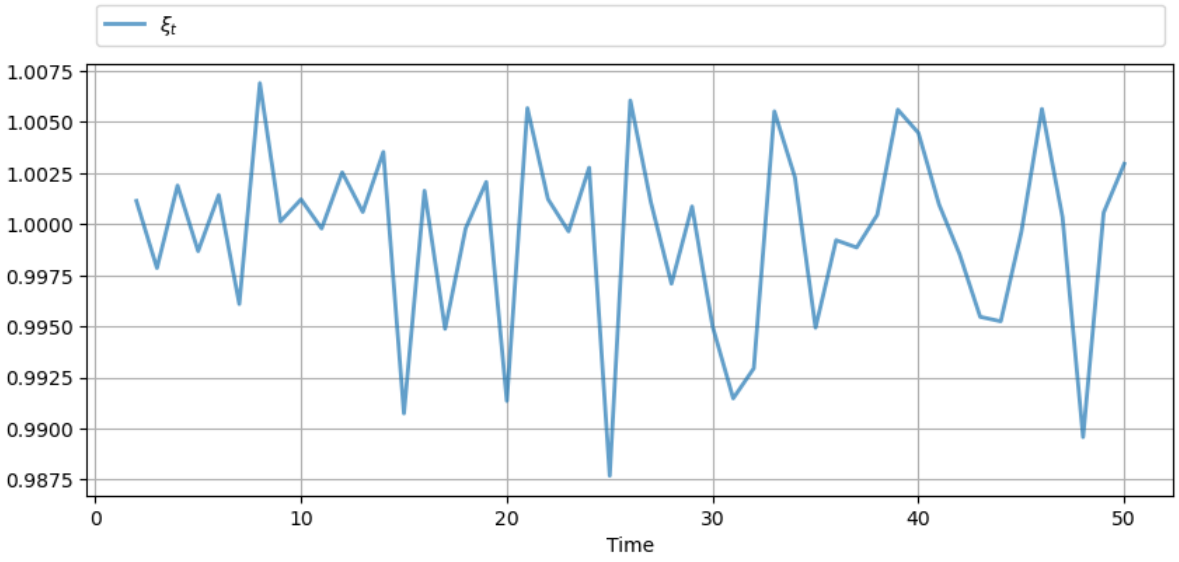
economy = Economy(β=β, Sg=Sg, Sd=Sd, Sb=Sb,
                  Ss=Ss, discrete=False, proc=(A, C))

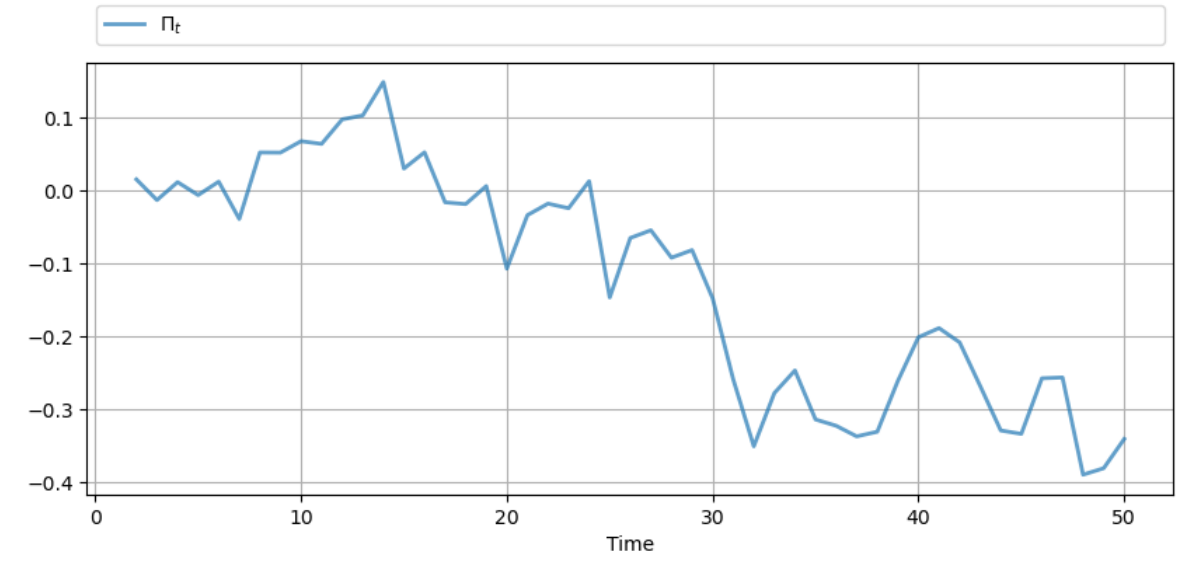
T = 50
path = compute_paths(T, economy)

gen_fig_1(path)
```




```
gen_fig_2(path)
```


ξ_t
1.0075
1.0050
1.0025
1.0000
0.9975
0.9950
0.9925
0.9900
0.9875
0
10
20
30
40
50
Time


Π_t
0.1
0.0
−0.1
−0.2
−0.3
−0.4
0
10
20
30
40
50
Time

Part III

Multiple Agent Models

CHAPTER

THIRTEEN

DEFAULT RISK AND INCOME FLUCTUATIONS

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install --upgrade quantecon
```

13.1 Overview

This lecture computes versions of Arellano's [Arellano, 2008] model of sovereign default.

The model describes interactions among default risk, output, and an equilibrium interest rate that includes a premium for endogenous default risk.

The decision maker is a government of a small open economy that borrows from risk-neutral foreign creditors.

The foreign lenders must be compensated for default risk.

The government borrows and lends abroad in order to smooth the consumption of its citizens.

The government repays its debt only if it wants to, but declining to pay has adverse consequences.

The interest rate on government debt adjusts in response to the state-dependent default probability chosen by government.

The model yields outcomes that help interpret sovereign default experiences, including

- countercyclical interest rates on sovereign debt
- countercyclical trade balances
- high volatility of consumption relative to output

Notably, long recessions caused by bad draws in the income process increase the government's incentive to default.

This can lead to

- spikes in interest rates
- temporary losses of access to international credit markets
- large drops in output, consumption, and welfare
- large capital outflows during recessions

Such dynamics are consistent with experiences of many countries.

Let's start with some imports:

```
import matplotlib.pyplot as plt
import numpy as np
import quantecon as qe
from numba import njit, prange
```

13.2 Structure

In this section we describe the main features of the model.

13.2.1 Output, Consumption and Debt

A small open economy is endowed with an exogenous stochastically fluctuating potential output stream $\{y_t\}$.

Potential output is realized only in periods in which the government honors its sovereign debt.

The output good can be traded or consumed.

The sequence $\{y_t\}$ is described by a Markov process with stochastic density kernel $p(y, y')$.

Households within the country are identical and rank stochastic consumption streams according to

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t u(c_t) \tag{13.1}$$

Here

- $0 < \beta < 1$ is a time discount factor
- u is an increasing and strictly concave utility function

Consumption sequences enjoyed by households are affected by the government's decision to borrow or lend internationally.

The government is benevolent in the sense that its aim is to maximize (13.1).

The government is the only domestic actor with access to foreign credit.

Because household are averse to consumption fluctuations, the government will try to smooth consumption by borrowing from (and lending to) foreign creditors.

13.2.2 Asset Markets

The only credit instrument available to the government is a one-period bond traded in international credit markets.

The bond market has the following features

- The bond matures in one period and is not state contingent.
- A purchase of a bond with face value B' is a claim to B' units of the consumption good next period.
- To purchase B' next period costs qB' now, or, what is equivalent.
- For selling $-B'$ units of next period goods the seller earns $-qB'$ of today's goods.
  - If $B' < 0$, then $-qB'$ units of the good are received in the current period, for a promise to repay $-B'$ units next period.
  - There is an equilibrium price function $q(B', y)$ that makes q depend on both B' and y.

Earnings on the government portfolio are distributed (or, if negative, taxed) lump sum to households.

When the government is not excluded from financial markets, the one-period national budget constraint is

$$c = y + B - q(B', y)B' \tag{13.2}$$

Here and below, a prime denotes a next period value or a claim maturing next period.

To rule out Ponzi schemes, we also require that $B \geq -Z$ in every period.

- Z is chosen to be sufficiently large that the constraint never binds in equilibrium.

13.2.3 Financial Markets

Foreign creditors

- are risk neutral
- know the domestic output stochastic process $\{y_t\}$ and observe $y_t, y_{t-1}, \ldots$, at time t
- can borrow or lend without limit in an international credit market at a constant international interest rate r
- receive full payment if the government chooses to pay
- receive zero if the government defaults on its one-period debt due

When a government is expected to default next period with probability δ, the expected value of a promise to pay one unit of consumption next period is $1 - \delta$.

Therefore, the discounted expected value of a promise to pay B next period is

$$q = \frac{1 - \delta}{1 + r} \tag{13.3}$$

Next we turn to how the government in effect chooses the default probability δ.

13.2.4 Government's Decisions

At each point in time t, the government chooses between

1. defaulting
2. meeting its current obligations and purchasing or selling an optimal quantity of one-period sovereign debt

Defaulting means declining to repay all of its current obligations.

If the government defaults in the current period, then consumption equals current output.

But a sovereign default has two consequences:

1. Output immediately falls from y to $h(y)$, where $0 \leq h(y) \leq y$.
    - It returns to y only after the country regains access to international credit markets.
2. The country loses access to foreign credit markets.

13.2.5 Reentering International Credit Market

While in a state of default, the economy regains access to foreign credit in each subsequent period with probability θ.

13.3 Equilibrium

Informally, an equilibrium is a sequence of interest rates on its sovereign debt, a stochastic sequence of government default decisions and an implied flow of household consumption such that

1. Consumption and assets satisfy the national budget constraint.
2. The government maximizes household utility taking into account
    - the resource constraint
    - the effect of its choices on the price of bonds
    - consequences of defaulting now for future net output and future borrowing and lending opportunities
3. The interest rate on the government's debt includes a risk-premium sufficient to make foreign creditors expect on average to earn the constant risk-free international interest rate.

To express these ideas more precisely, consider first the choices of the government, which

1. enters a period with initial assets B, or what is the same thing, initial debt to be repaid now of $-B$
2. observes current output y, and
3. chooses either
    1. to default, or
    2. to pay $-B$ and set next period's debt due to $-B'$

In a recursive formulation,

- state variables for the government comprise the pair (B, y)
- $v(B, y)$ is the optimum value of the government's problem when at the beginning of a period it faces the choice of whether to honor or default
- $v_c(B, y)$ is the value of choosing to pay obligations falling due
- $v_d(y)$ is the value of choosing to default

$v_d(y)$ does not depend on B because, when access to credit is eventually regained, net foreign assets equal 0.

Expressed recursively, the value of defaulting is

$$v_d(y) = u(h(y)) + \beta \int \{\theta v(0, y') + (1 - \theta) v_d(y')\} p(y, y') dy'$$

The value of paying is

$$v_c(B, y) = \max_{B' \geq -Z} \left\{ u(y - q(B', y)B' + B) + \beta \int v(B', y') p(y, y') dy' \right\}$$

The three value functions are linked by

$$v(B, y) = \max\{v_c(B, y), v_d(y)\}$$

The government chooses to default when

$$v_c(B, y) < v_d(y)$$

and hence given B' the probability of default next period is

$$\delta(B', y) := \int \mathbb{1}\{v_c(B', y') < v_d(y')\} p(y, y') dy' \tag{13.4}$$

Given zero profits for foreign creditors in equilibrium, we can combine (13.3) and (13.4) to pin down the bond price function:

$$q(B', y) = \frac{1 - \delta(B', y)}{1 + r} \tag{13.5}$$

13.3.1 Definition of Equilibrium

An *equilibrium* is

- a pricing function $q(B', y)$,
- a triple of value functions $(v_c(B, y), v_d(y), v(B, y))$,
- a decision rule telling the government when to default and when to pay as a function of the state (B, y), and
- an asset accumulation rule that, conditional on choosing not to default, maps (B, y) into B'

such that

- The three Bellman equations for $(v_c(B, y), v_d(y), v(B, y))$ are satisfied
- Given the price function $q(B', y)$, the default decision rule and the asset accumulation decision rule attain the optimal value function $v(B, y)$, and
- The price function $q(B', y)$ satisfies equation (13.5)

13.4 Computation

Let's now compute an equilibrium of Arellano's model.

The equilibrium objects are the value function $v(B, y)$, the associated default decision rule, and the pricing function $q(B', y)$.

We'll use our code to replicate Arellano's results.

After that we'll perform some additional simulations.

We use a slightly modified version of the algorithm recommended by Arellano.

- The appendix to [Arellano, 2008] recommends value function iteration until convergence, updating the price, and then repeating.
- Instead, we update the bond price at every value function iteration step.

The second approach is faster and the two different procedures deliver very similar results.

Here is a more detailed description of our algorithm:

1. Guess a pair of non-default and default value functions v_c and v_d.
2. Using these functions, calculate the value function v, the corresponding default probabilities and the price function q.
3. At each pair (B, y),
    1. update the value of defaulting $v_d(y)$.

2. update the value of remaining $v_c(B, y)$.

4. Check for convergence. If converged, stop – if not, go to step 2.

We use simple discretization on a grid of asset holdings and income levels.

The output process is discretized using a quadrature method due to Tauchen.

As we have in other places, we accelerate our code using Numba.

We define a class that will store parameters, grids and transition probabilities.

```
class Arellano_Economy:
    " Stores data and creates primitives for the Arellano economy. "

    def __init__(self,
            B_grid_size= 251,    # Grid size for bonds
            B_grid_min=-0.45,    # Smallest B value
            B_grid_max=0.45,     # Largest B value
            y_grid_size=51,      # Grid size for income
            β=0.953,             # Time discount parameter
            γ=2.0,               # Utility parameter
            r=0.017,             # Lending rate
            ρ=0.945,             # Persistence in the income process
            η=0.025,             # Standard deviation of the income process
            θ=0.282,             # Prob of re-entering financial markets
            def_y_param=0.969): # Parameter governing income in default

        # Save parameters
        self.β, self.γ, self.r, = β, γ, r
        self.ρ, self.η, self.θ = ρ, η, θ

        self.y_grid_size = y_grid_size
        self.B_grid_size = B_grid_size
        self.B_grid = np.linspace(B_grid_min, B_grid_max, B_grid_size)
        mc = qe.markov.tauchen(y_grid_size, ρ, η, 0, 3)
        self.y_grid, self.P = np.exp(mc.state_values), mc.P

        # The index at which B_grid is (close to) zero
        self.B0_idx = np.searchsorted(self.B_grid, 1e-10)

        # Output recieved while in default, with same shape as y_grid
        self.def_y = np.minimum(def_y_param * np.mean(self.y_grid), self.y_grid)

    def params(self):
        return self.β, self.γ, self.r, self.ρ, self.η, self.θ

    def arrays(self):
        return self.P, self.y_grid, self.B_grid, self.def_y, self.B0_idx
```

Notice how the class returns the data it stores as simple numerical values and arrays via the methods `params` and `arrays`.

We will use this data in the Numba-jitted functions defined below.

Jitted functions prefer simple arguments, since type inference is easier.

Here is the utility function.

```
@njit
def u(c, γ):
    return c**(1-γ)/(1-γ)
```

Here is a function to compute the bond price at each state, given v_c and v_d.

```
@njit
def compute_q(v_c, v_d, q, params, arrays):
    """
    Compute the bond price function q(b, y) at each (b, y) pair.

    This function writes to the array q that is passed in as an argument.
    """

    # Unpack
    β, γ, r, ρ, η, θ = params
    P, y_grid, B_grid, def_y, B0_idx = arrays

    for B_idx in range(len(B_grid)):
        for y_idx in range(len(y_grid)):
            # Compute default probability and corresponding bond price
            delta = P[y_idx, v_c[B_idx, :] < v_d].sum()
            q[B_idx, y_idx] = (1 - delta ) / (1 + r)
```

Next we introduce Bellman operators that updated v_d and v_c.

```
@njit
def T_d(y_idx, v_c, v_d, params, arrays):
    """
    The RHS of the Bellman equation when income is at index y_idx and
    the country has chosen to default.  Returns an update of v_d.
    """
    # Unpack
    β, γ, r, ρ, η, θ = params
    P, y_grid, B_grid, def_y, B0_idx = arrays

    current_utility = u(def_y[y_idx], γ)
    v = np.maximum(v_c[B0_idx, :], v_d)
    cont_value = np.sum((θ * v + (1 - θ) * v_d) * P[y_idx, :])

    return current_utility + β * cont_value


@njit
def T_c(B_idx, y_idx, v_c, v_d, q, params, arrays):
    """
    The RHS of the Bellman equation when the country is not in a
    defaulted state on their debt.  Returns a value that corresponds to
    v_c[B_idx, y_idx], as well as the optimal level of bond sales B'.
    """
    # Unpack
    β, γ, r, ρ, η, θ = params
    P, y_grid, B_grid, def_y, B0_idx = arrays
    B = B_grid[B_idx]
    y = y_grid[y_idx]
```

(continues on next page)

(continued from previous page)

```
    # Compute the RHS of Bellman equation
    current_max = -1e10
    # Step through choices of next period B'
    for Bp_idx, Bp in enumerate(B_grid):
        c = y + B - q[Bp_idx, y_idx] * Bp
        if c > 0:
            v = np.maximum(v_c[Bp_idx, :], v_d)
            val = u(c, γ) + β * np.sum(v * P[y_idx, :])
            if val > current_max:
                current_max = val
                Bp_star_idx = Bp_idx
    return current_max, Bp_star_idx
```

Here is a fast function that calls these operators in the right sequence.

```
@njit(parallel=True)
def update_values_and_prices(v_c, v_d,      # Current guess of value functions
                             B_star, q,     # Arrays to be written to
                             params, arrays):

    # Unpack
    β, γ, r, ρ, η, θ = params
    P, y_grid, B_grid, def_y, B0_idx = arrays
    y_grid_size = len(y_grid)
    B_grid_size = len(B_grid)

    # Compute bond prices and write them to q
    compute_q(v_c, v_d, q, params, arrays)

    # Allocate memory
    new_v_c = np.empty_like(v_c)
    new_v_d = np.empty_like(v_d)

    # Calculate and return new guesses for v_c and v_d
    for y_idx in prange(y_grid_size):
        new_v_d[y_idx] = T_d(y_idx, v_c, v_d, params, arrays)
        for B_idx in range(B_grid_size):
            new_v_c[B_idx, y_idx], Bp_idx = T_c(B_idx, y_idx,
                                                v_c, v_d, q, params, arrays)
            B_star[B_idx, y_idx] = Bp_idx

    return new_v_c, new_v_d
```

We can now write a function that will use the `Arellano_Economy` class and the functions defined above to compute the solution to our model.

We do not need to JIT compile this function since it only consists of outer loops (and JIT compiling makes almost zero difference).

In fact, one of the jobs of this function is to take an instance of `Arellano_Economy`, which is hard for the JIT compiler to handle, and strip it down to more basic objects, which are then passed out to jitted functions.

```
def solve(model, tol=1e-8, max_iter=10_000):
    """
    Given an instance of Arellano_Economy, this function computes the optimal
    policy and value functions.
```

(continues on next page)

(continued from previous page)

```
    """
    # Unpack
    params = model.params()
    arrays = model.arrays()
    y_grid_size, B_grid_size = model.y_grid_size, model.B_grid_size

    # Initial conditions for v_c and v_d
    v_c = np.zeros((B_grid_size, y_grid_size))
    v_d = np.zeros(y_grid_size)

    # Allocate memory
    q = np.empty_like(v_c)
    B_star = np.empty_like(v_c, dtype=int)

    current_iter = 0
    dist = np.inf
    while (current_iter < max_iter) and (dist > tol):

        if current_iter % 100 == 0:
            print(f"Entering iteration {current_iter}.")

        new_v_c, new_v_d = update_values_and_prices(v_c, v_d, B_star, q, params,␣
↪arrays)
        # Check tolerance and update
        dist = np.max(np.abs(new_v_c - v_c)) + np.max(np.abs(new_v_d - v_d))
        v_c = new_v_c
        v_d = new_v_d
        current_iter += 1

    print(f"Terminating at iteration {current_iter}.")
    return v_c, v_d, q, B_star
```

Finally, we write a function that will allow us to simulate the economy once we have the policy functions

```
def simulate(model, T, v_c, v_d, q, B_star, y_idx=None, B_idx=None):
    """
    Simulates the Arellano 2008 model of sovereign debt

    Here `model` is an instance of `Arellano_Economy` and `T` is the length of
    the simulation.  Endogenous objects `v_c`, `v_d`, `q` and `B_star` are
    assumed to come from a solution to `model`.

    """
    # Unpack elements of the model
    B0_idx = model.B0_idx
    y_grid = model.y_grid
    B_grid, y_grid, P = model.B_grid, model.y_grid, model.P

    # Set initial conditions to middle of grids
    if y_idx == None:
        y_idx = np.searchsorted(y_grid, y_grid.mean())
    if B_idx == None:
        B_idx = B0_idx
    in_default = False

    # Create Markov chain and simulate income process
```

(continues on next page)

(continued from previous page)

```
    mc = qe.MarkovChain(P, y_grid)
    y_sim_indices = mc.simulate_indices(T+1, init=y_idx)

    # Allocate memory for outputs
    y_sim = np.empty(T)
    y_a_sim = np.empty(T)
    B_sim = np.empty(T)
    q_sim = np.empty(T)
    d_sim = np.empty(T, dtype=int)

    # Perform simulation
    t = 0
    while t < T:

        # Store the value of y_t and B_t
        y_sim[t] = y_grid[y_idx]
        B_sim[t] = B_grid[B_idx]

        # if in default:
        if v_c[B_idx, y_idx] < v_d[y_idx] or in_default:
            y_a_sim[t] = model.def_y[y_idx]
            d_sim[t] = 1
            Bp_idx = B0_idx
            # Re-enter financial markets next period with prob θ
            in_default = False if np.random.rand() < model.θ else True
        else:
            y_a_sim[t] = y_sim[t]
            d_sim[t] = 0
            Bp_idx = B_star[B_idx, y_idx]

        q_sim[t] = q[Bp_idx, y_idx]

        # Update time and indices
        t += 1
        y_idx = y_sim_indices[t]
        B_idx = Bp_idx

    return y_sim, y_a_sim, B_sim, q_sim, d_sim
```

13.5 Results

Let's start by trying to replicate the results obtained in [Arellano, 2008].

In what follows, all results are computed using Arellano's parameter values.

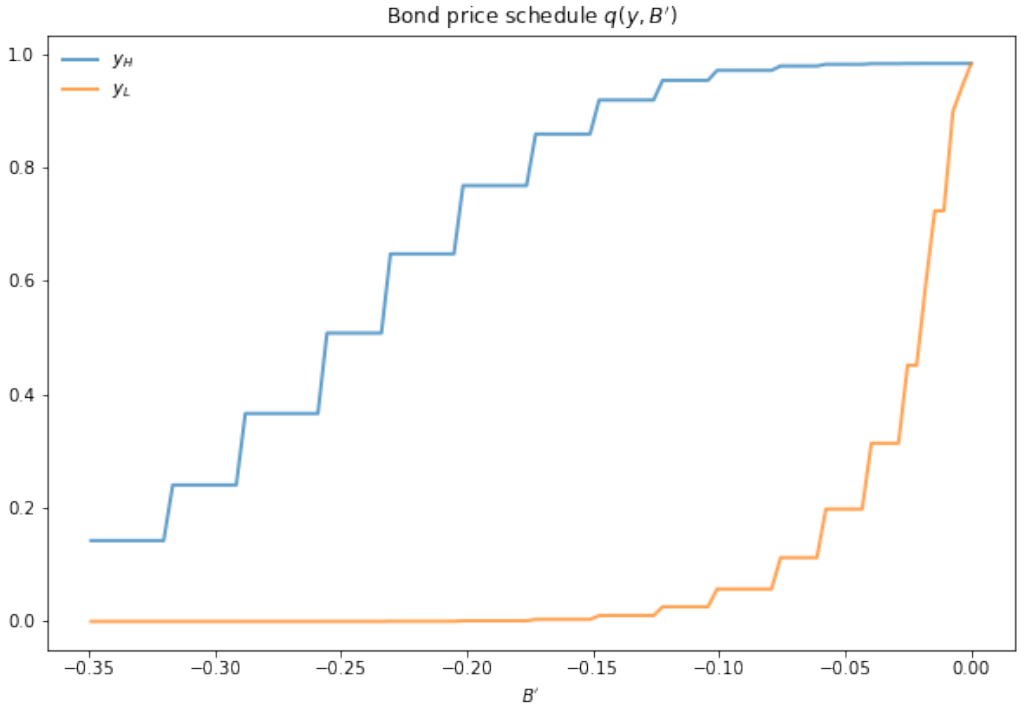
The values can be seen in the `__init__` method of the `Arellano_Economy` shown above.

For example, `r=0.017` matches the average quarterly rate on a 5 year US treasury over the period 1983–2001.

Details on how to compute the figures are reported as solutions to the exercises.

The first figure shows the bond price schedule and replicates Figure 3 of Arellano, where y_L and Y_H are particular below average and above average values of output y.

- y_L is 5% below the mean of the y grid values
- y_H is 5% above the mean of the y grid values



The grid used to compute this figure was relatively fine (`y_grid_size, B_grid_size = 51, 251`), which explains the minor differences between this and Arrelano's figure.

The figure shows that

- Higher levels of debt (larger $-B'$) induce larger discounts on the face value, which correspond to higher interest rates.
- Lower income also causes more discounting, as foreign creditors anticipate greater likelihood of default.

The next figure plots value functions and replicates the right hand panel of Figure 4 of [Arellano, 2008].

We can use the results of the computation to study the default probability $\delta(B', y)$ defined in (13.4).

The next plot shows these default probabilities over (B', y) as a heat map.

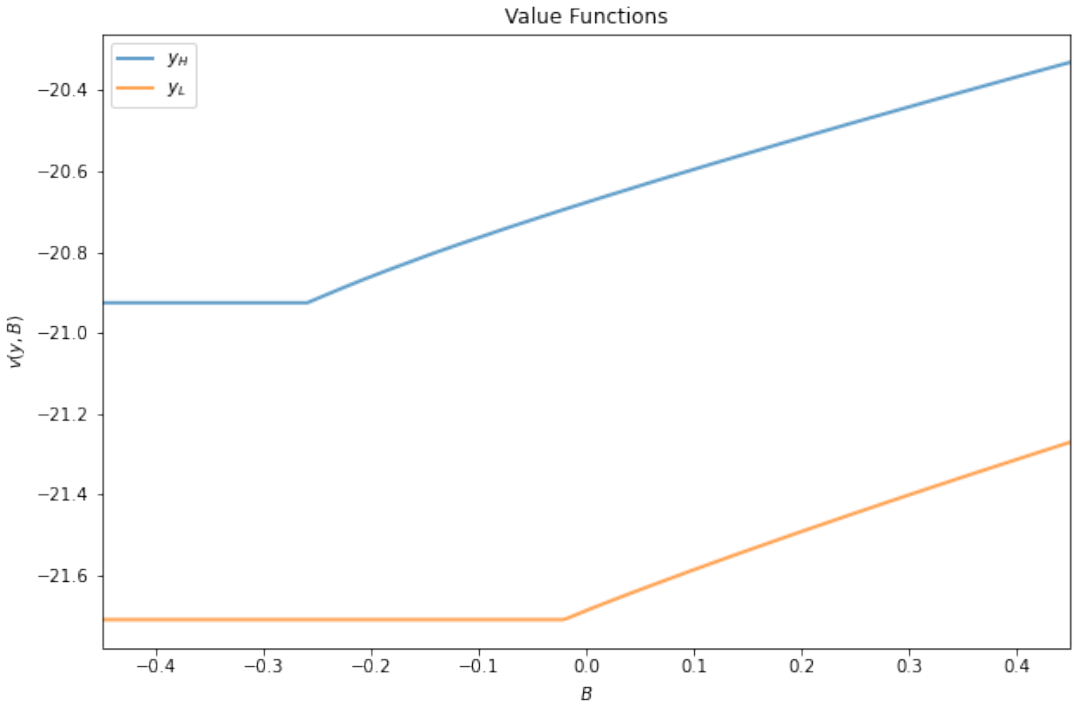
As anticipated, the probability that the government chooses to default in the following period increases with indebtedness and falls with income.

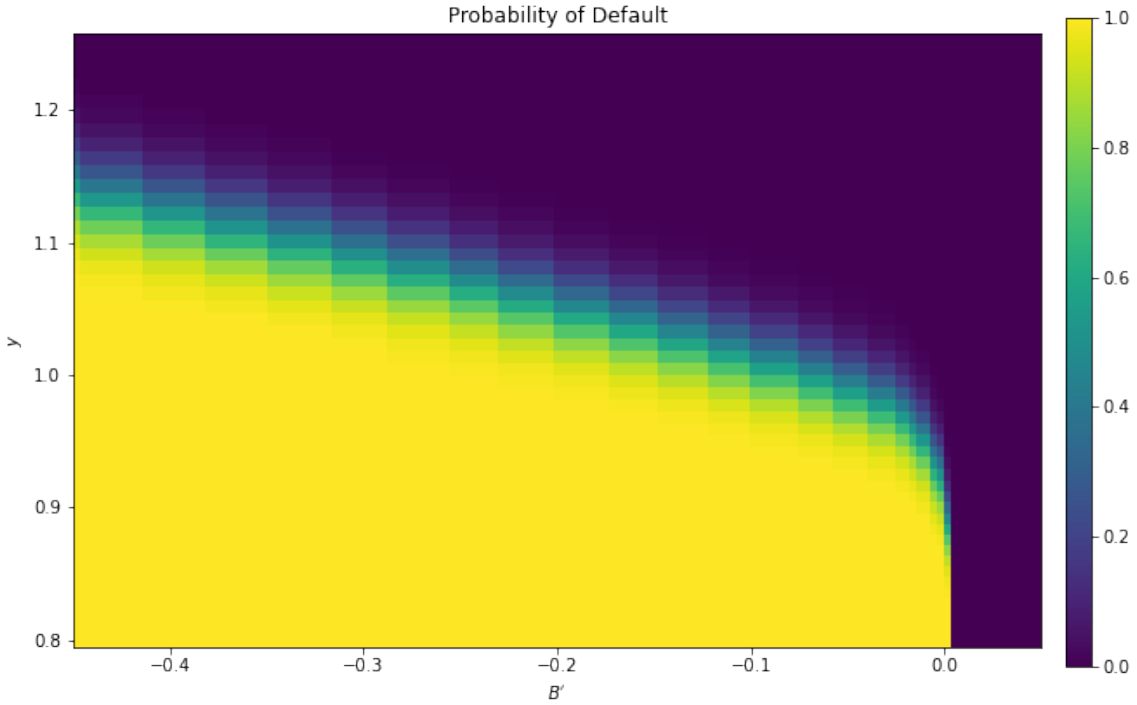
Next let's run a time series simulation of $\{y_t\}$, $\{B_t\}$ and $q(B_{t+1}, y_t)$.

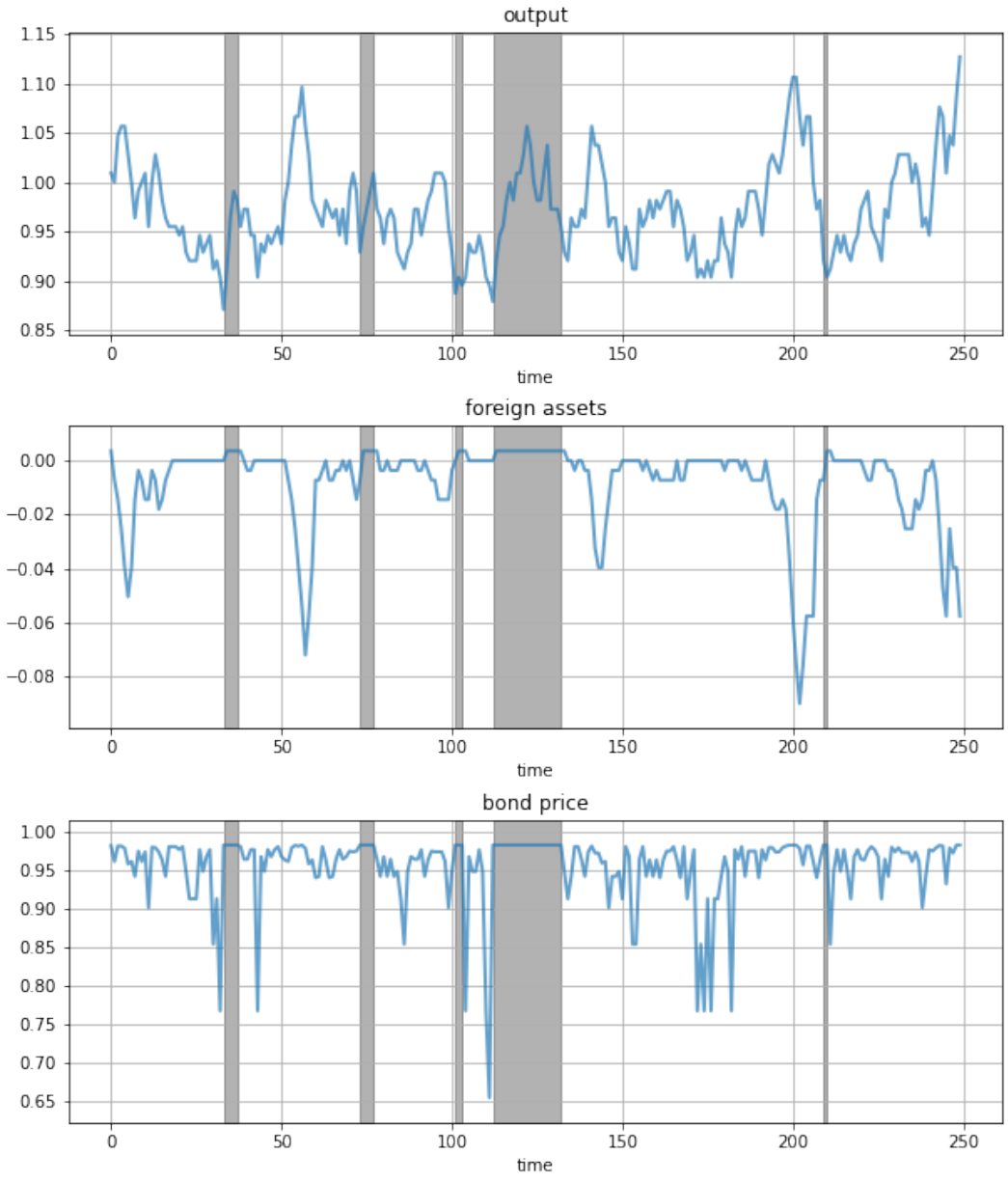
The grey vertical bars correspond to periods when the economy is excluded from financial markets because of a past default.

One notable feature of the simulated data is the nonlinear response of interest rates.

Periods of relative stability are followed by sharp spikes in the discount rate on government debt.


Value Functions
y_H
y_L
−20.4
−20.6
−20.8
−21.0
−21.2
−21.4
−21.6
v(y, B)
−0.4
−0.3
−0.2
−0.1
0.0
0.1
0.2
0.3
0.4
B


Probability of Default
1.2
1.1
1.0
0.9
0.8
y
−0.4
−0.3
−0.2
−0.1
0.0
B′
1.0
0.8
0.6
0.4
0.2
0.0


output
1.15
1.10
1.05
1.00
0.95
0.90
0.85
0
50
100
150
200
250
time
foreign assets
0.00
−0.02
−0.04
−0.06
−0.08
0
50
100
150
200
250
time
bond price
1.00
0.95
0.90
0.85
0.80
0.75
0.70
0.65
0
50
100
150
200
250
time

13.6 Exercises

Exercise 13.6.1

To the extent that you can, replicate the figures shown above

- Use the parameter values listed as defaults in `Arellano_Economy`.
- The time series will of course vary depending on the shock draws.

Solution to Exercise 13.6.1

Compute the value function, policy and equilibrium prices

```
ae = Arellano_Economy()
```

```
v_c, v_d, q, B_star = solve(ae)
```

```
Entering iteration 0.
```

```
Entering iteration 100.
```

```
Entering iteration 200.
```

```
Entering iteration 300.
```

```
Terminating at iteration 399.
```

Compute the bond price schedule as seen in figure 3 of Arellano (2008)

```
# Unpack some useful names
B_grid, y_grid, P = ae.B_grid, ae.y_grid, ae.P
B_grid_size, y_grid_size = len(B_grid), len(y_grid)
r = ae.r

# Create "Y High" and "Y Low" values as 5% devs from mean
high, low = np.mean(y_grid) * 1.05, np.mean(y_grid) * .95
iy_high, iy_low = (np.searchsorted(y_grid, x) for x in (high, low))

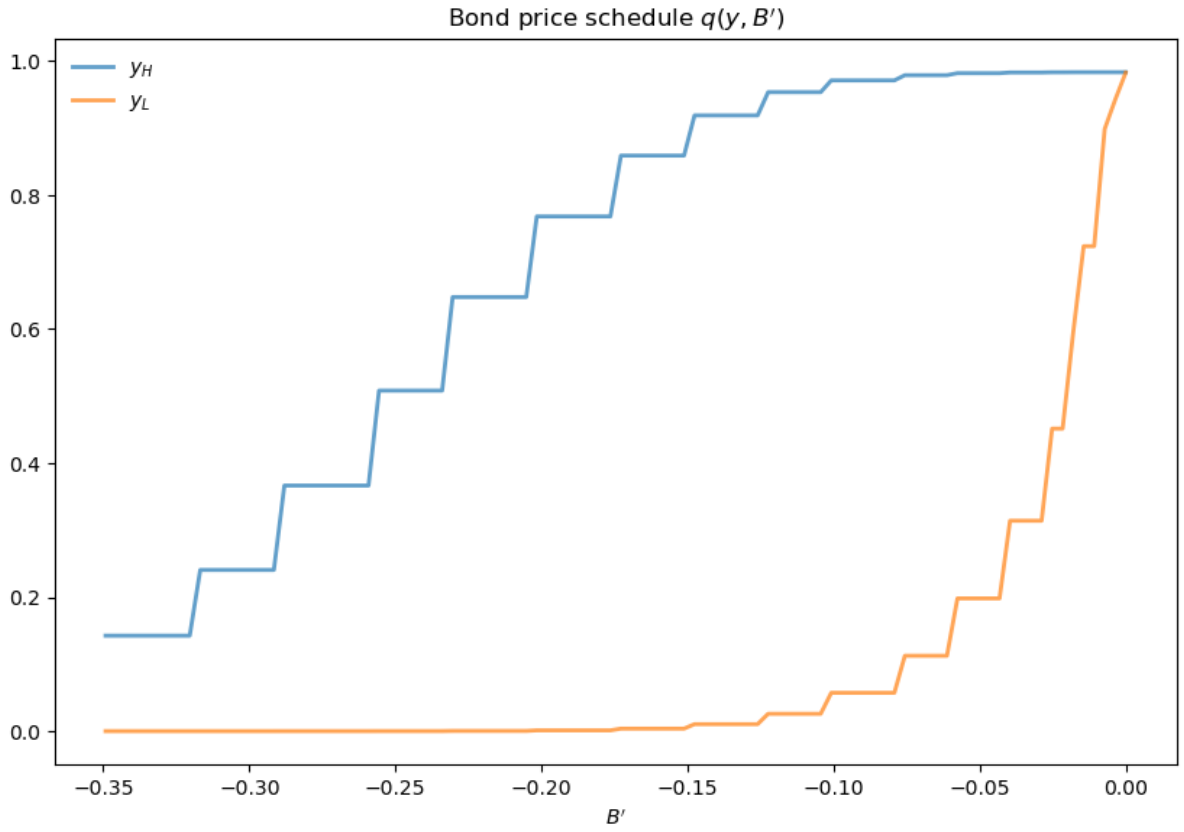
fig, ax = plt.subplots(figsize=(10, 6.5))
ax.set_title("Bond price schedule $q(y, B')$")

# Extract a suitable plot grid
x = []
q_low = []
q_high = []
for i, B in enumerate(B_grid):
    if -0.35 <= B <= 0:  # To match fig 3 of Arellano
        x.append(B)
        q_low.append(q[i, iy_low])
```

(continues on next page)

(continued from previous page)

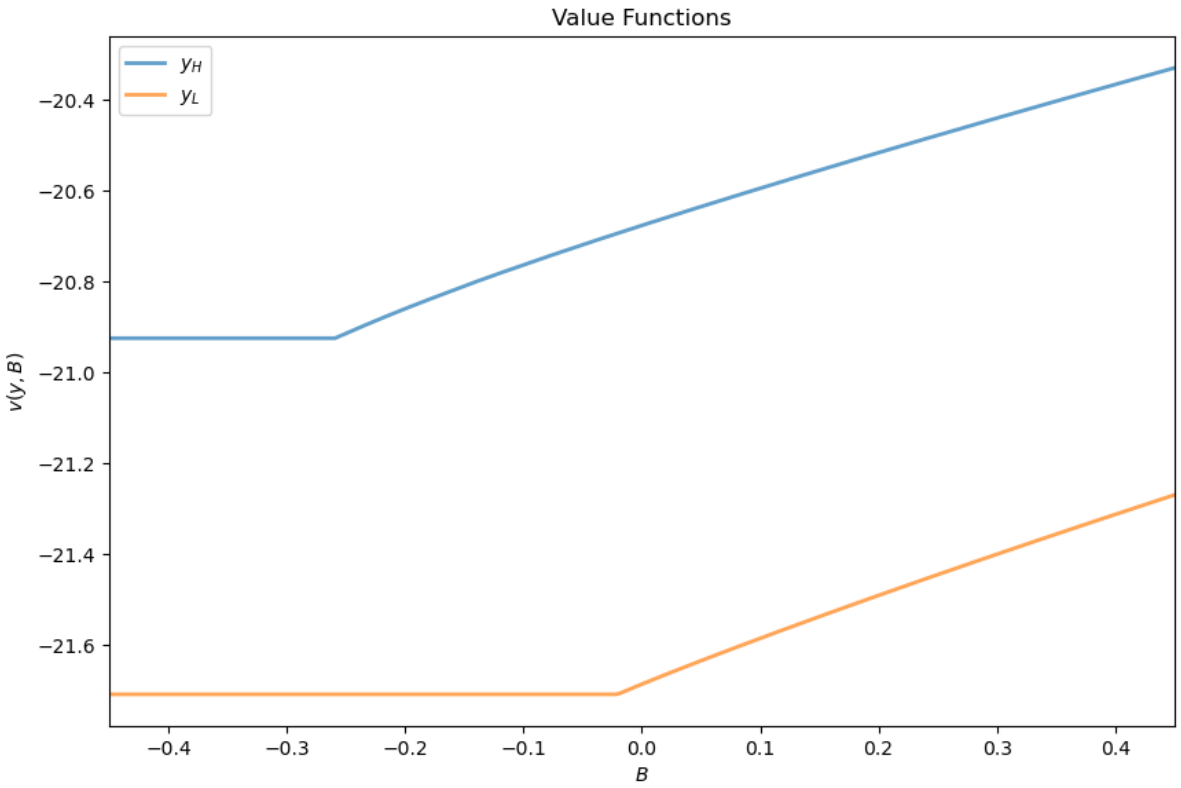
```
        q_high.append(q[i, iy_high])
ax.plot(x, q_high, label="$y_H$", lw=2, alpha=0.7)
ax.plot(x, q_low, label="$y_L$", lw=2, alpha=0.7)
ax.set_xlabel("$B'$")
ax.legend(loc='upper left', frameon=False)
plt.show()
```



Draw a plot of the value functions

```
v = np.maximum(v_c, np.reshape(v_d, (1, y_grid_size)))

fig, ax = plt.subplots(figsize=(10, 6.5))
ax.set_title("Value Functions")
ax.plot(B_grid, v[:, iy_high], label="$y_H$", lw=2, alpha=0.7)
ax.plot(B_grid, v[:, iy_low], label="$y_L$", lw=2, alpha=0.7)
ax.legend(loc='upper left')
ax.set(xlabel="$B$", ylabel="$v(y, B)$")
ax.set_xlim(min(B_grid), max(B_grid))
plt.show()
```

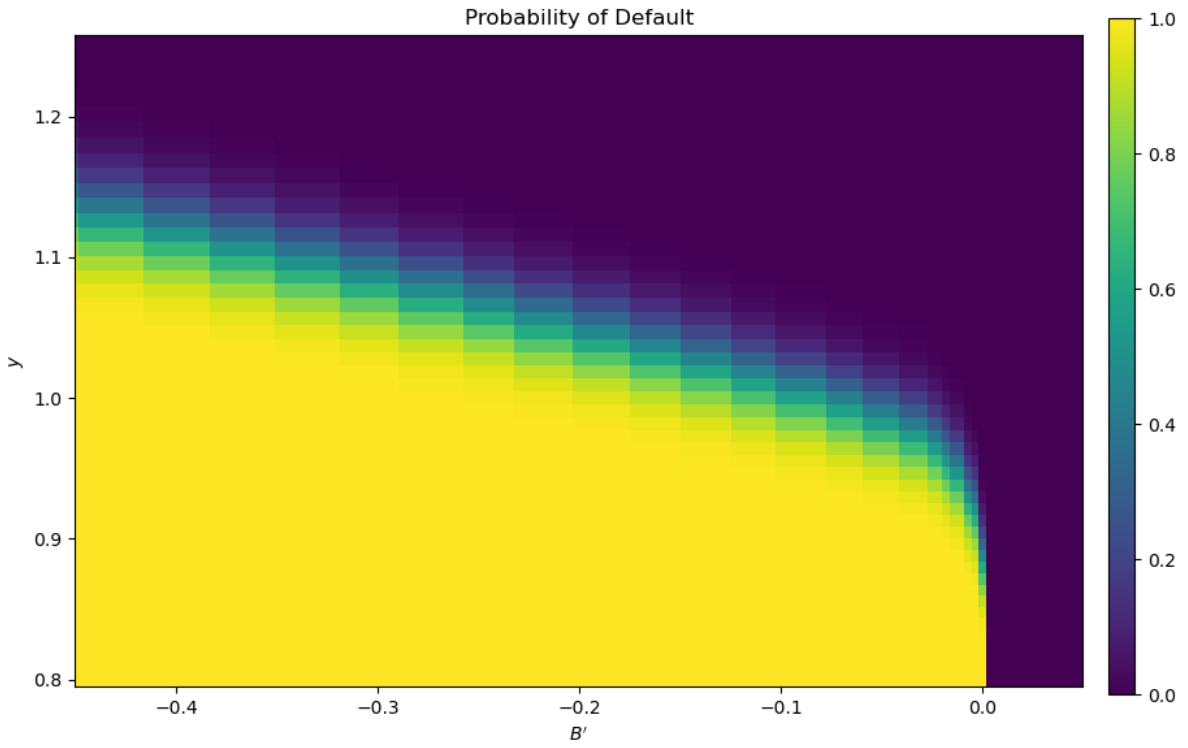


Draw a heat map for default probability

```
xx, yy = B_grid, y_grid
zz = np.empty_like(v_c)

for B_idx in range(B_grid_size):
    for y_idx in range(y_grid_size):
        zz[B_idx, y_idx] = P[y_idx, v_c[B_idx, :] < v_d].sum()

# Create figure
fig, ax = plt.subplots(figsize=(10, 6.5))
hm = ax.pcolormesh(xx, yy, zz.T)
cax = fig.add_axes([.92, .1, .02, .8])
fig.colorbar(hm, cax=cax)
ax.axis([xx.min(), 0.05, yy.min(), yy.max()])
ax.set(xlabel="$B'$", ylabel="$y$", title="Probability of Default")
plt.show()
```



Plot a time series of major variables simulated from the model

```
T = 250
np.random.seed(42)
y_sim, y_a_sim, B_sim, q_sim, d_sim = simulate(ae, T, v_c, v_d, q, B_star)

# Pick up default start and end dates
start_end_pairs = []
i = 0
while i < len(d_sim):
    if d_sim[i] == 0:
        i += 1
    else:
        # If we get to here we're in default
        start_default = i
        while i < len(d_sim) and d_sim[i] == 1:
            i += 1
        end_default = i - 1
        start_end_pairs.append((start_default, end_default))

plot_series = (y_sim, B_sim, q_sim)
titles = 'output', 'foreign assets', 'bond price'

fig, axes = plt.subplots(len(plot_series), 1, figsize=(10, 12))
fig.subplots_adjust(hspace=0.3)

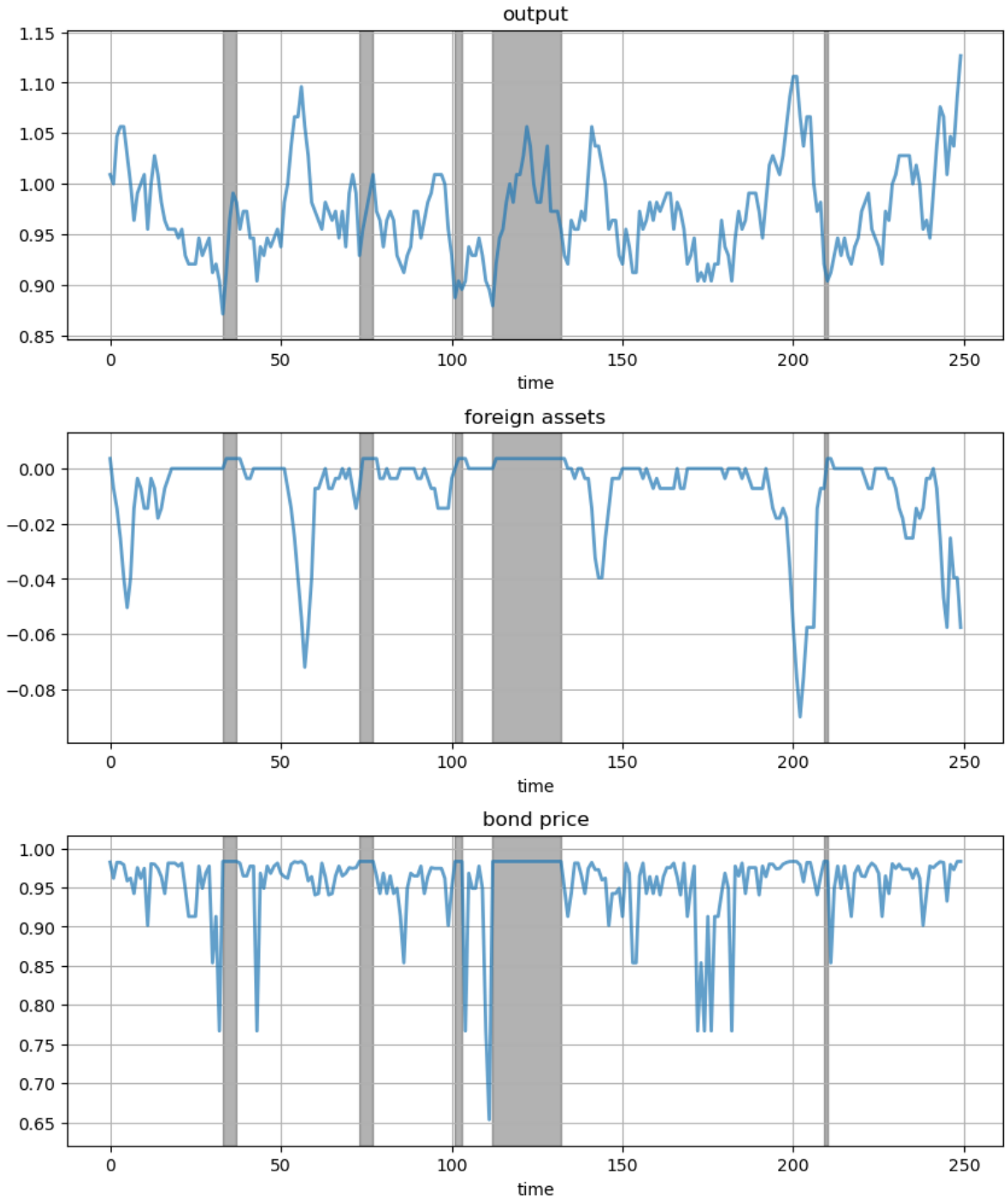
for ax, series, title in zip(axes, plot_series, titles):
    # Determine suitable y limits
    s_max, s_min = max(series), min(series)
    s_range = s_max - s_min
    y_max = s_max + s_range * 0.1
```

(continues on next page)

(continued from previous page)

```
    y_min = s_min - s_range * 0.1
    ax.set_ylim(y_min, y_max)
    for pair in start_end_pairs:
        ax.fill_between(pair, (y_min, y_min), (y_max, y_max),
                        color='k', alpha=0.3)
    ax.grid()
    ax.plot(range(T), series, lw=2, alpha=0.7)
    ax.set(title=title, xlabel="time")

plt.show()
```


output
1.15
1.10
1.05
1.00
0.95
0.90
0.85
0
50
100
150
200
250
time
foreign assets
0.00
−0.02
−0.04
−0.06
−0.08
0
50
100
150
200
250
time
bond price
1.00
0.95
0.90
0.85
0.80
0.75
0.70
0.65
0
50
100
150
200
250
time

CHAPTER

FOURTEEN

GLOBALIZATION AND CYCLES

14.1 Overview

In this lecture, we review the paper Globalization and Synchronization of Innovation Cycles by Kiminori Matsuyama, Laura Gardini and Iryna Sushko.

This model helps us understand several interesting stylized facts about the world economy.

One of these is synchronized business cycles across different countries.

Most existing models that generate synchronized business cycles do so by assumption, since they tie output in each country to a common shock.

They also fail to explain certain features of the data, such as the fact that the degree of synchronization tends to increase with trade ties.

By contrast, in the model we consider in this lecture, synchronization is both endogenous and increasing with the extent of trade integration.

In particular, as trade costs fall and international competition increases, innovation incentives become aligned and countries synchronize their innovation cycles.

Let's start with some imports:

```
import numpy as np
import matplotlib.pyplot as plt
from numba import jit
from ipywidgets import interact
```

14.1.1 Background

The model builds on work by Judd [Judd, 1985], Deneckner and Judd [Deneckere and Judd, 1992] and Helpman and Krugman [Helpman and Krugman, 1985] by developing a two-country model with trade and innovation.

On the technical side, the paper introduces the concept of coupled oscillators to economic modeling.

As we will see, coupled oscillators arise endogenously within the model.

Below we review the model and replicate some of the results on synchronization of innovation across countries.

14.2 Key Ideas

It is helpful to begin with an overview of the mechanism.

14.2.1 Innovation Cycles

As discussed above, two countries produce and trade with each other.

In each country, firms innovate, producing new varieties of goods and, in doing so, receiving temporary monopoly power.

Imitators follow and, after one period of monopoly, what had previously been new varieties now enter competitive production.

Firms have incentives to innovate and produce new goods when the mass of varieties of goods currently in production is relatively low.

In addition, there are strategic complementarities in the timing of innovation.

Firms have incentives to innovate in the same period, so as to avoid competing with substitutes that are competitively produced.

This leads to temporal clustering in innovations in each country.

After a burst of innovation, the mass of goods currently in production increases.

However, goods also become obsolete, so that not all survive from period to period.

This mechanism generates a cycle, where the mass of varieties increases through simultaneous innovation and then falls through obsolescence.

14.2.2 Synchronization

In the absence of trade, the timing of innovation cycles in each country is decoupled.

This will be the case when trade costs are prohibitively high.

If trade costs fall, then goods produced in each country penetrate each other's markets.

As illustrated below, this leads to synchronization of business cycles across the two countries.

14.3 Model

Let's write down the model more formally.

(The treatment is relatively terse since full details can be found in the original paper)

Time is discrete with $t = 0, 1, \ldots$.

There are two countries indexed by j or k.

In each country, a representative household inelastically supplies L_j units of labor at wage rate $w_{j,t}$.

Without loss of generality, it is assumed that $L_1 \geq L_2$.

Households consume a single nontradeable final good which is produced competitively.

Its production involves combining two types of tradeable intermediate inputs via

$$Y_{k,t} = C_{k,t} = \left(\frac{X^o_{k,t}}{1-\alpha}\right)^{1-\alpha} \left(\frac{X_{k,t}}{\alpha}\right)^{\alpha}$$

Here $X^o_{k,t}$ is a homogeneous input which can be produced from labor using a linear, one-for-one technology.

It is freely tradeable, competitively supplied, and homogeneous across countries.

By choosing the price of this good as numeraire and assuming both countries find it optimal to always produce the homogeneous good, we can set $w_{1,t} = w_{2,t} = 1$.

The good $X_{k,t}$ is a composite, built from many differentiated goods via

$$X_{k,t}^{1-\frac{1}{\sigma}} = \int_{\Omega_t} \left[x_{k,t}(\nu)\right]^{1-\frac{1}{\sigma}} d\nu$$

Here $x_{k,t}(\nu)$ is the total amount of a differentiated good $\nu \in \Omega_t$ that is produced.

The parameter $\sigma > 1$ is the direct partial elasticity of substitution between a pair of varieties and Ω_t is the set of varieties available in period t.

We can split the varieties into those which are supplied competitively and those supplied monopolistically; that is, $\Omega_t = \Omega_t^c + \Omega_t^m$.

14.3.1 Prices

Demand for differentiated inputs is

$$x_{k,t}(\nu) = \left(\frac{p_{k,t}(\nu)}{P_{k,t}}\right)^{-\sigma} \frac{\alpha L_k}{P_{k,t}}$$

Here

- $p_{k,t}(\nu)$ is the price of the variety ν and
- $P_{k,t}$ is the price index for differentiated inputs in k, defined by

$$\left[P_{k,t}\right]^{1-\sigma} = \int_{\Omega_t} [p_{k,t}(\nu)]^{1-\sigma} d\nu$$

The price of a variety also depends on the origin, j, and destination, k, of the goods because shipping varieties between countries incurs an iceberg trade cost $\tau_{j,k}$.

Thus the effective price in country k of a variety ν produced in country j becomes $p_{k,t}(\nu) = \tau_{j,k}\, p_{j,t}(\nu)$.

Using these expressions, we can derive the total demand for each variety, which is

$$D_{j,t}(\nu) = \sum_k \tau_{j,k} x_{k,t}(\nu) = \alpha A_{j,t} (p_{j,t}(\nu))^{-\sigma}$$

where

$$A_{j,t} := \sum_k \frac{\rho_{j,k} L_k}{(P_{k,t})^{1-\sigma}} \quad \text{and} \quad \rho_{j,k} = (\tau_{j,k})^{1-\sigma} \leq 1$$

It is assumed that $\tau_{1,1} = \tau_{2,2} = 1$ and $\tau_{1,2} = \tau_{2,1} = \tau$ for some $\tau > 1$, so that

$$\rho_{1,2} = \rho_{2,1} = \rho := \tau^{1-\sigma} < 1$$

The value $\rho \in [0, 1)$ is a proxy for the degree of globalization.

Producing one unit of each differentiated variety requires ψ units of labor, so the marginal cost is equal to ψ for $\nu \in \Omega_{j,t}$.

Additionally, all competitive varieties will have the same price (because of equal marginal cost), which means that, for all $\nu \in \Omega^c$,

$$p_{j,t}(\nu) = p^c_{j,t} := \psi \quad \text{and} \quad D_{j,t} = y^c_{j,t} := \alpha A_{j,t} (p^c_{j,t})^{-\sigma}$$

Monopolists will have the same marked-up price, so, for all $\nu \in \Omega^m$,

$$p_{j,t}(\nu) = p^m_{j,t} := \frac{\psi}{1 - \frac{1}{\sigma}} \quad \text{and} \quad D_{j,t} = y^m_{j,t} := \alpha A_{j,t} (p^m_{j,t})^{-\sigma}$$

Define

$$\theta := \frac{p^c_{j,t}}{p^m_{j,t}} \frac{y^c_{j,t}}{y^m_{j,t}} = \left(1 - \frac{1}{\sigma}\right)^{1-\sigma}$$

Using the preceding definitions and some algebra, the price indices can now be rewritten as

$$\left(\frac{P_{k,t}}{\psi}\right)^{1-\sigma} = M_{k,t} + \rho M_{j,t} \quad \text{where} \quad M_{j,t} := N^c_{j,t} + \frac{N^m_{j,t}}{\theta}$$

The symbols $N^c_{j,t}$ and $N^m_{j,t}$ will denote the measures of Ω^c and Ω^m respectively.

14.3.2 New Varieties

To introduce a new variety, a firm must hire f units of labor per variety in each country.

Monopolist profits must be less than or equal to zero in expectation, so

$$N^m_{j,t} \geq 0, \quad \pi^m_{j,t} := (p^m_{j,t} - \psi) y^m_{j,t} - f \leq 0 \quad \text{and} \quad \pi^m_{j,t} N^m_{j,t} = 0$$

With further manipulations, this becomes

$$N^m_{j,t} = \theta(M_{j,t} - N^c_{j,t}) \geq 0, \quad \frac{1}{\sigma}\left[\frac{\alpha L_j}{\theta(M_{j,t} + \rho M_{k,t})} + \frac{\alpha L_k}{\theta(M_{j,t} + M_{k,t}/\rho)}\right] \leq f$$

14.3.3 Law of Motion

With δ as the exogenous probability of a variety becoming obsolete, the dynamic equation for the measure of firms becomes

$$N^c_{j,t+1} = \delta(N^c_{j,t} + N^m_{j,t}) = \delta(N^c_{j,t} + \theta(M_{j,t} - N^c_{j,t}))$$

We will work with a normalized measure of varieties

$$n_{j,t} := \frac{\theta \sigma f N^c_{j,t}}{\alpha(L_1 + L_2)}, \quad i_{j,t} := \frac{\theta \sigma f N^m_{j,t}}{\alpha(L_1 + L_2)}, \quad m_{j,t} := \frac{\theta \sigma f M_{j,t}}{\alpha(L_1 + L_2)} = n_{j,t} + \frac{i_{j,t}}{\theta}$$

We also use $s_j := \frac{L_j}{L_1 + L_2}$ to be the share of labor employed in country j.

We can use these definitions and the preceding expressions to obtain a law of motion for $n_t := (n_{1,t}, n_{2,t})$.

In particular, given an initial condition, $n_0 = (n_{1,0}, n_{2,0}) \in \mathbb{R}^2_+$, the equilibrium trajectory, $\{n_t\}_{t=0}^\infty = \{(n_{1,t}, n_{2,t})\}_{t=0}^\infty$, is obtained by iterating on $n_{t+1} = F(n_t)$ where $F : \mathbb{R}^2_+ \to \mathbb{R}^2_+$ is given by

$$F(n_t) = \begin{cases} (\delta(\theta s_1(\rho) + (1-\theta) n_{1,t}), \delta(\theta s_2(\rho) + (1-\theta) n_{2,t})) & \text{for } n_t \in D_{LL} \\ (\delta n_{1,t}, \delta n_{2,t}) & \text{for } n_t \in D_{HH} \\ (\delta n_{1,t}, \delta(\theta h_2(n_{1,t}) + (1-\theta) n_{2,t})) & \text{for } n_t \in D_{HL} \\ (\delta(\theta h_1(n_{2,t}) + (1-\theta) n_{1,t}, \delta n_{2,t})) & \text{for } n_t \in D_{LH} \end{cases}$$

Here

$$
\begin{aligned}
D_{LL} &:= \{(n_1, n_2) \in \mathbb{R}^2_+ | n_j \leq s_j(\rho)\} \\
D_{HH} &:= \{(n_1, n_2) \in \mathbb{R}^2_+ | n_j \geq h_j(n_k)\} \\
D_{HL} &:= \{(n_1, n_2) \in \mathbb{R}^2_+ | n_1 \geq s_1(\rho) \text{ and } n_2 \leq h_2(n_1)\} \\
D_{LH} &:= \{(n_1, n_2) \in \mathbb{R}^2_+ | n_1 \leq h_1(n_2) \text{ and } n_2 \geq s_2(\rho)\}
\end{aligned}
$$

while

$$
s_1(\rho) = 1 - s_2(\rho) = \min\left\{\frac{s_1 - \rho s_2}{1 - \rho}, 1\right\}
$$

and $h_j(n_k)$ is defined implicitly by the equation

$$
1 = \frac{s_j}{h_j(n_k) + \rho n_k} + \frac{s_k}{h_j(n_k) + n_k/\rho}
$$

Rewriting the equation above gives us a quadratic equation in terms of $h_j(n_k)$.

Since we know $h_j(n_k) > 0$ then we can just solve the quadratic equation and return the positive root.

This gives us

$$
h_j(n_k)^2 + \left((\rho + \frac{1}{\rho})n_k - s_j - s_k\right) h_j(n_k) + (n_k^2 - \frac{s_j n_k}{\rho} - s_k n_k \rho) = 0
$$

14.4 Simulation

Let's try simulating some of these trajectories.

We will focus in particular on whether or not innovation cycles synchronize across the two countries.

As we will see, this depends on initial conditions.

For some parameterizations, synchronization will occur for "most" initial conditions, while for others synchronization will be rare.

The computational burden of testing synchronization across many initial conditions is not trivial.

In order to make our code fast, we will use just in time compiled functions that will get called and handled by our class.

These are the `@jit` statements that you see below (review this lecture if you don't recall how to use JIT compilation).

Here's the main body of code

```
@jit(nopython=True)
def _hj(j, nk, s1, s2, θ, δ, ρ):
    """
    If we expand the implicit function for h_j(n_k) then we find that
    it is quadratic. We know that h_j(n_k) > 0 so we can get its
    value by using the quadratic form
    """
    # Find out who's h we are evaluating
    if j == 1:
        sj = s1
        sk = s2
    else:
        sj = s2
```

(continues on next page)

(continued from previous page)

```
        sk = s1

    # Coefficients on the quadratic a x^2 + b x + c = 0
    a = 1.0
    b = ((ρ + 1 / ρ) * nk - sj - sk)
    c = (nk * nk - (sj * nk) / ρ - sk * ρ * nk)

    # Positive solution of quadratic form
    root = (-b + np.sqrt(b * b - 4 * a * c)) / (2 * a)

    return root

@jit(nopython=True)
def DLL(n1, n2, s1_ρ, s2_ρ, s1, s2, θ, δ, ρ):
    "Determine whether (n1, n2) is in the set DLL"
    return (n1 <= s1_ρ) and (n2 <= s2_ρ)

@jit(nopython=True)
def DHH(n1, n2, s1_ρ, s2_ρ, s1, s2, θ, δ, ρ):
    "Determine whether (n1, n2) is in the set DHH"
    return (n1 >= _hj(1, n2, s1, s2, θ, δ, ρ)) and \
           (n2 >= _hj(2, n1, s1, s2, θ, δ, ρ))

@jit(nopython=True)
def DHL(n1, n2, s1_ρ, s2_ρ, s1, s2, θ, δ, ρ):
    "Determine whether (n1, n2) is in the set DHL"
    return (n1 >= s1_ρ) and (n2 <= _hj(2, n1, s1, s2, θ, δ, ρ))

@jit(nopython=True)
def DLH(n1, n2, s1_ρ, s2_ρ, s1, s2, θ, δ, ρ):
    "Determine whether (n1, n2) is in the set DLH"
    return (n1 <= _hj(1, n2, s1, s2, θ, δ, ρ)) and (n2 >= s2_ρ)

@jit(nopython=True)
def one_step(n1, n2, s1_ρ, s2_ρ, s1, s2, θ, δ, ρ):
    """
    Takes a current value for (n_{1, t}, n_{2, t}) and returns the
    values (n_{1, t+1}, n_{2, t+1}) according to the law of motion.
    """
    # Depending on where we are, evaluate the right branch
    if DLL(n1, n2, s1_ρ, s2_ρ, s1, s2, θ, δ, ρ):
        n1_tp1 = δ * (θ * s1_ρ + (1 - θ) * n1)
        n2_tp1 = δ * (θ * s2_ρ + (1 - θ) * n2)
    elif DHH(n1, n2, s1_ρ, s2_ρ, s1, s2, θ, δ, ρ):
        n1_tp1 = δ * n1
        n2_tp1 = δ * n2
    elif DHL(n1, n2, s1_ρ, s2_ρ, s1, s2, θ, δ, ρ):
        n1_tp1 = δ * n1
        n2_tp1 = δ * (θ * _hj(2, n1, s1, s2, θ, δ, ρ) + (1 - θ) * n2)
    elif DLH(n1, n2, s1_ρ, s2_ρ, s1, s2, θ, δ, ρ):
        n1_tp1 = δ * (θ * _hj(1, n2, s1, s2, θ, δ, ρ) + (1 - θ) * n1)
        n2_tp1 = δ * n2

    return n1_tp1, n2_tp1

@jit(nopython=True)
```

(continues on next page)

(continued from previous page)

```
def n_generator(n1_0, n2_0, s1_ρ, s2_ρ, s1, s2, θ, δ, ρ):
    """
    Given an initial condition, continues to yield new values of
    n1 and n2
    """
    n1_t, n2_t = n1_0, n2_0
    while True:
        n1_tp1, n2_tp1 = one_step(n1_t, n2_t, s1_ρ, s2_ρ, s1, s2, θ, δ, ρ)
        yield (n1_tp1, n2_tp1)
        n1_t, n2_t = n1_tp1, n2_tp1

@jit(nopython=True)
def _pers_till_sync(n1_0, n2_0, s1_ρ, s2_ρ, s1, s2, θ, δ, ρ, maxiter, npers):
    """
    Takes initial values and iterates forward to see whether
    the histories eventually end up in sync.

    If countries are symmetric then as soon as the two countries have the
    same measure of firms then they will be synchronized -- However, if
    they are not symmetric then it is possible they have the same measure
    of firms but are not yet synchronized. To address this, we check whether
    firms stay synchronized for `npers` periods with Euclidean norm

    Parameters
    ----------
    n1_0 : scalar(Float)
        Initial normalized measure of firms in country one
    n2_0 : scalar(Float)
        Initial normalized measure of firms in country two
    maxiter : scalar(Int)
        Maximum number of periods to simulate
    npers : scalar(Int)
        Number of periods we would like the countries to have the
        same measure for

    Returns
    -------
    synchronized : scalar(Bool)
        Did the two economies end up synchronized
    pers_2_sync : scalar(Int)
        The number of periods required until they synchronized
    """
    # Initialize the status of synchronization
    synchronized = False
    pers_2_sync = maxiter
    iters = 0

    # Initialize generator
    n_gen = n_generator(n1_0, n2_0, s1_ρ, s2_ρ, s1, s2, θ, δ, ρ)

    # Will use a counter to determine how many times in a row
    # the firm measures are the same
    nsync = 0

    while (not synchronized) and (iters < maxiter):
        # Increment the number of iterations and get next values
```

(continues on next page)

(continued from previous page)

```
        iters += 1
        n1_t, n2_t = next(n_gen)

        # Check whether same in this period
        if abs(n1_t - n2_t) < 1e-8:
            nsync += 1
        # If not, then reset the nsync counter
        else:
            nsync = 0

        # If we have been in sync for npers then stop and countries
        # became synchronized nsync periods ago
        if nsync > npers:
            synchronized = True
            pers_2_sync = iters - nsync

    return synchronized, pers_2_sync

@jit(nopython=True)
def _create_attraction_basis(s1_ρ, s2_ρ, s1, s2, θ, δ, ρ,
        maxiter, npers, npts):
    # Create unit range with npts
    synchronized, pers_2_sync = False, 0
    unit_range = np.linspace(0.0, 1.0, npts)

    # Allocate space to store time to sync
    time_2_sync = np.empty((npts, npts))
    # Iterate over initial conditions
    for (i, n1_0) in enumerate(unit_range):
        for (j, n2_0) in enumerate(unit_range):
            synchronized, pers_2_sync = _pers_till_sync(n1_0, n2_0, s1_ρ,
                                                        s2_ρ, s1, s2, θ, δ,
                                                        ρ, maxiter, npers)
            time_2_sync[i, j] = pers_2_sync

    return time_2_sync


# == Now we define a class for the model == #

class MSGSync:
    """
    The paper "Globalization and Synchronization of Innovation Cycles" presents
    a two-country model with endogenous innovation cycles. Combines elements
    from Deneckere Judd (1985) and Helpman Krugman (1985) to allow for a
    model with trade that has firms who can introduce new varieties into
    the economy.

    We focus on being able to determine whether the two countries eventually
    synchronize their innovation cycles. To do this, we only need a few
    of the many parameters. In particular, we need the parameters listed
    below

    Parameters
    ----------
    s1 : scalar(Float)
```

(continues on next page)

(continued from previous page)

```
        Amount of total labor in country 1 relative to total worldwide labor
    θ : scalar(Float)
        A measure of how much more of the competitive variety is used in
        production of final goods
    δ : scalar(Float)
        Percentage of firms that are not exogenously destroyed every period
    ρ : scalar(Float)
        Measure of how expensive it is to trade between countries
    """
    def __init__(self, s1=0.5, θ=2.5, δ=0.7, ρ=0.2):
        # Store model parameters
        self.s1, self.θ, self.δ, self.ρ = s1, θ, δ, ρ

        # Store other cutoffs and parameters we use
        self.s2 = 1 - s1
        self.s1_ρ = self._calc_s1_ρ()
        self.s2_ρ = 1 - self.s1_ρ

    def _unpack_params(self):
        return self.s1, self.s2, self.θ, self.δ, self.ρ

    def _calc_s1_ρ(self):
        # Unpack params
        s1, s2, θ, δ, ρ = self._unpack_params()

        # s_1(ρ) = min(val, 1)
        val = (s1 - ρ * s2) / (1 - ρ)
        return min(val, 1)

    def simulate_n(self, n1_0, n2_0, T):
        """
        Simulates the values of (n1, n2) for T periods

        Parameters
        ----------
        n1_0 : scalar(Float)
            Initial normalized measure of firms in country one
        n2_0 : scalar(Float)
            Initial normalized measure of firms in country two
        T : scalar(Int)
            Number of periods to simulate

        Returns
        -------
        n1 : Array(Float64, ndim=1)
            A history of normalized measures of firms in country one
        n2 : Array(Float64, ndim=1)
            A history of normalized measures of firms in country two
        """
        # Unpack parameters
        s1, s2, θ, δ, ρ = self._unpack_params()
        s1_ρ, s2_ρ = self.s1_ρ, self.s2_ρ

        # Allocate space
        n1 = np.empty(T)
        n2 = np.empty(T)
```

(continues on next page)

(continued from previous page)

```
        # Create the generator
        n1[0], n2[0] = n1_0, n2_0
        n_gen = n_generator(n1_0, n2_0, s1_ρ, s2_ρ, s1, s2, θ, δ, ρ)

        # Simulate for T periods
        for t in range(1, T):
            # Get next values
            n1_tp1, n2_tp1 = next(n_gen)

            # Store in arrays
            n1[t] = n1_tp1
            n2[t] = n2_tp1

        return n1, n2

    def pers_till_sync(self, n1_0, n2_0, maxiter=500, npers=3):
        """
        Takes initial values and iterates forward to see whether
        the histories eventually end up in sync.

        If countries are symmetric then as soon as the two countries have the
        same measure of firms then they will be synchronized -- However, if
        they are not symmetric then it is possible they have the same measure
        of firms but are not yet synchronized. To address this, we check whether
        firms stay synchronized for `npers` periods with Euclidean norm

        Parameters
        ----------
        n1_0 : scalar(Float)
            Initial normalized measure of firms in country one
        n2_0 : scalar(Float)
            Initial normalized measure of firms in country two
        maxiter : scalar(Int)
            Maximum number of periods to simulate
        npers : scalar(Int)
            Number of periods we would like the countries to have the
            same measure for

        Returns
        -------
        synchronized : scalar(Bool)
            Did the two economies end up synchronized
        pers_2_sync : scalar(Int)
            The number of periods required until they synchronized
        """
        # Unpack parameters
        s1, s2, θ, δ, ρ = self._unpack_params()
        s1_ρ, s2_ρ = self.s1_ρ, self.s2_ρ

        return _pers_till_sync(n1_0, n2_0, s1_ρ, s2_ρ,
                               s1, s2, θ, δ, ρ, maxiter, npers)

    def create_attraction_basis(self, maxiter=250, npers=3, npts=50):
        """
        Creates an attraction basis for values of n on [0, 1] X [0, 1]
```

(continues on next page)

(continued from previous page)

```
        with npts in each dimension
        """
        # Unpack parameters
        s1, s2, θ, δ, ρ = self._unpack_params()
        s1_ρ, s2_ρ = self.s1_ρ, self.s2_ρ

        ab = _create_attraction_basis(s1_ρ, s2_ρ, s1, s2, θ, δ,
                                      ρ, maxiter, npers, npts)

        return ab
```

14.4.1 Time Series of Firm Measures

We write a short function below that exploits the preceding code and plots two time series.

Each time series gives the dynamics for the two countries.

The time series share parameters but differ in their initial condition.

Here's the function

```
def plot_timeseries(n1_0, n2_0, s1=0.5, θ=2.5,
                    δ=0.7, ρ=0.2, ax=None, title=''):
    """
    Plot a single time series with initial conditions
    """
    if ax is None:
        fig, ax = plt.subplots()

    # Create the MSG Model and simulate with initial conditions
    model = MSGSync(s1, θ, δ, ρ)
    n1, n2 = model.simulate_n(n1_0, n2_0, 25)

    ax.plot(np.arange(25), n1, label="$n_1$", lw=2)
    ax.plot(np.arange(25), n2, label="$n_2$", lw=2)

    ax.legend()
    ax.set(title=title, ylim=(0.15, 0.8))

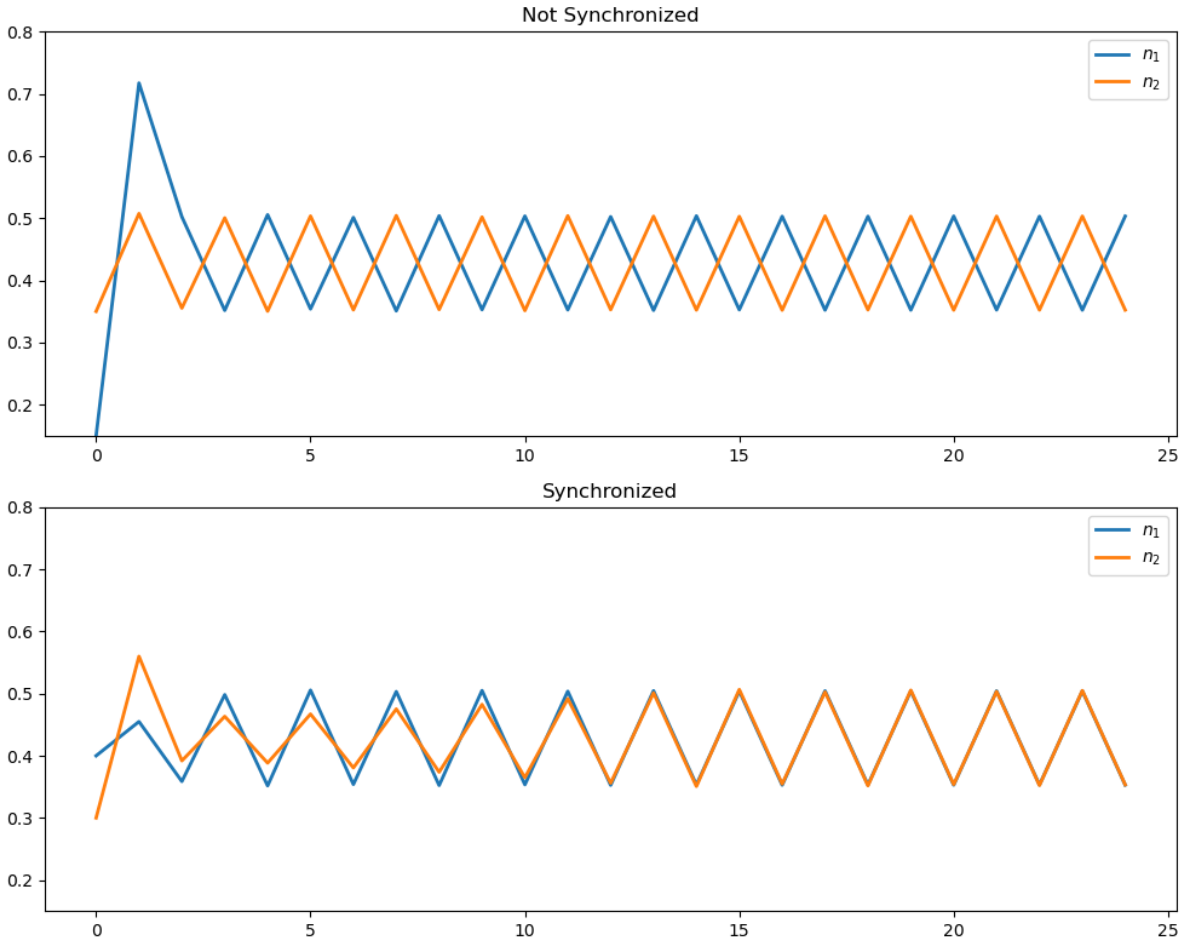
    return ax


# Create figure
fig, ax = plt.subplots(2, 1, figsize=(10, 8))

plot_timeseries(0.15, 0.35, ax=ax[0], title='Not Synchronized')
plot_timeseries(0.4, 0.3, ax=ax[1], title='Synchronized')

fig.tight_layout()

plt.show()
```



In the first case, innovation in the two countries does not synchronize.

In the second case, different initial conditions are chosen, and the cycles become synchronized.

14.4.2 Basin of Attraction

Next, let's study the initial conditions that lead to synchronized cycles more systematically.

We generate time series from a large collection of different initial conditions and mark those conditions with different colors according to whether synchronization occurs or not.

The next display shows exactly this for four different parameterizations (one for each subfigure).

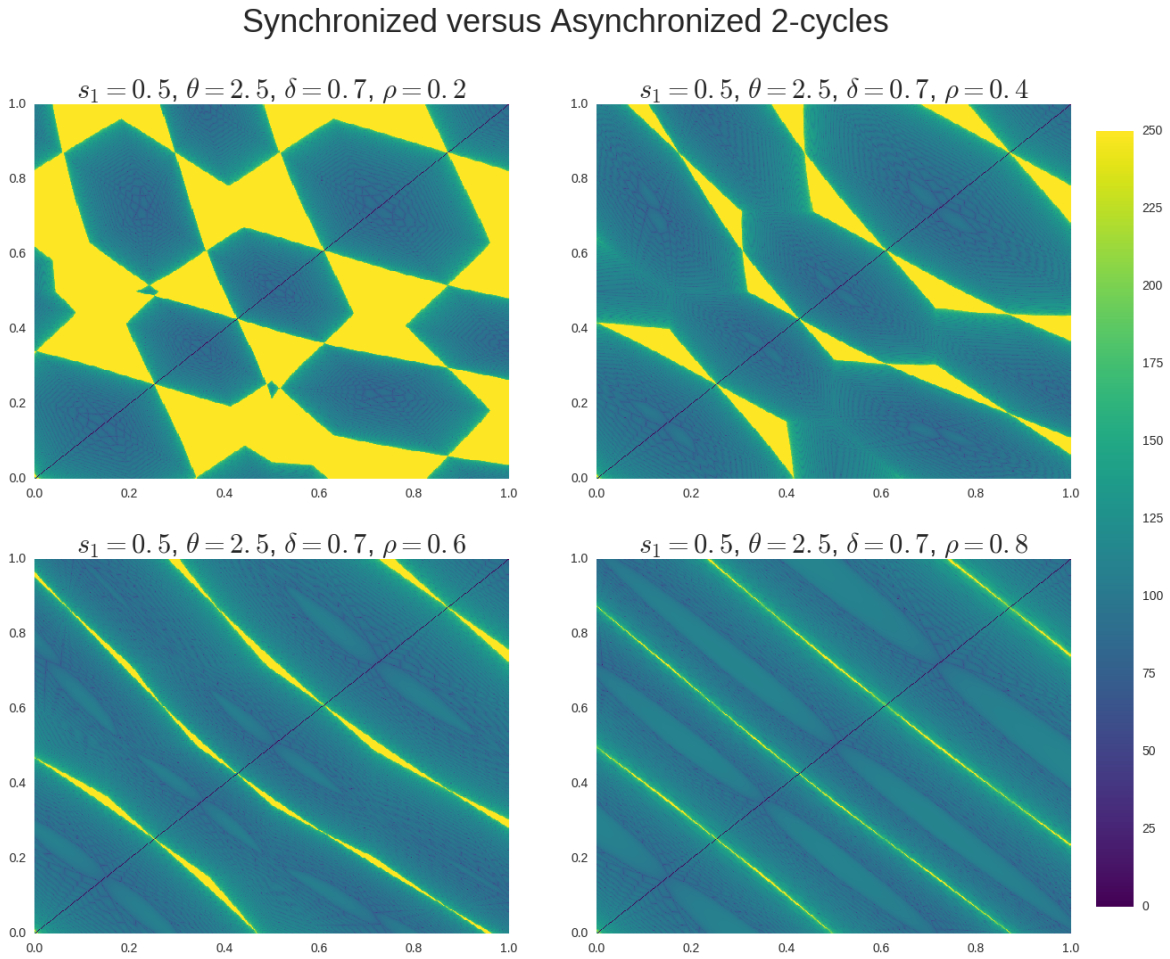
Dark colors indicate synchronization, while light colors indicate failure to synchronize.

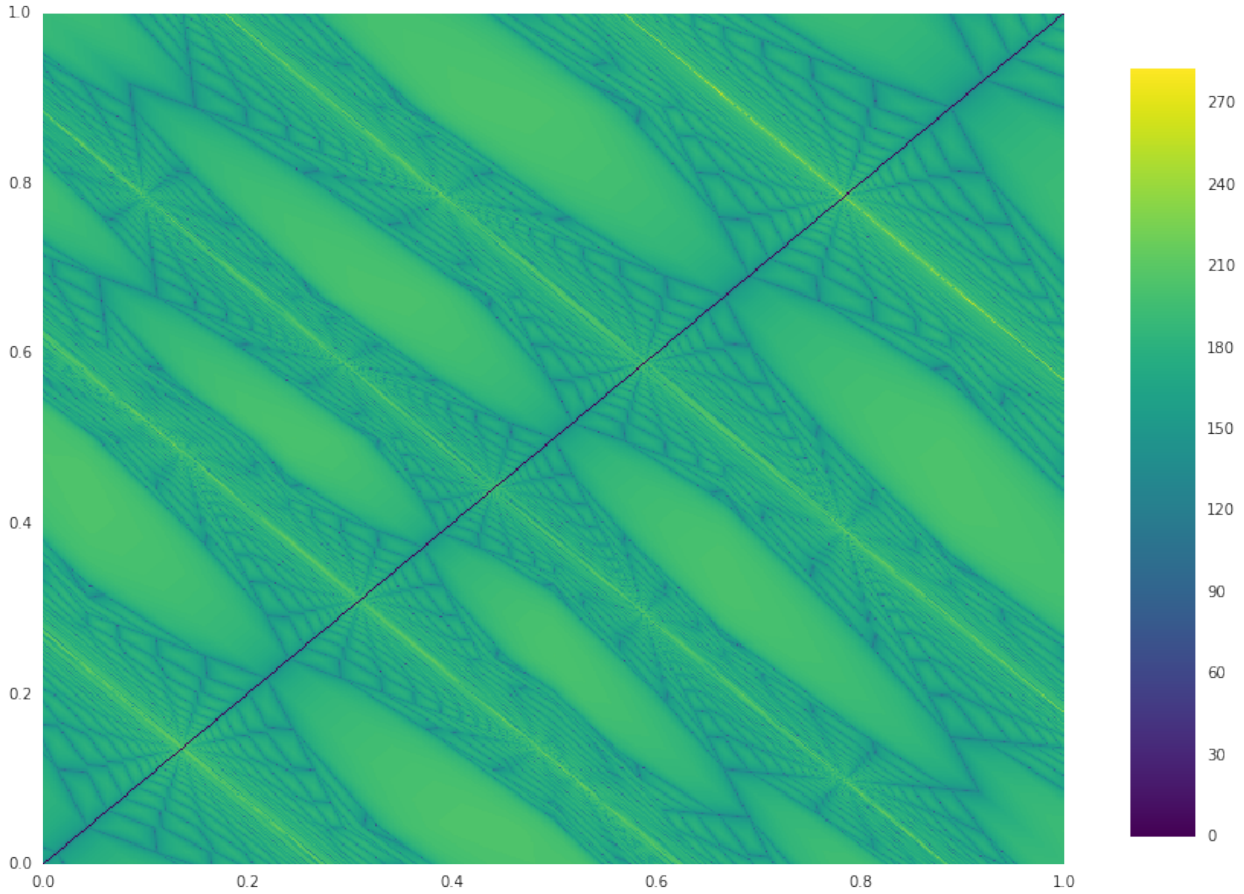
As you can see, larger values of ρ translate to more synchronization.

You are asked to replicate this figure in the exercises.

In the solution to the exercises, you'll also find a figure with sliders, allowing you to experiment with different parameters.

Here's one snapshot from the interactive figure


Synchronized versus Asynchronized 2-cycles
$s_1 = 0.5,\ \theta = 2.5,\ \delta = 0.7,\ \rho = 0.2$
$s_1 = 0.5,\ \theta = 2.5,\ \delta = 0.7,\ \rho = 0.4$
$s_1 = 0.5,\ \theta = 2.5,\ \delta = 0.7,\ \rho = 0.6$
$s_1 = 0.5,\ \theta = 2.5,\ \delta = 0.7,\ \rho = 0.8$
0.0
0.2
0.4
0.6
0.8
1.0
0
25
50
75
100
125
150
175
200
225
250


1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
270
240
210
180
150
120
90
60
30
0

14.5 Exercises

Exercise 14.5.1

Replicate the figure *shown above* by coloring initial conditions according to whether or not synchronization occurs from those conditions.

Solution to Exercise 14.5.1

```
def plot_attraction_basis(s1=0.5, θ=2.5, δ=0.7, ρ=0.2, npts=250, ax=None):
    if ax is None:
        fig, ax = plt.subplots()

    # Create attraction basis
    unitrange = np.linspace(0, 1, npts)
    model = MSGSync(s1, θ, δ, ρ)
    ab = model.create_attraction_basis(npts=npts)
    cf = ax.pcolormesh(unitrange, unitrange, ab, cmap="viridis")

    return ab, cf


fig = plt.figure(figsize=(14, 12))

# Left - Bottom - Width - Height
ax0 = fig.add_axes((0.05, 0.475, 0.38, 0.35), label="axes0")
ax1 = fig.add_axes((0.5, 0.475, 0.38, 0.35), label="axes1")
ax2 = fig.add_axes((0.05, 0.05, 0.38, 0.35), label="axes2")
ax3 = fig.add_axes((0.5, 0.05, 0.38, 0.35), label="axes3")

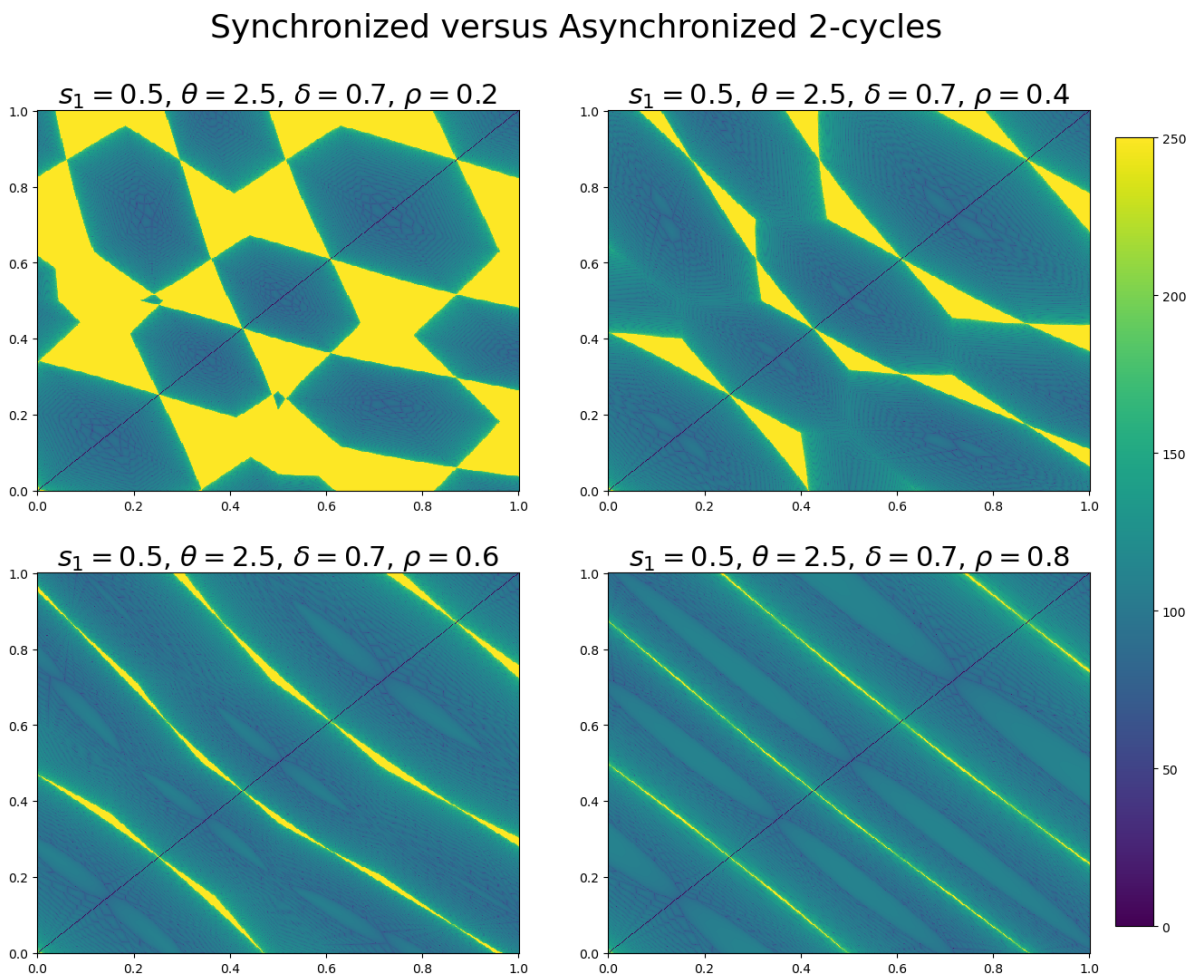
params = [[0.5, 2.5, 0.7, 0.2],
          [0.5, 2.5, 0.7, 0.4],
          [0.5, 2.5, 0.7, 0.6],
          [0.5, 2.5, 0.7, 0.8]]

ab0, cf0 = plot_attraction_basis(*params[0], npts=500, ax=ax0)
ab1, cf1 = plot_attraction_basis(*params[1], npts=500, ax=ax1)
ab2, cf2 = plot_attraction_basis(*params[2], npts=500, ax=ax2)
ab3, cf3 = plot_attraction_basis(*params[3], npts=500, ax=ax3)

cbar_ax = fig.add_axes([0.9, 0.075, 0.03, 0.725])
plt.colorbar(cf0, cax=cbar_ax)

ax0.set_title(r"$s_1=0.5$, $\theta=2.5$, $\delta=0.7$, $\rho=0.2$",
              fontsize=22)
ax1.set_title(r"$s_1=0.5$, $\theta=2.5$, $\delta=0.7$, $\rho=0.4$",
              fontsize=22)
ax2.set_title(r"$s_1=0.5$, $\theta=2.5$, $\delta=0.7$, $\rho=0.6$",
              fontsize=22)
ax3.set_title(r"$s_1=0.5$, $\theta=2.5$, $\delta=0.7$, $\rho=0.8$",
              fontsize=22)

fig.suptitle("Synchronized versus Asynchronized 2-cycles",
             x=0.475, y=0.915, size=26)
plt.show()
```



Additionally, instead of just seeing 4 plots at once, we might want to manually be able to change ρ and see how it affects the plot in real-time. Below we use an interactive plot to do this.

Note, interactive plotting requires the ipywidgets module to be installed and enabled.

Note: This interactive plot is disabled on this static webpage. In order to use this, we recommend to run this notebook locally.

```
def interact_attraction_basis(ρ=0.2, maxiter=250, npts=250):
    # Create the figure and axis that we will plot on
    fig, ax = plt.subplots(figsize=(12, 10))
    # Create model and attraction basis
    s1, θ, δ = 0.5, 2.5, 0.75
    model = MSGSync(s1, θ, δ, ρ)
    ab = model.create_attraction_basis(maxiter=maxiter, npts=npts)
    # Color map with colormesh
    unitrange = np.linspace(0, 1, npts)
    cf = ax.pcolormesh(unitrange, unitrange, ab, cmap="viridis")
    cbar_ax = fig.add_axes([0.95, 0.15, 0.05, 0.7])
    plt.colorbar(cf, cax=cbar_ax)
    plt.show()
    return None
```

```
fig = interact(interact_attraction_basis,
               ρ=(0.0, 1.0, 0.05),
               maxiter=(50, 5000, 50),
               npts=(25, 750, 25))
```

CHAPTER

FIFTEEN

COASE'S THEORY OF THE FIRM

15.1 Overview

In 1937, Ronald Coase wrote a brilliant essay on the nature of the firm [Coase, 1937].

Coase was writing at a time when the Soviet Union was rising to become a significant industrial power.

At the same time, many free-market economies were afflicted by a severe and painful depression.

This contrast led to an intensive debate on the relative merits of decentralized, price-based allocation versus top-down planning.

In the midst of this debate, Coase made an important observation: even in free-market economies, a great deal of top-down planning does in fact take place.

This is because *firms* form an integral part of free-market economies and, within firms, allocation is by planning.

In other words, free-market economies blend both planning (within firms) and decentralized production coordinated by prices.

The question Coase asked is this: if prices and free markets are so efficient, then why do firms even exist?

Couldn't the associated within-firm planning be done more efficiently by the market?

We'll use the following imports:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import fminbound
```

15.1.1 Why Firms Exist

On top of asking a deep and fascinating question, Coase also supplied an illuminating answer: firms exist because of transaction costs.

Here's one example of a transaction cost:

Suppose agent A is considering setting up a small business and needs a web developer to construct and help run an online store.

She can use the labor of agent B, a web developer, by writing up a freelance contract for these tasks and agreeing on a suitable price.

But contracts like this can be time-consuming and difficult to verify

- How will agent A be able to specify exactly what she wants, to the finest detail, when she herself isn't sure how the business will evolve?

- And what if she isn't familiar with web technology? How can she specify all the relevant details?
- And, if things go badly, will failure to comply with the contract be verifiable in court?

In this situation, perhaps it will be easier to *employ* agent B under a simple labor contract.

The cost of this contract is far smaller because such contracts are simpler and more standard.

The basic agreement in a labor contract is: B will do what A asks him to do for the term of the contract, in return for a given salary.

Making this agreement is much easier than trying to map every task out in advance in a contract that will hold up in a court of law.

So agent A decides to hire agent B and a firm of nontrivial size appears, due to transaction costs.

15.1.2 A Trade-Off

Actually, we haven't yet come to the heart of Coase's investigation.

The issue of why firms exist is a binary question: should firms have positive size or zero size?

A better and more general question is: **what determines the size of firms**?

The answer Coase came up with was that "a firm will tend to expand until the costs of organizing an extra transaction within the firm become equal to the costs of carrying out the same transaction by means of an exchange on the open market…" ([Coase, 1937], p. 395).

But what are these internal and external costs?

In short, Coase envisaged a trade-off between

- transaction costs, which add to the expense of operating *between* firms, and
- diminishing returns to management, which adds to the expense of operating *within* firms

We discussed an example of transaction costs above (contracts).

The other cost, diminishing returns to management, is a catch-all for the idea that big operations are increasingly costly to manage.

For example, you could think of management as a pyramid, so hiring more workers to implement more tasks requires expansion of the pyramid, and hence labor costs grow at a rate more than proportional to the range of tasks.

Diminishing returns to management makes in-house production expensive, favoring small firms.

15.1.3 Summary

Here's a summary of our discussion:

- Firms grow because transaction costs encourage them to take some operations in house.
- But as they get large, in-house operations become costly due to diminishing returns to management.
- The size of firms is determined by balancing these effects, thereby equalizing the marginal costs of each form of operation.

15.1.4 A Quantitative Interpretation

Coases ideas were expressed verbally, without any mathematics.

In fact, his essay is a wonderful example of how far you can get with clear thinking and plain English.

However, plain English is not good for quantitative analysis, so let's bring some mathematical and computation tools to bear.

In doing so we'll add a bit more structure than Coase did, but this price will be worth paying.

Our exposition is based on [Kikuchi *et al.*, 2018].

15.2 The Model

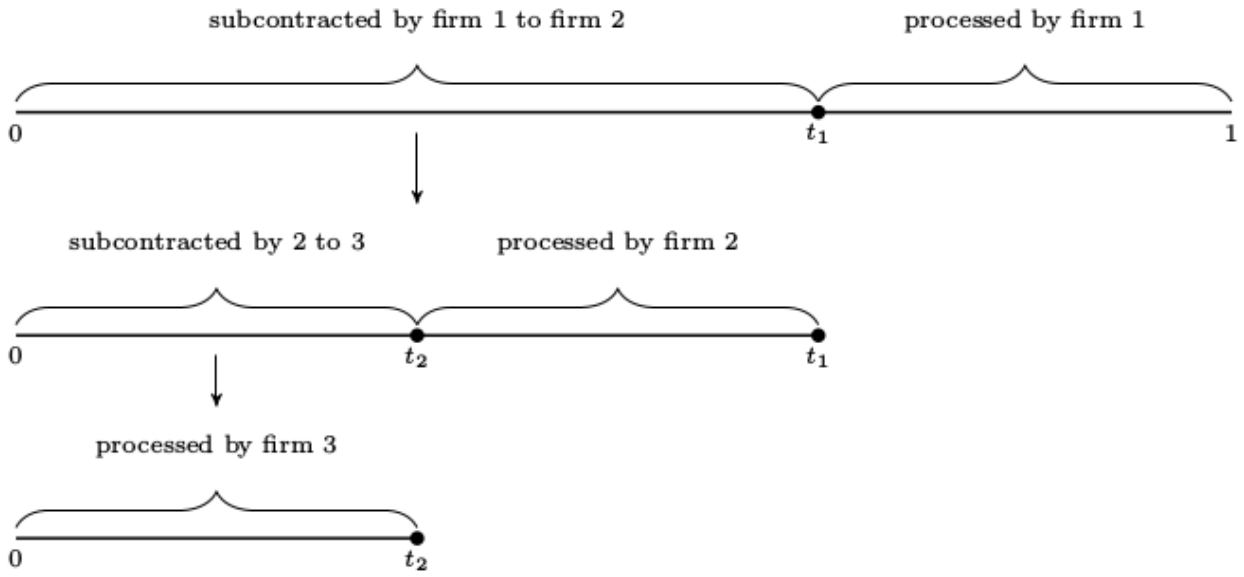
The model we study involves production of a single unit of a final good.

Production requires a linearly ordered chain, requiring sequential completion of a large number of processing stages.

The stages are indexed by $t \in [0, 1]$, with $t = 0$ indicating that no tasks have been undertaken and $t = 1$ indicating that the good is complete.

15.2.1 Subcontracting

The subcontracting scheme by which tasks are allocated across firms is illustrated in the figure below



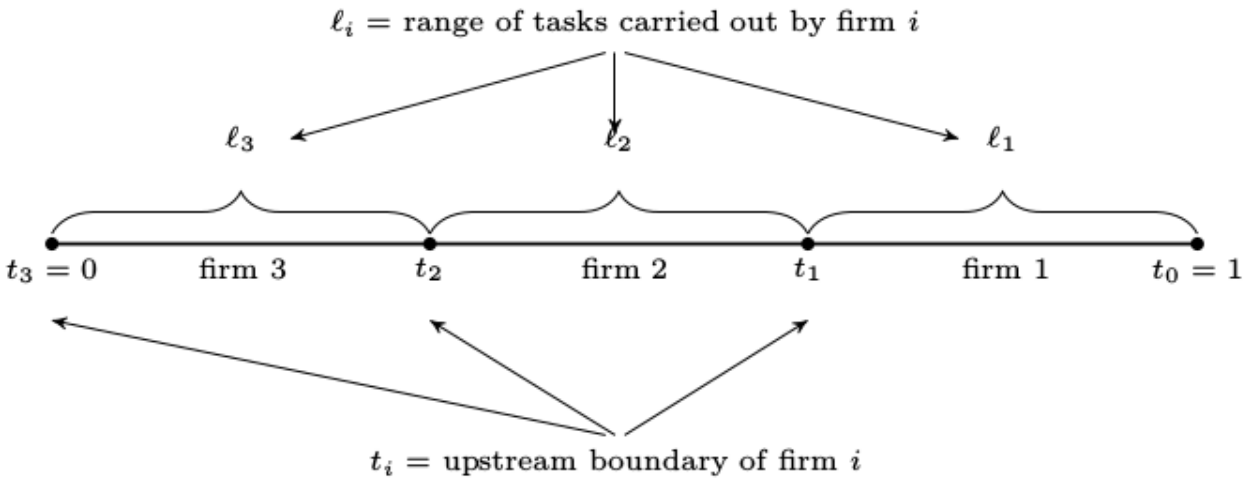
In this example,

- Firm 1 receives a contract to sell one unit of the completed good to a final buyer.
- Firm 1 then forms a contract with firm 2 to purchase the partially completed good at stage t_1, with the intention of implementing the remaining $1 - t_1$ tasks in-house (i.e., processing from stage t_1 to stage 1).
- Firm 2 repeats this procedure, forming a contract with firm 3 to purchase the good at stage t_2.
- Firm 3 decides to complete the chain, selecting $t_3 = 0$.

At this point, production unfolds in the opposite direction (i.e., from upstream to downstream).

- Firm 3 completes processing stages from $t_3 = 0$ up to t_2 and transfers the good to firm 2.
- Firm 2 then processes from t_2 up to t_1 and transfers the good to firm 1,
- Firm 1 processes from t_1 to 1 and delivers the completed good to the final buyer.

The length of the interval of stages (range of tasks) carried out by firm i is denoted by ℓ_i.



Each firm chooses only its *upstream* boundary, treating its downstream boundary as given.

The benefit of this formulation is that it implies a recursive structure for the decision problem for each firm.

In choosing how many processing stages to subcontract, each successive firm faces essentially the same decision problem as the firm above it in the chain, with the only difference being that the decision space is a subinterval of the decision space for the firm above.

We will exploit this recursive structure in our study of equilibrium.

15.2.2 Costs

Recall that we are considering a trade-off between two types of costs.

Let's discuss these costs and how we represent them mathematically.

Diminishing returns to management means rising costs per task when a firm expands the range of productive activities coordinated by its managers.

We represent these ideas by taking the cost of carrying out ℓ tasks in-house to be $c(\ell)$, where c is increasing and strictly convex.

Thus, the average cost per task rises with the range of tasks performed in-house.

We also assume that c is continuously differentiable, with $c(0) = 0$ and $c'(0) > 0$.

Transaction costs are represented as a wedge between the buyer's and seller's prices.

It matters little for us whether the transaction cost is borne by the buyer or the seller.

Here we assume that the cost is borne only by the buyer.

In particular, when two firms agree to a trade at face value v, the buyer's total outlay is δv, where $\delta > 1$.

The seller receives only v, and the difference is paid to agents outside the model.

15.3 Equilibrium

We assume that all firms are *ex-ante* identical and act as price takers.

As price takers, they face a price function p, which is a map from $[0, 1]$ to $\mathbb{R}_+$, with $p(t)$ interpreted as the price of the good at processing stage t.

There is a countable infinity of firms indexed by i and no barriers to entry.

The cost of supplying the initial input (the good processed up to stage zero) is set to zero for simplicity.

Free entry and the infinite fringe of competitors rule out positive profits for incumbents, since any incumbent could be replaced by a member of the competitive fringe filling the same role in the production chain.

Profits are never negative in equilibrium because firms can freely exit.

15.3.1 Informal Definition of Equilibrium

An equilibrium in this setting is an allocation of firms and a price function such that

1. all active firms in the chain make zero profits, including suppliers of raw materials
2. no firm in the production chain has an incentive to deviate, and
3. no inactive firms can enter and extract positive profits

15.3.2 Formal Definition of Equilibrium

Let's make this definition more formal.

(You might like to skip this section on first reading)

An **allocation** of firms is a nonnegative sequence $\{\ell_i\}_{i \in \mathbb{N}}$ such that $\ell_i = 0$ for all sufficiently large i.

Recalling the figures above,

- ℓ_i represents the range of tasks implemented by the i-th firm

As a labeling convention, we assume that firms enter in order, with firm 1 being the furthest downstream.

An allocation $\{\ell_i\}$ is called **feasible** if $\sum_{i \geq 1} \ell_i = 1$.

In a feasible allocation, the entire production process is completed by finitely many firms.

Given a feasible allocation, $\{\ell_i\}$, let $\{t_i\}$ represent the corresponding transaction stages, defined by

$$t_0 = s \quad \text{and} \quad t_i = t_{i-1} - \ell_i \tag{15.1}$$

In particular, t_{i-1} is the downstream boundary of firm i and t_i is its upstream boundary.

As transaction costs are incurred only by the buyer, its profits are

$$\pi_i = p(t_{i-1}) - c(\ell_i) - \delta p(t_i) \tag{15.2}$$

Given a price function p and a feasible allocation $\{\ell_i\}$, let

- $\{t_i\}$ be the corresponding firm boundaries.
- $\{\pi_i\}$ be corresponding profits, as defined in (15.2).

This price-allocation pair is called an **equilibrium** for the production chain if

1. $p(0) = 0$,
2. $\pi_i = 0$ for all i, and
3. $p(s) - c(s-t) - \delta p(t) \leq 0$ for any pair s, t with $0 \leq s \leq t \leq 1$.

The rationale behind these conditions was given in our informal definition of equilibrium above.

15.4 Existence, Uniqueness and Computation of Equilibria

We have defined an equilibrium but does one exist? Is it unique? And, if so, how can we compute it?

15.4.1 A Fixed Point Method

To address these questions, we introduce the operator T mapping a nonnegative function p on $[0, 1]$ to Tp via

$$Tp(s) = \min_{t \leq s} \{c(s-t) + \delta p(t)\} \quad \text{for all} \quad s \in [0, 1]. \tag{15.3}$$

Here and below, the restriction $0 \leq t$ in the minimum is understood.

The operator T is similar to a Bellman operator.

Under this analogy, p corresponds to a value function and δ to a discount factor.

But $\delta > 1$, so T is not a contraction in any obvious metric, and in fact, $T^n p$ diverges for many choices of p.

Nevertheless, there exists a domain on which T is well-behaved: the set of convex increasing continuous functions $p\colon [0, 1] \to \mathbb{R}$ such that $c'(0)s \leq p(s) \leq c(s)$ for all $0 \leq s \leq 1$.

We denote this set of functions by $\mathcal{P}$.

In [Kikuchi *et al.*, 2018] it is shown that the following statements are true:

1. T maps $\mathcal{P}$ into itself.
2. T has a unique fixed point in $\mathcal{P}$, denoted below by p^*.
3. For all $p \in \mathcal{P}$ we have $T^k p \to p^*$ uniformly as $k \to \infty$.

Now consider the choice function

$$t^*(s) := \text{ the solution to } \min_{t \leq s} \{c(s-t) + \delta p^*(t)\} \tag{15.4}$$

By definition, $t^*(s)$ is the cost-minimizing upstream boundary for a firm that is contracted to deliver the good at stage s and faces the price function p^*.

Since p^* lies in $\mathcal{P}$ and since c is strictly convex, it follows that the right-hand side of (15.4) is continuous and strictly convex in t.

Hence the minimizer $t^*(s)$ exists and is uniquely defined.

We can use t^* to construct an equilibrium allocation as follows:

Recall that firm 1 sells the completed good at stage $s = 1$, its optimal upstream boundary is $t^*(1)$.

Hence firm 2's optimal upstream boundary is $t^*(t^*(1))$.

Continuing in this way produces the sequence $\{t_i^*\}$ defined by

$$t_0^* = 1 \quad \text{and} \quad t_i^* = t^*(t_{i-1}) \tag{15.5}$$

The sequence ends when a firm chooses to complete all remaining tasks.

We label this firm (and hence the number of firms in the chain) as

$$n^* := \inf\{i \in \mathbb{N} \,:\, t_i^* = 0\} \tag{15.6}$$

The task allocation corresponding to (15.5) is given by $\ell_i^* := t_{i-1}^* - t_i^*$ for all i.

In [Kikuchi *et al.*, 2018] it is shown that

1. The value n^* in (15.6) is well-defined and finite,
2. the allocation $\{\ell_i^*\}$ is feasible, and
3. the price function p^* and this allocation together forms an equilibrium for the production chain.

While the proofs are too long to repeat here, much of the insight can be obtained by observing that, as a fixed point of T, the equilibrium price function must satisfy

$$p^*(s) = \min_{t \le s} \{c(s-t) + \delta p^*(t)\} \quad \text{for all} \quad s \in [0,1] \tag{15.7}$$

From this equation, it is clear that so profits are zero for all incumbent firms.

15.4.2 Marginal Conditions

We can develop some additional insights on the behavior of firms by examining marginal conditions associated with the equilibrium.

As a first step, let $\ell^*(s) := s - t^*(s)$.

This is the cost-minimizing range of in-house tasks for a firm with downstream boundary s.

In [Kikuchi *et al.*, 2018] it is shown that t^* and ℓ^* are increasing and continuous, while p^* is continuously differentiable at all $s \in (0,1)$ with

$$(p^*)'(s) = c'(\ell^*(s)) \tag{15.8}$$

Equation (15.8) follows from $p^*(s) = \min_{t \le s} \{c(s-t) + \delta p^*(t)\}$ and the envelope theorem for derivatives.

A related equation is the first order condition for $p^*(s) = \min_{t \le s} \{c(s-t) + \delta p^*(t)\}$, the minimization problem for a firm with upstream boundary s, which is

$$\delta (p^*)'(t^*(s)) = c'(s - t^*(s)) \tag{15.9}$$

This condition matches the marginal condition expressed verbally by Coase that we stated above:

> "A firm will tend to expand until the costs of organizing an extra transaction within the firm become equal to the costs of carrying out the same transaction by means of an exchange on the open market…"

Combining (15.8) and (15.9) and evaluating at $s = t_i$, we see that active firms that are adjacent satisfy

$$\delta\, c'(\ell_{i+1}^*) = c'(\ell_i^*) \tag{15.10}$$

In other words, the marginal in-house cost per task at a given firm is equal to that of its upstream partner multiplied by gross transaction cost.

This expression can be thought of as a **Coase–Euler equation,** which determines inter-firm efficiency by indicating how two costly forms of coordination (markets and management) are jointly minimized in equilibrium.

15.5 Implementation

For most specifications of primitives, there is no closed-form solution for the equilibrium as far as we are aware.

However, we know that we can compute the equilibrium corresponding to a given transaction cost parameter δ and a cost function c by applying the results stated above.

In particular, we can

1. fix initial condition $p \in \mathcal{P}$,
2. iterate with T until $T^n p$ has converged to p^*, and
3. recover firm choices via the choice function (15.3)

At each iterate, we will use continuous piecewise linear interpolation of functions.

To begin, here's a class to store primitives and a grid:

```
class ProductionChain:

    def __init__(self,
                 n=1000,
                 delta=1.05,
                 c=lambda t: np.exp(10 * t) - 1):

        self.n, self.delta, self.c = n, delta, c
        self.grid = np.linspace(1e-04, 1, n)
```

Now let's implement and iterate with T until convergence.

Recalling that our initial condition must lie in $\mathcal{P}$, we set $p_0 = c$

```
def compute_prices(pc, tol=1e-5, max_iter=5000):
    """
    Compute prices by iterating with T

        * pc is an instance of ProductionChain
        * The initial condition is p = c

    """
    delta, c, n, grid = pc.delta, pc.c, pc.n, pc.grid
    p = c(grid)  # Initial condition is c(s), as an array
    new_p = np.empty_like(p)
    error = tol + 1
    i = 0

    while error > tol and i < max_iter:
        for j, s in enumerate(grid):
            Tp = lambda t: delta * np.interp(t, grid, p) + c(s - t)
            new_p[j] = Tp(fminbound(Tp, 0, s))
        error = np.max(np.abs(p - new_p))
        p = new_p
        i = i + 1

    if i < max_iter:
        print(f"Iteration converged in {i} steps")
    else:
        print(f"Warning: iteration hit upper bound {max_iter}")
```

(continues on next page)

(continued from previous page)

```
    p_func = lambda x: np.interp(x, grid, p)
    return p_func
```

The next function computes optimal choice of upstream boundary and range of task implemented for a firm face price function p_function and with downstream boundary s.

```
def optimal_choices(pc, p_function, s):
    """
    Takes p_func as the true function, minimizes on [0,s]

    Returns optimal upstream boundary t_star and optimal size of
    firm ell_star

    In fact, the algorithm minimizes on [-1,s] and then takes the
    max of the minimizer and zero. This results in better results
    close to zero

    """
    delta, c = pc.delta, pc.c
    f = lambda t: delta * p_function(t) + c(s - t)
    t_star = max(fminbound(f, -1, s), 0)
    ell_star = s - t_star
    return t_star, ell_star
```

The allocation of firms can be computed by recursively stepping through firms' choices of their respective upstream boundary, treating the previous firm's upstream boundary as their own downstream boundary.

In doing so, we start with firm 1, who has downstream boundary $s = 1$.

```
def compute_stages(pc, p_function):
    s = 1.0
    transaction_stages = [s]
    while s > 0:
        s, ell = optimal_choices(pc, p_function, s)
        transaction_stages.append(s)
    return np.array(transaction_stages)
```

Let's try this at the default parameters.

The next figure shows the equilibrium price function, as well as the boundaries of firms as vertical lines

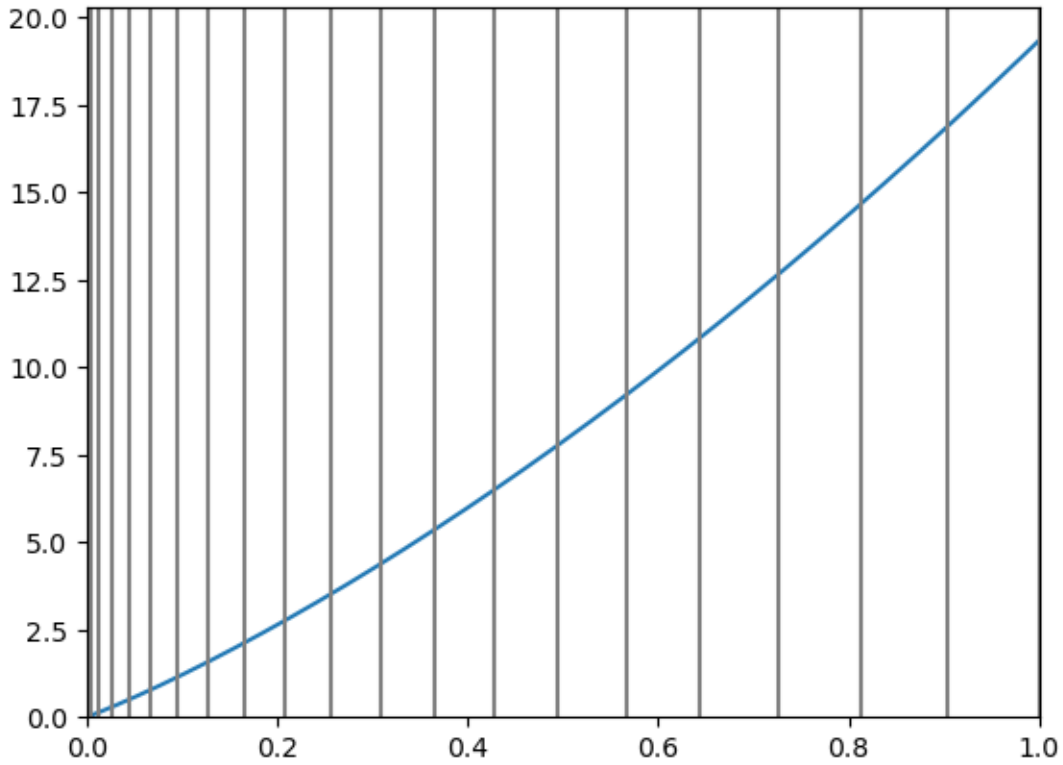
```
pc = ProductionChain()
p_star = compute_prices(pc)

transaction_stages = compute_stages(pc, p_star)

fig, ax = plt.subplots()

ax.plot(pc.grid, p_star(pc.grid))
ax.set_xlim(0.0, 1.0)
ax.set_ylim(0.0)
for s in transaction_stages:
    ax.axvline(x=s, c="0.5")
plt.show()
```

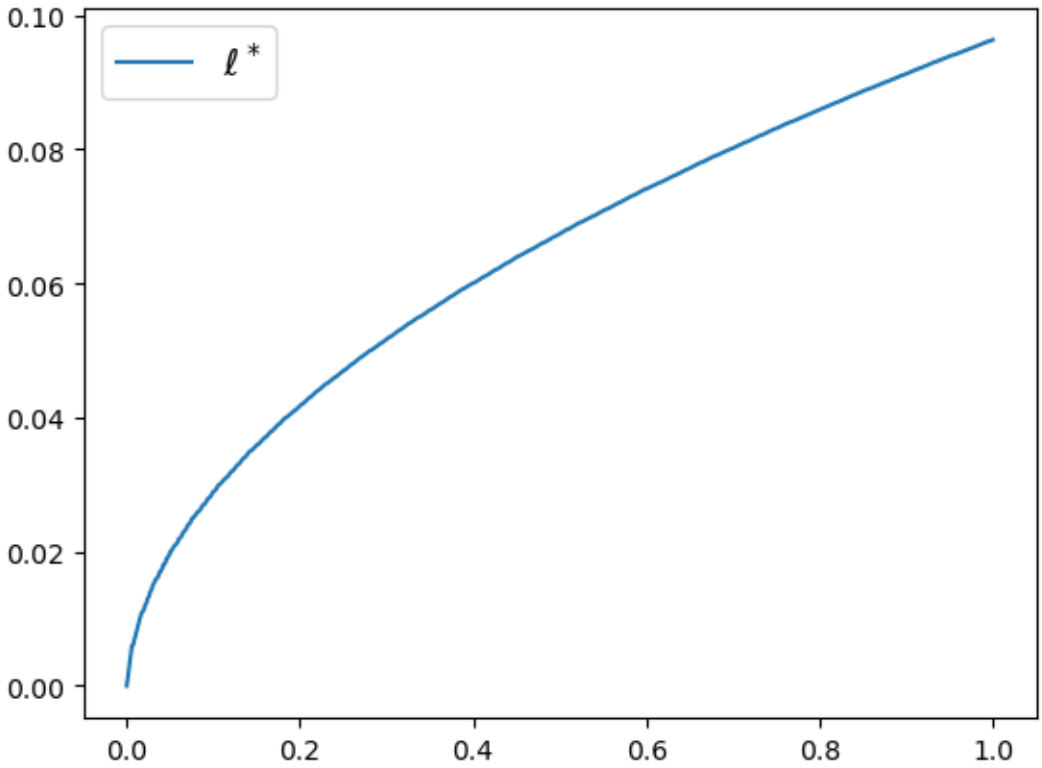
```
Iteration converged in 2 steps
```



Here's the function ℓ^*, which shows how large a firm with downstream boundary s chooses to be

```
ell_star = np.empty(pc.n)
for i, s in enumerate(pc.grid):
    t, e = optimal_choices(pc, p_star, s)
    ell_star[i] = e

fig, ax = plt.subplots()
ax.plot(pc.grid, ell_star, label=r"$\ell^*$")
ax.legend(fontsize=14)
plt.show()
```



Note that downstream firms choose to be larger, a point we return to below.

15.6 Exercises

Exercise 15.6.1

The number of firms is endogenously determined by the primitives.

What do you think will happen in terms of the number of firms as δ increases? Why?

Check your intuition by computing the number of firms at delta in (1.01, 1.05, 1.1).

Solution to Exercise 15.6.1

Here is one solution

```
for delta in (1.01, 1.05, 1.1):

    pc = ProductionChain(delta=delta)
    p_star = compute_prices(pc)
    transaction_stages = compute_stages(pc, p_star)
    num_firms = len(transaction_stages)
    print(f"When delta={delta} there are {num_firms} firms")
```

```
Iteration converged in 2 steps
When delta=1.01 there are 64 firms
```

```
Iteration converged in 2 steps
When delta=1.05 there are 41 firms
```

```
Iteration converged in 2 steps
When delta=1.1 there are 35 firms
```

Exercise 15.6.2

The **value added** of firm i is $v_i := p^*(t_{i-1}) - p^*(t_i)$.

One of the interesting predictions of the model is that value added is increasing with downstreamness, as are several other measures of firm size.

Can you give any intution?

Try to verify this phenomenon (value added increasing with downstreamness) using the code above.

Solution to Exercise 15.6.2

Firm size increases with downstreamness because p^*, the equilibrium price function, is increasing and strictly convex.

This means that, for a given producer, the marginal cost of the input purchased from the producer just upstream from itself in the chain increases as we go further downstream.

Hence downstream firms choose to do more in house than upstream firms — and are therefore larger.

The equilibrium price function is strictly convex due to both transaction costs and diminishing returns to management.

One way to put this is that firms are prevented from completely mitigating the costs associated with diminishing returns to management — which induce convexity — by transaction costs. This is because transaction costs force firms to have nontrivial size.

Here's one way to compute and graph value added across firms

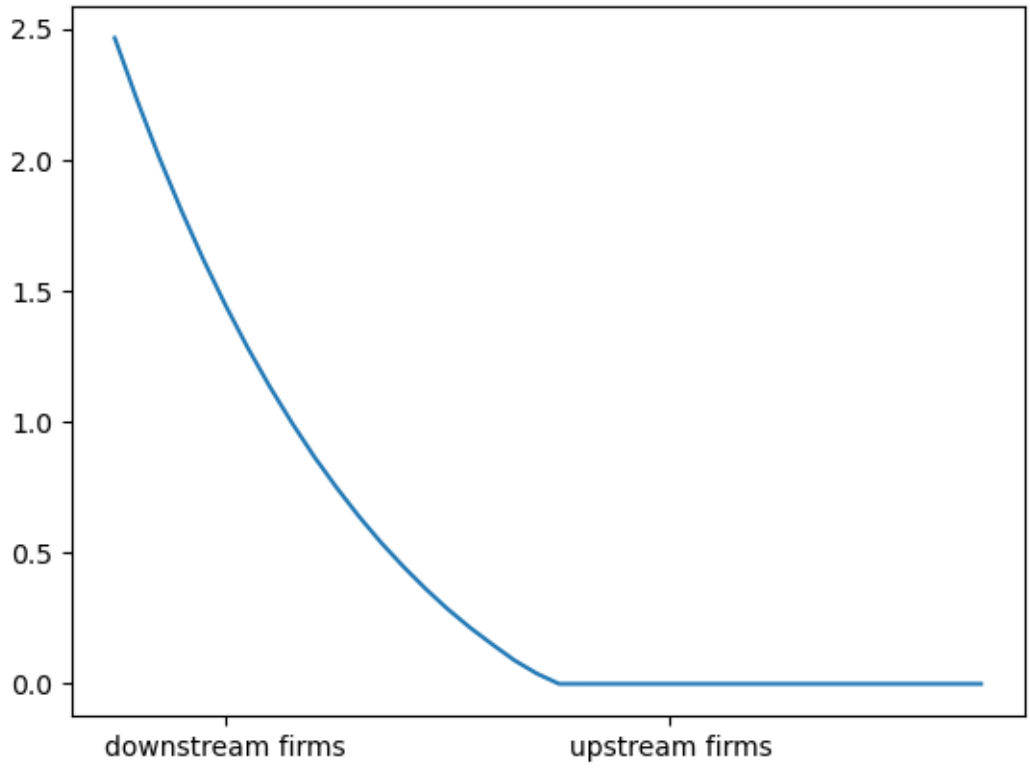
```
pc = ProductionChain()
p_star = compute_prices(pc)
stages = compute_stages(pc, p_star)

va = []

for i in range(len(stages) - 1):
    va.append(p_star(stages[i]) - p_star(stages[i+1]))

fig, ax = plt.subplots()
ax.plot(va, label="value added by firm")
ax.set_xticks((5, 25))
ax.set_xticklabels(("downstream firms", "upstream firms"))
plt.show()
```

```
Iteration converged in 2 steps
```


2.5
2.0
1.5
1.0
0.5
0.0
downstream firms
upstream firms

CHAPTER

SIXTEEN

COMPOSITE SORTING

16.1 Overview

Optimal transport theory studies how a marginal probabilty measure can be related to another marginal probability measure in an ideal way.

- here ideal means to minimize some cost criterion.

The output of such a theory is a **coupling** of the two probability measures, i.e., a joint probabilty measure having those two marginal probability measures.

This lecture describes how Job Boerma, Aleh Tsyvinski, Ruodo Wang, and Zhenyuan Zhang [Boerma *et al.*, 2024] used optimal transport theory to formulate and compute an equilibrium of a model in which wages and allocations of workers across jobs adjust to match measures of different types with measures of different types of occupations.

Production technologies allow firms to reshape costs of mismatch so that they become concave.

It is then possible that in equilibrium there is neither **positive assortive** nor **negative assorting** matching, an outcome that [Boerma *et al.*, 2024] call **composite assortive** matching.

For example, with composite matching in an equilibrium model with workers of different types, ex ante identical *workers* can sort into different *occupations*, some positively and some negatively.

[Boerma *et al.*, 2024] show how composite matching can generate distinct non-trivial frequency distributions of labor earnings within and across occupations.

This lecture describes the [Boerma *et al.*, 2024] model and presents Python code for computing equilibria.

The lecture then applies the code to the [Boerma *et al.*, 2024] model of labor markets.

As with an earlier QuantEcon lecture on optimal transport, a key tool will be linear programming.

As we'll see, [Boerma *et al.*, 2024] also deploy dynamic programming in creative ways at important points in their analysis. So as you read this lecture, please watch for Bellman equations that might remind you of ideas encountered in this earlier QuantEcon lecture and this QuantEcon book.

16.2 Setup

X and Y are finite sets that represent two distinct types of people to be matched.

For each $x \in X$, let a positive integer n_x be the number of agents of type x.

Similarly, let a positive integer m_y be the agents of agents of type $y \in Y$.

We refer to these two measures as *marginals*.

We assume that

$$\sum_{x \in X} n_x = \sum_{y \in Y} m_y =: N$$

so that the matching problem is *balanced*.

Given a **cost function** $c \colon X \times Y \to \mathbb{R}$, the (discrete) **optimal transport problem** is

$$\begin{aligned} \min_{\mu \geq 0} & \sum_{(x,y) \in X \times Y} \mu_{xy} c_{xy} \\ \text{s.t.} & \sum_{x \in X} \mu_{xy} = n_x \\ & \sum_{y \in Y} \mu_{xy} = m_y \end{aligned}$$

Given our discreteness assumptions about X and Y, the problem admits an integer solution $\mu \in \mathbb{Z}_+^{X \times Y}$, i.e., μ_{xy} is a non-negative integer for each $x \in X, y \in Y$.

We will study integer solutions.

Two points about restricting ourselves to integer solutions are worth mentioning:

- it is without loss of generality for computational purposes, since every problem with float marginals can be transformed into an equivalent problem with integer marginals;
- although the mathematical structure that we present actually works for arbitrary real marginals, some of our Python implementations would fail to work with float arithmetic.

We focus on a specific instance of an optimal transport problem:

We assume that X and Y are finite subsets of $\mathbb{R}$ and that the cost function satisfies $c_{xy} = h(|x - y|)$ for all $x, y \in \mathbb{R}$, for an $h : \mathbb{R}_+ \to \mathbb{R}_+$ that is *strictly concave* and *strictly increasing* and *grounded* (i.e., $h(0) = 0$).

Such an h satisfies the following

Lemma. If $h \colon \mathbb{R}_+ \to \mathbb{R}_+$ is strictly concave and grounded, then h is strictly subadditive, i.e. for all $x, y \in \mathbb{R}_+, 0 < x < y$, we have

$$h(x + y) < h(x) + h(y)$$

Proof. For $\alpha \in (0, 1)$ and $x > 0$ we have, by strict concavity and groundedness, $h(\alpha x) > \alpha h(x) + (1-\alpha)h(0) = \alpha h(x)$.

Now fix $x, y \in \mathbb{R}_+, 0 < x < y$, and let $\alpha = \frac{x}{x+y}$; the previous observation gives $h(x) = h(\alpha(x + y)) > \alpha h(x + y)$ and $h(y) = h((1 - \alpha)(x + y)) > (1 - \alpha)h(x + y)$; summing these inequality delivers the result. □

In the following implementation we assume that the cost function is $c_{xy} = |x - y|^{1/\zeta}$ for $\zeta > 1$, i.e. $h(z) = z^{1/\zeta}$ for $z \in \mathbb{R}_+$.

Hence, our problem is

$$\begin{aligned} \min_{\mu \in \mathbb{Z}_+^{X \times Y}} & \sum_{(x,y) \in X \times Y} \mu_{xy} |x - y|^{1/\zeta} \\ \text{s.t.} & \sum_{x \in X} \mu_{xy} = n_x \\ & \sum_{y \in Y} \mu_{xy} = m_y \end{aligned}$$

Let's start setting up some Python code.

We use the following imports:

```
import numpy as np
from scipy.optimize import linprog
from itertools import chain
import pandas as pd
from collections import namedtuple


import matplotlib.pyplot as plt
import matplotlib.patches as patches
from matplotlib.ticker import MaxNLocator
from matplotlib import cm
from matplotlib.colors import Normalize
```

The following Python class takes as inputs sets of types $X, Y \subset \mathbb{R}$, marginals n, m with positive integer entries such that $\sum_{x \in X} n_x = \sum_{y \in Y} m_y$ and cost parameter $\zeta > 1$.

The cost function is stored as an $|X| \times |Y|$ matrix with (x, y)-entry equal to $|x - y|^{1/\zeta}$, i.e., the cost of matching an agent of type $x \in X$ with an agent of type $y \in Y$.

```
class ConcaveCostOT():
    def __init__(self, X_types=None, Y_types=None, n_x =None, m_y=None, ζ=2):

        # Sets of types
        self.X_types, self.Y_types = X_types, Y_types

        # Marginals
        if X_types is not None and Y_types is not None:
            non_empty_types = True
            self.n_x = np.ones(len(X_types), dtype=int) if n_x is None else n_x
            self.m_y = np.ones(len(Y_types), dtype=int) if m_y is None else m_y
        else:
            non_empty_types = False
            self.n_x, self.m_y = n_x, m_y

        # Cost function: |X|x|Y| matrix
        self.ζ = ζ
        if non_empty_types:
            self.cost_x_y = np.abs(X_types[:, None] - Y_types[None, :]) \
            ** (1 / ζ)
        else:
            self.cost_x_y = None
```

Let's consider a random instance with given numbers of types $|X|$ and $|Y|$ and a given number of agents.

First, we generate random types X and Y.

Then we generate random quantities for each type so that there are N agents for each side.

```
number_of_x_types = 20
number_of_y_types = 20
N_agents_per_side = 60

np.random.seed(1)

## Genetate random types
# generate random support for distributions of types
support_size = 50
random_support = np.unique(np.random.uniform(0,200, size=support_size))

# generate types
X_types_example = np.random.choice(random_support,
                    size=number_of_x_types, replace=False)
Y_types_example = np.random.choice(random_support,
                    size=number_of_y_types, replace=False)

## Generate random integer types quantities summing to N_agents_per_side

# generate integer vectors of lenght n_types summing to n_agents
def random_marginal(n_types, n_agents):
    cuts = np.sort(np.random.choice(np.arange(1,n_agents),
                                    size= n_types-1, replace=False))
    segments = np.diff(np.concatenate(([0], cuts, [n_agents])))
    return segments

# Create a method to assign random marginals to our class
def assign_random_marginals(self,random_seed):
    np.random.seed(random_seed)
    self.n_x = random_marginal(len(self.X_types), N_agents_per_side)
    self.m_y = random_marginal(len(self.Y_types), N_agents_per_side)

ConcaveCostOT.assign_random_marginals = assign_random_marginals

# Create an instance of our class and generate random marginals
example_pb = ConcaveCostOT(X_types_example, Y_types_example, ζ=2)
example_pb.assign_random_marginals(random_seed=1)
```

We use F (resp. G) to denote the cumulative distribution function associated to the measure n (resp. m)

Thus, $F(z) = \sum_{x \leq z : n_x > 0} n_x$ and $G(z) = \sum_{y \leq z : m_y > 0} m_y$ for $z \in \mathbb{R}$.

Notice that we not normalizing the measures so $F(\infty) = G(\infty) = N$.

The following method plots the marginals on the real line

- blue for X types,
- red for Y types.

Note that there are possible overlaps between X and Y.

```
def plot_marginals(self, figsize=(15, 8), title='Distributions of types'):

    plt.figure(figsize=figsize)

    # Scatter plot n_x
```

(continues on next page)

(continued from previous page)

```
    plt.scatter(self.X_types, self.n_x, color='blue', label='n_x')
    plt.vlines(self.X_types, ymin=0, ymax= self.n_x,
               color='blue', linestyles='dashed')

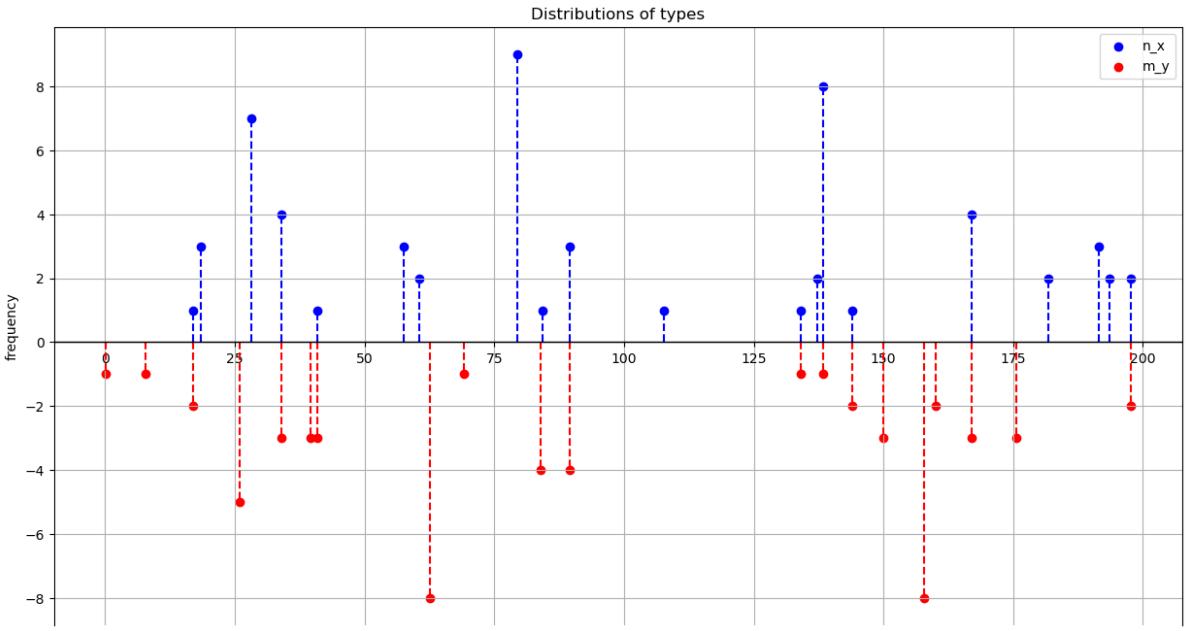
    # Scatter plot m_y
    plt.scatter(self.Y_types, - self.m_y, color='red', label='m_y')
    plt.vlines(self.Y_types, ymin=0, ymax=- self.m_y,
               color='red', linestyles='dashed')

    # Add grid and y=0 axis
    plt.grid(True)
    plt.axhline(0, color='black', linewidth=1)
    plt.gca().spines['bottom'].set_position(('data', 0))

    # Labeling the axes and the title
    plt.ylabel('frequency')
    plt.title(title)
    plt.gca().yaxis.set_major_locator(MaxNLocator(integer=True))
    plt.legend()
    plt.show()

ConcaveCostOT.plot_marginals = plot_marginals
```

```
example_pb.plot_marginals()
```



16.3 Characterization of primal solution

16.3.1 Three properties of an optimal solution

We now indicate important properties that are satisfied by an optimal solution.

1. Maximal number of perfect pairs
2. No intersecting pairs
3. Layering

Maximal number of perfect pairs

If $(z, z) \in X \times Y$ for some $z \in \mathbb{R}$ then in each optimal solution there are $\min\{n_z, m_z\}$ matches between type $z \in X$ and $z \in Y$.

Indeed, assume by contradiction that at an optimal solution we have (z, y) and (x, z) matched in positive amounts for $y, x \neq z$.

We can verify that reassigning the minimum of such quantities to the pairs (z, z) and (x, y) improves upon the current matching since

$$h(|x - y|) \leq h(|x - z| + |z - y|) < h(|x - z|) + h(|z - y|)$$

where the first inequality follows from triangle inequality and the fact that h is increasing and the strict inequality from strict subadditivity.

We can then repeat the operation for any other analogous pair of matches involving z, while improving the value, until we have mass $\min\{n_z, m_z\}$ on match (z, z).

Viewing the matching μ as a measure on $X \times Y$ with marginals n and m, this property says that in any optimal μ we have $\mu_{zz} = n_z \wedge m_z$ for (z, z) in the diagonal $\{(x, y) \in X \times Y : x = y\}$ of $\mathbb{R} \times \mathbb{R}$.

The following method finds perfect pairs and returns the on-diagonal matchings as well as the residual off-diagonal marginals.

```
def match_perfect_pairs(self):

    # Find pairs on diagonal and related mass
    perfect_pairs_x, perfect_pairs_y = np.where(
                        self.X_types[:,None] == self.Y_types[None,:])
    Λ_q = np.minimum(self.n_x[perfect_pairs_x] ,self.m_y[perfect_pairs_y])

    # Compute off-diagonal residual masses for each side
    n_x_off_diag = self.n_x.copy()
    n_x_off_diag[perfect_pairs_x]-= Λ_q

    m_y_off_diag = self.m_y.copy()
    m_y_off_diag[perfect_pairs_y] -= Λ_q

    # Compute on-diagonal matching
    matching_diag = np.zeros((len(self.X_types), len(self.Y_types)), dtype=int)
    matching_diag[perfect_pairs_x, perfect_pairs_y] = Λ_q

    return n_x_off_diag, m_y_off_diag , matching_diag

ConcaveCostOT.match_perfect_pairs = match_perfect_pairs
```

```
n_x_off_diag, m_y_off_diag , matching_diag = example_pb.match_perfect_pairs()
print(f"On-diagonal matches: {matching_diag.sum()}")
print(f"Residual types in X: {len(n_x_off_diag[n_x_off_diag >0])}")
print(f"Residual types in Y: {len(m_y_off_diag[m_y_off_diag >0])}")
```

```
On-diagonal matches: 15
Residual types in X: 14
Residual types in Y: 16
```

We can therefore create a new instance with the residual marginals that will feature no perfect pairs.

Later we shall add the on-diagonal matching to the solution of this new instance.

We refer to this instance as "off-diagonal" since the product measure of the residual marginals $n \otimes m$ feature zeros mass on the diagonal of $\mathbb{R} \times \mathbb{R}$.

In the rest of this section, we will focus on this instance.

We create a subclass to study the residual off-diagonal problem.

The subclass inherits the attributes and the modules from the original class.

We let $Z := X \sqcup Y$, where $\sqcup$ denotes the union of disjoint sets. We will

- index types X as $\{0, \dots, |X| - 1\}$ and types Y as $\{|X|, \dots, |X| + |Y| - 1\}$;
- store the cost function as a $|Z| \times |Z|$ matrix with entry (z, z') equal to c_{xy} if $z = x \in X$ and $z' = y \in Y$ or $z = y \in Y$ and $z' = x \in X$ or equal to $+\infty$ if z and z' belong to the same side
  - (the latter is just customary, since these "infinitely penalized" entries are actually never accessed in the implementation);
- let q be a vector of size $|Z|$ whose z-th entry equals n_x if type x is the z-th smallest type in Z and $-m_y$ if type y is the z-th smallest type in Z; hence q encodes capacities of both sides on the (ascending) sorted set of types.

Finally, we add a method to flexibly add a pair (i, j) with $i \in \{0, \dots, |X| - 1\}, j \in \{|X|, \dots, |X| + |Y| - 1\}$ or $j \in \{0, \dots, |X| - 1\}, i \in \{|X|, \dots, |X| + |Y| - 1\}$ to a matching matrix of size $|X| \times |Y|$.

```
class OffDiagonal(ConcaveCostOT):
    def __init__(self, X_types, Y_types, n_x, m_y, ζ):
        super().__init__(X_types, Y_types, n_x, m_y, ζ)

        # Types (unsorted)
        self.types_list = np.concatenate((X_types,Y_types))

        # Cost function: |Z|x|Z| matrix
        self.cost_z_z = np.ones((len(self.types_list),
                                 len(self.types_list))) * np.inf

        # upper-right block
        self.cost_z_z[:len(self.X_types), len(self.X_types):] = self.cost_x_y

        # lower-left block
        self.cost_z_z[len(self.X_types):, :len(self.X_types)] = self.cost_x_y.T

        ## Distributions of types
        # sorted types and index identifier for each z in support
        self.type_z = np.argsort(self.types_list)
        self.support_z = self.types_list[self.type_z]
```

(continues on next page)

(continued from previous page)

```
        # signed quantity for each type z
        self.q_z = np.concatenate([n_x, - m_y])[self.type_z]

    # Mathod that adds to matching matrix a pair (i,j)
    def add_pair_to_matching(self, pair_ids, matching):
        if pair_ids[0] < pair_ids[1]:
            # the pair of indices correspond to a pair (x,y)
            matching[pair_ids[0], pair_ids[1]-len(self.X_types)] = 1
        else:
            # the pair of indices correspond to a pair (y,x)
            matching[pair_ids[1], pair_ids[0]-len(self.X_types)] = 1
```

We add a function that returns an instance of the off-diagonal subclass as well as the on-diagonal matching and the indices of the residual off-diagonal types.

These indices will come handy for adding the off-diagonal matching matrix to the diagonal matching matrix we just found, since the former will have a smaller size if there are perfect pairs in the original problem.

```
def generate_offD_onD_matching(self):
    # Match perfect pairs and compute on-diagonal matching
    n_x_off_diag, m_y_off_diag , matching_diag = self.match_perfect_pairs()

    # Find indices of residual non-zero quantities for each side
    nonzero_id_x = np.flatnonzero(n_x_off_diag)
    nonzero_id_y = np.flatnonzero(m_y_off_diag)

    # Create new instance with off-diagonal types
    off_diagonal = OffDiagonal(self.X_types[nonzero_id_x],
                               self.Y_types[nonzero_id_y],
                               n_x_off_diag[nonzero_id_x],
                               m_y_off_diag[nonzero_id_y],
                               self.ζ)

    return off_diagonal, (nonzero_id_x, nonzero_id_y, matching_diag)

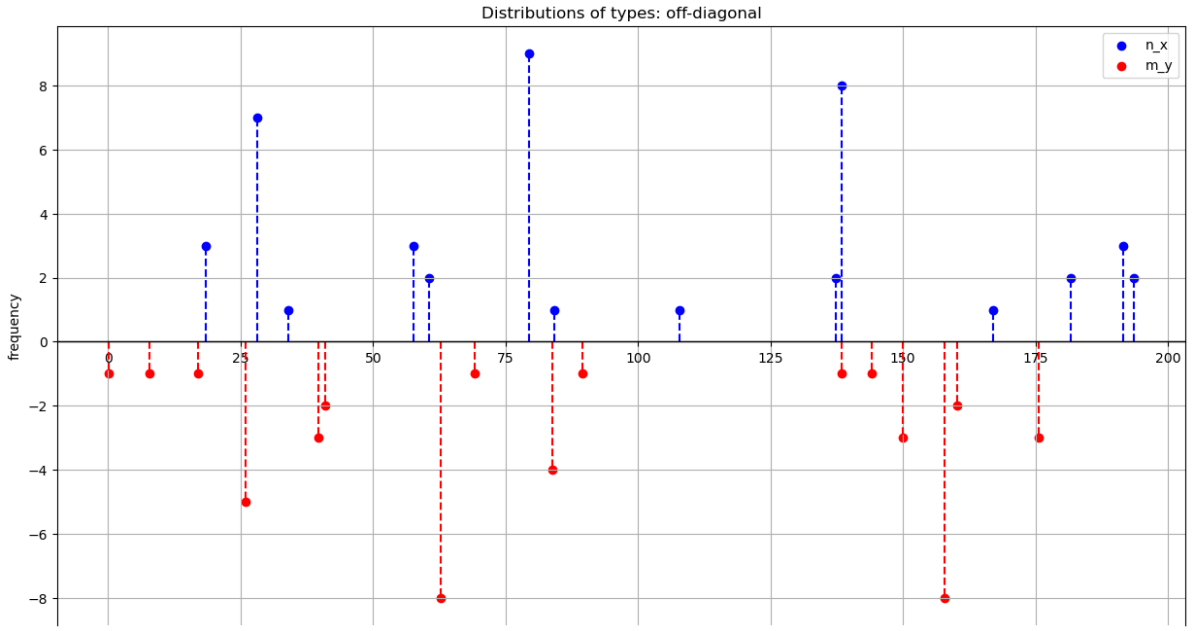
ConcaveCostOT.generate_offD_onD_matching = generate_offD_onD_matching
```

We apply it to our example:

```
example_off_diag, _ = example_pb.generate_offD_onD_matching()
```

Let's plot the residual marginals to verify visually that there are no overlappings between types from distinct sides in the off-diagonal instance.

```
example_off_diag.plot_marginals(title='Distributions of types: off-diagonal')
```



To prepare the way for the second property, represent both types on the real line and draw semicirles that join (x, y) for all pairs $(x, y) \in X \times Y$ that are matched in a solution.

In terms of these semicircles we assert the

No intersecting pairs property

- unless they share one of the endpoints, the semicirles do not intersect.

To prove the property, we reason by contradiction.

Let's consider types $x, x' \in X$ and $y, y' \in Y$.

Matched pairs cain "intersect" (or be tangent).

We will show that in both cases the partial matching among types x, x', y, y' can be improved by *uncrossing*, i.e. reassigning the quantities while improving on the solution and reducing the number of intersecting pairs.

The first case of intersecting pairs is

$$x < y < y' < x'$$

with pairs (x, y') and (x', y) matched in positive quantities.

Then it follows from strict monotonicity of h that $h(|x - y|) < h(|x - y'|)$ and $h(|x' - y'|) < h(|x' - y|)$, hence $h(|x - y|) + h(|x' - y'|) < h(|x - y'|) + h(|x' - y|)$.

Therefore, we can take the minimum of the masses of the matched pairs (x, y') and (x', y) and reallocate it to the pairs (x, y) and (x', y'), therby strictly improving the cost among x, y, x', y'.

The second case of intersecting pairs is

$$x < x' < y' < y$$

with pairs (x, y') and (x', y) matched.

In this case we have

$$|x - y'| + |x' - y| = |x - y| + |x' - y'|$$

Letting $\alpha := \frac{|x-y|+|x'-y|}{|x-y'|-|x'-y|} \in (0,1)$, we have $|x-y| = \alpha|x-y'|+(1-\alpha)|x'-y|$ and $|x'-y'| = (1-\alpha)|x-y'|+\alpha|x'-y|$.

Hence, by strict concavity of h,

$$h(|x-y|) + h(|x'-y'|) < \alpha h(|x-y'|) + (1-\alpha)h(|x'-y|) + (1-\alpha)h(|x-y'|) + \alpha h(|x'-y|) = h(|x-y'|) + h(|x'-y|).$$

Therefore, as in the first case, we can strictly improve the cost among x, y, x', y' by uncrossing the pairs.

Finally, it remains to argue that in both cases *uncrossing* operations do not increase the number of intersections with other matched pairs.

It can indeed be shown on a case-by-case basis that, in both of the above cases, for any other matched pair (x'', y'') the number of intersections between pairs $(x, y), (x', y')$ and the pair (x'', y'') (i.e., after uncrossing) is not larger than the number of intersections between pairs $(x, y'), (x', y)$ and the pair (x'', y'') (i.e., before uncrossing), hence the uncrossing operations above reduce the number of intersections.

We conclude that if a matching features intersecting pairs, it can be modified via a sequence of uncrossing operations into a matching without intersecting pairs while improving on the value.

We now consider the third property.

Layering

Recall that there are $2N$ individual agents, each agent i having type $z_i \in X \sqcup Y$.

When we introduce the off diagonal matching, to stress that the types sets are disjoint now.

To simplify our explanation of this property, assume for now that each agent has its own distinct type (i.e., $|X| = |Y| = N$ and $n = m = \mathbf{1}_N$), in which case the optimal transport problem is also referred to as *assignment problem*.

Let's index agents according to their types:

$$z_1 < z_2 \cdots < z_{2N-1} < z_{2N}.$$

Suppose that agents i of type z_i and j of type z_j, with $z_i < z_j$, are matched in a particular optimal solution.

Then there is an equal number of agents from each side in $\{i+1, \dots, j-1\}$, if this set is not empty.

Indeed, if this were not the case, then some agent $k \in \{i+1, j-1\}$ would be matched with some agent ℓ with $\ell \notin \{i, \dots, j\}$, i.e., there would be types

$$z_i < z_k < z_j < z_\ell$$

with matches (z_i, z_j) and (z_k, z_ℓ), violating the no intersecting pairs property.

We conclude that we can define a binary relation on $[N]$ such that $i \sim j$ if there is an equal number of agents of each side in $\{i, i+1, \dots, j\}$ (or if this set is empty).

This is an equivalence relation, so we can find associated equivalence classes that we call *layers*.

By the reasoning above, in an optimal solution all pairs i, j (of opposite sides) which are matched belong to the same layer, hence we can solve the assignment problem associated to each layer and then add up the solutions.

In terms of distributions, i and j, of types $x \in X$ and $y \in Y$ respectively, belong to the same layer (i.e., $x \sim y$) if and only if $F(y-) - F(x) = G(y-) - G(x)$.

If F and G were continuous, then $F(y) - F(x) = G(y) - G(x) \iff F(x) - G(x) = F(y) - G(y)$.

This suggests that the following quantity plays an important role:

$$H(z) := F(z) - G(z), \text{ for } z \in \mathbb{R}.$$

Returning to our general (integer) discrete setting, let's plot H.

Notice that H is right-continuous (being the difference of right-continuous functions) and that upward (resp. downward) jumps correspond to point masses of agents with types from X (resp. Y).

```
def plot_H_z(self, figsize=(15, 8), range_x_axis=None, scatter=True):
    # Determine H(z) = F(z) - G(z)
    H_z = np.cumsum(self.q_z)

    # Plot H(z)
    plt.figure(figsize=figsize)
    plt.axhline(0, color='black', linewidth=1)

    # determine the step points for horizontal lines
    step = np.concatenate(([self.support_z.min() - .05 * self.support_z.ptp()],
                           self.support_z,
                          [self.support_z.max() + .05 * self.support_z.ptp()]))
    height = np.concatenate(([0], H_z, [0]))

    # plot the horizontal lines of the step function
    for i in range(len(step) - 1):
        plt.plot([step[i], step[i+1]], [height[i], height[i]], color='black')

    # draw dashed vertical lines for the step function
    for i in range(1, len(step) - 1):
        plt.plot([step[i], step[i]], [height[i-1], height[i]],
                 color='black', linestyle='--')

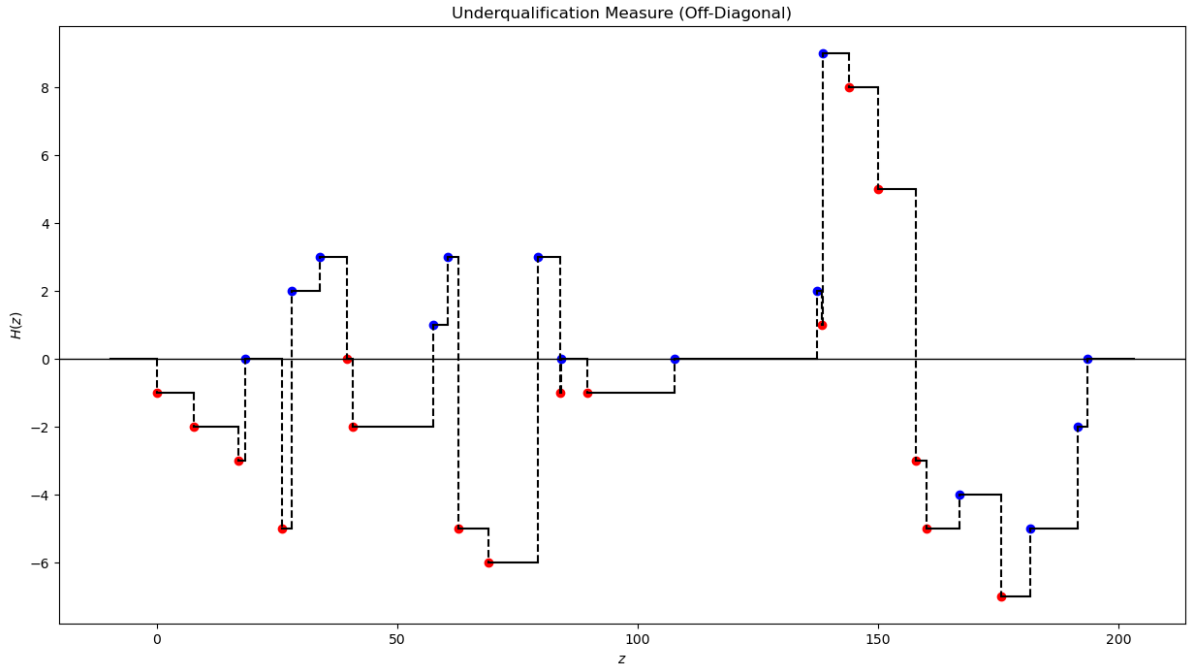
    # plot discontinuities points of H(z)
    if scatter:
        plt.scatter(np.sort(self.X_types), H_z[self.q_z > 0], color='blue')
        plt.scatter(np.sort(self.Y_types), H_z[self.q_z < 0], color='red')

    if range_x_axis is not None:
        plt.xlim(range_x_axis)

    # Add labels and title
    plt.title('Underqualification Measure (Off-Diagonal)')
    plt.xlabel('$z$')
    plt.ylabel('$H(z)$')
    plt.grid(False)
    plt.gca().yaxis.set_major_locator(MaxNLocator(integer=True))
    plt.show()

OffDiagonal.plot_H_z = plot_H_z
```

```
example_off_diag.plot_H_z()
```



The layering property extends to the general discrete setting.

There are $|H(\mathbb{R})| - 1$ layers in total.

Enumerating the range of H as $H(\mathbb{R}) = \{h_1, h_2, \dots, h_{|H(\mathbb{R})|}\}$ with $h_1 < h_2 < \cdots < h_{|H(\mathbb{R})|}$, we can define layer L_ℓ, for $\ell \in \{1, \dots, |H(\mathbb{R})| - 1\}$ as the collection of types $z \in Z$ such that

$$H(z-) \leq h_{\ell-1} < h_\ell \leq H(z),$$

(which are types in X), or

$$H(z) \leq h_{\ell-1} < h_\ell \leq H(z-),$$

which are types in Y.

The *mass* associated with layer L_ℓ is $M_\ell = h_{\ell+1} - h_\ell$.

Intuitively, a layer L_ℓ consists of some mass M_ℓ, of multiple types in Z, i.e. the problem within the layer is *unitary*.

A unitary problem is essentially an assignment problem up to a constant: we can solve the problem with unit mass and then rescale a solution by M_ℓ.

Moreover, each layer L_ℓ contains an even number of types $N_\ell \in 2\mathbb{N}$, which are alternating, i.e., ordering them as $z_1 < z_2 \cdots < z_{N_\ell - 1} < z_{N_\ell}$ all odd (or even, respectively) indexed types belong to the same side.

The following method finds the layers associated with distributions F and G.

Again, types in X are indexed with $\{0, \dots, |X| - 1\}$ and types in Y with $\{|X|, \dots, |X| + |Y| - 1\}$.

Using these indices (instead of the types themselves) to represent the layers allows keeping track of sides types in each layer, without adding an additional bit of information that would identify the side of the first type in the layer, which, because a layer is alternating, would then allow identifying sides of all types in the layer.

In addition, using indices will let us extract the cost function within a layer from the cost function $c_{zz'}$ computed offline.

```
def find_layers(self):
    # Compute H(z) on the joint support
    H_z = np.concatenate([[0], np.cumsum(self.q_z)])

    # Compute the range of H, i.e. H(R), stored in ascending order
    layers_height = np.unique(H_z)

    # Compute the mass of each layer
    layers_mass = np.diff(layers_height)

    # Compute layers
    # the following |H(R)|x|Z| matrix has entry (z,l) equal to 1 iff type z belongs␣
 ↪to layer l
    layers_01 = ((H_z[None, :-1] <= layers_height[:-1, None])
                 * (layers_height[1:, None] <= H_z[None, 1:]) |
                (H_z[None, 1:] <= layers_height[:-1, None])
                 * (layers_height[1:, None] <= H_z[None, :-1]))

    # each layer is reshaped as a list of indices correponding to types
    layers = [self.type_z[layers_01[ell]]
                for ell in range(len(layers_height)-1)]

    return layers, layers_mass, layers_height, H_z

OffDiagonal.find_layers = find_layers
```

```
layers_list_example, layers_mass_example, _, _ = example_off_diag.find_layers()
print(layers_list_example)
```

```
[array([23, 10]), array([27,  3, 23, 10]), array([16,  2, 21,  3, 25,  8, 23, 12]),
 ↪ array([16,  2, 21,  3, 25, 12]), array([22,  0, 16,  2, 21,  3, 18, 12]),␣
 ↪array([15,  0, 16,  2, 14,  5, 21,  3, 18,  9]), array([20,  0, 16,  2, 14,  5,␣
 ↪21,  3, 19, 11, 24,  1, 18,  9]), array([ 2, 26,  5, 21,  3, 19,  4, 18]),␣
 ↪array([ 2, 26,  7, 21,  3, 19,  4, 17,  6, 18]), array([13, 26,  7, 21,  3, 19, ␣
 ↪6, 18]), array([ 6, 18]), array([ 6, 28]), array([ 6, 29])]
```

The following method provides a graphical representation of the layers.

From the picture it is easy to spot two key features described above:

- types are alternating
- the layer problem is unitary

```
def plot_layers(self, figsize=(15, 8)):
    # Find layers
    layers, layers_mass , layers_height, H_z = self.find_layers()

    plt.figure(figsize=figsize)

    # Plot H(z)
    step = np.concatenate(([self.support_z.min() - .05 * self.support_z.ptp()],
                           self.support_z,
                           [self.support_z.max() + .05 * self.support_z.ptp()]))
    height = np.concatenate((H_z, [0]))
    plt.step(step, height, where='post', color='black', label='CDF', zorder=1)
```

(continues on next page)

(continued from previous page)

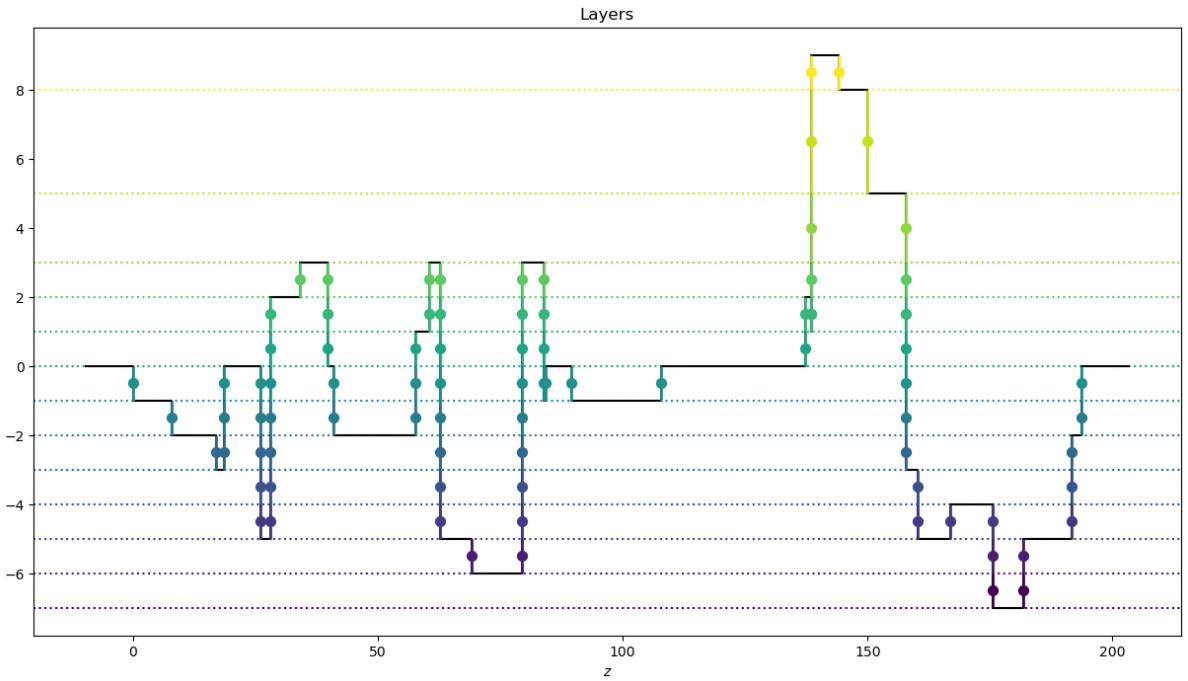
```
    # Plot layers
    colors = cm.viridis(np.linspace(0, 1, len(layers)))
    for  ell, layer  in enumerate(layers):
        plt.vlines(self.types_list[layer], layers_height[ell] ,
                   layers_height[ell] + layers_mass[ell],
                   color=colors[ell], linewidth=2)
        plt.scatter(self.types_list[layer],
                    np.ones(len(layer)) * layers_height[ell]
                    +.5 * layers_mass[ell],
                    color=colors[ell], s=50)

        plt.axhline(layers_height[ell], color=colors[ell],
                    linestyle=':', linewidth=1.5, zorder=0)

    # Add labels and title
    plt.xlabel('$z$')
    plt.title('Layers')
    plt.gca().yaxis.set_major_locator(MaxNLocator(integer=True))
    plt.show()

OffDiagonal.plot_layers = plot_layers
```

```
example_off_diag.plot_layers()
```



16.3.2 Solving a layer

Recall that layer L_ℓ consists of a list of distinct types from $Y \sqcup X$

$$z_1 < z_2 \cdots < z_{N_\ell - 1} < z_{N_\ell},$$

which is alternating.

The problem within a layer is unitary.

Hence we can solve the problem with unit masses and later rescale the solution by the layer's mass M_ℓ.

Let us select a layer from the example above (we pick the one with maximum number of types) and plot the types on the real line

```
# Pick layer with maximum number of types
layer_id_example = max(enumerate(layers_list_example),
                       key = lambda x: len(x[1]))[0]
layer_example = layers_list_example[layer_id_example]


# Plot layer types
def plot_layer_types(self, layer, mass, figsize=(15, 3)):

    plt.figure(figsize=figsize)

    # Scatter plot n_x
    x_layer = layer[layer < len(self.X_types)]
    y_layer = layer[layer >= len(self.X_types)] - len(self.X_types)
    M_ell = np.ones(len(x_layer))* mass

    plt.scatter(self.X_types[x_layer], M_ell, color='blue', label='X types')
    plt.vlines(self.X_types[x_layer], ymin=0, ymax= M_ell,
               color='blue', linestyles='dashed')

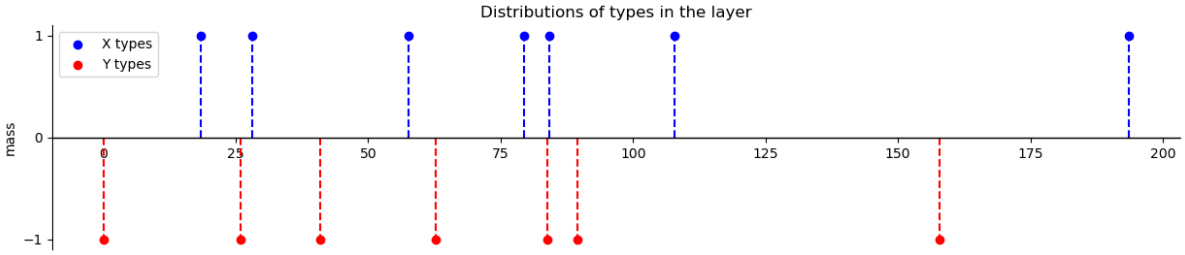
    # Scatter plot m_y
    plt.scatter(self.Y_types[y_layer], - M_ell, color='red', label='Y types')
    plt.vlines(self.Y_types[y_layer], ymin=0, ymax=- M_ell,
               color='red', linestyles='dashed')

    # Add grid and y=0 axis
    # plt.grid(True)
    plt.axhline(0, color='black', linewidth=1)
    plt.gca().spines['bottom'].set_position(('data', 0))

    # Labeling the axes and the title
    plt.ylabel('mass')
    plt.title('Distributions of types in the layer')
    plt.gca().yaxis.set_major_locator(MaxNLocator(integer=True))
    plt.gca().spines['top'].set_visible(False)
    plt.gca().spines['right'].set_visible(False)
    plt.legend()
    plt.show()

ConcaveCostOT.plot_layer_types = plot_layer_types

example_off_diag.plot_layer_types(layer_example,
                                  layers_mass_example[layer_id_example])
```



Given the structure of a layer and the *no intersecting pairs* property, the optimal matching and value of the layer can be found recursively.

Indeed, if in certain optimal matching 1 and $j \in [N_\ell]$, $j-1$ odd, are paired, then there is no matching between agents in $[2, j-1]$ and those in $[j+1, N_\ell]$ (if both are non empty, i.e., j is not 2 or N_ℓ).

Hence in such optimal solution agents in $[2, j-1]$ are matched among themselves.

Since $[z_2, z_{j-1}]$ (as well as $[z_{j+1}, z_{N_\ell}]$) is alternating, we can reason recursively.

Let V_{ij} be the optimal value of matching agents in $[i, j]$ with $i, j \in [N_\ell]$, $j - i \in \{1, 3, \dots, N_\ell - 1\}$.

Suppose that we computed the value V_{ij} for all $i, j \in [N_\ell]$ with $i - j \in \{1, 3, \dots, t-2\}$ for some odd natural number t.

Then for $i, j \in [N_\ell]$ with $i - j = t$

$$V_{ij} = \min_{k \in \{i+1, i+3, \dots, j\}} \{c_{ik} + V_{i+1,k-1} + V_{k+1,j}\} \tag{16.1}$$

with the RHS depending only on previously computed values.

We set the boundary conditions at $t = -1$: $V_{i+1,i} = 0$ for each $i \in [N_\ell]$, so that we can apply the same Bellman equation (16.1) at $t = 1$.

The following method takes as input the layer types indices and computes the value function as a matrix $[V_{ij}]_{i \in [N_\ell+1], j \in [N_\ell]}$.

In order to distinguish entries that are relevant for our computations from those that are never accessed, we initialize this matrix as full of NaN values.

```
def solve_bellman_eqs(self,layer):
    # Recover cost function within the layer
    cost_i_j = self.cost_z_z[layer[:,None],layer[None,:]]

    # Initialize value function
    V_i_j = np.full((len(layer)+1,len(layer)), np.nan)

    # Add boundary conditions
    i_bdry = np.arange(len(layer))
    V_i_j[i_bdry+1, i_bdry] = 0

    t = 1
    while t < len(layer):
        # Select agents i in [n_L-t] (with potential partners j's in [t,n_L])
        i_t = np.arange(len(layer)-t)

        # For each i, select each k with |k-i| <= t
        # (potential partners of i within segment)
        index_ik =  i_t[:,None] + np.arange(1, t+1, 2)[None,:]

        # Compute optimal value for pairs with |i-j| = t
```

(continues on next page)

(continued from previous page)

```
        V_i_j[i_t, i_t + t] = (cost_i_j[i_t[:,None], index_ik] +
                               V_i_j[i_t[:,None] + 1, index_ik - 1] +
                               V_i_j[index_ik + 1, i_t[:,None] + t]).min(1)
        # Go to next odd integer
        t += 2

    return V_i_j

OffDiagonal.solve_bellman_eqs = solve_bellman_eqs
```

Let's compute values for the layer from our example.

Only non-NaN entries are actually used in the computations.

```
# Compute layer value function
V_i_j = example_off_diag.solve_bellman_eqs(layer_example)

print(f"Type indices in the layer: {layer_example}")
print('############################')
print("Section of the Value function of the layer:")
print(V_i_j.round(2)[:min(10, V_i_j.shape[0]),
                     :min(10, V_i_j.shape[1])])
```

```
Type indices in the layer: [20  0 16  2 14  5 21  3 19 11 24  1 18  9]
############################
Section of the Value function of the layer:
[[  nan  4.29   nan  5.73   nan  9.82   nan 13.9    nan 14.52]
 [ 0.     nan  2.75   nan  6.17   nan  8.44   nan 10.56   nan]
 [  nan  0.     nan  1.44   nan  5.52   nan  9.6    nan 10.22]
 [  nan   nan  0.     nan  3.58   nan  5.84   nan  7.96   nan]
 [  nan   nan   nan  0.     nan  4.08   nan  8.16   nan  8.78]
 [  nan   nan   nan   nan  0.     nan  2.26   nan  4.38   nan]
 [  nan   nan   nan   nan   nan  0.     nan  4.08   nan  4.7 ]
 [  nan   nan   nan   nan   nan   nan  0.     nan  2.12   nan]
 [  nan   nan   nan   nan   nan   nan   nan  0.     nan  0.62]
 [  nan   nan   nan   nan   nan   nan   nan   nan  0.     nan]]
```

Policy evaluation

Having computed the value function, we can proceed to compute the optimal matching as the *policy* that attains the value function that solves the Bellman equation (16.1).

We start from agent 1 and match it with the k that achieves the minimum in the equation associated with $V_{1,2N_\ell}$.

Then we store segments $[2, k-1]$ and $[k+1, 2N_\ell]$ (if not empty).

In general, given a segment $[i, j]$, we match i with k that achieves the minimum in the equation associated with V_{ij} and store the segments $[i, k-1]$ and $[k+1, j]$ (if not empty).

The algorithm proceeds until there are no segments left.

```
def find_layer_matching(self, V_i_j, layer):
    # Initialize
    segments_to_process  = [np.arange(len(layer))]
    matching = np.zeros((len(self.X_types),len(self.Y_types)), bool)

    while segments_to_process:
```

(continues on next page)

(continued from previous page)

```
        # Pick i, first agent of the segment
        # and potential partners i+1,i+3,..., in the segment
        segment = segments_to_process[0]
        i_0 = segment[0]
        potential_matches = np.arange(i_0, segment[-1], 2) + 1

        # Compute optimal partner j_i
        obj = (self.cost_z_z[layer[i_0],layer[potential_matches]] +
                V_i_j[i_0 +1, potential_matches -1] +
                V_i_j[potential_matches +1,segment[-1]])

        j_i_0 = np.argmin(obj)*2 + (i_0 + 1)

        # Add matched pair (i,j_i)
        self.add_pair_to_matching(layer[[i_0,j_i_0]], matching)

        # Update segments to process:
        # remove current segment
        segments_to_process = segments_to_process[1:]

        # add [i+1,j-1] and [j+1,last agent of the segment]
        if j_i_0 > i_0 + 1:
            segments_to_process.append(np.arange(i_0 + 1, j_i_0))
        if j_i_0 < segment[-1]:
            segments_to_process.append(np.arange(j_i_0 + 1, segment[-1] + 1))

    return matching

OffDiagonal.find_layer_matching = find_layer_matching
```

Lets apply this method our example to find the matching within the layer and then rescale it by M_ℓ.

Note that the unscaled value equals V_{1,N_ℓ}.

```
matching_layer = example_off_diag.find_layer_matching(V_i_j,layer_example)
print(f"Value of the layer (unscaled): {(matching_layer * example_off_diag.cost_x_y).
 ↪sum()}")
print(f"Value of the layer (scaled by the mass = {layers_mass_example[layer_id_
 ↪example]}): "
      f"{layers_mass_example[layer_id_example] * (matching_layer * example_off_diag.
 ↪cost_x_y).sum()}")
```

```
Value of the layer (unscaled): 24.764959193288938
Value of the layer (scaled by the mass = 1): 24.764959193288938
```

The following method plots the matching within a layer.

We apply it to the layer from our example.

```
def plot_layer_matching(self, layer, matching_layer):
    # Create the figure and axis
    fig, ax = plt.subplots(figsize=(15, 15))

    # Plot the points on the x-axis
    X_types_layer = self.X_types[layer[layer < len(self.X_types)]]
```

(continues on next page)

(continued from previous page)

```
    Y_types_layer = self.Y_types[layer[layer >= len(self.X_types)]
                    - len(self.X_types)]
    ax.scatter(X_types_layer, np.zeros_like(X_types_layer), color='blue',
               s = 20, zorder=5)
    ax.scatter(Y_types_layer, np.zeros_like(Y_types_layer), color='red',
               s = 20, zorder=5)

    # Draw semicircles for each row in matchings
    matched_types = np.where(matching_layer >0)
    matched_types_x = self.X_types[matched_types[0]]
    matched_types_y = self.Y_types[matched_types[1]]

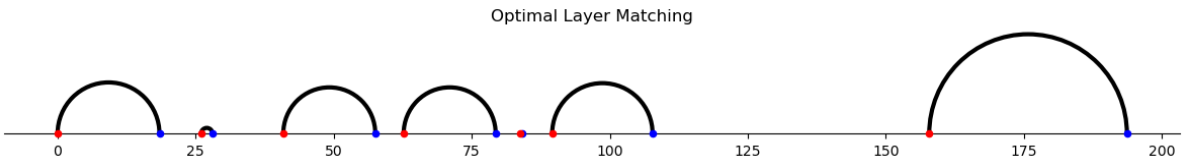
    for iter in range(len(matched_types_x)):
        width = abs(matched_types_x[iter] - matched_types_y[iter])
        center = (matched_types_x[iter] + matched_types_y[iter]) / 2
        height = width
        semicircle = patches.Arc((center, 0), width, height, theta1=0,
                     theta2=180, lw=3)
        ax.add_patch(semicircle)

    # Add title and layout settings
    plt.title('Optimal Layer Matching' )
    ax.set_aspect('equal')
    plt.gca().spines['bottom'].set_position(('data', 0))
    ax.spines['left'].set_color('none')
    ax.spines['top'].set_color('none')
    ax.spines['right'].set_color('none')
    ax.yaxis.set_ticks([])
    ax.set_ylim(bottom= -self.support_z.ptp() / 100)

    plt.show()

ConcaveCostOT.plot_layer_matching = plot_layer_matching
```

```
example_off_diag.plot_layer_matching(layer_example, matching_layer)
```



Solving a layer in a smarter way

We now present two key results in the context of OT with concave type costs.

We refer [Boerma *et al.*, 2024] and [Delon *et al.*, 2011] for proofs.

Consider the problem faced within a layer, i.e., types from $Y \sqcup X$

$$z_1 < z_2 \cdots < z_{N_\ell - 1} < z_{N_\ell}, \quad N_\ell \in 2\mathbb{N}$$

are alternating and the problem is unitary.

Given a matching on $[1, k]$, $k \in [N_\ell]$, k even, we say that a matched pair (i, j) within this matching is *hidden* if there is a matched pair (i', j') with $i' < i < j < j'$.

Visually, the arc joining (i', j') surmounts the arc joining (i, j).

Theorem (DSS) Given an optimal matching on $[1, k]$, if (i, j) is hidden in this matching, then the pair (i, j) belongs to every optimal matching on $[1, 2N_\ell]$ and is hidden in this matching too.

As a consequence, there exists a more efficient way to compute the value function within a layer.

It can be shown that the solving the following second-order difference equations delivers the same result as the Bellman equations (16.1) presented above:

$$V_{ij} = \min\{c_{ij} + V_{i+1,j-1}, V_{i+2,j} + V_{i,j-2} - V_{i+2,j-2}\}$$

for $i, j \in [N_\ell]$, $j - i$ odd, with boundary conditions $V_{i+1,i} = 0$ for $i \in [0, N_\ell]$ and $V_{i+2,i-1} = -c_{i,i+1}$ for $i \in [N_\ell - 1]$.

The following method uses these equations to compute the value function that is stored as a matrix $[V_{ij}]_{i\in[N_\ell+1],j\in[N_\ell+1]}$.

```
def solve_bellman_eqs_DSS(self,layer):
    # Recover cost function within the layer
    cost_i_j = self.cost_z_z[layer[:,None],layer[None,:]]

    # Initialize value function
    V_i_j = np.full((len(layer)+1,len(layer)+1), np.nan)

    # Add boundary conditions
    V_i_j[np.arange(len(layer)+1), np.arange(len(layer)+1)] = 0
    i_bdry = np.arange(len(layer)-1)
    V_i_j[i_bdry+2,i_bdry] = - cost_i_j[i_bdry, i_bdry+1]

    t = 1
    while t < len(layer):
        # Select agents i in [n_l-t] and potential partner j=i+t for each i
        i_t = np.arange(len(layer)-t)
        j_t = i_t + t +1

        # Compute optimal values for ij with j-i = t
        V_i_j[i_t, j_t] = np.minimum(cost_i_j[i_t, j_t-1]
                        + V_i_j[i_t + 1, j_t - 1],
                        V_i_j[i_t, j_t - 2] + V_i_j[i_t + 2, j_t]
                        - V_i_j[i_t + 2, j_t - 2])

        ## Go to next odd integer
        t += 2

    return V_i_j

OffDiagonal.solve_bellman_eqs_DSS = solve_bellman_eqs_DSS
```

Let's apply the algorithm to our example and compare outcomes with those attained with the Bellman equations above.

```
V_i_j_DSS = example_off_diag.solve_bellman_eqs_DSS(layer_example)

print(f"Type indices of the layer: {layer_example}")
print('############################')

print("Section of Value function of the layer:")
print(V_i_j_DSS.round(2)[:min(10, V_i_j_DSS.shape[0]),
                         :min(10, V_i_j_DSS.shape[1])])

print('############################')
```

(continues on next page)

(continued from previous page)

```
print(f"Difference with previous Bellman equations: \
      {(V_i_j_DSS[:,1:] - V_i_j)[V_i_j >= 0].sum()}")
```

```
Type indices of the layer: [20  0 16  2 14  5 21  3 19 11 24  1 18  9]
##########################
Section of Value function of the layer:
[[ 0.    nan  4.29   nan  5.73   nan  9.82   nan 13.9    nan]
 [  nan  0.    nan  2.75   nan  6.17   nan  8.44   nan 10.56]
 [-4.29   nan  0.    nan  1.44   nan  5.52   nan  9.6    nan]
 [  nan -2.75   nan  0.    nan  3.58   nan  5.84   nan  7.96]
 [  nan   nan -1.44   nan  0.    nan  4.08   nan  8.16   nan]
 [  nan   nan   nan -3.58   nan  0.    nan  2.26   nan  4.38]
 [  nan   nan   nan   nan -4.08   nan  0.    nan  4.08   nan]
 [  nan   nan   nan   nan   nan -2.26   nan  0.    nan  2.12]
 [  nan   nan   nan   nan   nan   nan -4.08   nan  0.    nan]
 [  nan   nan   nan   nan   nan   nan   nan -2.12   nan  0.  ]]
##########################
Difference with previous Bellman equations:       4.440892098500626e-14
```

We can actually compute the optimal matching within the layer simultaneously with computing the value function, rather than sequentially.

The key idea is that, if at some step of the computation of the values the left branch of the minimum above achieves the minimum, say $V_{ij} = c_{ij} + V_{i+1,j-1}$, then (i, j) are optimally matched on $[i, j]$ and by the theorem above we get that a matching on $[i+1, j-1]$ which achieves $V_{i+1,j-1}$ belongs to an optimal matching on the whole layer (since it is covered by the arc (i, j) in $[i, j]$).

We can therefore proceed as follows

We initialize an empty matching and a list with all the agents in the layer (representing the agents which are not matched yet).

Then whenever the left branch of the minimum is achieved for some (i, j) in the computation of V, we take the collections of agents $k_1, \dots, k_M$ in $[i + 1, j - 1]$ (in ascending order, i.e. with $z_{k_p} < z_{k_{p+1}}$) that are not matched yet (if any) and add to the matching the pairs $(k_1, k_2), (k_3, k_4), \dots, (k_{M-1}, k_M)$.

Thus, we match each unmatched agent k_p in $[i+1, j-1]$ with the closest unmatched right neighbour k_{p+1} (starting from k_1).

Intuitively, if k_p were optimally matched with some k_q in $[i + 1, j - 1]$ and not with k_{p+1}, then k_{p+1} would have already been hidden by the match (k_p, k_q) from some previous computation (because $|k_p - k_q| < |i - j|$) and it would therefore be matched.

Finally, if the process above leaves some umatched agents, we proceed by matching each of these agent with the closest unmatched right neighbour, starting again from the leftmost of these collection.

To gain understanding, note that this situation happens when the left branch is achieved only for pairs i, j with $|i-j| = 1$, which leads to the optimal matching $(1, 2), (2, 3), \dots, (n_\ell - 1, n_\ell)$.

```
def find_layer_matching_DSS(self,layer):
    # Recover cost function within the layer
    cost_i_j = self.cost_z_z[layer[:,None],layer[None,:]]

    # Add boundary conditions
    V_i_j = np.zeros((len(layer)+1,len(layer)+1))
    i_bdry = np.arange(len(layer)-1)
    V_i_j[i_bdry+2,i_bdry] = - cost_i_j[i_bdry, i_bdry+1]
```

(continues on next page)

(continued from previous page)

```
    # Initialize matching and list of to-match agents
    unmatched = np.ones(len(layer), dtype = bool)
    matching = np.zeros((len(self.X_types),len(self.Y_types)), bool)

    t = 1
    while t < len(layer):
        # Compute optimal value for pairs with |i-j| = t
        i_t = np.arange(len(layer)-t)
        j_t = i_t + t + 1

        left_branch = cost_i_j[i_t, j_t-1] + V_i_j[i_t + 1, j_t - 1]
        V_i_j[i_t, j_t] = np.minimum(left_branch, V_i_j[i_t, j_t - 2]
                        + V_i_j[i_t + 2, j_t] - V_i_j[i_t + 2, j_t - 2])

        # Select each i for which left branch achieves minimum in the V_{i,i+t}
        left_branch_achieved = i_t[left_branch == V_i_j[i_t, j_t]]

        # Update matching
        for i in left_branch_achieved:
            # for each agent k in [i+1,i+t-1]
            for k in np.arange(i+1,i+t)[unmatched[range(i+1,i+t)]]:
                # if k is unmatched
                if unmatched[k] == True:
                    # find unmatched right neighbour
                    j_k = np.arange(k+1,len(layer))[unmatched[k+1:]][0]
                    # add pair to matching
                    self.add_pair_to_matching(layer[[k, j_k]], matching)
                    # remove pair from unmatched agents list
                    unmatched[[k, j_k]] = False

        # go to next odd integer
        t += 2

    # Each umatched agent is matched with next unmatched agent
    for i in np.arange(len(layer))[unmatched]:
        # if i is unmatched
        if unmatched[i] == True:
            # find unmatched right neighbour
            j_i = np.arange(i+1,len(layer))[unmatched[i+1:]][0]
            # add pair to matching
            self.add_pair_to_matching(layer[[i, j_i]], matching)
            # remove pair from unmatched agents list
            unmatched[[i, j_i]] = False

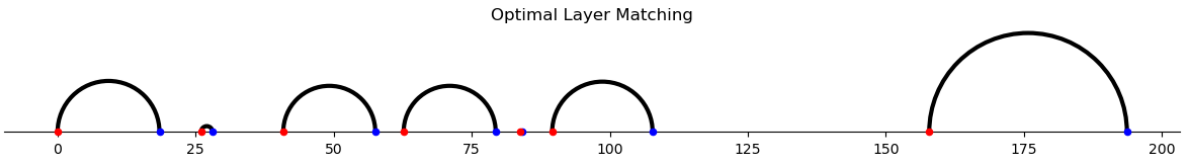
    return matching

OffDiagonal.find_layer_matching_DSS = find_layer_matching_DSS
```

```
matching_layer_DSS = example_off_diag.find_layer_matching_DSS(layer_example)
print(f" Value of layer with DSS recursive equations \
{(matching_layer_DSS * example_off_diag.cost_x_y).sum()}")
print(f" Value of layer with Bellman equations \
{(matching_layer * example_off_diag.cost_x_y).sum()}")
```

```
Value of layer with DSS recursive equations 24.764959193288938
Value of layer with Bellman equations 24.764959193288938
```

```
example_off_diag.plot_layer_matching(layer_example, matching_layer_DSS)
```



16.4 Solving primal problem

The following method assembles our components in order to solve the primal problem.

First, if matches are perfect pairs, we store the on-diagonal matching and create an off-diagonal instance with the residual marginals.

Then we compute the set of layers of the residual distributions.

Finally, we solve each layer and put together matchings within each layer with the on-diagonal matchings.

We then return the full matching, the off-diagonal matching, and the off-diagonal instance.

```
def solve_primal_pb(self):
    # Compute on-diagonal matching, create new instance with resitual types
    off_diagoff_diagonal, match_tuple = self.generate_offD_onD_matching()
    nonzero_id_x, nonzero_id_y, matching_diag = match_tuple

    # Compute layers
    layers_list, layers_mass, _, _ = off_diagoff_diagonal.find_layers()

    # Solve layers to compute off-diagonal matching
    matching_off_diag = np.zeros_like(off_diagoff_diagonal.cost_x_y, dtype=int)

    for ell, layer  in enumerate(layers_list):
        V_i_j = off_diagoff_diagonal.solve_bellman_eqs(layer)
        matching_off_diag += layers_mass[ell] \
                    * off_diagoff_diagonal.find_layer_matching(V_i_j, layer)

    # Add together on- and off-diagonal matchings
    matching = matching_diag.copy()
    matching[np.ix_(nonzero_id_x, nonzero_id_y)] += matching_off_diag

    return matching, matching_off_diag, off_diagoff_diagonal

ConcaveCostOT.solve_primal_pb = solve_primal_pb
```

```
matching, matching_off_diag, off_diagoff_diagonal = example_pb.solve_primal_pb()
```

We implement a similar method that adopts the DSS algorithm

```
def solve_primal_DSS(self):
    # Compute on-diagonal matching, create new instance with resitual types
    off_diagoff_diagonal, match_tuple = self.generate_offD_onD_matching()
    nonzero_id_x, nonzero_id_y, matching_diag = match_tuple

    # Find layers
    layers, layers_mass, _, _ = off_diagoff_diagonal.find_layers()

    # Solve layers to compute off-diagonal matching
    matching_off_diag = np.zeros_like(off_diagoff_diagonal.cost_x_y, dtype=int)

    for ell, layer  in enumerate(layers):
        matching_off_diag += layers_mass[ell] \
                           * off_diagoff_diagonal.find_layer_matching_DSS(layer)

    # Add together on- and off-diagonal matchings
    matching = matching_diag.copy()
    matching[np.ix_(nonzero_id_x, nonzero_id_y)] += matching_off_diag

    return matching, matching_off_diag, off_diagoff_diagonal

ConcaveCostOT.solve_primal_DSS = solve_primal_DSS
```

```
DSS_tuple = example_pb.solve_primal_DSS()
matching_DSS, matching_off_diag_DSS, off_diagoff_diagonal_DSS = DSS_tuple
```

By drawing semicircles joining the matched agents (with distinct types), we can visualize the off-diagonal matching.

In the following figure, widths and colors of semicirles indicate relative numbers of agents that are “transported” along an arc.

```
def plot_matching(self, matching_off_diag, title, figsize=(15, 15),
                  add_labels=False, plot_H_z=False, scatter=True):

    # Create the figure and axis
    fig, ax = plt.subplots(figsize=figsize)

    # Plot types on the real line
    if scatter:
        ax.scatter(self.X_types, np.zeros_like(self.X_types), color='blue',
                   s=20, zorder=5)
        ax.scatter(self.Y_types, np.zeros_like(self.Y_types), color='red',
                   s=20, zorder=5)

    # Add labels for X_types and Y_types if add_labels is True
    if add_labels:
        # Remove x-axis ticks
        ax.set_xticks([])

        # Add labels
        for i, x in enumerate(self.X_types):
            ax.annotate(f'$x_{{{i }}}$', (x, 0), textcoords="offset points",
                        xytext=(0, -15), ha='center', color='blue', fontsize=12)
        for j, y in enumerate(self.Y_types):
            ax.annotate(f'$y_{{{j }}}$', (y, 0), textcoords="offset points",
                        xytext=(0, -15), ha='center', color='red', fontsize=12)
```

(continues on next page)

(continued from previous page)

```
    # Draw semicircles for each pair of matched types
    matched_types = np.where(matching_off_diag > 0)
    matched_types_x = self.X_types[matched_types[0]]
    matched_types_y = self.Y_types[matched_types[1]]

    count = matching_off_diag[matched_types]
    colors = plt.cm.Greys(np.linspace(0.5, 1.5, count.max() + 1))
    max_height = 0
    for iter in range(len(count)):
        width = abs(matched_types_x[iter] - matched_types_y[iter])
        center = (matched_types_x[iter] + matched_types_y[iter]) / 2
        height = width
        max_height = max(max_height, height)
        semicircle = patches.Arc((center, 0), width, height,
                                 theta1=0, theta2=180,
                                 color=colors[count[iter]],
                                 lw=count[iter] * (2.2 / count.max()))
        ax.add_patch(semicircle)

    # Title and layout settings for the main plot
    plt.title(title)
    ax.set_aspect('equal')
    plt.axhline(0, color='black', linewidth=1)
    ax.spines['bottom'].set_position(('data', 0))
    ax.spines['left'].set_color('none')
    ax.spines['top'].set_color('none')
    ax.spines['right'].set_color('none')
    ax.yaxis.set_ticks([])
    ax.set_ylim(- self.X_types.ptp() / 10,
                (max_height / 2) + self.X_types.ptp()*.01)

    # Plot H_z on the main axis if enabled
    if plot_H_z:
        H_z = np.cumsum(self.q_z)

        step = np.concatenate(([self.support_z.min()
                                - .02 * self.support_z.ptp()],
                               self.support_z,
                               [self.support_z.max()
                               + .02 * self.support_z.ptp()]))

        H_z = H_z/H_z.ptp() * self.support_z.ptp() /2
        height = np.concatenate(([0], H_z, [0]))

        # Plot the compressed H_z on the same main x-axis
        ax.step(step, height, color='green', lw=2,
                               label='$H_z$', where='post')

        # Set the y-limit to keep H_z and maximum circle size in the plot
        ax.set_ylim(np.min(H_z) - H_z.ptp() *.01,
                    np.maximum(np.max(H_z), max_height / 2) + H_z.ptp() *.01)

        # Add label and legend for H_z
        ax.legend(loc="upper right")
```

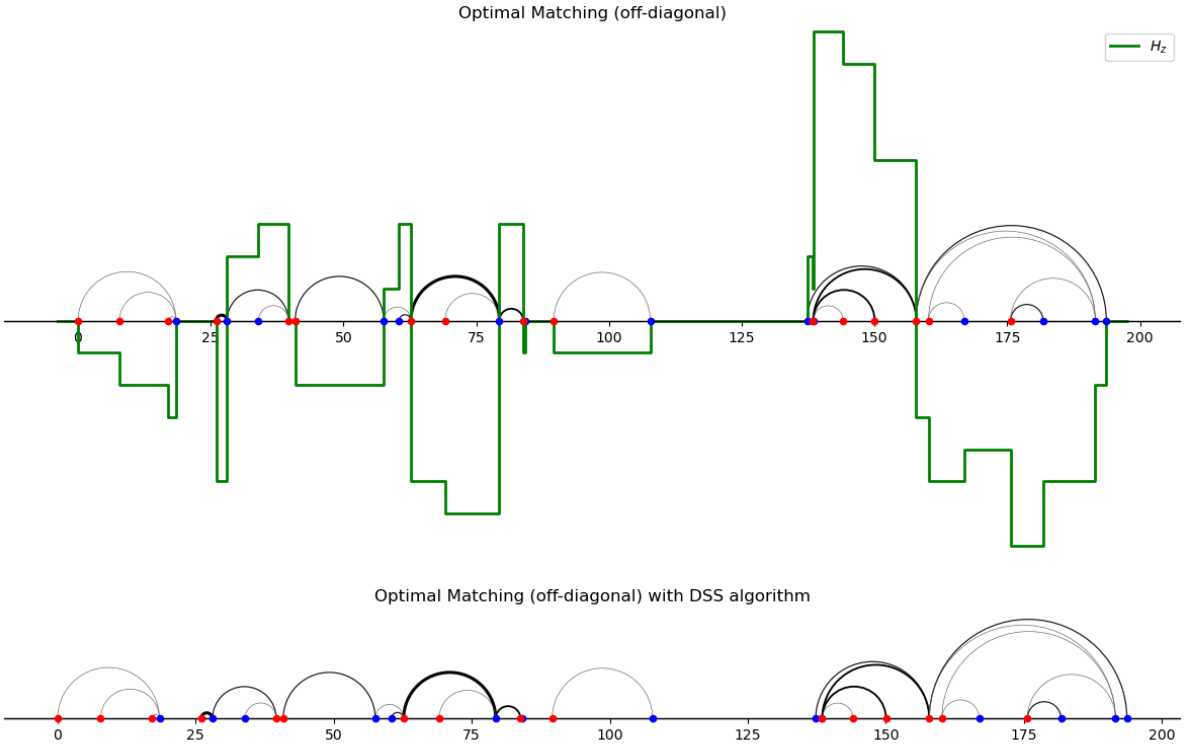
(continues on next page)

(continued from previous page)

```
        plt.show()

ConcaveCostOT.plot_matching = plot_matching
```

```
off_diagoff_diagonal.plot_matching(matching_off_diag,
            title='Optimal Matching (off-diagonal)', plot_H_z=True)
off_diagoff_diagonal_DSS.plot_matching(matching_off_diag_DSS,
            title='Optimal Matching (off-diagonal) with DSS algorithm')
```



16.4.1 Verify with linear programming

Let's verify some of the proceeding findings using linear programming.

```
def solve_1to1(c_i_j, n_x, m_y, return_dual=False):
    n, m = np.shape(c_i_j)

    # Constraint matrix
    M_z_a = np.vstack([np.kron(np.eye(n), np.ones(m)),
                       np.kron(np.ones(n), np.eye(m))])
    # Constraint vector
    q = np.concatenate((n_x, m_y))

    # Solve the linear programming problem using linprog from scipy
    result = linprog(c_i_j.flatten(), A_eq=M_z_a, b_eq=q,
```

(continues on next page)

(continued from previous page)

```
                        bounds=(0, None), method='highs')

    if return_dual:
        return (np.round(result.x).astype(int).reshape([n, m]),
                result.eqlin.marginals)
    else:
        return np.round(result.x).astype(int).reshape([n, m])
```

```
mu_x_y_LP = solve_1to1(example_pb.cost_x_y,
                       example_pb.n_x,
                       example_pb.m_y)
print(f"Value of LP (scipy): {(mu_x_y_LP * example_pb.cost_x_y).sum()}")
print(f"Value (plain Bellman equations): {(matching * example_pb.cost_x_y).sum()}")
print(f"Value (DSS): {(matching_DSS * example_pb.cost_x_y).sum()}")
```

```
Value of LP (scipy): 143.45490363125705
Value (plain Bellman equations): 143.45490363125705
Value (DSS): 143.45490363125705
```

16.5 Examples

16.5.1 Example 1

We study optimal transport problems on the real line with cost $c(x, y) = h(|x - y|)$ for a strictly concave and increasing function $h : \mathbb{R}_+ \to \mathbb{R}_+$.

The outcome is called *composite sorting*.

Here, we will focus on $c(x, y) = |x - y|^{\frac{1}{\zeta}}$ for $\zeta > 1$

To appreciate differences with *positive assortative matching* (PAM) note that the latter is induced by a cost of the form $h(x - y)$ for some strictly convex $h : \mathbb{R} \to \mathbb{R}_+$.

See Santambrogio 2015, Ch. 2.2.

For example, the cost function $|x - y|^p, p > 1$ induces PAM.

On the other hand, *negative assortative matching* (NAM) arises if $c(x, y) = h(x - y)$ with $h : \mathbb{R} \to \mathbb{R}_+$ strictly concave.

For example, the cost function $-|x - y|^p, p > 1$, induces NAM.

Thus, NAM corresponds to a matching that *maximizes* a transport problem criterion with *gain* function $g(x, y) = |x-y|^p$.

Note how PAM and NAM differ from **composite sorting**

Composite sorting is induced by a cost that is the composition of a strictly concave increasing function h and a convex function $|\cdot|$ applied to displacement $x - y$.

Different functions h potentially induce different matchings.

The following example shows that composite matching can feature both positive and negative assortative patterns.

Suppose that there are two agents per side and types

$$x_0 < y_0 < x_1 < y_1$$

There are two feasible matchings, one corresponding to PAM, the other to NAM.

- The first features two displacements $|x_0 - y_0|, |x_1 - y_1|$
- The second features a large displacement $|x_0 - y_1|$ and a small displacement $|x_1 - y_0|$.

Evidently,

- PAM corresponds to the matching with two medium side displacement because the correponding cost is strictly convex and increasing in the the displacement.
- NAM corresponds to the matching with a small displacement and a large displacement because the gain is strictly convex and increasing in the displacement.

In this example, composite sorting ends up coinciding with NAM, but this is something of a coincidence

- Thus, note that in composite matching the cost function is strictly concave and increasing in the displacement.

```
N = 2
p = 2
ζ = 2

# Solve composite sorting problem
example_1 = ConcaveCostOT(np.array([0,5]),
                          np.array([4,10]),
                          ζ=ζ)
matching_CS, _ ,_ = example_1.solve_primal_DSS()

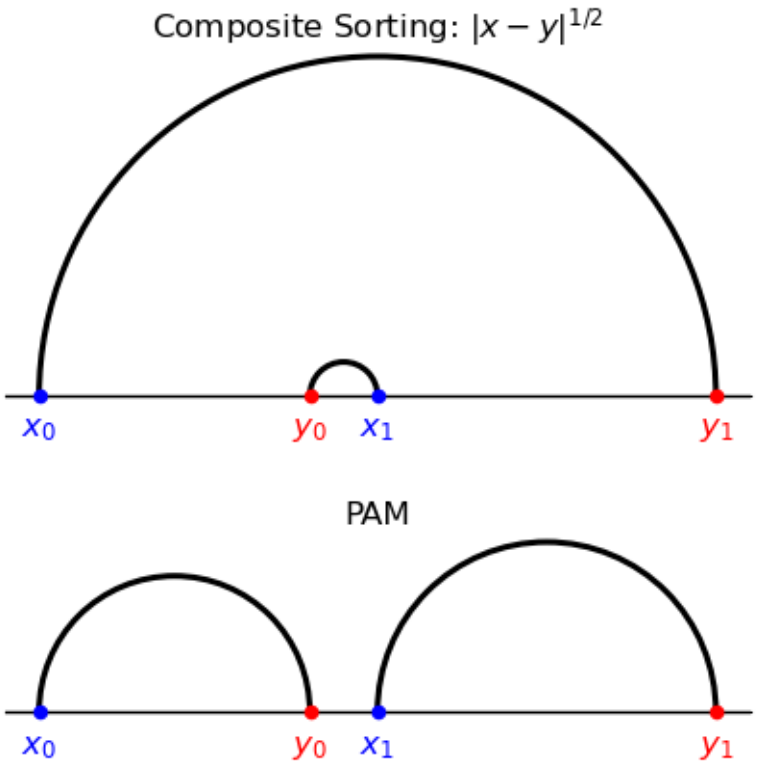
# Solve PAM and NAM
# I use the linear programs to compute PAM and NAM,
# but of course they can be computed directly

convex_cost = np.abs(example_1.X_types[:,None] - example_1.Y_types[None,:])**p

#PAM: |x-y|^p , p>1
matching_PAM = solve_1to1(convex_cost, example_1.n_x, example_1.m_y)

#NAM: -|x-y|^p , p>1
matching_NAM = solve_1to1(-convex_cost, example_1.n_x, example_1.m_y)

# Plot the matchings
example_1.plot_matching(matching_CS,
                title=f'Composite Sorting: $|x-y|^{{1/{ζ}}}$',
                figsize=(5,5), add_labels=True)
example_1.plot_matching(matching_PAM, title='PAM',
                figsize=(5,5), add_labels=True)
```



To explore the coincidental resemblence to a NAM outcome, let's shift left type y_0 while keeping it in between x_0 and x_1.

PAM and NAM are invariant to any such shift.

However, for a large enough shift, composite sorting now coindices with PAM.

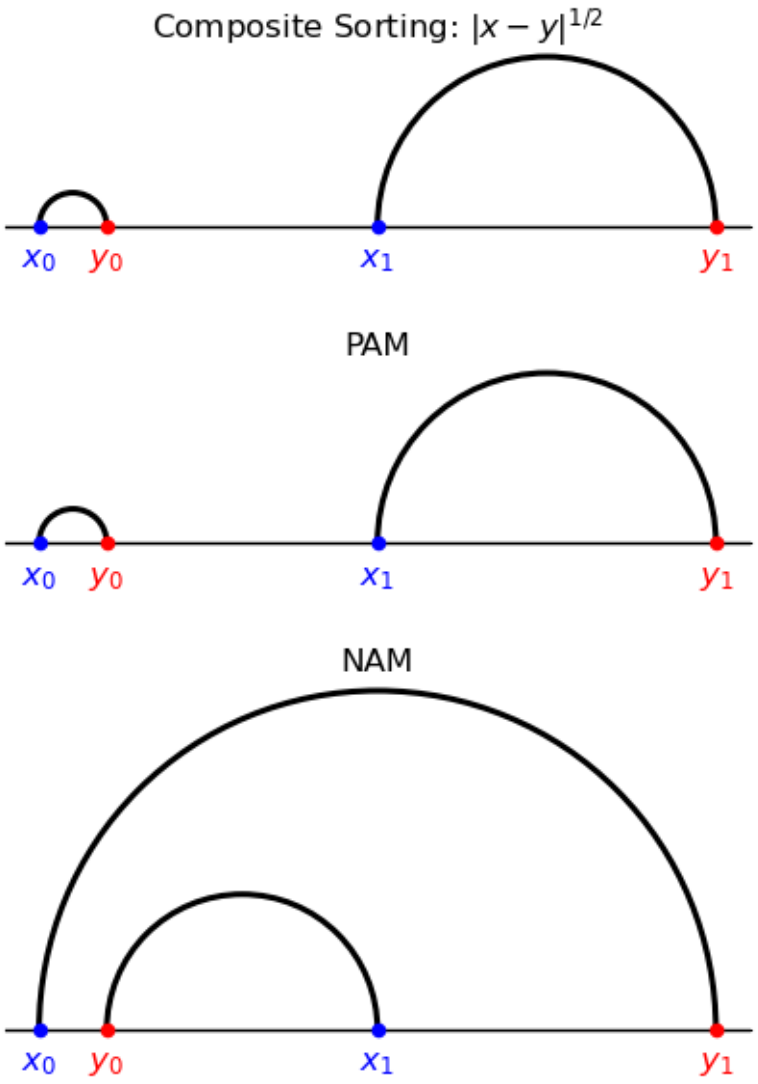
```
N = 2
ζ = 2
p = 2

# Solve composite sorting problem
example_1 = ConcaveCostOT(np.array([0,5]),
                          np.array([1,10]) ,
                           ζ = ζ)
matching_CS, _ ,_ = example_1.solve_primal_DSS()

# Solve PAM and NAM
convex_cost = np.abs(example_1.X_types[:,None] - example_1.Y_types[None,:])**p

matching_PAM = solve_1to1(convex_cost, example_1.n_x, example_1.m_y)
matching_NAM = solve_1to1(-convex_cost, example_1.n_x, example_1.m_y)

# Plot the matchings
example_1.plot_matching(matching_CS,
        title = f'Composite Sorting: $|x-y|^{{1/{ζ}}}$',
        figsize = (5,5), add_labels = True)
example_1.plot_matching(matching_PAM, title = 'PAM',
        figsize = (5,5), add_labels = True)
example_1.plot_matching(matching_NAM, title = 'NAM',
        figsize = (5,5), add_labels = True)
```



Finally, notice that the **Monge problem** cost function $|x-y|$ equals the limit of the composite sorting cost $|x-y|^{1/\zeta}$ as $\zeta \downarrow 1$ and also the limit of $|x-y|^p$ as $p \downarrow 1$.

Evidently, the Monge problem is solved by both the PAM and the composite sorting assignment that arises for $\zeta \downarrow 1$.

In the following example, the Monge cost of the composite sorting assignment equals the Monge cost of PAM.

Consequently, it is optimal for the Monge problem.

```
N = 10

ζ = 1.01
p = 2
np.random.seed(1)
X_types = np.random.uniform(0,10, size=N)
Y_types = np.random.uniform(0,10, size=N)

# Solve composite sorting problem
example_1 = ConcaveCostOT(X_types, Y_types, ζ=ζ)
```

(continues on next page)

(continued from previous page)

```
matching_CS, _ ,_ = example_1.solve_primal_DSS()

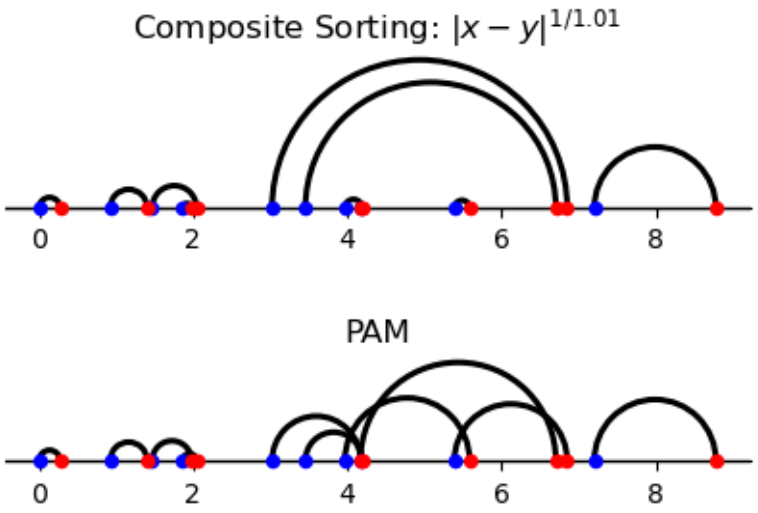
# Solve PAM and NAM
convex_cost = np.abs(X_types[:,None] - Y_types[None,:])** p

matching_PAM = solve_1to1(convex_cost, example_1.n_x, example_1.m_y)
matching_NAM = solve_1to1(-convex_cost, example_1.n_x, example_1.m_y)


example_1.plot_matching(matching_CS,
            title=f'Composite Sorting: $|x-y|^{{1/{ζ}}}$', figsize=(5,5))
example_1.plot_matching(matching_PAM, title = 'PAM', figsize=(5,5))

monge_cost_comp = (matching_CS * np.abs(X_types[:,None] - Y_types[None,:])).sum()
monge_cost_PAM = (matching_PAM * np.abs(example_1.X_types[:,None]
                                - example_1.Y_types[None,:])).sum()
print("Monge cost of the composite matching assignment:")
print(monge_cost_comp)
print("Monge cost of PAM:")
print(monge_cost_PAM)
```

```
Monge cost of the composite matching assignment:
10.530287849572634
Monge cost of PAM:
10.530287849572636
```



16.5.2 Example 2

The following example has five agents per side.

The composite sorting assignment differs from both PAM and NAM.

Composite sorting features a hierarchical structure, with each hierarchy positively sorted.

Indeed, consider the composite sorting assignment and note that

- the only arcs *visible from above* are the ones corresponding to pairings (x_0, y_3) and (y_4, x_4);
- after removing these agents, the only arcs visible from above correspond to (x_1, y_1) and (x_3, y_2) ;
- after removing these agents, the only arc/pairing left is (x_2, y_0).

Note that, at each iteration, the partial assignment corresponding to the arcs visible from above features positive assortativeness.

Another distinct feature of composite matching stands out from the figures:

- **arcs do not intersect**

```
N = 5
ζ = 2
p = 2

X_types_example_2 = np.array([-2,0,2,9, 15])
Y_types_example_2 = np.array([3,6,10,12, 14])

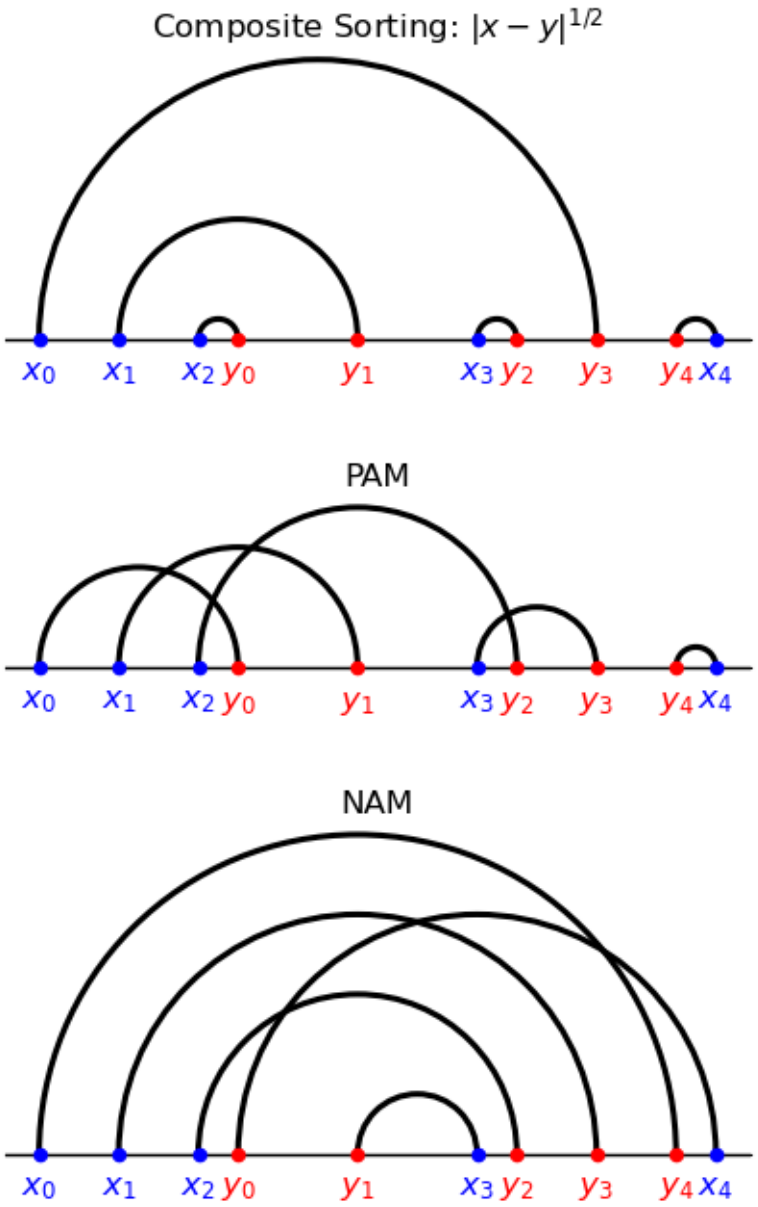
# Solve composite sorting problem
example_2 = ConcaveCostOT(X_types_example_2, Y_types_example_2, ζ=ζ)

matching_CS, _ ,_ = example_2.solve_primal_DSS()

# Solve PAM and NAM
convex_cost = np.abs(X_types_example_2[:,None] - Y_types_example_2[None,:])** p

matching_PAM = solve_1to1(convex_cost, example_2.n_x, example_2.m_y)
matching_NAM = solve_1to1(-convex_cost, example_2.n_x, example_2.m_y)


example_2.plot_matching(matching_CS, title = 'Composite Sorting: $|x-y|^{1/2}$',
                        figsize = (5,5), add_labels=True)
example_2.plot_matching(matching_PAM, title = 'PAM',
                        figsize = (5,5), add_labels=True)
example_2.plot_matching(matching_NAM, title = 'NAM',
                        figsize = (5,5), add_labels=True)
```



16.5.3 Example 3

[Boerma *et al.*, 2024] provide the following example.

There are four agents per side and three types per side (so the problem is not unitary, as opposed to the examples above).

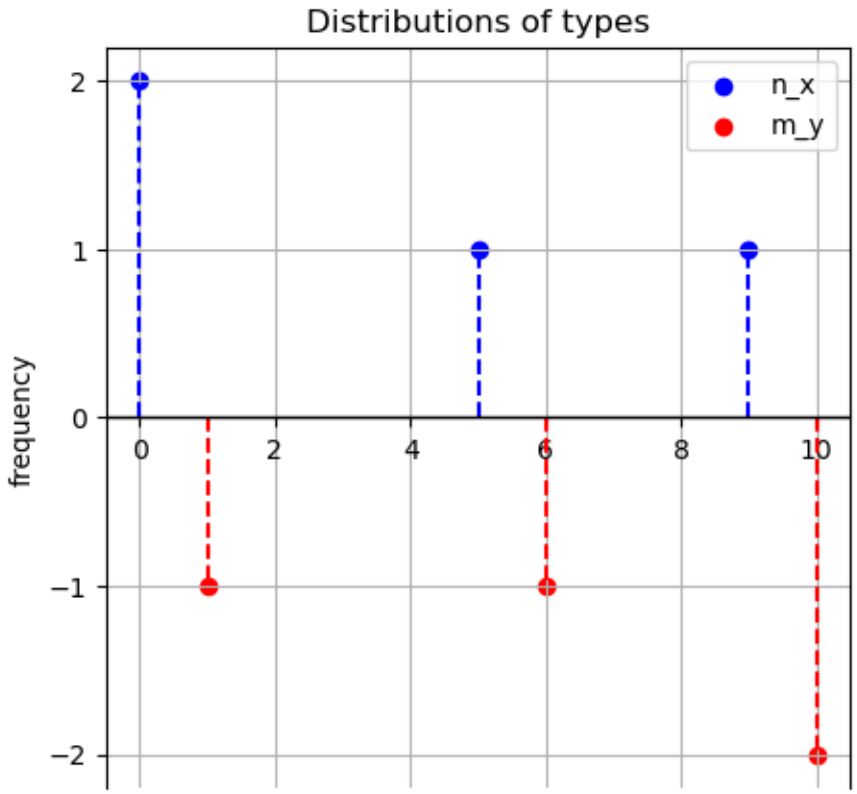
```
X_types_example_3 = np.array([0,5,9])
Y_types_example_3 = np.array([1,6,10])
n_x_example_3 =  np.array([2,1,1], dtype= int)
m_y_example_3 =  np.array([1,1,2], dtype= int)


example_3 = ConcaveCostOT(X_types_example_3, Y_types_example_3,
```

(continues on next page)

(continued from previous page)

```
                            n_x_example_3, m_y_example_3, ζ = 2)
example_3.plot_marginals(figsize = (5,5))
```



In the case of positive assortative matching (PAM), the two agents with lowest value x_0 are matched with the lowest valued agents on the other side y_0, y_1.

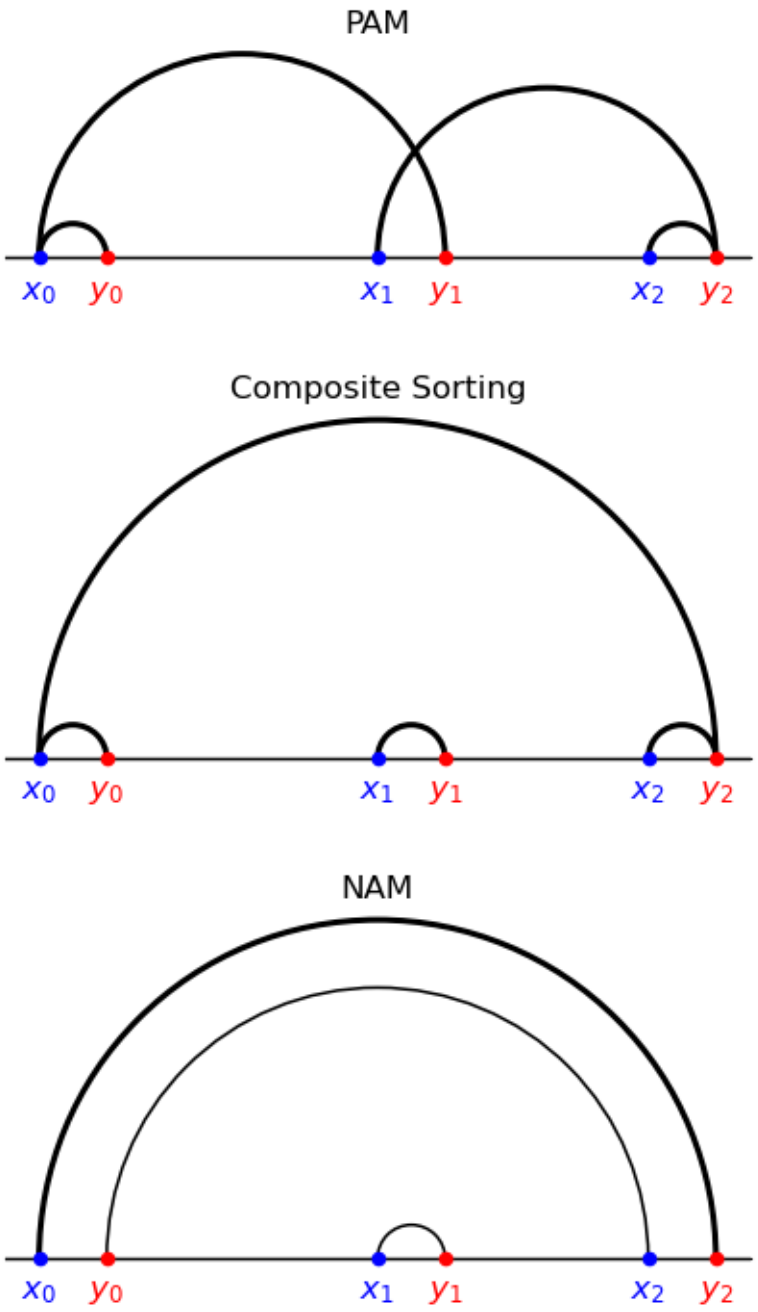
Similarly, the agents with highest value y_2 are matched with the highest valued types on the other side, x_1 and x_2.

Composite sorting features both negative and positive sorting patterns: agents of type x_0 are matched with both the bottom y_0 and the top y_2 of the distribution.

```
matching_CS, _ ,_ = example_3.solve_primal_DSS()

convex_cost = np.abs(example_3.X_types[:,None] - example_3.Y_types[None,:])**2
matching_PAM = solve_1to1(convex_cost, example_3.n_x, example_3.m_y)
matching_NAM = solve_1to1(-convex_cost, example_3.n_x, example_3.m_y)

example_3.plot_matching(matching_PAM, title = 'PAM',
                        figsize = (5,5), add_labels= True)
example_3.plot_matching(matching_CS, title = 'Composite Sorting',
                        figsize = (5,5), add_labels= True)
example_3.plot_matching(matching_NAM, title = 'NAM',
                        figsize = (5,5), add_labels= True)
```


PAM
x_0 y_0 x_1 y_1 x_2 y_2
Composite Sorting
x_0 y_0 x_1 y_1 x_2 y_2
NAM
x_0 y_0 x_1 y_1 x_2 y_2

16.6 Dual Solution

Let's recall the formulation

$$
\begin{aligned}
V_P = \min_{\mu \geq 0} & \sum_{(x,y) \in X \times Y} \mu_{xy} c_{xy} \\
\text{s.t.} & \sum_{x \in X} \mu_{xy} = n_x \\
& \sum_{y \in Y} \mu_{xy} = m_y
\end{aligned}
$$

The *dual problem* is

$$
\begin{aligned}
V_D = \max_{\phi, \psi} & \sum_{x \in X} n_x \phi_x + \sum_{y \in Y} m_y \psi_y \\
\text{s.t. } & \phi_x + \psi_y \leq c_{xy}
\end{aligned}
$$

where (ϕ, ψ) are dual variables, which can be interpreted as shadow cost of agents in X and Y, respectively.

Since the dual is feasible and bounded, $V_P = V_D$ (*strong duality* prevails).

Assume now that $y_{xy} = \alpha_x + \gamma_y - c_{xy}$ is the output generated by matching x and y.

It includes the sum of x and y specific amenities/outputs minus the cost c_{xy}.

Then we can formulate the following problem and its dual

$$
\begin{aligned}
W_P = \max_{\mu \geq 0} & \sum_{(x,y) \in X \times Y} \mu_{xy} y_{xy} \\
\text{s.t.} & \sum_{x \in X} \mu_{xy} = n_x \\
& \sum_{y \in Y} \mu_{xy} = m_y
\end{aligned}
$$

$$
\begin{aligned}
W_D = \min_{u, v} & \sum_{x \in X} n_x u_x + \sum_{y \in Y} m_y v_y \\
\text{s.t. } & u_x + v_y \geq y_{xy}
\end{aligned}
$$

Given the constraints, the primal problem W_P does not depend on α, γ and it has the same solutions as the cost minimization problem V_P.

The values are related by $W_P = \sum_{x \in X} n_x \alpha_x + \sum_{y \in Y} m_y \gamma_y - V_P$.

The dual solutions of V_D and W_D are related by $u_x = \alpha_x - \phi_x$ and $v_y = \gamma_y - \psi_y$.

The dual solution (u, v) of W_D can be interpreted as equilibrium utilities of the agents, which include the individual specific amenities and equilibrium shadow costs.

[Boerma *et al.*, 2024] propose an efficient method to compute the dual variables from the optimal matching (primal solution) in the case of composite sorting.

Let's generate an instance and compute the optimal matching.

```
num_agents = 8

np.random.seed(1)

X_types_assignment_pb = np.random.uniform(0, 10, size=num_agents)
```

(continues on next page)

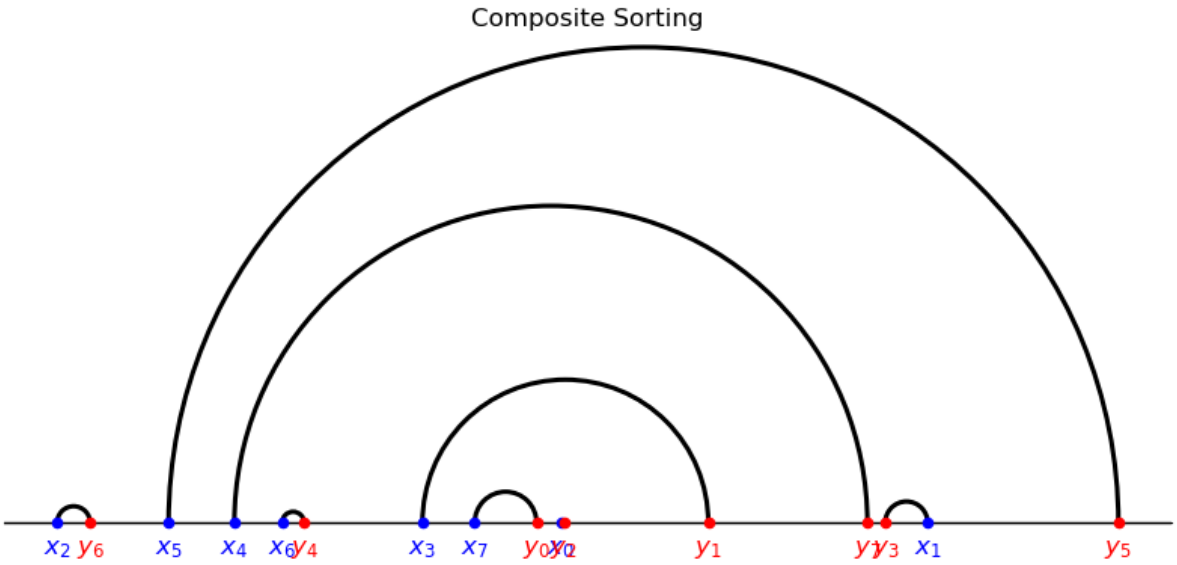
(continued from previous page)

```
Y_types_assignment_pb = np.random.uniform(0, 10, size=num_agents)


# Create instance of the problem
exam_assign = ConcaveCostOT(X_types_assignment_pb, Y_types_assignment_pb)

# Solve primal problem
assignment, assignment_OD, exam_assign_OD = exam_assign.solve_primal_DSS()

# Plot matching
add_labels = True if num_agents < 16 else False
exam_assign_OD.plot_matching(assignment_OD, title = f'Composite Sorting',
                             figsize=(10,10), add_labels=add_labels)
```



Having computed the optimal matching, we say that a pair (x_0, y_0) is a *subpair* of a matched pair (x, y) if x_0, y_0 are in the open interval between x and y and the pair (x_0, y_0) is not nested.

The following method computes the subpairs of the optimal matching of the off-diagonal instance.

The output of this method is a dictionary with keys corresponding to matched pairs and an "artificial pair" which collects all arcs which are visible from above.

Values of each key (x_0, y_0) are the subpairs ordered so that the first subpair is the subpair with the x type closest to x_0 and the last subpair is the subpair with the y type closest to y_0.

```
def sort_subpairs(self, subpairs, x_smaller_y=True ):

    x_key = min if x_smaller_y else max
    y_key = max if x_smaller_y else min

    first_pair = x_key(subpairs, key=lambda pair: self.X_types[pair[0]])
    last_pair = y_key(subpairs, key=lambda pair: self.Y_types[pair[1]])

    intermediate_pairs = [pair for pair in subpairs
```

(continues on next page)

(continued from previous page)

```
                            if pair != first_pair and pair != last_pair]

    return [first_pair] + intermediate_pairs + [last_pair]

ConcaveCostOT.sort_subpairs = sort_subpairs

def find_subpairs(self, matching, return_pairs_between = False):

    # Create set of matched pairs of types and add an artificial pair
    matched_pairs = set( zip(* np.where(matching > 0)))

    # Initialize dictionary to store subpairs
    subpairs = {}
    pairs_between = {}

    # Find subpairs (both nested and non-nested) for each matched pair
    for matched_pair in matched_pairs | {'artificial_pair'}:
        # Determine the interval of the matched pair
        if matched_pair != 'artificial_pair':
            min_type, max_type = sorted([self.X_types[matched_pair[0]],
                                         self.Y_types[matched_pair[1]]])
        else:
            min_type, max_type = (-np.inf, np.inf)

        # Add all pairs in the interval to the list of nested_subpairs
        pairs_between[matched_pair] = {
                    pair for pair in matched_pairs if pair != matched_pair and
                    min_type <= self.X_types[pair[0]] <= max_type and
                    min_type <= self.Y_types[pair[1]] <= max_type}

    subpairs = {key: value.copy() for key, value in pairs_between.items()}

    # Remove nested pairs
    for matched_pair in matched_pairs | {'artificial_pair'}:
        # Compute all nested subpairs
        nested_subpairs = set(chain.from_iterable(subpairs[pair]
                                  for pair in subpairs[matched_pair]))
        # Remove nested pairs from subpairs[matched_pair]
        subpairs[matched_pair] -= nested_subpairs
        # subpairs[matched_pair].discard(matched_pair)
        subpairs[matched_pair] = list(subpairs[matched_pair])

        # Order the subpairs:
        # the first (last) pair should have x (y) close to pair_x (pair_y)
        if matched_pair != 'artificial_pair' and len(subpairs[matched_pair]) > 1:
            subpairs[matched_pair] = self.sort_subpairs(
                subpairs[matched_pair],
                x_smaller_y=self.X_types[matched_pair[0]]
                            < self.Y_types[matched_pair[1]])

    if return_pairs_between:
        return subpairs, pairs_between
    return subpairs

OffDiagonal.find_subpairs = find_subpairs
```

```
subpairs, pairs_between = exam_assign_OD.find_subpairs(assignment,
                                                       return_pairs_between = True)
subpairs
```

```
{(5, 5): [(4, 7), (1, 3)],
 (3, 1): [(7, 0), (0, 2)],
 (7, 0): [],
 (6, 4): [],
 (0, 2): [],
 'artificial_pair': [(5, 5), (2, 6)],
 (2, 6): [],
 (1, 3): [],
 (4, 7): [(6, 4), (3, 1)]}
```

The algorithm to compute the dual variables has a hierarchical structure: it starts from the matched pairs with no subpairs and then moves to those pairs whose subpairs have been already processed.

We can visualize the hierarchical structure by computing the order in which he pairs will be processed and plotting the matching with color of the arcs corresponding the hierarchy.

```
## Compute Hierarchies

def find_hierarchies(subpairs):

    # Initialize sets for faster membership checks
    pairs_to_process = set(subpairs.keys())  # All pairs to process
    processed_pairs = set()  # Pairs that have been processed

    # Initialize ready_to_process with pairs that have no subpairs
    ready_to_process = {pair for pair, sublist in subpairs.items()
                        if len(sublist) == 0}

    # Initialize hierarchies with the first level
    hierarchies = [list(ready_to_process)]

    # Continue processing while there are unprocessed pairs
    while len(processed_pairs) < len(subpairs):
        # Mark ready_to_process pairs as processed
        processed_pairs.update(ready_to_process)

        # Remove ready_to_process pairs from pairs_to_process
        pairs_to_process -= ready_to_process

        # Find new ready_to_process pairs that have all their subpairs processed
        ready_to_process = {
            pair for pair in pairs_to_process
            if all(subpair in processed_pairs for subpair in subpairs[pair])}

        # Append the new ready_to_process to hierarchies
        hierarchies.append(list(ready_to_process))

    return hierarchies

## Plot Hierarchies

def plot_hierarchies(self, subpairs, scatter=True, range_x_axis=None):
```

(continues on next page)

(continued from previous page)

```
    # Compute hierarchies
    hierarchies = find_hierarchies(subpairs)

    # Create the figure and axis
    fig, ax = plt.subplots(figsize=(15, 15))

    # Plot types on the real line (blue for X_types, red for Y_types)
    size_marker = 20 if scatter else 0
    ax.scatter(self.X_types, np.zeros_like(self.X_types), color='blue',
                    s=size_marker, zorder=5, label='X_types')
    ax.scatter(self.Y_types, np.zeros_like(self.Y_types), color='red',
                    s=size_marker, zorder=5, label='Y_types')

    # Plot arcs
    # Create a colormap ('viridis' or 'coolwarm', 'plasma')
    cmap = plt.colormaps['plasma']
    for level, hierarchy in enumerate(hierarchies):
        color = (cmap(level / (len(hierarchies) - 1))
                        if len(hierarchies) > 1 else cmap(0))
        for pair in hierarchy:
            if pair == 'artificial_pair':
                continue

            min_type, max_type = sorted([self.X_types[pair[0]],
                                         self.Y_types[pair[1]]])
            width = max_type - min_type
            center = (max_type + min_type) / 2
            # Semicircle height can be the same as the width for a perfect arc
            height = width
            semicircle = patches.Arc((center, 0), width, height,
                            theta1=0, theta2=180,
                            color=color, lw = 3)
            ax.add_patch(semicircle)

    if range_x_axis is not None:
        ax.set_xlim(range_x_axis)
        ax.set_ylim(- self.X_types.ptp() / 10,
                    (range_x_axis[1] - range_x_axis[0]) / 2 )

    # Title and layout settings for the main plot
    plt.title('Hierarchies of the optimal matching (off-diagonal)')
    ax.set_aspect('equal')
    plt.axhline(0, color='black', linewidth=1)
    ax.spines['bottom'].set_position(('data', 0))
    ax.spines['left'].set_color('none')
    ax.spines['top'].set_color('none')
    ax.spines['right'].set_color('none')
    ax.yaxis.set_ticks([])  # Hide the y-axis ticks

    # Add a colorbar to represent hierarchy levels
    sm = cm.ScalarMappable(cmap=cmap,
                    norm=Normalize(vmin=0, vmax= len(hierarchies) - 1))
    sm.set_array([])
    cbar = plt.colorbar(sm, ax=ax, orientation='vertical', pad=0.1, shrink=0.2)
    # Show only min and max levels
    cbar.set_ticks([0, len(hierarchies) - 1])
```

(continues on next page)

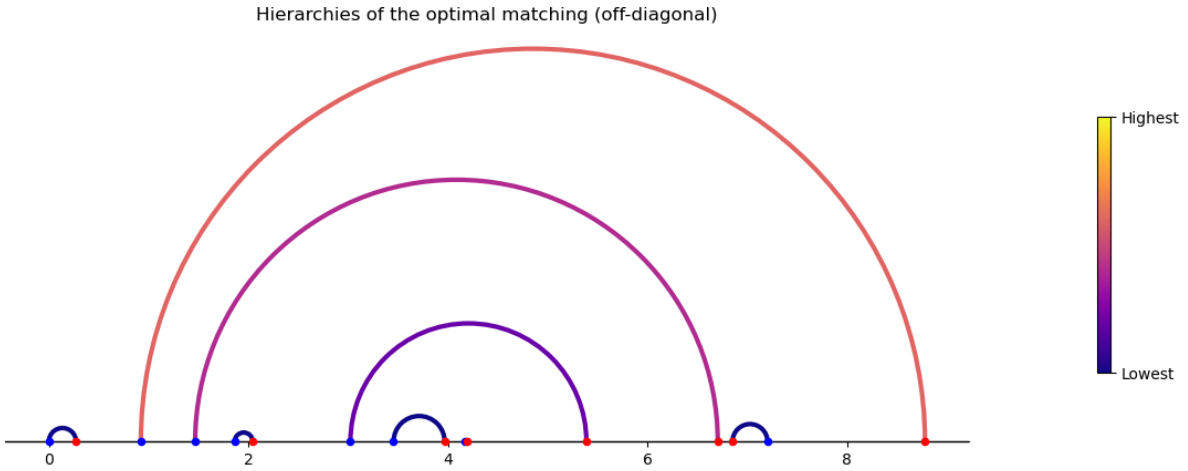
(continued from previous page)

```
    # Label the ticks for clarity
    cbar.set_ticklabels(['Lowest', 'Highest'])

    plt.show()

OffDiagonal.plot_hierarchies = plot_hierarchies
```

```
exam_assign_OD.plot_hierarchies(subpairs)
```



We proceed to describe and implement the algorithm to compute the dual solution.

As already mentioned, the algorithm starts from the matched pairs (x_0, y_0) with no subpairs and assigns the (temporary) values $\psi_{x_0} = c_{x_0 y_0}$ and $\psi_{y_0} = 0$, i.e. the x type sustains the whole cost of matching.

The algorithm then proceeds sequentially by processing any matched pair whose subpairs have already been processed.

After picking any such matched pair (x_0, y_0), the dual variables already computed for the processed subpairs need to be made "comparable".

Indeed, for any subpair (x_1, y_1) of (x_0, y_0), the dual variables of all the types between the x_1 and y_1 satisfy dual feasibility and complementary slackness *locally*, i.e. $\phi_x + \psi_y \leq c_{xy}$ with equality if (x, y) is a matched pair for all types x, y between x_0 and y_0.

But dual feasibility is not satisfied globally in general, for instance it might not be satisfied for two subpairs (x_1, y_1) and (x_2, y_2) of (x_0, y_0).

Therefore, letting $(x_1, y_1), \dots, (x_p, y_p)$ be the subpairs of (x_0, y_0), we compute the solution $(\beta_2, \dots, \beta_p)$ of the linear system

$$\max(c_{x_0 y_0} - c_{x_0 y_i} - c_{x_j y_0}, -c_{x_j y_i}) + c_{x_i y_i} \leq \sum_{k=i+1}^{j} \beta_k \leq \min(c_{x_0 y_j} + c_{x_i y_0} - c_{x_0 y_0}, c_{x_i y_j}) - c_{x_j y_j}, \quad \text{for all } 1 \leq i < j \leq p.$$

Then for all $i \in [p]$ compute the adjustment $\Delta_i = \sum_{k=i+1}^{p} \beta_k + \phi_{x_p} - \phi_{x_1}$ and modify the dual variables

$$\begin{aligned} \phi_x &\leftarrow \phi_x + \Delta_i \\ \psi_y &\leftarrow \psi_y - \Delta_i, \end{aligned}$$

for all matched pairs (x, y) between x_i and y_i.

After this step, the dual variables of the types between x_0 and y_0 satisfy dual feasibility and complementary slackness; we can then proceed to compute the dual variables for x_0 and y_0 by setting

$$\psi_{y_0} = \min_{i \in [p]} \{c_{x_i y_0} - \phi_{x_i}\}$$
$$\phi_{x_0} = c_{x_0 y_0} - \psi_{y_0}.$$

The pair (x_0, y_0) is now processed.

The following method computes the solution β of the linear system of inequalities above.

```
def compute_betas(self, pair, subpairs):
    types_subpairs = np.array(subpairs)

    # Define the bounds of the linear inequality system
    if pair == 'artificial_pair':
        bounds = (- self.cost_x_y[types_subpairs[:,0][:,None],
                 types_subpairs[:,1][None,:]]
                 + self.cost_x_y[types_subpairs[:,0],
                                 types_subpairs[:,1]][None,:])
    else:
        bounds = (np.maximum(
        self.cost_x_y[pair]
        - self.cost_x_y[pair[0], types_subpairs[:,1]][None,:]
        - self.cost_x_y[types_subpairs[:,0],pair[1]][:,None],
        - self.cost_x_y[types_subpairs[:,0][:,None],
        types_subpairs[:,1][None,:]]
        )
        + self.cost_x_y[types_subpairs[:,0], types_subpairs[:,1]][None,:])

    # Define linear inequality system
    num_subpairs = len(types_subpairs)
    c_1 = (np.arange(num_subpairs)[:, None, None]
            >= np.arange(num_subpairs)[None, None, :])
    c_2 = (np.arange(num_subpairs)[None, None, :]
            > np.arange(num_subpairs)[ None,:, None])
    sum_tensor = (c_1 & c_2).astype(int)

    sum_tensor -= sum_tensor.transpose(1, 0, 2)

    # Solve the system of linear inequalities
    result = linprog(c = np.zeros(num_subpairs),
                     A_ub= - sum_tensor.reshape(num_subpairs**2, num_subpairs),
                     b_ub= - bounds.flatten(),
                     bounds=(None,None),
                     method='highs')

    beta = result.x
    beta[0] = 0

    return beta

OffDiagonal.compute_betas = compute_betas
```

The following method iteratively processes the matched pairs of the off-diagonal matching as explained above.

```
def compute_dual_off_diagonal(self, subpairs, pairs_between):

```

(continues on next page)

(continued from previous page)

```
    # Initialize dual variables
    ϕ_x = np.zeros(len(self.X_types))
    ψ_y = np.zeros(len(self.Y_types))

    # Initialize sets for faster membership checks
    pairs_to_process = set(subpairs.keys())  # All pairs to process
    processed_pairs = set()  # Pairs that have been processed

    # Initialize ready_to_process with pairs that have no subpairs
    ready_to_process = {pair for pair, sublist in subpairs.items()
                        if len(sublist) == 0}

    while len(processed_pairs) < len(subpairs):

        # 1. Pick any subpair which is ready to process
        for pair in ready_to_process:

            # 2. If there are no subpairs, ψ_x = c_{xy} and ψ_y = 0
            if len(subpairs[pair]) == 0:
                ϕ_x[pair[0]] = self.cost_x_y[pair]
                ψ_y[pair[1]] = 0

            # 3. If there are subpairs:
            else:
                # (a) compute betas
                beta = self.compute_betas(pair, subpairs[pair])

                # (b) adjust potentials of types between each subpair of the pair
                for i, subpair in enumerate(subpairs[pair]):
                    # update potentials of these types
                    types_between_subpair = np.array(
                                list(pairs_between[subpair]) + [subpair])

                    Δ_subpair = (beta[np.arange(i+1,len(subpairs[pair]))].sum()
                                 + ϕ_x[subpairs[pair][-1][0]]
                                 - ϕ_x[subpair[0]])

                    ϕ_x[ types_between_subpair[:,0]] += Δ_subpair
                    ψ_y[ types_between_subpair[:,1]] -= Δ_subpair

                # (c) compute potentials of the pair
                subpairs_x = np.array(subpairs[pair])[:,0]
                subpairs_y = np.array(subpairs[pair])[:,1]

                if pair != 'artificial_pair':
                    if pair[0] == subpairs_x[0]:
                        ψ_y[pair[1]] = np.min(self.cost_x_y[pair[0], subpairs_y]
                                    - ψ_y[subpairs_y]) + self.cost_x_y[pair]
                    else:
                        ψ_y[pair[1]] = np.min(self.cost_x_y[subpairs_x,
                                         pair[1]] - ϕ_x[subpairs_x] )


                    ϕ_x[pair[0]] =  self.cost_x_y[pair] - ψ_y[pair[1]]

            # Add pair to processed pairs
```

(continues on next page)

(continued from previous page)

```
            processed_pairs.add(pair)

        # Remove ready_to_process from pairs_to_process
        pairs_to_process -= ready_to_process

        # Add to ready_to_process pairs for which all subpairs are in processed_pairs
        ready_to_process = {pair for pair in pairs_to_process
            if all(subpair in processed_pairs for subpair in subpairs[pair])}

    return ϕ_x, ψ_y

OffDiagonal.compute_dual_off_diagonal = compute_dual_off_diagonal
```

We apply the algorithm to our example and check that dual feasibility ($\phi_x + \psi_y \leq c_{xy}$ for all $x \in X$ and $y \in Y$) as well as strong duality ($V_P = V_D$) are satisfied.

```
ϕ_x , ψ_y = exam_assign_OD.compute_dual_off_diagonal(subpairs, pairs_between)

# Check dual feasibility
dual_feasibility_i_j = ϕ_x[:,None] + ψ_y[None,:] - exam_assign_OD.cost_x_y
print('Violations of dual feasibility:' , np.sum(dual_feasibility_i_j > 1e-10))

dual_sol = (exam_assign_OD.n_x * ϕ_x).sum() + (exam_assign_OD.m_y* ψ_y).sum()
primal_sol = (assignment_OD * exam_assign_OD.cost_x_y).sum()

# Check strong duality
print('Value of dual solution: ', dual_sol)
print('Value of primal solution: ', primal_sol)

# # Check the value of the primal problem
if len(exam_assign_OD.n_x) * len(exam_assign_OD.m_y) < 1000:
    mu_x_y , p_z= solve_1to1(exam_assign_OD.cost_x_y,
                             exam_assign_OD.n_x,
                             exam_assign_OD.m_y,
                             return_dual = True)
    print('Value of primal solution (scipy)',
    (mu_x_y * exam_assign_OD.cost_x_y).sum())
```

```
Violations of dual feasibility: 0
Value of dual solution:  9.03369035213102
Value of primal solution:  9.03369035213102
Value of primal solution (scipy) 9.03369035213102
```

Having computed the dual variables of the off-diagonal types, we compute the dual variables for perfecly matched pairs by setting

$$\phi_x = \min_{y \in Y^{OD}} \{c_{xy} - \psi_y\}$$
$$\psi_y = \min_{x \in X^{OD}} \{c_{xy} - \phi_x\}$$

where X^{OD} and Y^{OD} are the types of the off-diagonal instance, for which the dual variables have already been computed.

The following method computes the full dual solution from the primal solution.

```
def compute_dual_solution(self, matching_off_diag):

    # Compute the dual solution for the off-diagonal types
    off_diag, match_tuple = self.generate_offD_onD_matching()
    nonzero_id_x, nonzero_id_y, matching_diag = match_tuple

    subpairs, pairs_between = off_diag.find_subpairs(matching_off_diag,
                                                     return_pairs_between = True)
    ϕ_x_off_diag, ψ_x_off_diag = off_diag.compute_dual_off_diagonal(
                                        subpairs,pairs_between)

    # Compute the dual solution for the on-diagonal types
    ϕ_x = np.ones(len(self.X_types)) * np.inf
    ψ_y = np.ones(len(self.Y_types)) * np.inf

    ϕ_x[nonzero_id_x] = ϕ_x_off_diag
    ψ_y[nonzero_id_y] = ψ_x_off_diag

    ϕ_x = np.min( self.cost_x_y - ψ_y[None,:] , axis = 1)
    ψ_y = np.min( self.cost_x_y - ϕ_x[:,None] , axis = 0)

    return ϕ_x, ψ_y

ConcaveCostOT.compute_dual_solution = compute_dual_solution
```

```
ϕ_x, ψ_y = exam_assign.compute_dual_solution(assignment_OD)

dual_feasibility_i_j = ϕ_x[:,None] + ψ_y[None,:] - exam_assign.cost_x_y
print('Violations of dual feasibility:' , np.sum(dual_feasibility_i_j > 1e-10))
print('Value of dual solution: ', (exam_assign.n_x * ϕ_x).sum()
                                 + (exam_assign.m_y * ψ_y).sum())
print('Value of primal solution: ', (assignment * exam_assign.cost_x_y).sum())
```

```
Violations of dual feasibility: 0
Value of dual solution:  9.03369035213102
Value of primal solution:  9.03369035213102
```

16.7 Application

16.7.1 Data

We now replicate the empirical analysis carried out by [Boerma *et al.*, 2024].

The dataset is obtained from the American Community Survey and contains individual level data on income, age and occupation.

The occupation of each individual consists of a Standard Occupational Classification (SOC) code.

There are 497 codes in total.

We consider only employed (civilian) individuals with ages between 25 and 60 from 2010 to 2017.

To visualize log-wage dispersion, we group the individuals by occupation and compute the mean and standard deviation of the wages within each occupation.

Then we sort occupations by average log-earnings within each occupation.

The resulting dataset is included in the dataset `acs_data_summary.csv`

```
data_path = '_static/lecture_specific/match_transport/'
occupation_df = pd.read_csv(data_path + 'acs_data_summary.csv')
```

We plot the wage standard deviation for the sorted occupations.

```
# Scatter plot wage dispersion for each occupation
plt.figure(figsize=(10, 6))

# Scatter plot with marker size proportional to count
plt.scatter(
    occupation_df.index,
    occupation_df['std_Earnings'],
    # marker_sizes
    s = 1000 * (occupation_df['count'] / occupation_df['count'].max()),
    # transparency
    alpha = 0.5,
    label = 'Occupations'
)

# Polynomial interpolation
x = np.arange(len(occupation_df))
y = occupation_df['std_Earnings']
degree = 5
p = np.poly1d(np.polyfit(x, y, degree)  )
plt.plot(x, p(x), color='red')

# Add labels and title
plt.xlabel("Occupations", fontsize=12)
plt.ylabel("Wage Dispersion", fontsize=12)
plt.xticks([], fontsize=8)

plt.show()
```

We also plot the average wages for each occupation (SOC code). Again, occupations are ordered by increasing average wage.

```
# Scatter plot average wage for each occupation
plt.figure(figsize=(10, 6))

# Scatter plot with marker size proportional to count
plt.scatter(
    occupation_df.index,
    occupation_df['mean_Earnings'],
    alpha = 0.5, # transparency
    label = 'Occupations'
)

# Polynomial interpolation
x = np.arange(len(occupation_df))
y = occupation_df['mean_Earnings']
degree = 5
p = np.poly1d(np.polyfit(x, y, degree)  )
```

(continues on next page)

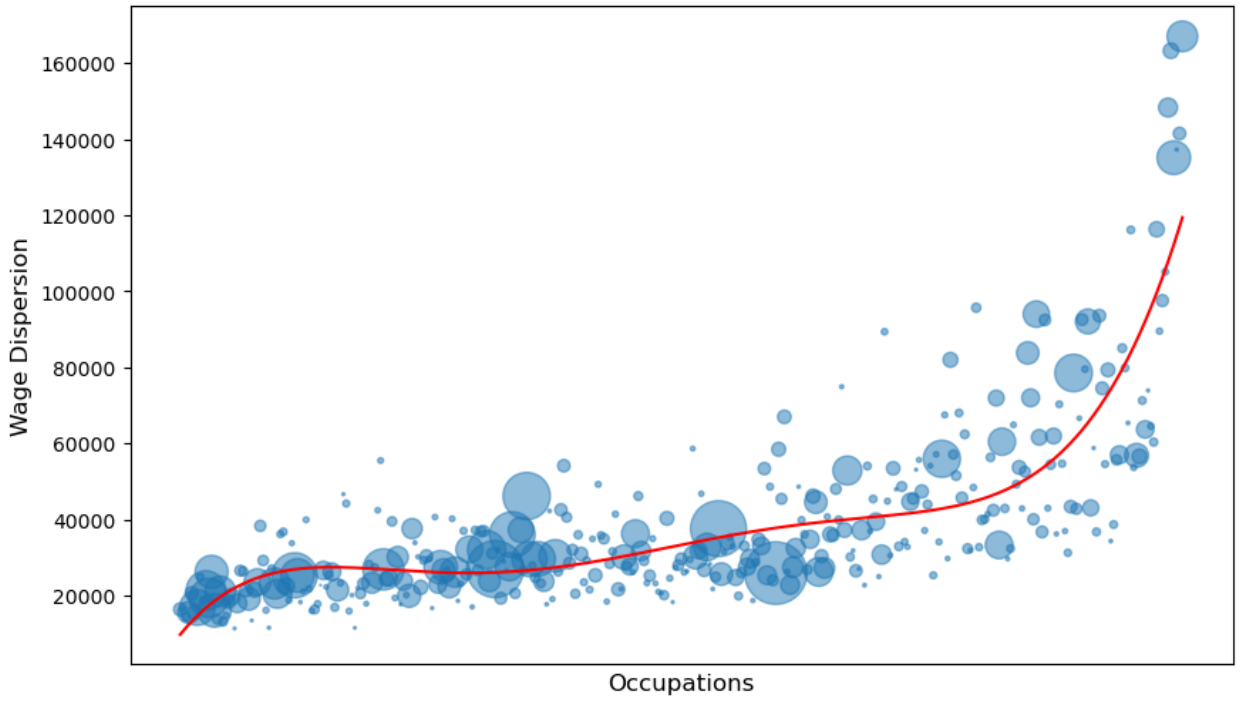


Fig. 16.1: Average wage for each Standard Occupational Classification (SOC) code. The codes are sorted by average wage on the horizontal axis. In red, a polynomial of degree 5 is fitted to the data. The size of the marker is proportional to the number of individuals in the occupation.

(continued from previous page)

```
plt.plot(x, p(x), color='red')

# Add labels and title
plt.xlabel("Occupations", fontsize=12)
plt.ylabel("Average Wage", fontsize=12)
plt.xticks([], fontsize=8)

plt.show()
```

16.7.2 Model

```
parameters_1980 = namedtuple('Params_Jobs', [
    'mean_1', 'var_1', 'mean_2', 'var_2', 'mixing_weight', 'var_workers'
])(
    mean_1=0.38,
    var_1=0.06,
    mean_2=0.0,
    var_2=0.75,
    mixing_weight=0.36,
    var_workers=0.2
)
```

(continues on next page)

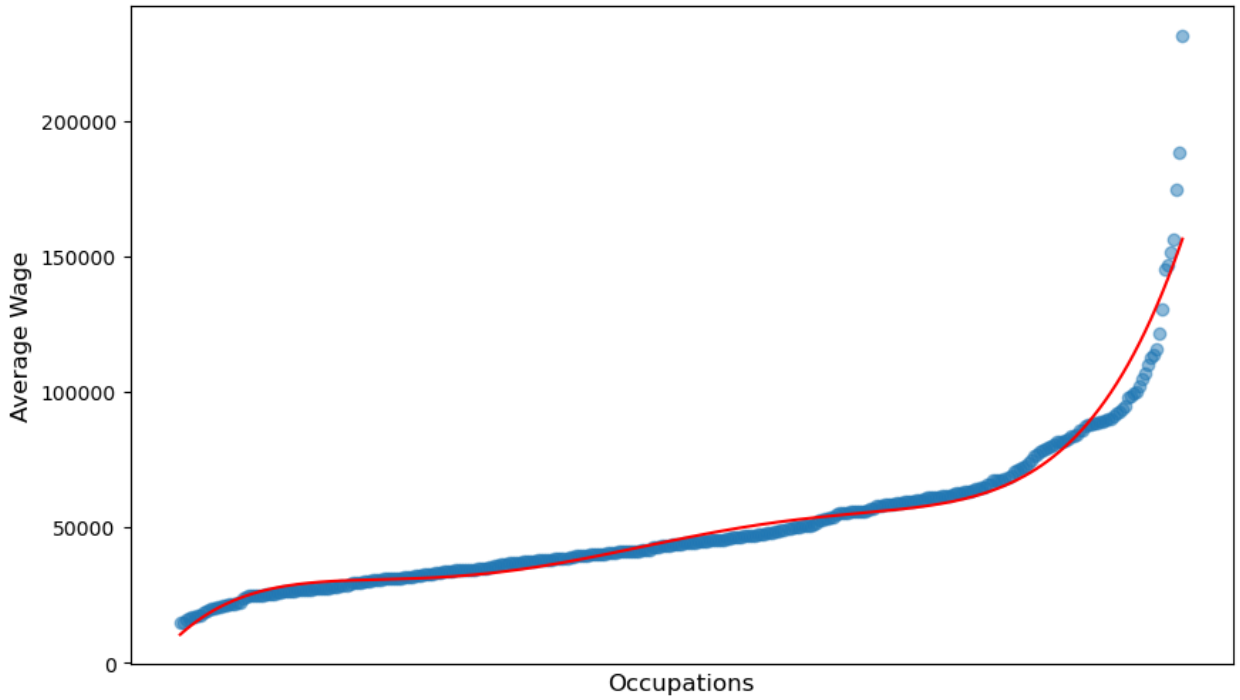


Fig. 16.2: Average wage for each Standard Occupational Classification (SOC) code. The codes are sorted by average wage on the horizontal axis. In red, a polynomial of degree 5 is fitted to the data.

(continued from previous page)

```
num_agents=1500
```

```
def generate_types_application(self, num_agents, params, random_seed=1):

    mean_1, var_1, mean_2, var_2, mixing_weight, var_workers = params

    np.random.seed(random_seed)

    # Job types
    job_types = np.where(np.random.rand(num_agents) < mixing_weight,
                    np.random.lognormal(mean_1, var_1, num_agents),
                    np.random.lognormal(mean_2, var_2, num_agents))

    # Worker types
    mean_workers = - var_workers/ 2
    worker_types = np.random.lognormal(mean_workers, var_workers, num_agents)

    # Check that worker and job types have distinct values
    assert len(np.unique(worker_types)) == num_agents
    assert len(np.unique(job_types)) == num_agents

    # Assign types to the instance
    self.X_types = worker_types
    self.Y_types = job_types
```

(continues on next page)

(continued from previous page)

```
    # Assign unitary marginals
    self.n_x = np.ones(num_agents, dtype=int)
    self.m_y = np.ones(num_agents, dtype=int)

    # Assign cost matrix
    self.cost_x_y = np.abs(worker_types[:, None] \
                    - job_types[None, :]) ** (1/self.ζ)

ConcaveCostOT.generate_types_application = generate_types_application

# Create an instance of ConcaveCostOT class and generate types
model_1980 = ConcaveCostOT()
model_1980.generate_types_application(num_agents, parameters_1980)
```

Since we will consider examples with a large number of agents, it will be convenient to visualize the distributions as histograms approximating the pdfs.

```
def plot_marginals_pdf(self, bins, figsize=(15, 8),
                       range_x_axis=None, title='Distributions of types'):

    plt.figure(figsize=figsize)

    # Plotting histogram for X_types (approximating PDF)
    plt.hist(self.X_types, bins=bins, density=True, color='blue', alpha=0.7,
             label='PDF of worker types',
             edgecolor='blue', range = range_x_axis)

    # Plotting histogram for Y_types (approximating PDF)
    counts, edges = np.histogram(self.Y_types, bins=bins,
                                 density=True, range=range_x_axis)
    plt.bar(edges[:-1], -counts, width=np.diff(edges), color='red', alpha=0.7,
            label='PDF of job types ', align='edge', edgecolor='red')

    # Add grid and y=0 axis
    plt.grid(False)
    plt.axhline(0, color='black', linewidth=1)
    plt.gca().spines['bottom'].set_position(('data', 0))

    # Set the x-axis limits based on the range argument
    if range_x_axis is not None:
        plt.xlim(range_x_axis)

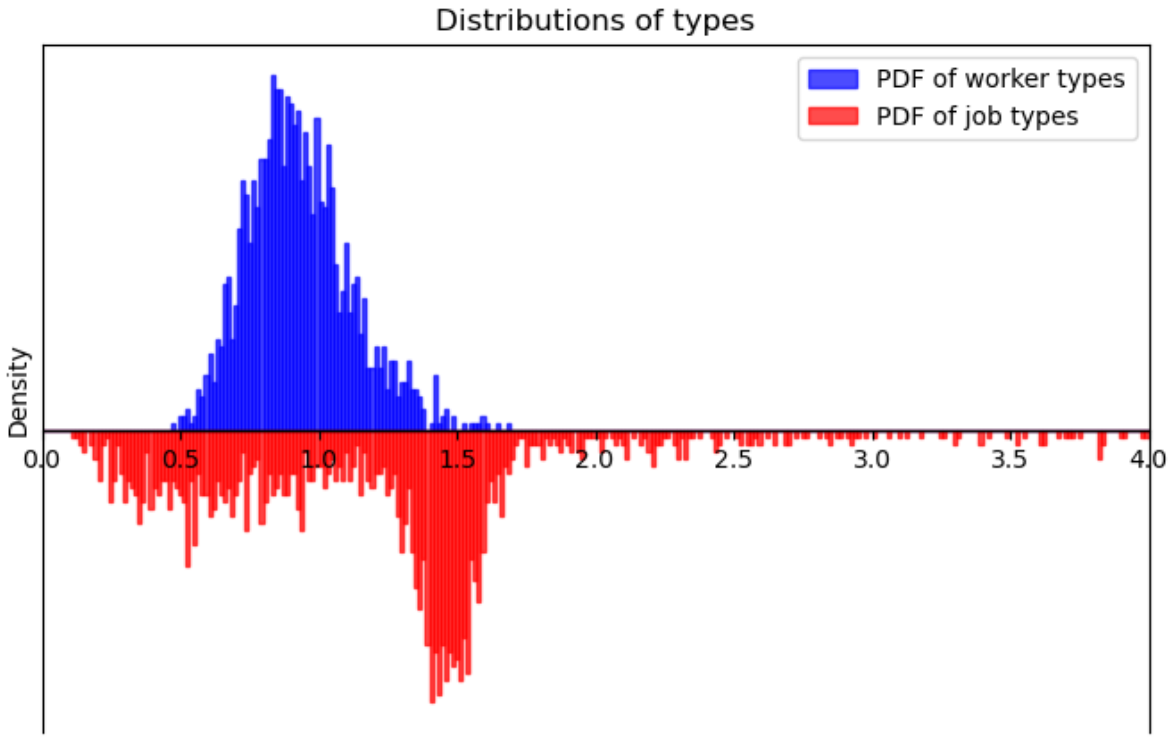
    # Labeling the axes and the title
    plt.ylabel('Density')
    plt.title(title)
    plt.gca().yaxis.set_major_locator(MaxNLocator(integer=True))
    plt.legend()
    plt.yticks([])

    plt.show()

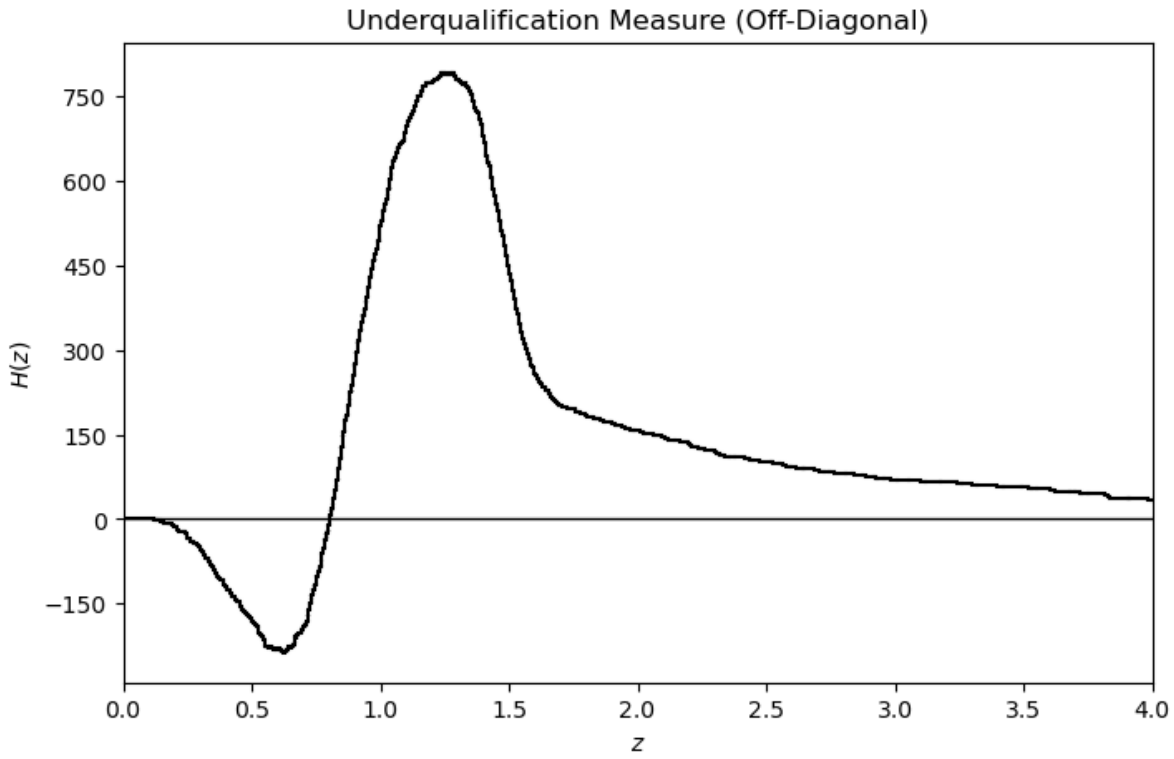
ConcaveCostOT.plot_marginals_pdf = plot_marginals_pdf
```

We plot the hystograms and the measure of underqualification for the worker types and job types. We then compute the primal solution and plot the matching.

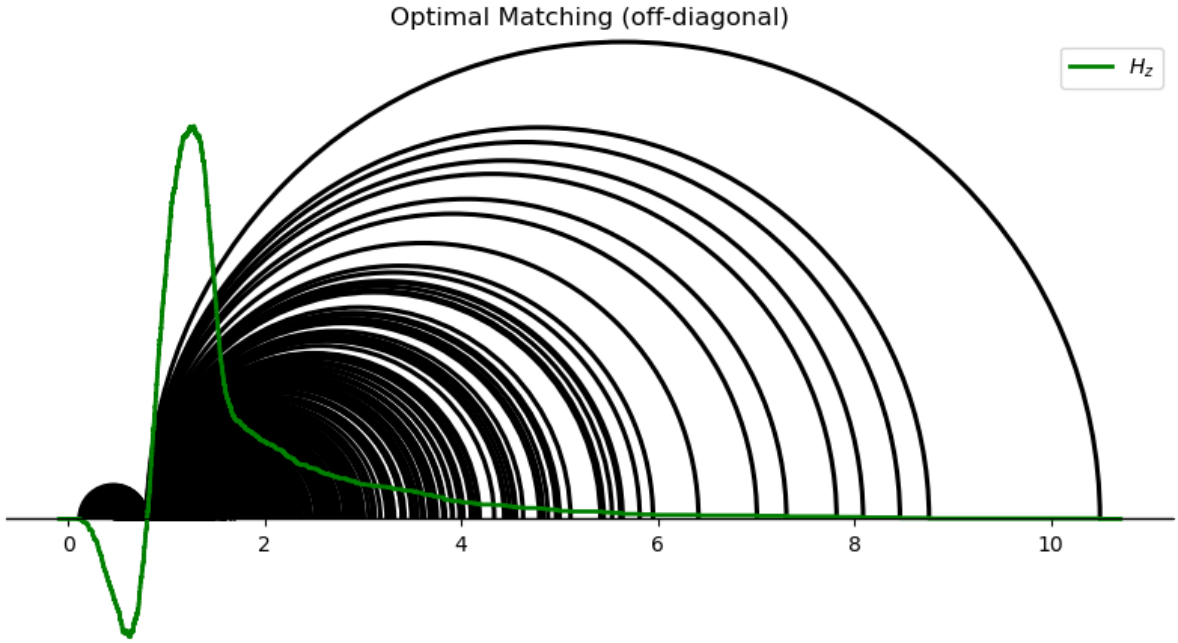
```
# Plot pdf
range_x_axis = (0, 4)
model_1980.plot_marginals_pdf(figsize=(8, 5),
                              bins=300, range_x_axis=range_x_axis)
```




```
# Plot H_z
model_OD_1980 , _ = model_1980.generate_offD_onD_matching()
model_OD_1980.plot_H_z(figsize=(8, 5), range_x_axis=range_x_axis, scatter=False)
```




```
# Compute optimal matching and plot off diagonal matching
matching_1980, matching_OD_1980, model_OD_1980 = model_1980.solve_primal_DSS()
model_OD_1980.plot_matching(matching_OD_1980,
                            title = 'Optimal Matching (off-diagonal)',
                            figsize=(10, 10), plot_H_z=True, scatter=False)
```



From the optimal matching we compute and visualize the hierarchies.

We then find the dual solution (ϕ, ψ) and compute the wages as $w_x = g(x) - \phi_x$, assuming that the type-specific productivity of type x is $g(x) = x$.

```
# Find subpairs and plot hierarchies
subpairs, pairs_between = model_OD_1980.find_subpairs(matching_OD_1980,
                                        return_pairs_between=True)
model_OD_1980.plot_hierarchies(subpairs, scatter=False,
                                        range_x_axis=range_x_axis)

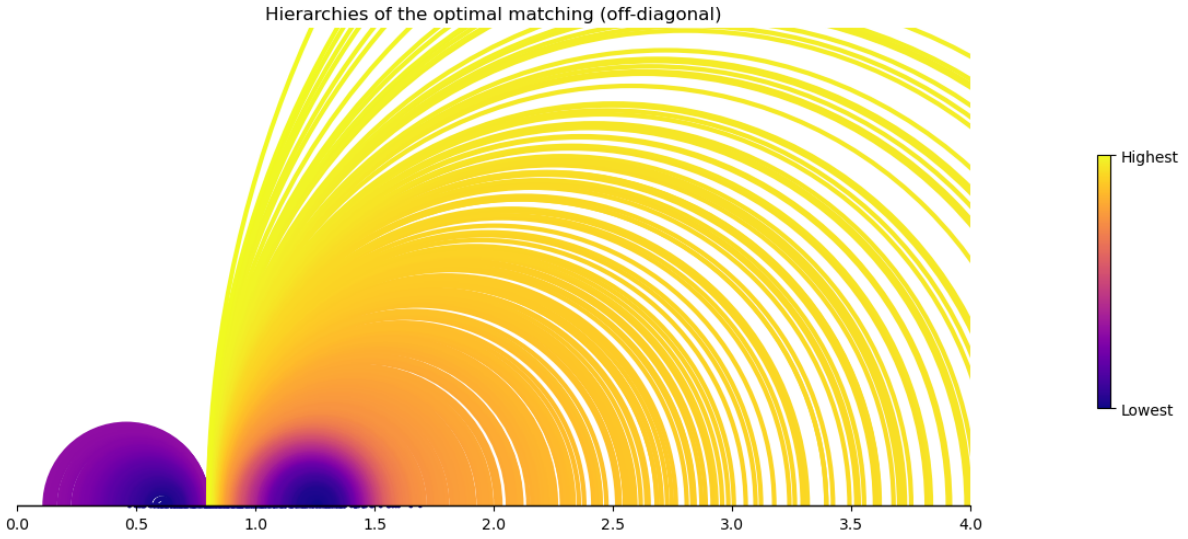
# Compute dual solution: φ_x and ψ_y
ϕ_worker_x_1980 , ψ_firm_y_1980 = model_OD_1980.compute_dual_off_diagonal(
                                        subpairs, pairs_between)

# Check dual feasibility
dual_feasibility_i_j = ϕ_worker_x_1980[:,None] + ψ_firm_y_1980[None,:] \
                       - model_OD_1980.cost_x_y
print('Dual feasibility violation:', dual_feasibility_i_j.max())

# Check strong duality
dual_sol = (model_OD_1980.n_x * ϕ_worker_x_1980).sum() \
           + (model_OD_1980.m_y * ψ_firm_y_1980).sum()
primal_sol = (matching_OD_1980 * model_OD_1980.cost_x_y).sum()

print('Value of dual solution: ', dual_sol)
print('Value of primal solution: ', primal_sol)

# Compute wages: wage_x = x - φ_x
wage_worker_x_1980 = model_1980.X_types - ϕ_worker_x_1980
```




```
Dual feasibility violation: 4.322138159526534e-08
Value of dual solution:  825.8167029184153
Value of primal solution:  825.8167029184157
```

Let's plot average wages and wage dispersion generated by the model.

```
def plot_wages_application(wages):

```

(continues on next page)

(continued from previous page)

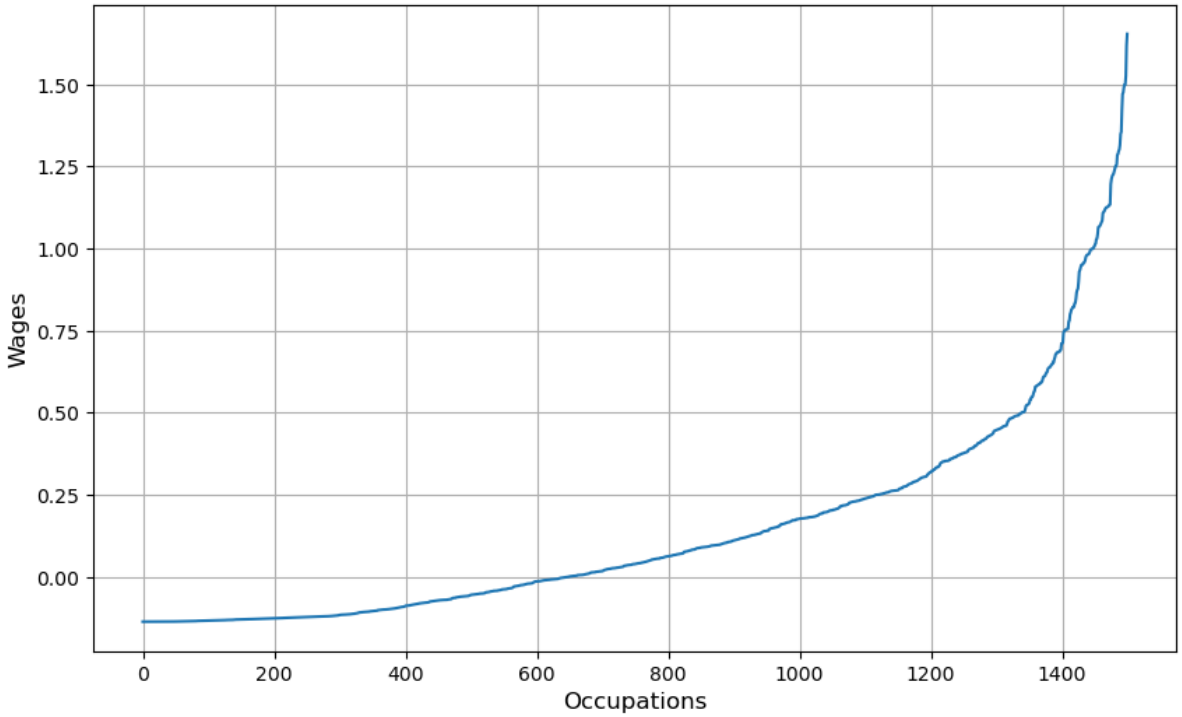
```
    plt.figure(figsize=(10, 6))
    plt.plot(np.sort(wages), label='Wages')
    plt.xlabel("Occupations", fontsize=12)
    plt.ylabel("Wages", fontsize=12)
    plt.grid(True)
    plt.show()


def plot_wage_dispersion_model(wage_worker_x, bins=100,
                               title='Wage Dispersion', figsize=(10, 6)):
    # Compute the percentiles
    percentiles = np.linspace(0, 100, bins + 1)
    bin_edges = np.percentile(wage_worker_x, percentiles)

    # Compute the standard deviation within each percentile range
    stds = []
    for i in range(bins):
        # Compute the standard deviation for the current bin
        bin_data = wage_worker_x[
        (wage_worker_x >= bin_edges[i]) & (wage_worker_x < bin_edges[i + 1])]
        if len(bin_data) > 1:
            stds.append(np.std(bin_data))
        else:
            stds.append(0)

    # Plot the standard deviations for each percentile as bars
    plt.figure(figsize=figsize)
    plt.bar(range(bins), stds, width=1.0, color='grey',
                           alpha=0.7, edgecolor='white')
    plt.xlabel('Percentile', fontsize=12)
    plt.ylabel('Standard Deviation', fontsize=12)
    plt.title(title, fontsize=14)
    plt.grid(axis='y', linestyle='-', alpha=0.6)


plot_wages_application(wage_worker_x_1980)
```




```
plot_wage_dispersion_model(wage_worker_x_1980, bins=100)
```

Part IV

Dynamic Linear Economies

CHAPTER

SEVENTEEN

RECURSIVE MODELS OF DYNAMIC LINEAR ECONOMIES

"Mathematics is the art of giving the same name to different things" – Henri Poincare

"Complete market economies are all alike" – Robert E. Lucas, Jr., (1989)

"Every partial equilibrium model can be reinterpreted as a general equilibrium model." – Anonymous

17.1 A Suite of Models

This lecture presents a class of linear-quadratic-Gaussian models of general economic equilibrium designed by Lars Peter Hansen and Thomas J. Sargent [Hansen and Sargent, 2013].

The class of models is implemented in a Python class DLE that is part of quantecon.

Subsequent lectures use the DLE class to implement various instances that have appeared in the economics literature

1. *Growth in Dynamic Linear Economies*
2. *Lucas Asset Pricing using DLE*
3. *IRFs in Hall Model*
4. *Permanent Income Using the DLE class*
5. *Rosen schooling model*
6. *Cattle cycles*
7. *Shock Non Invertibility*

17.1.1 Overview of the Models

In saying that "complete markets are all alike", Robert E. Lucas, Jr. was noting that all of them have

- a commodity space.
- a space dual to the commodity space in which prices reside.
- endowments of resources.
- peoples' preferences over goods.
- physical technologies for transforming resources into goods.
- random processes that govern shocks to technologies and preferences and associated information flows.
- a single budget constraint per person.
- the existence of a representative consumer even when there are many people in the model.

- a concept of competitive equilibrium.
- theorems connecting competitive equilibrium allocations to allocations that would be chosen by a benevolent social planner.

The models have **no frictions** such as …

- Enforcement difficulties
- Information asymmetries
- Other forms of transactions costs
- Externalities

The models extensively use the powerful ideas of

- Indexing commodities and their prices by time (John R. Hicks).
- Indexing commodities and their prices by chance (Kenneth Arrow).

Much of the imperialism of complete markets models comes from applying these two tricks.

The Hicks trick of indexing commodities by time is the idea that **dynamics are a special case of statics**.

The Arrow trick of indexing commodities by chance is the idea that **analysis of trade under uncertainty is a special case of the analysis of trade under certainty**.

The [Hansen and Sargent, 2013] class of models specify the commodity space, preferences, technologies, stochastic shocks and information flows in ways that allow the models to be analyzed completely using only the tools of linear time series models and linear-quadratic optimal control described in the two lectures Linear State Space Models and Linear Quadratic Control.

There are costs and benefits associated with the simplifications and specializations needed to make a particular model fit within the [Hansen and Sargent, 2013] class

- the costs are that linear-quadratic structures are sometimes too confining.
- benefits include computational speed, simplicity, and ability to analyze many model features analytically or nearly analytically.

A variety of superficially different models are all instances of the [Hansen and Sargent, 2013] class of models

- Lucas asset pricing model
- Lucas-Prescott model of investment under uncertainty
- Asset pricing models with habit persistence
- Rosen-Topel equilibrium model of housing
- Rosen schooling models
- Rosen-Murphy-Scheinkman model of cattle cycles
- Hansen-Sargent-Tallarini model of robustness and asset pricing
- Many more …

The diversity of these models conceals an essential unity that illustrates the quotation by Robert E. Lucas, Jr., with which we began this lecture.

17.1.2 Forecasting?

A consequence of a single budget constraint per person plus the Hicks-Arrow tricks is that households and firms need not forecast.

But there exist equivalent structures called **recursive competitive equilibria** in which they do appear to need to forecast.

In these structures, to forecast, households and firms use:

- equilibrium pricing functions, and
- knowledge of the Markov structure of the economy's state vector.

17.1.3 Theory and Econometrics

For an application of the [Hansen and Sargent, 2013] class of models, the outcome of theorizing is a stochastic process, i.e., a probability distribution over sequences of prices and quantities, indexed by parameters describing preferences, technologies, and information flows.

Another name for that object is a likelihood function, a key object of both frequentist and Bayesian statistics.

There are two important uses of an **equilibrium stochastic process** or **likelihood function**.

The first is to solve the **direct problem**.

The **direct problem** takes as inputs values of the parameters that define preferences, technologies, and information flows and as an output characterizes or simulates random paths of quantities and prices.

The second use of an equilibrium stochastic process or likelihood function is to solve the **inverse problem**.

The **inverse problem** takes as an input a time series sample of observations on a subset of prices and quantities determined by the model and from them makes inferences about the parameters that define the model's preferences, technologies, and information flows.

17.1.4 More Details

A [Hansen and Sargent, 2013] economy consists of **lists of matrices** that describe peoples' household technologies, their preferences over consumption services, their production technologies, and their information sets.

There are complete markets in history-contingent commodities.

Competitive equilibrium allocations and prices

- satisfy equations that are easy to write down and solve
- have representations that are convenient econometrically

Different example economies manifest themselves simply as different settings for various matrices.

[Hansen and Sargent, 2013] use these tools:

- A theory of recursive dynamic competitive economies
- Linear optimal control theory
- Recursive methods for estimating and interpreting vector autoregressions

The models are flexible enough to express alternative senses of a representative household

- A single 'stand-in' household of the type used to good effect by Edward C. Prescott.
- Heterogeneous households satisfying conditions for Gorman aggregation into a representative household.

- Heterogeneous household technologies that violate conditions for Gorman aggregation but are still susceptible to aggregation into a single representative household via 'non-Gorman' or 'mongrel' aggregation'.

These three alternative types of aggregation have different consequences in terms of how prices and allocations can be computed.

In particular, can prices and an aggregate allocation be computed before the equilibrium allocation to individual heterogeneous households is computed?

- Answers are "Yes" for Gorman aggregation, "No" for non-Gorman aggregation.

In summary, the insights and practical benefits from economics to be introduced in this lecture are

- Deeper understandings that come from recognizing common underlying structures.
- Speed and ease of computation that comes from unleashing a common suite of Python programs.

We'll use the following **mathematical tools**

- Stochastic Difference Equations (Linear).
- Duality: LQ Dynamic Programming and Linear Filtering are the same things mathematically.
- The Spectral Factorization Identity (for understanding vector autoregressions and non-Gorman aggregation).

So here is our roadmap.

We'll describe sets of matrices that pin down

- Information
- Technologies
- Preferences

Then we'll describe

- Equilibrium concept and computation
- Econometric representation and estimation

17.1.5 Stochastic Model of Information Flows and Outcomes

We'll use stochastic linear difference equations to describe information flows **and** equilibrium outcomes.

The sequence $\{w_t : t = 1, 2, ...\}$ is said to be a martingale difference sequence adapted to $\{J_t : t = 0, 1, ...\}$ if $E(w_{t+1}|J_t) = 0$ for $t = 0, 1, ...$.

The sequence $\{w_t : t = 1, 2, ...\}$ is said to be conditionally homoskedastic if $E(w_{t+1}w'_{t+1} \mid J_t) = I$ for $t = 0, 1, ...$.

We assume that the $\{w_t : t = 1, 2, ...\}$ process is conditionally homoskedastic.

Let $\{x_t : t = 1, 2, ...\}$ be a sequence of n-dimensional random vectors, i.e. an n-dimensional stochastic process.

The process $\{x_t : t = 1, 2, ...\}$ is constructed recursively using an initial random vector $x_0 \sim \mathcal{N}(\hat{x}_0, \Sigma_0)$ and a time-invariant law of motion:

$$x_{t+1} = Ax_t + Cw_{t+1}$$

for $t = 0, 1, ...$ where A is an n by n matrix and C is an n by N matrix.

Evidently, the distribution of x_{t+1} conditional on x_t is $\mathcal{N}(Ax_t, CC')$.

17.1.6 Information Sets

Let J_0 be generated by x_0 and J_t be generated by $x_0, w_1, \dots, w_t$, which means that J_t consists of the set of all measurable functions of $\{x_0, w_1, \dots, w_t\}$.

17.1.7 Prediction Theory

The optimal forecast of x_{t+1} given current information is

$$E(x_{t+1} \mid J_t) = Ax_t$$

and the one-step-ahead forecast error is

$$x_{t+1} - E(x_{t+1} \mid J_t) = Cw_{t+1}$$

The covariance matrix of x_{t+1} conditioned on J_t is

$$E(x_{t+1} - E(x_{t+1} \mid J_t))(x_{t+1} - E(x_{t+1} \mid J_t))' = CC'$$

A nonrecursive expression for x_t as a function of $x_0, w_1, w_2, \dots, w_t$ is

$$\begin{aligned} x_t &= Ax_{t-1} + Cw_t \\ &= A^2 x_{t-2} + ACw_{t-1} + Cw_t \\ &= \Big[\sum_{\tau=0}^{t-1} A^\tau C w_{t-\tau}\Big] + A^t x_0 \end{aligned}$$

Shift forward in time:

$$x_{t+j} = \sum_{s=0}^{j-1} A^s C w_{t+j-s} + A^j x_t$$

Projecting on the information set $\{x_0, w_t, w_{t-1}, \dots, w_1\}$ gives

$$E_t x_{t+j} = A^j x_t$$

where $E_t(\cdot) \equiv E[(\cdot) \mid x_0, w_t, w_{t-1}, \dots, w_1] = E(\cdot) \mid J_t$, and x_t is in J_t.

It is useful to obtain the covariance matrix of the j-step-ahead prediction error $x_{t+j} - E_t x_{t+j} = \sum_{s=0}^{j-1} A^s C w_{t-s+j}$. Evidently,

$$E_t(x_{t+j} - E_t x_{t+j})(x_{t+j} - E_t x_{t+j})' = \sum_{k=0}^{j-1} A^k CC' A^{k'} \equiv v_j$$

v_j can be calculated recursively via

$$\begin{aligned} v_1 &= CC' \\ v_j &= CC' + Av_{j-1}A', \quad j \geq 2 \end{aligned}$$

17.1.8 Orthogonal Decomposition

To decompose these covariances into parts attributable to the individual components of w_t, we let i_τ be an N-dimensional column vector of zeroes except in position τ, where there is a one. Define a matrix $\upsilon_{j,\tau}$

$$\upsilon_{j,\tau} = \sum_{k=0}^{j-1} A^k C i_\tau i_\tau' C' A'^k.$$

Note that $\sum_{\tau=1}^{N} i_\tau i_\tau' = I$, so that we have

$$\sum_{\tau=1}^{N} \upsilon_{j,\tau} = \upsilon_j$$

Evidently, the matrices $\{\upsilon_{j,\tau}, \tau = 1, \ldots, N\}$ give an orthogonal decomposition of the covariance matrix of j-step-ahead prediction errors into the parts attributable to each of the components $\tau = 1, \ldots, N$.

17.1.9 Taste and Technology Shocks

$E(w_t \mid J_{t-1}) = 0$ and $E(w_t w_t' \mid J_{t-1}) = I$ for $t = 1, 2, \ldots$

$$b_t = U_b z_t \text{ and } d_t = U_d z_t,$$

U_b and U_d are matrices that select entries of z_t. The law of motion for $\{z_t : t = 0, 1, \ldots\}$ is

$$z_{t+1} = A_{22} z_t + C_2 w_{t+1} \text{ for } t = 0, 1, \ldots$$

where z_0 is a given initial condition. The eigenvalues of the matrix A_{22} have absolute values that are less than or equal to one.

Thus, in summary, our model of **information and shocks** is

$$\begin{aligned} z_{t+1} &= A_{22} z_t + C_2 w_{t+1} \\ b_t &= U_b z_t \\ d_t &= U_d z_t. \end{aligned}$$

We can now briefly summarize other components of our economies, in particular

- Production technologies
- Household technologies
- Household preferences

17.1.10 Production Technology

Where c_t is a vector of consumption rates, k_t is a vector of physical capital goods, g_t is a vector intermediate productions goods, d_t is a vector of technology shocks, the production technology is

$$\begin{aligned} \Phi_c c_t + \Phi_g g_t + \Phi_i i_t &= \Gamma k_{t-1} + d_t \\ k_t &= \Delta_k k_{t-1} + \Theta_k i_t \\ g_t \cdot g_t &= \ell_t^2 \end{aligned}$$

Here $\Phi_c, \Phi_g, \Phi_i, \Gamma, \Delta_k, \Theta_k$ are all matrices conformable to the vectors they multiply and ℓ_t is a disutility generating resource supplied by the household.

For technical reasons that facilitate computations, we make the following.

Assumption: $[\Phi_c \ \Phi_g]$ is nonsingular.

17.1.11 Household Technology

Households confront a technology that allows them to devote consumption goods to construct a vector h_t of household capital goods and a vector s_t of utility generating house services

$$s_t = \Lambda h_{t-1} + \Pi c_t$$
$$h_t = \Delta_h h_{t-1} + \Theta_h c_t$$

where $\Lambda, \Pi, \Delta_h, \Theta_h$ are matrices that pin down the household technology.

We make the following

Assumption: The absolute values of the eigenvalues of Δ_h are less than or equal to one.

Below, we'll outline further assumptions that we shall occasionally impose.

17.1.12 Preferences

Where b_t is a stochastic process of preference shocks that will play the role of demand shifters, the representative household orders stochastic processes of consumption services s_t according to

$$\Big(\frac{1}{2}\Big) E \sum_{t=0}^{\infty} \beta^t [(s_t - b_t) \cdot (s_t - b_t) + \ell_t^2] \big| J_0, \ 0 < \beta < 1$$

We now proceed to give examples of production and household technologies that appear in various models that appear in the literature.

First, we give examples of production Technologies

$$\Phi_c c_t + \Phi_g g_t + \Phi_i i_t = \Gamma k_{t-1} + d_t$$

$$\mid g_t \mid \leq \ell_t$$

so we'll be looking for specifications of the matrices $\Phi_c, \Phi_g, \Phi_i, \Gamma, \Delta_k, \Theta_k$ that define them.

17.1.13 Endowment Economy

There is a single consumption good that cannot be stored over time.

In time period t, there is an endowment d_t of this single good.

There is neither a capital stock, nor an intermediate good, nor a rate of investment.

So $c_t = d_t$.

To implement this specification, we can choose A_{22}, C_2, and U_d to make d_t follow any of a variety of stochastic processes.

To satisfy our earlier rank assumption, we set:

$$c_t + i_t = d_{1t}$$

$$g_t = \phi_1 i_t$$

where ϕ_1 is a small positive number.

To implement this version, we set $\Delta_k = \Theta_k = 0$ and

$$\Phi_c = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \Phi_i = \begin{bmatrix} 1 \\ \phi_1 \end{bmatrix}, \ \Phi_g = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \ \Gamma = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \ d_t = \begin{bmatrix} d_{1t} \\ 0 \end{bmatrix}$$

We can use this specification to create a linear-quadratic version of Lucas's (1978) asset pricing model.

17.1.14 Single-Period Adjustment Costs

There is a single consumption good, a single intermediate good, and a single investment good.

The technology is described by

$$\begin{aligned} c_t &= \gamma k_{t-1} + d_{1t}, \ \ \gamma > 0 \\ \phi_1 i_t &= g_t + d_{2t}, \ \ \phi_1 > 0 \\ \ell_t^2 &= g_t^2 \\ k_t &= \delta_k k_{t-1} + i_t, \ 0 < \delta_k < 1 \end{aligned}$$

Set

$$\Phi_c = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \ \Phi_g = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \ \Phi_i = \begin{bmatrix} 0 \\ \phi_1 \end{bmatrix}$$

$$\Gamma = \begin{bmatrix} \gamma \\ 0 \end{bmatrix}, \ \Delta_k = \delta_k, \ \Theta_k = 1$$

We set A_{22}, C_2 and U_d to make $(d_{1t}, d_{2t})' = d_t$ follow a desired stochastic process.

Now we describe some examples of preferences, which as we have seen are ordered by

$$-\left(\frac{1}{2}\right) E \sum_{t=0}^{\infty} \beta^t \left[(s_t - b_t) \cdot (s_t - b_t) + (\ell_t)^2\right] \mid J_0 \quad , \ 0 < \beta < 1$$

where household services are produced via the household technology

$$h_t = \Delta_h h_{t-1} + \Theta_h c_t$$

$$s_t = \Lambda h_{t-1} + \Pi c_t$$

and we make

Assumption: The absolute values of the eigenvalues of Δ_h are less than or equal to one.

Later we shall introduce **canonical** household technologies that satisfy an 'invertibility' requirement relating sequences $\{s_t\}$ of services and $\{c_t\}$ of consumption flows.

And we'll describe how to obtain a canonical representation of a household technology from one that is not canonical.

Here are some examples of household preferences.

Time Separable preferences

$$-\frac{1}{2} E \sum_{t=0}^{\infty} \beta^t \left[(c_t - b_t)^2 + \ell_t^2\right] \mid J_0 \quad , \ 0 < \beta < 1$$

Consumer Durables

$$h_t = \delta_h h_{t-1} + c_t \quad , \ 0 < \delta_h < 1$$

Services at t are related to the stock of durables at the beginning of the period:

$$s_t = \lambda h_{t-1} \ , \ \lambda > 0$$

Preferences are ordered by

$$-\frac{1}{2} E \sum_{t=0}^{\infty} \beta^t \left[(\lambda h_{t-1} - b_t)^2 + \ell_t^2\right] \mid J_0$$

Set $\Delta_h = \delta_h, \Theta_h = 1, \Lambda = \lambda, \Pi = 0$.

Habit Persistence

$$-\Big(\frac{1}{2}\Big) E \sum_{t=0}^{\infty} \beta^t \Big[\big(c_t - \lambda(1-\delta_h) \sum_{j=0}^{\infty} \delta_h^j \, c_{t-j-1} - b_t\big)^2 + \ell_t^2 \Big] | J_0$$

$$0 < \beta < 1 \ , \ 0 < \delta_h < 1 \ , \ \lambda > 0$$

Here the effective bliss point $b_t + \lambda(1-\delta_h) \sum_{j=0}^{\infty} \delta_h^j \, c_{t-j-1}$ shifts in response to a moving average of past consumption.

Initial Conditions

Preferences of this form require an initial condition for the geometric sum $\sum_{j=0}^{\infty} \delta_h^j c_{t-j-1}$ that we specify as an initial condition for the 'stock of household durables,' h_{-1}.

Set

$$h_t = \delta_h h_{t-1} + (1-\delta_h) c_t \quad , \ 0 < \delta_h < 1$$

$$h_t = (1-\delta_h) \sum_{j=0}^{t} \delta_h^j \, c_{t-j} + \delta_h^{t+1} \, h_{-1}$$

$$s_t = -\lambda h_{t-1} + c_t, \ \lambda > 0$$

To implement, set $\Lambda = -\lambda, \ \Pi = 1, \ \Delta_h = \delta_h, \ \Theta_h = 1 - \delta_h$.

Seasonal Habit Persistence

$$-\Big(\frac{1}{2}\Big) E \sum_{t=0}^{\infty} \beta^t \Big[\big(c_t - \lambda(1-\delta_h) \sum_{j=0}^{\infty} \delta_h^j \, c_{t-4j-4} - b_t\big)^2 + \ell_t^2 \Big]$$

$$0 < \beta < 1 \ , \ 0 < \delta_h < 1 \ , \ \lambda > 0$$

Here the effective bliss point $b_t + \lambda(1-\delta_h) \sum_{j=0}^{\infty} \delta_h^j \, c_{t-4j-4}$ shifts in response to a moving average of past consumptions of the same quarter.

To implement, set

$$\tilde{h}_t = \delta_h \tilde{h}_{t-4} + (1-\delta_h) c_t \quad , \ 0 < \delta_h < 1$$

This implies that

$$h_t = \begin{bmatrix} \tilde{h}_t \\ \tilde{h}_{t-1} \\ \tilde{h}_{t-2} \\ \tilde{h}_{t-3} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & \delta_h \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} \tilde{h}_{t-1} \\ \tilde{h}_{t-2} \\ \tilde{h}_{t-3} \\ \tilde{h}_{t-4} \end{bmatrix} + \begin{bmatrix} (1-\delta_h) \\ 0 \\ 0 \\ 0 \end{bmatrix} c_t$$

with consumption services

$$s_t = -\begin{bmatrix} 0 & 0 & 0 & -\lambda \end{bmatrix} h_{t-1} + c_t \quad , \ \lambda > 0$$

Adjustment Costs.

Recall

$$-\Big(\frac{1}{2}\Big) E \sum_{t=0}^{\infty} \beta^t [(c_t - b_{1t})^2 + \lambda^2 (c_t - c_{t-1})^2 + \ell_t^2] \mid J_0$$

$$0 < \beta < 1 \quad , \ \lambda > 0$$

To capture adjustment costs, set

$$h_t = c_t$$

$$s_t = \begin{bmatrix} 0 \\ -\lambda \end{bmatrix} h_{t-1} + \begin{bmatrix} 1 \\ \lambda \end{bmatrix} c_t$$

so that

$$s_{1t} = c_t$$

$$s_{2t} = \lambda(c_t - c_{t-1})$$

We set the first component b_{1t} of b_t to capture the stochastic bliss process and set the second component identically equal to zero.

Thus, we set $\Delta_h = 0, \Theta_h = 1$

$$\Lambda = \begin{bmatrix} 0 \\ -\lambda \end{bmatrix} \ , \ \Pi = \begin{bmatrix} 1 \\ \lambda \end{bmatrix}$$

Multiple Consumption Goods

$$\Lambda = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \text{ and } \Pi = \begin{bmatrix} \pi_1 & 0 \\ \pi_2 & \pi_3 \end{bmatrix}$$

$$-\frac{1}{2}\beta^t(\Pi c_t - b_t)'(\Pi c_t - b_t)$$

$$\mu_t = -\beta^t[\Pi'\Pi\, c_t - \Pi'\, b_t]$$

$$c_t = -(\Pi'\Pi)^{-1}\beta^{-t}\mu_t + (\Pi'\Pi)^{-1}\Pi' b_t$$

This is called the **Frisch demand function** for consumption.

We can think of the vector μ_t as playing the role of prices, up to a common factor, for all dates and states.

The scale factor is determined by the choice of numeraire.

Notions of **substitutes and complements** can be defined in terms of these Frisch demand functions.

Two goods can be said to be **substitutes** if the cross-price effect is positive and to be **complements** if this effect is negative.

Hence this classification is determined by the off-diagonal element of $-(\Pi'\Pi)^{-1}$, which is equal to $\pi_2\pi_3/\det(\Pi'\Pi)$.

If π_2 and π_3 have the same sign, the goods are substitutes.

If they have opposite signs, the goods are complements.

To summarize, our economic structure consists of the matrices that define the following components:

Information and shocks

$$\begin{aligned} z_{t+1} &= A_{22} z_t + C_2 w_{t+1} \\ b_t &= U_b z_t \\ d_t &= U_d z_t \end{aligned}$$

Production Technology

$$\begin{aligned} \Phi_c c_t + \Phi_g g_t + \Phi_i i_t &= \Gamma k_{t-1} + d_t \\ k_t &= \Delta_k k_{t-1} + \Theta_k i_t \\ g_t \cdot g_t &= \ell_t^2 \end{aligned}$$

Household Technology

$$
\begin{aligned}
s_t &= \Lambda h_{t-1} + \Pi c_t \\
h_t &= \Delta_h h_{t-1} + \Theta_h c_t
\end{aligned}
$$

Preferences

$$
\Big(\frac{1}{2}\Big) E \sum_{t=0}^{\infty} \beta^t [(s_t - b_t) \cdot (s_t - b_t) + \ell_t^2] \big| J_0, \ 0 < \beta < 1
$$

Next steps: we move on to discuss two closely connected concepts

- A Planning Problem or Optimal Resource Allocation Problem
- Competitive Equilibrium

17.1.15 Optimal Resource Allocation

Imagine a planner who chooses sequences $\{c_t, i_t, g_t\}_{t=0}^{\infty}$ to maximize

$$
-(1/2) E \sum_{t=0}^{\infty} \beta^t [(s_t - b_t) \cdot (s_t - b_t) + g_t \cdot g_t] \big| J_0
$$

subject to the constraints

$$
\begin{aligned}
\Phi_c c_t + \Phi_g \, g_t + \Phi_i i_t &= \Gamma k_{t-1} + d_t, \\
k_t &= \Delta_k k_{t-1} + \Theta_k i_t, \\
h_t &= \Delta_h h_{t-1} + \Theta_h c_t, \\
s_t &= \Lambda h_{t-1} + \Pi c_t, \\
z_{t+1} &= A_{22} z_t + C_2 w_{t+1}, \ b_t = U_b z_t, \ \text{ and } \ d_t = U_d z_t
\end{aligned}
$$

and initial conditions for h_{-1}, k_{-1}, and z_0.

Throughout, we shall impose the following **square summability** conditions

$$
E \sum_{t=0}^{\infty} \beta^t h_t \cdot h_t \mid J_0 < \infty \ \text{ and } \ E \sum_{t=0}^{\infty} \beta^t k_t \cdot k_t \mid J_0 < \infty
$$

Define:

$$
L_0^2 = [\{y_t\} : y_t \text{ is a random variable in } \ J_t \ \text{ and } \ E \sum_{t=0}^{\infty} \beta^t y_t^2 \mid J_0 < +\infty]
$$

Thus, we require that each component of h_t and each component of k_t belong to L_0^2.

We shall compare and utilize two approaches to solving the planning problem

- Lagrangian formulation
- Dynamic programming

17.1.16 Lagrangian Formulation

Form the Lagrangian

$$\begin{aligned}\mathcal{L} = -E\sum_{t=0}^{\infty}\beta^t\Big[\Big(\frac{1}{2}\Big)[(s_t-b_t)\cdot(s_t-b_t)+g_t\cdot g_t]\\ + M_t^{d\prime}\cdot(\Phi_c c_t+\Phi_g g_t+\Phi_i i_t-\Gamma k_{t-1}-d_t)\\ + M_t^{k\prime}\cdot(k_t-\Delta_k k_{t-1}-\Theta_k i_t)\\ + M_t^{h\prime}\cdot(h_t-\Delta_h h_{t-1}-\Theta_h c_t)\\ + M_t^{s\prime}\cdot(s_t-\Lambda h_{t-1}-\Pi c_t)\Big]\Big|J_0\end{aligned}$$

The planner maximizes $\mathcal{L}$ with respect to the quantities $\{c_t, i_t, g_t\}_{t=0}^{\infty}$ and minimizes with respect to the Lagrange multipliers $M_t^d, M_t^k, M_t^h, M_t^s$.

First-order necessary conditions for maximization with respect to c_t, g_t, h_t, i_t, k_t, and s_t, respectively, are:

$$\begin{aligned}-\Phi_c' M_t^d + \Theta_h' M_t^h + \Pi' M_t^s &= 0,\\ -g_t - \Phi_g' M_t^d &= 0,\\ -M_t^h + \beta E(\Delta_h' M_{t+1}^h + \Lambda' M_{t+1}^s)\mid J_t &= 0,\\ -\Phi_i' M_t^d + \Theta_k' M_t^k &= 0,\\ -M_t^k + \beta E(\Delta_k' M_{t+1}^k + \Gamma' M_{t+1}^d)\mid J_t &= 0,\\ -s_t + b_t - M_t^s &= 0\end{aligned}$$

for $t = 0, 1, \ldots$.

In addition, we have the complementary slackness conditions (these recover the original transition equations) and also transversality conditions

$$\begin{aligned}\lim_{t\to\infty}\beta^t E[M_t^{k\prime}k_t]\mid J_0 = 0\\ \lim_{t\to\infty}\beta^t E[M_t^{h\prime}h_t]\mid J_0 = 0\end{aligned}$$

The system formed by the FONCs and the transition equations can be handed over to Python.

Python will solve the planning problem for fixed parameter values.

Here are the **Python Ready Equations**

$$\begin{aligned}-\Phi_c' M_t^d + \Theta_h' M_t^h + \Pi' M_t^s &= 0,\\ -g_t - \Phi_g' M_t^d &= 0,\\ -M_t^h + \beta E(\Delta_h' M_{t+1}^h + \Lambda' M_{t+1}^s)\mid J_t &= 0,\\ -\Phi_i' M_t^d + \Theta_k' M_t^k &= 0,\\ -M_t^k + \beta E(\Delta_k' M_{t+1}^k + \Gamma' M_{t+1}^d)\mid J_t &= 0,\\ -s_t + b_t - M_t^s &= 0\\ \Phi_c c_t + \Phi_g g_t + \Phi_i i_t &= \Gamma k_{t-1} + d_t,\\ k_t &= \Delta_k k_{t-1} + \Theta_k i_t,\\ h_t &= \Delta_h h_{t-1} + \Theta_h c_t,\\ s_t &= \Lambda h_{t-1} + \Pi c_t,\\ z_{t+1} &= A_{22} z_t + C_2 w_{t+1},\ b_t = U_b z_t,\ \text{ and }\ d_t = U_d z_t\end{aligned}$$

The Lagrange multipliers or **shadow prices** satisfy

$$M_t^s = b_t - s_t$$

$$M_t^h = E\left[\sum_{\tau=1}^{\infty} \beta^\tau (\Delta_h')^{\tau-1} \Lambda' M_{t+\tau}^s \mid J_t\right]$$

$$M_t^d = \begin{bmatrix} \Phi_c' \\ \Phi_g' \end{bmatrix}^{-1} \begin{bmatrix} \Theta_h' M_t^h + \Pi' M_t^s \\ -g_t \end{bmatrix}$$

$$M_t^k = E\left[\sum_{\tau=1}^{\infty} \beta^\tau (\Delta_k')^{\tau-1} \Gamma' M_{t+\tau}^d \mid J_t\right]$$

$$M_t^i = \Theta_k' M_t^k$$

Although it is possible to use matrix operator methods to solve the above **Python ready equations**, that is not the approach we'll use.

Instead, we'll use dynamic programming to get recursive representations for both quantities and shadow prices.

17.1.17 Dynamic Programming

Dynamic Programming always starts with the word **let**.

Thus, let $V(x_0)$ be the optimal value function for the planning problem as a function of the initial state vector x_0.

(Thus, in essence, dynamic programming amounts to an application of a **guess and verify** method in which we begin with a guess about the answer to the problem we want to solve. That's why we start with **let** $V(x_0)$ be the (value of the) answer to the problem, then establish and verify a bunch of conditions $V(x_0)$ has to satisfy if indeed it is the answer)

The optimal value function $V(x)$ satisfies the **Bellman equation**

$$V(x_0) = \max_{c_0, i_0, g_0} [-.5[(s_0 - b_0) \cdot (s_0 - b_0) + g_0 \cdot g_0] + \beta E V(x_1)]$$

subject to the linear constraints

$$\begin{aligned} \Phi_c c_0 + \Phi_g g_0 + \Phi_i i_0 &= \Gamma k_{-1} + d_0, \\ k_0 &= \Delta_k k_{-1} + \Theta_k i_0, \\ h_0 &= \Delta_h h_{-1} + \Theta_h c_0, \\ s_0 &= \Lambda h_{-1} + \Pi c_0, \\ z_1 &= A_{22} z_0 + C_2 w_1, \; b_0 = U_b z_0 \text{ and } d_0 = U_d z_0 \end{aligned}$$

Because this is a linear-quadratic dynamic programming problem, it turns out that the value function has the form

$$V(x) = x' P x + \rho$$

Thus, we want to solve an instance of the following linear-quadratic dynamic programming problem:

Choose a contingency plan for $\{x_{t+1}, u_t\}_{t=0}^{\infty}$ to maximize

$$-E \sum_{t=0}^{\infty} \beta^t [x_t' R x_t + u_t' Q u_t + 2 u_t' W' x_t], \; 0 < \beta < 1$$

subject to

$$x_{t+1} = A x_t + B u_t + C w_{t+1}, \; t \geq 0$$

where x_0 is given; x_t is an $n \times 1$ vector of state variables, and u_t is a $k \times 1$ vector of control variables.

We assume w_{t+1} is a martingale difference sequence with $E w_t w_t' = I$, and that C is a matrix conformable to x and w.

The optimal value function $V(x)$ satisfies the Bellman equation

$$V(x_t) = \max_{u_t}\Big\{-(x_t' R x_t + u_t' Q u_t + 2u_t' W x_t) + \beta E_t V(x_{t+1})\Big\}$$

where maximization is subject to

$$x_{t+1} = Ax_t + Bu_t + Cw_{t+1},\ t \geq 0$$

$$V(x_t) = -x_t' P x_t - \rho$$

P satisfies

$$P = R + \beta A'PA - (\beta A'PB + W)(Q + \beta B'PB)^{-1}(\beta B'PA + W')$$

This equation in P is called the **algebraic matrix Riccati equation**.

The optimal decision rule is $u_t = -Fx_t$, where

$$F = (Q + \beta B'PB)^{-1}(\beta B'PA + W')$$

The optimum decision rule for u_t is independent of the parameters C, and so of the noise statistics.

Iterating on the Bellman operator leads to

$$V_{j+1}(x_t) = \max_{u_t}\Big\{-(x_t' R x_t + u_t' Q u_t + 2u_t' W x_t) + \beta E_t V_j(x_{t+1})\Big\}$$

$$V_j(x_t) = -x_t' P_j x_t - \rho_j$$

where P_j and ρ_j satisfy the equations

$$P_{j+1} = R + \beta A'P_jA - (\beta A'P_jB + W)(Q + \beta B'P_jB)^{-1}(\beta B'P_jA + W')$$
$$\rho_{j+1} = \beta\rho_j + \beta \text{ trace } P_jCC'$$

We can now state the planning problem as a dynamic programming problem

$$\max_{\{u_t, x_{t+1}\}} -E\sum_{t=0}^{\infty}\beta^t[x_t' R x_t + u_t' Q u_t + 2u_t' W' x_t], \quad 0 < \beta < 1$$

where maximization is subject to

$$x_{t+1} = Ax_t + Bu_t + Cw_{t+1},\ t \geq 0$$

$$x_t = \begin{bmatrix} h_{t-1} \\ k_{t-1} \\ z_t \end{bmatrix}, \qquad u_t = i_t$$

where

$$A = \begin{bmatrix} \Delta_h & \Theta_h U_c[\Phi_c\ \ \Phi_g]^{-1}\Gamma & \Theta_h U_c[\Phi_c\ \ \Phi_g]^{-1}U_d \\ 0 & \Delta_k & 0 \\ 0 & 0 & A_{22} \end{bmatrix}$$

$$B = \begin{bmatrix} -\Theta_h U_c[\Phi_c\ \ \Phi_g]^{-1}\Phi_i \\ \Theta_k \\ 0 \end{bmatrix},\ C = \begin{bmatrix} 0 \\ 0 \\ C_2 \end{bmatrix}$$

$$\begin{bmatrix} x_t \\ u_t \end{bmatrix}' S \begin{bmatrix} x_t \\ u_t \end{bmatrix} = \begin{bmatrix} x_t \\ u_t \end{bmatrix}' \begin{bmatrix} R & W \\ W' & Q \end{bmatrix} \begin{bmatrix} x_t \\ u_t \end{bmatrix}$$

$$S = (G'G + H'H)/2$$

$$H = [\Lambda \ \vdots \ \Pi U_c[\Phi_c \ \ \Phi_g]^{-1}\Gamma \ \vdots \ \Pi U_c[\Phi_c \ \ \Phi_g]^{-1}U_d - U_b \ \vdots \ -\Pi U_c[\Phi_c \ \ \Phi_g]^{-1}\Phi_i]$$

$$G = U_g[\Phi_c \ \ \Phi_g]^{-1}[0 \ \vdots \ \Gamma \ \vdots \ U_d \ \vdots \ -\Phi_i].$$

Lagrange multipliers as gradient of value function

A useful fact is that Lagrange multipliers equal gradients of the planner's value function

$$\mathcal{M}_t^k = M_k x_t \ \text{ and } \ M_t^h = M_h x_t \ \text{ where}$$
$$M_k = 2\beta[0 \ I \ 0]PA^o$$
$$M_h = 2\beta[I \ 0 \ 0]PA^o$$

$$\mathcal{M}_t^s = M_s x_t \ \text{ where } \ M_s = (S_b - S_s) \ \text{ and } \ S_b = [0 \ 0 \ U_b]$$

$$\mathcal{M}_t^d = M_d x_t \ \text{ where } \ M_d = \begin{bmatrix}\Phi_c' \\ \Phi_g'\end{bmatrix}^{-1}\begin{bmatrix}\Theta_h' M_h + \Pi' M_s \\ -S_g\end{bmatrix}$$

$$\mathcal{M}_t^c = M_c x_t \ \text{ where } \ M_c = \Theta_h' M_h + \Pi' M_s$$

$$\mathcal{M}_t^i = M_i x_t \ \text{ where } \ M_i = \Theta_k' M_k$$

We will use this fact and these equations to compute competitive equilibrium prices.

17.1.18 Other mathematical infrastructure

Let's start with describing the **commodity space** and **pricing functional** for our competitive equilibrium.

For the **commodity space**, we use

$$L_0^2 = [\{y_t\} : y_t \text{ is a random variable in } \ J_t \ \text{ and } \ E\sum_{t=0}^{\infty}\beta^t y_t^2 \mid J_0 < +\infty]$$

For **pricing functionals**, we express values as inner products

$$\pi(c) = E\sum_{t=0}^{\infty}\beta^t p_t^0 \cdot c_t \mid J_0$$

where p_t^0 belongs to L_0^2.

With these objects in our toolkit, we move on to state the problem of a **Representative Household in a competitive equilibrium**.

17.1.19 Representative Household

The representative household owns endowment process and initial stocks of h and k and chooses stochastic processes for $\{c_t, \ s_t, \ h_t, \ \ell_t\}_{t=0}^{\infty}$, each element of which is in L_0^2, to maximize

$$-\frac{1}{2}E_0\sum_{t=0}^{\infty}\beta^t\left[(s_t - b_t)\cdot(s_t - b_t) + \ell_t^2\right]$$

subject to

$$E\sum_{t=0}^{\infty}\beta^t p_t^0 \cdot c_t \mid J_0 = E\sum_{t=0}^{\infty}\beta^t\left(w_t^0\ell_t + \alpha_t^0 \cdot d_t\right) \mid J_0 + v_0 \cdot k_{-1}$$

$$s_t = \Lambda h_{t-1} + \Pi c_t$$

$$h_t = \Delta_h h_{t-1} + \Theta_h c_t, \quad h_{-1}, k_{-1} \ \text{ given}$$

We now describe the problems faced by two types of firms called type I and type II.

17.1.20 Type I Firm

A type I firm rents capital and labor and endowments and produces c_t, i_t.

It chooses stochastic processes for $\{c_t, i_t, k_t, \ell_t, g_t, d_t\}$, each element of which is in L_0^2, to maximize

$$E_0 \sum_{t=0}^{\infty} \beta^t \left(p_t^0 \cdot c_t + q_t^0 \cdot i_t - r_t^0 \cdot k_{t-1} - w_t^0 \ell_t - \alpha_t^0 \cdot d_t\right)$$

subject to

$$\Phi_c \, c_t + \Phi_g \, g_t + \Phi_i \, i_t = \Gamma k_{t-1} + d_t$$

$$-\ell_t^2 + g_t \cdot g_t = 0$$

17.1.21 Type II Firm

A firm of type II acquires capital via investment and then rents stocks of capital to the c, i-producing type I firm.

A type II firm is a price taker facing the vector v_0 and the stochastic processes $\{r_t^0, q_t^0\}$.

The firm chooses k_{-1} and stochastic processes for $\{k_t, i_t\}_{t=0}^{\infty}$ to maximize

$$E \sum_{t=0}^{\infty} \beta^t (r_t^0 \cdot k_{t-1} - q_t^0 \cdot i_t) \mid J_0 - v_0 \cdot k_{-1}$$

subject to

$$k_t = \Delta_k k_{t-1} + \Theta_k i_t$$

17.1.22 Competitive Equilibrium: Definition

We can now state the following.

Definition: A competitive equilibrium is a price system $[v_0, \{p_t^0, w_t^0, \alpha_t^0, q_t^0, r_t^0\}_{t=0}^{\infty}]$ and an allocation $\{c_t, i_t, k_t, h_t, g_t, d_t\}_{t=0}^{\infty}$ that satisfy the following conditions:

- Each component of the price system and the allocation resides in the space L_0^2.
- Given the price system and given h_{-1}, k_{-1}, the allocation solves the representative household's problem and the problems of the two types of firms.

Versions of the two classical welfare theorems prevail under our assumptions.

We exploit that fact in our algorithm for computing a competitive equilibrium.

Step 1: Solve the planning problem by using dynamic programming.

The allocation (i.e., **quantities**) that solve the planning problem **are** the competitive equilibrium quantities.

Step 2: use the following formulas to compute the **equilibrium price system**

$$p_t^0 = \left[\Pi' M_t^s + \Theta_h' M_t^h\right] / \mu_0^w = M_t^c / \mu_0^w$$

$$w_t^0 = \mid S_g x_t \mid / \mu_0^w$$

$$r_t^0 = \Gamma' M_t^d / \mu_0^w$$

$$q_t^0 = \Theta_k' M_t^k / \mu_0^w = M_t^i / \mu_0^w$$
$$\alpha_t^0 = M_t^d / \mu_0^w$$
$$v_0 = \Gamma' M_0^d / \mu_0^w + \Delta_k' M_0^k / \mu_0^w$$

Verification: With this price system, values can be assigned to the Lagrange multipliers for each of our three classes of agents that cause all first-order necessary conditions to be satisfied at these prices and at the quantities associated with the optimum of the planning problem.

17.1.23 Asset pricing

An important use of an equilibrium pricing system is to do asset pricing.

Thus, imagine that we are presented a dividend stream: $\{y_t\} \in L_0^2$ and want to compute the value of a perpetual claim to this stream.

To value this asset we simply take **price times quantity** and add to get an asset value: $a_0 = E \sum_{t=0}^{\infty} \beta^t \, p_t^0 \cdot y_t \mid J_0$.

To compute ao we proceed as follows.

We let

$$y_t = U_a \, x_t$$
$$a_0 = E \sum_{t=0}^{\infty} \beta^t \, x_t' \, Z_a x_t \mid J_0$$
$$Z_a = U_a' M_c / \mu_0^w$$

We have the following convenient formulas:

$$a_0 = x_0' \, \mu_a \, x_0 + \sigma_a$$
$$\mu_a = \sum_{\tau=0}^{\infty} \beta^\tau \, (A^{o\prime})^\tau \, Z_a \, A^{o\tau}$$
$$\sigma_a = \frac{\beta}{1-\beta} \text{ trace} \left(Z_a \sum_{\tau=0}^{\infty} \beta^\tau \, (A^o)^\tau \, CC' (A^{o\prime})^\tau \right)$$

17.1.24 Re-Opening Markets

We have assumed that all trading occurs once-and-for-all at time $t = 0$.

If we were to **re-open markets** at some time $t > 0$ at time t wealth levels implicitly defined by time 0 trades, we would obtain the same equilibrium allocation (i.e., quantities) and the following time t price system

$$L_t^2 = [\{y_s\}_{s=t}^{\infty} : \; y_s \text{ is a random variable in } J_s \text{ for } s \geq t$$
$$\text{and } E \sum_{s=t}^{\infty} \beta^{s-t} \, y_s^2 \mid J_t < +\infty].$$
$$p_s^t = M_c x_s / [\bar{e}_j M_c x_t], \qquad s \geq t$$
$$w_s^t = \mid S_g x_s \mid / [\bar{e}_j M_c x_t], \quad s \geq t$$
$$r_s^t = \Gamma' M_d x_s / [\bar{e}_j M_c x_t], \quad s \geq t$$
$$q_s^t = M_i x_s / [\bar{e}_j \, M_c x_t], \qquad s \geq t$$
$$\alpha_s^t = M_d x_s / [\bar{e}_j \, M_c x_t], \quad s \geq t$$
$$v_t = [\Gamma' M_d + \Delta_k' M_k] x_t / \, [\bar{e}_j \, M_c x_t]$$

17.2 Econometrics

Up to now, we have described how to solve the **direct problem** that maps model parameters into an (equilibrium) stochastic process of prices and quantities.

Recall the **inverse problem** of inferring model parameters from a single realization of a time series of some of the prices and quantities.

Another name for the inverse problem is **econometrics**.

An advantage of the [Hansen and Sargent, 2013] structure is that it comes with a self-contained theory of econometrics.

It is really just a tale of two state-space representations.

Here they are:

Original State-Space Representation:

$$\begin{aligned} x_{t+1} &= A^o x_t + C w_{t+1} \\ y_t &= G x_t + v_t \end{aligned}$$

where v_t is a martingale difference sequence of measurement errors that satisfies $Ev_t v_t' = R, Ew_{t+1} v_s' = 0$ for all $t+1 \geq s$ and

$$x_0 \sim \mathcal{N}(\hat{x}_0, \Sigma_0)$$

Innovations Representation:

$$\begin{aligned} \hat{x}_{t+1} &= A^o \hat{x}_t + K_t a_t \\ y_t &= G \hat{x}_t + a_t, \end{aligned}$$

where $a_t = y_t - E[y_t | y^{t-1}], Ea_t a_t' \equiv \Omega_t = G\Sigma_t G' + R$.

Compare numbers of shocks in the two representations:

- $n_w + n_y$ versus n_y

Compare spaces spanned

- $H(y^t) \subset H(w^t, v^t)$
- $H(y^t) = H(a^t)$

Kalman Filter:.

Kalman gain:

$$K_t = A^o \Sigma_t G' (G \Sigma_t G' + R)^{-1}$$

Riccati Difference Equation:

$$\begin{aligned} \Sigma_{t+1} = & A^o \Sigma_t A^{o\prime} + CC' \\ & - A^o \Sigma_t G' (G \Sigma_t G' + R)^{-1} G \Sigma_t A^{o\prime} \end{aligned}$$

Innovations Representation as Whitener

Whitening Filter:

$$\begin{aligned} a_t &= y_t - G \hat{x}_t \\ \hat{x}_{t+1} &= A^o \hat{x}_t + K_t a_t \end{aligned}$$

can be used recursively to construct a record of innovations $\{a_t\}_{t=0}^T$ from an $(\hat{x}_0, \Sigma_0)$ and a record of observations $\{y_t\}_{t=0}^T$.

Limiting Time-Invariant Innovations Representation

$$\begin{aligned}\Sigma &= A^o \Sigma A^{o\prime} + CC' \\ &\quad - A^o \Sigma G' (G\Sigma G' + R)^{-1} G \Sigma A^{o\prime} \\ K &= A^o \Sigma_t G' (G\Sigma G' + R)^{-1}\end{aligned}$$

$$\begin{aligned}\hat{x}_{t+1} &= A^o \hat{x}_t + K a_t \\ y_t &= G\hat{x}_t + a_t\end{aligned}$$

where $Ea_t a_t' \equiv \Omega = G\Sigma G' + R$.

17.2.1 Factorization of Likelihood Function

Sample of observations $\{y_s\}_{s=0}^T$ on a $(n_y \times 1)$ vector.

$$\begin{aligned}f(y_T, y_{T-1}, \dots, y_0) &= f_T(y_T | y_{T-1}, \dots, y_0) f_{T-1}(y_{T-1} | y_{T-2}, \dots, y_0) \cdots f_1(y_1 | y_0) f_0(y_0) \\ &= g_T(a_T) g_{T-1}(a_{T-1}) \dots g_1(a_1) f_0(y_0).\end{aligned}$$

Gaussian Log-Likelihood:

$$-.5 \sum_{t=0}^{T} \left\{ n_y \ln(2\pi) + \ln |\Omega_t| + a_t' \Omega_t^{-1} a_t \right\}$$

17.2.2 Covariance Generating Functions

Autocovariance: $C_x(\tau) = Ex_t x_{t-\tau}'$.

Generating Function: $S_x(z) = \sum_{\tau=-\infty}^{\infty} C_x(\tau) z^\tau, z \in C$.

17.2.3 Spectral Factorization Identity

Original state-space representation has too many shocks and implies:

$$S_y(z) = G(zI - A^o)^{-1} CC' (z^{-1} I - (A^o)')^{-1} G' + R$$

Innovations representation has as many shocks as dimension of y_t and implies

$$S_y(z) = [G(zI - A^o)^{-1} K + I][G\Sigma G' + R][K'(z^{-1} I - A^{o\prime})^{-1} G' + I]$$

Equating these two leads to:

$$\begin{aligned}&G(zI - A^o)^{-1} CC' (z^{-1} I - A^{o\prime})^{-1} G' + R = \\ &[G(zI - A^o)^{-1} K + I][G\Sigma G' + R][K'(z^{-1} I - A^{o\prime})^{-1} G' + I].\end{aligned}$$

Key Insight: The zeros of the polynomial $\det[G(zI - A^o)^{-1} K + I]$ all lie inside the unit circle, which means that a_t lies in the space spanned by square summable linear combinations of y^t.

$$H(a^t) = H(y^t)$$

Key Property: Invertibility

17.2.4 Wold and Vector Autoregressive Representations

Let's start with some lag operator arithmetic.

The lag operator L and the inverse lag operator L^{-1} each map an infinite sequence into an infinite sequence according to the transformation rules

$$Lx_t \equiv x_{t-1}$$

$$L^{-1}x_t \equiv x_{t+1}$$

A **Wold moving average representation** for $\{y_t\}$ is

$$y_t = [G(I - A^o L)^{-1} KL + I]a_t$$

Applying the inverse of the operator on the right side and using

$$[G(I - A^o L)^{-1} KL + I]^{-1} = I - G[I - (A^o - KG)L]^{-1} KL$$

gives the **vector autoregressive representation**

$$y_t = \sum_{j=1}^{\infty} G(A^o - KG)^{j-1} K y_{t-j} + a_t$$

17.3 Dynamic Demand Curves and Canonical Household Technologies

17.3.1 Canonical Household Technologies

$$\begin{aligned} h_t &= \Delta_h h_{t-1} + \Theta_h c_t \\ s_t &= \Lambda h_{t-1} + \Pi c_t \\ b_t &= U_b z_t \end{aligned}$$

Definition: A household service technology $(\Delta_h, \Theta_h, \Pi, \Lambda, U_b)$ is said to be **canonical** if

- Π is nonsingular, and
- the absolute values of the eigenvalues of $(\Delta_h - \Theta_h \Pi^{-1} \Lambda)$ are strictly less than $1/\sqrt{\beta}$.

Key invertibility property: A canonical household service technology maps a service process $\{s_t\}$ in L_0^2 into a corresponding consumption process $\{c_t\}$ for which the implied household capital stock process $\{h_t\}$ is also in L_0^2.

An inverse household technology:

$$\begin{aligned} c_t &= -\Pi^{-1} \Lambda h_{t-1} + \Pi^{-1} s_t \\ h_t &= (\Delta_h - \Theta_h \Pi^{-1} \Lambda) h_{t-1} + \Theta_h \Pi^{-1} s_t \end{aligned}$$

The restriction on the eigenvalues of the matrix $(\Delta_h - \Theta_h \Pi^{-1} \Lambda)$ keeps the household capital stock $\{h_t\}$ in L_0^2.

17.3.2 Dynamic Demand Functions

$$\rho_t^0 \equiv \Pi^{-1\prime}\Big[p_t^0 - \Theta_h' E_t \sum_{\tau=1}^{\infty} \beta^\tau (\Delta_h' - \Lambda'\Pi^{-1\prime}\Theta_h')^{\tau-1}\Lambda'\Pi^{-1\prime} p_{t+\tau}^0\Big]$$

$$s_{i,t} = \Lambda h_{i,t-1}$$
$$h_{i,t} = \Delta_h h_{i,t-1}$$

where $h_{i,-1} = h_{-1}$.

$$W_0 = E_0 \sum_{t=0}^{\infty} \beta^t (w_t^0 \ell_t + \alpha_t^0 \cdot d_t) + v_0 \cdot k_{-1}$$

$$\mu_0^w = \frac{E_0 \sum_{t=0}^{\infty} \beta^t \rho_t^0 \cdot (b_t - s_{i,t}) - W_0}{E_0 \sum_{t=0}^{\infty} \beta^t \rho_t^0 \cdot \rho_t^0}$$

$$\begin{aligned} c_t &= -\Pi^{-1}\Lambda h_{t-1} + \Pi^{-1} b_t - \Pi^{-1}\mu_0^w E_t\{\Pi'^{-1} - \Pi'^{-1}\Theta_h' \\ &\quad [I - (\Delta_h' - \Lambda'\Pi'^{-1}\Theta_h')\beta L^{-1}]^{-1}\Lambda'\Pi'^{-1}\beta L^{-1}\} p_t^0 \\ h_t &= \Delta_h h_{t-1} + \Theta_h c_t \end{aligned}$$

This system expresses consumption demands at date t as functions of: (i) time-t conditional expectations of future scaled Arrow-Debreu prices $\{p_{t+s}^0\}_{s=0}^{\infty}$; (ii) the stochastic process for the household's endowment $\{d_t\}$ and preference shock $\{b_t\}$, as mediated through the multiplier μ_0^w and wealth W_0; and (iii) past values of consumption, as mediated through the state variable h_{t-1}.

17.4 Gorman Aggregation and Engel Curves

We shall explore how the dynamic demand schedule for consumption goods opens up the possibility of satisfying Gorman's (1953) conditions for aggregation in a heterogeneous consumer model.

The first equation of our demand system is an Engel curve for consumption that is linear in the marginal utility μ_0^2 of individual wealth with a coefficient on μ_0^w that depends only on prices.

The multiplier μ_0^w depends on wealth in an affine relationship, so that consumption is linear in wealth.

In a model with multiple consumers who have the same household technologies $(\Delta_h, \Theta_h, \Lambda, \Pi)$ but possibly different preference shock processes and initial values of household capital stocks, the coefficient on the marginal utility of wealth is the same for all consumers.

Gorman showed that when Engel curves satisfy this property, there exists a unique community or aggregate preference ordering over aggregate consumption that is independent of the distribution of wealth.

17.4.1 Re-Opened Markets

$$\rho_t^t \equiv \Pi^{-1\prime}\Big[p_t^t - \Theta_h' E_t \sum_{\tau=1}^{\infty} \beta^\tau (\Delta_h' - \Lambda'\Pi^{-1\prime}\Theta_h')^{\tau-1}\Lambda'\Pi^{-1\prime} p_{t+\tau}^t\Big]$$

$$s_{i,t} = \Lambda h_{i,t-1}$$
$$h_{i,t} = \Delta_h h_{i,t-1},$$

where now $h_{i,t-1} = h_{t-1}$. Define time t wealth W_t

$$W_t = E_t \sum_{j=0}^{\infty} \beta^j (w_{t+j}^t \ell_{t+j} + \alpha_{t+j}^t \cdot d_{t+j}) + v_t \cdot k_{t-1}$$

$$\mu_t^w = \frac{E_t \sum_{j=0}^{\infty} \beta^j \rho_{t+j}^t \cdot (b_{t+j} - s_{i,t+j}) - W_t}{E_t \sum_{t=0}^{\infty} \beta^j \rho_{t+j}^t \cdot \rho_{t+j}^t}$$

$$\begin{aligned} c_t &= -\Pi^{-1}\Lambda h_{t-1} + \Pi^{-1} b_t - \Pi^{-1}\mu_t^w E_t\{\Pi'^{-1} - \Pi'^{-1}\Theta_h' \\ &\quad [I - (\Delta_h' - \Lambda'\Pi'^{-1}\Theta_h')\beta L^{-1}]^{-1}\Lambda'\Pi'^{-1}\beta L^{-1}\} p_t^t \\ h_t &= \Delta_h h_{t-1} + \Theta_h c_t \end{aligned}$$

17.4.2 Dynamic Demand

Define a time t continuation of a sequence $\{z_t\}_{t=0}^{\infty}$ as the sequence $\{z_\tau\}_{\tau=t}^{\infty}$. The demand system indicates that the time t vector of demands for c_t is influenced by:

Through the multiplier μ_t^w, the time t continuation of the preference shock process $\{b_t\}$ and the time t continuation of $\{s_{i,t}\}$.

The time $t-1$ level of household durables h_{t-1}.

Everything that affects the household's time t wealth, including its stock of physical capital k_{t-1} and its value v_t, the time t continuation of the factor prices $\{w_t, \alpha_t\}$, the household's continuation endowment process, and the household's continuation plan for $\{\ell_t\}$.

The time t continuation of the vector of prices $\{p_t^t\}$.

17.4.3 Attaining a Canonical Household Technology

Apply the following version of a factorization identity:

$$\begin{aligned} &[\Pi + \beta^{1/2} L^{-1}\Lambda(I - \beta^{1/2}L^{-1}\Delta_h)^{-1}\Theta_h]'[\Pi + \beta^{1/2}L\Lambda(I - \beta^{1/2}L\Delta_h)^{-1}\Theta_h] \\ &\quad = [\hat{\Pi} + \beta^{1/2}L^{-1}\hat{\Lambda}(I - \beta^{1/2}L^{-1}\Delta_h)^{-1}\Theta_h]'[\hat{\Pi} + \beta^{1/2}L\hat{\Lambda}(I - \beta^{1/2}L\Delta_h)^{-1}\Theta_h] \end{aligned}$$

The factorization identity guarantees that the $[\hat{\Lambda}, \hat{\Pi}]$ representation satisfies both requirements for a canonical representation.

17.5 Partial Equilibrium

Now we'll provide quick overviews of examples of economies that fit within our framework

We provide details for a number of these examples in subsequent lectures

1. *Growth in Dynamic Linear Economies*
2. *Lucas Asset Pricing using DLE*
3. *IRFs in Hall Model*
4. *Permanent Income Using the DLE class*
5. *Rosen schooling model*
6. *Cattle cycles*

7. *Shock Non Invertibility*

We'll start with an example of a **partial equilibrium** in which we posit demand and supply curves

Suppose that we want to capture the dynamic demand curve:

$$\begin{aligned} c_t &= -\Pi^{-1}\Lambda h_{t-1} + \Pi^{-1} b_t - \Pi^{-1}\mu_0^w E_t\{\Pi'^{-1} - \Pi'^{-1}\Theta_h' \\ &\quad [I - (\Delta_h' - \Lambda'\Pi'^{-1}\Theta_h')\beta L^{-1}]^{-1}\Lambda'\Pi'^{-1}\beta L^{-1}\}p_t \\ h_t &= \Delta_h h_{t-1} + \Theta_h c_t \end{aligned}$$

From material described earlier in this lecture, we know how to reverse engineer preferences that generate this demand system

- note how the demand equations are cast in terms of the matrices in our standard preference representation

Now let's turn to supply.

A representative firm takes as given and beyond its control the stochastic process $\{p_t\}_{t=0}^{\infty}$.

The firm sells its output c_t in a competitive market each period.

Only spot markets convene at each date $t \geq 0$.

The firm also faces an exogenous process of cost disturbances d_t.

The firm chooses stochastic processes $\{c_t, g_t, i_t, k_t\}_{t=0}^{\infty}$ to maximize

$$E_0 \sum_{t=0}^{\infty} \beta^t \{p_t \cdot c_t - g_t \cdot g_t/2\}$$

subject to given k_{-1} and

$$\begin{aligned} \Phi_c c_t + \Phi_i i_t + \Phi_g g_t &= \Gamma k_{t-1} + d_t \\ k_t &= \Delta_k k_{t-1} + \Theta_k i_t. \end{aligned}$$

17.6 Equilibrium Investment Under Uncertainty

A representative firm maximizes

$$E \sum_{t=0}^{\infty} \beta^t \{p_t c_t - g_t^2/2\}$$

subject to the technology

$$\begin{aligned} c_t &= \gamma k_{t-1} \\ k_t &= \delta_k k_{t-1} + i_t \\ g_t &= f_1 i_t + f_2 d_t \end{aligned}$$

where d_t is a cost shifter, $\gamma > 0$, and $f_1 > 0$ is a cost parameter and $f_2 = 1$. Demand is governed by

$$p_t = \alpha_0 - \alpha_1 c_t + u_t$$

where u_t is a demand shifter with mean zero and α_0, α_1 are positive parameters.

Assume that u_t, d_t are uncorrelated first-order autoregressive processes.

17.7 A Rosen-Topel Housing Model

$$R_t = b_t + \alpha h_t$$

$$p_t = E_t \sum_{\tau=0}^{\infty} (\beta \delta_h)^\tau R_{t+\tau}$$

where h_t is the stock of housing at time t R_t is the rental rate for housing, p_t is the price of new houses, and b_t is a demand shifter; $\alpha < 0$ is a demand parameter, and δ_h is a depreciation factor for houses.

We cast this demand specification within our class of models by letting the stock of houses h_t evolve according to

$$h_t = \delta_h h_{t-1} + c_t, \quad \delta_h \in (0, 1)$$

where c_t is the rate of production of new houses.

Houses produce services s_t according to $s_t = \bar{\lambda} h_t$ or $s_t = \lambda h_{t-1} + \pi c_t$, where $\lambda = \bar{\lambda}\delta_h, \pi = \bar{\lambda}$.

We can take $\bar{\lambda}\rho_t^0 = R_t$ as the rental rate on housing at time t, measured in units of time t consumption (housing).

Demand for housing services is

$$s_t = b_t - \mu_0 \rho_t^0$$

where the price of new houses p_t is related to ρ_t^0 by $\rho_t^0 = \pi^{-1}[p_t - \beta\delta_h E_t p_{t+1}]$.

17.8 Cattle Cycles

Rosen, Murphy, and Scheinkman (1994). Let p_t be the price of freshly slaughtered beef, m_t the feeding cost of preparing an animal for slaughter, $\tilde{h}_t$ the one-period holding cost for a mature animal, $\gamma_1 \tilde{h}_t$ the one-period holding cost for a yearling, and $\gamma_0 \tilde{h}_t$ the one-period holding cost for a calf.

The cost processes $\{\tilde{h}_t, m_t\}_{t=0}^{\infty}$ are exogenous, while the stochastic process $\{p_t\}_{t=0}^{\infty}$ is determined by a rational expectations equilibrium. Let $\tilde{x}_t$ be the breeding stock, and $\tilde{y}_t$ be the total stock of animals.

The law of motion for cattle stocks is

$$\tilde{x}_t = (1 - \delta)\tilde{x}_{t-1} + g\tilde{x}_{t-3} - c_t$$

where c_t is a rate of slaughtering. The total head-count of cattle

$$\tilde{y}_t = \tilde{x}_t + g\tilde{x}_{t-1} + g\tilde{x}_{t-2}$$

is the sum of adults, calves, and yearlings, respectively.

A representative farmer chooses $\{c_t, \tilde{x}_t\}$ to maximize

$$E_0 \sum_{t=0}^{\infty} \beta^t \{p_t c_t - \tilde{h}_t \tilde{x}_t - (\gamma_0 \tilde{h}_t)(g\tilde{x}_{t-1}) - (\gamma_1 \tilde{h}_t)(g\tilde{x}_{t-2}) - m_t c_t$$

$$- \Psi(\tilde{x}_t, \tilde{x}_{t-1}, \tilde{x}_{t-2}, c_t)\}$$

where

$$\Psi = \frac{\psi_1}{2}\tilde{x}_t^2 + \frac{\psi_2}{2}\tilde{x}_{t-1}^2 + \frac{\psi_3}{2}\tilde{x}_{t-2}^2 + \frac{\psi_4}{2}c_t^2$$

Demand is governed by

$$c_t = \alpha_0 - \alpha_1 p_t + \tilde{d}_t$$

where $\alpha_0 > 0$, $\alpha_1 > 0$, and $\{\tilde{d}_t\}_{t=0}^{\infty}$ is a stochastic process with mean zero representing a demand shifter.

For more details see *Cattle cycles*

17.9 Models of Occupational Choice and Pay

We'll describe the following pair of schooling models that view education as a time-to-build process:

- Rosen schooling model for engineers
- Two-occupation model

17.9.1 Market for Engineers

Ryoo and Rosen's (2004) [Ryoo and Rosen, 2004] model consists of the following equations:

first, a demand curve for engineers

$$w_t = -\alpha_d N_t + \epsilon_{1t} \ , \ \alpha_d > 0$$

second, a time-to-build structure of the education process

$$N_{t+k} = \delta_N N_{t+k-1} + n_t \ , \ 0 < \delta_N < 1$$

third, a definition of the discounted present value of each new engineering student

$$v_t = \beta^k E_t \sum_{j=0}^{\infty} (\beta \delta_N)^j w_{t+k+j};$$

and fourth, a supply curve of new students driven by v_t

$$n_t = \alpha_s v_t + \epsilon_{2t} \ , \ \alpha_s > 0$$

Here $\{\epsilon_{1t}, \epsilon_{2t}\}$ are stochastic processes of labor demand and supply shocks.

Definition: A partial equilibrium is a stochastic process $\{w_t, N_t, v_t, n_t\}_{t=0}^{\infty}$ satisfying these four equations, and initial conditions $N_{-1}, n_{-s}, s = 1, \ldots, -k$.

We sweep the time-to-build structure and the demand for engineers into the household technology and putting the supply of new engineers into the technology for producing goods.

$$s_t = [\lambda_1 \ 0 \ \ldots \ 0] \begin{bmatrix} h_{1t-1} \\ h_{2t-1} \\ \vdots \\ h_{k+1,t-1} \end{bmatrix} + 0 \cdot c_t$$

$$\begin{bmatrix} h_{1t} \\ h_{2t} \\ \vdots \\ h_{k,t} \\ h_{k+1,t} \end{bmatrix} = \begin{bmatrix} \delta_N & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & \cdots & 0 & 1 \\ 0 & 0 & 0 & \cdots & 0 \end{bmatrix} \begin{bmatrix} h_{1t-1} \\ h_{2t-1} \\ \vdots \\ h_{k,t-1} \\ h_{k+1,t-1} \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix} c_t$$

This specification sets Rosen's $N_t = h_{1t-1}, n_t = c_t, h_{\tau+1,t-1} = n_{t-\tau}, \tau = 1, \ldots, k$, and uses the home-produced service to capture the demand for labor. Here λ_1 embodies Rosen's demand parameter α_d.

- The supply of new workers becomes our consumption.
- The dynamic demand curve becomes Rosen's dynamic supply curve for new workers.

Remark: This has an Imai-Keane flavor.

For more details and Python code see *Rosen schooling model*.

17.9.2 Skilled and Unskilled Workers

First, a demand curve for labor

$$\begin{bmatrix} w_{ut} \\ w_{st} \end{bmatrix} = \alpha_d \begin{bmatrix} N_{ut} \\ N_{st} \end{bmatrix} + \epsilon_{1t}$$

where α_d is a (2×2) matrix of demand parameters and ϵ_{1t} is a vector of demand shifters second, time-to-train specifications for skilled and unskilled labor, respectively:

$$\begin{aligned} N_{st+k} &= \delta_N N_{st+k-1} + n_{st} \\ N_{ut} &= \delta_N N_{ut-1} + n_{ut}; \end{aligned}$$

where N_{st}, N_{ut} are stocks of the two types of labor, and n_{st}, n_{ut} are entry rates into the two occupations.

third, definitions of discounted present values of new entrants to the skilled and unskilled occupations, respectively:

$$\begin{aligned} v_{st} &= E_t \beta^k \sum_{j=0}^{\infty} (\beta \delta_N)^j w_{st+k+j} \\ v_{ut} &= E_t \sum_{j=0}^{\infty} (\beta \delta_N)^j w_{ut+j} \end{aligned}$$

where w_{ut}, w_{st} are wage rates for the two occupations; and fourth, supply curves for new entrants:

$$\begin{bmatrix} n_{st} \\ n_{ut} \end{bmatrix} = \alpha_s \begin{bmatrix} v_{ut} \\ v_{st} \end{bmatrix} + \epsilon_{2t}$$

Short Cut

As an alternative, Siow simply used the **equalizing differences** condition

$$v_{ut} = v_{st}$$

17.10 Permanent Income Models

We'll describe a class of permanent income models that feature

- Many consumption goods and services
- A single capital good with $R\beta = 1$
- The physical production technology

$$\begin{aligned} \phi_c \cdot c_t + i_t &= \gamma k_{t-1} + e_t \\ k_t &= k_{t-1} + i_t \\ \phi_i i_t - g_t &= 0 \end{aligned}$$

Implication One:

Equality of Present Values of Moving Average Coefficients of c and e

$$k_{t-1} = \beta \sum_{j=0}^{\infty} \beta^j (\phi_c \cdot c_{t+j} - e_{t+j}) \quad \Rightarrow$$

$$k_{t-1} = \beta \sum_{j=0}^{\infty} \beta^j E(\phi_c \cdot c_{t+j} - e_{t+j}) | J_t \quad \Rightarrow$$

$$\sum_{j=0}^{\infty} \beta^j (\phi_c)' \chi_j = \sum_{j=0}^{\infty} \beta^j \epsilon_j$$

where $\chi_j w_t$ is the response of c_{t+j} to w_t and $\epsilon_j w_t$ is the response of endowment e_{t+j} to w_t:

Implication Two:

Martingales

$$\mathcal{M}_t^k = E(\mathcal{M}_{t+1}^k | J_t)$$
$$\mathcal{M}_t^e = E(\mathcal{M}_{t+1}^e | J_t)$$

and

$$\mathcal{M}_t^c = (\Phi_c)' \mathcal{M}_t^d = \phi_c M_t^e$$

For more details see *Permanent Income Using the DLE class*

Testing Permanent Income Models:

We have two types of implications of permanent income models:

- Equality of present values of moving average coefficients.
- Martingale $\mathcal{M}_t^k$.

These have been tested in work by Hansen, Sargent, and Roberts (1991) [Sargent *et al.*, 1991] and by Attanasio and Pavoni (2011) [Attanasio and Pavoni, 2011].

17.11 Gorman Heterogeneous Households

We now assume that there is a finite number of households, each with its own household technology and preferences over consumption services.

Household j orders preferences over consumption processes according to

$$-\left(\frac{1}{2}\right) E \sum_{t=0}^{\infty} \beta^t \left[(s_{jt} - b_{jt}) \cdot (s_{jt} - b_{jt}) + \ell_{jt}^2\right] \mid J_0$$

$$s_{jt} = \Lambda \, h_{j,t-1} + \Pi \, c_{jt}$$

$$h_{jt} = \Delta_h \, h_{j,t-1} + \Theta_h \, c_{jt}$$

and $h_{j,-1}$ is given

$$b_{jt} = U_{bj} z_t$$

$$E \sum_{t=0}^{\infty} \beta^t \, p_t^0 \cdot c_{jt} \mid J_0 = E \sum_{t=0}^{\infty} \beta^t \left(w_t^0 \, \ell_{jt} + \alpha_t^0 \cdot d_{jt}\right) \mid J_0 + v_0 \cdot k_{j,-1},$$

where $k_{j,-1}$ is given. The j^{th} consumer owns an endowment process d_{jt}, governed by the stochastic process $d_{jt} = U_{dj} \, z_t$.

We refer to this as a setting with Gorman heterogeneous households.

This specification confines heterogeneity among consumers to:

- differences in the preference processes $\{b_{jt}\}$, represented by different selections of U_{bj}
- differences in the endowment processes $\{d_{jt}\}$, represented by different selections of U_{dj}
- differences in $h_{j,-1}$ and

- differences in $k_{j,-1}$

The matrices Λ, Π, Δ_h, Θ_h do not depend on j.

This makes everybody's demand system have the form described earlier, with different μ_{j0}^w's (reflecting different wealth levels) and different b_{jt} preference shock processes and initial conditions for household capital stocks.

Punchline: there exists a representative consumer.

We can use the representative consumer to compute a competitive equilibrium **aggregate** allocation and price system.

With the equilibrium aggregate allocation and price system in hand, we can then compute allocations to each household.

Computing Allocations to Individuals:

Set

$$\ell_{jt} = (\mu_{0j}^w / \mu_{0a}^w)\ell_{at}$$

Then solve the following equation for μ_{0j}^w:

$$\mu_{0j}^w E_0 \sum_{t=0}^{\infty} \beta^t \{\rho_t^0 \cdot \rho_t^0 + (w_t^0/\mu_{0a}^w)\ell_{at}\} = E_0 \sum_{t=0}^{\infty} \beta^t \{\rho_t^0 \cdot (b_{jt} - s_{jt}^i) - \alpha_t^0 \cdot d_{jt}\} - v_0 k_{j,-1}$$

$$s_{jt} - b_{jt} = \mu_{0j}^w \rho_t^0$$

$$c_{jt} = -\Pi^{-1}\Lambda h_{j,t-1} + \Pi^{-1} s_{jt}$$

$$h_{jt} = (\Delta_h - \Theta_h \Pi^{-1}\Lambda) h_{j,t-1} + \Pi^{-1}\Theta_h s_{jt}$$

Here $h_{j,-1}$ given.

17.12 Non-Gorman Heterogeneous Households

We now describe a less tractable type of heterogeneity across households that we dub **Non-Gorman heterogeneity**.

Here is the specification:

Preferences and Household Technologies:

$$-\frac{1}{2} E \sum_{t=0}^{\infty} \beta^t \left[(s_{it} - b_{it}) \cdot (s_{it} - b_{it}) + \ell_{it}^2\right] \mid J_0$$

$$s_{it} = \Lambda_i h_{it-1} + \Pi_i \, c_{it}$$

$$h_{it} = \Delta_{h_i} \, h_{it-1} + \Theta_{h_i} c_{it} \, , \; i = 1, 2.$$

$$b_{it} = U_{bi} z_t$$

$$z_{t+1} = A_{22} z_t + C_2 w_{t+1}$$

Production Technology

$$\Phi_c(c_{1t} + c_{2t}) + \Phi_g g_t + \Phi_i i_t = \Gamma k_{t-1} + d_{1t} + d_{2t}$$

$$k_t = \Delta_k k_{t-1} + \Theta_k i_t$$

$$g_t \cdot g_t = \ell_t^2, \qquad \ell_t = \ell_{1t} + \ell_{2t}$$

$$d_{it} = U_{d_i} z_t, \quad i = 1, 2$$

Pareto Problem:

$$-\frac{1}{2}\lambda E_0\sum_{t=0}^{\infty}\beta^t[(s_{1t}-b_{1t})\cdot(s_{1t}-b_{1t})+\ell_{1t}^2]$$
$$-\frac{1}{2}(1-\lambda)E_0\sum_{t=0}^{\infty}\beta^t[(s_{2t}-b_{2t})\cdot(s_{2t}-b_{2t})+\ell_{2t}^2]$$

Mongrel Aggregation: Static

There is what we call a kind of **mongrel aggregation** in this setting.

We first describe the idea within a simple static setting in which there is a single consumer static inverse demand with implied preferences:

$$c_t=\Pi^{-1}b_t-\mu_0\Pi^{-1}\Pi^{-1\prime}p_t$$

An inverse demand curve is

$$p_t=\mu_0^{-1}\Pi' b_t-\mu_0^{-1}\Pi'\Pi c_t$$

Integrating the marginal utility vector shows that preferences can be taken to be

$$(-2\mu_0)^{-1}(\Pi c_t-b_t)\cdot(\Pi c_t-b_t)$$

Key Insight: Factor the inverse of a 'covariance matrix'.

Now assume that there are two consumers, $i=1,2$, with demand curves

$$c_{it}=\Pi_i^{-1}b_{it}-\mu_{0i}\Pi_i^{-1}\Pi_i^{-1\prime}p_t$$

$$c_{1t}+c_{2t}=(\Pi_1^{-1}b_{1t}+\Pi_2^{-1}b_{2t})-(\mu_{01}\Pi_1^{-1}\Pi_1^{-1\prime}+\mu_{02}\Pi_2\Pi_2^{-1\prime})p_t$$

Setting $c_{1t}+c_{2t}=c_t$ and solving for p_t gives

$$\begin{aligned}p_t=&(\mu_{01}\Pi_1^{-1}\Pi_1^{-1\prime}+\mu_{02}\Pi_2^{-1}\Pi_2^{-1\prime})^{-1}(\Pi_1^{-1}b_{1t}+\Pi_2^{-1}b_{2t})\\&-(\mu_{01}\Pi_1^{-1}\Pi_1^{-1\prime}+\mu_{02}\Pi_2^{-1}\Pi_2^{-1\prime})^{-1}c_t\end{aligned}$$

Punchline: choose Π associated with the aggregate ordering to satisfy

$$\mu_0^{-1}\Pi'\Pi=(\mu_{01}\Pi_1^{-1}\Pi_2^{-1\prime}+\mu_{02}\Pi_2^{-1}\Pi_2^{-1\prime})^{-1}$$

Dynamic Analogue:

We now describe how to extend mongrel aggregation to a dynamic setting.

The key comparison is

- Static: factor a covariance matrix-like object
- Dynamic: factor a spectral-density matrix-like object

Programming Problem for Dynamic Mongrel Aggregation:

Our strategy for deducing the mongrel preference ordering over $c_t=c_{1t}+c_{2t}$ is to solve the programming problem: choose $\{c_{1t},c_{2t}\}$ to maximize the criterion

$$\sum_{t=0}^{\infty}\beta^t[\lambda(s_{1t}-b_{1t})\cdot(s_{1t}-b_{1t})+(1-\lambda)(s_{2t}-b_{2t})\cdot(s_{2t}-b_{2t})]$$

subject to

$$
\begin{aligned}
h_{jt} &= \Delta_{hj}\, h_{jt-1} + \Theta_{hj}\, c_{jt}, j = 1, 2 \\
s_{jt} &= \Delta_j h_{jt-1} + \Pi_j c_{jt}\ , j = 1, 2 \\
c_{1t} + c_{2t} &= c_t
\end{aligned}
$$

subject to $(h_{1,-1},\, h_{2,-1})$ given and $\{b_{1t}\}$, $\{b_{2t}\}$, $\{c_t\}$ being known and fixed sequences.

Substituting the $\{c_{1t},\, c_{2t}\}$ sequences that solve this problem as functions of $\{b_{1t},\, b_{2t},\, c_t\}$ into the objective determines a mongrel preference ordering over $\{c_t\} = \{c_{1t} + c_{2t}\}$.

In solving this problem, it is convenient to proceed by using Fourier transforms. For details, please see [Hansen and Sargent, 2013] where they deploy a

Secret Weapon: Another application of the spectral factorization identity.

Concluding remark: The [Hansen and Sargent, 2013] class of models described in this lecture are all complete markets models. We have exploited the fact that complete market models **are all alike** to allow us to define a class that **gives the same name to different things** in the spirit of Henri Poincare.

Could we create such a class for **incomplete markets** models?

That would be nice, but before trying it would be wise to contemplate the remainder of a statement by Robert E. Lucas, Jr., with which we began this lecture.

> "Complete market economies are all alike but each incomplete market economy is incomplete in its own individual way." Robert E. Lucas, Jr., (1989)

CHAPTER

EIGHTEEN

GROWTH IN DYNAMIC LINEAR ECONOMIES

This is another member of a suite of lectures that use the quantecon DLE class to instantiate models within the [Hansen and Sargent, 2013] class of models described in detail in *Recursive Models of Dynamic Linear Economies*.

In addition to what's included in Anaconda, this lecture uses the quantecon library.

```
!pip install --upgrade quantecon
```

This lecture describes several complete market economies having a common linear-quadratic-Gaussian structure.

Three examples of such economies show how the DLE class can be used to compute equilibria of such economies in Python and to illustrate how different versions of these economies can or cannot generate sustained growth.

We require the following imports

```
import numpy as np
import matplotlib.pyplot as plt
from quantecon import DLE
```

18.1 Common Structure

Our example economies have the following features

- Information flows are governed by an exogenous stochastic process z_t that follows

$$z_{t+1} = A_{22} z_t + C_2 w_{t+1}$$

  where w_{t+1} is a martingale difference sequence.

- Preference shocks b_t and technology shocks d_t are linear functions of z_t

$$b_t = U_b z_t$$

$$d_t = U_d z_t$$

- Consumption and physical investment goods are produced using the following technology

$$\Phi_c c_t + \Phi_g g_t + \Phi_i i_t = \Gamma k_{t-1} + d_t$$

$$k_t = \Delta_k k_{t-1} + \Theta_k i_t$$

$$g_t \cdot g_t = l_t^2$$

  where c_t is a vector of consumption goods, g_t is a vector of intermediate goods, i_t is a vector of investment goods, k_t is a vector of physical capital goods, and l_t is the amount of labor supplied by the representative household.

- Preferences of a representative household are described by

$$-\frac{1}{2}\mathbb{E}\sum_{t=0}^{\infty}\beta^t[(s_t-b_t)\cdot(s_t-b_t)+l_t^2], 0<\beta<1$$

$$s_t=\Lambda h_{t-1}+\Pi c_t$$

$$h_t=\Delta_h h_{t-1}+\Theta_h c_t$$

where s_t is a vector of consumption services, and h_t is a vector of household capital stocks.

Thus, an instance of this class of economies is described by the matrices

$$\{A_{22},C_2,U_b,U_d,\Phi_c,\Phi_g,\Phi_i,\Gamma,\Delta_k,\Theta_k,\Lambda,\Pi,\Delta_h,\Theta_h\}$$

and the scalar β.

18.2 A Planning Problem

The first welfare theorem asserts that a competitive equilibrium allocation solves the following planning problem.

Choose $\{c_t,s_t,i_t,h_t,k_t,g_t\}_{t=0}^{\infty}$ to maximize

$$-\frac{1}{2}\mathbb{E}\sum_{t=0}^{\infty}\beta^t[(s_t-b_t)\cdot(s_t-b_t)+g_t\cdot g_t]$$

subject to the linear constraints

$$\Phi_c c_t+\Phi_g g_t+\Phi_i i_t=\Gamma k_{t-1}+d_t$$

$$k_t=\Delta_k k_{t-1}+\Theta_k i_t$$

$$h_t=\Delta_h h_{t-1}+\Theta_h c_t$$

$$s_t=\Lambda h_{t-1}+\Pi c_t$$

and

$$z_{t+1}=A_{22}z_t+C_2 w_{t+1}$$

$$b_t=U_b z_t$$

$$d_t=U_d z_t$$

The DLE class in Python maps this planning problem into a linear-quadratic dynamic programming problem and then solves it by using QuantEcon's LQ class.

(See Section 5.5 of Hansen & Sargent (2013) [Hansen and Sargent, 2013] for a full description of how to map these economies into an LQ setting, and how to use the solution to the LQ problem to construct the output matrices in order to simulate the economies)

The state for the LQ problem is

$$x_t=\begin{bmatrix}h_{t-1}\\k_{t-1}\\z_t\end{bmatrix}$$

and the control variable is $u_t = i_t$.

Once the LQ problem has been solved, the law of motion for the state is

$$x_{t+1} = (A - BF)x_t + Cw_{t+1}$$

where the optimal control law is $u_t = -Fx_t$.

Letting $A^o = A - BF$ we write this law of motion as

$$x_{t+1} = A^o x_t + Cw_{t+1}$$

18.3 Example Economies

Each of the example economies shown here will share a number of components. In particular, for each we will consider preferences of the form

$$-\frac{1}{2}\mathbb{E}\sum_{t=0}^{\infty}\beta^t[(s_t - b_t)^2 + l_t^2], 0 < \beta < 1$$

$$s_t = \lambda h_{t-1} + \pi c_t$$

$$h_t = \delta_h h_{t-1} + \theta_h c_t$$

$$b_t = U_b z_t$$

Technology of the form

$$c_t + i_t = \gamma_1 k_{t-1} + d_{1t}$$

$$k_t = \delta_k k_{t-1} + i_t$$

$$g_t = \phi_1 i_t \, , \phi_1 > 0$$

$$\begin{bmatrix} d_{1t} \\ 0 \end{bmatrix} = U_d z_t$$

And information of the form

$$z_{t+1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0.8 & 0 \\ 0 & 0 & 0.5 \end{bmatrix} z_t + \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} w_{t+1}$$

$$U_b = \begin{bmatrix} 30 & 0 & 0 \end{bmatrix}$$

$$U_d = \begin{bmatrix} 5 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

We shall vary $\{\lambda, \pi, \delta_h, \theta_h, \gamma_1, \delta_k, \phi_1\}$ and the initial state x_0 across the three economies.

18.3.1 Example 1: Hall (1978)

First, we set parameters such that consumption follows a random walk. In particular, we set

$$\lambda = 0, \pi = 1, \gamma_1 = 0.1, \phi_1 = 0.00001, \delta_k = 0.95, \beta = \frac{1}{1.05}$$

(In this economy δ_h and θ_h are arbitrary as household capital does not enter the equation for consumption services We set them to values that will become useful in Example 3)

It is worth noting that this choice of parameter values ensures that $\beta(\gamma_1 + \delta_k) = 1$.

For simulations of this economy, we choose an initial condition of

$$x_0 = \begin{bmatrix} 5 & 150 & 1 & 0 & 0 \end{bmatrix}'$$

```
# Parameter Matrices
γ_1 = 0.1
ϕ_1 = 1e-5

ϕ_c, ϕ_g, ϕ_i, γ, δ_k, θ_k = (np.array([[1], [0]]),
                               np.array([[0], [1]]),
                               np.array([[1], [-ϕ_1]]),
                               np.array([[γ_1], [0]]),
                               np.array([[.95]]),
                               np.array([[1]]))

β, l_λ, π_h, δ_h, θ_h = (np.array([[1 / 1.05]]),
                         np.array([[0]]),
                         np.array([[1]]),
                         np.array([[.9]]),
                         np.array([[1]]) - np.array([[.9]]))

a22, c2, ub, ud = (np.array([[1,   0,   0],
                             [0, 0.8,   0],
                             [0,   0, 0.5]]),
                   np.array([[0, 0],
                             [1, 0],
                             [0, 1]]),
                   np.array([[30, 0, 0]]),
                   np.array([[5, 1, 0],
                             [0, 0, 0]]))

# Initial condition
x0 = np.array([[5], [150], [1], [0], [0]])

info1 = (a22, c2, ub, ud)
tech1 = (ϕ_c, ϕ_g, ϕ_i, γ, δ_k, θ_k)
pref1 = (β, l_λ, π_h, δ_h, θ_h)
```

These parameter values are used to define an economy of the DLE class.

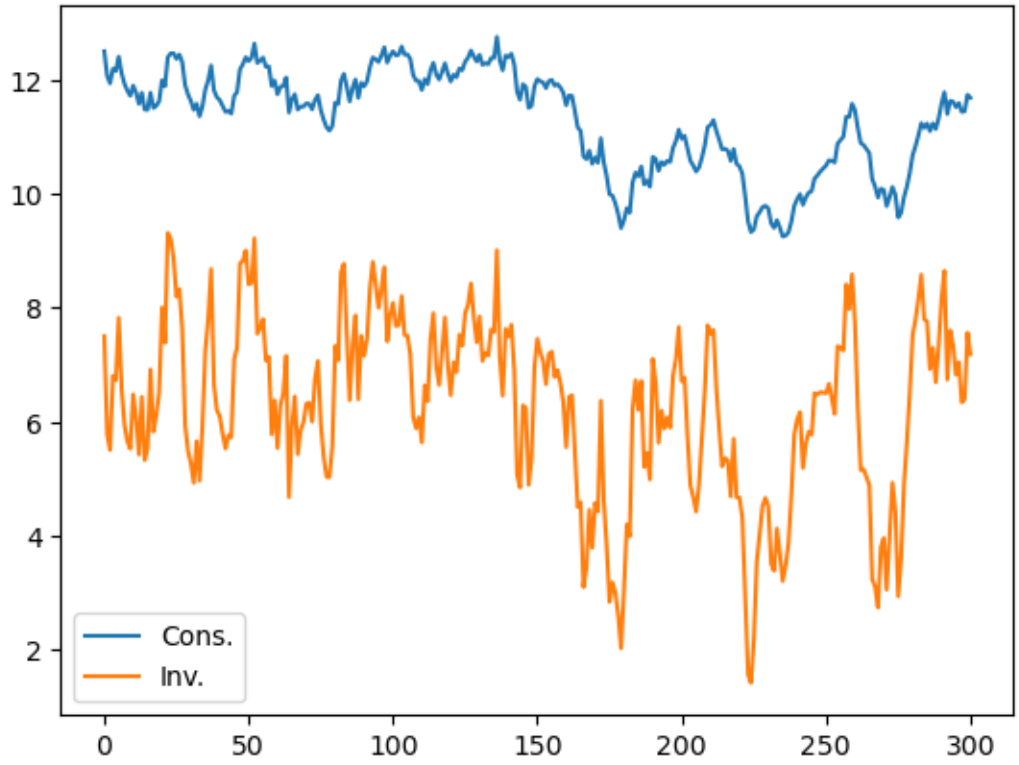
```
econ1 = DLE(info1, tech1, pref1)
```

We can then simulate the economy for a chosen length of time, from our initial state vector x_0

```
econ1.compute_sequence(x0, ts_length=300)
```

The economy stores the simulated values for each variable. Below we plot consumption and investment

```
# This is the right panel of Fig 5.7.1 from p.105 of HS2013
plt.plot(econ1.c[0], label='Cons.')
plt.plot(econ1.i[0], label='Inv.')
plt.legend()
plt.show()
```



Inspection of the plot shows that the sample paths of consumption and investment drift in ways that suggest that each has or nearly has a **random walk** or **unit root** component.

This is confirmed by checking the eigenvalues of A^o

```
econ1.endo, econ1.exo
```

```
(array([0.9, 1. ]), array([1. , 0.8, 0.5]))
```

The endogenous eigenvalue that appears to be unity reflects the random walk character of consumption in Hall's model.

- Actually, the largest endogenous eigenvalue is very slightly below 1.
- This outcome comes from the small adjustment cost ϕ_1.

```
econ1.endo[1]
```

```
0.9999999999904767
```

The fact that the largest endogenous eigenvalue is strictly less than unity in modulus means that it is possible to compute the non-stochastic steady state of consumption, investment and capital.

```
econ1.compute_steadystate()
np.set_printoptions(precision=3, suppress=True)
print(econ1.css, econ1.iss, econ1.kss)
```

```
[[4.999]] [[-0.001]] [[-0.023]]
```

However, the near-unity endogenous eigenvalue means that these steady state values are of little relevance.

18.3.2 Example 2: Altered Growth Condition

We generate our next economy by making two alterations to the parameters of Example 1.

- First, we raise ϕ_1 from 0.00001 to 1.
  - This will lower the endogenous eigenvalue that is close to 1, causing the economy to head more quickly to the vicinity of its non-stochastic steady-state.
- Second, we raise γ_1 from 0.1 to 0.15.
  - This has the effect of raising the optimal steady-state value of capital.

We also start the economy off from an initial condition with a lower capital stock

$$x_0 = \begin{bmatrix} 5 & 20 & 1 & 0 & 0 \end{bmatrix}'$$

Therefore, we need to define the following new parameters

```
γ2 = 0.15
γ22 = np.array([[γ2], [0]])

ϕ_12 = 1
ϕ_i2 = np.array([[1], [-ϕ_12]])

tech2 = (ϕ_c, ϕ_g, ϕ_i2, γ22, δ_k, θ_k)

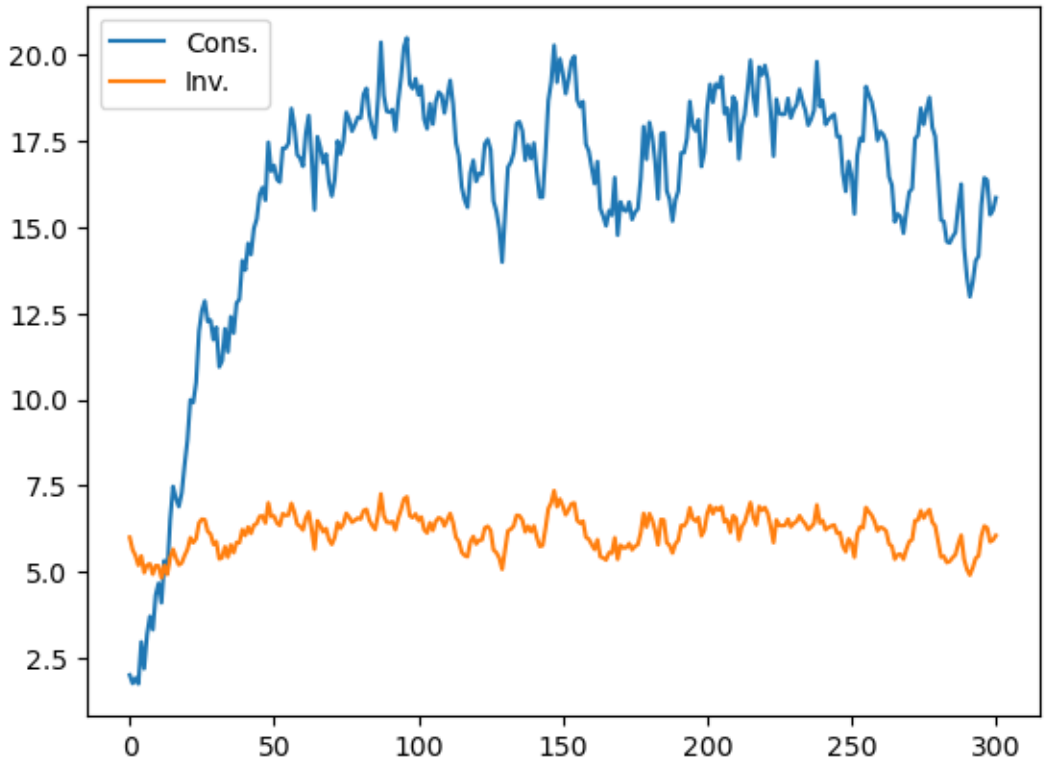
x02 = np.array([[5], [20], [1], [0], [0]])
```

Creating the DLE class and then simulating gives the following plot for consumption and investment

```
econ2 = DLE(info1, tech2, pref1)

econ2.compute_sequence(x02, ts_length=300)

plt.plot(econ2.c[0], label='Cons.')
plt.plot(econ2.i[0], label='Inv.')
plt.legend()
plt.show()
```



Simulating our new economy shows that consumption grows quickly in the early stages of the sample.

However, it then settles down around the new non-stochastic steady-state level of consumption of 17.5, which we find as follows

```
econ2.compute_steadystate()
print(econ2.css, econ2.iss, econ2.kss)
```

```
[[17.5]] [[6.25]] [[125.]]
```

The economy converges faster to this level than in Example 1 because the largest endogenous eigenvalue of A^o is now significantly lower than 1.

```
econ2.endo, econ2.exo
```

```
(array([0.9  , 0.952]), array([1. , 0.8, 0.5]))
```

18.3.3 Example 3: A Jones-Manuelli (1990) Economy

For our third economy, we choose parameter values with the aim of generating *sustained* growth in consumption, investment and capital.

To do this, we set parameters so that Jones and Manuelli's "growth condition" is just satisfied.

In our notation, just satisfying the growth condition is actually equivalent to setting $\beta(\gamma_1 + \delta_k) = 1$, the condition that was necessary for consumption to be a random walk in Hall's model.

Thus, we lower γ_1 back to 0.1.

In our model, this is a necessary but not sufficient condition for growth.

To generate growth we set preference parameters to reflect habit persistence.

In particular, we set $\lambda = -1$, $\delta_h = 0.9$ and $\theta_h = 1 - \delta_h = 0.1$.

This makes preferences assume the form

$$-\frac{1}{2}\mathbb{E}\sum_{t=0}^{\infty}\beta^t[(c_t - b_t - (1-\delta_h)\sum_{j=0}^{\infty}\delta_h^j c_{t-j-1})^2 + l_t^2]$$

These preferences reflect habit persistence

- the effective "bliss point" $b_t + (1-\delta_h)\sum_{j=0}^{\infty}\delta_h^j c_{t-j-1}$ now shifts in response to a moving average of past consumption

Since δ_h and θ_h were defined earlier, the only change we need to make from the parameters of Example 1 is to define the new value of λ.

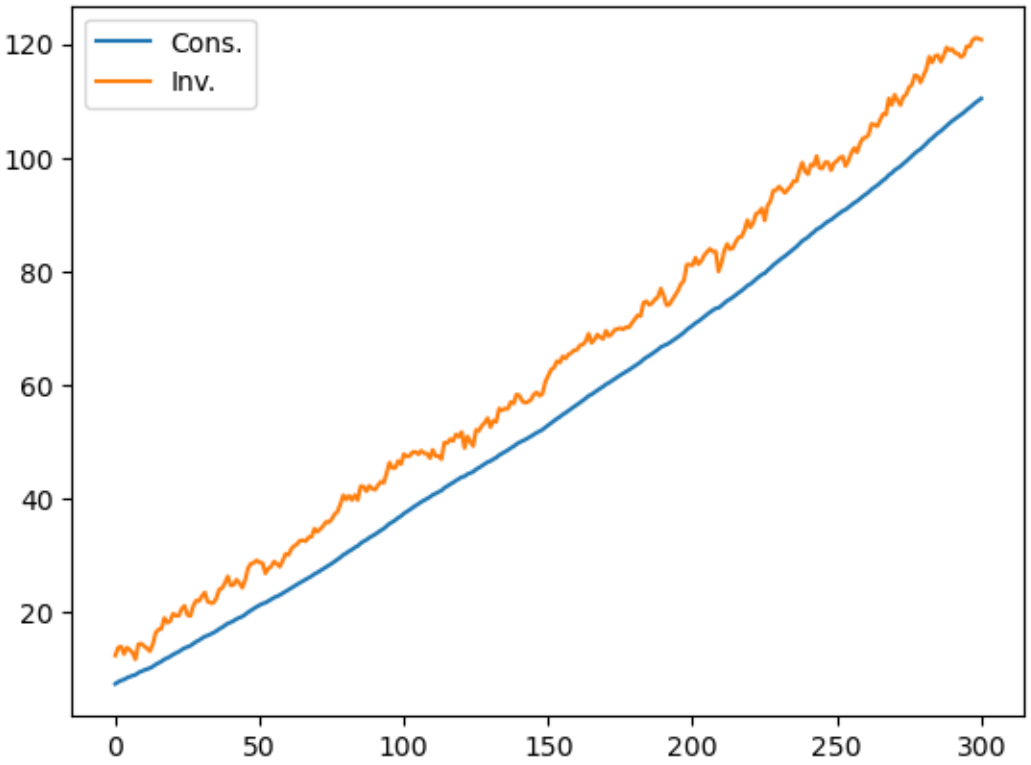
```
l_λ2 = np.array([[-1]])
pref2 = (β, l_λ2, π_h, δ_h, θ_h)
```

```
econ3 = DLE(info1, tech1, pref2)
```

We simulate this economy from the original state vector

```
econ3.compute_sequence(x0, ts_length=300)

# This is the right panel of Fig 5.10.1 from p.110 of HS2013
plt.plot(econ3.c[0], label='Cons.')
plt.plot(econ3.i[0], label='Inv.')
plt.legend()
plt.show()
```



Thus, adding habit persistence to the Hall model of Example 1 is enough to generate sustained growth in our economy.

The eigenvalues of A^o in this new economy are

```
econ3.endo, econ3.exo
```

```
(array([1.+0.j, 1.-0.j]), array([1. , 0.8, 0.5]))
```

We now have two unit endogenous eigenvalues. One stems from satisfying the growth condition (as in Example 1).

The other unit eigenvalue results from setting $\lambda = -1$.

To show the importance of both of these for generating growth, we consider the following experiments.

18.3.4 Example 3.1: Varying Sensitivity

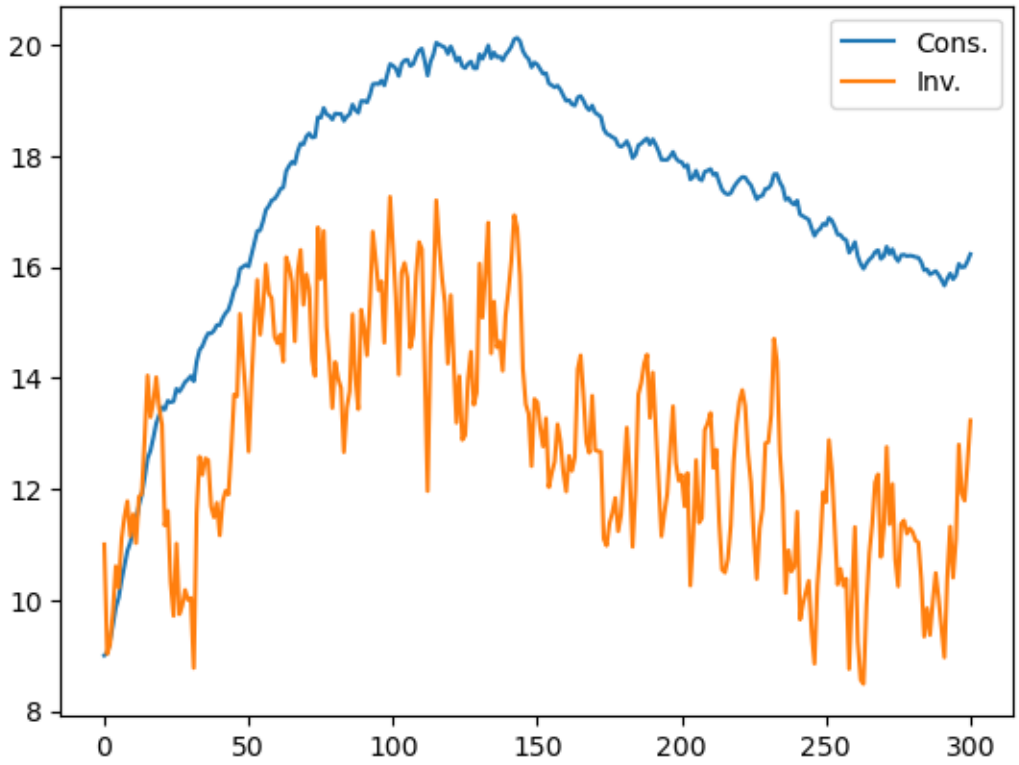
Next we raise λ to -0.7

```
l_λ3 = np.array([[-0.7]])
pref3 = (β, l_λ3, π_h, δ_h, θ_h)

econ4 = DLE(info1, tech1, pref3)

econ4.compute_sequence(x0, ts_length=300)

plt.plot(econ4.c[0], label='Cons.')
plt.plot(econ4.i[0], label='Inv.')
plt.legend()
plt.show()
```



We no longer achieve sustained growth if λ is raised from -1 to -0.7.

This is related to the fact that one of the endogenous eigenvalues is now less than 1.

```
econ4.endo, econ4.exo
```

```
(array([0.97, 1.  ]), array([1. , 0.8, 0.5]))
```

18.3.5 Example 3.2: More Impatience

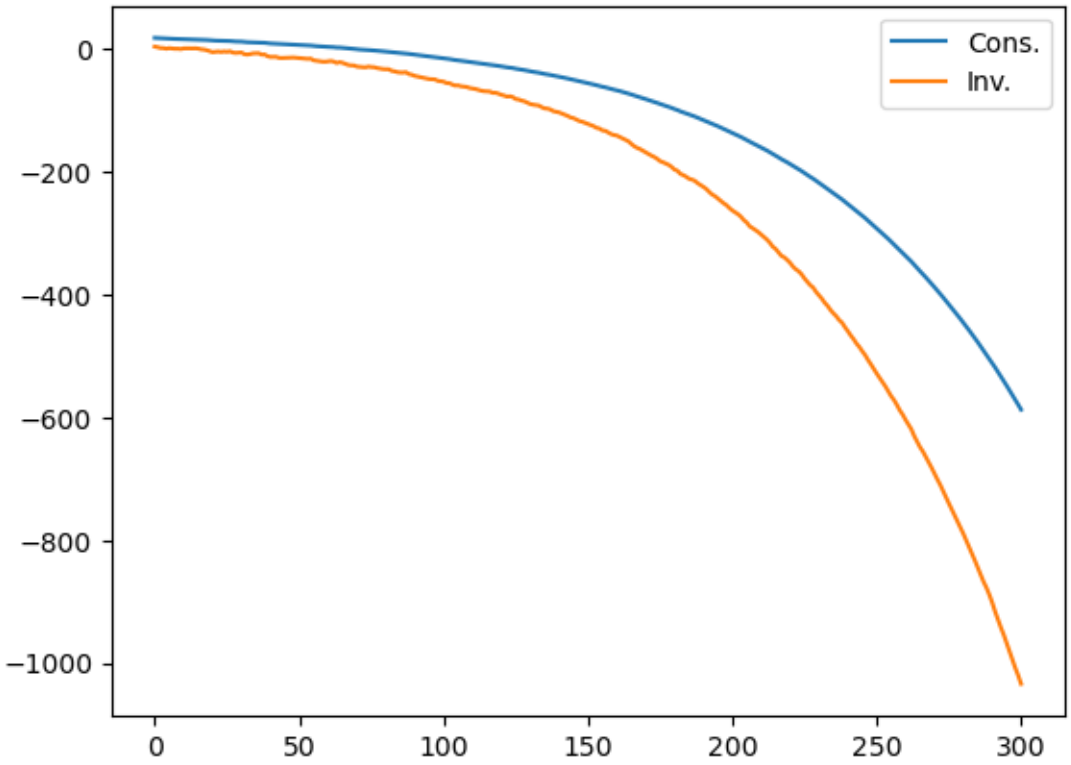
Next let's lower β to 0.94

```
β_2 = np.array([[0.94]])
pref4 = (β_2, l_λ, π_h, δ_h, θ_h)

econ5 = DLE(info1, tech1, pref4)

econ5.compute_sequence(x0, ts_length=300)

plt.plot(econ5.c[0], label='Cons.')
plt.plot(econ5.i[0], label='Inv.')
plt.legend()
plt.show()
```



Growth also fails if we lower β, since we now have $\beta(\gamma_1 + \delta_k) < 1$.

Consumption and investment explode downwards, as a lower value of β causes the representative consumer to front-load consumption.

This explosive path shows up in the second endogenous eigenvalue now being larger than one.

```
econ5.endo, econ5.exo
```

```
(array([0.9  , 1.013]), array([1. , 0.8, 0.5]))
```

CHAPTER

NINETEEN

LUCAS ASSET PRICING USING DLE

This is one of a suite of lectures that use the quantecon DLE class to instantiate models within the [Hansen and Sargent, 2013] class of models described in detail in *Recursive Models of Dynamic Linear Economies*.

In addition to what's in Anaconda, this lecture uses the quantecon library

```
!pip install --upgrade quantecon
```

This lecture uses the DLE class to price payout streams that are linear functions of the economy's state vector, as well as risk-free assets that pay out one unit of the first consumption good with certainty.

We assume basic knowledge of the class of economic environments that fall within the domain of the DLE class.

Many details about the basic environment are contained in the lecture *Growth in Dynamic Linear Economies*.

We'll also need the following imports

```
import numpy as np
import matplotlib.pyplot as plt
from quantecon import DLE
```

We use a linear-quadratic version of an economy that Lucas (1978) [Lucas, 1978] used to develop an equilibrium theory of asset prices:

Preferences

$$-\frac{1}{2}\mathbb{E}\sum_{t=0}^{\infty}\beta^t[(c_t-b_t)^2+l_t^2]|J_0$$

$$s_t = c_t$$

$$b_t = U_b z_t$$

Technology

$$c_t = d_{1t}$$

$$k_t = \delta_k k_{t-1} + i_t$$

$$g_t = \phi_1 i_t\,, \phi_1 > 0$$

$$\begin{bmatrix} d_{1t} \\ 0 \end{bmatrix} = U_d z_t$$

Information

$$z_{t+1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0.8 & 0 \\ 0 & 0 & 0.5 \end{bmatrix} z_t + \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} w_{t+1}$$

$$U_b = \begin{bmatrix} 30 & 0 & 0 \end{bmatrix}$$
$$U_d = \begin{bmatrix} 5 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$
$$x_0 = \begin{bmatrix} 5 & 150 & 1 & 0 & 0 \end{bmatrix}'$$

19.1 Asset Pricing Equations

[Hansen and Sargent, 2013] show that the time t value of a permanent claim to a stream $y_s = U_a x_s \, , s \geq t$ is:

$$a_t = (x_t' \mu_a x_t + \sigma_a)/(\bar{e}_1 M_c x_t)$$

with

$$\mu_a = \sum_{\tau=0}^{\infty} \beta^\tau (A^{o'})^\tau Z_a A^{o\tau}$$

$$\sigma_a = \frac{\beta}{1-\beta} \text{trace}(Z_a \sum_{\tau=0}^{\infty} \beta^\tau (A^o)^\tau C C' (A^{o'})^\tau)$$

where

$$Z_a = U_a' M_c$$

The use of $\bar{e}_1$ indicates that the first consumption good is the numeraire.

19.2 Asset Pricing Simulations

```
gam = 0
γ = np.array([[gam], [0]])
ϕ_c = np.array([[1], [0]])
ϕ_g = np.array([[0], [1]])
ϕ_1 = 1e-4
ϕ_i = np.array([[0], [-ϕ_1]])
δ_k = np.array([[.95]])
θ_k = np.array([[1]])
β = np.array([[1 / 1.05]])
ud = np.array([[5, 1, 0],
               [0, 0, 0]])
a22 = np.array([[1,   0,    0],
                [0, 0.8,    0],
                [0,   0, 0.5]])
c2 = np.array([[0, 1, 0],
               [0, 0, 1]]).T
l_λ = np.array([[0]])
π_h = np.array([[1]])
δ_h = np.array([[.9]])
θ_h = np.array([[1]]) - δ_h
ub = np.array([[30, 0, 0]])
x0 = np.array([[5, 150, 1, 0, 0]]).T

info1 = (a22, c2, ub, ud)
tech1 = (ϕ_c, ϕ_g, ϕ_i, γ, δ_k, θ_k)
pref1 = (β, l_λ, π_h, δ_h, θ_h)
```

```
econ1 = DLE(info1, tech1, pref1)
```

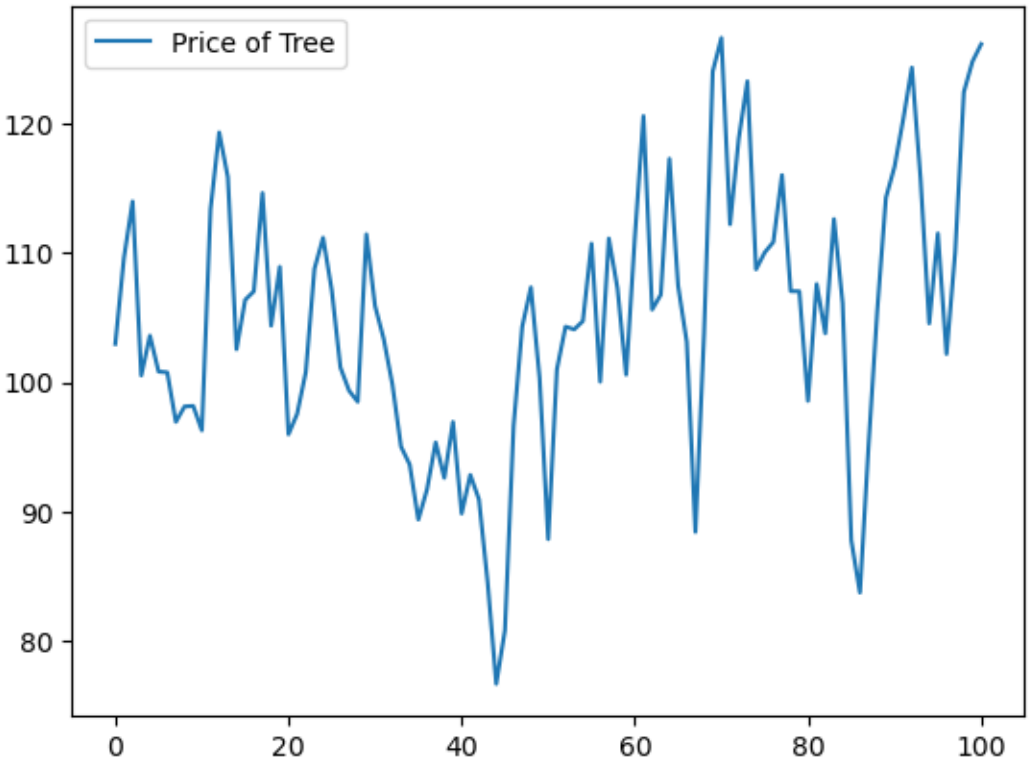
After specifying a "Pay" matrix, we simulate the economy.

The particular choice of "Pay" used below means that we are pricing a perpetual claim on the endowment process d_{1t}

```
econ1.compute_sequence(x0, ts_length=100, Pay=np.array([econ1.Sd[0, :]]))
```

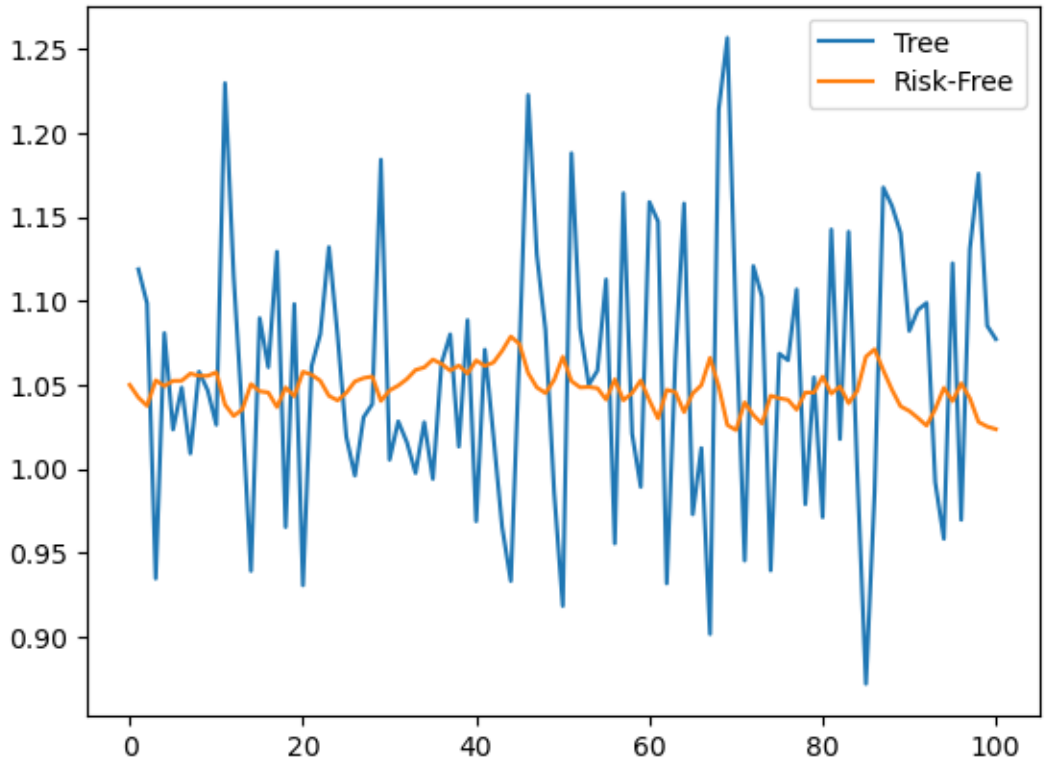
The graph below plots the price of this claim over time:

```
### Fig 7.12.1 from p.147 of HS2013
plt.plot(econ1.Pay_Price, label='Price of Tree')
plt.legend()
plt.show()
```



The next plot displays the realized gross rate of return on this "Lucas tree" as well as on a risk-free one-period bond:

```
### Left panel of Fig 7.12.2 from p.148 of HS2013
plt.plot(econ1.Pay_Gross, label='Tree')
plt.plot(econ1.R1_Gross, label='Risk-Free')
plt.legend()
plt.show()
```



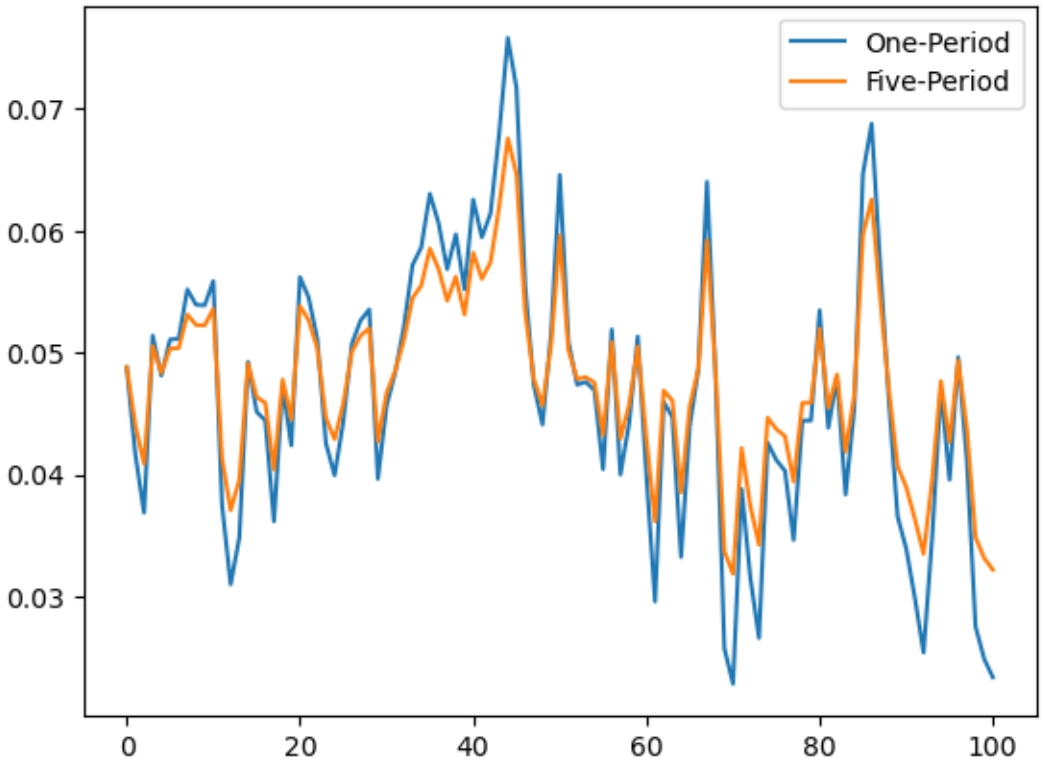

```
np.corrcoef(econ1.Pay_Gross[1:, 0], econ1.R1_Gross[1:, 0])
```

```
array([[ 1.       , -0.5079089],
       [-0.5079089,  1.       ]])
```

Above we have also calculated the correlation coefficient between these two returns.

To give an idea of how the term structure of interest rates moves in this economy, the next plot displays the *net* rates of return on one-period and five-period risk-free bonds:

```
### Right panel of Fig 7.12.2 from p.148 of HS2013
plt.plot(econ1.R1_Net, label='One-Period')
plt.plot(econ1.R5_Net, label='Five-Period')
plt.legend()
plt.show()
```



From the above plot, we can see the tendency of the term structure to slope up when rates are low and to slope down when rates are high.

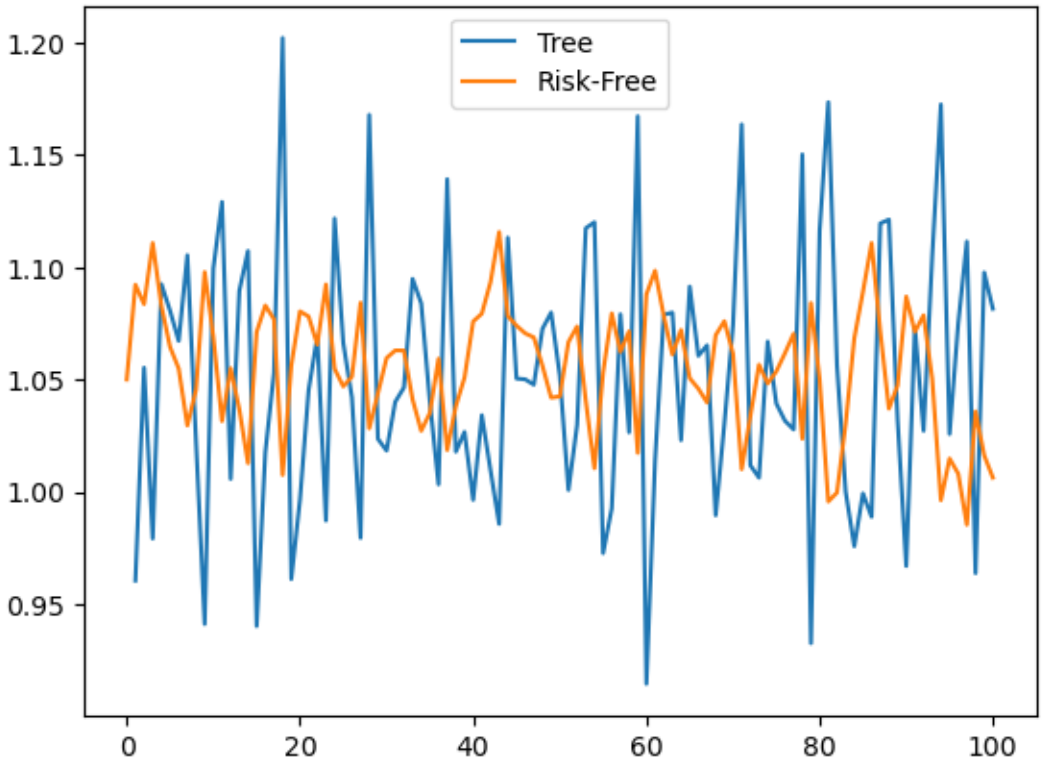
Comparing it to the previous plot of the price of the "Lucas tree", we can also see that net rates of return are low when the price of the tree is high, and vice versa.

We now plot the realized gross rate of return on a "Lucas tree" as well as on a risk-free one-period bond when the autoregressive parameter for the endowment process is reduced to 0.4:

```
a22_2 = np.array([[1,   0,   0],
                  [0, 0.4,   0],
                  [0,   0, 0.5]])
info2 = (a22_2, c2, ub, ud)

econ2 = DLE(info2, tech1, pref1)
econ2.compute_sequence(x0, ts_length=100, Pay=np.array([econ2.Sd[0, :]]))
```

```
### Left panel of Fig 7.12.3 from p.148 of HS2013
plt.plot(econ2.Pay_Gross, label='Tree')
plt.plot(econ2.R1_Gross, label='Risk-Free')
plt.legend()
plt.show()
```



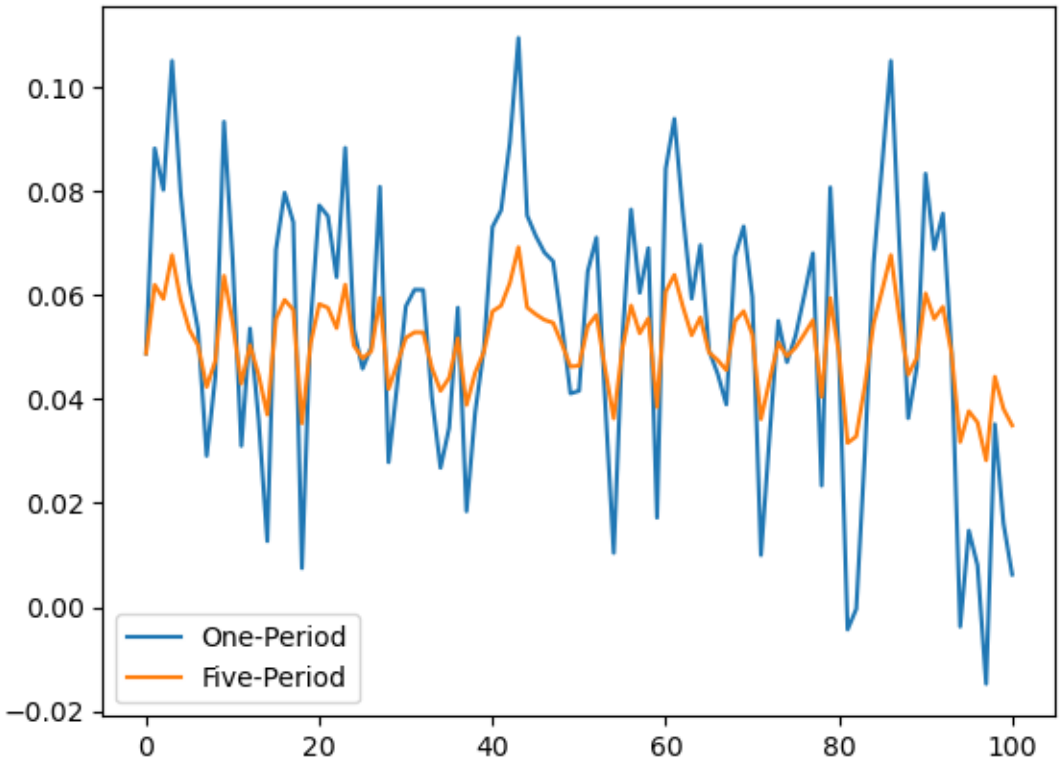

```
np.corrcoef(econ2.Pay_Gross[1:, 0], econ2.R1_Gross[1:, 0])
```

```
array([[ 1.        , -0.63959678],
       [-0.63959678,  1.        ]])
```

The correlation between these two gross rates is now more negative.

Next, we again plot the *net* rates of return on one-period and five-period risk-free bonds:

```
### Right panel of Fig 7.12.3 from p.148 of HS2013
plt.plot(econ2.R1_Net, label='One-Period')
plt.plot(econ2.R5_Net, label='Five-Period')
plt.legend()
plt.show()
```



We can see the tendency of the term structure to slope up when rates are low (and down when rates are high) has been accentuated relative to the first instance of our economy.

CHAPTER

TWENTY

IRFS IN HALL MODELS

This is another member of a suite of lectures that use the quantecon DLE class to instantiate models within the [Hansen and Sargent, 2013] class of models described in detail in *Recursive Models of Dynamic Linear Economies*.

In addition to what's in Anaconda, this lecture uses the quantecon library.

```
!pip install --upgrade quantecon
```

We'll make these imports:

```
import numpy as np
import matplotlib.pyplot as plt
from quantecon import DLE
```

This lecture shows how the DLE class can be used to create impulse response functions for three related economies, starting from Hall (1978) [Hall, 1978].

Knowledge of the basic economic environment is assumed.

See the lecture "Growth in Dynamic Linear Economies" for more details.

20.1 Example 1: Hall (1978)

First, we set parameters to make consumption (almost) follow a random walk.

We set

$$\lambda = 0, \pi = 1, \gamma_1 = 0.1, \phi_1 = 0.00001, \delta_k = 0.95, \beta = \frac{1}{1.05}$$

(In this example δ_h and θ_h are arbitrary as household capital does not enter the equation for consumption services.

We set them to values that will become useful in Example 3)

It is worth noting that this choice of parameter values ensures that $\beta(\gamma_1 + \delta_k) = 1$.

For simulations of this economy, we choose an initial condition of:

$$x_0 = \begin{bmatrix} 5 & 150 & 1 & 0 & 0 \end{bmatrix}'$$

```
γ_1 = 0.1
γ = np.array([[γ_1], [0]])
ϕ_c = np.array([[1], [0]])
ϕ_g = np.array([[0], [1]])
```

(continues on next page)

(continued from previous page)

```
ϕ_1 = 1e-5
ϕ_i = np.array([[1], [-ϕ_1]])
δ_k = np.array([[.95]])
θ_k = np.array([[1]])
β = np.array([[1 / 1.05]])
l_λ = np.array([[0]])
π_h = np.array([[1]])
δ_h = np.array([[.9]])
θ_h = np.array([[1]])
a22 = np.array([[1,   0,   0],
                [0, 0.8,   0],
                [0,   0, 0.5]])
c2 = np.array([[0, 0],
               [1, 0],
               [0, 1]])
ud = np.array([[5, 1, 0],
               [0, 0, 0]])
ub = np.array([[30, 0, 0]])
x0 = np.array([[5], [150], [1], [0], [0]])

info1 = (a22, c2, ub, ud)
tech1 = (ϕ_c, ϕ_g, ϕ_i, γ, δ_k, θ_k)
pref1 = (β, l_λ, π_h, δ_h, θ_h)
```

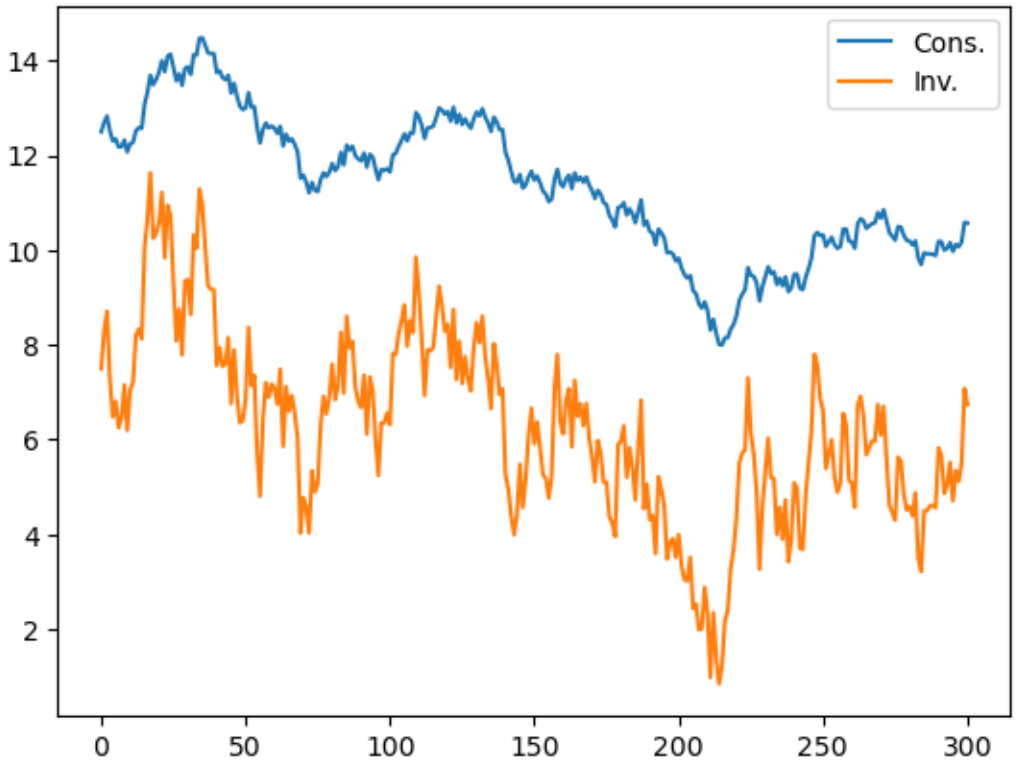
These parameter values are used to define an economy of the DLE class.

We can then simulate the economy for a chosen length of time, from our initial state vector x_0.

The economy stores the simulated values for each variable. Below we plot consumption and investment:

```
econ1 = DLE(info1, tech1, pref1)
econ1.compute_sequence(x0, ts_length=300)

# This is the right panel of Fig 5.7.1 from p.105 of HS2013
plt.plot(econ1.c[0], label='Cons.')
plt.plot(econ1.i[0], label='Inv.')
plt.legend()
plt.show()
```

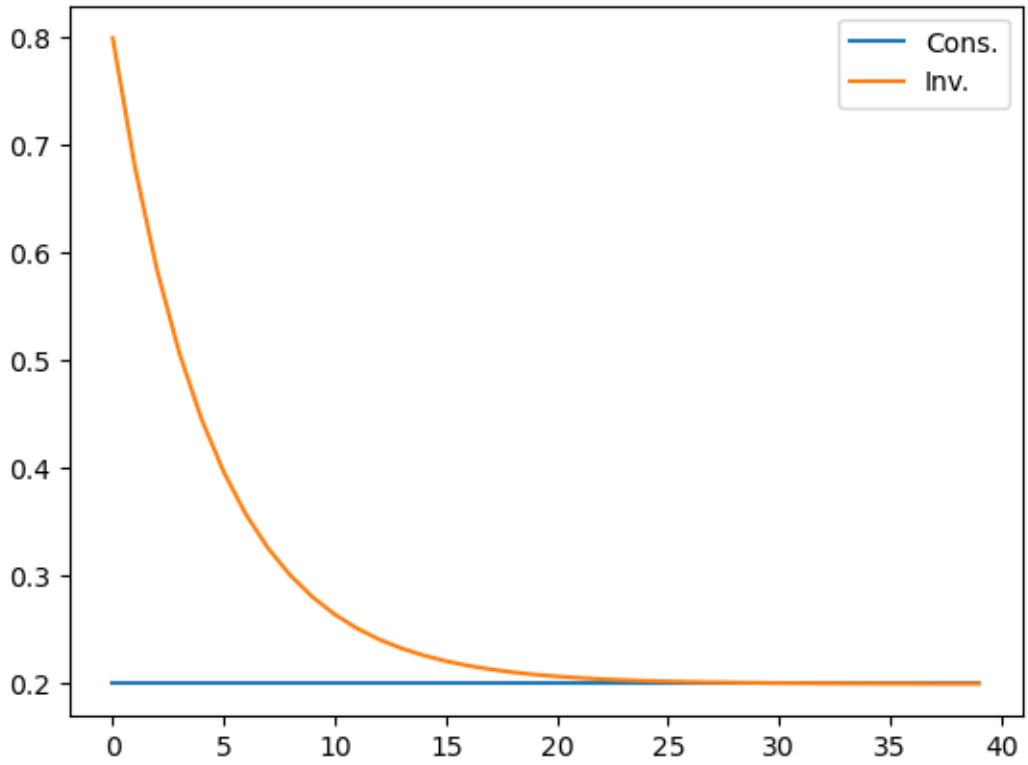


The DLE class can be used to create impulse response functions for each of the endogenous variables: $\{c_t, s_t, h_t, i_t, k_t, g_t\}$.

If no selector vector for the shock is specified, the default choice is to give IRFs to the first shock in w_{t+1}.

Below we plot the impulse response functions of investment and consumption to an endowment innovation (the first shock) in the Hall model:

```
econ1.irf(ts_length=40, shock=None)
# This is the left panel of Fig 5.7.1 from p.105 of HS2013
plt.plot(econ1.c_irf, label='Cons.')
plt.plot(econ1.i_irf, label='Inv.')
plt.legend()
plt.show()
```



It can be seen that the endowment shock has permanent effects on the level of both consumption and investment, consistent with the endogenous unit eigenvalue in this economy.

Investment is much more responsive to the endowment shock at shorter time horizons.

20.2 Example 2: Higher Adjustment Costs

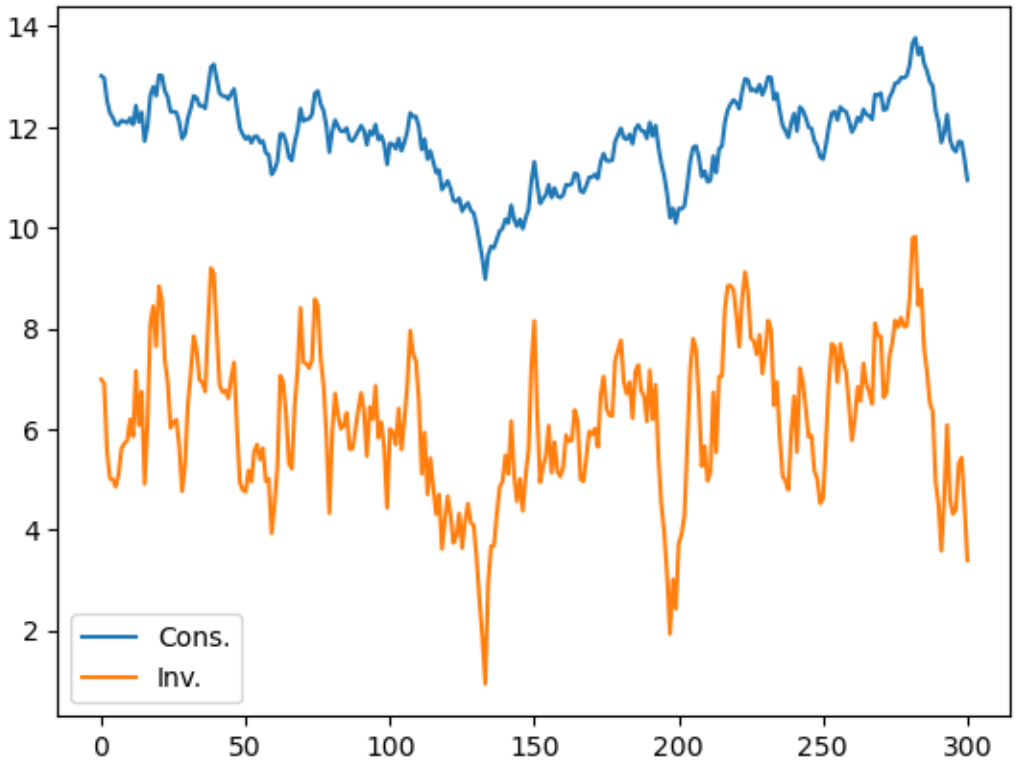
We generate our next economy by making only one change to the parameters of Example 1: we raise the parameter associated with the cost of adjusting capital,ϕ_1, from 0.00001 to 0.2.

This will lower the endogenous eigenvalue that is unity in Example 1 to a value slightly below 1.

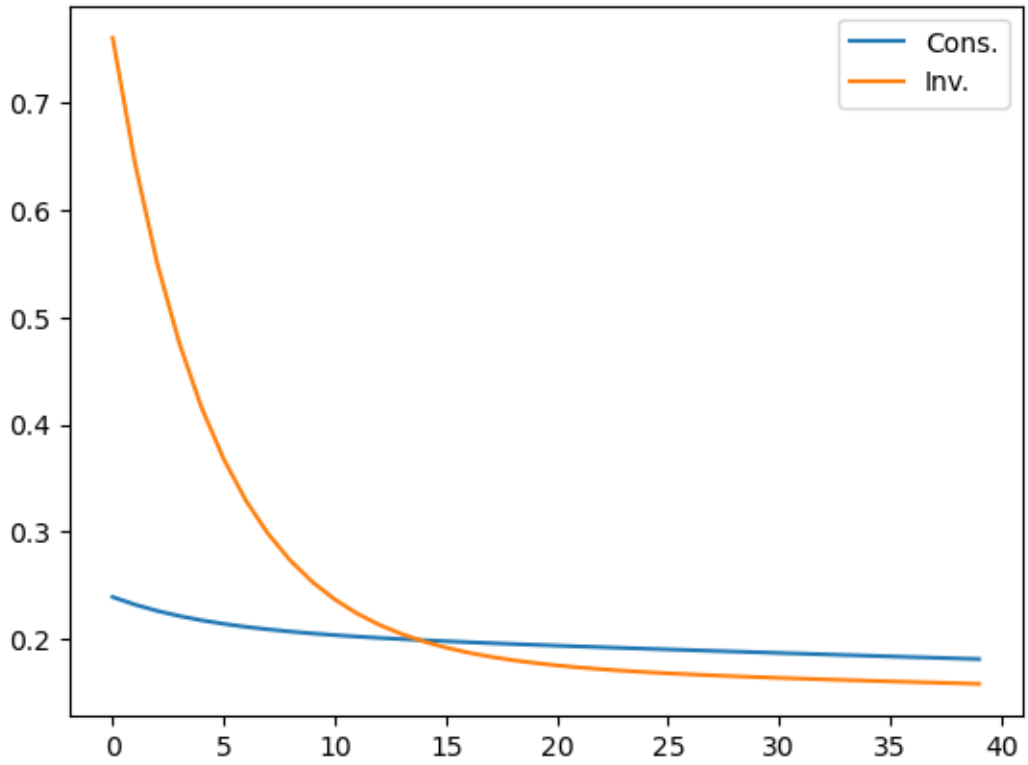
```
ϕ_12 = 0.2
ϕ_i2 = np.array([[1], [-ϕ_12]])
tech2 = (ϕ_c, ϕ_g, ϕ_i2, γ, δ_k, θ_k)

econ2 = DLE(info1, tech2, pref1)
econ2.compute_sequence(x0, ts_length = 300)

# This is the right panel of Fig 5.8.1 from p.106 of HS2013
plt.plot(econ2.c[0], label='Cons.')
plt.plot(econ2.i[0], label='Inv.')
plt.legend()
plt.show()
```




```
econ2.irf(ts_length=40,shock=None)
# This is the left panel of Fig 5.8.1 from p.106 of HS2013
plt.plot(econ2.c_irf,label='Cons.')
plt.plot(econ2.i_irf,label='Inv.')
plt.legend()
plt.show()
```




```
econ2.endo
```

```
array([0.9       , 0.99657126])
```

```
econ2.compute_steadystate()
print(econ2.css, econ2.iss, econ2.kss)
```

```
[[5.]] [[2.92940472e-12]] [[5.85879555e-11]]
```

The first graph shows that there seems to be a downward trend in both consumption and investment.

This is a consequence of the decrease in the largest endogenous eigenvalue from unity in the earlier economy, caused by the higher adjustment cost.

The present economy has a nonstochastic steady state value of 5 for consumption and 0 for both capital and investment.

Because the largest endogenous eigenvalue is still close to 1, the economy heads only slowly towards these mean values.

The impulse response functions now show that an endowment shock does not have a permanent effect on the levels of either consumption or investment.

20.3 Example 3: Durable Consumption Goods

We generate our third economy by raising ϕ_1 further, to 1.0. We also raise the production function parameter from 0.1 to 0.15 (which raises the non-stochastic steady state value of capital above zero).

We also change the specification of preferences to make the consumption good *durable*.

Specifically, we allow for a single durable household good obeying:

$$h_t = \delta_h h_{t-1} + c_t \, , 0 < \delta_h < 1$$

Services are related to the stock of durables at the beginning of the period:

$$s_t = \lambda h_{t-1} \, , \lambda > 0$$

And preferences are ordered by:

$$-\frac{1}{2}\mathbb{E}\sum_{t=0}^{\infty}\beta^t[(\lambda h_{t-1} - b_t)^2 + l_t^2]|J_0$$

To implement this, we set $\lambda = 0.1$ and $\pi = 0$ (we have already set $\theta_h = 1$ and $\delta_h = 0.9$).

We start from an initial condition that makes consumption begin near around its non-stochastic steady state.

```
ϕ_13 = 1
ϕ_i3 = np.array([[1], [-ϕ_13]])

γ_12 = 0.15
γ_2 = np.array([[γ_12], [0]])

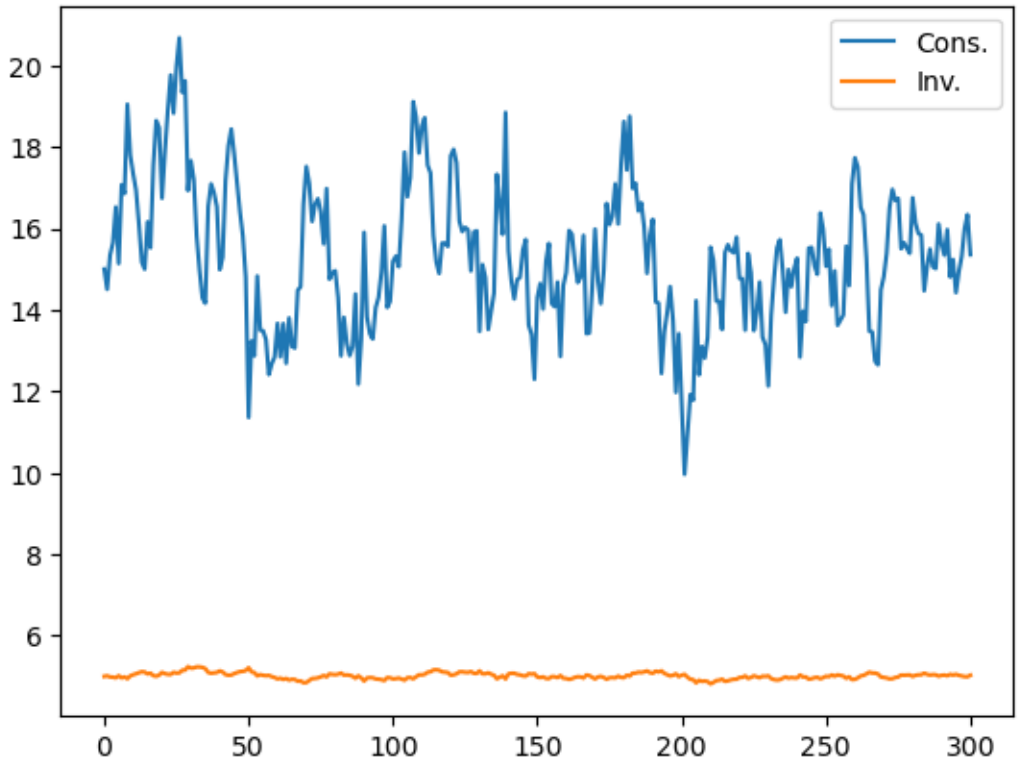
l_λ2 = np.array([[0.1]])
π_h2 = np.array([[0]])

x01 = np.array([[150], [100], [1], [0], [0]])

tech3 = (ϕ_c, ϕ_g, ϕ_i3, γ_2, δ_k, θ_k)
pref2 = (β, l_λ2, π_h2, δ_h, θ_h)

econ3 = DLE(info1, tech3, pref2)
econ3.compute_sequence(x01, ts_length=300)

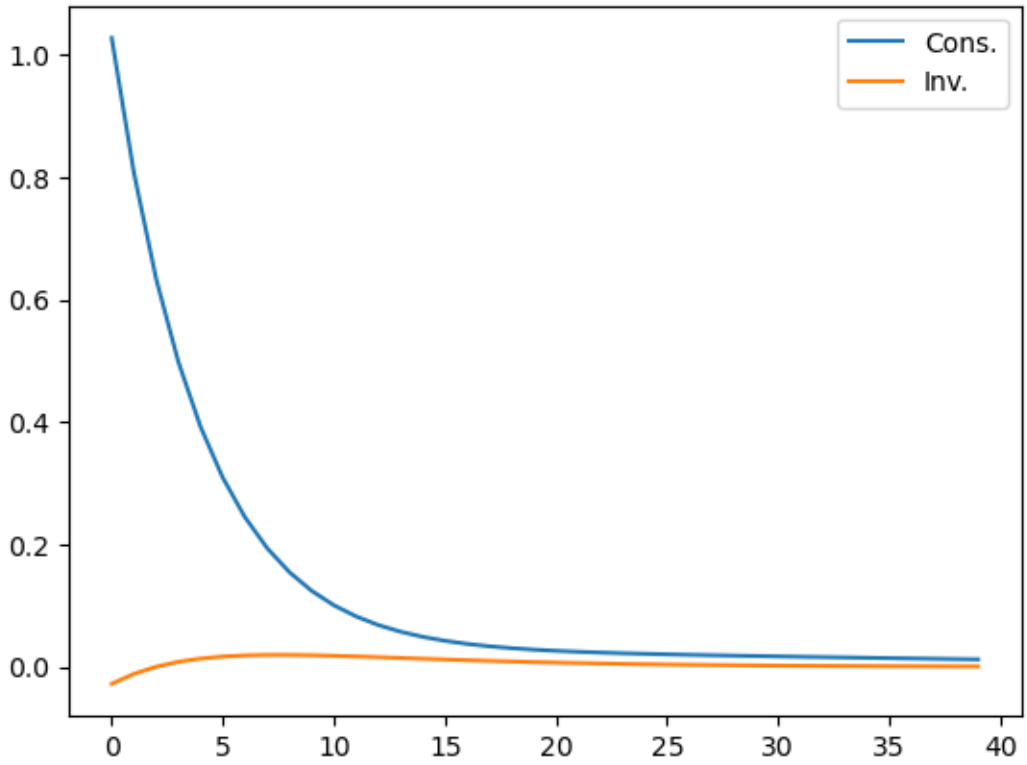
# This is the right panel of Fig 5.11.1 from p.111 of HS2013
plt.plot(econ3.c[0], label='Cons.')
plt.plot(econ3.i[0], label='Inv.')
plt.legend()
plt.show()
```



In contrast to Hall's original model of Example 1, it is now investment that is much smoother than consumption.

This illustrates how making consumption goods durable tends to undo the strong consumption smoothing result that Hall obtained.

```
econ3.irf(ts_length=40, shock=None)
# This is the left panel of Fig 5.11.1 from p.111 of HS2013
plt.plot(econ3.c_irf, label='Cons.')
plt.plot(econ3.i_irf, label='Inv.')
plt.legend()
plt.show()
```



The impulse response functions confirm that consumption is now much more responsive to an endowment shock (and investment less so) than in Example 1.

As in Example 2, the endowment shock has permanent effects on neither variable.

CHAPTER

TWENTYONE

PERMANENT INCOME MODEL USING THE DLE CLASS

This lecture is part of a suite of lectures that use the quantecon DLE class to instantiate models within the [Hansen and Sargent, 2013] class of models described in detail in *Recursive Models of Dynamic Linear Economies*.

In addition to what's included in Anaconda, this lecture uses the quantecon library.

```
!pip install --upgrade quantecon
```

This lecture adds a third solution method for the linear-quadratic-Gaussian permanent income model with $\beta R = 1$, complementing the other two solution methods described in Optimal Savings I: The Permanent Income Model and Optimal Savings II: LQ Techniques and this Jupyter notebook.

The additional solution method uses the **DLE** class.

In this way, we map the permanent income model into the framework of Hansen & Sargent (2013) "Recursive Models of Dynamic Linear Economies" [Hansen and Sargent, 2013].

We'll also require the following imports

```
import numpy as np
import matplotlib.pyplot as plt
from quantecon import DLE

np.set_printoptions(suppress=True, precision=4)
```

21.1 The Permanent Income Model

The LQ permanent income model is an example of a **savings problem**.

A consumer has preferences over consumption streams that are ordered by the utility functional

$$E_0 \sum_{t=0}^{\infty} \beta^t u(c_t) \tag{21.1}$$

where E_t is the mathematical expectation conditioned on the consumer's time t information, c_t is time t consumption, $u(c)$ is a strictly concave one-period utility function, and $\beta \in (0, 1)$ is a discount factor.

The LQ model gets its name partly from assuming that the utility function u is quadratic:

$$u(c) = -.5(c - \gamma)^2$$

where $\gamma > 0$ is a bliss level of consumption.

The consumer maximizes the utility functional (21.1) by choosing a consumption, borrowing plan $\{c_t, b_{t+1}\}_{t=0}^{\infty}$ subject to the sequence of budget constraints

$$c_t + b_t = R^{-1} b_{t+1} + y_t, t \geq 0 \tag{21.2}$$

where y_t is an exogenous stationary endowment process, R is a constant gross risk-free interest rate, b_t is one-period risk-free debt maturing at t, and b_0 is a given initial condition.

We shall assume that $R^{-1} = \beta$.

Equation (21.2) is linear.

We use another set of linear equations to model the endowment process.

In particular, we assume that the endowment process has the state-space representation

$$\begin{aligned} z_{t+1} &= A_{22} z_t + C_2 w_{t+1} \\ y_t &= U_y z_t \end{aligned} \tag{21.3}$$

where w_{t+1} is an IID process with mean zero and identity contemporaneous covariance matrix, A_{22} is a stable matrix, its eigenvalues being strictly below unity in modulus, and U_y is a selection vector that identifies y with a particular linear combination of the z_t.

We impose the following condition on the consumption, borrowing plan:

$$E_0 \sum_{t=0}^{\infty} \beta^t b_t^2 < +\infty \tag{21.4}$$

This condition suffices to rule out Ponzi schemes.

(We impose this condition to rule out a borrow-more-and-more plan that would allow the household to enjoy bliss consumption forever)

The state vector confronting the household at t is

$$x_t = \begin{bmatrix} z_t \\ b_t \end{bmatrix}$$

where b_t is its one-period debt falling due at the beginning of period t and z_t contains all variables useful for forecasting its future endowment.

We assume that $\{y_t\}$ follows a second order univariate autoregressive process:

$$y_{t+1} = \alpha + \rho_1 y_t + \rho_2 y_{t-1} + \sigma w_{t+1}$$

21.1.1 Solution with the DLE Class

One way of solving this model is to map the problem into the framework outlined in Section 4.8 of [Hansen and Sargent, 2013] by setting up our technology, information and preference matrices as follows:

Technology: $\phi_c = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \phi_g = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \phi_i = \begin{bmatrix} -1 \\ -0.00001 \end{bmatrix}, \Gamma = \begin{bmatrix} -1 \\ 0 \end{bmatrix}, \Delta_k = 0, \ \Theta_k = R.$

Information: $A_{22} = \begin{bmatrix} 1 & 0 & 0 \\ \alpha & \rho_1 & \rho_2 \\ 0 & 1 & 0 \end{bmatrix}, C_2 = \begin{bmatrix} 0 \\ \sigma \\ 0 \end{bmatrix}, U_b = \begin{bmatrix} \gamma & 0 & 0 \end{bmatrix}, U_d = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}.$

Preferences: $\Lambda = 0, \Pi = 1, \Delta_h = 0, \Theta_h = 0.$

We set parameters

$\alpha = 10, \beta = 0.95, \rho_1 = 0.9, \rho_2 = 0, \sigma = 1$

(The value of γ does not affect the optimal decision rule)

The chosen matrices mean that the household's technology is:

$$c_t + k_{t-1} = i_t + y_t$$

$$\frac{k_t}{R} = i_t$$

$$l_t^2 = (0.00001)^2 i_t$$

Combining the first two of these gives the budget constraint of the permanent income model, where $k_t = b_{t+1}$.

The third equation is a very small penalty on debt-accumulation to rule out Ponzi schemes.

We set up this instance of the DLE class below:

```
α, β, ρ_1, ρ_2, σ = 10, 0.95, 0.9, 0, 1

γ = np.array([[-1], [0]])
ϕ_c = np.array([[1], [0]])
ϕ_g = np.array([[0], [1]])
ϕ_1 = 1e-5
ϕ_i = np.array([[-1], [-ϕ_1]])
δ_k = np.array([[0]])
θ_k = np.array([[1 / β]])
β = np.array([[β]])
l_λ = np.array([[0]])
π_h = np.array([[1]])
δ_h = np.array([[0]])
θ_h = np.array([[0]])

a22 = np.array([[1,   0,   0],
                [α, ρ_1, ρ_2],
                [0, 1, 0]])

c2 = np.array([[0], [σ], [0]])
ud = np.array([[0, 1, 0],
               [0, 0, 0]])
ub = np.array([[100, 0, 0]])

x0 = np.array([[0], [0], [1], [0], [0]])

info1 = (a22, c2, ub, ud)
tech1 = (ϕ_c, ϕ_g, ϕ_i, γ, δ_k, θ_k)
pref1 = (β, l_λ, π_h, δ_h, θ_h)
econ1 = DLE(info1, tech1, pref1)
```

To check the solution of this model with that from the **LQ** problem, we select the S_c matrix from the DLE class.

The solution to the DLE economy has:

$$c_t = S_c x_t$$

```
econ1.Sc
```

```
array([[ 0.    , -0.05  , 65.5172,  0.3448,  0.    ]])
```

The state vector in the DLE class is:

$$x_t = \begin{bmatrix} h_{t-1} \\ k_{t-1} \\ z_t \end{bmatrix}$$

where $k_{t-1} = b_t$ is set up to be b_t in the permanent income model.

The state vector in the LQ problem is $\begin{bmatrix} z_t \\ b_t \end{bmatrix}$.

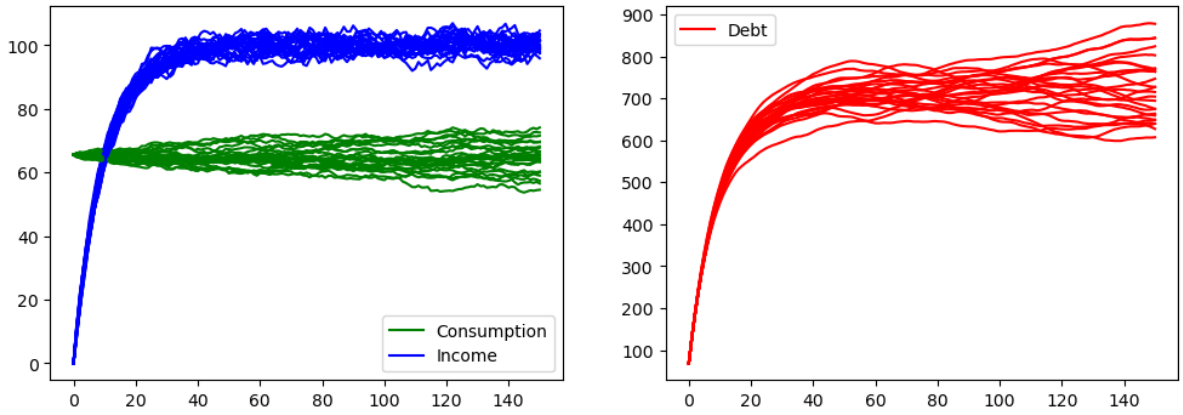
Consequently, the relevant elements of `econ1.Sc` are the same as in $-F$ occur when we apply other approaches to the same model in the lecture Optimal Savings II: LQ Techniques and this Jupyter notebook.

The plot below quickly replicates the first two figures of that lecture and that notebook to confirm that the solutions are the same

```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))

for i in range(25):
    econ1.compute_sequence(x0, ts_length=150)
    ax1.plot(econ1.c[0], c='g')
    ax1.plot(econ1.d[0], c='b')
ax1.plot(econ1.c[0], label='Consumption', c='g')
ax1.plot(econ1.d[0], label='Income', c='b')
ax1.legend()

for i in range(25):
    econ1.compute_sequence(x0, ts_length=150)
    ax2.plot(econ1.k[0], color='r')
ax2.plot(econ1.k[0], label='Debt', c='r')
ax2.legend()
plt.show()
```



CHAPTER

TWENTYTWO

ROSEN SCHOOLING MODEL

This lecture is yet another part of a suite of lectures that use the quantecon DLE class to instantiate models within the [Hansen and Sargent, 2013] class of models described in detail in *Recursive Models of Dynamic Linear Economies*.

In addition to what's included in Anaconda, this lecture uses the quantecon library

```
!pip install --upgrade quantecon
```

We'll also need the following imports:

```
import numpy as np
import matplotlib.pyplot as plt
from collections import namedtuple
from quantecon import DLE
```

22.1 A One-Occupation Model

Ryoo and Rosen's (2004) [Ryoo and Rosen, 2004] partial equilibrium model determines

- a stock of "Engineers" N_t
- a number of new entrants in engineering school, n_t
- the wage rate of engineers, w_t

It takes k periods of schooling to become an engineer.

The model consists of the following equations:

- a demand curve for engineers:

$$w_t = -\alpha_d N_t + \epsilon_{dt}$$

- a time-to-build structure of the education process:

$$N_{t+k} = \delta_N N_{t+k-1} + n_t$$

- a definition of the discounted present value of each new engineering student:

$$v_t = \beta_k \mathbb{E} \sum_{j=0}^{\infty} (\beta \delta_N)^j w_{t+k+j}$$

- a supply curve of new students driven by present value v_t:

$$n_t = \alpha_s v_t + \epsilon_{st}$$

22.2 Mapping into HS2013 Framework

We represent this model in the [Hansen and Sargent, 2013] framework by

- sweeping the time-to-build structure and the demand for engineers into the household technology, and
- putting the supply of engineers into the technology for producing goods

22.2.1 Preferences

$$\Pi = 0, \Lambda = \begin{bmatrix} \alpha_d & 0 & \cdots & 0 \end{bmatrix}, \Delta_h = \begin{bmatrix} \delta_N & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & \cdots & 0 & 1 \\ 0 & 0 & 0 & \cdots & 0 \end{bmatrix}, \Theta_h = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix}$$

where Λ is a k+1 x 1 matrix, Δ_h is a k_1 x k+1 matrix, and Θ_h is a k+1 x 1 matrix.

This specification sets $N_t = h_{1t-1}$, $n_t = c_t$, $h_{\tau+1,t-1} = n_{t-(k-\tau)}$ for $\tau = 1, ..., k$.

Below we set things up so that the number of years of education, k, can be varied.

22.2.2 Technology

To capture Ryoo and Rosen's [Ryoo and Rosen, 2004] supply curve, we use the physical technology:

$$c_t = i_t + d_{1t}$$

$$\psi_1 i_t = g_t$$

where ψ_1 is inversely proportional to α_s.

22.2.3 Information

Because we want $b_t = \epsilon_{dt}$ and $d_{1t} = \epsilon_{st}$, we set

$$A_{22} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \rho_s & 0 \\ 0 & 0 & \rho_d \end{bmatrix}, C_2 = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}, U_b = \begin{bmatrix} 30 & 0 & 1 \end{bmatrix}, U_d = \begin{bmatrix} 10 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

where ρ_s and ρ_d describe the persistence of the supply and demand shocks

```
Information = namedtuple('Information', ['a22', 'c2','ub','ud'])
Technology = namedtuple('Technology', ['ϕ_c', 'ϕ_g', 'ϕ_i', 'γ', 'δ_k', 'θ_k'])
Preferences = namedtuple('Preferences', ['β', 'l_λ', 'π_h', 'δ_h', 'θ_h'])
```

22.2.4 Effects of Changes in Education Technology and Demand

We now study how changing

- the number of years of education required to become an engineer and
- the slope of the demand curve

affects responses to demand shocks.

To begin, we set $k = 4$ and $\alpha_d = 0.1$

```
k = 4  # Number of periods of schooling required to become an engineer

β = np.array([[1 / 1.05]])
α_d = np.array([[0.1]])
α_s = 1
ε_1 = 1e-7
λ_1 = np.full((1, k), ε_1)
# Use of ε_1 is trick to aquire detectability, see HS2013 p. 228 footnote 4
l_λ = np.hstack((α_d, λ_1))
π_h = np.array([[0]])

δ_n = np.array([[0.95]])
d1 = np.vstack((δ_n, np.zeros((k - 1, 1))))
d2 = np.hstack((d1, np.eye(k)))
δ_h = np.vstack((d2, np.zeros((1, k + 1))))

θ_h = np.vstack((np.zeros((k, 1)),
                 np.ones((1, 1))))

ψ_1 = 1 / α_s

ϕ_c = np.array([[1], [0]])
ϕ_g = np.array([[0], [-1]])
ϕ_i = np.array([[-1], [ψ_1]])
γ = np.array([[0], [0]])

δ_k = np.array([[0]])
θ_k = np.array([[0]])

ρ_s = 0.8
ρ_d = 0.8

a22 = np.array([[1,   0,   0],
                [0, ρ_s,   0],
                [0,   0, ρ_d]])

c2 = np.array([[0, 0], [10, 0], [0, 10]])
ub = np.array([[30, 0, 1]])
ud = np.array([[10, 1, 0], [0, 0, 0]])

info1 = Information(a22, c2, ub, ud)
tech1 = Technology(ϕ_c, ϕ_g, ϕ_i, γ, δ_k, θ_k)
pref1 = Preferences(β, l_λ, π_h, δ_h, θ_h)

econ1 = DLE(info1, tech1, pref1)
```

We create three other instances by:

1. Raising α_d to 2
2. Raising k to 7
3. Raising k to 10

```
α_d = np.array([[2]])
l_λ = np.hstack((α_d, λ_1))
pref2 = Preferences(β, l_λ, π_h, δ_h, θ_h)
econ2 = DLE(info1, tech1, pref2)

α_d = np.array([[0.1]])

k = 7
λ_1 = np.full((1, k), ε_1)
l_λ = np.hstack((α_d, λ_1))
d1 = np.vstack((δ_n, np.zeros((k - 1, 1))))
d2 = np.hstack((d1, np.eye(k)))
δ_h = np.vstack((d2, np.zeros((1, k+1))))
θ_h = np.vstack((np.zeros((k, 1)),
                 np.ones((1, 1))))

Pref3 = Preferences(β, l_λ, π_h, δ_h, θ_h)
econ3 = DLE(info1, tech1, Pref3)

k = 10
λ_1 = np.full((1, k), ε_1)
l_λ = np.hstack((α_d, λ_1))
d1 = np.vstack((δ_n, np.zeros((k - 1, 1))))
d2 = np.hstack((d1, np.eye(k)))
δ_h = np.vstack((d2, np.zeros((1, k + 1))))
θ_h = np.vstack((np.zeros((k, 1)),
                 np.ones((1, 1))))

pref4 = Preferences(β, l_λ, π_h, δ_h, θ_h)
econ4 = DLE(info1, tech1, pref4)

shock_demand = np.array([[0], [1]])

econ1.irf(ts_length=25, shock=shock_demand)
econ2.irf(ts_length=25, shock=shock_demand)
econ3.irf(ts_length=25, shock=shock_demand)
econ4.irf(ts_length=25, shock=shock_demand)
```

The first figure plots the impulse response of n_t (on the left) and N_t (on the right) to a positive demand shock, for $\alpha_d = 0.1$ and $\alpha_d = 2$.

When $\alpha_d = 2$, the number of new students n_t rises initially, but the response then turns negative.

A positive demand shock raises wages, drawing new students into the profession.

However, these new students raise N_t.

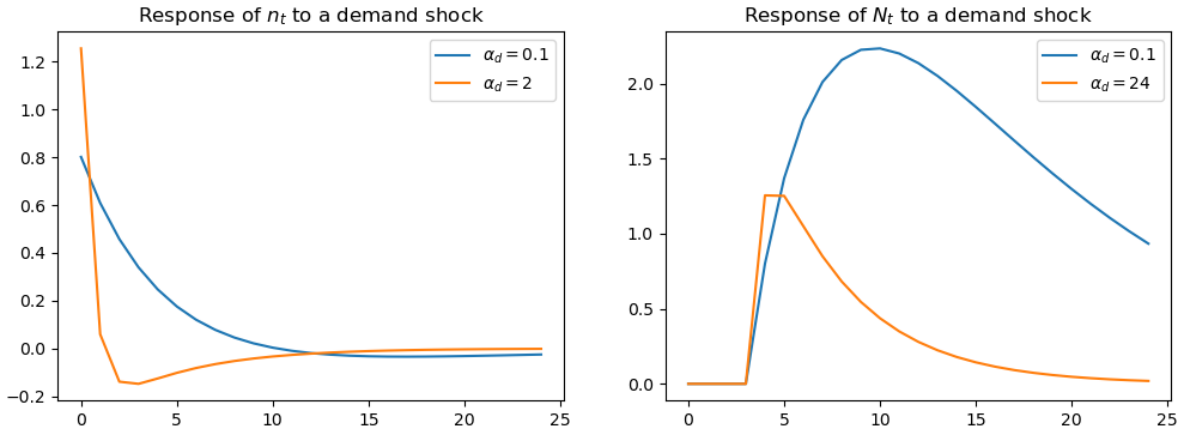
The higher is α_d, the larger the effect of this rise in N_t on wages.

This counteracts the demand shock's positive effect on wages, reducing the number of new students in subsequent periods.

Consequently, when α_d is lower, the effect of a demand shock on N_t is larger

```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))
ax1.plot(econ1.c_irf,label=r'$\alpha_d = 0.1$')
ax1.plot(econ2.c_irf,label=r'$\alpha_d = 2$')
ax1.legend()
ax1.set_title('Response of $n_t$ to a demand shock')

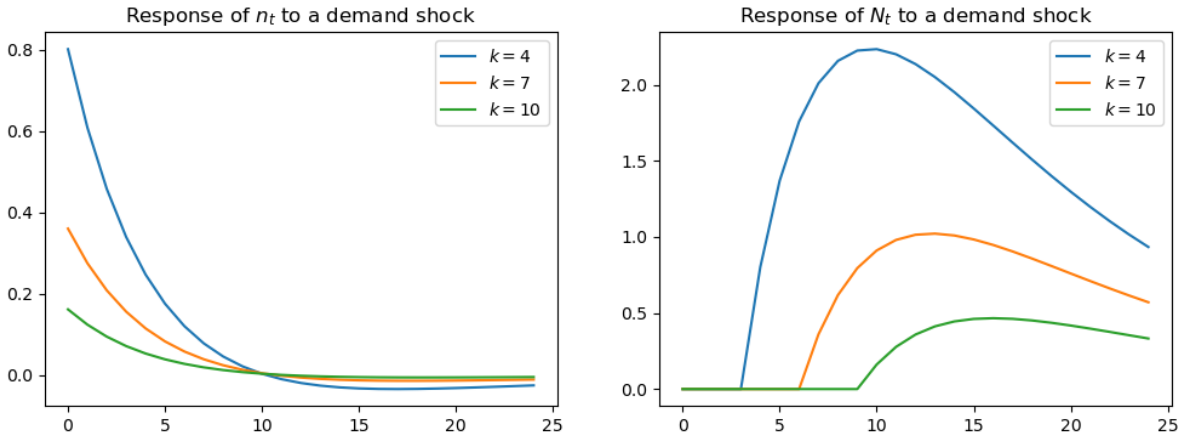
ax2.plot(econ1.h_irf[:, 0], label=r'$\alpha_d = 0.1$')
ax2.plot(econ2.h_irf[:, 0], label=r'$\alpha_d = 24$')
ax2.legend()
ax2.set_title('Response of $N_t$ to a demand shock')
plt.show()
```



The next figure plots the impulse response of n_t (on the left) and N_t (on the right) to a positive demand shock, for $k = 4$, $k = 7$ and $k = 10$ (with $\alpha_d = 0.1$)

```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))
ax1.plot(econ1.c_irf, label='$k=4$')
ax1.plot(econ3.c_irf, label='$k=7$')
ax1.plot(econ4.c_irf, label='$k=10$')
ax1.legend()
ax1.set_title('Response of $n_t$ to a demand shock')

ax2.plot(econ1.h_irf[:,0], label='$k=4$')
ax2.plot(econ3.h_irf[:,0], label='$k=7$')
ax2.plot(econ4.h_irf[:,0], label='$k=10$')
ax2.legend()
ax2.set_title('Response of $N_t$ to a demand shock')
plt.show()
```



Both panels in the above figure show that raising k lowers the effect of a positive demand shock on entry into the engineering profession.

Increasing the number of periods of schooling lowers the number of new students in response to a demand shock.

This occurs because with longer required schooling, new students ultimately benefit less from the impact of that shock on wages.

CHAPTER

TWENTYTHREE

CATTLE CYCLES

This is another member of a suite of lectures that use the quantecon DLE class to instantiate models within the [Hansen and Sargent, 2013] class of models described in detail in *Recursive Models of Dynamic Linear Economies*.

In addition to what's in Anaconda, this lecture uses the quantecon library.

```
!pip install --upgrade quantecon
```

This lecture uses the DLE class to construct instances of the "Cattle Cycles" model of Rosen, Murphy and Scheinkman (1994) [Rosen *et al.*, 1994].

That paper constructs a rational expectations equilibrium model to understand sources of recurrent cycles in US cattle stocks and prices.

We make the following imports:

```
import numpy as np
import matplotlib.pyplot as plt
from collections import namedtuple
from quantecon import DLE
from math import sqrt
```

23.1 The Model

The model features a static linear demand curve and a "time-to-grow" structure for cattle.

Let p_t be the price of slaughtered beef, m_t the cost of preparing an animal for slaughter, h_t the holding cost for a mature animal, $\gamma_1 h_t$ the holding cost for a yearling, and $\gamma_0 h_t$ the holding cost for a calf.

The cost processes $\{h_t, m_t\}_{t=0}^{\infty}$ are exogenous, while the price process $\{p_t\}_{t=0}^{\infty}$ is determined within a rational expectations equilibrium.

Let x_t be the breeding stock, and y_t be the total stock of cattle.

The law of motion for the breeding stock is

$$x_t = (1-\delta)x_{t-1} + gx_{t-3} - c_t$$

where $g < 1$ is the number of calves that each member of the breeding stock has each year, and c_t is the number of cattle slaughtered.

The total headcount of cattle is

$$y_t = x_t + gx_{t-1} + gx_{t-2}$$

This equation states that the total number of cattle equals the sum of adults, calves and yearlings, respectively.

A representative farmer chooses $\{c_t, x_t\}$ to maximize:

$$\mathbb{E}_0 \sum_{t=0}^{\infty} \beta^t \{p_t c_t - h_t x_t - \gamma_0 h_t (g x_{t-1}) - \gamma_1 h_t (g x_{t-2}) - m_t c_t - \frac{\psi_1}{2} x_t^2 - \frac{\psi_2}{2} x_{t-1}^2 - \frac{\psi_3}{2} x_{t-3}^2 - \frac{\psi_4}{2} c_t^2\}$$

subject to the law of motion for x_t, taking as given the stochastic laws of motion for the exogenous processes, the equilibrium price process, and the initial state $[x_{-1}, x_{-2}, x_{-3}]$.

Remark The ψ_j parameters are very small quadratic costs that are included for technical reasons to make well posed and well behaved the linear quadratic dynamic programming problem solved by the fictitious planner who in effect chooses equilibrium quantities and shadow prices.

Demand for beef is government by $c_t = a_0 - a_1 p_t + \tilde{d}_t$ where $\tilde{d}_t$ is a stochastic process with mean zero, representing a demand shifter.

23.2 Mapping into HS2013 Framework

23.2.1 Preferences

We set $\Lambda = 0, \Delta_h = 0, \Theta_h = 0, \Pi = \alpha_1^{-\frac{1}{2}}$ and $b_t = \Pi \tilde{d}_t + \Pi \alpha_0$.

With these settings, the FOC for the household's problem becomes the demand curve of the "Cattle Cycles" model.

23.2.2 Technology

To capture the law of motion for cattle, we set

$$\Delta_k = \begin{bmatrix} (1-\delta) & 0 & g \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}, \ \Theta_k = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

(where $i_t = -c_t$).

To capture the production of cattle, we set

$$\Phi_c = \begin{bmatrix} 1 \\ f_1 \\ 0 \\ 0 \\ -f_7 \end{bmatrix}, \ \Phi_g = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \ \Phi_i = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \ \Gamma = \begin{bmatrix} 0 & 0 & 0 \\ f_1(1-\delta) & 0 & g f_1 \\ f_3 & 0 & 0 \\ 0 & f_5 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

23.2.3 Information

We set

$$A_{22} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \rho_1 & 0 & 0 \\ 0 & 0 & \rho_2 & 0 \\ 0 & 0 & 0 & \rho_3 \end{bmatrix}, C_2 = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 15 \end{bmatrix}, U_b = \begin{bmatrix} \Pi \alpha_0 & 0 & 0 & \Pi \end{bmatrix}, U_d = \begin{bmatrix} 0 \\ f_2 U_h \\ f_4 U_h \\ f_6 U_h \\ f_8 U_h \end{bmatrix}$$

To map this into our class, we set $f_1^2 = \frac{\Psi_1}{2}$, $f_2^2 = \frac{\Psi_2}{2}$, $f_3^2 = \frac{\Psi_3}{2}$, $2 f_1 f_2 = 1$, $2 f_3 f_4 = \gamma_0 g$, $2 f_5 f_6 = \gamma_1 g$.

```
# We define namedtuples in this way as it allows us to check, for example,
# what matrices are associated with a particular technology.

Information = namedtuple('Information', ['a22', 'c2', 'ub', 'ud'])
Technology = namedtuple('Technology', ['ϕ_c', 'ϕ_g', 'ϕ_i', 'γ', 'δ_k', 'θ_k'])
Preferences = namedtuple('Preferences', ['β', 'l_λ', 'π_h', 'δ_h', 'θ_h'])
```

We set parameters to those used by [Rosen *et al.*, 1994]

```
β = np.array([[0.909]])
lλ = np.array([[0]])

a1 = 0.5
πh = np.array([[1 / (sqrt(a1))]])
δh = np.array([[0]])
θh = np.array([[0]])

δ = 0.1
g = 0.85
f1 = 0.001
f3 = 0.001
f5 = 0.001
f7 = 0.001

ϕc = np.array([[1], [f1], [0], [0], [-f7]])

ϕg = np.array([[0, 0, 0, 0],
               [1, 0, 0, 0],
               [0, 1, 0, 0],
               [0, 0, 1,0],
               [0, 0, 0, 1]])

ϕi = np.array([[1], [0], [0], [0], [0]])

γ = np.array([[           0,  0,       0],
              [f1 * (1 - δ),  0, g * f1],
              [           f3,  0,       0],
              [            0, f5,       0],
              [            0,  0,       0]])

δk = np.array([[1 - δ, 0, g],
               [    1, 0, 0],
               [    0, 1, 0]])

θk = np.array([[1], [0], [0]])

ρ1 = 0
ρ2 = 0
ρ3 = 0.6
a0 = 500
γ0 = 0.4
γ1 = 0.7
f2 = 1 / (2 * f1)
f4 = γ0 * g / (2 * f3)
f6 = γ1 * g / (2 * f5)
f8 = 1 / (2 * f7)
```

(continues on next page)

(continued from previous page)

```
a22 = np.array([[1, 0, 0, 0],
                [0, ρ1, 0, 0],
                [0, 0, ρ2, 0],
                [0, 0, 0, ρ3]])

c2 = np.array([[0, 0,  0],
               [1, 0,  0],
               [0, 1,  0],
               [0, 0, 15]])

πh_scalar = πh.item()
ub = np.array([[πh_scalar * a0, 0, 0, πh_scalar]])
uh = np.array([[50, 1, 0, 0]])
um = np.array([[100, 0, 1, 0]])
ud = np.vstack(([0, 0, 0, 0],
                f2 * uh, f4 * uh, f6 * uh, f8 * um))
```

Notice that we have set $\rho_1 = \rho_2 = 0$, so h_t and m_t consist of a constant and a white noise component.

We set up the economy using tuples for information, technology and preference matrices below.

We also construct two extra information matrices, corresponding to cases when $\rho_3 = 1$ and $\rho_3 = 0$ (as opposed to the baseline case of $\rho_3 = 0.6$).

```
info1 = Information(a22, c2, ub, ud)
tech1 = Technology(ϕc, ϕg, ϕi, γ, δk, θk)
pref1 = Preferences(β, lλ, πh, δh, θh)

ρ3_2 = 1
a22_2 = np.array([[1,  0,  0,    0],
                  [0, ρ1,  0,    0],
                  [0,  0, ρ2,    0],
                  [0,  0,  0, ρ3_2]])

info2 = Information(a22_2, c2, ub, ud)

ρ3_3 = 0
a22_3 = np.array([[1,  0,  0,    0],
                  [0, ρ1,  0,    0],
                  [0,  0, ρ2,    0],
                  [0,  0,  0, ρ3_3]])

info3 = Information(a22_3, c2, ub, ud)

# Example of how we can look at the matrices associated with a given namedtuple
info1.a22
```

```
array([[1. , 0. , 0. , 0. ],
       [0. , 0. , 0. , 0. ],
       [0. , 0. , 0. , 0. ],
       [0. , 0. , 0. , 0.6]])
```

```
# Use tuples to define DLE class
econ1 = DLE(info1, tech1, pref1)
econ2 = DLE(info2, tech1, pref1)
```

(continues on next page)

(continued from previous page)

```
econ3 = DLE(info3, tech1, pref1)

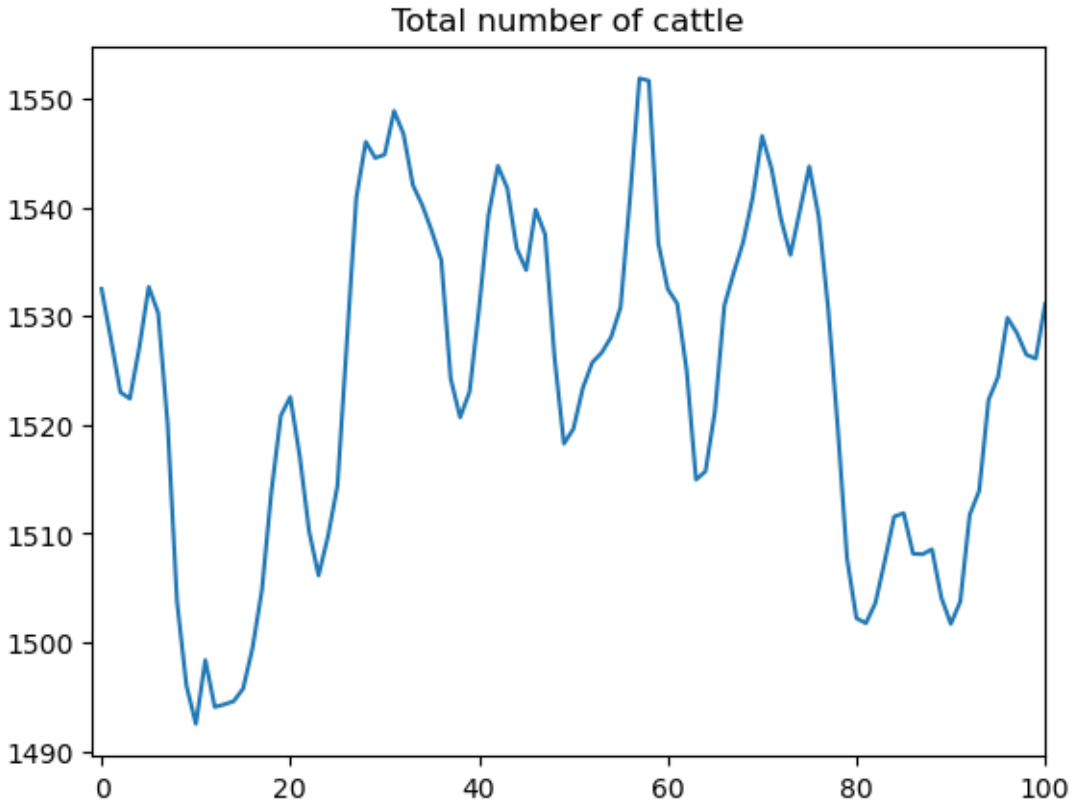
# Calculate steady-state in baseline case and use to set the initial condition
econ1.compute_steadystate(nnc=4)
x0 = econ1.zz
```

```
econ1.compute_sequence(x0, ts_length=100)
```

[Rosen *et al.*, 1994] use the model to understand the sources of recurrent cycles in total cattle stocks.

Plotting y_t for a simulation of their model shows its ability to generate cycles in quantities

```
# Calculation of y_t
totalstock = econ1.k[0] + g * econ1.k[1] + g * econ1.k[2]
fig, ax = plt.subplots()
ax.plot(totalstock)
ax.set_xlim((-1, 100))
ax.set_title('Total number of cattle')
plt.show()
```



In their Figure 3, [Rosen *et al.*, 1994] plot the impulse response functions of consumption and the breeding stock of cattle to the demand shock, $\tilde{d}_t$, under the three different values of ρ_3.

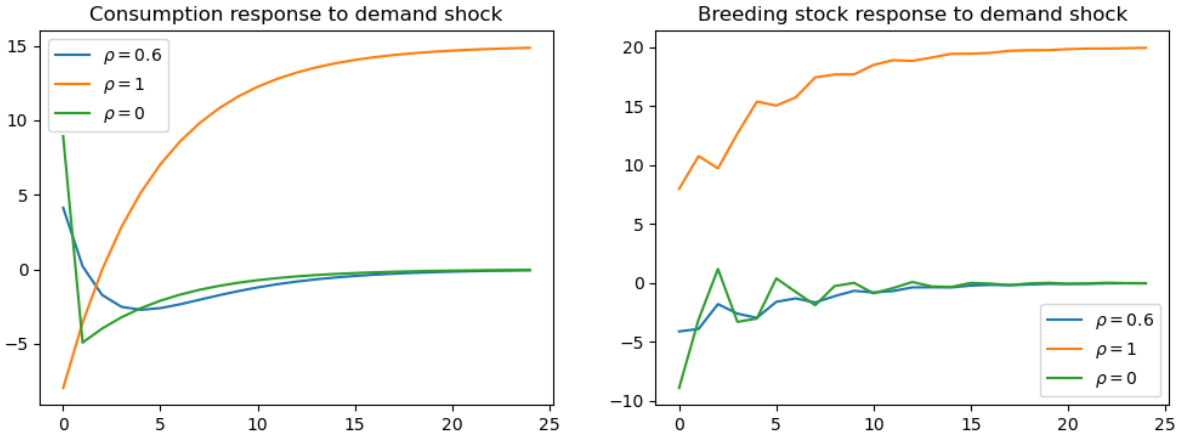
We replicate their Figure 3 below

```
shock_demand = np.array([[0], [0], [1]])

econ1.irf(ts_length=25, shock=shock_demand)
econ2.irf(ts_length=25, shock=shock_demand)
econ3.irf(ts_length=25, shock=shock_demand)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))
ax1.plot(econ1.c_irf, label=r'$\rho=0.6$')
ax1.plot(econ2.c_irf, label=r'$\rho=1$')
ax1.plot(econ3.c_irf, label=r'$\rho=0$')
ax1.set_title('Consumption response to demand shock')
ax1.legend()

ax2.plot(econ1.k_irf[:, 0], label=r'$\rho=0.6$')
ax2.plot(econ2.k_irf[:, 0], label=r'$\rho=1$')
ax2.plot(econ3.k_irf[:, 0], label=r'$\rho=0$')
ax2.set_title('Breeding stock response to demand shock')
ax2.legend()
plt.show()
```



The above figures show how consumption patterns differ markedly, depending on the persistence of the demand shock:

- If it is purely transitory ($\rho_3 = 0$) then consumption rises immediately but is later reduced to build stocks up again.
- If it is permanent ($\rho_3 = 1$), then consumption falls immediately, in order to build up stocks to satisfy the permanent rise in future demand.

In Figure 4 of their paper, [Rosen *et al.*, 1994] plot the response to a demand shock of the breeding stock *and* the total stock, for $\rho_3 = 0$ and $\rho_3 = 0.6$.

We replicate their Figure 4 below

```
total1_irf = econ1.k_irf[:, 0] + g * econ1.k_irf[:, 1] + g * econ1.k_irf[:, 2]
total3_irf = econ3.k_irf[:, 0] + g * econ3.k_irf[:, 1] + g * econ3.k_irf[:, 2]

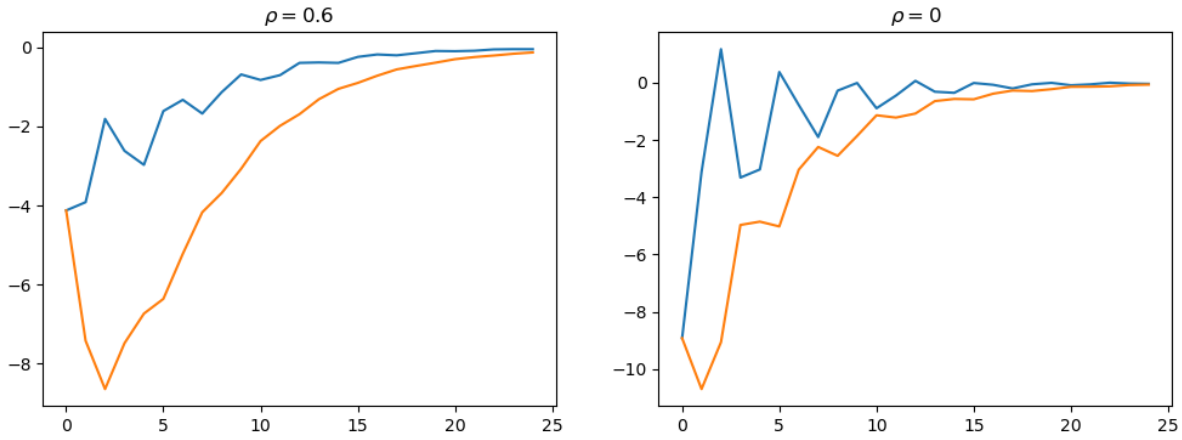
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))
ax1.plot(econ1.k_irf[:, 0], label='Breeding Stock')
ax1.plot(total1_irf, label='Total Stock')
ax1.set_title(r'$\rho=0.6$')

ax2.plot(econ3.k_irf[:, 0], label='Breeding Stock')
ax2.plot(total3_irf, label='Total Stock')
```

(continues on next page)

(continued from previous page)

```
ax2.set_title(r'$\rho=0$')
plt.show()
```



The fact that y_t is a weighted moving average of x_t creates a humped shape response of the total stock in response to demand shocks, contributing to the cyclicality seen in the first graph of this lecture.

CHAPTER

TWENTYFOUR

SHOCK NON INVERTIBILITY

24.1 Overview

This is another member of a suite of lectures that use the quantecon DLE class to instantiate models within the [Hansen and Sargent, 2013] class of models described in *Recursive Models of Dynamic Linear Economies*.

In addition to what's in Anaconda, this lecture uses the quantecon library.

```
!pip install --upgrade quantecon
```

We'll make these imports:

```
import numpy as np
import quantecon as qe
import matplotlib.pyplot as plt
from quantecon import DLE
from math import sqrt
```

This lecture describes an early contribution to what is now often called a **news and noise** issue.

In particular, it analyzes a **shock-invertibility** issue that is endemic within a class of permanent income models.

Technically, the invertibility problem indicates a situation in which histories of the shocks in an econometrician's autoregressive or Wold moving average representation span a smaller information space than do the shocks that are seen by the agents inside the econometrician's model.

An econometrician who is unaware of the problem would misinterpret shocks and likely responses to them.

A shock-invertibility that is technically close to the one studied here is discussed by Eric Leeper, Todd Walker, and Susan Yang [Leeper *et al.*, 2013] in their analysis of **fiscal foresight**.

A distinct shock-invertibility issue is present in the special LQ consumption smoothing model in this quantecon lecture *Information and Consumption Smoothing*.

24.2 Model

We consider the following modification of Robert Hall's (1978) model [Hall, 1978] in which the endowment process is the sum of two orthogonal autoregressive processes:

Preferences

$$-\frac{1}{2}\mathbb{E}\sum_{t=0}^{\infty}\beta^t[(c_t-b_t)^2+l_t^2]|J_0$$

$$s_t = c_t$$

$$b_t = U_b z_t$$

Technology

$$c_t + i_t = \gamma k_{t-1} + d_t$$

$$k_t = \delta_k k_{t-1} + i_t$$

$$g_t = \phi_1 i_t\,, \phi_1 > 0$$

$$g_t \cdot g_t = l_t^2$$

Information

$$z_{t+1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.9 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} z_t + \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 4 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} w_{t+1}$$

$$U_b = \begin{bmatrix} 30 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$U_d = \begin{bmatrix} 5 & 1 & 1 & 0.8 & 0.6 & 0.4 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

The preference shock is constant at 30, while the endowment process is the sum of a constant and two orthogonal processes.

Specifically:

$$d_t = 5 + d_{1t} + d_{2t}$$

$$d_{1t} = 0.9 d_{1t-1} + w_{1t}$$

$$d_{2t} = 4w_{2t} + 0.8(4w_{2t-1}) + 0.6(4w_{2t-2}) + 0.4(4w_{2t-3})$$

d_{1t} is a first-order AR process, while d_{2t} is a third-order pure moving average process.

```
y_1 = 0.05
y = np.array([[y_1], [0]])
ϕ_c = np.array([[1], [0]])
ϕ_g = np.array([[0], [1]])
ϕ_1 = 0.00001
ϕ_i = np.array([[1], [-ϕ_1]])
δ_k = np.array([[1]])
θ_k = np.array([[1]])
```

(continues on next page)

(continued from previous page)

```
β = np.array([[1 / 1.05]])
l_λ = np.array([[0]])
π_h = np.array([[1]])
δ_h = np.array([[.9]])
θ_h = np.array([[1]]) - δ_h
ud = np.array([[5, 1, 1, 0.8, 0.6, 0.4],
               [0, 0, 0,   0,   0,   0]])
a22 = np.zeros((6, 6))
# Chase's great trick
a22[[0, 1, 3, 4, 5], [0, 1, 2, 3, 4]] = np.array([1.0, 0.9, 1.0, 1.0, 1.0])
c2 = np.zeros((6, 2))
c2[[1, 2], [0, 1]] = np.array([1.0, 4.0])
ub = np.array([[30, 0, 0, 0, 0, 0]])
x0 = np.array([[5], [150], [1], [0], [0], [0], [0], [0]])

info1 = (a22, c2, ub, ud)
tech1 = (ϕ_c, ϕ_g, ϕ_i, γ, δ_k, θ_k)
pref1 = (β, l_λ, π_h, δ_h, θ_h)

econ1 = DLE(info1, tech1, pref1)
```

We define the household's net of interest deficit as $c_t - d_t$.

Hall's model imposes "expected present-value budget balance" in the sense that

$$\mathbb{E} \sum_{j=0}^{\infty} \beta^j (c_{t+j} - d_{t+j}) | J_t = \beta^{-1} k_{t-1} \, \forall t$$

Define a moving average representation of $(c_t, c_t - d_t)$ in terms of the w_ts to be:

$$\begin{bmatrix} c_t \\ c_t - d_t \end{bmatrix} = \begin{bmatrix} \sigma_1(L) \\ \sigma_2(L) \end{bmatrix} w_t$$

Hall's model imposes the restriction $\sigma_2(\beta) = [0 \;\; 0]$.

- The consumer who lives inside this model observes histories of both components of the endowment process d_{1t} and d_{2t}.
- The econometrician has data on the history of the pair $[c_t, d_t]$, but not directly on the history of w_t's.
- The econometrician obtains a Wold representation for the process $[c_t, c_t - d_t]$:

$$\begin{bmatrix} c_t \\ c_t - d_t \end{bmatrix} = \begin{bmatrix} \sigma_1^*(L) \\ \sigma_2^*(L) \end{bmatrix} u_t$$

A representation with equivalent shocks would be recovered by estimating a bivariate vector autoregression for $c_t, c_t - d_t$.

The Appendix of chapter 8 of [Hansen and Sargent, 2013] explains why the impulse response functions in the Wold representation estimated by the econometrician do not resemble the impulse response functions that depict the response of consumption and the net-of-interest deficit to innovations w_t to the consumer's information.

Technically, $\sigma_2(\beta) = [0 \;\; 0]$ implies that the history of u_ts spans a *smaller* linear space than does the history of w_ts.

This means that u_t will typically be a distributed lag of w_t that is not concentrated at zero lag:

$$u_t = \sum_{j=0}^{\infty} \alpha_j w_{t-j}$$

Thus, the econometrician's news u_t typically responds belatedly to the consumer's news w_t.

24.3 Code

We will construct Figures from Chapter 8 Appendix E of [Hansen and Sargent, 2013] to illustrate these ideas:

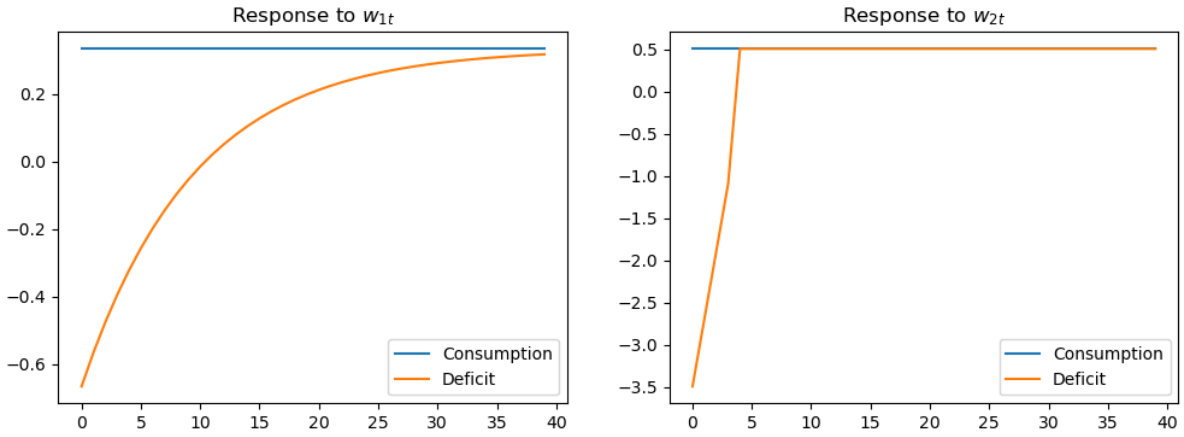
```
# This is Fig 8.E.1 from p.188 of HS2013

econ1.irf(ts_length=40, shock=None)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))
ax1.plot(econ1.c_irf, label='Consumption')
ax1.plot(econ1.c_irf - econ1.d_irf[:,0].reshape(40,1), label='Deficit')
ax1.legend()
ax1.set_title('Response to $w_{1t}$')

shock2 = np.array([[0], [1]])
econ1.irf(ts_length=40, shock=shock2)

ax2.plot(econ1.c_irf, label='Consumption')
ax2.plot(econ1.c_irf - econ1.d_irf[:,0].reshape(40, 1), label='Deficit')
ax2.legend()
ax2.set_title('Response to $w_{2t}$')
plt.show()
```



The above figure displays the impulse response of consumption and the net-of-interest deficit to the innovations w_t to the consumer's non-financial income or endowment process.

Consumption displays the characteristic "random walk" response with respect to each innovation.

Each endowment innovation leads to a temporary surplus followed by a permanent net-of-interest deficit.

The temporary surplus just offsets the permanent deficit in terms of expected present value.

```
G_HS = np.vstack([econ1.Sc, econ1.Sc-econ1.Sd[0, :].reshape(1, 8)])
H_HS = 1e-8 * np.eye(2)  # Set very small so there is no measurement error
lss_hs = qe.LinearStateSpace(econ1.A0, econ1.C, G_HS, H_HS)

hs_kal = qe.Kalman(lss_hs)
w_lss = hs_kal.whitener_lss()
ma_coefs = hs_kal.stationary_coefficients(50, 'ma')

# This is Fig 8.E.2 from p.189 of HS2013
```

(continues on next page)

(continued from previous page)

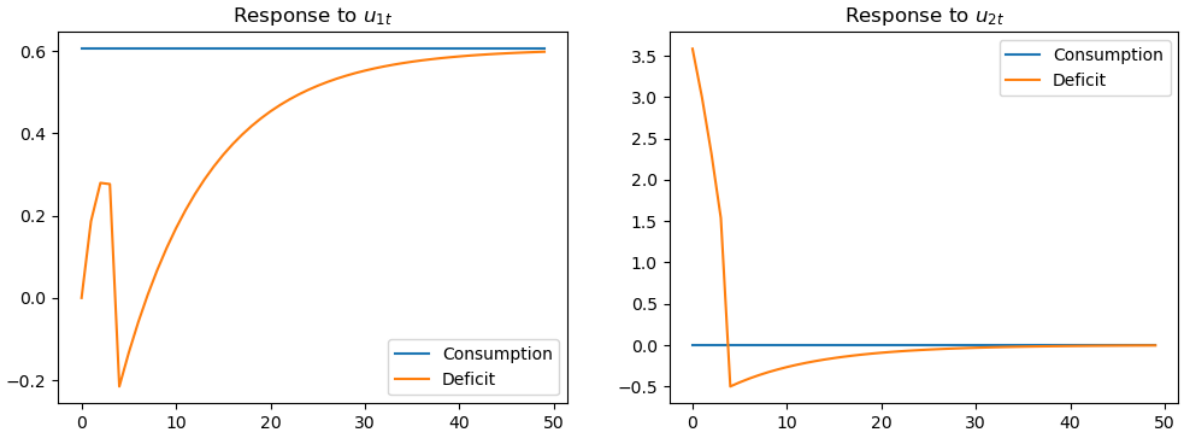
```
ma_coefs = ma_coefs
jj = 50
y1_w1 = np.empty(jj)
y2_w1 = np.empty(jj)
y1_w2 = np.empty(jj)
y2_w2 = np.empty(jj)

for t in range(jj):
    y1_w1[t] = ma_coefs[t][0, 0]
    y1_w2[t] = ma_coefs[t][0, 1]
    y2_w1[t] = ma_coefs[t][1, 0]
    y2_w2[t] = ma_coefs[t][1, 1]

# This scales the impulse responses to match those in the book
y1_w1 = sqrt(hs_kal.stationary_innovation_covar()[0, 0]) * y1_w1
y2_w1 = sqrt(hs_kal.stationary_innovation_covar()[0, 0]) * y2_w1
y1_w2 = sqrt(hs_kal.stationary_innovation_covar()[1, 1]) * y1_w2
y2_w2 = sqrt(hs_kal.stationary_innovation_covar()[1, 1]) * y2_w2

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))
ax1.plot(y1_w1, label='Consumption')
ax1.plot(y2_w1, label='Deficit')
ax1.legend()
ax1.set_title('Response to $u_{1t}$')

ax2.plot(y1_w2, label='Consumption')
ax2.plot(y2_w2, label='Deficit')
ax2.legend()
ax2.set_title('Response to $u_{2t}$')
plt.show()
```



The above figure displays the impulse response of consumption and the deficit to the innovations in the econometrician's Wold representation

- this is the object that would be recovered from a high order vector autoregression on the econometrician's observations.

Consumption responds only to the first innovation

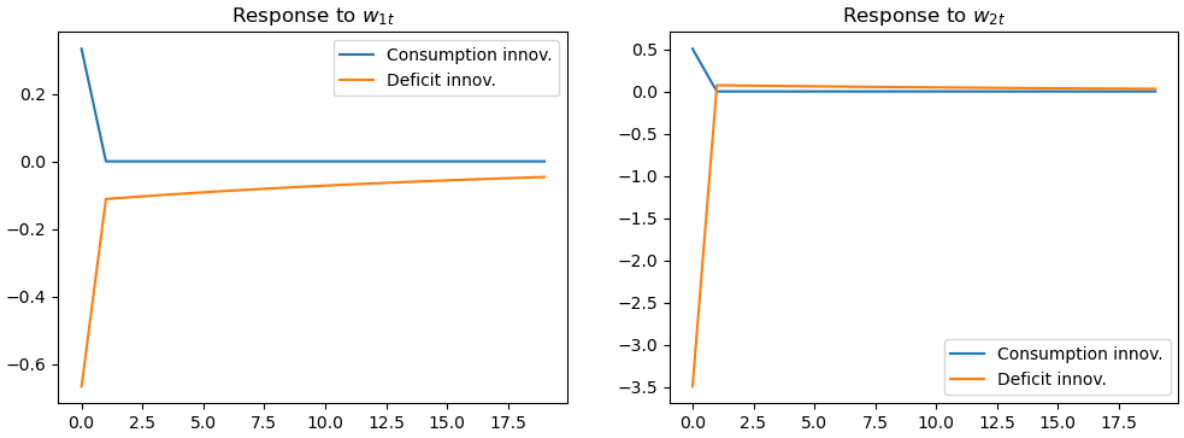
- this is indicative of the Granger causality imposed on the $[c_t, c_t - d_t]$ process by Hall's model: consumption Granger causes $c_t - d_t$, with no reverse causality.

```
# This is Fig 8.E.3 from p.189 of HS2013

jj = 20
irf_wlss = w_lss.impulse_response(jj)
ycoefs = irf_wlss[1]
# Pull out the shocks
a1_w1 = np.empty(jj)
a1_w2 = np.empty(jj)
a2_w1 = np.empty(jj)
a2_w2 = np.empty(jj)

for t in range(jj):
    a1_w1[t] = ycoefs[t][0, 0]
    a1_w2[t] = ycoefs[t][0, 1]
    a2_w1[t] = ycoefs[t][1, 0]
    a2_w2[t] = ycoefs[t][1, 1]

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))
ax1.plot(a1_w1, label='Consumption innov.')
ax1.plot(a2_w1, label='Deficit innov.')
ax1.set_title('Response to $w_{1t}$')
ax1.legend()
ax2.plot(a1_w2, label='Consumption innov.')
ax2.plot(a2_w2, label='Deficit innov.')
ax2.legend()
ax2.set_title('Response to $w_{2t}$')
plt.show()
```



The above figure displays the impulse responses of u_t to w_t, as depicted in:

$$u_t = \sum_{j=0}^{\infty} \alpha_j w_{t-j}$$

While the responses of the innovations to consumption are concentrated at lag zero for both components of w_t, the responses of the innovations to $(c_t - d_t)$ are spread over time (especially in response to w_{1t}).

Thus, the innovations to $(c_t - d_t)$ as revealed by the vector autoregression depend on what the economic agent views as "old news".

Part V

Risk, Model Uncertainty, and Robustness

CHAPTER

TWENTYFIVE

RISK AND MODEL UNCERTAINTY

25.1 Overview

As an introduction to one possible approach to modeling **Knightian uncertainty**, this lecture describes static representations of five classes of preferences over risky prospects.

These preference specifications allow us to distinguish **risk** from **uncertainty** along lines proposed by [Knight, 1921].

All five preference specifications incorporate **risk aversion**, meaning displeasure from risks governed by well known probability distributions.

Two of them also incorporate **uncertainty aversion**, meaning dislike of not knowing a probability distribution.

The preference orderings are

- Expected utility preferences
- Constraint preferences
- Multiplier preferences
- Risk-sensitive preferences
- Ex post Bayesian expected utility preferences

This labeling scheme is taken from [Hansen and Sargent, 2001].

Constraint and multiplier preferences express aversion to not knowing a unique probability distribution that describes random outcomes.

Expected utility, risk-sensitive, and ex post Bayesian expected utility preferences all attribute a unique known probability distribution to a decision maker.

We present things in a simple before-and-after one-period setting.

In addition to learning about these preference orderings, this lecture also describes some interesting code for computing and graphing some representations of indifference curves, utility functions, and related objects.

Staring at these indifference curves provides insights into the different preferences.

Watch for the presence of a kink at the 45 degree line for the constraint preference indifference curves.

We begin with some that we'll use to create some graphs.

```
# Package imports
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
from matplotlib import rc, cm
```

(continues on next page)

(continued from previous page)

```
from mpl_toolkits.mplot3d import Axes3D
from scipy import optimize, stats
from scipy.io import loadmat
from matplotlib.collections import LineCollection
from matplotlib.colors import ListedColormap, BoundaryNorm
from numba import njit
```

25.2 Basic objects

Basic ingredients are

- a set of states of the world
- plans describing outcomes as functions of the state of the world,
- a utility function mapping outcomes into utilities
- either a probability distribution or a **set** of probability distributions over states of the world; and
- a way of measuring a discrepancy between two probability distributions.

In more detail, we'll work with the following setting.

- A finite set of possible **states** $I = \{i = 1, \ldots, I\}$.
- A (consumption) **plan** is a function $c : I \to \mathbb{R}$.
- $u : \mathbb{R} \to \mathbb{R}$ is a **utility function**.
- π is an $I \times 1$ vector of nonnegative **probabilities** over states, with $\pi_i \geq 0, \sum_{i=1}^{I} \pi_i = 1$.
- **Relative entropy** $\text{ent}(\pi, \hat{\pi})$ of a probability vector $\hat{\pi}$ with respect to a probability vector π is the expected value of the logarithm of the likelihood ratio $m_i \doteq \left(\frac{\hat{\pi}_i}{\pi_i}\right)$ under distribution $\hat{\pi}$ defined as:

$$\text{ent}(\pi, \hat{\pi}) = \sum_{i=1}^{I} \hat{\pi}_i \log\Big(\frac{\hat{\pi}_i}{\pi_i}\Big) = \sum_{i=1}^{I} \pi_i \Big(\frac{\hat{\pi}_i}{\pi_i}\Big) \log\Big(\frac{\hat{\pi}_i}{\pi_i}\Big)$$

or

$$\text{ent}(\pi, \hat{\pi}) = \sum_{i=1}^{I} \pi_i m_i \log m_i.$$

Remark: A likelihood ratio m_i is a discrete random variable. For any discrete random variable $\{x_i\}_{i=1}^{I}$, the expected value of x under the $\hat{\pi}_i$ distribution can be represented as the expected value under the π distribution of the product of x_i times the `shock' m_i:

$$\hat{E}x = \sum_{i=1}^{I} x_i \hat{\pi}_i = \sum_{i=1}^{I} m_i x_i \pi_i = Emx,$$

where $\hat{E}$ is the mathematical expectation under the $\hat{\pi}$ distribution and E is the expectation under the π distribution.

Evidently,

$$\hat{E}1 = Em = 1$$

and relative entropy is

$$Em \log m = \hat{E} \log m.$$

In the three figures below, we plot relative entropy from several perspectives.

Our first figure depicts entropy as a function of $\hat{\pi}_1$ when $I = 2$ and $\pi_1 = .5$.

When $\pi_1 \in (0, 1)$, entropy is finite for both $\hat{\pi}_1 = 0$ and $\hat{\pi}_1 = 1$ because $\lim_{x \to 0} x \log x = 0$

However, when $\pi_1 = 0$ or $\pi_1 = 1$, entropy is infinite.

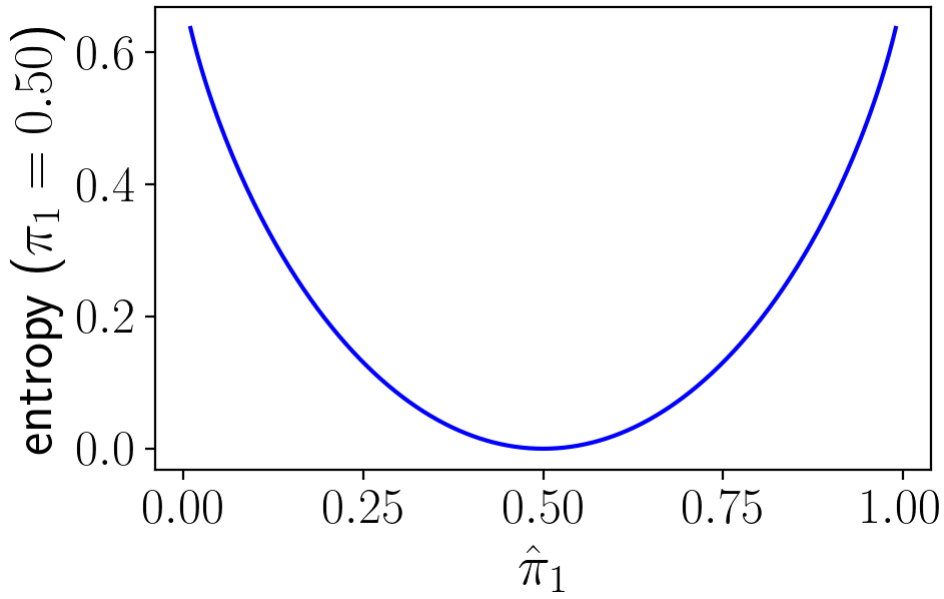
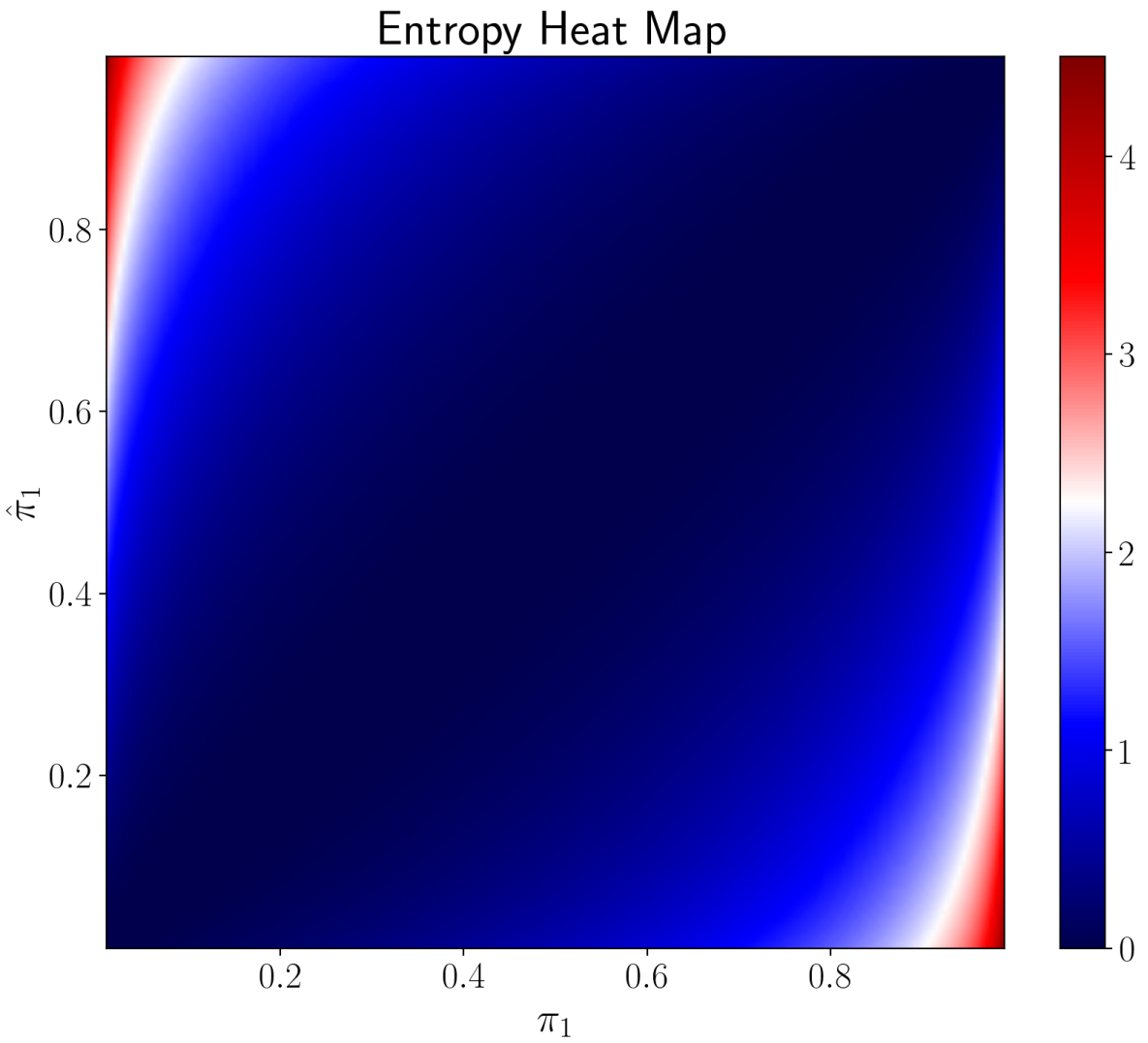


Fig. 25.1: Figure 1

The heat maps in the next two figures vary both $\hat{\pi}_1$ and π_1.

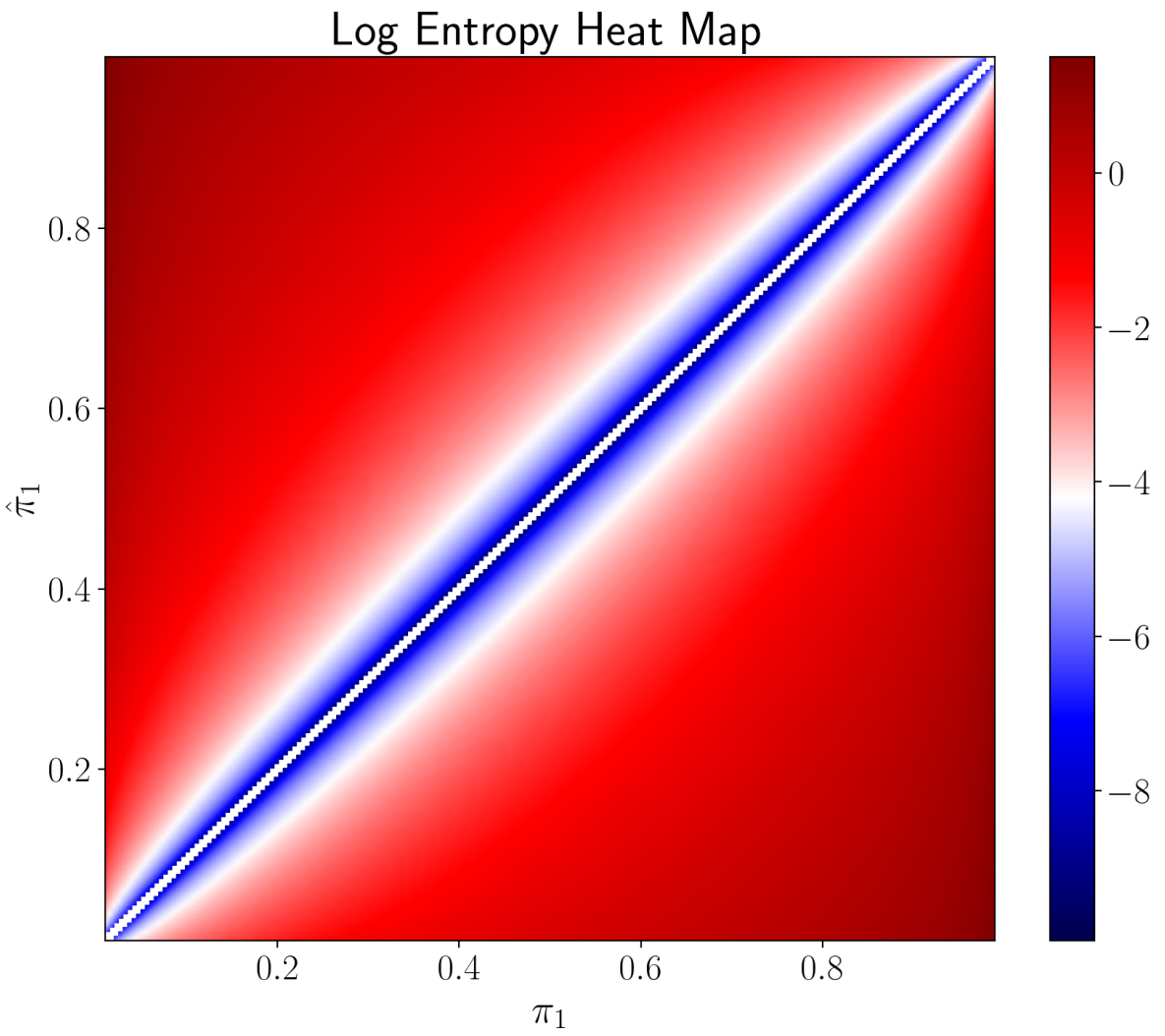
The following figure plots entropy.



The next figure plots the logarithm of entropy.

```
3.8205752275831846
```

```
/tmp/ipykernel_7948/3759713737.py:2: RuntimeWarning: divide by zero encountered in␣
↪log
  plt.pcolormesh(x, y, np.log(ent_vals_mat.T), shading='gouraud', cmap='seismic')
```



25.3 Five preference specifications

We describe five types of preferences over plans.

- Expected utility preferences
- Constraint preferences
- Multiplier preferences
- Risk-sensitive preferences
- Ex post Bayesian expected utility preferences

Expected utility, risk-sensitive, and ex post Bayesian prefernces are each cast in terms of a unique probability distribution, so they can express risk-aversion, but not model ambiguity aversion.

Multiplier and constraint prefernces both express aversion to concerns about model misppecification, i.e., model uncertainty; both are cast in terms of a set or sets of probability distributions.

- The set of distributions expresses the decision maker's ambiguity about the probability model.

- Minimization over probability distributions expresses his aversion to ambiguity.

25.4 Expected utility

A decision maker is said to have **expected utility preferences** when he ranks plans c by their expected utilities

$$\sum_{i=1}^{I} u(c_i)\pi_i, \tag{25.1}$$

where u is a unique utility function and π is a unique probability measure over states.

- A known π expresses risk.
- Curvature of u expresses risk aversion.

25.5 Constraint preferences

A decision maker is said to have **constraint preferences** when he ranks plans c according to

$$\min_{\{m_i \geq 0\}_{i=1}^{I}} \sum_{i=1}^{I} m_i \pi_i u(c_i) \tag{25.2}$$

subject to

$$\sum_{i=1}^{I} \pi_i m_i \log m_i \leq \eta \tag{25.3}$$

and

$$\sum_{i=1}^{I} \pi_i m_i = 1. \tag{25.4}$$

In (25.3), $\eta \geq 0$ defines an entropy ball of probability distributions $\hat{\pi} = m\pi$ that surround a baseline distribution π.

As noted earlier, $\sum_{i=1}^{I} m_i \pi_i u(c_i)$ is the expected value of $u(c)$ under a twisted probability distribution $\{\hat{\pi}_i\}_{i=1}^{I} = \{m_i \pi_i\}_{i=1}^{I}$.

Larger values of the entropy constraint η indicate more apprehension about the baseline probability distribution $\{\pi_i\}_{i=1}^{I}$.

Following [Hansen and Sargent, 2001] and [Hansen and Sargent, 2008], we call minimization problem (25.2) subject to (25.3) and(25.4) a **constraint problem**.

To find minimizing probabilities, we form a Lagrangian

$$L = \sum_{i=1}^{I} m_i \pi_i u(c_i) + \tilde{\theta}[\sum_{i=1}^{I} \pi_i m_i \log m_i - \eta] \tag{25.5}$$

where $\tilde{\theta} \geq 0$ is a Lagrange multiplier associated with the entropy constraint.

Subject to the additional constraint that $\sum_{i=1}^{I} m_i \pi_i = 1$, we want to minimize (25.5) with respect to $\{m_i\}_{i=1}^{I}$ and to maximize it with respect to $\tilde{\theta}$.

The minimizing probability distortions (likelihood ratios) are

$$\tilde{m}_i(c;\tilde{\theta}) = \frac{\exp(-u(c_i)/\tilde{\theta})}{\sum_j \pi_j \exp(-u(c_j)/\tilde{\theta})}. \tag{25.6}$$

To compute the Lagrange multiplier $\tilde{\theta}(c,\eta)$, we must solve

$$\sum_i \pi_i \tilde{m}_i(c;\tilde{\theta}) \log(\tilde{m}_i(c;\tilde{\theta})) = \eta$$

or

$$\sum_i \pi_i \frac{\exp(-u(c_i)/\tilde{\theta})}{\sum_j \pi_j \exp(-u(c_j)/\tilde{\theta})} \log\left[\frac{\exp(-u(c_i)/\tilde{\theta})}{\sum_j \pi_j \exp(-u(c_j)/\tilde{\theta})}\right] = \eta \tag{25.7}$$

for $\tilde{\theta} = \tilde{\theta}(c;\eta)$.

For a fixed η, the $\tilde{\theta}$ that solves equation (25.7) is evidently a function of the consumption plan c.

With $\tilde{\theta}(c;\eta)$ in hand we can obtain worst-case probabilities as functions $\pi_i \tilde{m}_i(c;\eta)$ of η.

The **indirect (expected) utility function** under constraint preferences is

$$\sum_{i=1}^I \pi_i \tilde{m}_i(c_i;\eta) u(c_i) = \sum_{i=1}^I \pi_i \left[\frac{\exp(-\tilde{\theta}^{-1}u(c_i))}{\sum_{j=1}^I \exp(-\tilde{\theta}^{-1}u(c_j))\pi_j}\right] u(c_i). \tag{25.8}$$

Entropy evaluated at the minimizing probability distortion (25.6) equals $E\tilde{m}\log\tilde{m}$ or

$$\begin{aligned} & \sum_{i=1}^I \left[\frac{\exp(-\tilde{\theta}^{-1}u(c_i))}{\sum_{j=1}^I \exp(-\tilde{\theta}^{-1}u(c_j))\pi_j}\right] \times \\ & \left\{-\tilde{\theta}^{-1}u(c_i) + \log\left(\sum_{j=1}^I \exp(-\tilde{\theta}^{-1}u(c_j))\pi_j\right)\right\}\pi_i \\ = \quad & -\tilde{\theta}^{-1}\sum_{i=1}^I \pi_i \left[\frac{\exp(-\tilde{\theta}^{-1}u(c_i))}{\sum_{j=1}^I \exp(-\tilde{\theta}^{-1}u(c_j))\pi_j}\right] u(c_i) \\ & + \log\left(\sum_{j=1}^I \exp(-\tilde{\theta}^{-1}u(c_j))\pi_j\right). \end{aligned} \tag{25.9}$$

Expression (25.9) implies that

$$\begin{aligned} -\tilde{\theta}\log\left(\sum_{j=1}^I \exp(-\tilde{\theta}^{-1}u(c_j))\pi_j\right) = & \sum_{i=1}^I \pi_i \left[\frac{\exp(-\tilde{\theta}^{-1}u(c_i))}{\sum_{j=1}^I \exp(-\tilde{\theta}^{-1}u(c_j))\pi_j}\right] u(c_i) \\ & + \tilde{\theta}(c;\eta)\sum_{i=1}^I \log\tilde{m}_i(c;\eta)\tilde{m}_i(c;\eta)\pi_i, \end{aligned} \tag{25.10}$$

where the last term is $\tilde{\theta}$ times the entropy of the worst-case probability distribution.

25.6 Multiplier preferences

A decision maker is said to have **multiplier preferences** when he ranks consumption plans c according to

$$\mathsf{T}u(c) \doteq \min_{\{m_i \geq 0\}_{i=1}^I} \sum_{i=1}^I \pi_i m_i [u(c_i) + \theta \log m_i] \tag{25.11}$$

where minimization is subject to

$$\sum_{i=1}^I \pi_i m_i = 1.$$

Here $\theta \in (\underline{\theta}, +\infty)$ is a 'penalty parameter' that governs a 'cost' to an 'evil alter ego' who distorts probabilities by choosing $\{m_i\}_{i=1}^I$.

Lower values of the penalty parameter θ express more apprehension about the baseline probability distribution π.

Following [Hansen and Sargent, 2001] and [Hansen and Sargent, 2008], we call the minimization problem on the right side of (25.11) a **multiplier problem**.

The minimizing probability distortion that solves the multiplier problem is

$$\hat{m}_i(c;\theta) = \frac{\exp(-u(c_i)/\theta)}{\sum_j \pi_j \exp(-u(c_j)/\theta)}. \tag{25.12}$$

We can solve

$$\sum_i \pi_i \frac{\exp(-u(c_i)/\theta)}{\sum_j \pi_j \exp(-u(c_j)/\theta)} \log\left[\frac{\exp(-u(c_i)/\theta)}{\sum_j \pi_j \exp(-u(c_j)/\theta)}\right] = \tilde{\eta} \tag{25.13}$$

to find an entropy level $\tilde{\eta}(c;\theta)$ associated with multiplier preferences with penalty parameter θ and allocation c.

For a fixed θ, the $\tilde{\eta}$ that solves equation (25.13) is a function of the consumption plan c

The forms of expressions (25.6) and (25.12) are identical, but the Lagrange multiplier $\tilde{\theta}$ appears in (25.6), while the penalty parameter θ appears in (25.12).

Formulas (25.6) and (25.12) show that worst-case probabilities are **context specific** in the sense that they depend on both the utility function u and the consumption plan c.

If we add θ times entropy under the worst-case model to expected utility under the worst-case model, we find that the **indirect expected utility function** under multiplier preferences is

$$-\theta \log\left(\sum_{j=1}^I \exp(-\theta^{-1}u(c_j))\pi_j\right). \tag{25.14}$$

25.7 Risk-sensitive preferences

Substituting $\hat{m}_i$ into $\sum_{i=1}^I \pi_i \hat{m}_i [u(c_i) + \theta \log \hat{m}_i]$ gives the indirect utility function

$$\mathsf{T}u(c) \doteq -\theta \log \sum_{i=1}^I \pi_i \exp(-u(c_i)/\theta). \tag{25.15}$$

Here $\mathsf{T}u$ in (25.15) is the **risk-sensitivity** operator of [Jacobson, 1973], [Whittle, 1981], and [Whittle, 1990].

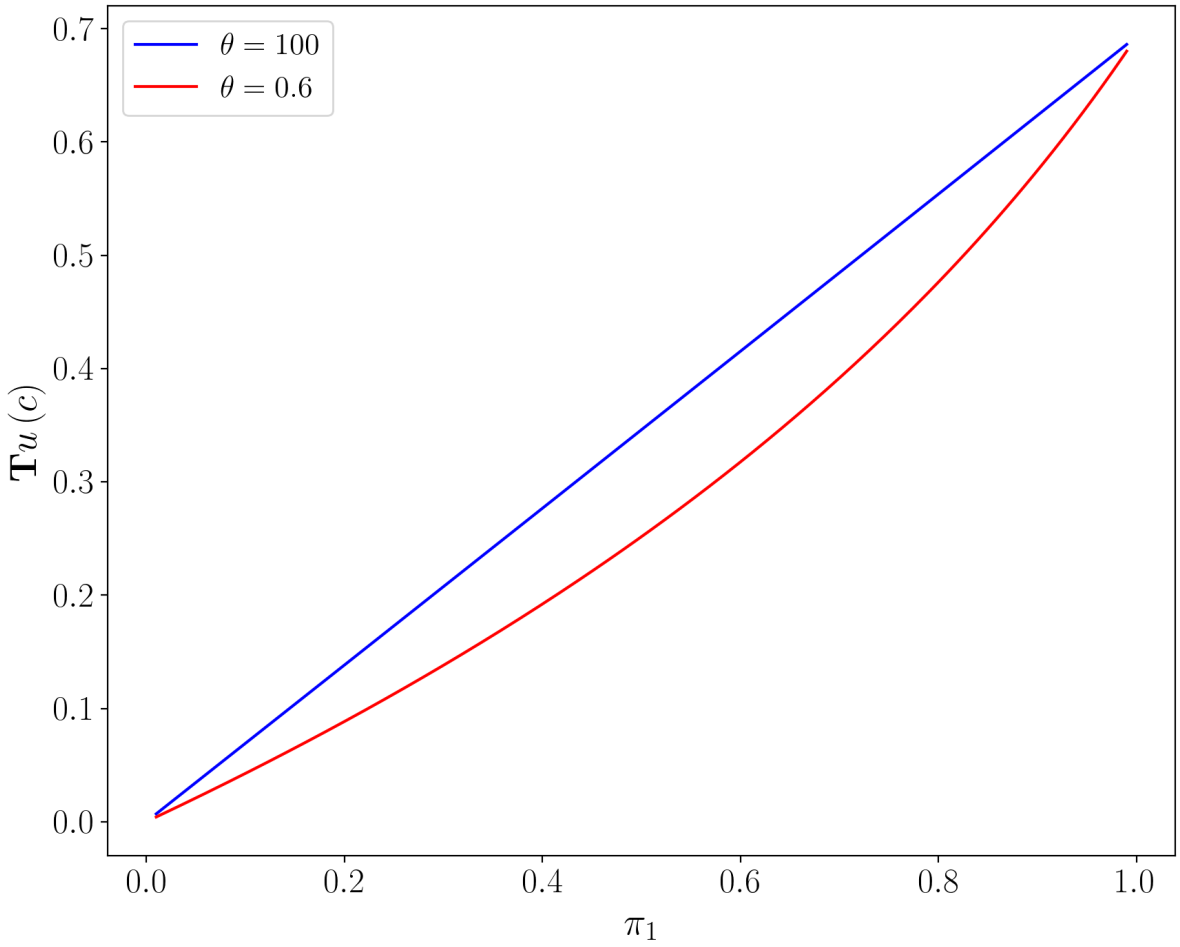
It defines a **risk-sensitive** preference ordering over plans c.

Because it is not linear in utilities $u(c_i)$ and probabilities π_i, it is said not to be separable across states.

Because risk-sensitive preferences use a unique probability distribution, they apparently express no model distrust or ambiguity.

Instead, they make an additional adjustment for risk-aversion beyond that embedded in the curvature of u.

For $I = 2, c_1 = 2, c_2 = 1, u(c) = \ln c$, the following figure plots the risk-sensitive criterion $\mathsf{T}u(c)$ defined in (25.15) as a function of π_1 for values of θ of 100 and .6.

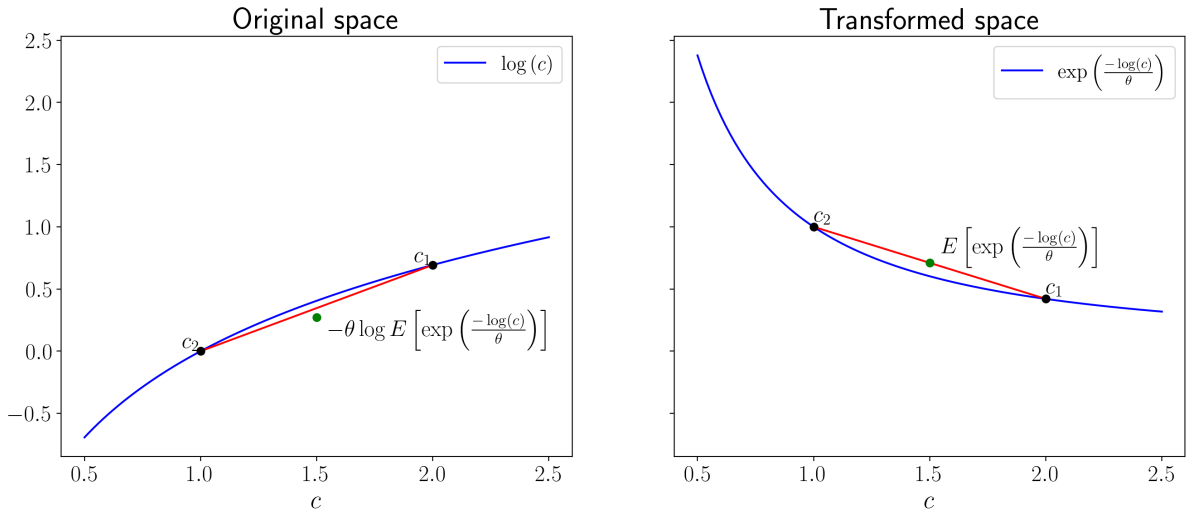


For large values of θ, $\mathsf{T}u(c)$ is approximately linear in the probability π_1, but for lower values of θ, $\mathsf{T}u(c)$ has considerable curvature as a function of π_1.

Under expected utility, i.e., $\theta = +\infty$, $\mathsf{T}u(c)$ is linear in π_1, but it is convex as a function of π_1 when $\theta < +\infty$.

The two panels in the next figure below can help us to visualize the extra adjustment for risk that the risk-sensitive operator entails.

This will help us understand how the $\mathbf{T}$ transformation works by envisioning what function is being averaged.



The panel on the right portrays how the transformation $\exp\left(\frac{-u(c)}{\theta}\right)$ sends $u(c)$ to a new function by (i) flipping the sign, and (ii) increasing curvature in proportion to θ.

In the left panel, the red line is our tool for computing the mathematical expectation for different values of π.

The green lot indicates the mathematical expectation of $\exp\left(\frac{-u(c)}{\theta}\right)$ when $\pi = .5$.

Notice that the distance between the green dot and the curve is greater in the transformed space than the original space as a result of additional curvature.

The inverse transformation $\theta \log E\left[\exp\left(\frac{-u(c)}{\theta}\right)\right]$ generates the green dot on the left panel that constitutes the risk-sensitive utility index.

The gap between the green dot and the red line on the left panel measures the additional adjustment for risk that risk-sensitive preferences make relative to plain vanilla expected utility preferences.

25.7.1 Digression on moment generating functions

The risk-sensitivity operator T is intimately connected to a moment generating function.

In particular, a principal constinuent of the T operator, namely,

$$E \exp(-u(c_i)/\theta) = \sum_{i=1}^{I} \pi_i \exp(-u(c_i)/\theta)$$

is evidently a **moment generating function** for the random variable $u(c_i)$, while

$$g(\theta^{-1}) \doteq \log \sum_{i=1}^{I} \pi_i \exp(-u(c_i)/\theta)$$

is a **cumulant generating function**,

$$g(\theta^{-1}) = \sum_{j=1}^{\infty} \kappa_j \frac{(-\theta^{-1})^j}{j!}.$$

where κ_j is the jth cumulant of the random variable $u(c)$.

Then

$$\mathsf{T}u(c) = -\theta g(\theta^{-1}) = -\theta \sum_{j=1}^{\infty} \kappa_j \frac{(-\theta^{-1})^j}{j!}.$$

In general, when $\theta < +\infty$, $\mathsf{T}u(c)$ depends on cumulants of all orders.

These statements extend to cases with continuous probability distributions for c and therefore for $u(c)$.

For the special case $u(c) \sim \mathcal{N}(\mu_u, \sigma_u^2)$, $\kappa_1 = \mu_u, \kappa_2 = \sigma_u^2$, and $\kappa_j = 0 \; \forall j \geq 3$, so

$$\mathsf{T}u(c) = \mu_u - \frac{1}{2\theta}\sigma_u^2, \tag{25.16}$$

which becomes expected utility μ_u when $\theta^{-1} = 0$.

The right side of equation (25.16) is a special case of **stochastic differential utility** preferences in which consumption plans are ranked not just by their expected utilities μ_u but also the variances σ_u^2 of their expected utilities.

25.8 Ex post Bayesian preferences

A decision maker is said to have **ex post Bayesian preferences** when he ranks consumption plans according to the expected utility function

$$\sum_i \hat{\pi}_i(c^*)u(c_i) \tag{25.17}$$

where $\hat{\pi}(c^*)$ is the worst-case probability distribution associated with multiplier or constraint preferences evaluated at a particular consumption plan $c^* = \{c_i^*\}_{i=1}^I$.

At c^*, an ex post Bayesian's indifference curves are tangent to those for multiplier and constraint preferences with appropriately chosen θ and η, respectively.

25.9 Comparing preferences

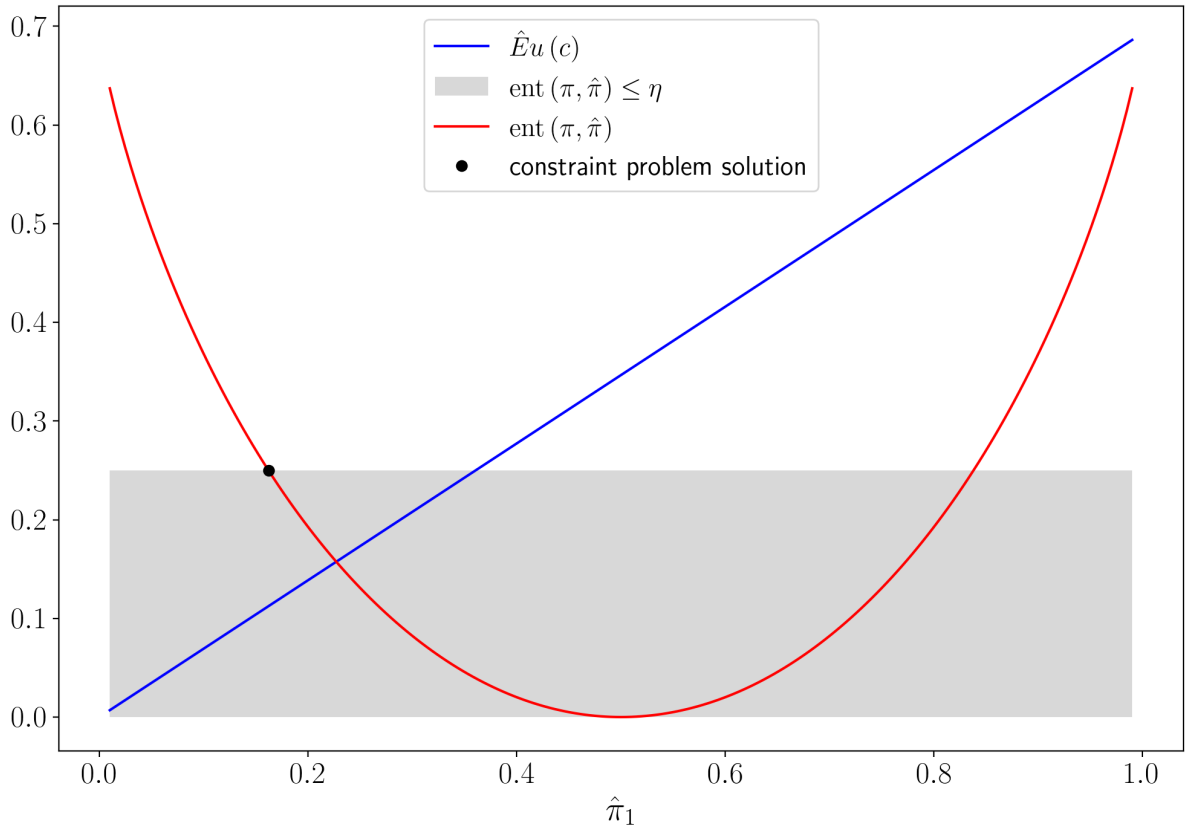
For the special case in which $I = 2$, $c_1 = 2, c_2 = 1$, $u(c) = \ln c$, and $\pi_1 = .5$, the following two figures depict how worst-case probabilities are determined under constraint and multiplier preferences, respectively.

The first figure graphs entropy as a function of $\hat{\pi}_1$.

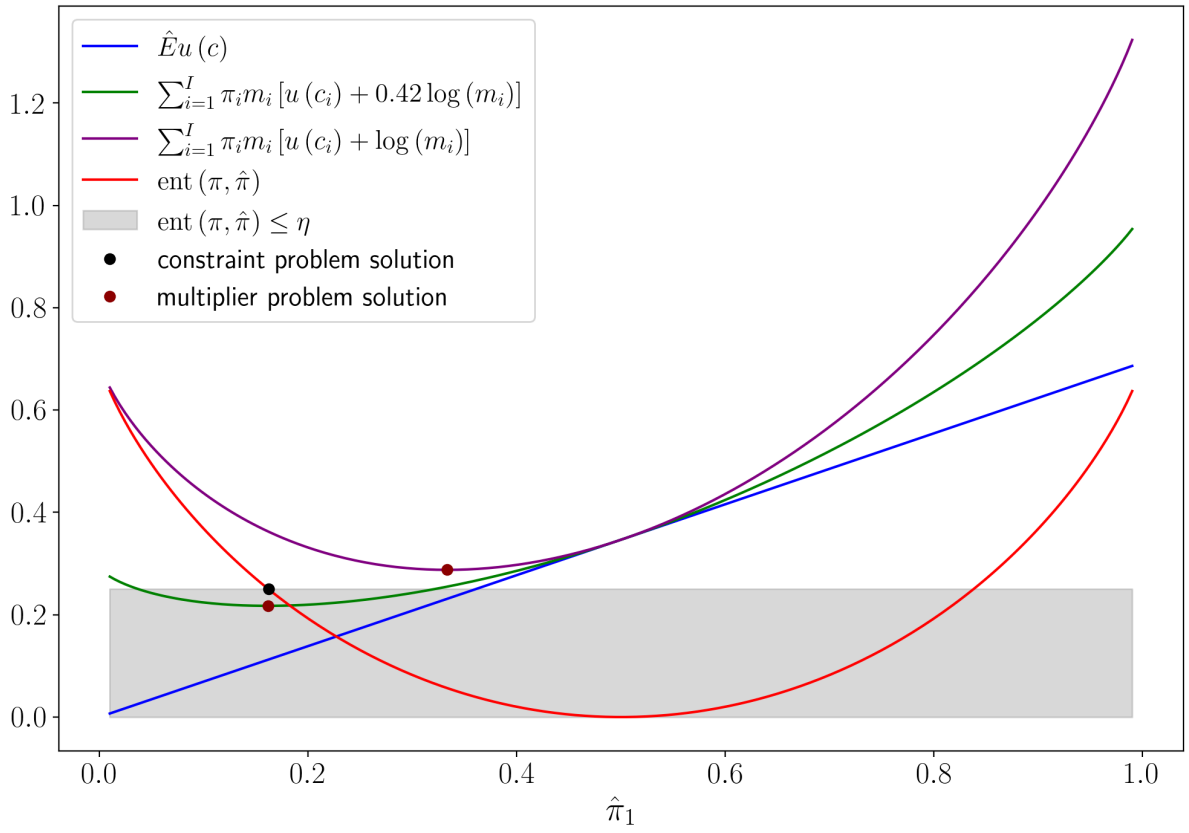
It also plots expected utility under the twisted probability distribution, namely, $\hat{E}u(c) = u(c_2) + \hat{\pi}_1(u(c_1) - u(c_2))$, which is evidently a linear function of $\hat{\pi}_1$.

The entropy constraint $\sum_{i=1}^I \pi_i m_i \log m_i \leq \eta$ implies a convex set $\hat{\Pi}_1$ of $\hat{\pi}_1$'s that constrains the adversary who chooses $\hat{\pi}_1$, namely, the set of $\hat{\pi}_1$'s for which the entropy curve lies below the horizontal dotted line at an entropy level of $\eta = .25$.

Unless $u(c_1) = u(c_2)$, the $\hat{\pi}_1$ that minimizes $\hat{E}u(c)$ is at the boundary of the set $\hat{\Pi}_1$.



The next figure shows the function $\sum_{i=1}^{I} \pi_i m_i [u(c_i) + \theta \log m_i]$ that is to be minimized in the multiplier problem. The argument of the function is $\hat{\pi}_1 = m_1 \pi_1$.



Evidently, from this figure and also from formula (25.12), lower values of θ lead to lower, and thus more distorted, minimizing values of $\hat{\pi}_1$.

The figure indicates how one can construct a Lagrange multiplier $\tilde{\theta}$ associated with a given entropy constraint η and a given consumption plan.

Thus, to draw the figure, we set the penalty parameter for multiplier preferences θ so that the minimizing $\hat{\pi}_1$ equals the minimizing $\hat{\pi}_1$ for the constraint problem from the previous figure.

The penalty parameter $\theta = .42$ also equals the Lagrange multiplier $\tilde{\theta}$ on the entropy constraint for the constraint preferences depicted in the previous figure because the $\hat{\pi}_1$ that minimizes the asymmetric curve associated with penalty parameter $\theta = .42$ is the same $\hat{\pi}_1$ associated with the intersection of the entropy curve and the entropy constraint dashed vertical line.

25.10 Risk aversion and misspecification aversion

All five types of preferences use curvature of u to express risk aversion.

Constraint preferences express **concern about misspecification** or **ambiguity** for short with a positive η that circumscribes an entropy ball around an approximating probability distribution π, and *aversion aversion to model misspecification* through minimization with respect to a likelihood ratio m.

Multiplier preferences express misspecification concerns with a parameter $\theta < +\infty$ that penalizes deviations from the approximating model as measured by relative entropy, and they express aversion to misspecification concerns with minimization over a probability distortion m.

By penalizing minimization over the likelihood ratio m, a decrease in θ represents an **increase** in ambiguity (or what [Knight, 1921] called uncertainty) about the specification of the baseline approximating model $\{\pi_i\}_{i=1}^{I}$.

Formulas (25.6) assert that the decision maker acts as if he is pessimistic relative to an approximating model π.

It expresses what [Bucklew, 2004] [p. 27] calls a statistical version of *Murphy's law*:

> **The probability of anything happening is in inverse ratio to its desirability.**

The minimizing likelihood ratio $\hat{m}$ slants worst-case probabilities $\hat{\pi}$ exponentially to increase probabilities of events that give lower utilities.

As expressed by the value function bound (25.19) to be displayed below, the decision maker uses **pessimism** instrumentally to protect himself against model misspecification.

The penalty parameter θ for multipler preferences or the entropy level η that determines the Lagrange multiplier $\tilde{\theta}$ for constraint preferences controls how adversely the decision maker exponentially slants probabilities.

A decision rule is said to be **undominated** in the sense of Bayesian decision theory if there exists a probability distribution π for which it is optimal.

A decision rule is said to be **admissible** if it is undominated.

[Hansen and Sargent, 2008] use ex post Bayesian preferences to show that robust decision rules are undominated and therefore admissible.

25.11 Indifference curves

Indifference curves illuminate how concerns about robustness affect asset pricing and utility costs of fluctuations. For $I = 2$, the slopes of the indifference curves for our five preference specifications are

- Expected utility:

$$\frac{dc_2}{dc_1} = -\frac{\pi_1}{\pi_2}\frac{u'(c_1)}{u'(c_2)}$$

- Constraint and ex post Bayesian preferences:

$$\frac{dc_2}{dc_1} = -\frac{\hat{\pi}_1}{\hat{\pi}_2}\frac{u'(c_1)}{u'(c_2)}$$

  where $\hat{\pi}_1, \hat{\pi}_2$ are the minimizing probabilities computed from the worst-case distortions (25.6) from the constraint problem at (c_1, c_2).

- Multiplier and risk-sensitive preferences:

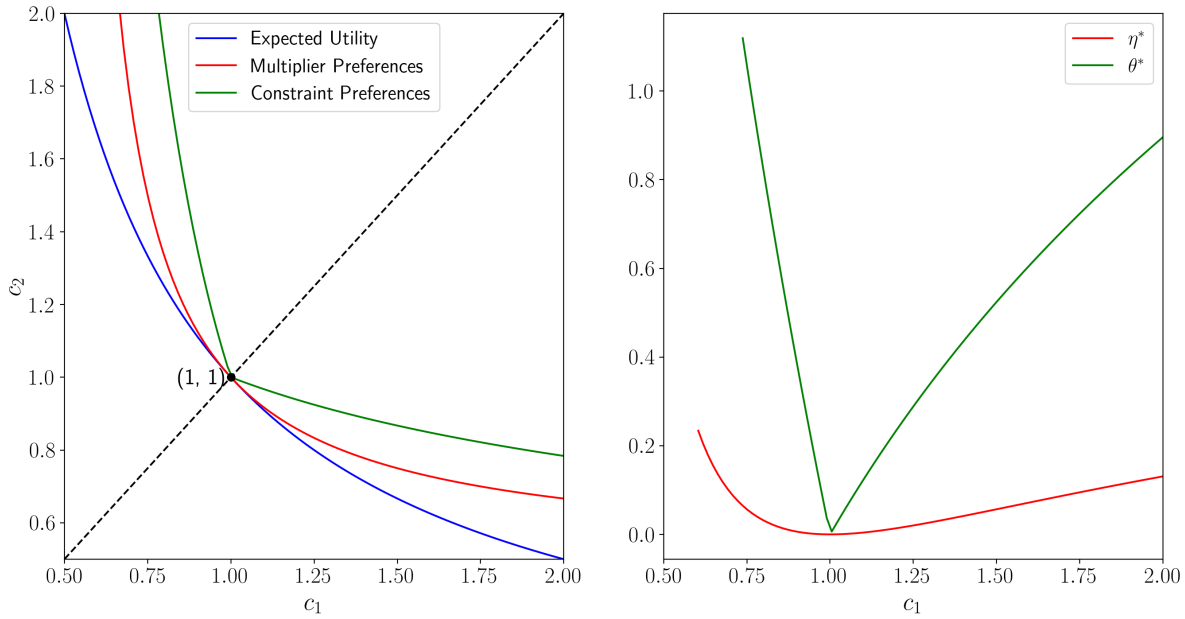
$$\frac{dc_2}{dc_1} = -\frac{\pi_1}{\pi_2}\frac{\exp(-u(c_1)/\theta)}{\exp(-u(c_2)/\theta)}\frac{u'(c_1)}{u'(c_2)}$$

When $c_1 > c_2$, the exponential twisting formula (25.12) implies that $\hat{\pi}_1 < \pi_1$, which in turn implies that the indifference curves through (c_1, c_2) for both constraint and multiplier preferences are flatter than the indifference curve associated with expected utility preferences.

As we shall see soon when we discuss state price deflators, this gives rise to higher estimates of prices of risk.

For an example with $u(c) = \ln c$, $I = 2$, and $\pi_1 = .5$, the next two figures show indifference curves for expected utility, multiplier, and constraint preferences.

The following figure shows indifference curves going through a point along the 45 degree line.



Kink at 45 degree line

Notice the kink in the indifference curve for constraint preferences at the 45 degree line.

To understand the source of the kink, consider how the Lagrange multiplier and worst-case probabilities vary with the consumption plan under constraint preferences.

For fixed η, a given plan c, and a utility function increasing in c, worst case probabilities are **fixed numbers** $\hat{\pi}_1 < .5$ when $c_1 > c_2$ and $\hat{\pi}_1 > .5$ when $c_2 > c_1$.

This pattern makes the Lagrange multiplier $\tilde{\theta}$ vary discontinuously at $\hat{\pi}_1 = .5$.

The discontinuity in the worst case $\hat{\pi}_1$ at the 45 degree line accounts for the kink at the 45 degree line in an indifference curve for constraint preferences associated with a given positive entropy constraint η.

The code for generating the preceding figure is somewhat intricate we formulate a root finding problem for finding indifference curves.

Here is a brief literary description of the method we use.

Parameters

- Consumption bundle $c = (1, 1)$
- Penalty parameter $\theta = 2$
- Utility function $u = \log$
- Probability vector $\pi = (0.5, 0.5)$

Algorithm:

- Compute $\bar{u} = \pi_1 u(c_1) + \pi_2 u(c_2)$
- Given values for c_1, solve for values of c_2 such that $\bar{u} = u(c_1, c_2)$:
  - Expected utility: $c_{2,EU} = u^{-1}\left(\frac{\bar{u} - \pi_1 u(c_1)}{\pi_2}\right)$
  - Multiplier preferences: solve $\bar{u} - \sum_i \pi_i \frac{\exp\left(\frac{-u(c_i)}{\theta}\right)}{\sum_j \exp\left(\frac{-u(c_j)}{\theta}\right)} \left(u(c_i) + \theta \log\left(\frac{\exp\left(\frac{-u(c_i)}{\theta}\right)}{\sum_j \exp\left(\frac{-u(c_j)}{\theta}\right)} \right) \right) = 0$ numerically

- Constraint preference: solve $\bar{u} - \sum_i \pi_i \frac{\exp\left(\frac{-u(c_i)}{\theta^*}\right)}{\sum_j \exp\left(\frac{-u(c_j)}{\theta^*}\right)} u(c_i) = 0$ numerically where θ^* solves $\sum_i \pi_i \frac{\exp\left(\frac{-u(c_i)}{\theta^*}\right)}{\sum_j \exp\left(\frac{-u(c_j)}{\theta^*}\right)} \log\left(\frac{\exp\left(\frac{-u(c_i)}{\theta^*}\right)}{\sum_j \exp\left(\frac{-u(c_j)}{\theta^*}\right)}\right) - \eta = 0$ numerically.

Remark: It seems that the constraint problem is hard to solve in its original form, i.e. by finding the distorting measure that minimizes the expected utility.

It seems that viewing equation (25.7) as a root finding problem works much better.

But notice that equation (25.7) does not always have a solution.

Under $u = \log$, $c_1 = c_2 = 1$, we have:

$$\sum_i \pi_i \frac{\exp\left(\frac{-u(c_i)}{\tilde{\theta}}\right)}{\sum_j \pi_j \exp\left(\frac{-u(c_j)}{\tilde{\theta}}\right)} \log\left(\frac{\exp\left(\frac{-u(c_i)}{\tilde{\theta}}\right)}{\sum_j \pi_j \exp\left(\frac{-u(c_j)}{\tilde{\theta}}\right)}\right) = 0$$

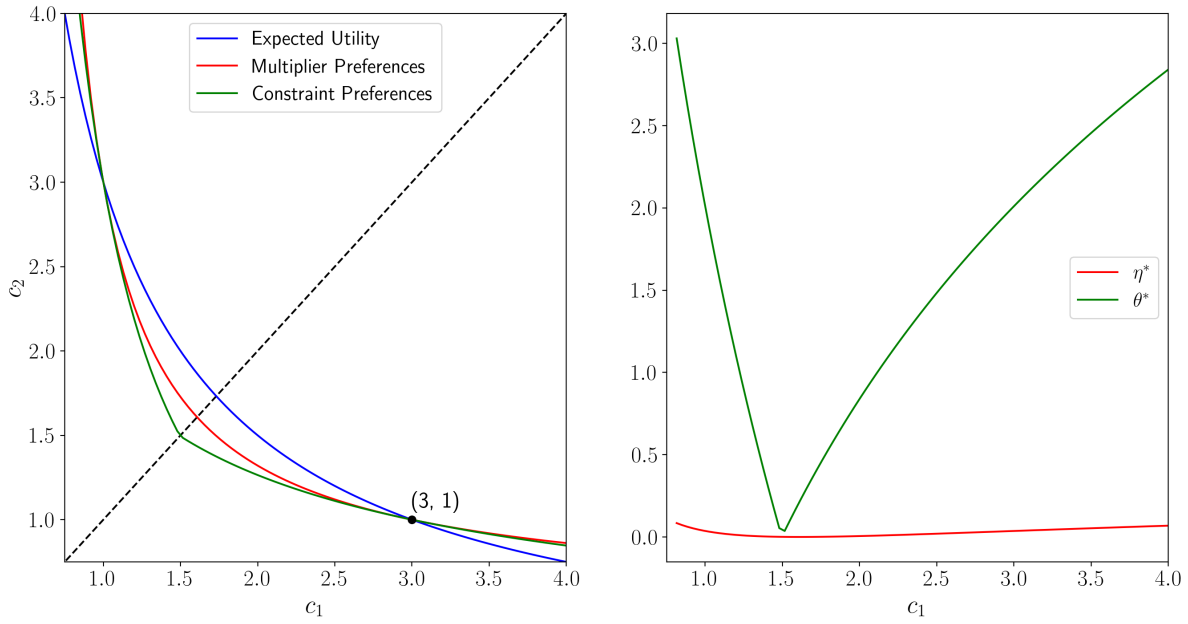
Conjecture: when our numerical method fails it because the derivative of the objective doesn't exist for our choice of parameters.

Remark: It is tricky to get the algorithm to work properly for all values of c_1. In particular, parameters were chosen with graduate student descent.

Tangent indifference curves off 45 degree line

For a given η and a given allocatin (c_1, c_2) off the 45 degree line, by solving equations (25.7) and (25.13), we can find $\tilde{\theta}(\eta, c)$ and $\tilde{\eta}(\theta, c)$ that make indifference curves for multiplier and constraint preferences be tangent to one another.

The following figure shows indifference curves for multiplier and constraint preferences through a point off the 45 degree line, namely, $(c(1), c(2)) = (3, 1)$, at which η and θ are adjusted to render the indifference curves for constraint and multiplier preferences tangent.



Note that all three lines of the left graph intersect at (1, 3). While the intersection at (3, 1) is hard-coded, the intersection at (1,3) arises from the computation, which confirms that the code seems to be working properly.

As we move along the (kinked) indifference curve for the constraint preferences for a given η, the worst-case probabilities remain constant, but the Lagrange multiplier $\tilde{\theta}$ on the entropy constraint $\sum_{i=1}^I m_i \log m_i \leq \eta$ varies with (c_1, c_2).

As we move along the (smooth) indifference curve for the multiplier preferences for a given penalty parameter θ, the implied entropy $\tilde{\eta}$ from equation (25.13) and the worst-case probabilities both change with (c_1, c_2).

For constraint preferences, there is a kink in the indifference curve.

For ex post Bayesian preferences, there are effectively two sets of indifference curves depending on which side of the 45 degree line the (c_1, c_2) endowment point sits.

There are two sets of indifference curves because, while the worst-case probabilities differ above and below the 45 degree line, the idea of ex post Bayesian preferences is to use a *single* probability distribution to compute expected utilities for all consumption bundles.

Indifference curves through point $(c_1, c_2) = (3, 1)$ for expected logarithmic utility (less curved smooth line), multiplier (more curved line), constraint (solid line kinked at 45 degree line), and *ex post* Bayesian (dotted lines) preferences. The worst-case probability $\hat{\pi}_1 < .5$ when $c_1 = 3 > c_2 = 1$ and $\hat{\pi}_1 > .5$ when $c_1 = 1 < c_2 = 3$.

25.12 State price deflators

Concerns about model uncertainty boost prices of risk that are embedded in state-price deflators. With complete markets, let q_i be the price of consumption in state i.

The budget set of a representative consumer having endowment $\bar{c} = \{\bar{c}_i\}_{i=1}^I$ is expressed by $\sum_i^I q_i(c_i - \bar{c}_i) \leq 0$.

When a representative consumer has multiplier preferences, the state prices are

$$q_i = \pi_i \hat{m}_i u'(\bar{c}_i) = \pi_i \left(\frac{\exp(-u(\bar{c}_i)/\theta)}{\sum_j \pi_j \exp(-u(\bar{c}_j)/\theta)} \right) u'(\bar{c}_i). \tag{25.18}$$

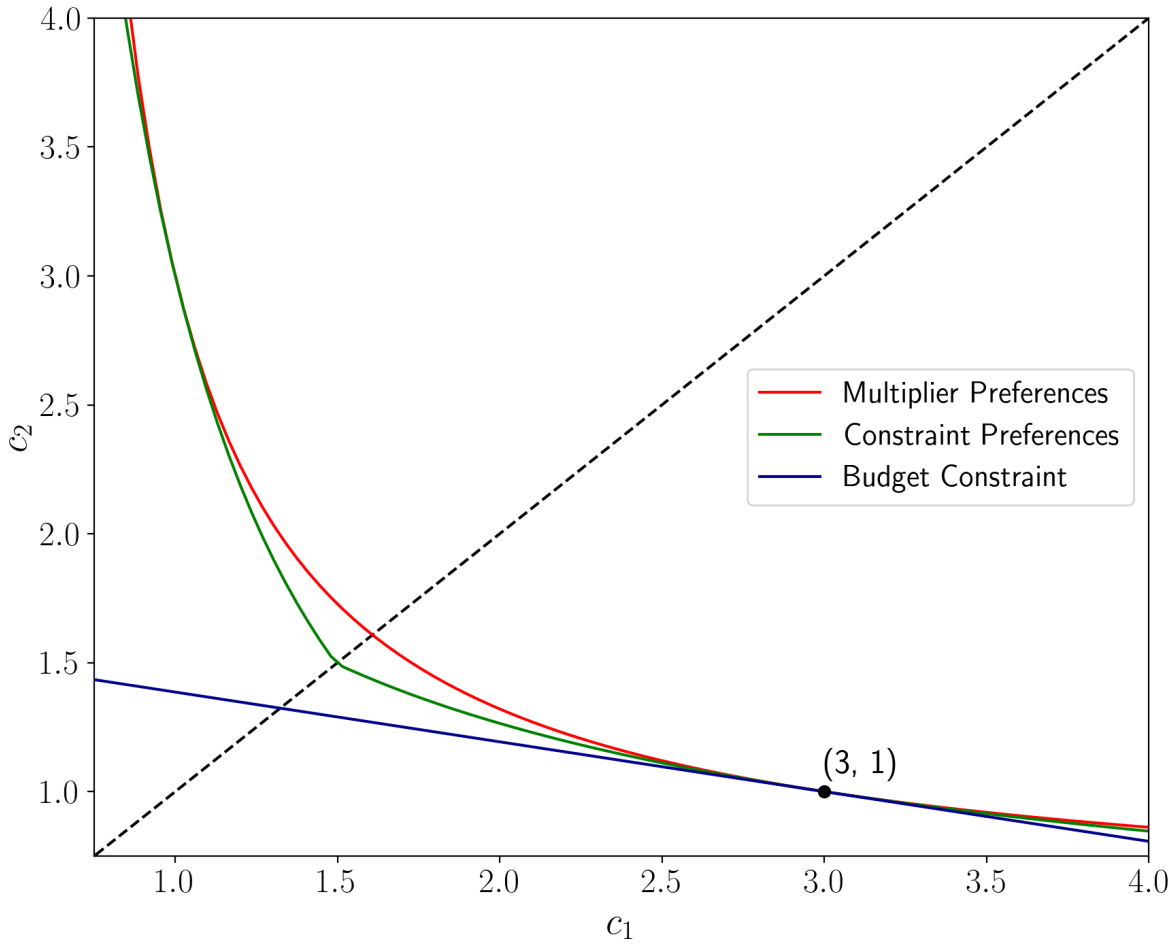
The worst-case likelihood ratio $\hat{m}_i$ operates to increase prices q_i in relatively low utility states i.

State prices agree under multiplier and constraint preferences when η and θ are adjusted according to (25.7) or (25.13) to make the indifference curves tangent at the endowment point.

The next figure can help us think about state-price deflators under our different preference orderings.

In this figure, budget line and indifference curves through point $(c_1, c_2) = (3, 1)$ for expected logarithmic utility, multiplier, constraint (kinked at 45 degree line), and *ex post* Bayesian (dotted lines) preferences.

Figure 2.7:



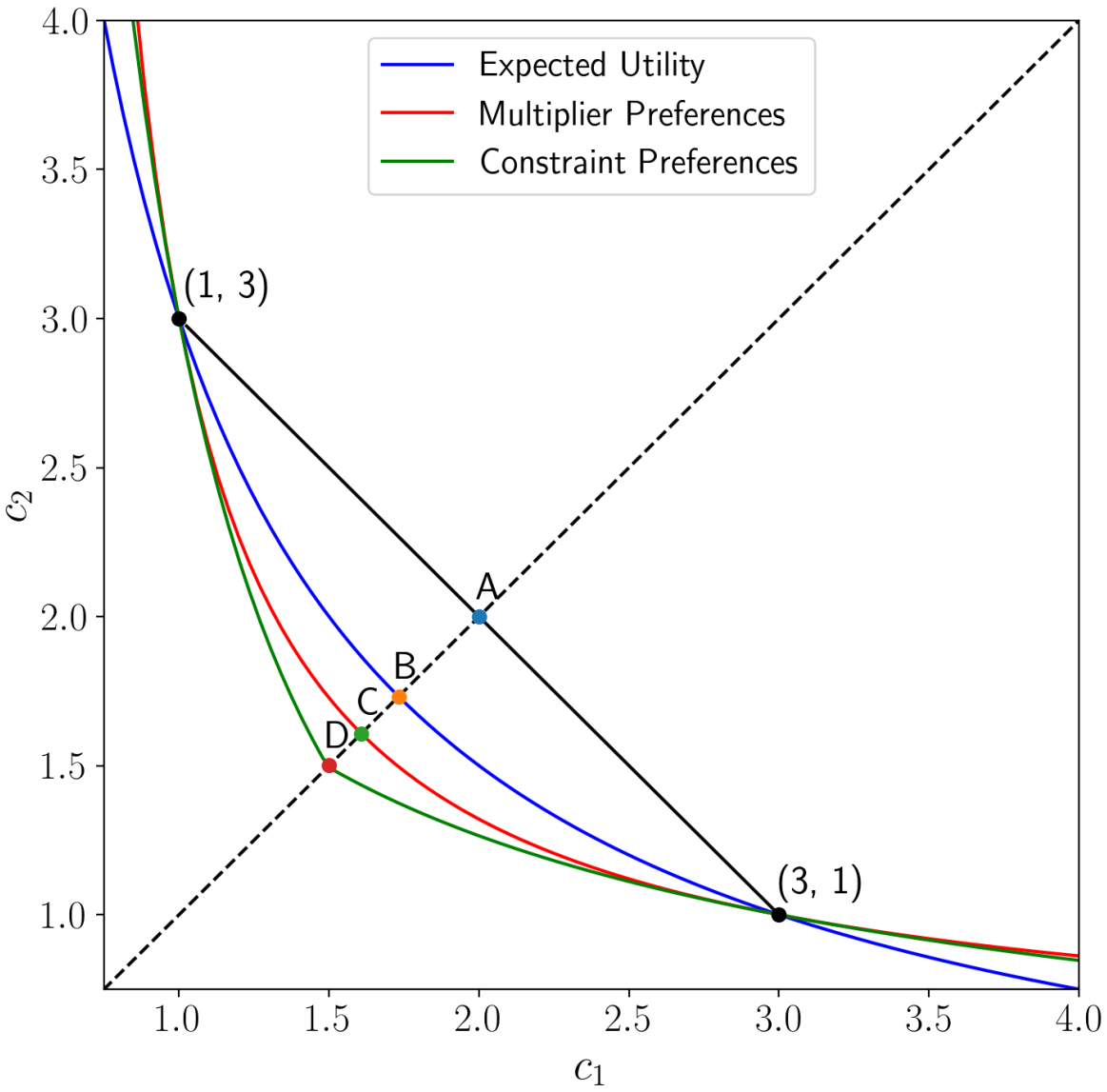
Because budget constraints are linear, asset prices are identical under multiplier and constraint preferences for which θ and η are adjusted to verify (25.7) or (25.13) at a given consumption endowment $\{c_i\}_{i=1}^I$.

However, as we note next, though they are tangent at the endowment point, the fact that indifference curves differ for multiplier and constraint preferences means that certainty equivalent consumption compensations of the kind that [Lucas, 1987], [Hansen *et al.*, 1999], [Tallarini, 2000], and [Barillas *et al.*, 2009] used to measure the costs of business cycles must differ.

25.12.1 Consumption-equivalent measures of uncertainty aversion

For each of our five types of preferences, the following figure allows us to construct a certainty equivalent point (c^*, c^*) on the 45 degree line that renders the consumer indifferent between it and the risky point $(c(1), c(2)) = (3, 1)$.

Figure 2.8:



The figure indicates that the certainty equivalent level c^* is higher for the consumer with expected utility preferences than for the consumer with multiplier preferences, and that it is higher for the consumer with multiplier preferences than for the consumer with constraint preferences.

The gap between these certainty equivalents measures the uncertainty aversion of the multiplier preferences or constraint preferences consumer.

The gap between the expected value $.5c(1) + .5c(2)$ at point A and the certainty equivalent for the expected utility decision maker at point B is a measure of his risk aversion.

The gap between points B and C measures the multiplier preference consumer's aversion to model uncertainty.

The gap between points B and D measures the constraint preference consumer's aversion to model uncertainty.

25.13 Iso-utility and iso-entropy curves and expansion paths

The following figures show iso-entropy and iso-utility lines for the special case in which $I = 3$, $\pi_1 = .3, \pi_2 = .4$, and the utility function is $u(c) = \frac{c^{1-\alpha}}{1-\alpha}$ with $\alpha = 0$ and $\alpha = 3$, respectively, for the fixed plan $c(1) = 1, c(2) = 2, c(3) = 3$.

The iso-utility lines are the level curves of

$$\hat{\pi}_1 c_1 + \hat{\pi}_2 c_2 + (1 - \hat{\pi}_1 - \hat{\pi}_2) c_3$$

and are linear in $(\hat{\pi}_1, \hat{\pi}_2)$.

This is what it means to say 'expected utility is linear in probabilities.'

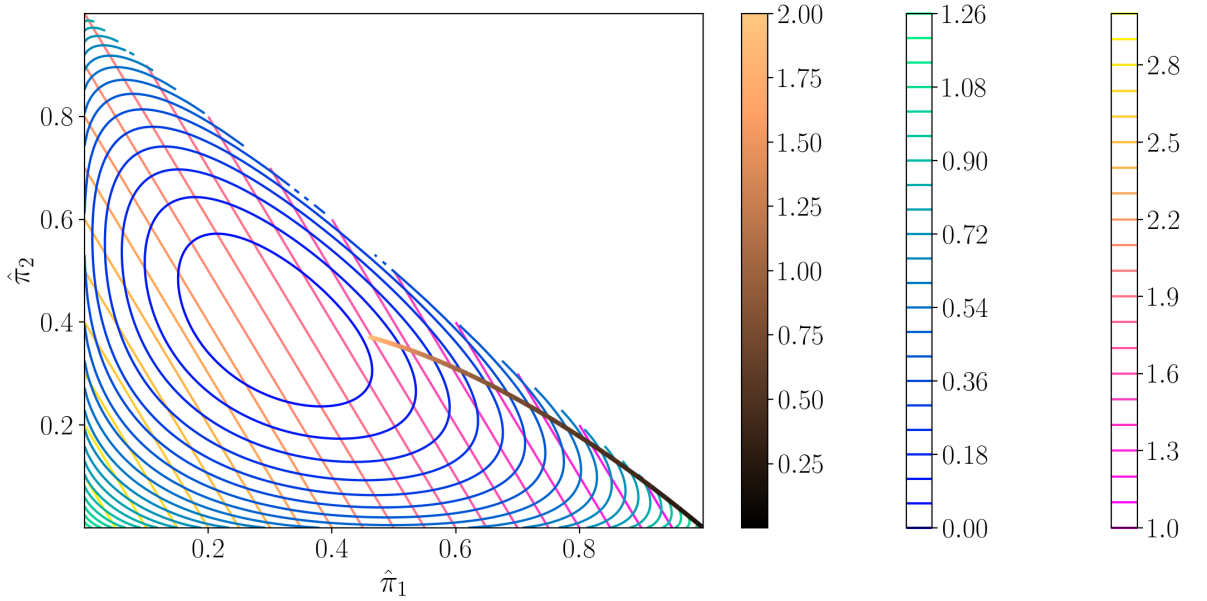
Both figures plot the locus of points of tangency between the iso-entropy and the iso-utility curves that is traced out as one varies θ^{-1} in the interval $[0, 2]$.

While the iso-entropy lines are identical in the two figures, these 'expansion paths' differ because the utility functions differ, meaning that for a given θ and (c_1, c_2, c_3) triple, the worst-case probabilities $\hat{\pi}_i(\theta) = \pi_i \frac{\exp(-u(c_i)/\theta)}{E \exp(-u(c)/\theta)}$ differ as we vary θ, causing the associated entropies to differ.

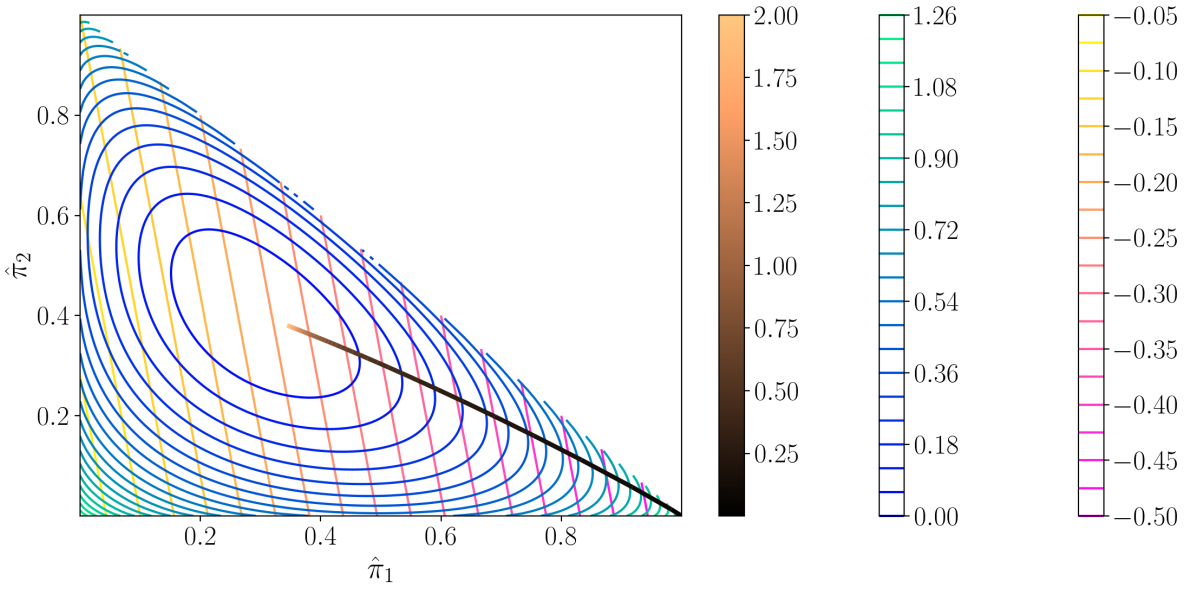
Color bars:

- First color bar: variation in θ
- Second color bar: variation in utility levels
- Third color bar: variation in entropy levels

```
/tmp/ipykernel_7948/3904427642.py:36: RuntimeWarning: invalid value encountered in␣
↪divide
  m =  m_unnormalized / (π * m_unnormalized).sum()
```




```
/tmp/ipykernel_7948/3904427642.py:35: RuntimeWarning: overflow encountered in exp
  m_unnormalized = np.exp(-u(c) / θ)
/tmp/ipykernel_7948/3904427642.py:36: RuntimeWarning: invalid value encountered in␣
↪divide
  m =  m_unnormalized / (π * m_unnormalized).sum()
```



25.14 Bounds on expected utility

Suppose that a decision maker wants a lower bound on expected utility $\sum_{i=1}^{I} \hat{\pi}_i u(c_i)$ that is satisfied for **any** distribution $\hat{\pi}$ with relative entropy less than or equal to η.

An attractive feature of multiplier and constraint preferences is that they carry with them such a bound.

To show this, it is useful to collect some findings in the following string of inequalities associated with multiplier preferences:

$$\begin{aligned} \mathsf{T}_\theta u(c) = & \quad -\theta \log \sum_{i=1}^{I} \exp\left(\frac{-u(c_i)}{\theta}\right)\pi_i \\ = & \quad \sum_{i=1}^{I} m_i^* \pi_i (u(c_i) + \theta \log m_i^*) \\ \leq & \quad \sum_{i=1}^{I} m_i \pi_i u(c_i) + \theta \sum_{i=1}^{i} m_i \log m_i \pi_i \end{aligned}$$

where $m_i^* \propto \exp\left(\frac{-u(c_i)}{\theta}\right)$ are the worst-case distortions to probabilities.

The inequality in the last line just asserts that minimizers minimize.

Therefore, we have the following useful bound:

$$\sum_{i=1}^{I} m_i \pi_i u(c_i) \geq \mathsf{T}_\theta u(c) - \theta \sum_{i=1}^{I} \pi_i m_i \log m_i. \tag{25.19}$$

The left side is expected utility under the probability distribution $\{m_i \pi_i\}$.

The right side is a *lower bound* on expected utility under *all* distributions expressed as an affine function of relative entropy $\sum_{i=1}^{I} \pi_i m_i \log m_i$.

The bound is attained for $m_i = m_i^* \propto \exp\left(\frac{-u(c_i)}{\theta}\right)$.

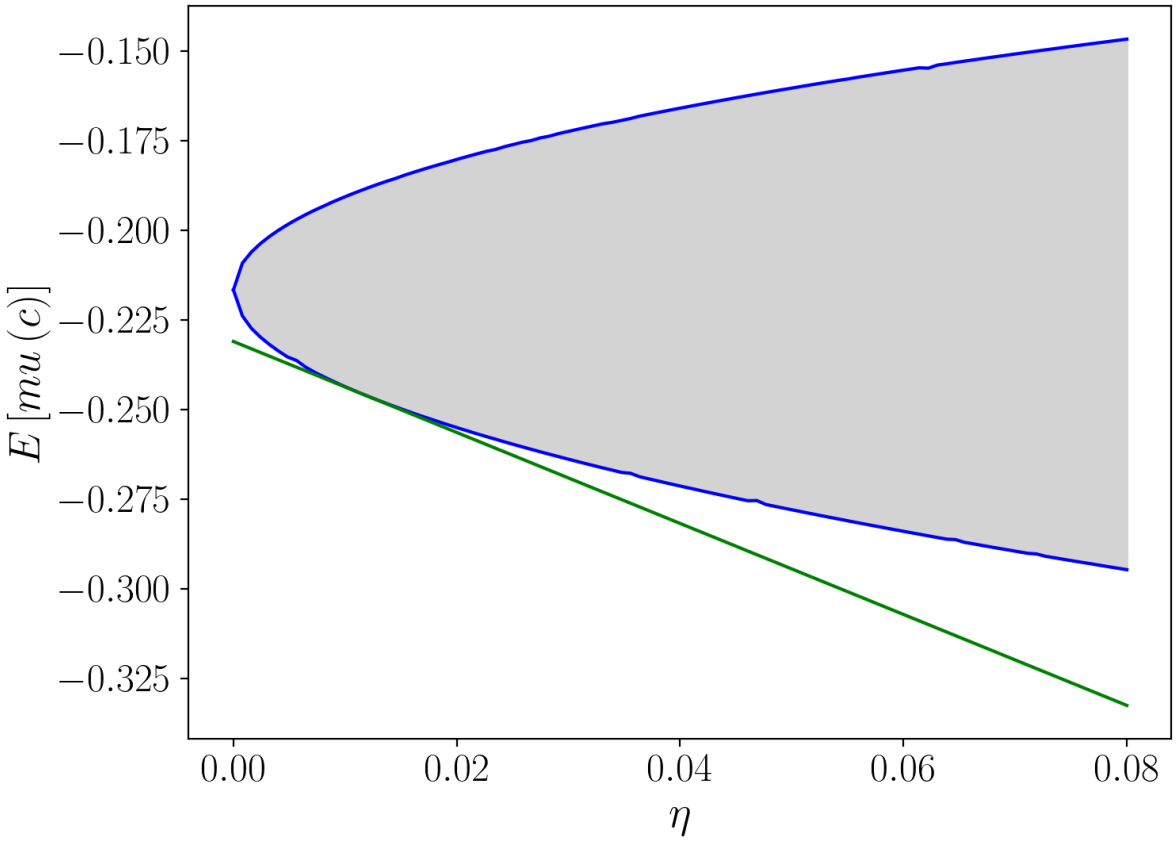
The *intercept* in the bound is the risk-sensitive criterion $\mathsf{T}_\theta u(c)$, while the *slope* is the penalty parameter θ.

Lowering θ does two things:

- it lowers the intercept $\mathsf{T}_\theta u(c)$, which makes the bound less informative for small values of entropy; and
- it lowers the absolute value of the slope, which makes the bound more informative for larger values of relative entropy $\sum_{i=1}^I \pi_i m_i \log m_i$.

The following figure reports best-case and worst-case expected utilities.

We calculate the lines in this figure numerically by solving optimization problems with respect to the change of measure.



In this figure, expected utility is on the co-ordinate axis while entropy is on the ordinate axis.

The *lower* curved line depicts expected utility under the worst-case model associated with each value of entropy η recorded on the ordinate axis, i.e., it is $\sum_{i=1}^I \pi_i \tilde{m}_i(\tilde{\theta}(c,\eta))u(c_i)$, where $\tilde{m}_i(\tilde{\theta}(\eta)) \propto \exp\left(\frac{-u(c_i)}{\tilde{\theta}}\right)$ and $\tilde{\theta}$ is the Lagrange multiplier associated with the constraint that entropy cannot exceed the value on the ordinate axis.

The *higher* curved line depicts expected utility under the *best*-case model indexed by the value of the Lagrange multiplier $\check{\theta} > 0$ associated with each value of entropy less than or equal to η recorded on the ordinate axis, i.e., it is $\sum_{i=1}^I \pi_i \check{m}_i(\check{\theta}(\eta))u(c_i)$ where $\check{m}_i(\check{\theta}(c,\eta)) \propto \exp\left(\frac{u(c_i)}{\check{\theta}}\right)$.

(Here $\check{\theta}$ is the Lagrange multiplier associated with max-max expected utility.)

Points between these two curves are possible values of expected utility for some distribution with entropy less than or equal to the value η on the ordinate axis.

The straight line depicts the right side of inequality (25.19) for a particular value of the penalty parameter θ.

As noted, when one lowers θ, the intercept $\mathsf{T}_\theta u(c)$ and the absolute value of the slope both decrease.

Thus, as θ is lowered, $\mathsf{T}_\theta u(c)$ becomes a more conservative estimate of expected utility under the approximating model π.

However, as θ is lowered, the robustness bound (25.19) becomes more informative for sufficiently large values of entropy.

The slope of straight line depicting a bound is $-\theta$ and the projection of the point of tangency with the curved depicting the lower bound of expected utility is the entropy associated with that θ when it is interpreted as a Lagrange multiplier on the entropy constraint in the constraint problem .

This is an application of the envelope theorem.

25.15 Why entropy?

Beyond the helpful mathematical fact that it leads directly to convenient exponential twisting formulas (25.6) and (25.12) for worst-case probability distortions, there are two related justifications for using entropy to measure discrepancies between probability distribution.

One arises from the role of entropy in statistical tests for discriminating between models.

The other comes from axioms.

25.15.1 Entropy and statistical detection

Robust control theory starts with a decision maker who has constructed a good baseline approximating model whose free parameters he has estimated to fit historical data well.

The decision maker recognizes that actual outcomes might be generated by one of a vast number of other models that fit the historical data nearly as well as his.

Therefore, he wants to evaluate outcomes under a set of alternative models that are plausible in the sense of being statistically close to his model.

He uses relative entropy to quantify what close means.

[Anderson *et al.*, 2003] and [Barillas *et al.*, 2009]describe links between entropy and large deviations bounds on test statistics for discriminating between models, in particular, statistics that describe the probability of making an error in applying a likelihood ratio test to decide whether model A or model B generated a data record of length T.

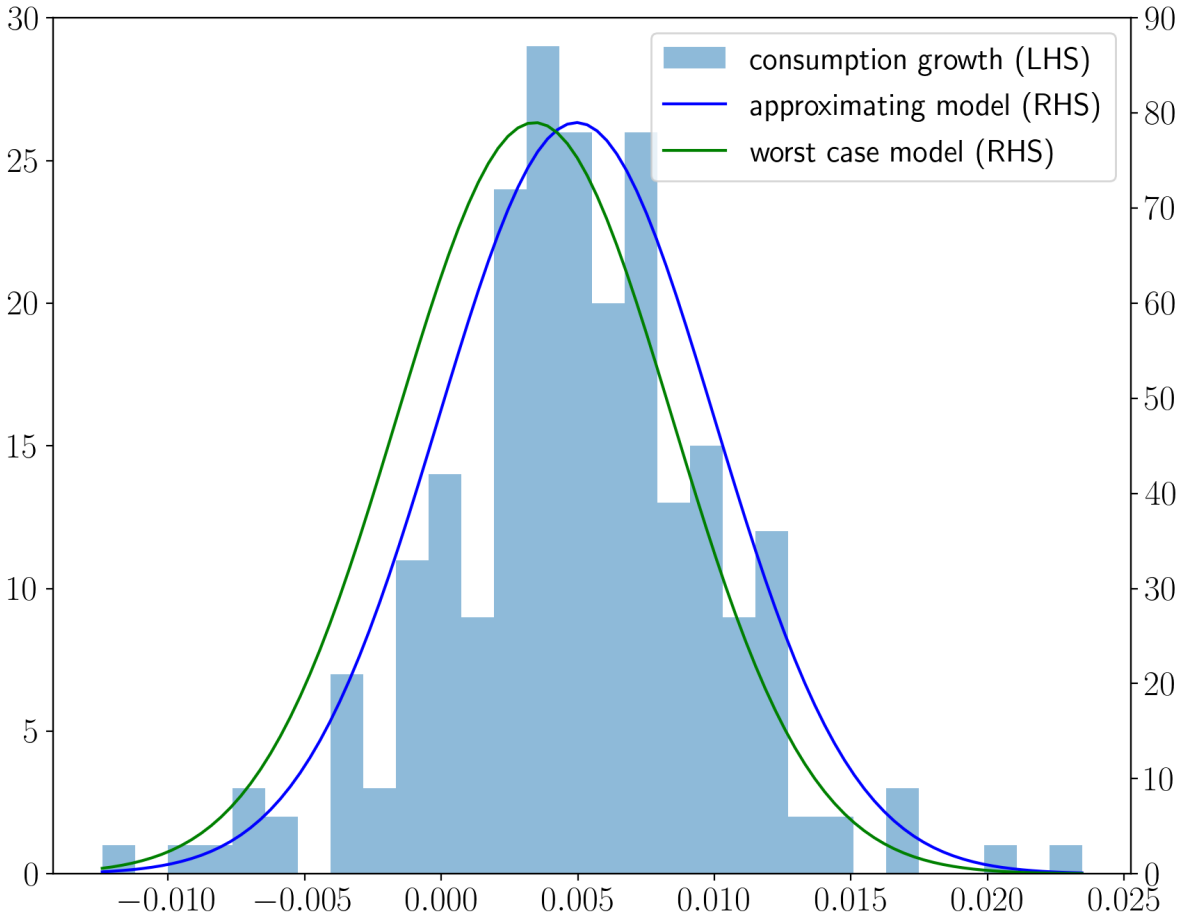
For a given sample size, an informative bound on the detection error probability is a function of the entropy parameter η in constraint preferences. [Anderson *et al.*, 2003] and [Barillas *et al.*, 2009] use detection error probabilities to calibrate reasonable values of η.

[Anderson *et al.*, 2003] and [Hansen and Sargent, 2008] also use detection error probabilities to calibrate reasonable values of the penalty parameter θ in multiplier preferences.

For a fixed sample size and a fixed θ, they would calculate the worst-case $\hat{m}_i(\theta)$, an associated entropy $\eta(\theta)$, and an associated detection error probability. In this way they build up a detection error probability as a function of θ.

They then invert this function to calibrate θ to deliver a reasonable detection error probability.

To indicate outcomes from this approach, the following figure plots the histogram for U.S. quarterly consumption growth along with a representative agent's approximating density and a worst-case density that [Barillas *et al.*, 2009] show imply high measured market prices of risk even when a representative consumer has the unit coefficient of relative risk aversion associated with a logarithmic one-period utility function.



The density for the approximating model is $\log c_{t+1} - \log c_t = \mu + \sigma_c \epsilon_{t+1}$ where $\epsilon_{t+1} \sim N(0,1)$ and μ and σ_c are estimated by maximum likelihood from the U.S. quarterly data in the histogram over the period 1948.I-2006.IV.

The consumer's value function under logarithmic utility implies that the worst-case model is $\log c_{t+1} - \log c_t = (\mu + \sigma_c w) + \sigma_c \tilde{\epsilon}_{t+1}$ where $\{\tilde{\epsilon}_{t+1}\}$ is also a normalized Gaussian random sequence and where w is calculated by setting a detection error probability to .05.

The worst-case model appears to fit the histogram nearly as well as the approximating model.

25.15.2 Axiomatic justifications

Multiplier and constraint preferences are both special cases of what [Maccheroni *et al.*, 2006] call variational preferences.

They provide an axiomatic foundation for variational preferences and describe how they express ambiguity aversion.

Constraint preferences are particular instances of the multiple priors model of [Gilboa and Schmeidler, 1989].

CHAPTER

TWENTYSIX

ETYMOLOGY OF ENTROPY

This lecture describes and compares several notions of entropy.

Among the senses of entropy, we'll encounter these

- A measure of **uncertainty** of a random variable advanced by Claude Shannon [Shannon and Weaver, 1949]
- A key object governing thermodynamics
- Kullback and Leibler's measure of the statistical divergence between two probability distributions
- A measure of the volatility of stochastic discount factors that appear in asset pricing theory
- Measures of unpredictability that occur in classical Wiener-Kolmogorov linear prediction theory
- A frequency domain criterion for constructing robust decision rules

The concept of entropy plays an important role in robust control formulations described in this lecture Risk and Model Uncertainty and in this lecture Robustness.

26.1 Information Theory

In information theory [Shannon and Weaver, 1949], entropy is a measure of the unpredictability of a random variable.

To illustrate things, let X be a discrete random variable taking values $x_1, \ldots, x_n$ with probabilities $p_i = \text{Prob}(X = x_i) \geq 0, \sum_i p_i = 1$.

Claude Shannon's [Shannon and Weaver, 1949] definition of entropy is

$$H(p) = \sum_i p_i \log_b(p_i^{-1}) = -\sum_i p_i \log_b(p_i). \tag{26.1}$$

where $\log_b$ denotes the log function with base b.

Inspired by the limit

$$\lim_{p \downarrow 0} p \log p = \lim_{p \downarrow 0} \frac{\log p}{p^{-1}} = \lim_{p \downarrow 0} p = 0,$$

we set $p \log p = 0$ in equation (26.1).

Typical bases for the logarithm are 2, e, and 10.

In the information theory literature, logarithms of base 2, e, and 10 are associated with units of information called bits, nats, and dits, respectively.

Shannon typically used base 2.

26.2 A Measure of Unpredictability

For a discrete random variable X with probability density $p = \{p_i\}_{i=1}^n$, the **surprisal** for state i is $s_i = \log\left(\frac{1}{p_i}\right)$.

The quantity $\log\left(\frac{1}{p_i}\right)$ is called the **surprisal** because it is inversely related to the likelihood that state i will occur.

Note that entropy $H(p)$ equals the **expected surprisal**

$$H(p) = \sum_i p_i s_i = \sum_i p_i \log\left(\frac{1}{p_i}\right) = -\sum_i p_i \log\left(p_i\right).$$

26.2.1 Example

Take a possibly unfair coin, so $X = \{0, 1\}$ with $p = \text{Prob}(X = 1) = p \in [0, 1]$.

Then

$$H(p) = -(1-p)\log(1-p) - p\log p.$$

Evidently,

$$H'(p) = \log(1-p) - \log p = 0$$

at $p = .5$ and $H''(p) = -\frac{1}{1-p} - \frac{1}{p} < 0$ for $p \in (0, 1)$.

So $p = .5$ maximizes entropy, while entropy is minimized at $p = 0$ and $p = 1$.

Thus, among all coins, a fair coin is the most unpredictable.

See Fig. 26.1

26.2.2 Example

Take an n-sided possibly unfair die with a probability distribution $\{p_i\}_{i=1}^n$. The die is fair if $p_i = \frac{1}{n} \forall i$.

Among all dies, a fair die maximizes entropy.

For a fair die, entropy equals $H(p) = -n^{-1}\sum_i \log\left(\frac{1}{n}\right) = \log(n)$.

To specify the expected number of bits needed to isolate the outcome of one roll of a fair n-sided die requires $\log_2(n)$ bits of information.

For example, if $n = 2$, $\log_2(2) = 1$.

For $n = 3$, $\log_2(3) = 1.585$.

26.3 Mathematical Properties of Entropy

For a discrete random variable with probability vector p, entropy $H(p)$ is a function that satisfies

- H is *continuous*.
- H is *symmetric*: $H(p_1, p_2, \ldots, p_n) = H(p_{r_1}, \ldots, p_{r_n})$ for any permutation $r_1, \ldots, r_n$ of $1, \ldots, n$.
- A uniform distribution maximizes $H(p)$: $H(p_1, \ldots, p_n) \leq H(\frac{1}{n}, \ldots, \frac{1}{n})$.
- Maximum entropy increases with the number of states: $H(\frac{1}{n}, \ldots, \frac{1}{n}) \leq H(\frac{1}{n+1}, \ldots, \frac{1}{n+1})$.
- Entropy is not affected by events zero probability.

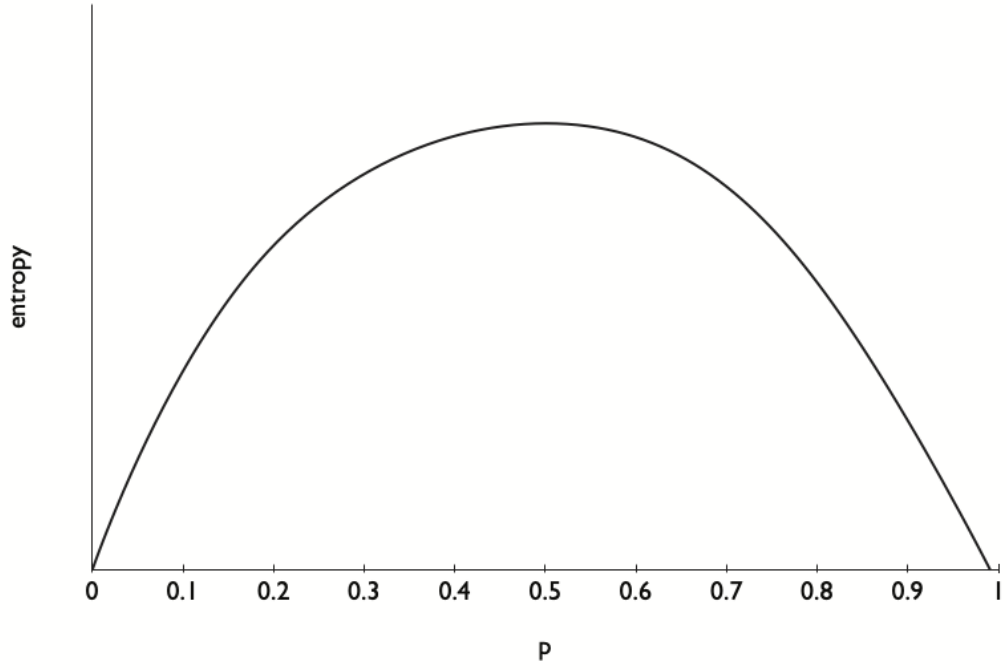


Fig. 26.1: Entropy as a function of $\hat{\pi}_1$ when $\pi_1 = .5$.

26.4 Conditional Entropy

Let (X, Y) be a bivariate discrete random vector with outcomes $x_1, \ldots, x_n$ and $y_1, \ldots, y_m$, respectively, occurring with probability density $p(x_i, y_i)$.

Conditional entropy $H(X|Y)$ is defined as

$$H(X|Y) = \sum_{i,j} p(x_i, y_j) \log \frac{p(y_j)}{p(x_i, y_j)}. \tag{26.2}$$

Here $\frac{p(y_j)}{p(x_i, y_j)}$, the reciprocal of the conditional probability of x_i given y_j, can be defined as the **conditional surprisal**.

26.5 Independence as Maximum Conditional Entropy

Let $m = n$ and $[x_1, \ldots, x_n] = [y_1, \ldots, y_n]$.

Let $\sum_j p(x_i, y_j) = \sum_j p(x_j, y_i)$ for all i, so that the marginal distributions of x and y are identical.

Thus, x and y are identically distributed, but they are not necessarily independent.

Consider the following problem: choose a joint distribution $p(x_i, y_j)$ to maximize conditional entropy (26.2) subject to the restriction that x and y are identically distributed.

The conditional-entropy-maximizing $p(x_i, y_j)$ sets

$$\frac{p(x_i, y_j)}{p(y_j)} = \sum_j p(x_i, y_j) = p(x_i) \forall i.$$

Thus, among all joint distributions with identical marginal distributions, the conditional entropy maximizing joint distribution makes x and y be independent.

26.6 Thermodynamics

Josiah Willard Gibbs (see https://en.wikipedia.org/wiki/Josiah_Willard_Gibbs) defined entropy as

$$S = -k_B \sum_i p_i \log p_i \tag{26.3}$$

where p_i is the probability of a micro state and k_B is Boltzmann's constant.

- The Boltzmann constant k_b relates energy at the micro particle level with the temperature observed at the macro level. It equals what is called a gas constant divided by an Avogadro constant.

The second law of thermodynamics states that the entropy of a closed physical system increases until S defined in (26.3) attains a maximum.

26.7 Statistical Divergence

Let X be a discrete state space $x_1, \ldots, x_n$ and let p and q be two discrete probability distributions on X.

Assume that $\frac{p_i}{q_t} \in (0, \infty)$ for all i for which $p_i > 0$.

Then the Kullback-Leibler statistical divergence, also called **relative entropy**, is defined as

$$D(p|q) = \sum_i p_i \log\left(\frac{p_i}{q_i}\right) = \sum_i q_i \left(\frac{p_i}{q_i}\right) \log\left(\frac{p_i}{q_i}\right). \tag{26.4}$$

Evidently,

$$\begin{aligned} D(p|q) &= -\sum_i p_i \log q_i + \sum_i p_i \log p_i \\ &= H(p, q) - H(p), \end{aligned}$$

where $H(p, q) = \sum_i p_i \log q_i$ is the cross-entropy.

It is easy to verify, as we have done above, that $D(p|q) \geq 0$ and that $D(p|q) = 0$ implies that $p_i = q_i$ when $q_i > 0$.

26.8 Continuous distributions

For a continuous random variable, Kullback-Leibler divergence between two densities p and q is defined as

$$D(p|q) = \int p(x) \log\left(\frac{p(x)}{q(x)}\right) d\,x.$$

26.9 Relative entropy and Gaussian distributions

We want to compute relative entropy for two continuous densities ϕ and $\hat{\phi}$ when ϕ is $N(0, I)$ and $\hat{\phi}$ is $N(w, \Sigma)$, where the covariance matrix Σ is nonsingular.

We seek a formula for

$$\text{ent} = \int (\log \hat{\phi}(\varepsilon) - \log \phi(\varepsilon)) \hat{\phi}(\varepsilon) d\varepsilon.$$

Claim

$$\text{ent} = -\frac{1}{2} \log \det \Sigma + \frac{1}{2} w'w + \frac{1}{2} \text{trace}(\Sigma - I). \tag{26.5}$$

Proof

The log likelihood ratio is

$$\log \hat{\phi}(\varepsilon) - \log \phi(\varepsilon) = \frac{1}{2} \left[-(\varepsilon - w)' \Sigma^{-1} (\varepsilon - w) + \varepsilon'\varepsilon - \log \det \Sigma \right]. \tag{26.6}$$

Observe that

$$-\int \frac{1}{2} (\varepsilon - w)' \Sigma^{-1} (\varepsilon - w) \hat{\phi}(\varepsilon) d\varepsilon = -\frac{1}{2} \text{trace}(I).$$

Applying the identity $\varepsilon = w + (\varepsilon - w)$ gives

$$\frac{1}{2} \varepsilon'\varepsilon = \frac{1}{2} w'w + \frac{1}{2} (\varepsilon - w)'(\varepsilon - w) + w'(\varepsilon - w).$$

Taking mathematical expectations

$$\frac{1}{2} \int \varepsilon'\varepsilon \hat{\phi}(\varepsilon) d\varepsilon = \frac{1}{2} w'w + \frac{1}{2} \text{trace}(\Sigma).$$

Combining terms gives

$$\text{ent} = \int (\log \hat{\phi} - \log \phi) \hat{\phi} d\varepsilon = -\frac{1}{2} \log \det \Sigma + \frac{1}{2} w'w + \frac{1}{2} \text{trace}(\Sigma - I). \tag{26.7}$$

which agrees with equation (26.5). Notice the separate appearances of the mean distortion w and the covariance distortion $\Sigma - I$ in equation (26.7).

Extension

Let $N_0 = \mathcal{N}(\mu_0, \Sigma_0)$ and $N_1 = \mathcal{N}(\mu_1, \Sigma_1)$ be two multivariate Gaussian distributions.

Then

$$D(N_0|N_1) = \frac{1}{2} \left(\text{trace}(\Sigma_1^{-1} \Sigma_0) + (\mu_1 - \mu_0)' \Sigma_1^{-1} (\mu_1 - \mu_0) - \log \left(\frac{\det \Sigma_0}{\det \Sigma_1} \right) - k \right). \tag{26.8}$$

26.10 Von Neumann Entropy

Let P and Q be two positive-definite symmetric matrices.

A measure of the divergence between two P and Q is

$$D(P|Q) = \text{trace}(P \ln P - P \ln Q - P + Q)$$

where the log of a matrix is defined here (https://en.wikipedia.org/wiki/Logarithm_of_a_matrix).

A density matrix P from quantum mechanics is a positive definite matrix with trace 1.

The von Neumann entropy of a density matrix P is

$$S = -\text{trace}(P \ln P)$$

26.11 Backus-Chernov-Zin Entropy

After flipping signs, [Backus *et al.*, 2014] use Kullback-Leibler relative entropy as a measure of volatility of stochastic discount factors that they assert is useful for characterizing features of both the data and various theoretical models of stochastic discount factors.

Where p_{t+1} is the physical or true measure, p^*_{t+1} is the risk-neutral measure, and E_t denotes conditional expectation under the p_{t+1} measure, [Backus *et al.*, 2014] define entropy as

$$L_t(p^*_{t+1}/p_{t+1}) = -E_t \log(p^*_{t+1}/p_{t+1}). \tag{26.9}$$

Evidently, by virtue of the minus sign in equation (26.9),

$$L_t(p^*_{t+1}/p_{t+1}) = D_{KL,t}(p^*_{t+1}|p_{t+1}), \tag{26.10}$$

where $D_{KL,t}$ denotes conditional relative entropy.

Let m_{t+1} be a stochastic discount factor, r_{t+1} a gross one-period return on a risky security, and $(r^1_{t+1})^{-1} \equiv q^1_t = E_t m_{t+1}$ be the reciprocal of a risk-free one-period gross rate of return. Then

$$E_t(m_{t+1} r_{t+1}) = 1$$

[Backus *et al.*, 2014] note that a stochastic discount factor satisfies

$$m_{t+1} = q^1_t p^*_{t+1}/p_{t+1}.$$

They derive the following **entropy bound**

$$EL_t(m_{t+1}) \geq E(\log r_{t+1} - \log r^1_{t+1})$$

which they propose as a complement to a Hansen-Jagannathan [Hansen and Jagannathan, 1991] bound.

26.12 Wiener-Kolmogorov Prediction Error Formula as Entropy

Let $\{x_t\}_{t=-\infty}^{\infty}$ be a covariance stationary stochastic process with mean zero and spectral density $S_x(\omega)$.

The variance of x is

$$\sigma_x^2 = \left(\frac{1}{2\pi}\right) \int_{-\pi}^{\pi} S_x(\omega) d\omega.$$

As described in chapter XIV of [Sargent, 1987], the Wiener-Kolmogorov formula for the one-period ahead prediction error is

$$\sigma_\epsilon^2 = \exp\left[\left(\frac{1}{2\pi}\right) \int_{-\pi}^{\pi} \log S_x(\omega) d\omega\right]. \tag{26.11}$$

Occasionally the logarithm of the one-step-ahead prediction error σ_ϵ^2 is called entropy because it measures unpredictability.

Consider the following problem reminiscent of one described earlier.

Problem:

Among all covariance stationary univariate processes with unconditional variance σ_x^2, find a process with maximal one-step-ahead prediction error.

The maximizer is a process with spectral density

$$S_x(\omega) = 2\pi\sigma_x^2.$$

Thus, among all univariate covariance stationary processes with variance σ_x^2, a process with a flat spectral density is the most uncertain, in the sense of one-step-ahead prediction error variance.

This no-patterns-across-time outcome for a temporally dependent process resembles the no-pattern-across-states outcome for the static entropy maximizing coin or die in the classic information theoretic analysis described above.

26.13 Multivariate Processes

Let y_t be an $n \times 1$ covariance stationary stochastic process with mean 0 with matrix covariogram $C_y(j) = Ey_t y_{t-j}'$ and spectral density matrix

$$S_y(\omega) = \sum_{j=-\infty}^{\infty} e^{-i\omega j} C_y(j), \quad \omega \in [-\pi, \pi].$$

Let

$$y_t = D(L)\epsilon_t \equiv \sum_{j=0}^{\infty} D_j \epsilon_t$$

be a Wold representation for y, where $D(0)\epsilon_t$ is a vector of one-step-ahead errors in predicting y_t conditional on the infinite history $y^{t-1} = [y_{t-1}, y_{t-2}, ...]$ and ϵ_t is an $n \times 1$ vector of serially uncorrelated random disturbances with mean zero and identity contemporaneous covariance matrix $E\epsilon_t\epsilon_t' = I$.

Linear-least-squares predictors have one-step-ahead prediction error $D(0)D(0)'$ that satisfies

$$\log\det[D(0)D(0)'] = \left(\frac{1}{2\pi}\right)\int_{-\pi}^{\pi} \log\det[S_y(\omega)]d\omega. \tag{26.12}$$

Being a measure of the unpredictability of an $n \times 1$ vector covariance stationary stochastic process, the left side of (26.12) is sometimes called entropy.

26.14 Frequency Domain Robust Control

Chapter 8 of [Hansen and Sargent, 2008] adapts work in the control theory literature to define a **frequency domain entropy** criterion for robust control as

$$\int_\Gamma \log\det[\theta I - G_F(\zeta)'G_F(\zeta)]d\lambda(\zeta), \tag{26.13}$$

where $\theta \in (\underline{\theta}, +\infty)$ is a positive robustness parameter and $G_F(\zeta)$ is a ζ-transform of the objective function.

Hansen and Sargent [Hansen and Sargent, 2008] show that criterion (26.13) can be represented as

$$\log\det[D(0)'D(0)] = \int_\Gamma \log\det[\theta I - G_F(\zeta)'G_F(\zeta)]d\lambda(\zeta), \tag{26.14}$$

for an appropriate covariance stationary stochastic process derived from $\theta, G_F(\zeta)$.

This explains the moniker **maximum entropy** robust control for decision rules F designed to maximize criterion (26.13).

26.15 Relative Entropy for a Continuous Random Variable

Let x be a continuous random variable with density $\phi(x)$, and let $g(x)$ be a nonnegative random variable satisfying $\int g(x)\phi(x)dx = 1$.

The relative entropy of the distorted density $\widehat{\phi}(x) = g(x)\phi(x)$ is defined as

$$\text{ent}(g) = \int g(x)\log g(x)\phi(x)dx.$$

Fig. 26.2 plots the functions $g\log g$ and $g-1$ over the interval $g \geq 0$.

That relative entropy $\text{ent}(g) \geq 0$ can be established by noting (a) that $g\log g \geq g-1$ (see Fig. 26.2) and (b) that under ϕ, $Eg = 1$.

Fig. 26.3 and Fig. 26.4 display aspects of relative entropy visually for a continuous random variable x for two densities with likelihood ratio $g \geq 0$.

Where the numerator density is $\mathcal{N}(0,1)$, for two denominator Gaussian densities $\mathcal{N}(0,1.5)$ and $\mathcal{N}(0,.95)$, respectively, Fig. 26.3 and Fig. 26.4 display the functions $g\log g$ and $g-1$ as functions of x.

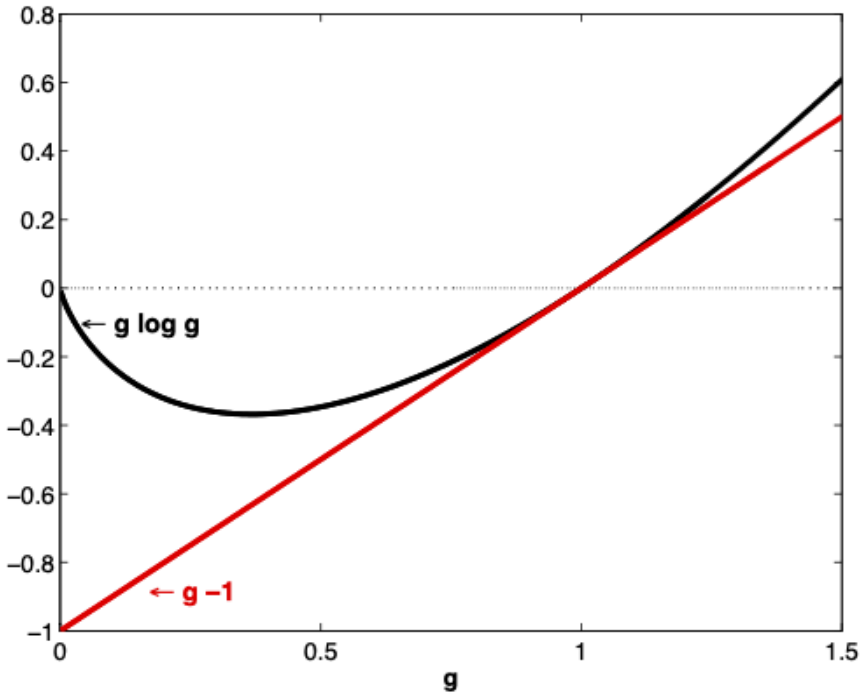


Fig. 26.2: The function $g\log g$ for $g \geq 0$. For a random variable g with $Eg = 1$, $Eg\log g \geq 0$.

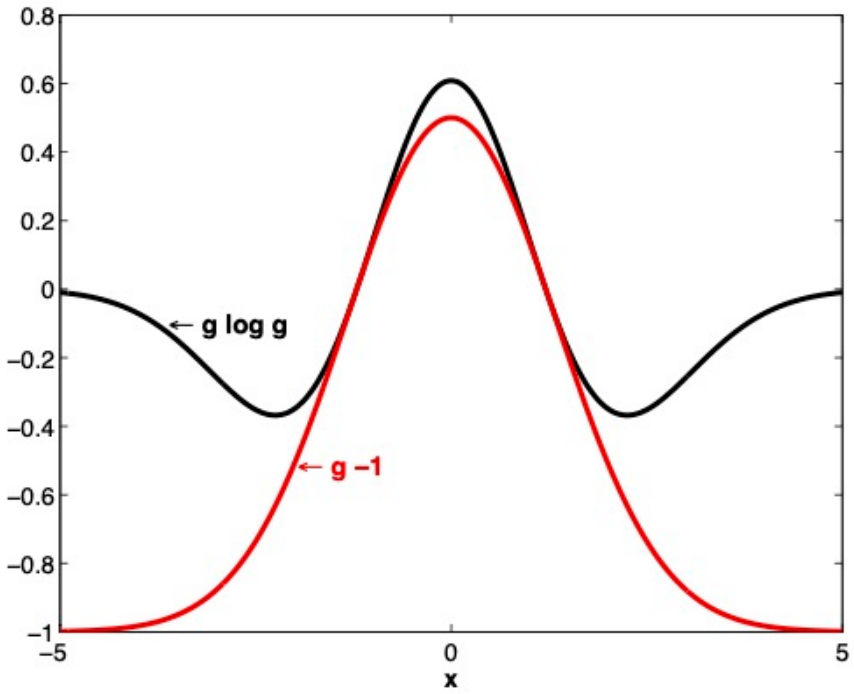


Fig. 26.3: Graphs of $g \log g$ and $g - 1$ where g is the ratio of the density of a $\mathcal{N}(0, 1)$ random variable to the density of a $\mathcal{N}(0, 1.5)$ random variable. Under the $\mathcal{N}(0, 1.5)$ density, $Eg = 1$.

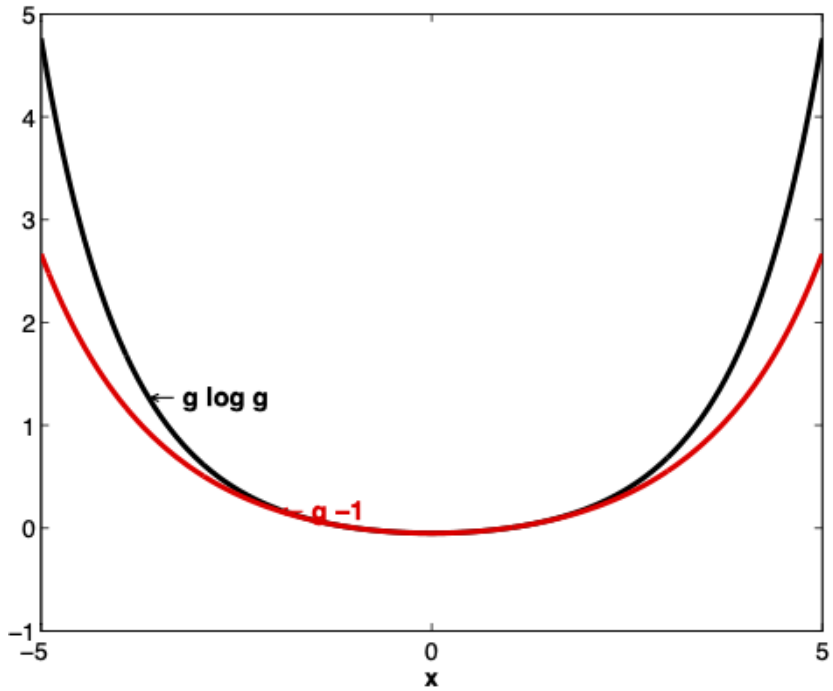


Fig. 26.4: $g \log g$ and $g - 1$ where g is the ratio of the density of a $\mathcal{N}(0, 1)$ random variable to the density of a $\mathcal{N}(0, 1.5)$ random variable. Under the $\mathcal{N}(0, 1.5)$ density, $Eg = 1$.

CHAPTER

TWENTYSEVEN

ROBUSTNESS

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install --upgrade quantecon
```

27.1 Overview

This lecture modifies a Bellman equation to express a decision-maker's doubts about transition dynamics.

His specification doubts make the decision-maker want a *robust* decision rule.

Robust means insensitive to misspecification of transition dynamics.

The decision-maker has a single *approximating model* of the transition dynamics.

He calls it *approximating* to acknowledge that he doesn't completely trust it.

He fears that transition dynamics are actually determined by another model that he cannot describe explicitly.

All that he knows is that the actual data-generating model is in some (uncountable) set of models that surrounds his approximating model.

He quantifies the discrepancy between his approximating model and the genuine data-generating model by using a quantity called *entropy*.

(We'll explain what entropy means below)

He wants a decision rule that will work well enough no matter which of those other models actually governs outcomes.

This is what it means for his decision rule to be "robust to misspecification of an approximating model".

This may sound like too much to ask for, but ....

... a *secret weapon* is available to design robust decision rules.

The secret weapon is max-min control theory.

A value-maximizing decision-maker enlists the aid of an (imaginary) value-minimizing model chooser to construct *bounds* on the value attained by a given decision rule under different models of the transition dynamics.

The original decision-maker uses those bounds to construct a decision rule with an assured performance level, no matter which model actually governs outcomes.

Note: In reading this lecture, please don't think that our decision-maker is paranoid when he conducts a worst-case analysis. By designing a rule that works well against a worst-case, his intention is to construct a rule that will work well across a *set* of models.

Let's start with some imports:

```
import pandas as pd
import numpy as np
from scipy.linalg import eig
import matplotlib.pyplot as plt
import quantecon as qe
```

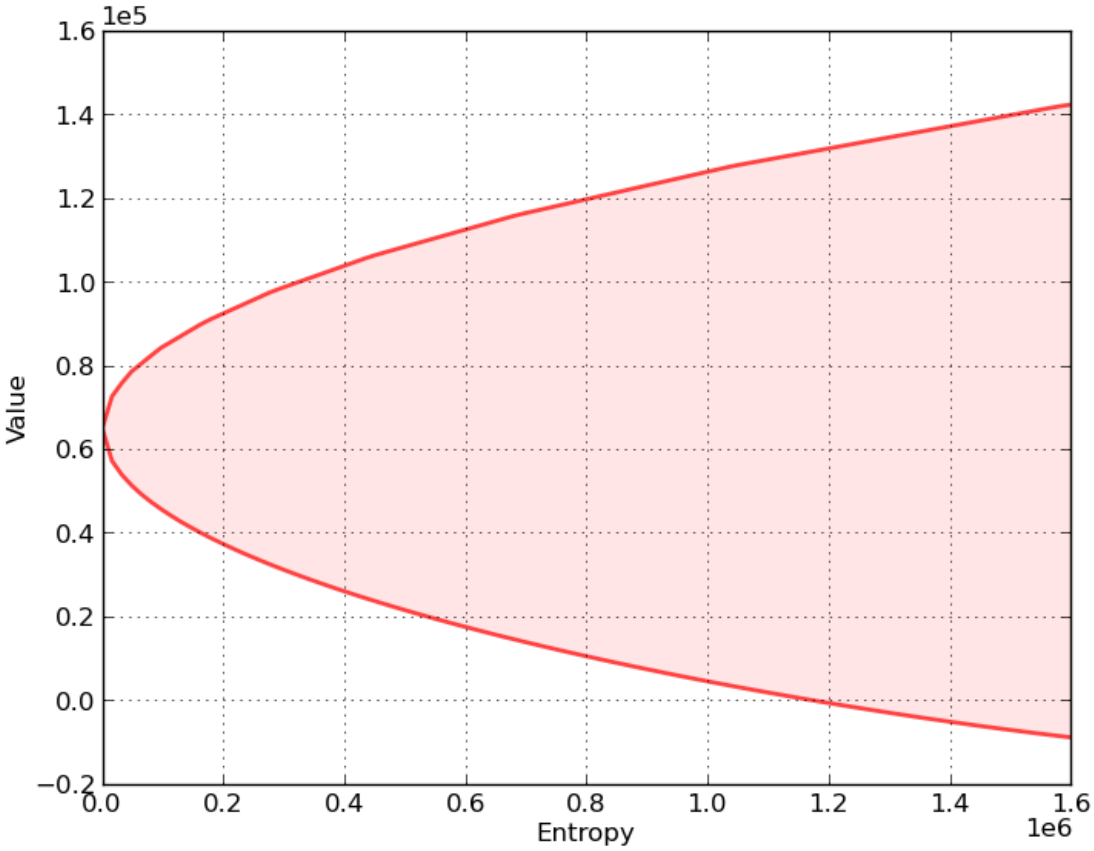
27.1.1 Sets of Models Imply Sets Of Values

Our "robust" decision-maker wants to know how well a given rule will work when he does not *know* a single transition law ....

... he wants to know *sets* of values that will be attained by a given decision rule F under a *set* of transition laws.

Ultimately, he wants to design a decision rule F that shapes the *set* of values in ways that he prefers.

With this in mind, consider the following graph, which relates to a particular decision problem to be explained below



The figure shows a *value-entropy correspondence* for a particular decision rule F.

The shaded set is the graph of the correspondence, which maps entropy to a set of values associated with a set of models that surround the decision-maker's approximating model.

Here

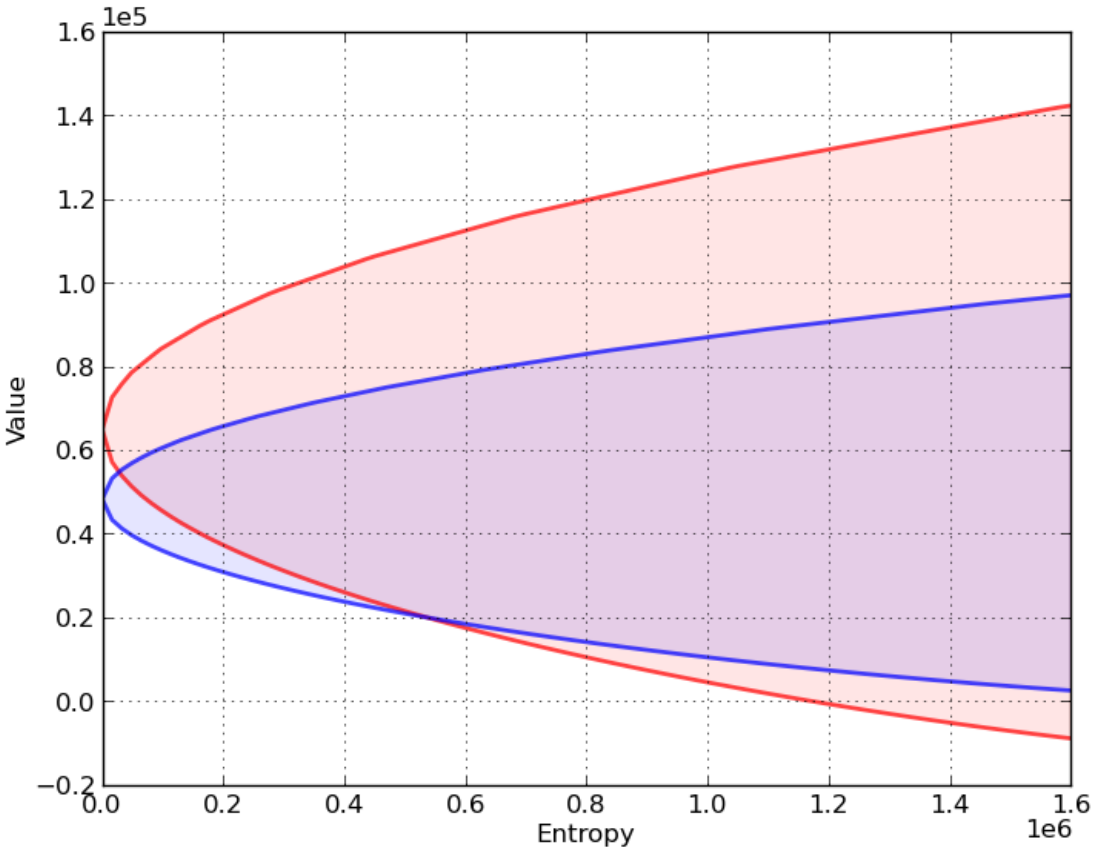
- *Value* refers to a sum of discounted rewards obtained by applying the decision rule F when the state starts at some fixed initial state x_0.
- *Entropy* is a non-negative number that measures the size of a set of models surrounding the decision-maker's approximating model.
  - Entropy is zero when the set includes only the approximating model, indicating that the decision-maker completely trusts the approximating model.
  - Entropy is bigger, and the set of surrounding models is bigger, the less the decision-maker trusts the approximating model of the transition dynamics.

The shaded region indicates that for **all** models having entropy less than or equal to the number on the horizontal axis, the value obtained will be somewhere within the indicated set of values.

Now let's compare sets of values associated with two different decision rules, F_r and F_b.

In the next figure,

- The red set shows the value-entropy correspondence for decision rule F_r.
- The blue set shows the value-entropy correspondence for decision rule F_b.



The blue correspondence is skinnier than the red correspondence.

This conveys the sense in which the decision rule F_b is *more robust* than the decision rule F_r

- *more robust* means that the set of values is less sensitive to *increasing misspecification* as measured by entropy

Notice that the less robust rule F_r promises higher values for small misspecifications (small entropy).

(But it is more fragile in the sense that it is more sensitive to perturbations of the approximating model)

Below we'll explain in detail how to construct these sets of values for a given F, but for now ....

Here is a hint about the *secret weapons* we'll use to construct these sets

- We'll use some min problems to construct the lower bounds
- We'll use some max problems to construct the upper bounds

We will also describe how to choose F to shape the sets of values.

This will involve crafting a *skinnier* set at the cost of a lower *level* (at least for low values of entropy).

27.1.2 Inspiring Video

If you want to understand more about why one serious quantitative researcher is interested in this approach, we recommend Lars Peter Hansen's Nobel lecture.

27.1.3 Other References

Our discussion in this lecture is based on

- [Hansen and Sargent, 2000]
- [Hansen and Sargent, 2008]

27.2 The Model

For simplicity, we present ideas in the context of a class of problems with linear transition laws and quadratic objective functions.

To fit in with our earlier lecture on LQ control, we will treat loss minimization rather than value maximization.

To begin, recall the infinite horizon LQ problem, where an agent chooses a sequence of controls $\{u_t\}$ to minimize

$$\sum_{t=0}^{\infty} \beta^t \{x_t' R x_t + u_t' Q u_t\} \tag{27.1}$$

subject to the linear law of motion

$$x_{t+1} = Ax_t + Bu_t + Cw_{t+1}, \qquad t = 0, 1, 2, ... \tag{27.2}$$

As before,

- 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 $n \times j$
- R is $n \times n$ and Q is $k \times k$

Here x_t is the state, u_t is the control, and w_t is a shock vector.

For now, we take $\{w_t\} := \{w_t\}_{t=1}^{\infty}$ to be deterministic — a single fixed sequence.

We also allow for *model uncertainty* on the part of the agent solving this optimization problem.

In particular, the agent takes $w_t = 0$ for all $t \geq 0$ as a benchmark model but admits the possibility that this model might be wrong.

As a consequence, she also considers a set of alternative models expressed in terms of sequences $\{w_t\}$ that are more or less "close" to the zero sequence.

She seeks a policy that will do well enough for a set of alternative models whose members are pinned down by sequences $\{w_t\}$.

A sequence $\{w_t\}$ might represent

- nonlinearities absent from the approximating model
- time variations in parameters of the approximating model
- omitted state variables in the approximating model
- neglected history dependencies ...
- and other potential sources of misspecification

Soon we'll quantify the quality of a model specification in terms of the maximal size of the discounted sum $\sum_{t=0}^{\infty} \beta^{t+1} w'_{t+1} w_{t+1}$.

27.3 Constructing More Robust Policies

If our agent takes $\{w_t\}$ as a given deterministic sequence, then, drawing on ideas in earlier lectures on dynamic programming, we can anticipate Bellman equations such as

$$J_{t-1}(x) = \min_u \{x' R x + u' Q u + \beta J_t(Ax + Bu + Cw)\}$$

(Here J depends on t because the sequence $\{w_t\}$ is not recursive)

Our tool for studying robustness is to construct a rule that works well even if an adverse sequence $\{w_t\}$ occurs.

In our framework, "adverse" means "loss increasing".

As we'll see, this will eventually lead us to construct a Bellman equation

$$J(x) = \min_u \max_w \{x' R x + u' Q u + \beta \left[J(Ax + Bu + Cw) - \theta w' w\right]\} \tag{27.3}$$

Notice that we've added the penalty term $-\theta w' w$.

Since $w' w = \|w\|^2$, this term becomes influential when w moves away from the origin.

The penalty parameter θ controls how much we penalize the maximizing agent for "harming" the minimizing agent.

By raising θ more and more, we more and more limit the ability of maximizing agent to distort outcomes relative to the approximating model.

So bigger θ is implicitly associated with smaller distortion sequences $\{w_t\}$.

27.3.1 Analyzing the Bellman Equation

So what does J in (27.3) look like?

As with the ordinary LQ control model, J takes the form $J(x) = x'Px$ for some symmetric positive definite matrix P.

One of our main tasks will be to analyze and compute the matrix P.

Related tasks will be to study associated feedback rules for u_t and w_{t+1}.

First, using matrix calculus, you will be able to verify that

$$\begin{aligned} \max_w \{(Ax + Bu + Cw)'P(Ax + Bu + Cw) - \theta w'w\} \\ = (Ax + Bu)'\mathcal{D}(P)(Ax + Bu) \end{aligned} \tag{27.4}$$

where

$$\mathcal{D}(P) := P + PC(\theta I - C'PC)^{-1}C'P \tag{27.5}$$

and I is a $j \times j$ identity matrix. Substituting this expression for the maximum into (27.3) yields

$$x'Px = \min_u \{x'Rx + u'Qu + \beta\,(Ax + Bu)'\mathcal{D}(P)(Ax + Bu)\} \tag{27.6}$$

Using similar mathematics, the solution to this minimization problem is $u = -Fx$ where $F := (Q + \beta B'\mathcal{D}(P)B)^{-1}\beta B'\mathcal{D}(P)A$.

Substituting this minimizer back into (27.6) and working through the algebra gives $x'Px = x'\mathcal{B}(\mathcal{D}(P))x$ for all x, or, equivalently,

$$P = \mathcal{B}(\mathcal{D}(P))$$

where $\mathcal{D}$ is the operator defined in (27.5) and

$$\mathcal{B}(P) := R - \beta^2 A'PB(Q + \beta B'PB)^{-1}B'PA + \beta A'PA$$

The operator $\mathcal{B}$ is the standard (i.e., non-robust) LQ Bellman operator, and $P = \mathcal{B}(P)$ is the standard matrix Riccati equation coming from the Bellman equation — see this discussion.

Under some regularity conditions (see [Hansen and Sargent, 2008]), the operator $\mathcal{B} \circ \mathcal{D}$ has a unique positive definite fixed point, which we denote below by $\hat{P}$.

A robust policy, indexed by θ, is $u = -\hat{F}x$ where

$$\hat{F} := (Q + \beta B'\mathcal{D}(\hat{P})B)^{-1}\beta B'\mathcal{D}(\hat{P})A \tag{27.7}$$

We also define

$$\hat{K} := (\theta I - C'\hat{P}C)^{-1}C'\hat{P}(A - B\hat{F}) \tag{27.8}$$

The interpretation of $\hat{K}$ is that $w_{t+1} = \hat{K}x_t$ on the worst-case path of $\{x_t\}$, in the sense that this vector is the maximizer of (27.4) evaluated at the fixed rule $u = -\hat{F}x$.

Note that $\hat{P}, \hat{F}, \hat{K}$ are all determined by the primitives and θ.

Note also that if θ is very large, then $\mathcal{D}$ is approximately equal to the identity mapping.

Hence, when θ is large, $\hat{P}$ and $\hat{F}$ are approximately equal to their standard LQ values.

Furthermore, when θ is large, $\hat{K}$ is approximately equal to zero.

Conversely, smaller θ is associated with greater fear of model misspecification and greater concern for robustness.

27.4 Robustness as Outcome of a Two-Person Zero-Sum Game

What we have done above can be interpreted in terms of a two-person zero-sum game in which $\hat{F}, \hat{K}$ are Nash equilibrium objects.

Agent 1 is our original agent, who seeks to minimize loss in the LQ program while admitting the possibility of misspecification.

Agent 2 is an imaginary malevolent player.

Agent 2's malevolence helps the original agent to compute bounds on his value function across a set of models.

We begin with agent 2's problem.

27.4.1 Agent 2's Problem

Agent 2

1. knows a fixed policy F specifying the behavior of agent 1, in the sense that $u_t = -Fx_t$ for all t
2. responds by choosing a shock sequence $\{w_t\}$ from a set of paths sufficiently close to the benchmark sequence $\{0, 0, 0, ...\}$

A natural way to say "sufficiently close to the zero sequence" is to restrict the summed inner product $\sum_{t=1}^{\infty} w_t' w_t$ to be small.

However, to obtain a time-invariant recursive formulation, it turns out to be convenient to restrict a discounted inner product

$$\sum_{t=1}^{\infty} \beta^t w_t' w_t \leq \eta \tag{27.9}$$

Now let F be a fixed policy, and let $J_F(x_0, \mathbf{w})$ be the present-value cost of that policy given sequence $\mathbf{w} := \{w_t\}$ and initial condition $x_0 \in \mathbb{R}^n$.

Substituting $-Fx_t$ for u_t in (27.1), this value can be written as

$$J_F(x_0, \mathbf{w}) := \sum_{t=0}^{\infty} \beta^t x_t' (R + F'QF) x_t \tag{27.10}$$

where

$$x_{t+1} = (A - BF)x_t + Cw_{t+1} \tag{27.11}$$

and the initial condition x_0 is as specified in the left side of (27.10).

Agent 2 chooses $\mathbf{w}$ to maximize agent 1's loss $J_F(x_0, \mathbf{w})$ subject to (27.9).

Using a Lagrangian formulation, we can express this problem as

$$\max_{\mathbf{w}} \sum_{t=0}^{\infty} \beta^t \left\{ x_t' (R + F'QF) x_t - \beta\theta (w_{t+1}' w_{t+1} - \eta) \right\}$$

where $\{x_t\}$ satisfied (27.11) and θ is a Lagrange multiplier on constraint (27.9).

For the moment, let's take θ as fixed, allowing us to drop the constant $\beta\theta\eta$ term in the objective function, and hence write the problem as

$$\max_{\mathbf{w}} \sum_{t=0}^{\infty} \beta^t \left\{ x_t' (R + F'QF) x_t - \beta\theta w_{t+1}' w_{t+1} \right\}$$

or, equivalently,

$$\min_{\mathbf{w}} \sum_{t=0}^{\infty} \beta^t \left\{ -x_t'(R + F'QF)x_t + \beta\theta w_{t+1}' w_{t+1} \right\} \tag{27.12}$$

subject to (27.11).

What's striking about this optimization problem is that it is once again an LQ discounted dynamic programming problem, with $\mathbf{w} = \{w_t\}$ as the sequence of controls.

The expression for the optimal policy can be found by applying the usual LQ formula (see here).

We denote it by $K(F, \theta)$, with the interpretation $w_{t+1} = K(F, \theta)x_t$.

The remaining step for agent 2's problem is to set θ to enforce the constraint (27.9), which can be done by choosing $\theta = \theta_\eta$ such that

$$\beta \sum_{t=0}^{\infty} \beta^t x_t' K(F, \theta_\eta)' K(F, \theta_\eta) x_t = \eta \tag{27.13}$$

Here x_t is given by (27.11) — which in this case becomes $x_{t+1} = (A - BF + CK(F, \theta))x_t$.

27.4.2 Using Agent 2's Problem to Construct Bounds on Value Sets

The Lower Bound

Define the minimized object on the right side of problem (27.12) as $R_\theta(x_0, F)$.

Because "minimizers minimize" we have

$$R_\theta(x_0, F) \leq \sum_{t=0}^{\infty} \beta^t \left\{ -x_t'(R + F'QF)x_t \right\} + \beta\theta \sum_{t=0}^{\infty} \beta^t w_{t+1}' w_{t+1},$$

where $x_{t+1} = (A - BF + CK(F, \theta))x_t$ and x_0 is a given initial condition.

This inequality in turn implies the inequality

$$R_\theta(x_0, F) - \theta \text{ ent} \leq \sum_{t=0}^{\infty} \beta^t \left\{ -x_t'(R + F'QF)x_t \right\} \tag{27.14}$$

where

$$\text{ent} := \beta \sum_{t=0}^{\infty} \beta^t w_{t+1}' w_{t+1}$$

The left side of inequality (27.14) is a straight line with slope $-\theta$.

Technically, it is a "separating hyperplane".

At a particular value of entropy, the line is tangent to the lower bound of values as a function of entropy.

In particular, the lower bound on the left side of (27.14) is attained when

$$\text{ent} = \beta \sum_{t=0}^{\infty} \beta^t x_t' K(F, \theta)' K(F, \theta) x_t \tag{27.15}$$

To construct the *lower bound* on the set of values associated with all perturbations $\mathbf{w}$ satisfying the entropy constraint (27.9) at a given entropy level, we proceed as follows:

- For a given θ, solve the minimization problem (27.12).
- Compute the minimizer $R_\theta(x_0, F)$ and the associated entropy using (27.15).
- Compute the lower bound on the value function $R_\theta(x_0, F) - \theta$ ent and plot it against ent.
- Repeat the preceding three steps for a range of values of θ to trace out the lower bound.

Note: This procedure sweeps out a set of separating hyperplanes indexed by different values for the Lagrange multiplier θ.

The Upper Bound

To construct an *upper bound* we use a very similar procedure.

We simply replace the *minimization* problem (27.12) with the *maximization* problem

$$V_{\tilde{\theta}}(x_0, F) = \max_{\mathbf{w}} \sum_{t=0}^{\infty} \beta^t \left\{ -x_t'(R + F'QF)x_t - \beta\tilde{\theta} w_{t+1}' w_{t+1} \right\} \tag{27.16}$$

where now $\tilde{\theta} > 0$ penalizes the choice of $\mathbf{w}$ with larger entropy.

(Notice that $\tilde{\theta} = -\theta$ in problem (27.12))

Because "maximizers maximize" we have

$$V_{\tilde{\theta}}(x_0, F) \geq \sum_{t=0}^{\infty} \beta^t \left\{ -x_t'(R + F'QF)x_t \right\} - \beta\tilde{\theta} \sum_{t=0}^{\infty} \beta^t w_{t+1}' w_{t+1}$$

which in turn implies the inequality

$$V_{\tilde{\theta}}(x_0, F) + \tilde{\theta} \text{ ent} \geq \sum_{t=0}^{\infty} \beta^t \left\{ -x_t'(R + F'QF)x_t \right\} \tag{27.17}$$

where

$$\text{ent} \equiv \beta \sum_{t=0}^{\infty} \beta^t w_{t+1}' w_{t+1}$$

The left side of inequality (27.17) is a straight line with slope $\tilde{\theta}$.

The upper bound on the left side of (27.17) is attained when

$$\text{ent} = \beta \sum_{t=0}^{\infty} \beta^t x_t' K(F, \tilde{\theta})' K(F, \tilde{\theta}) x_t \tag{27.18}$$

To construct the *upper bound* on the set of values associated all perturbations $\mathbf{w}$ with a given entropy we proceed much as we did for the lower bound

- For a given $\tilde{\theta}$, solve the maximization problem (27.16).
- Compute the maximizer $V_{\tilde{\theta}}(x_0, F)$ and the associated entropy using (27.18).
- Compute the upper bound on the value function $V_{\tilde{\theta}}(x_0, F) + \tilde{\theta}$ ent and plot it against ent.
- Repeat the preceding three steps for a range of values of $\tilde{\theta}$ to trace out the upper bound.

Reshaping the Set of Values

Now in the interest of *reshaping* these sets of values by choosing F, we turn to agent 1's problem.

27.4.3 Agent 1's Problem

Now we turn to agent 1, who solves

$$\min_{\{u_t\}} \sum_{t=0}^{\infty} \beta^t \left\{ x_t' R x_t + u_t' Q u_t - \beta \theta w_{t+1}' w_{t+1} \right\} \tag{27.19}$$

where $\{w_{t+1}\}$ satisfies $w_{t+1} = K x_t$.

In other words, agent 1 minimizes

$$\sum_{t=0}^{\infty} \beta^t \left\{ x_t' (R - \beta \theta K' K) x_t + u_t' Q u_t \right\} \tag{27.20}$$

subject to

$$x_{t+1} = (A + CK) x_t + B u_t \tag{27.21}$$

Once again, the expression for the optimal policy can be found here — we denote it by $\tilde{F}$.

27.4.4 Nash Equilibrium

Clearly, the $\tilde{F}$ we have obtained depends on K, which, in agent 2's problem, depended on an initial policy F.

Holding all other parameters fixed, we can represent this relationship as a mapping Φ, where

$$\tilde{F} = \Phi(K(F, \theta))$$

The map $F \mapsto \Phi(K(F, \theta))$ corresponds to a situation in which

1. agent 1 uses an arbitrary initial policy F
2. agent 2 best responds to agent 1 by choosing $K(F, \theta)$
3. agent 1 best responds to agent 2 by choosing $\tilde{F} = \Phi(K(F, \theta))$

As you may have already guessed, the robust policy $\hat{F}$ defined in (27.7) is a fixed point of the mapping Φ.

In particular, for any given θ,

1. $K(\hat{F}, \theta) = \hat{K}$, where $\hat{K}$ is as given in (27.8)
2. $\Phi(\hat{K}) = \hat{F}$

A sketch of the proof is given in *the appendix*.

27.5 The Stochastic Case

Now we turn to the stochastic case, where the sequence $\{w_t\}$ is treated as an IID sequence of random vectors.

In this setting, we suppose that our agent is uncertain about the *conditional probability distribution* of w_{t+1}.

The agent takes the standard normal distribution $N(0, I)$ as the baseline conditional distribution, while admitting the possibility that other "nearby" distributions prevail.

These alternative conditional distributions of w_{t+1} might depend nonlinearly on the history $x_s, s \leq t$.

To implement this idea, we need a notion of what it means for one distribution to be near another one.

Here we adopt a very useful measure of closeness for distributions known as the *relative entropy*, or Kullback-Leibler divergence.

For densities p, q, the Kullback-Leibler divergence of q from p is defined as

$$D_{KL}(p, q) := \int \ln \left[\frac{p(x)}{q(x)} \right] p(x) \, dx$$

Using this notation, we replace (27.3) with the stochastic analog

$$J(x) = \min_u \max_{\psi \in \mathcal{P}} \left\{ x' R x + u' Q u + \beta \left[\int J(Ax + Bu + Cw) \, \psi(dw) - \theta D_{KL}(\psi, \phi) \right] \right\} \tag{27.22}$$

Here $\mathcal{P}$ represents the set of all densities on $\mathbb{R}^n$ and ϕ is the benchmark distribution $N(0, I)$.

The distribution ϕ is chosen as the least desirable conditional distribution in terms of next period outcomes, while taking into account the penalty term $\theta D_{KL}(\psi, \phi)$.

This penalty term plays a role analogous to the one played by the deterministic penalty $\theta w' w$ in (27.3), since it discourages large deviations from the benchmark.

27.5.1 Solving the Model

The maximization problem in (27.22) appears highly nontrivial — after all, we are maximizing over an infinite dimensional space consisting of the entire set of densities.

However, it turns out that the solution is tractable, and in fact also falls within the class of normal distributions.

First, we note that J has the form $J(x) = x' P x + d$ for some positive definite matrix P and constant real number d.

Moreover, it turns out that if $(I - \theta^{-1} C' P C)^{-1}$ is nonsingular, then

$$\begin{aligned} \max_{\psi \in \mathcal{P}} \left\{ \int (Ax + Bu + Cw)' P (Ax + Bu + Cw) \, \psi(dw) - \theta D_{KL}(\psi, \phi) \right\} \\ = (Ax + Bu)' \mathcal{D}(P) (Ax + Bu) + \kappa(\theta, P) \end{aligned} \tag{27.23}$$

where

$$\kappa(\theta, P) := \theta \ln[\det(I - \theta^{-1} C' P C)^{-1}]$$

and the maximizer is the Gaussian distribution

$$\psi = N\left((\theta I - C' P C)^{-1} C' P (Ax + Bu), (I - \theta^{-1} C' P C)^{-1} \right) \tag{27.24}$$

Substituting the expression for the maximum into Bellman equation (27.22) and using $J(x) = x' P x + d$ gives

$$x' P x + d = \min_u \left\{ x' R x + u' Q u + \beta \, (Ax + Bu)' \mathcal{D}(P) (Ax + Bu) + \beta \left[d + \kappa(\theta, P) \right] \right\} \tag{27.25}$$

Since constant terms do not affect minimizers, the solution is the same as (27.6), leading to

$$x'Px + d = x'\mathcal{B}(\mathcal{D}(P))x + \beta\left[d + \kappa(\theta, P)\right]$$

To solve this Bellman equation, we take $\hat{P}$ to be the positive definite fixed point of $\mathcal{B} \circ \mathcal{D}$.

In addition, we take $\hat{d}$ as the real number solving $d = \beta\left[d + \kappa(\theta, P)\right]$, which is

$$\hat{d} := \frac{\beta}{1-\beta}\kappa(\theta, P) \tag{27.26}$$

The robust policy in this stochastic case is the minimizer in (27.25), which is once again $u = -\hat{F}x$ for $\hat{F}$ given by (27.7).

Substituting the robust policy into (27.24) we obtain the worst-case shock distribution:

$$w_{t+1} \sim N(\hat{K}x_t, (I - \theta^{-1}C'\hat{P}C)^{-1})$$

where $\hat{K}$ is given by (27.8).

Note that the mean of the worst-case shock distribution is equal to the same worst-case w_{t+1} as in the earlier deterministic setting.

27.5.2 Computing Other Quantities

Before turning to implementation, we briefly outline how to compute several other quantities of interest.

Worst-Case Value of a Policy

One thing we will be interested in doing is holding a policy fixed and computing the discounted loss associated with that policy.

So let F be a given policy and let $J_F(x)$ be the associated loss, which, by analogy with (27.22), satisfies

$$J_F(x) = \max_{\psi \in \mathcal{P}} \left\{ x'(R + F'QF)x + \beta \left[\int J_F((A - BF)x + Cw)\,\psi(dw) - \theta D_{KL}(\psi, \phi) \right] \right\}$$

Writing $J_F(x) = x'P_Fx + d_F$ and applying the same argument used to derive (27.23) we get

$$x'P_Fx + d_F = x'(R + F'QF)x + \beta\left[x'(A - BF)'\mathcal{D}(P_F)(A - BF)x + d_F + \kappa(\theta, P_F)\right]$$

To solve this we take P_F to be the fixed point

$$P_F = R + F'QF + \beta(A - BF)'\mathcal{D}(P_F)(A - BF)$$

and

$$d_F := \frac{\beta}{1-\beta}\kappa(\theta, P_F) = \frac{\beta}{1-\beta}\theta \ln[\det(I - \theta^{-1}C'P_FC)^{-1}] \tag{27.27}$$

If you skip ahead to *the appendix*, you will be able to verify that $-P_F$ is the solution to the Bellman equation in agent 2's problem *discussed above* — we use this in our computations.

27.6 Implementation

The QuantEcon.py package provides a class called `RBLQ` for implementation of robust LQ optimal control.

The code can be found on GitHub.

Here is a brief description of the methods of the class

- `d_operator()` and `b_operator()` implement $\mathcal{D}$ and $\mathcal{B}$ respectively
- `robust_rule()` and `robust_rule_simple()` both solve for the triple $\hat{F}, \hat{K}, \hat{P}$, as described in equations (27.7) – (27.8) and the surrounding discussion
  - `robust_rule()` is more efficient
  - `robust_rule_simple()` is more transparent and easier to follow
- `K_to_F()` and `F_to_K()` solve the decision problems of *agent 1* and *agent 2* respectively
- `compute_deterministic_entropy()` computes the left-hand side of (27.13)
- `evaluate_F()` computes the loss and entropy associated with a given policy — see *this discussion*

27.7 Application

Let us consider a monopolist similar to this one, but now facing model uncertainty.

The inverse demand function is $p_t = a_0 - a_1 y_t + d_t$.

where

$$d_{t+1} = \rho d_t + \sigma_d w_{t+1}, \quad \{w_t\} \overset{\text{IID}}{\sim} N(0,1)$$

and all parameters are strictly positive.

The period return function for the monopolist is

$$r_t = p_t y_t - \gamma \frac{(y_{t+1} - y_t)^2}{2} - c y_t$$

Its objective is to maximize expected discounted profits, or, equivalently, to minimize $\mathbb{E}\sum_{t=0}^{\infty} \beta^t (-r_t)$.

To form a linear regulator problem, we take the state and control to be

$$x_t = \begin{bmatrix} 1 \\ y_t \\ d_t \end{bmatrix} \quad \text{and} \quad u_t = y_{t+1} - y_t$$

Setting $b := (a_0 - c)/2$ we define

$$R = -\begin{bmatrix} 0 & b & 0 \\ b & -a_1 & 1/2 \\ 0 & 1/2 & 0 \end{bmatrix} \quad \text{and} \quad Q = \gamma/2$$

For the transition matrices, we set

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & \rho \end{bmatrix}, \qquad B = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \qquad C = \begin{bmatrix} 0 \\ 0 \\ \sigma_d \end{bmatrix}$$

Our aim is to compute the value-entropy correspondences *shown above*.

The parameters are

$$a_0 = 100, a_1 = 0.5, \rho = 0.9, \sigma_d = 0.05, \beta = 0.95, c = 2, \gamma = 50.0$$

The standard normal distribution for w_t is understood as the agent's baseline, with uncertainty parameterized by θ.

We compute value-entropy correspondences for two policies

1. The no concern for robustness policy F_0, which is the ordinary LQ loss minimizer.
2. A "moderate" concern for robustness policy F_b, with $\theta = 0.02$.

The code for producing the graph shown above, with blue being for the robust policy, is as follows

```
# Model parameters

a_0 = 100
a_1 = 0.5
ρ = 0.9
σ_d = 0.05
β = 0.95
c = 2
γ = 50.0

θ = 0.02
ac = (a_0 - c) / 2.0

# Define LQ matrices

R = np.array([[0.,    ac,    0.],
              [ac, -a_1,  0.5],
              [0.,  0.5,  0.]])

R = -R  # For minimization
Q = γ / 2

A = np.array([[1., 0., 0.],
              [0., 1., 0.],
              [0., 0., ρ]])
B = np.array([[0.],
              [1.],
              [0.]])
C = np.array([[0.],
              [0.],
              [σ_d]])

# ------------------------------------------------------------------------ #
#                                 Functions                                #
# ------------------------------------------------------------------------ #


def evaluate_policy(θ, F):

    """
    Given θ (scalar, dtype=float) and policy F (array_like), returns the
    value associated with that policy under the worst case path for {w_t},
    as well as the entropy level.
    """
```

(continues on next page)

(continued from previous page)

```
    rlq = qe.RBLQ(Q, R, A, B, C, β, θ)
    K_F, P_F, d_F, O_F, o_F = rlq.evaluate_F(F)
    x0 = np.array([[1.], [0.], [0.]])
    value = - x0.T @ P_F @ x0 - d_F
    entropy = x0.T @ O_F @ x0 + o_F
    return list(map(float, (value.item(), entropy.item())))


def value_and_entropy(emax, F, bw, grid_size=1000):

    """
    Compute the value function and entropy levels for a θ path
    increasing until it reaches the specified target entropy value.

    Parameters
    ==========
    emax: scalar
        The target entropy value

    F: array_like
        The policy function to be evaluated

    bw: str
        A string specifying whether the implied shock path follows best
        or worst assumptions. The only acceptable values are 'best' and
        'worst'.

    Returns
    =======
    df: pd.DataFrame
        A pandas DataFrame containing the value function and entropy
        values up to the emax parameter. The columns are 'value' and
        'entropy'.
    """

    if bw == 'worst':
        θs = 1 / np.linspace(1e-8, 1000, grid_size)
    else:
        θs = -1 / np.linspace(1e-8, 1000, grid_size)

    df = pd.DataFrame(index=θs, columns=('value', 'entropy'))

    for θ in θs:
        df.loc[θ] = evaluate_policy(θ, F)
        if df.loc[θ, 'entropy'] >= emax:
            break

    df = df.dropna(how='any')
    return df


# ------------------------------------------------------------------------- #
#                                    Main                                   #
# ------------------------------------------------------------------------- #
```

(continues on next page)

(continued from previous page)

```
# Compute the optimal rule
optimal_lq = qe.LQ(Q, R, A, B, C, beta=β)
Po, Fo, do = optimal_lq.stationary_values()

# Compute a robust rule given θ
baseline_robust = qe.RBLQ(Q, R, A, B, C, β, θ)
Fb, Kb, Pb = baseline_robust.robust_rule()

# Check the positive definiteness of worst-case covariance matrix to
# ensure that θ exceeds the breakdown point
test_matrix = np.identity(Pb.shape[0]) - (C.T @ Pb @ C) / θ
eigenvals, eigenvecs = eig(test_matrix)
assert (eigenvals >= 0).all(), 'θ below breakdown point.'


emax = 1.6e6

optimal_best_case = value_and_entropy(emax, Fo, 'best')
robust_best_case = value_and_entropy(emax, Fb, 'best')
optimal_worst_case = value_and_entropy(emax, Fo, 'worst')
robust_worst_case = value_and_entropy(emax, Fb, 'worst')

fig, ax = plt.subplots()

ax.set_xlim(0, emax)
ax.set_ylabel("Value")
ax.set_xlabel("Entropy")
ax.grid()

for axis in 'x', 'y':
    plt.ticklabel_format(style='sci', axis=axis, scilimits=(0, 0))

plot_args = {'lw': 2, 'alpha': 0.7}

colors = 'r', 'b'

df_pairs = ((optimal_best_case, optimal_worst_case),
            (robust_best_case, robust_worst_case))


class Curve:

    def __init__(self, x, y):
        self.x, self.y = x, y

    def __call__(self, z):
        return np.interp(z, self.x, self.y)


for c, df_pair in zip(colors, df_pairs):
    curves = []
    for df in df_pair:
        # Plot curves
        x, y = df['entropy'], df['value']
        x, y = (np.asarray(a, dtype='float') for a in (x, y))
        egrid = np.linspace(0, emax, 100)
```

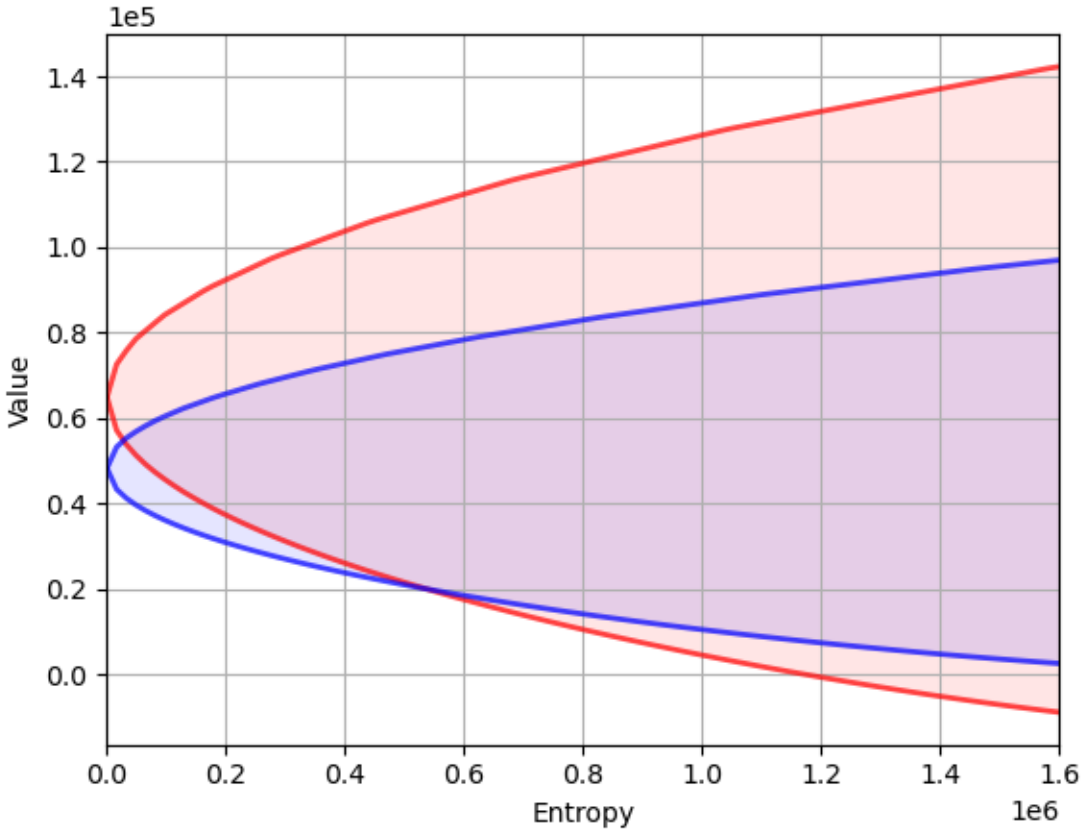
(continues on next page)

(continued from previous page)

```
        curve = Curve(x, y)
        print(ax.plot(egrid, curve(egrid), color=c, **plot_args))
        curves.append(curve)
    # Color fill between curves
    ax.fill_between(egrid,
                    curves[0](egrid),
                    curves[1](egrid),
                    color=c, alpha=0.1)

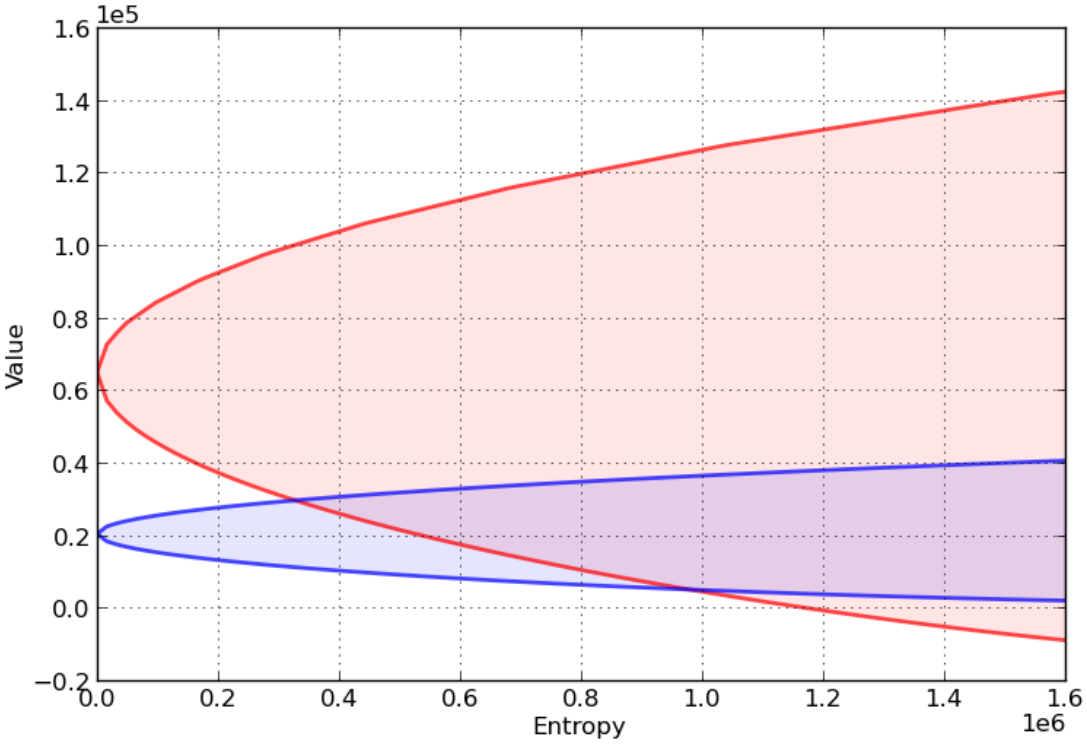
plt.show()
```

```
[<matplotlib.lines.Line2D object at 0x7fcb49be73b0>]
[<matplotlib.lines.Line2D object at 0x7fcb49920110>]
[<matplotlib.lines.Line2D object at 0x7fcb499205c0>]
[<matplotlib.lines.Line2D object at 0x7fcb49920aa0>]
```



Here's another such figure, with $\theta = 0.002$ instead of 0.02

Can you explain the different shape of the value-entropy correspondence for the robust policy?



27.8 Appendix

We sketch the proof only of the first claim in *this section*, which is that, for any given θ, $K(\hat{F}, \theta) = \hat{K}$, where $\hat{K}$ is as given in (27.8).

This is the content of the next lemma.

Lemma. If $\hat{P}$ is the fixed point of the map $\mathcal{B} \circ \mathcal{D}$ and $\hat{F}$ is the robust policy as given in (27.7), then

$$K(\hat{F}, \theta) = (\theta I - C' \hat{P} C)^{-1} C' \hat{P} (A - B\hat{F}) \tag{27.28}$$

Proof: As a first step, observe that when $F = \hat{F}$, the Bellman equation associated with the LQ problem (27.11) – (27.12) is

$$\tilde{P} = -R - \hat{F}' Q \hat{F} - \beta^2 (A - B\hat{F})' \tilde{P} C (\beta\theta I + \beta C' \tilde{P} C)^{-1} C' \tilde{P} (A - B\hat{F}) + \beta (A - B\hat{F})' \tilde{P} (A - B\hat{F}) \tag{27.29}$$

(revisit this discussion if you don't know where (27.29) comes from) and the optimal policy is

$$w_{t+1} = -\beta (\beta\theta I + \beta C' \tilde{P} C)^{-1} C' \tilde{P} (A - B\hat{F}) x_t$$

Suppose for a moment that $-\hat{P}$ solves the Bellman equation (27.29).

In this case, the policy becomes

$$w_{t+1} = (\theta I - C' \hat{P} C)^{-1} C' \hat{P} (A - B\hat{F}) x_t$$

which is exactly the claim in (27.28).

Hence it remains only to show that $-\hat{P}$ solves (27.29), or, in other words,

$$\hat{P} = R + \hat{F}'Q\hat{F} + \beta(A - B\hat{F})'\hat{P}C(\theta I - C'\hat{P}C)^{-1}C'\hat{P}(A - B\hat{F}) + \beta(A - B\hat{F})'\hat{P}(A - B\hat{F})$$

Using the definition of $\mathcal{D}$, we can rewrite the right-hand side more simply as

$$R + \hat{F}'Q\hat{F} + \beta(A - B\hat{F})'\mathcal{D}(\hat{P})(A - B\hat{F})$$

Although it involves a substantial amount of algebra, it can be shown that the latter is just $\hat{P}$.

Hint: Use the fact that $\hat{P} = \mathcal{B}(\mathcal{D}(\hat{P}))$

CHAPTER

TWENTYEIGHT

ROBUST MARKOV PERFECT EQUILIBRIUM

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install --upgrade quantecon
```

28.1 Overview

This lecture describes a Markov perfect equilibrium with robust agents.

We focus on special settings with

- two players
- quadratic payoff functions
- linear transition rules for the state vector

These specifications simplify calculations and allow us to give a simple example that illustrates basic forces.

This lecture is based on ideas described in chapter 15 of [Hansen and Sargent, 2008] and in Markov perfect equilibrium and *Robustness*.

Let's start with some standard imports:

```
import numpy as np
import quantecon as qe
from scipy.linalg import solve
import matplotlib.pyplot as plt
```

28.1.1 Basic Setup

Decisions of two agents affect the motion of a state vector that appears as an argument of payoff functions of both agents.

As described in Markov perfect equilibrium, when decision-makers have no concerns about the robustness of their decision rules to misspecifications of the state dynamics, a Markov perfect equilibrium can be computed via backward recursion on two sets of equations

- a pair of Bellman equations, one for each agent.
- a pair of equations that express linear decision rules for each agent as functions of that agent's continuation value function as well as parameters of preferences and state transition matrices.

This lecture shows how a similar equilibrium concept and similar computational procedures apply when we impute concerns about robustness to both decision-makers.

A Markov perfect equilibrium with robust agents will be characterized by

- a pair of Bellman equations, one for each agent.
- a pair of equations that express linear decision rules for each agent as functions of that agent's continuation value function as well as parameters of preferences and state transition matrices.
- a pair of equations that express linear decision rules for worst-case shocks for each agent as functions of that agent's continuation value function as well as parameters of preferences and state transition matrices.

Below, we'll construct a robust firms version of the classic duopoly model with adjustment costs analyzed in Markov perfect equilibrium.

28.2 Linear Markov Perfect Equilibria with Robust Agents

As we saw in Markov perfect equilibrium, the study of Markov perfect equilibria in dynamic 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.

28.2.1 Modified Coupled Linear Regulator Problems

We consider a general linear quadratic regulator game with two players, each of whom fears model misspecifications.

We often call the players agents.

The agents share a common baseline model for the transition dynamics of the state vector

- this is a counterpart of a 'rational expectations' assumption of shared beliefs

But now one or more agents doubt that the baseline model is correctly specified.

The agents express the possibility that their baseline specification is incorrect by adding a contribution Cv_{it} to the time t transition law for the state

- C is the usual *volatility matrix* that appears in stochastic versions of optimal linear regulator problems.
- v_{it} is a possibly history-dependent vector of distortions to the dynamics of the state that agent i uses to represent misspecification of the original model.

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 a sequence $\{u_{-it}\}$ as given and chooses a sequence $\{u_{it}\}$ to minimize and $\{v_{it}\}$ to maximize

$$\sum_{t=t_0}^{t_1-1} \beta^{t-t_0} \{x_t' R_i x_t + u_{it}' Q_i u_{it} + u_{-it}' S_i u_{-it} + 2x_t' W_i u_{it} + 2u_{-it}' M_i u_{it} - \theta_i v_{it}' v_{it}\} \tag{28.1}$$

while thinking that the state evolves according to

$$x_{t+1} = Ax_t + B_1 u_{1t} + B_2 u_{2t} + Cv_{it} \tag{28.2}$$

Here

- x_t is an $n \times 1$ state vector, u_{it} is a $k_i \times 1$ vector of controls for player i, and

- v_{it} is an $h \times 1$ vector of distortions to the state dynamics that concern 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 $n \times k_i$
- C is $n \times h$
- $\theta_i \in [\underline{\theta}_i, +\infty]$ is a scalar multiplier parameter of player i

If $\theta_i = +\infty$, player i completely trusts the baseline model.

If $\theta_i <_\infty$, player i suspects that some other unspecified model actually governs the transition dynamics.

The term $\theta_i v_{it}' v_{it}$ is a time t contribution to an entropy penalty that an (imaginary) loss-maximizing agent inside agent i's mind charges for distorting the law of motion in a way that harms agent i.

- the imaginary loss-maximizing agent helps the loss-minimizing agent by helping him construct bounds on the behavior of his decision rule over a large **set** of alternative models of state transition dynamics.

28.2.2 Computing Equilibrium

We formulate a linear robust 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.

Player i's malevolent alter ego employs decision rules $v_{it} = K_{it} x_t$ where K_{it} is an $h \times n$ matrix.

A robust Markov perfect equilibrium is a pair of sequences $\{F_{1t}, F_{2t}\}$ and a pair of sequences $\{K_{1t}, K_{2t}\}$ over $t = t_0, \ldots, t_1 - 1$ that satisfy

- $\{F_{1t}, K_{1t}\}$ solves player 1's robust decision problem, taking $\{F_{2t}\}$ as given, and
- $\{F_{2t}, K_{2t}\}$ solves player 2's robust decision problem, taking $\{F_{1t}\}$ as given.

If we substitute $u_{2t} = -F_{2t} x_t$ into (28.1) and (28.2), then player 1's problem becomes minimization-maximization of

$$\sum_{t=t_0}^{t_1-1} \beta^{t-t_0} \left\{ x_t' \Pi_{1t} x_t + u_{1t}' Q_1 u_{1t} + 2 u_{1t}' \Gamma_{1t} x_t - \theta_1 v_{1t}' v_{1t} \right\} \tag{28.3}$$

subject to

$$x_{t+1} = \Lambda_{1t} x_t + B_1 u_{1t} + C v_{1t} \tag{28.4}$$

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 robust dynamic programming problem of the type studied in the *Robustness* lecture, which can be solved by working backward.

Maximization with respect to distortion v_{1t} leads to the following version of the $\mathcal{D}$ operator from the *Robustness* lecture, namely

$$\mathcal{D}_1(P) := P + PC(\theta_1 I - C'PC)^{-1}C'P \tag{28.5}$$

The matrix F_{1t} in the policy rule $u_{1t} = -F_{1t}x_t$ that solves agent 1's problem satisfies

$$F_{1t} = (Q_1 + \beta B_1'\mathcal{D}_1(P_{1t+1})B_1)^{-1}(\beta B_1'\mathcal{D}_1(P_{1t+1})\Lambda_{1t} + \Gamma_{1t}) \tag{28.6}$$

where P_{1t} solves the matrix Riccati difference equation

$$\begin{aligned} P_{1t} = \Pi_{1t} - (\beta B_1'\mathcal{D}_1(P_{1t+1})\Lambda_{1t} + \Gamma_{1t})'(Q_1 + \beta B_1'\mathcal{D}_1(P_{1t+1})B_1)^{-1}(\beta B_1'\mathcal{D}_1(P_{1t+1})\Lambda_{1t} + \Gamma_{1t}) + \\ \beta\Lambda_{1t}'\mathcal{D}_1(P_{1t+1})\Lambda_{1t} \end{aligned} \tag{28.7}$$

Similarly, the policy that solves player 2's problem is

$$F_{2t} = (Q_2 + \beta B_2'\mathcal{D}_2(P_{2t+1})B_2)^{-1}(\beta B_2'\mathcal{D}_2(P_{2t+1})\Lambda_{2t} + \Gamma_{2t}) \tag{28.8}$$

where P_{2t} solves

$$\begin{aligned} P_{2t} = \Pi_{2t} - (\beta B_2'\mathcal{D}_2(P_{2t+1})\Lambda_{2t} + \Gamma_{2t})'(Q_2 + \beta B_2'\mathcal{D}_2(P_{2t+1})B_2)^{-1}(\beta B_2'\mathcal{D}_2(P_{2t+1})\Lambda_{2t} + \Gamma_{2t}) + \\ \beta\Lambda_{2t}'\mathcal{D}_2(P_{2t+1})\Lambda_{2t} \end{aligned} \tag{28.9}$$

Here in all cases $t = t_0, \ldots, t_1 - 1$ and the terminal conditions are $P_{it_1} = 0$.

The solution procedure is to use equations (28.6), (28.7), (28.8), and (28.9), and "work backwards" from time $t_1 - 1$.

Since we're working backwards, P_{1t+1} and P_{2t+1} are taken as given at each stage.

Moreover, since

- some terms on the right-hand side of (28.6) contain F_{2t}
- some terms on the right-hand side of (28.8) contain F_{1t}

we need to solve these $k_1 + k_2$ equations simultaneously.

28.2.3 Key Insight

As in Markov perfect equilibrium, a key insight here is that equations (28.6) and (28.8) 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 (28.7) and (28.9).

Notice how j's control law F_{jt} is a function of $\{F_{is}, s \geq t, i \neq j\}$.

Thus, agent i's choice of $\{F_{it}; t = t_0, \ldots, t_1 - 1\}$ influences agent j's choice of control laws.

However, in the Markov perfect equilibrium of this game, each agent is assumed to ignore the influence that his choice exerts on the other agent's choice.

After these equations have been solved, we can also deduce associated sequences of worst-case shocks.

28.2.4 Worst-case Shocks

For agent i the maximizing or worst-case shock v_{it} is

$$v_{it} = K_{it} x_t$$

where

$$K_{it} = \theta_i^{-1}(I - \theta_i^{-1} C' P_{i,t+1} C)^{-1} C' P_{i,t+1}(A - B_1 F_{it} - B_2 F_{2t})$$

28.2.5 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.

28.2.6 Implementation

We use the function nnash_robust to compute a Markov perfect equilibrium of the infinite horizon linear quadratic dynamic game with robust planers in the manner described above.

28.3 Application

28.3.1 A Duopoly Model

Without concerns for robustness, the model is identical to the duopoly model from the Markov perfect equilibrium lecture.

To begin, we briefly review the structure of that model.

Two firms are the only producers of a good the demand for which is governed by a linear inverse demand function

$$p = a_0 - a_1(q_1 + q_2) \tag{28.10}$$

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 (28.10) 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:

$$\pi_i = p q_i - \gamma(\hat{q}_i - q_i)^2, \quad \gamma > 0, \tag{28.11}$$

Substituting the inverse demand curve (28.10) into (28.11) lets us express the one-period payoff as

$$\pi_i(q_i, q_{-i}, \hat{q}_i) = a_0 q_i - a_1 q_i^2 - a_1 q_i q_{-i} - \gamma(\hat{q}_i - q_i)^2, \tag{28.12}$$

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}).

This completes our review of the duopoly model without concerns for robustness.

Now we activate robustness concerns of both firms.

To map a robust version of the duopoly model into coupled robust linear-quadratic dynamic programming problems, we again 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 (28.11) for the two firms in the duopoly model.

The law of motion for the state x_t is $x_{t+1} = Ax_t + B_1 u_{1t} + B_2 u_{2t}$ where

$$A := \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad B_1 := \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \quad B_2 := \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

A robust 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:

$$x_{t+1} = (A - B_1 F_1 - B_1 F_2) x_t \tag{28.13}$$

28.3.2 Parameters and Solution

Consider the duopoly model with parameter values of:

- $a_0 = 10$
- $a_1 = 2$
- $\beta = 0.96$
- $\gamma = 12$

From these, we computed the infinite horizon MPE without robustness using the code

```
import numpy as np
import quantecon as qe

# Parameters
a0 = 10.0
a1 = 2.0
β = 0.96
```

(continues on next page)

(continued from previous page)

```
y = 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 = y
S1 = S2 = W1 = W2 = M1 = M2 = 0.0

# Solve using QE's nnash function
F1, F2, P1, P2 = qe.nnash(A, B1, B2, R1, R2, Q1,
                          Q2, S1, S2, W1, W2, M1,
                          M2, beta=β)

# Display policies
print("Computed policies for firm 1 and firm 2:\n")
print(f"F1 = {F1}")
print(f"F2 = {F2}")
print("\n")
```

```
Computed policies for firm 1 and firm 2:

F1 = [[-0.66846615  0.29512482  0.07584666]]
F2 = [[-0.66846615  0.07584666  0.29512482]]
```

Markov Perfect Equilibrium with Robustness

We add robustness concerns to the Markov Perfect Equilibrium model by extending the function `qe.nnash` (link) into a robustness version by adding the maximization operator $\mathcal{D}(P)$ into the backward induction.

The MPE with robustness function is `nnash_robust`.

The function's code is as follows

```
def nnash_robust(A, C, B1, B2, R1, R2, Q1, Q2, S1, S2, W1, W2, M1, M2,
                 θ1, θ2, beta=1.0, tol=1e-8, max_iter=1000):

    r"""
    Compute the limit of a Nash linear quadratic dynamic game with
    robustness concern.

    In this problem, player i minimizes
    .. math::
        \sum_{t=0}^{\infty}
```

(continues on next page)

(continued from previous page)

```
        \left\{
            x_t' r_i x_t + 2 x_t' w_i
            u_{it} +u_{it}' q_i u_{it} + u_{jt}' s_i u_{jt} + 2 u_{jt}'
            m_i u_{it}
        \right\}
    subject to the law of motion
    .. math::
        x_{it+1} = A x_t + b_1 u_{1t} + b_2 u_{2t} + C w_{it+1}
    and a perceived control law :math:`u_j(t) = - f_j x_t` for the other
    player.

    The player i also concerns about the model misspecification,
    and maximizes
    .. math::
        \sum_{t=0}^{\infty}
        \left\{
            \beta^{t+1} \theta_{i} w_{it+1}'w_{it+1}
        \right\}

    The solution computed in this routine is the :math:`f_i` and
    :math:`P_i` of the associated double optimal linear regulator
    problem.

    Parameters
    ----------
    A : scalar(float) or array_like(float)
        Corresponds to the MPE equations, should be of size (n, n)
    C : scalar(float) or array_like(float)
        As above, size (n, c), c is the size of w
    B1 : scalar(float) or array_like(float)
        As above, size (n, k_1)
    B2 : scalar(float) or array_like(float)
        As above, size (n, k_2)
    R1 : scalar(float) or array_like(float)
        As above, size (n, n)
    R2 : scalar(float) or array_like(float)
        As above, size (n, n)
    Q1 : scalar(float) or array_like(float)
        As above, size (k_1, k_1)
    Q2 : scalar(float) or array_like(float)
        As above, size (k_2, k_2)
    S1 : scalar(float) or array_like(float)
        As above, size (k_1, k_1)
    S2 : scalar(float) or array_like(float)
        As above, size (k_2, k_2)
    W1 : scalar(float) or array_like(float)
        As above, size (n, k_1)
    W2 : scalar(float) or array_like(float)
        As above, size (n, k_2)
    M1 : scalar(float) or array_like(float)
        As above, size (k_2, k_1)
    M2 : scalar(float) or array_like(float)
        As above, size (k_1, k_2)
    θ1 : scalar(float)
            Robustness parameter of player 1
    θ2 : scalar(float)
```

(continues on next page)

(continued from previous page)

```
        Robustness parameter of player 2
    beta : scalar(float), optional(default=1.0)
         Discount factor
    tol : scalar(float), optional(default=1e-8)
         This is the tolerance level for convergence
    max_iter : scalar(int), optional(default=1000)
        This is the maximum number of iterations allowed

    Returns
    -------
    F1 : array_like, dtype=float, shape=(k_1, n)
        Feedback law for agent 1
    F2 : array_like, dtype=float, shape=(k_2, n)
        Feedback law for agent 2
    P1 : array_like, dtype=float, shape=(n, n)
        The steady-state solution to the associated discrete matrix
        Riccati equation for agent 1
    P2 : array_like, dtype=float, shape=(n, n)
        The steady-state solution to the associated discrete matrix
        Riccati equation for agent 2
    """

    # Unload parameters and make sure everything is a matrix
    params = A, C, B1, B2, R1, R2, Q1, Q2, S1, S2, W1, W2, M1, M2
    params = map(np.asmatrix, params)
    A, C, B1, B2, R1, R2, Q1, Q2, S1, S2, W1, W2, M1, M2 = params


    # Multiply A, B1, B2 by sqrt(β) to enforce discounting
    A, B1, B2 = [np.sqrt(β) * x for x in (A, B1, B2)]

    # Initial values
    n = A.shape[0]
    k_1 = B1.shape[1]
    k_2 = B2.shape[1]

    v1 = np.eye(k_1)
    v2 = np.eye(k_2)
    P1 = np.eye(n) * 1e-5
    P2 = np.eye(n) * 1e-5
    F1 = np.random.randn(k_1, n)
    F2 = np.random.randn(k_2, n)


    for it in range(max_iter):
        # Update
        F10 = F1
        F20 = F2

        I = np.eye(C.shape[1])

        # D1(P1)
        # Note: INV1 may not be solved if the matrix is singular
        INV1 = solve(θ1 * I - C.T @ P1 @ C, I)
        D1P1 =  P1 + P1 @ C @ INV1 @ C.T @ P1
```

(continues on next page)

(continued from previous page)

```
        # D2(P2)
        # Note: INV2 may not be solved if the matrix is singular
        INV2 = solve(θ2 * I - C.T @ P2 @ C, I)
        D2P2 =  P2 + P2 @ C @ INV2 @ C.T @ P2

        G2 = solve(Q2 + B2.T @ D2P2 @ B2, v2)
        G1 = solve(Q1 + B1.T @ D1P1 @ B1, v1)
        H2 = G2 @ B2.T @ D2P2
        H1 = G1 @ B1.T @ D1P1

        # Break up the computation of F1, F2
        F1_left = v1 - (H1 @ B2 + G1 @ M1.T) @ (H2 @ B1 + G2 @ M2.T)
        F1_right = H1 @ A + G1 @ W1.T - \
                   (H1 @ B2 + G1 @ M1.T) @ (H2 @ A + G2 @ W2.T)
        F1 = solve(F1_left, F1_right)
        F2 = H2 @ A + G2 @ W2.T - (H2 @ B1 + G2 @ M2.T) @ F1

        Λ1 = A - B2 @ F2
        Λ2 = A - B1 @ F1
        Π1 = R1 + F2.T @ S1 @ F2
        Π2 = R2 + F1.T @ S2 @ F1
        Γ1 = W1.T - M1.T @ F2
        Γ2 = W2.T - M2.T @ F1

        # Compute P1 and P2
        P1 = Π1 - (B1.T @ D1P1 @ Λ1 + Γ1).T @ F1 + \
             Λ1.T @ D1P1 @ Λ1
        P2 = Π2 - (B2.T @ D2P2 @ Λ2 + Γ2).T @ F2 + \
             Λ2.T @ D2P2 @ Λ2

        dd = np.max(np.abs(F10 - F1)) + np.max(np.abs(F20 - F2))

        if dd < tol:  # success!
            break

    else:
        raise ValueError(f'No convergence: Iteration limit of {max_iter} \
            reached in nnash')

    return F1, F2, P1, P2
```

28.3.3 Some Details

Firm i wants to minimize

$$\sum_{t=t_0}^{t_1-1} \beta^{t-t_0} \{x_t' R_i x_t + u_{it}' Q_i u_{it} + u_{-it}' S_i u_{-it} + 2 x_t' W_i u_{it} + 2 u_{-it}' M_i u_{it}\}$$

where

$$x_t := \begin{bmatrix} 1 \\ q_{1t} \\ q_{2t} \end{bmatrix} \quad \text{and} \quad u_{it} := q_{i,t+1} - q_{it}, \quad i = 1, 2$$

and

$$R_1 := \begin{bmatrix} 0 & -\frac{a_0}{2} & 0 \\ -\frac{a_0}{2} & a_1 & \frac{a_1}{2} \\ 0 & \frac{a_1}{2} & 0 \end{bmatrix}, \quad R_2 := \begin{bmatrix} 0 & 0 & -\frac{a_0}{2} \\ 0 & 0 & \frac{a_1}{2} \\ -\frac{a_0}{2} & \frac{a_1}{2} & a_1 \end{bmatrix}, \quad Q_1 = Q_2 = \gamma, \quad S_1 = S_2 = 0, \quad W_1 = W_2 = 0, \quad M_1 = M_2 = 0$$

The parameters of the duopoly model are:

- $a_0 = 10$
- $a_1 = 2$
- $\beta = 0.96$
- $\gamma = 12$

```
# 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
```

Consistency Check

We first conduct a comparison test to check if `nnash_robust` agrees with `qe.nnash` in the non-robustness case in which each $\theta_i \approx +\infty$

```
# 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)

# Solve using nnash_robust
F1r, F2r, P1r, P2r = nnash_robust(A, np.zeros((3, 1)), B1, B2, R1, R2, Q1,
                                  Q2, S1, S2, W1, W2, M1, M2, 1e-10,
                                  1e-10, beta=β)

print('F1 and F1r should be the same: ', np.allclose(F1, F1r))
print('F2 and F2r should be the same: ', np.allclose(F1, F1r))
```

(continues on next page)

(continued from previous page)

```
print('P1 and P1r should be the same: ', np.allclose(P1, P1r))
print('P2 and P2r should be the same: ', np.allclose(P1, P1r))
```

```
F1 and F1r should be the same:  True
F2 and F2r should be the same:  True
P1 and P1r should be the same:  True
P2 and P2r should be the same:  True
```

We can see that the results are consistent across the two functions.

Comparative Dynamics under Baseline Transition Dynamics

We want to compare the dynamics of price and output under the baseline MPE model with those under the baseline model under the robust decision rules within the robust MPE.

This means that we simulate the state dynamics under the MPE equilibrium **closed-loop** transition matrix

$$A^o = A - B_1 F_1 - B_2 F_2$$

where F_1 and F_2 are the firms' robust decision rules within the robust markov_perfect equilibrium

- by simulating under the baseline model transition dynamics and the robust MPE rules we are in assuming that at the end of the day firms' concerns about misspecification of the baseline model do not materialize.
- a short way of saying this is that misspecification fears are all 'just in the minds' of the firms.
- simulating under the baseline model is a common practice in the literature.
- note that *some* assumption about the model that actually governs the data has to be made in order to create a simulation.
- later we will describe the (erroneous) beliefs of the two firms that justify their robust decisions as best responses to transition laws that are distorted relative to the baseline model.

After simulating x_t under the baseline transition dynamics and robust decision rules $F_i, i = 1, 2$, we extract and plot industry output $q_t = q_{1t} + q_{2t}$ and price $p_t = a_0 - a_1 q_t$.

Here we set the robustness and volatility matrix parameters as follows:

- $\theta_1 = 0.02$
- $\theta_2 = 0.04$
- $C = \begin{pmatrix} 0 \\ 0.01 \\ 0.01 \end{pmatrix}$

Because we have set $\theta_1 < \theta_2 < +\infty$ we know that

- both firms fear that the baseline specification of the state transition dynamics are incorrect.
- firm 1 fears misspecification more than firm 2.

```
# Robustness parameters and matrix
C = np.asmatrix([[0], [0.01], [0.01]])
θ1 = 0.02
θ2 = 0.04
n = 20
```

(continues on next page)

(continued from previous page)

```
# Solve using nnash_robust
F1r, F2r, P1r, P2r = nnash_robust(A, C, B1, B2, R1, R2, Q1,
                                  Q2, S1, S2, W1, W2, M1, M2,
                                  θ1, θ2, beta=β)



# MPE output and price
AF = A - B1 @ F1 - B2 @ F2
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


# RMPE output and price
AO = A - B1 @ F1r - B2 @ F2r
xr = np.empty((3, n))
xr[:, 0] = 1, 1, 1
for t in range(n - 1):
    xr[:, t+1] = AO @ xr[:, t]
qr1 = xr[1, :]
qr2 = xr[2, :]
qr = qr1 + qr2       # Total output, RMPE
pr = a0 - a1 * qr    # Price, RMPE

# RMPE heterogeneous beliefs output and price
I = np.eye(C.shape[1])
INV1 = solve(θ1 * I - C.T @ P1 @ C, I)
K1 =  P1 @ C @ INV1 @ C.T @ P1 @ AO
AOCK1 = AO + C.T @ K1

INV2 = solve(θ2 * I - C.T @ P2 @ C, I)
K2 =  P2 @ C @ INV2 @ C.T @ P2 @ AO
AOCK2 = AO + C.T @ K2
xrp1 = np.empty((3, n))
xrp2 = np.empty((3, n))
xrp1[:, 0] = 1, 1, 1
xrp2[:, 0] = 1, 1, 1
for t in range(n - 1):
    xrp1[:, t + 1] = AOCK1 @ xrp1[:, t]
    xrp2[:, t + 1] = AOCK2 @ xrp2[:, t]
qrp11 = xrp1[1, :]
qrp12 = xrp1[2, :]
qrp21 = xrp2[1, :]
qrp22 = xrp2[2, :]
qrp1 = qrp11 + qrp12       # Total output, RMPE from player 1's belief
qrp2 = qrp21 + qrp22       # Total output, RMPE from player 2's belief
prp1 = a0 - a1 * qrp1      # Price, RMPE from player 1's belief
prp2 = a0 - a1 * qrp2      # Price, RMPE from player 2's belief
```

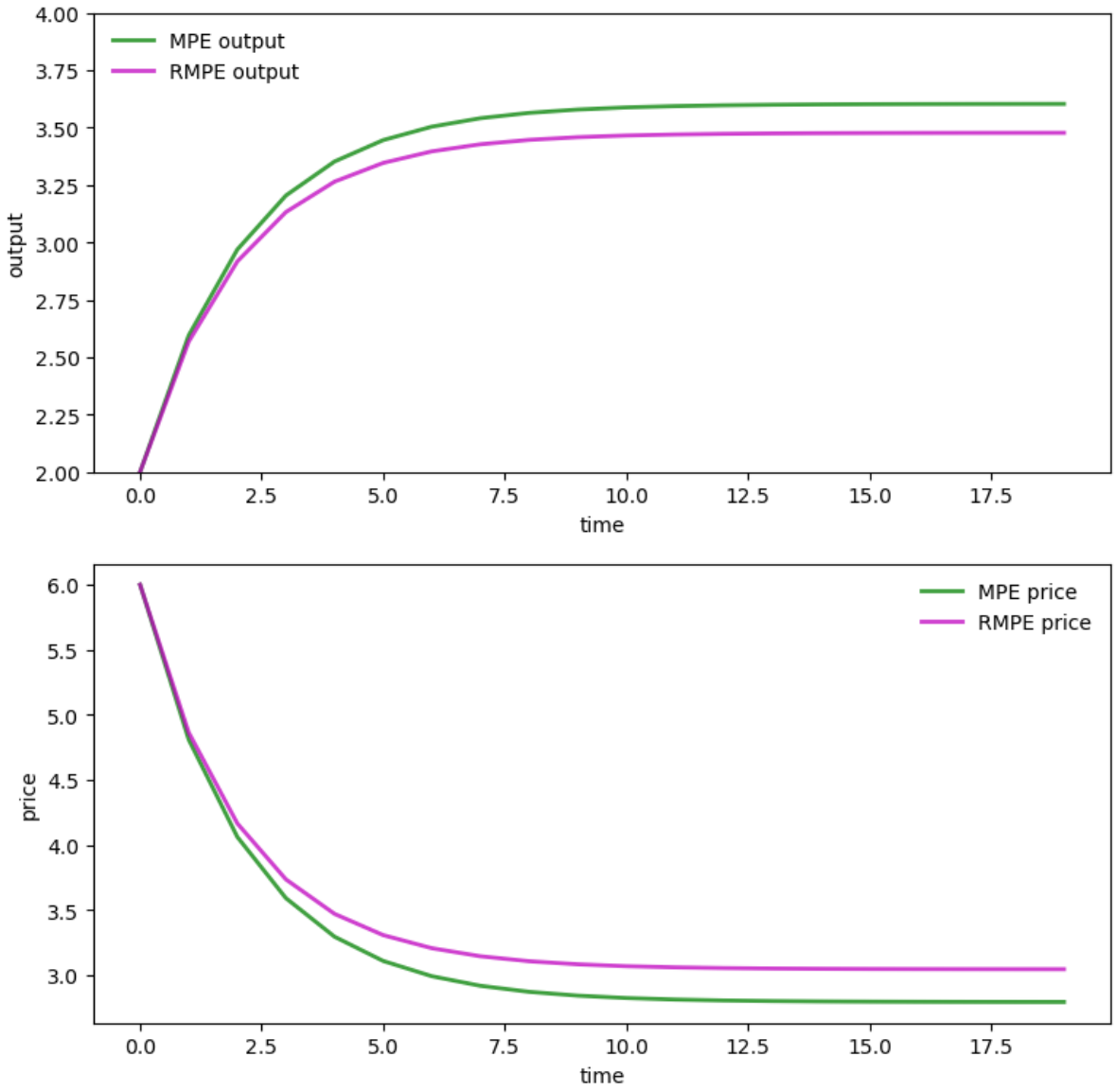
The following code prepares graphs that compare market-wide output $q_{1t} + q_{2t}$ and the price of the good p_t under equilibrium decision rules $F_i, i = 1, 2$ from an ordinary Markov perfect equilibrium and the decision rules under a Markov perfect equilibrium with robust firms with multiplier parameters $\theta_i, i = 1, 2$ set as described above.

Both industry output and price are under the transition dynamics associated with the baseline model; only the decision rules F_i differ across the two equilibrium objects presented.

```
fig, axes = plt.subplots(2, 1, figsize=(9, 9))

ax = axes[0]
ax.plot(q, 'g-', lw=2, alpha=0.75, label='MPE output')
ax.plot(qr, 'm-', lw=2, alpha=0.75, label='RMPE output')
ax.set(ylabel="output", xlabel="time", ylim=(2, 4))
ax.legend(loc='upper left', frameon=0)

ax = axes[1]
ax.plot(p, 'g-', lw=2, alpha=0.75, label='MPE price')
ax.plot(pr, 'm-', lw=2, alpha=0.75, label='RMPE price')
ax.set(ylabel="price", xlabel="time")
ax.legend(loc='upper right', frameon=0)
plt.show()
```



Under the dynamics associated with the baseline model, the price path is higher with the Markov perfect equilibrium robust decision rules than it is with decision rules for the ordinary Markov perfect equilibrium.

So is the industry output path.

To dig a little beneath the forces driving these outcomes, we want to plot q_{1t} and q_{2t} in the Markov perfect equilibrium with robust firms and to compare them with corresponding objects in the Markov perfect equilibrium without robust firms

```
fig, axes = plt.subplots(2, 1, figsize=(9, 9))

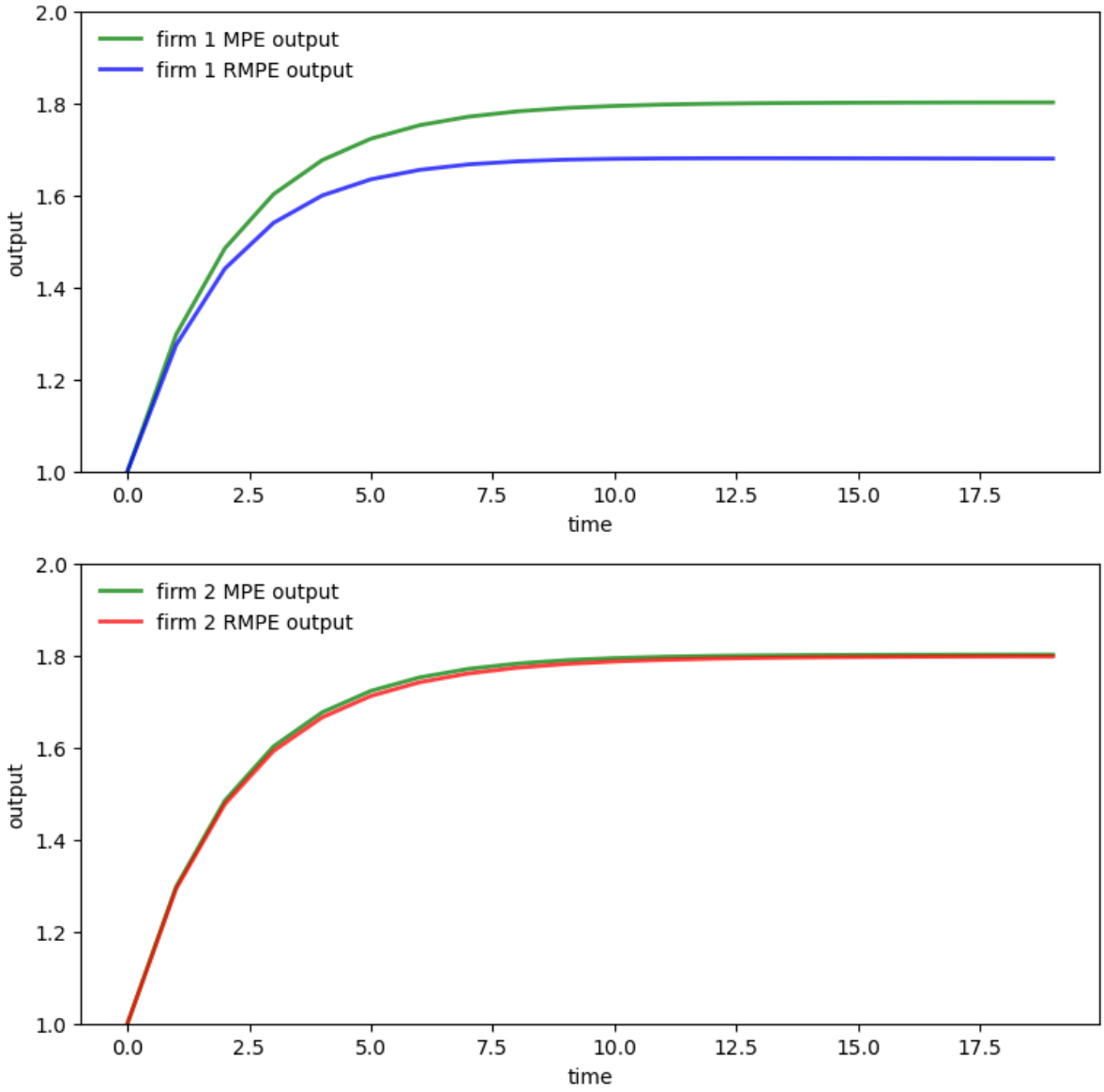
ax = axes[0]
ax.plot(q1, 'g-', lw=2, alpha=0.75, label='firm 1 MPE output')
ax.plot(qr1, 'b-', lw=2, alpha=0.75, label='firm 1 RMPE output')
ax.set(ylabel="output", xlabel="time", ylim=(1, 2))
ax.legend(loc='upper left', frameon=0)

ax = axes[1]
```

(continues on next page)

(continued from previous page)

```
ax.plot(q2, 'g-', lw=2, alpha=0.75, label='firm 2 MPE output')
ax.plot(qr2, 'r-', lw=2, alpha=0.75, label='firm 2 RMPE output')
ax.set(ylabel="output", xlabel="time", ylim=(1, 2))
ax.legend(loc='upper left', frameon=0)
plt.show()
```



Evidently, firm 1's output path is substantially lower when firms are robust firms while firm 2's output path is virtually the same as it would be in an ordinary Markov perfect equilibrium with no robust firms.

Recall that we have set $\theta_1 = .02$ and $\theta_2 = .04$, so that firm 1 fears misspecification of the baseline model substantially more than does firm 2

- but also please notice that firm 2's behavior in the Markov perfect equilibrium with robust firms responds to the decision rule $F_1 x_t$ employed by firm 1.
- thus it is something of a coincidence that its output is almost the same in the two equilibria.

Larger concerns about misspecification induce firm 1 to be more cautious than firm 2 in predicting market price and the output of the other firm.

To explore this, we study next how *ex-post* the two firms' beliefs about state dynamics differ in the Markov perfect equilibrium with robust firms.

(by *ex-post* we mean *after* extremization of each firm's intertemporal objective)

Heterogeneous Beliefs

As before, let $A^o = A - B_1F_1^r - B_2F_2^r$, where in a robust MPE, F_i^r is a robust decision rule for firm i.

Worst-case forecasts of x_t starting from $t = 0$ differ between the two firms.

This means that worst-case forecasts of industry output $q_{1t} + q_{2t}$ and price p_t also differ between the two firms.

To find these worst-case beliefs, we compute the following three "closed-loop" transition matrices

- A^o
- $A^o + CK_1$
- $A^o + CK_2$

We call the first transition law, namely, A^o, the baseline transition under firms' robust decision rules.

We call the second and third worst-case transitions under robust decision rules for firms 1 and 2.

From $\{x_t\}$ paths generated by each of these transition laws, we pull off the associated price and total output sequences.

The following code plots them

```
print('Baseline Robust transition matrix AO is: \n', np.round(AO, 3))
print('Player 1\'s worst-case transition matrix AOCK1 is: \n', \
np.round(AOCK1, 3))
print('Player 2\'s worst-case transition matrix AOCK2 is: \n', \
np.round(AOCK2, 3))
```

```
Baseline Robust transition matrix AO is:
 [[ 1.     0.     0.   ]
 [ 0.666  0.682 -0.074]
 [ 0.671 -0.071  0.694]]
Player 1's worst-case transition matrix AOCK1 is:
 [[ 0.998  0.002  0.   ]
 [ 0.664  0.685 -0.074]
 [ 0.669 -0.069  0.694]]
Player 2's worst-case transition matrix AOCK2 is:
 [[ 0.999  0.     0.001]
 [ 0.665  0.683 -0.073]
 [ 0.67  -0.071  0.695]]
```

```
# == Plot == #
fig, axes = plt.subplots(2, 1, figsize=(9, 9))

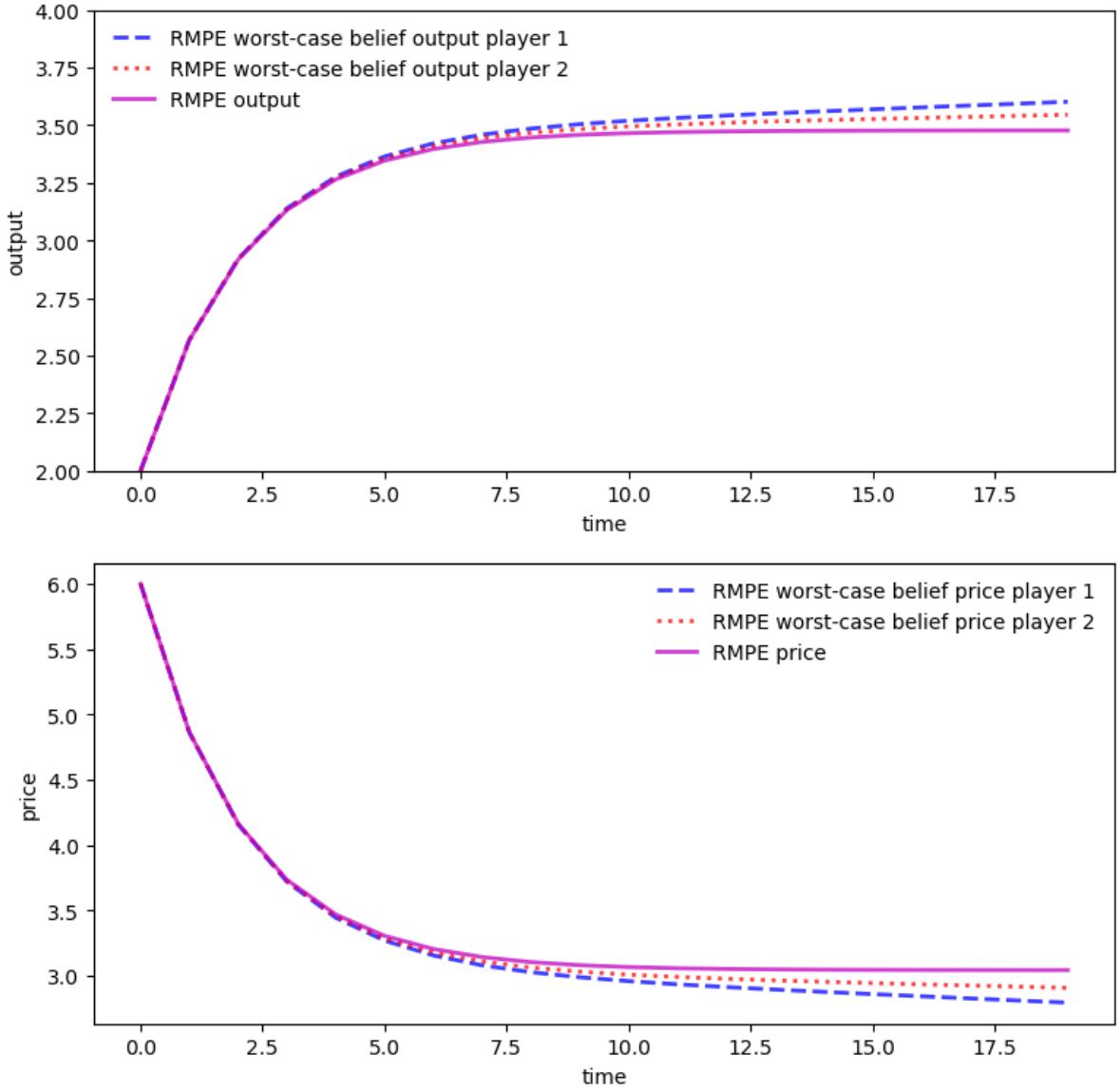
ax = axes[0]
ax.plot(qrp1, 'b--', lw=2, alpha=0.75,
    label='RMPE worst-case belief output player 1')
ax.plot(qrp2, 'r:', lw=2, alpha=0.75,
    label='RMPE worst-case belief output player 2')
```

(continues on next page)

(continued from previous page)

```
ax.plot(qr, 'm-', lw=2, alpha=0.75, label='RMPE output')
ax.set(ylabel="output", xlabel="time", ylim=(2, 4))
ax.legend(loc='upper left', frameon=0)

ax = axes[1]
ax.plot(prp1, 'b--', lw=2, alpha=0.75,
    label='RMPE worst-case belief price player 1')
ax.plot(prp2, 'r:', lw=2, alpha=0.75,
    label='RMPE worst-case belief price player 2')
ax.plot(pr, 'm-', lw=2, alpha=0.75, label='RMPE price')
ax.set(ylabel="price", xlabel="time")
ax.legend(loc='upper right', frameon=0)
plt.show()
```



We see from the above graph that under robustness concerns, player 1 and player 2 have heterogeneous beliefs about total output and the goods price even though they share the same baseline model and information

- firm 1 thinks that total output will be higher and price lower than does firm 2
- this leads firm 1 to produce less than firm 2

These beliefs justify (or **rationalize**) the Markov perfect equilibrium robust decision rules.

This means that the robust rules are the unique **optimal** rules (or best responses) to the indicated worst-case transition dynamics.

([Hansen and Sargent, 2008] discuss how this property of robust decision rules is connected to the concept of *admissibility* in Bayesian statistical decision theory)

Part VI

Time Series Models

CHAPTER

TWENTYNINE

COVARIANCE STATIONARY PROCESSES

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install --upgrade quantecon
```

29.1 Overview

In this lecture we study covariance stationary linear stochastic processes, a class of models routinely used to study economic and financial time series.

This class has the advantage of being

1. simple enough to be described by an elegant and comprehensive theory
2. relatively broad in terms of the kinds of dynamics it can represent

We consider these models in both the time and frequency domain.

29.1.1 ARMA Processes

We will focus much of our attention on linear covariance stationary models with a finite number of parameters.

In particular, we will study stationary ARMA processes, which form a cornerstone of the standard theory of time series analysis.

Every ARMA process can be represented in linear state space form.

However, ARMA processes have some important structure that makes it valuable to study them separately.

29.1.2 Spectral Analysis

Analysis in the frequency domain is also called spectral analysis.

In essence, spectral analysis provides an alternative representation of the autocovariance function of a covariance stationary process.

Having a second representation of this important object

- shines a light on the dynamics of the process in question
- allows for a simpler, more tractable representation in some important cases

The famous *Fourier transform* and its inverse are used to map between the two representations.

29.1.3 Other Reading

For supplementary reading, see

- [Ljungqvist and Sargent, 2018], chapter 2
- [Sargent, 1987], chapter 11
- John Cochrane's notes on time series analysis, chapter 8
- [Shiriaev, 1995], chapter 6
- [Cryer and Chan, 2008], all

Let's start with some imports:

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

29.2 Introduction

Consider a sequence of random variables $\{X_t\}$ indexed by $t \in \mathbb{Z}$ and taking values in $\mathbb{R}$.

Thus, $\{X_t\}$ begins in the infinite past and extends to the infinite future — a convenient and standard assumption.

As in other fields, successful economic modeling typically assumes the existence of features that are constant over time.

If these assumptions are correct, then each new observation $X_t, X_{t+1}, \ldots$ can provide additional information about the time-invariant features, allowing us to learn from as data arrive.

For this reason, we will focus in what follows on processes that are *stationary* — or become so after a transformation (see for example *this lecture*).

29.2.1 Definitions

A real-valued stochastic process $\{X_t\}$ is called *covariance stationary* if

1. Its mean $\mu := \mathbb{E}X_t$ does not depend on t.
2. For all k in $\mathbb{Z}$, the k-th autocovariance $\gamma(k) := \mathbb{E}(X_t - \mu)(X_{t+k} - \mu)$ is finite and depends only on k.

The function $\gamma \colon \mathbb{Z} \to \mathbb{R}$ is called the *autocovariance function* of the process.

Throughout this lecture, we will work exclusively with zero-mean (i.e., $\mu = 0$) covariance stationary processes.

The zero-mean assumption costs nothing in terms of generality since working with non-zero-mean processes involves no more than adding a constant.

29.2.2 Example 1: White Noise

Perhaps the simplest class of covariance stationary processes is the white noise processes.

A process $\{\epsilon_t\}$ is called a *white noise process* if

1. $\mathbb{E}\epsilon_t = 0$
2. $\gamma(k) = \sigma^2 \mathbf{1}\{k = 0\}$ for some $\sigma > 0$

(Here $\mathbf{1}\{k = 0\}$ is defined to be 1 if $k = 0$ and zero otherwise)

White noise processes play the role of **building blocks** for processes with more complicated dynamics.

29.2.3 Example 2: General Linear Processes

From the simple building block provided by white noise, we can construct a very flexible family of covariance stationary processes — the *general linear processes*

$$X_t = \sum_{j=0}^{\infty} \psi_j \epsilon_{t-j}, \qquad t \in \mathbb{Z} \tag{29.1}$$

where

- $\{\epsilon_t\}$ is white noise
- $\{\psi_t\}$ is a square summable sequence in $\mathbb{R}$ (that is, $\sum_{t=0}^{\infty} \psi_t^2 < \infty$)

The sequence $\{\psi_t\}$ is often called a *linear filter*.

Equation (29.1) is said to present a **moving average** process or a moving average representation.

With some manipulations, it is possible to confirm that the autocovariance function for (29.1) is

$$\gamma(k) = \sigma^2 \sum_{j=0}^{\infty} \psi_j \psi_{j+k} \tag{29.2}$$

By the Cauchy-Schwartz inequality, one can show that $\gamma(k)$ satisfies equation (29.2).

Evidently, $\gamma(k)$ does not depend on t.

29.2.4 Wold Representation

Remarkably, the class of general linear processes goes a long way towards describing the entire class of zero-mean covariance stationary processes.

In particular, Wold's decomposition theorem states that every zero-mean covariance stationary process $\{X_t\}$ can be written as

$$X_t = \sum_{j=0}^{\infty} \psi_j \epsilon_{t-j} + \eta_t$$

where

- $\{\epsilon_t\}$ is white noise
- $\{\psi_t\}$ is square summable
- $\psi_0 \epsilon_t$ is the one-step ahead prediction error in forecasting X_t as a linear least-squares function of the infinite history $X_{t-1}, X_{t-2}, \ldots$

- η_t can be expressed as a linear function of $X_{t-1}, X_{t-2}, \ldots$ and is perfectly predictable over arbitrarily long horizons

For the method of constructing a Wold representation, intuition, and further discussion, see [Sargent, 1987], p. 286.

29.2.5 AR and MA

General linear processes are a very broad class of processes.

It often pays to specialize to those for which there exists a representation having only finitely many parameters.

(Experience and theory combine to indicate that models with a relatively small number of parameters typically perform better than larger models, especially for forecasting)

One very simple example of such a model is the first-order autoregressive or AR(1) process

$$X_t = \phi X_{t-1} + \epsilon_t \quad \text{where} \quad |\phi| < 1 \quad \text{and } \{\epsilon_t\} \text{ is white noise} \tag{29.3}$$

By direct substitution, it is easy to verify that $X_t = \sum_{j=0}^{\infty} \phi^j \epsilon_{t-j}$.

Hence $\{X_t\}$ is a general linear process.

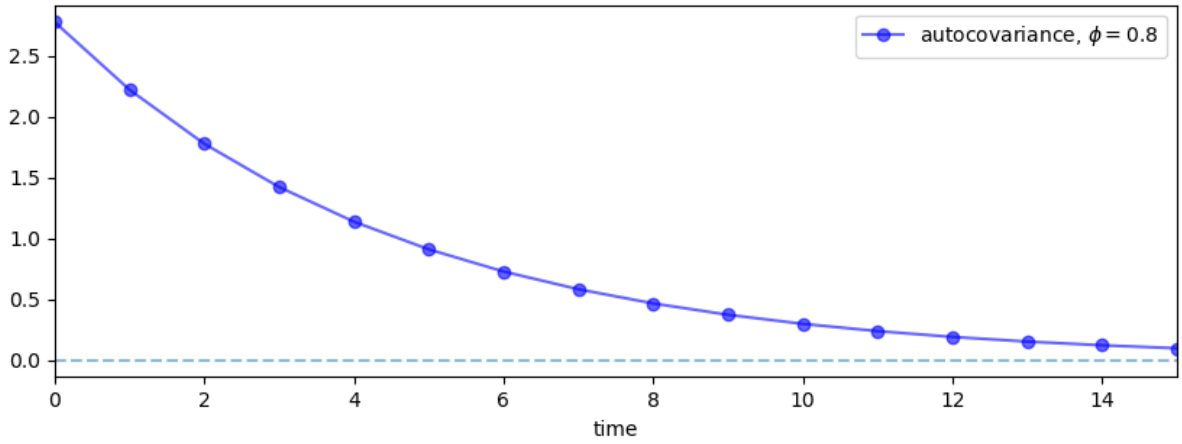
Applying (29.2) to the previous expression for X_t, we get the AR(1) autocovariance function

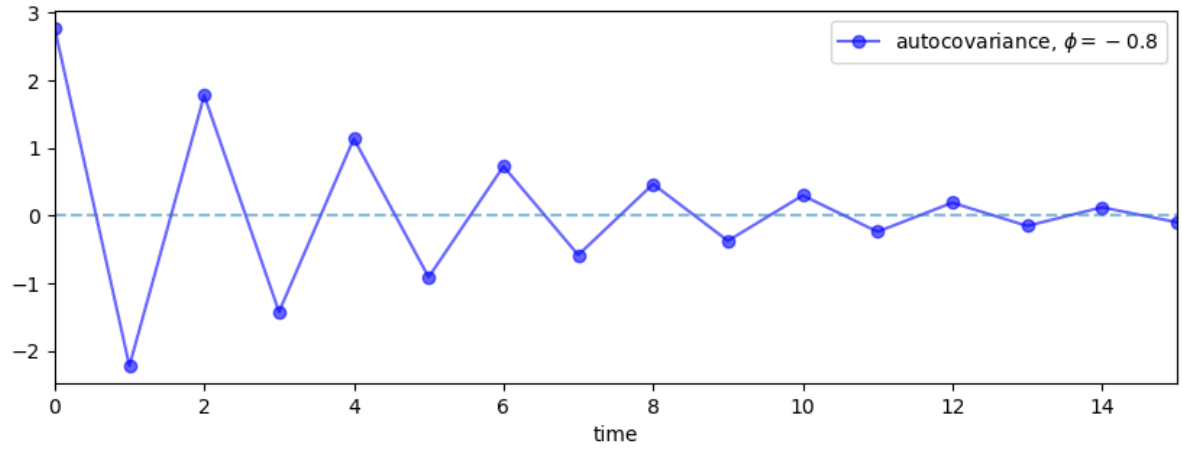
$$\gamma(k) = \phi^k \frac{\sigma^2}{1-\phi^2}, \qquad k = 0, 1, \ldots \tag{29.4}$$

The next figure plots an example of this function for $\phi = 0.8$ and $\phi = -0.8$ with $\sigma = 1$.

```
num_rows, num_cols = 2, 1
fig, axes = plt.subplots(num_rows, num_cols, figsize=(10, 8))
plt.subplots_adjust(hspace=0.4)

for i, ϕ in enumerate((0.8, -0.8)):
    ax = axes[i]
    times = list(range(16))
    acov = [ϕ**k / (1 - ϕ**2) for k in times]
    ax.plot(times, acov, 'bo-', alpha=0.6,
            label=fr'autocovariance, $\phi = {ϕ:.2}$')
    ax.legend(loc='upper right')
    ax.set(xlabel='time', xlim=(0, 15))
    ax.hlines(0, 0, 15, linestyle='--', alpha=0.5)
plt.show()
```





Another very simple process is the MA(1) process (here MA means "moving average")

$$X_t = \epsilon_t + \theta \epsilon_{t-1}$$

You will be able to verify that

$$\gamma(0) = \sigma^2(1 + \theta^2), \quad \gamma(1) = \sigma^2\theta, \quad \text{and} \quad \gamma(k) = 0 \quad \forall\, k > 1$$

The AR(1) can be generalized to an AR(p) and likewise for the MA(1).

Putting all of this together, we get the

29.2.6 ARMA Processes

A stochastic process $\{X_t\}$ is called an *autoregressive moving average process*, or ARMA(p, q), if it can be written as

$$X_t = \phi_1 X_{t-1} + \cdots + \phi_p X_{t-p} + \epsilon_t + \theta_1 \epsilon_{t-1} + \cdots + \theta_q \epsilon_{t-q} \tag{29.5}$$

where $\{\epsilon_t\}$ is white noise.

An alternative notation for ARMA processes uses the *lag operator* L.

Def. Given arbitrary variable Y_t, let $L^k Y_t := Y_{t-k}$.

It turns out that

- lag operators facilitate succinct representations for linear stochastic processes
- algebraic manipulations that treat the lag operator as an ordinary scalar are legitimate

Using L, we can rewrite (29.5) as

$$L^0 X_t - \phi_1 L^1 X_t - \cdots - \phi_p L^p X_t = L^0 \epsilon_t + \theta_1 L^1 \epsilon_t + \cdots + \theta_q L^q \epsilon_t \tag{29.6}$$

If we let $\phi(z)$ and $\theta(z)$ be the polynomials

$$\phi(z) := 1 - \phi_1 z - \cdots - \phi_p z^p \quad \text{and} \quad \theta(z) := 1 + \theta_1 z + \cdots + \theta_q z^q \tag{29.7}$$

then (29.6) becomes

$$\phi(L) X_t = \theta(L) \epsilon_t \tag{29.8}$$

In what follows we **always assume** that the roots of the polynomial $\phi(z)$ lie outside the unit circle in the complex plane.

This condition is sufficient to guarantee that the ARMA(p, q) process is covariance stationary.

In fact, it implies that the process falls within the class of general linear processes *described above*.

That is, given an ARMA(p, q) process $\{X_t\}$ satisfying the unit circle condition, there exists a square summable sequence $\{\psi_t\}$ with $X_t = \sum_{j=0}^{\infty} \psi_j \epsilon_{t-j}$ for all t.

The sequence $\{\psi_t\}$ can be obtained by a recursive procedure outlined on page 79 of [Cryer and Chan, 2008].

The function $t \mapsto \psi_t$ is often called the *impulse response function*.

29.3 Spectral Analysis

Autocovariance functions provide a great deal of information about covariance stationary processes.

In fact, for zero-mean Gaussian processes, the autocovariance function characterizes the entire joint distribution.

Even for non-Gaussian processes, it provides a significant amount of information.

It turns out that there is an alternative representation of the autocovariance function of a covariance stationary process, called the *spectral density*.

At times, the spectral density is easier to derive, easier to manipulate, and provides additional intuition.

29.3.1 Complex Numbers

Before discussing the spectral density, we invite you to recall the main properties of complex numbers (or *skip to the next section*).

It can be helpful to remember that, in a formal sense, complex numbers are just points $(x, y) \in \mathbb{R}^2$ endowed with a specific notion of multiplication.

When (x, y) is regarded as a complex number, x is called the *real part* and y is called the *imaginary part*.

The *modulus* or *absolute value* of a complex number $z = (x, y)$ is just its Euclidean norm in $\mathbb{R}^2$, but is usually written as $|z|$ instead of $\|z\|$.

The product of two complex numbers (x, y) and (u, v) is defined to be $(xu - vy, xv + yu)$, while addition is standard pointwise vector addition.

When endowed with these notions of multiplication and addition, the set of complex numbers forms a field — addition and multiplication play well together, just as they do in $\mathbb{R}$.

The complex number (x, y) is often written as $x + iy$, where i is called the *imaginary unit* and is understood to obey $i^2 = -1$.

The $x + iy$ notation provides an easy way to remember the definition of multiplication given above, because, proceeding naively,

$$(x + iy)(u + iv) = xu - yv + i(xv + yu)$$

Converted back to our first notation, this becomes $(xu - vy, xv + yu)$ as promised.

Complex numbers can be represented in the polar form $re^{i\omega}$ where

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

where $x = r \cos(\omega), y = r \sin(\omega)$, and $\omega = \arctan(y/z)$ or $\tan(\omega) = y/x$.

29.3.2 Spectral Densities

Let $\{X_t\}$ be a covariance stationary process with autocovariance function γ satisfying $\sum_k \gamma(k)^2 < \infty$.

The *spectral density* f of $\{X_t\}$ is defined as the discrete time Fourier transform of its autocovariance function γ.

$$f(\omega) := \sum_{k \in \mathbb{Z}} \gamma(k) e^{-i\omega k}, \qquad \omega \in \mathbb{R}$$

(Some authors normalize the expression on the right by constants such as $1/\pi$ — the convention chosen makes little difference provided you are consistent).

Using the fact that γ is *even*, in the sense that $\gamma(t) = \gamma(-t)$ for all t, we can show that

$$f(\omega) = \gamma(0) + 2 \sum_{k \geq 1} \gamma(k) \cos(\omega k) \tag{29.9}$$

It is not difficult to confirm that f is

- real-valued
- even ($f(\omega) = f(-\omega)$), and
- 2π-periodic, in the sense that $f(2\pi + \omega) = f(\omega)$ for all ω

It follows that the values of f on $[0, \pi]$ determine the values of f on all of $\mathbb{R}$ — the proof is an exercise.

For this reason, it is standard to plot the spectral density only on the interval $[0, \pi]$.

29.3.3 Example 1: White Noise

Consider a white noise process $\{\epsilon_t\}$ with standard deviation σ.

It is easy to check that in this case $f(\omega) = \sigma^2$. So f is a constant function.

As we will see, this can be interpreted as meaning that "all frequencies are equally present".

(White light has this property when frequency refers to the visible spectrum, a connection that provides the origins of the term "white noise")

29.3.4 Example 2: AR and MA and ARMA

It is an exercise to show that the MA(1) process $X_t = \theta\epsilon_{t-1} + \epsilon_t$ has a spectral density

$$f(\omega) = \sigma^2(1 + 2\theta\cos(\omega) + \theta^2) \tag{29.10}$$

With a bit more effort, it's possible to show (see, e.g., p. 261 of [Sargent, 1987]) that the spectral density of the AR(1) process $X_t = \phi X_{t-1} + \epsilon_t$ is

$$f(\omega) = \frac{\sigma^2}{1 - 2\phi\cos(\omega) + \phi^2} \tag{29.11}$$

More generally, it can be shown that the spectral density of the ARMA process (29.5) is

$$f(\omega) = \left|\frac{\theta(e^{i\omega})}{\phi(e^{i\omega})}\right|^2 \sigma^2 \tag{29.12}$$

where

- σ is the standard deviation of the white noise process $\{\epsilon_t\}$.
- the polynomials $\phi(\cdot)$ and $\theta(\cdot)$ are as defined in (29.7).

The derivation of (29.12) uses the fact that convolutions become products under Fourier transformations.

The proof is elegant and can be found in many places — see, for example, [Sargent, 1987], chapter 11, section 4.

It's a nice exercise to verify that (29.10) and (29.11) are indeed special cases of (29.12).

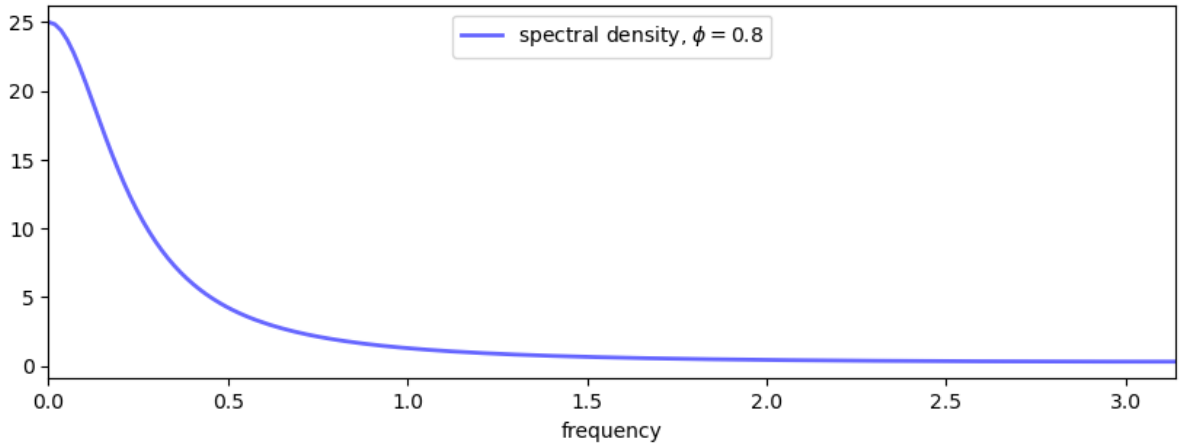
29.3.5 Interpreting the Spectral Density

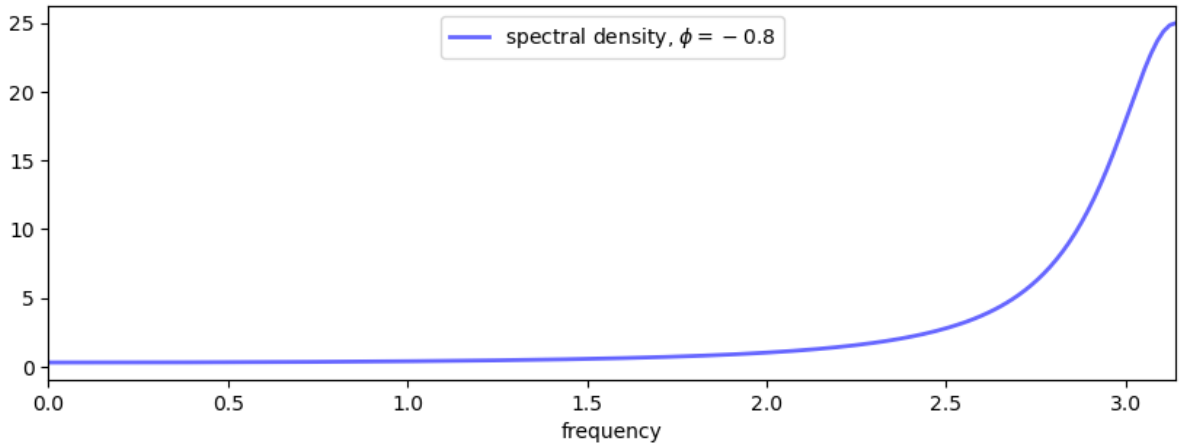
Plotting (29.11) reveals the shape of the spectral density for the AR(1) model when ϕ takes the values 0.8 and -0.8 respectively.

```
def ar1_sd(ϕ, ω):
    return 1 / (1 - 2 * ϕ * np.cos(ω) + ϕ**2)

ωs = np.linspace(0, np.pi, 180)
num_rows, num_cols = 2, 1
fig, axes = plt.subplots(num_rows, num_cols, figsize=(10, 8))
plt.subplots_adjust(hspace=0.4)

# Autocovariance when phi = 0.8
for i, ϕ in enumerate((0.8, -0.8)):
    ax = axes[i]
    sd = ar1_sd(ϕ, ωs)
    ax.plot(ωs, sd, 'b-', alpha=0.6, lw=2,
            label=fr'spectral density, $\phi = {ϕ:.2}$')
    ax.legend(loc='upper center')
    ax.set(xlabel='frequency', xlim=(0, np.pi))
plt.show()
```





These spectral densities correspond to the autocovariance functions for the AR(1) process shown above.

Informally, we think of the spectral density as being large at those $\omega \in [0, \pi]$ at which the autocovariance function seems approximately to exhibit big damped cycles.

To see the idea, let's consider why, in the lower panel of the preceding figure, the spectral density for the case $\phi = -0.8$ is large at $\omega = \pi$.

Recall that the spectral density can be expressed as

$$f(\omega) = \gamma(0) + 2 \sum_{k \geq 1} \gamma(k) \cos(\omega k) = \gamma(0) + 2 \sum_{k \geq 1} (-0.8)^k \cos(\omega k) \tag{29.13}$$

When we evaluate this at $\omega = \pi$, we get a large number because $\cos(\pi k)$ is large and positive when $(-0.8)^k$ is positive, and large in absolute value and negative when $(-0.8)^k$ is negative.

Hence the product is always large and positive, and hence the sum of the products on the right-hand side of (29.13) is large.

These ideas are illustrated in the next figure, which has k on the horizontal axis.

```
ϕ = -0.8
times = list(range(16))
y1 = [ϕ**k / (1 - ϕ**2) for k in times]
y2 = [np.cos(np.pi * k) for k in times]
```

(continues on next page)

(continued from previous page)

```
y3 = [a * b for a, b in zip(y1, y2)]

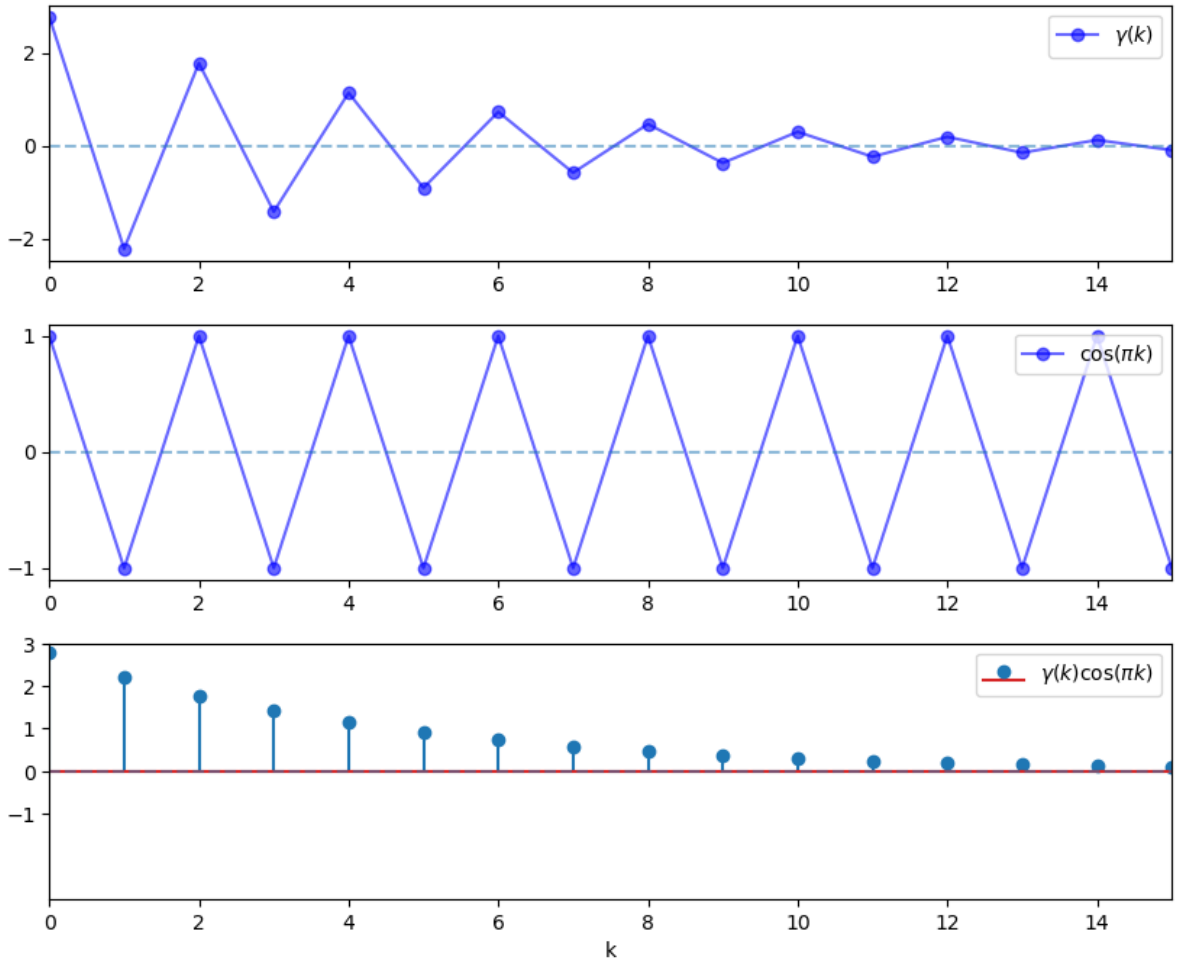
num_rows, num_cols = 3, 1
fig, axes = plt.subplots(num_rows, num_cols, figsize=(10, 8))
plt.subplots_adjust(hspace=0.25)

# Autocovariance when ϕ = -0.8
ax = axes[0]
ax.plot(times, y1, 'bo-', alpha=0.6, label=r'$\gamma(k)$')
ax.legend(loc='upper right')
ax.set(xlim=(0, 15), yticks=(-2, 0, 2))
ax.hlines(0, 0, 15, linestyle='--', alpha=0.5)

# Cycles at frequency π
ax = axes[1]
ax.plot(times, y2, 'bo-', alpha=0.6, label=r'$\cos(\pi k)$')
ax.legend(loc='upper right')
ax.set(xlim=(0, 15), yticks=(-1, 0, 1))
ax.hlines(0, 0, 15, linestyle='--', alpha=0.5)

# Product
ax = axes[2]
ax.stem(times, y3, label=r'$\gamma(k) \cos(\pi k)$')
ax.legend(loc='upper right')
ax.set(xlim=(0, 15), ylim=(-3, 3), yticks=(-1, 0, 1, 2, 3))
ax.hlines(0, 0, 15, linestyle='--', alpha=0.5)
ax.set_xlabel("k")

plt.show()
```



On the other hand, if we evaluate $f(\omega)$ at $\omega = \pi/3$, then the cycles are not matched, the sequence $\gamma(k)\cos(\omega k)$ contains both positive and negative terms, and hence the sum of these terms is much smaller.

```
ϕ = -0.8
times = list(range(16))
y1 = [ϕ**k / (1 - ϕ**2) for k in times]
y2 = [np.cos(np.pi * k/3) for k in times]
y3 = [a * b for a, b in zip(y1, y2)]

num_rows, num_cols = 3, 1
fig, axes = plt.subplots(num_rows, num_cols, figsize=(10, 8))
plt.subplots_adjust(hspace=0.25)

# Autocovariance when phi = -0.8
ax = axes[0]
ax.plot(times, y1, 'bo-', alpha=0.6, label=r'$\gamma(k)$')
ax.legend(loc='upper right')
ax.set(xlim=(0, 15), yticks=(-2, 0, 2))
ax.hlines(0, 0, 15, linestyle='--', alpha=0.5)

# Cycles at frequency π
ax = axes[1]
ax.plot(times, y2, 'bo-', alpha=0.6, label=r'$\cos(\pi k/3)$')
```

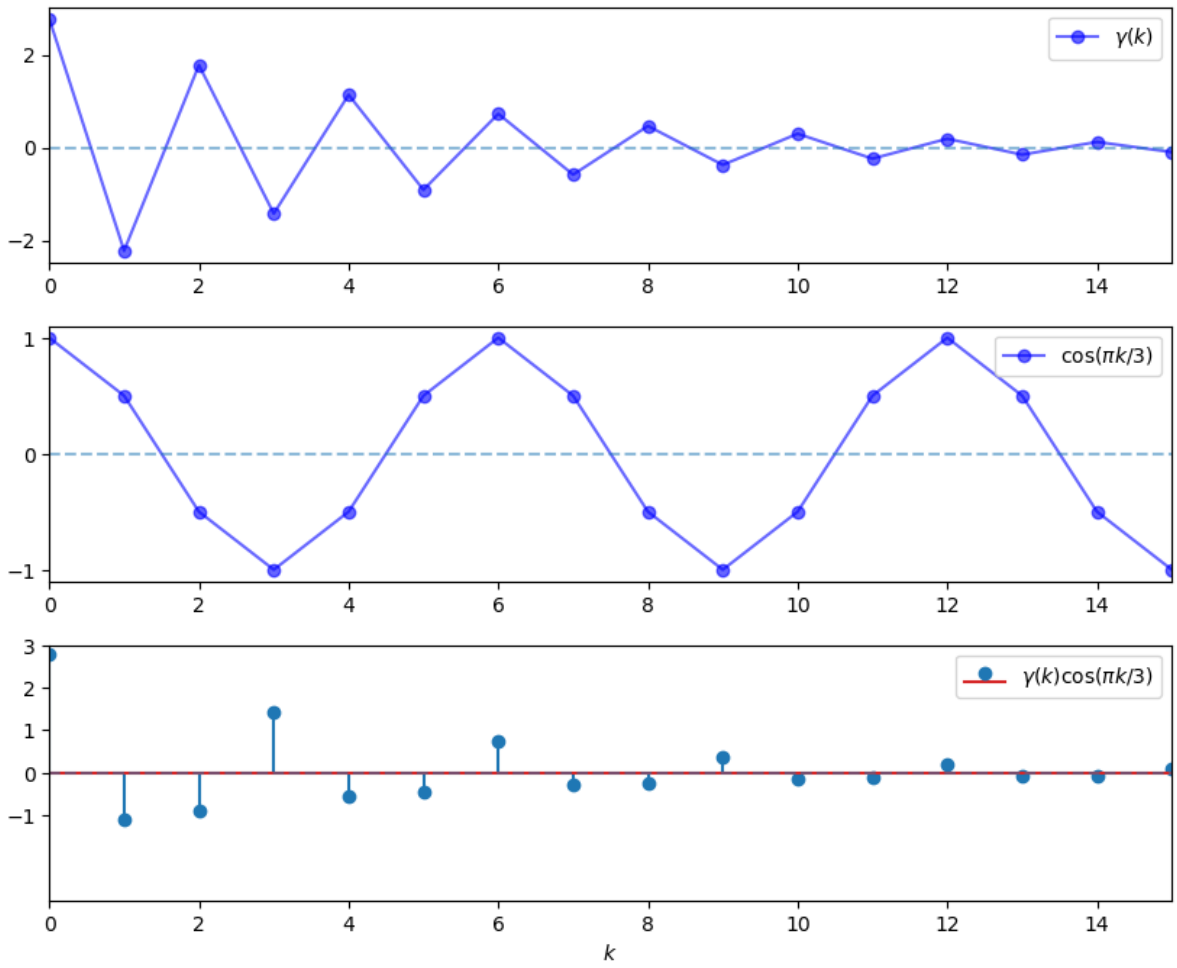
(continues on next page)

(continued from previous page)

```
ax.legend(loc='upper right')
ax.set(xlim=(0, 15), yticks=(-1, 0, 1))
ax.hlines(0, 0, 15, linestyle='--', alpha=0.5)

# Product
ax = axes[2]
ax.stem(times, y3, label=r'$\gamma(k) \cos(\pi k/3)$')
ax.legend(loc='upper right')
ax.set(xlim=(0, 15), ylim=(-3, 3), yticks=(-1, 0, 1, 2, 3))
ax.hlines(0, 0, 15, linestyle='--', alpha=0.5)
ax.set_xlabel("$k$")

plt.show()
```



In summary, the spectral density is large at frequencies ω where the autocovariance function exhibits damped cycles.

29.3.6 Inverting the Transformation

We have just seen that the spectral density is useful in the sense that it provides a frequency-based perspective on the autocovariance structure of a covariance stationary process.

Another reason that the spectral density is useful is that it can be "inverted" to recover the autocovariance function via the *inverse Fourier transform*.

In particular, for all $k \in \mathbb{Z}$, we have

$$\gamma(k) = \frac{1}{2\pi} \int_{-\pi}^{\pi} f(\omega) e^{i\omega k} d\omega \tag{29.14}$$

This is convenient in situations where the spectral density is easier to calculate and manipulate than the autocovariance function.

(For example, the expression (29.12) for the ARMA spectral density is much easier to work with than the expression for the ARMA autocovariance)

29.3.7 Mathematical Theory

This section is loosely based on [Sargent, 1987], p. 249-253, and included for those who

- would like a bit more insight into spectral densities
- and have at least some background in Hilbert space theory

Others should feel free to skip to the *next section* — none of this material is necessary to progress to computation.

Recall that every separable Hilbert space H has a countable orthonormal basis $\{h_k\}$.

The nice thing about such a basis is that every $f \in H$ satisfies

$$f = \sum_k \alpha_k h_k \quad \text{where} \quad \alpha_k := \langle f, h_k \rangle \tag{29.15}$$

where $\langle \cdot, \cdot \rangle$ denotes the inner product in H.

Thus, f can be represented to any degree of precision by linearly combining basis vectors.

The scalar sequence $\alpha = \{\alpha_k\}$ is called the *Fourier coefficients* of f, and satisfies $\sum_k |\alpha_k|^2 < \infty$.

In other words, α is in ℓ_2, the set of square summable sequences.

Consider an operator T that maps $\alpha \in \ell_2$ into its expansion $\sum_k \alpha_k h_k \in H$.

The Fourier coefficients of $T\alpha$ are just $\alpha = \{\alpha_k\}$, as you can verify by confirming that $\langle T\alpha, h_k \rangle = \alpha_k$.

Using elementary results from Hilbert space theory, it can be shown that

- T is one-to-one — if α and β are distinct in ℓ_2, then so are their expansions in H.
- T is onto — if $f \in H$ then its preimage in ℓ_2 is the sequence α given by $\alpha_k = \langle f, h_k \rangle$.
- T is a linear isometry — in particular, $\langle \alpha, \beta \rangle = \langle T\alpha, T\beta \rangle$.

Summarizing these results, we say that any separable Hilbert space is isometrically isomorphic to ℓ_2.

In essence, this says that each separable Hilbert space we consider is just a different way of looking at the fundamental space ℓ_2.

With this in mind, let's specialize to a setting where

- $\gamma \in \ell_2$ is the autocovariance function of a covariance stationary process, and f is the spectral density.

- $H = L_2$, where L_2 is the set of square summable functions on the interval $[-\pi, \pi]$, with inner product $\langle g, h \rangle = \int_{-\pi}^{\pi} g(\omega) h(\omega) d\omega$.
- $\{h_k\} =$ the orthonormal basis for L_2 given by the set of trigonometric functions.

$$h_k(\omega) = \frac{e^{i\omega k}}{\sqrt{2\pi}}, \quad k \in \mathbb{Z}, \quad \omega \in [-\pi, \pi]$$

Using the definition of T from above and the fact that f is even, we now have

$$T\gamma = \sum_{k \in \mathbb{Z}} \gamma(k) \frac{e^{i\omega k}}{\sqrt{2\pi}} = \frac{1}{\sqrt{2\pi}} f(\omega) \tag{29.16}$$

In other words, apart from a scalar multiple, the spectral density is just a transformation of $\gamma \in \ell_2$ under a certain linear isometry — a different way to view γ.

In particular, it is an expansion of the autocovariance function with respect to the trigonometric basis functions in L_2.

As discussed above, the Fourier coefficients of $T\gamma$ are given by the sequence γ, and, in particular, $\gamma(k) = \langle T\gamma, h_k \rangle$.

Transforming this inner product into its integral expression and using (29.16) gives (29.14), justifying our earlier expression for the inverse transform.

29.4 Implementation

Most code for working with covariance stationary models deals with ARMA models.

Python code for studying ARMA models can be found in the `tsa` submodule of statsmodels.

Since this code doesn't quite cover our needs — particularly vis-a-vis spectral analysis — we've put together the module arma.py, which is part of QuantEcon.py package.

The module provides functions for mapping ARMA(p, q) models into their

1. impulse response function
2. simulated time series
3. autocovariance function
4. spectral density

29.4.1 Application

Let's use this code to replicate the plots on pages 68–69 of [Ljungqvist and Sargent, 2018].

Here are some functions to generate the plots

```
def plot_impulse_response(arma, ax=None):
    if ax is None:
        ax = plt.gca()
    yi = arma.impulse_response()
    ax.stem(list(range(len(yi))), yi)
    ax.set(xlim=(-0.5), ylim=(min(yi)-0.1, max(yi)+0.1),
                title='Impulse response', xlabel='time', ylabel='response')
    return ax

def plot_spectral_density(arma, ax=None):
```

(continues on next page)

(continued from previous page)

```
    if ax is None:
        ax = plt.gca()
    w, spect = arma.spectral_density(two_pi=False)
    ax.semilogy(w, spect)
    ax.set(xlim=(0, np.pi), ylim=(0, np.max(spect)),
           title='Spectral density', xlabel='frequency', ylabel='spectrum')
    return ax

def plot_autocovariance(arma, ax=None):
    if ax is None:
        ax = plt.gca()
    acov = arma.autocovariance()
    ax.stem(list(range(len(acov))), acov)
    ax.set(xlim=(-0.5, len(acov) - 0.5), title='Autocovariance',
           xlabel='time', ylabel='autocovariance')
    return ax

def plot_simulation(arma, ax=None):
    if ax is None:
        ax = plt.gca()
    x_out = arma.simulation()
    ax.plot(x_out)
    ax.set(title='Sample path', xlabel='time', ylabel='state space')
    return ax

def quad_plot(arma):
    """
    Plots the impulse response, spectral_density, autocovariance,
    and one realization of the process.

    """
    num_rows, num_cols = 2, 2
    fig, axes = plt.subplots(num_rows, num_cols, figsize=(10, 7))
    plot_functions = [plot_impulse_response,
                      plot_spectral_density,
                      plot_autocovariance,
                      plot_simulation]
    for plot_func, ax in zip(plot_functions, axes.flatten()):
        plot_func(arma, ax)
    plt.tight_layout()
    plt.show()
```

Now let's call these functions to generate plots.

As a warmup, let's make sure things look right when we for the pure white noise model $X_t = \epsilon_t$.

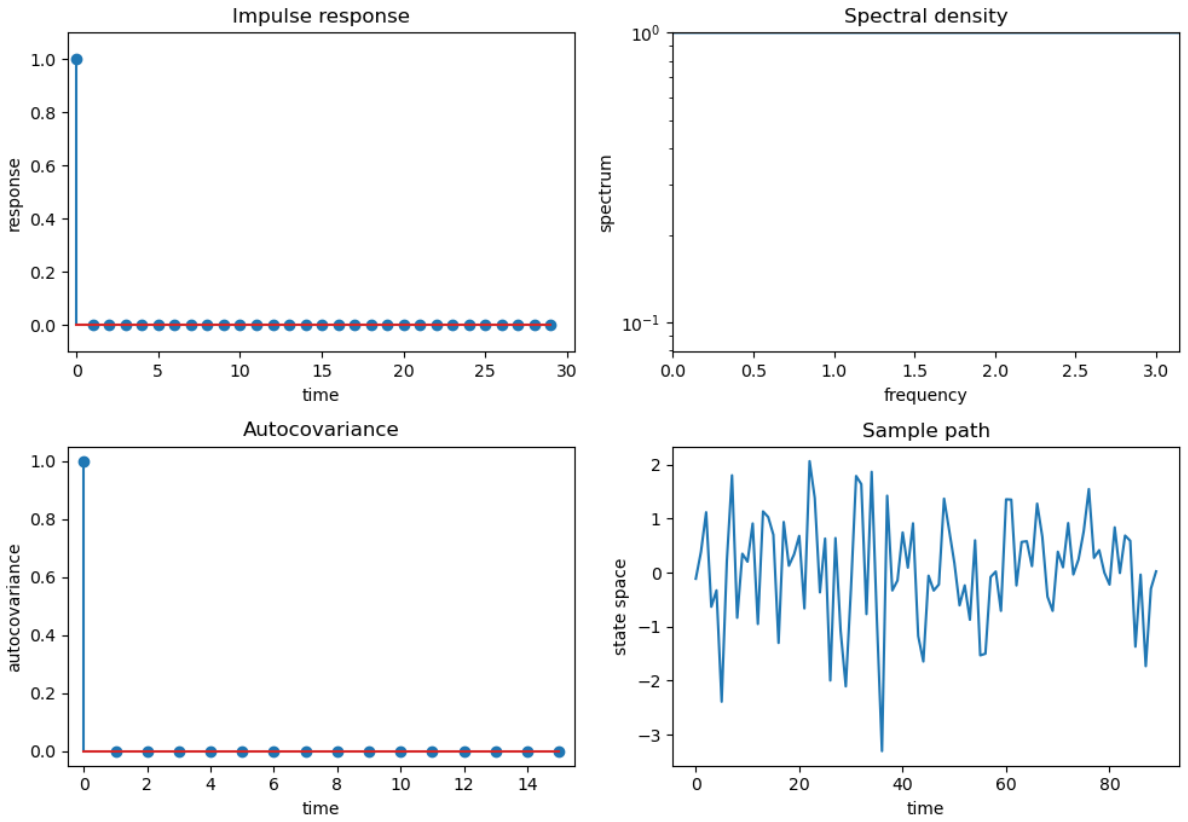
```
ϕ = 0.0
θ = 0.0
arma = qe.ARMA(ϕ, θ)
quad_plot(arma)
```

```
/home/runner/miniconda3/envs/quantecon/lib/python3.12/site-packages/matplotlib/
↪cbook.py:1762: ComplexWarning: Casting complex values to real discards the␣
↪imaginary part
  return math.isfinite(val)
```

(continues on next page)

(continued from previous page)

```
/home/runner/miniconda3/envs/quantecon/lib/python3.12/site-packages/matplotlib/
 ↪cbook.py:1398: ComplexWarning: Casting complex values to real discards the␣
 ↪imaginary part
  return np.asarray(x, float)
/tmp/ipykernel_7065/4271821819.py:15: UserWarning: Attempt to set non-positive␣
 ↪ylim on a log-scaled axis will be ignored.
  ax.set(xlim=(0, np.pi), ylim=(0, np.max(spect)),
/home/runner/miniconda3/envs/quantecon/lib/python3.12/site-packages/matplotlib/
 ↪transforms.py:993: ComplexWarning: Casting complex values to real discards the␣
 ↪imaginary part
  self._points[:, 1] = interval
```



If we look carefully, things look good: the spectrum is the flat line at 10^0 at the very top of the spectrum graphs, which is at it should be.

Also

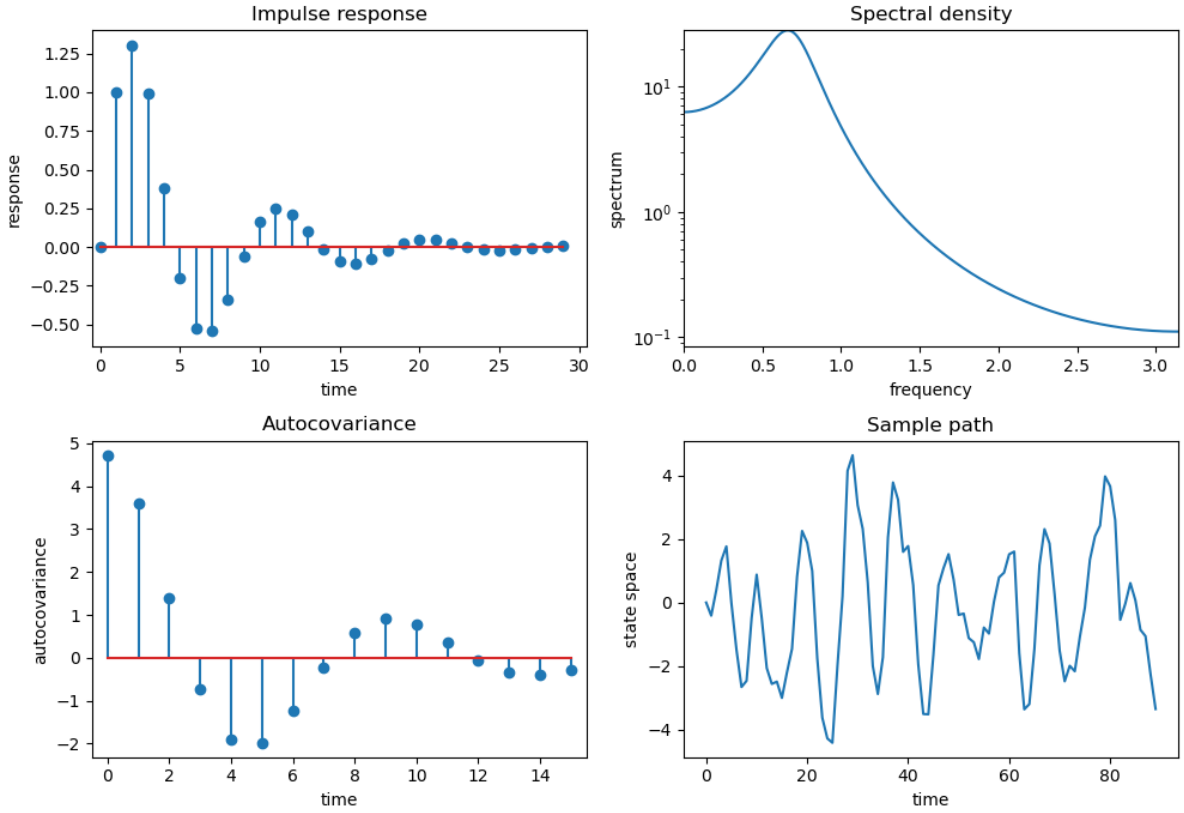
- the variance equals $1 = \frac{1}{2\pi} \int_{-\pi}^{\pi} 1 d\omega$ as it should.
- the covariogram and impulse response look as they should.
- it is actually challenging to visualize a time series realization of white noise – a sequence of surprises – but this too looks pretty good.

To get some more examples, as our laboratory we'll replicate quartets of graphs that [Ljungqvist and Sargent, 2018] use to teach "how to read spectral densities".

Ljunqvist and Sargent's first model is $X_t = 1.3X_{t-1} - .7X_{t-2} + \epsilon_t$

```
ϕ = 1.3, -.7
θ = 0.0
arma = qe.ARMA(ϕ, θ)
quad_plot(arma)
```

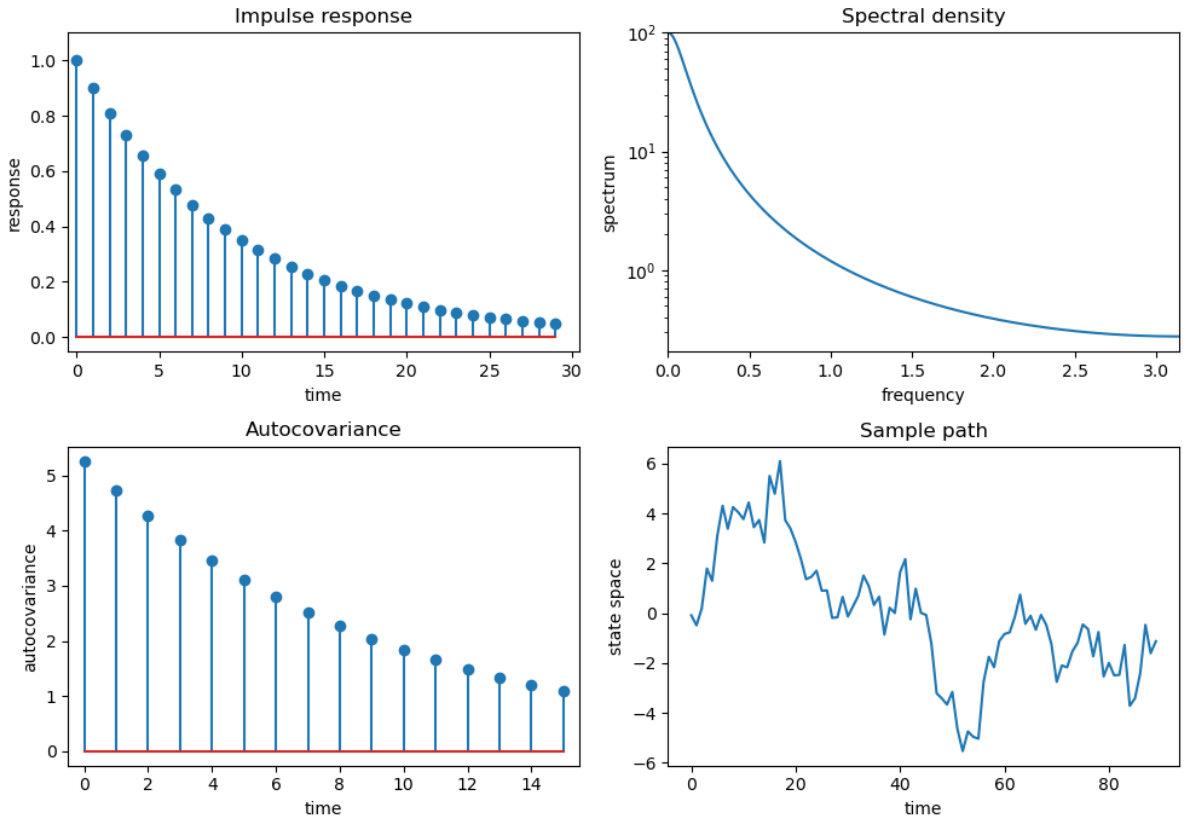
```
/tmp/ipykernel_7065/4271821819.py:15: UserWarning: Attempt to set non-positive␣
↪ylim on a log-scaled axis will be ignored.
  ax.set(xlim=(0, np.pi), ylim=(0, np.max(spect)),
```



Ljungqvist and Sargent's second model is $X_t = .9X_{t-1} + \epsilon_t$

```
ϕ = 0.9
θ = -0.0
arma = qe.ARMA(ϕ, θ)
quad_plot(arma)
```

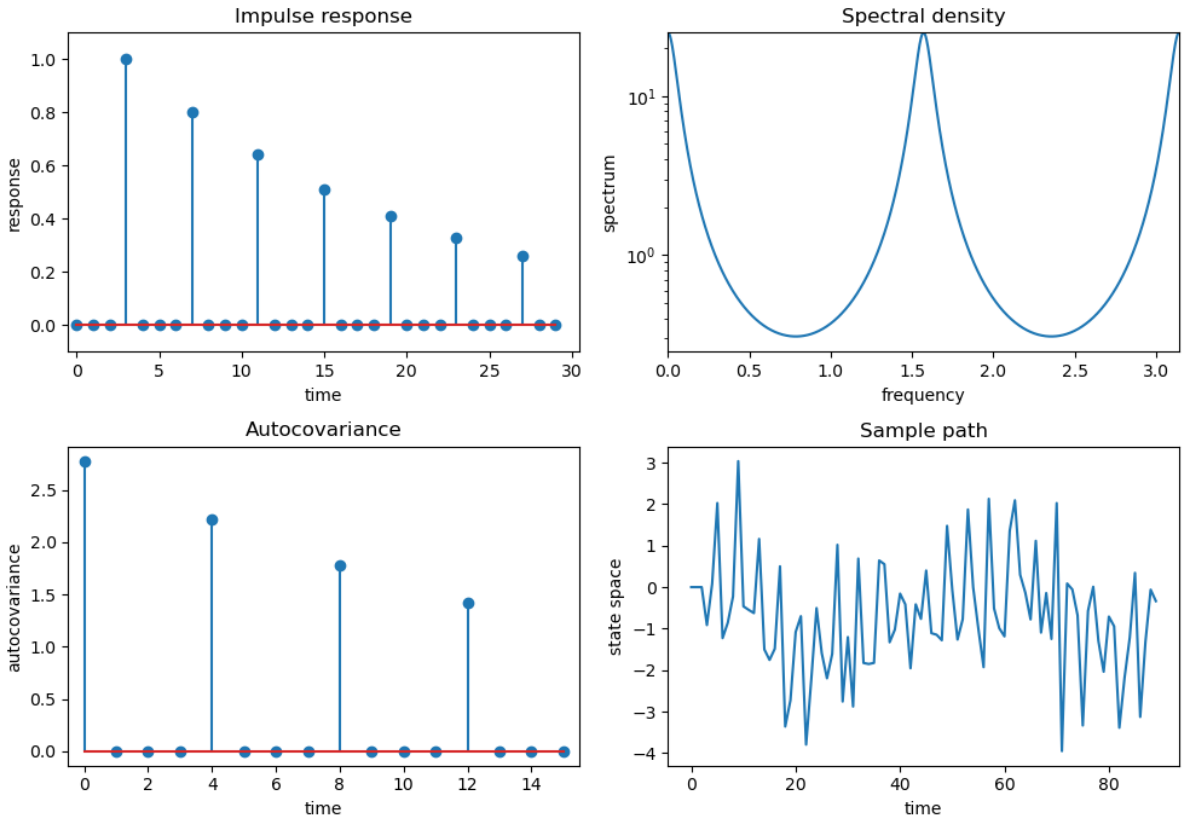
```
/tmp/ipykernel_7065/4271821819.py:15: UserWarning: Attempt to set non-positive␣
↪ylim on a log-scaled axis will be ignored.
  ax.set(xlim=(0, np.pi), ylim=(0, np.max(spect)),
```



Ljungqvist and Sargent's third model is $X_t = .8X_{t-4} + \epsilon_t$

```
ϕ = 0., 0., 0., .8
θ = -0.0
arma = qe.ARMA(ϕ, θ)
quad_plot(arma)
```

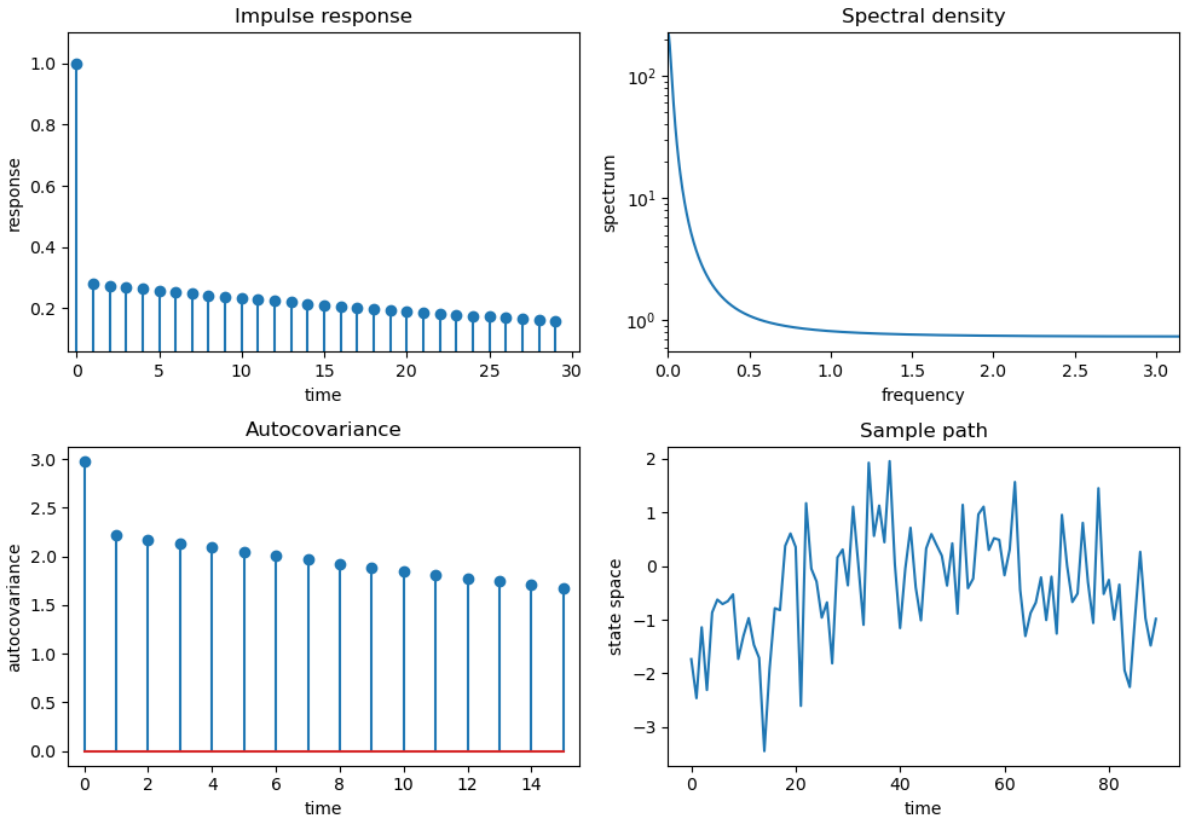
```
/tmp/ipykernel_7065/4271821819.py:15: UserWarning: Attempt to set non-positive␣
↪ylim on a log-scaled axis will be ignored.
  ax.set(xlim=(0, np.pi), ylim=(0, np.max(spect)),
```



Ljungqvist and Sargent's fourth model is $X_t = .98X_{t-1} + \epsilon_t - .7\epsilon_{t-1}$

```
ϕ = .98
θ = -0.7
arma = qe.ARMA(ϕ, θ)
quad_plot(arma)
```

```
/tmp/ipykernel_7065/4271821819.py:15: UserWarning: Attempt to set non-positive␣
↪ylim on a log-scaled axis will be ignored.
  ax.set(xlim=(0, np.pi), ylim=(0, np.max(spect)),
```



29.4.2 Explanation

The call

```
arma = ARMA(ϕ, θ, σ)
```

creates an instance `arma` that represents the ARMA(p, q) model

$$X_t = \phi_1 X_{t-1} + ... + \phi_p X_{t-p} + \epsilon_t + \theta_1 \epsilon_{t-1} + ... + \theta_q \epsilon_{t-q}$$

If `ϕ` and `θ` are arrays or sequences, then the interpretation will be

- `ϕ` holds the vector of parameters $(\phi_1, \phi_2, ..., \phi_p)$.
- `θ` holds the vector of parameters $(\theta_1, \theta_2, ..., \theta_q)$.

The parameter `σ` is always a scalar, the standard deviation of the white noise.

We also permit `ϕ` and `θ` to be scalars, in which case the model will be interpreted as

$$X_t = \phi X_{t-1} + \epsilon_t + \theta \epsilon_{t-1}$$

The two numerical packages most useful for working with ARMA models are `scipy.signal` and `numpy.fft`.

The package `scipy.signal` expects the parameters to be passed into its functions in a manner consistent with the alternative ARMA notation (29.8).

For example, the impulse response sequence $\{\psi_t\}$ discussed above can be obtained using `scipy.signal.dimpulse`, and the function call should be of the form

```
times, ψ = dimpulse((ma_poly, ar_poly, 1), n=impulse_length)
```

where `ma_poly` and `ar_poly` correspond to the polynomials in (29.7) — that is,

- `ma_poly` is the vector $(1, \theta_1, \theta_2, \ldots, \theta_q)$
- `ar_poly` is the vector $(1, -\phi_1, -\phi_2, \ldots, -\phi_p)$

To this end, we also maintain the arrays `ma_poly` and `ar_poly` as instance data, with their values computed automatically from the values of `phi` and `theta` supplied by the user.

If the user decides to change the value of either `theta` or `phi` ex-post by assignments such as `arma.phi = (0.5, 0.2)` or `arma.theta = (0, -0.1)`.

then `ma_poly` and `ar_poly` should update automatically to reflect these new parameters.

This is achieved in our implementation by using descriptors.

29.4.3 Computing the Autocovariance Function

As discussed above, for ARMA processes the spectral density has a *simple representation* that is relatively easy to calculate.

Given this fact, the easiest way to obtain the autocovariance function is to recover it from the spectral density via the inverse Fourier transform.

Here we use NumPy's Fourier transform package np.fft, which wraps a standard Fortran-based package called FFTPACK.

A look at the np.fft documentation shows that the inverse transform np.fft.ifft takes a given sequence $A_0, A_1, \ldots, A_{n-1}$ and returns the sequence $a_0, a_1, \ldots, a_{n-1}$ defined by

$$a_k = \frac{1}{n} \sum_{t=0}^{n-1} A_t e^{ik2\pi t/n}$$

Thus, if we set $A_t = f(\omega_t)$, where f is the spectral density and $\omega_t := 2\pi t/n$, then

$$a_k = \frac{1}{n} \sum_{t=0}^{n-1} f(\omega_t) e^{i\omega_t k} = \frac{1}{2\pi} \frac{2\pi}{n} \sum_{t=0}^{n-1} f(\omega_t) e^{i\omega_t k}, \qquad \omega_t := 2\pi t/n$$

For n sufficiently large, we then have

$$a_k \approx \frac{1}{2\pi} \int_0^{2\pi} f(\omega) e^{i\omega k} d\omega = \frac{1}{2\pi} \int_{-\pi}^{\pi} f(\omega) e^{i\omega k} d\omega$$

(You can check the last equality)

In view of (29.14), we have now shown that, for n sufficiently large, $a_k \approx \gamma(k)$ — which is exactly what we want to compute.

CHAPTER

THIRTY

ESTIMATION OF SPECTRA

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install --upgrade quantecon
```

30.1 Overview

In a *previous lecture*, we covered some fundamental properties of covariance stationary linear stochastic processes.

One objective for that lecture was to introduce spectral densities — a standard and very useful technique for analyzing such processes.

In this lecture, we turn to the problem of estimating spectral densities and other related quantities from data.

Estimates of the spectral density are computed using what is known as a periodogram — which in turn is computed via the famous fast Fourier transform.

Once the basic technique has been explained, we will apply it to the analysis of several key macroeconomic time series.

For supplementary reading, see [Sargent, 1987] or [Cryer and Chan, 2008].

Let's start with some standard imports:

```
import numpy as np
import matplotlib.pyplot as plt
from quantecon import ARMA, periodogram, ar_periodogram
```

30.2 Periodograms

Recall that the spectral density f of a covariance stationary process with autocorrelation function γ can be written

$$f(\omega) = \gamma(0) + 2 \sum_{k \geq 1} \gamma(k) \cos(\omega k), \qquad \omega \in \mathbb{R}$$

Now consider the problem of estimating the spectral density of a given time series, when γ is unknown.

In particular, let $X_0, \ldots, X_{n-1}$ be n consecutive observations of a single time series that is assumed to be covariance stationary.

The most common estimator of the spectral density of this process is the *periodogram* of $X_0, \ldots, X_{n-1}$, which is defined as

$$I(\omega) := \frac{1}{n} \left| \sum_{t=0}^{n-1} X_t e^{it\omega} \right|^2, \qquad \omega \in \mathbb{R} \tag{30.1}$$

(Recall that $|z|$ denotes the modulus of complex number z)

Alternatively, $I(\omega)$ can be expressed as

$$I(\omega) = \frac{1}{n} \left\{ \left[\sum_{t=0}^{n-1} X_t \cos(\omega t) \right]^2 + \left[\sum_{t=0}^{n-1} X_t \sin(\omega t) \right]^2 \right\}$$

It is straightforward to show that the function I is even and 2π-periodic (i.e., $I(\omega) = I(-\omega)$ and $I(\omega + 2\pi) = I(\omega)$ for all $\omega \in \mathbb{R}$).

From these two results, you will be able to verify that the values of I on $[0, \pi]$ determine the values of I on all of $\mathbb{R}$.

The next section helps to explain the connection between the periodogram and the spectral density.

30.2.1 Interpretation

To interpret the periodogram, it is convenient to focus on its values at the *Fourier frequencies*

$$\omega_j := \frac{2\pi j}{n}, \quad j = 0, \ldots, n-1$$

In what sense is $I(\omega_j)$ an estimate of $f(\omega_j)$?

The answer is straightforward, although it does involve some algebra.

With a bit of effort, one can show that for any integer $j > 0$,

$$\sum_{t=0}^{n-1} e^{it\omega_j} = \sum_{t=0}^{n-1} \exp\left\{ i 2\pi j \frac{t}{n} \right\} = 0$$

Letting $\bar{X}$ denote the sample mean $n^{-1} \sum_{t=0}^{n-1} X_t$, we then have

$$nI(\omega_j) = \left| \sum_{t=0}^{n-1} (X_t - \bar{X}) e^{it\omega_j} \right|^2 = \sum_{t=0}^{n-1} (X_t - \bar{X}) e^{it\omega_j} \sum_{r=0}^{n-1} (X_r - \bar{X}) e^{-ir\omega_j}$$

By carefully working through the sums, one can transform this to

$$nI(\omega_j) = \sum_{t=0}^{n-1} (X_t - \bar{X})^2 + 2 \sum_{k=1}^{n-1} \sum_{t=k}^{n-1} (X_t - \bar{X})(X_{t-k} - \bar{X}) \cos(\omega_j k)$$

Now let

$$\hat{\gamma}(k) := \frac{1}{n} \sum_{t=k}^{n-1} (X_t - \bar{X})(X_{t-k} - \bar{X}), \qquad k = 0, 1, \ldots, n-1$$

This is the sample autocovariance function, the natural "plug-in estimator" of the *autocovariance function* γ.

("Plug-in estimator" is an informal term for an estimator found by replacing expectations with sample means)

With this notation, we can now write

$$I(\omega_j) = \hat{\gamma}(0) + 2 \sum_{k=1}^{n-1} \hat{\gamma}(k) \cos(\omega_j k)$$

Recalling our expression for f given *above*, we see that $I(\omega_j)$ is just a sample analog of $f(\omega_j)$.

30.2.2 Calculation

Let's now consider how to compute the periodogram as defined in (30.1).

There are already functions available that will do this for us — an example is `statsmodels.tsa.stattools.periodogram` in the `statsmodels` package.

However, it is very simple to replicate their results, and this will give us a platform to make useful extensions.

The most common way to calculate the periodogram is via the discrete Fourier transform, which in turn is implemented through the fast Fourier transform algorithm.

In general, given a sequence $a_0, \ldots, a_{n-1}$, the discrete Fourier transform computes the sequence

$$A_j := \sum_{t=0}^{n-1} a_t \exp\left\{ i 2\pi \frac{tj}{n} \right\}, \qquad j = 0, \ldots, n-1$$

With `numpy.fft.fft` imported as `fft` and $a_0, \ldots, a_{n-1}$ stored in NumPy array `a`, the function call `fft(a)` returns the values $A_0, \ldots, A_{n-1}$ as a NumPy array.

It follows that when the data $X_0, \ldots, X_{n-1}$ are stored in array `X`, the values $I(\omega_j)$ at the Fourier frequencies, which are given by

$$\frac{1}{n} \left| \sum_{t=0}^{n-1} X_t \exp\left\{ i 2\pi \frac{tj}{n} \right\} \right|^2, \qquad j = 0, \ldots, n-1$$

can be computed by `np.abs(fft(X))**2 / len(X)`.

Note: The NumPy function `abs` acts elementwise, and correctly handles complex numbers (by computing their modulus, which is exactly what we need).

A function called `periodogram` that puts all this together can be found here.

Let's generate some data for this function using the `ARMA` class from QuantEcon.py (see the *lecture on linear processes* for more details).

Here's a code snippet that, once the preceding code has been run, generates data from the process

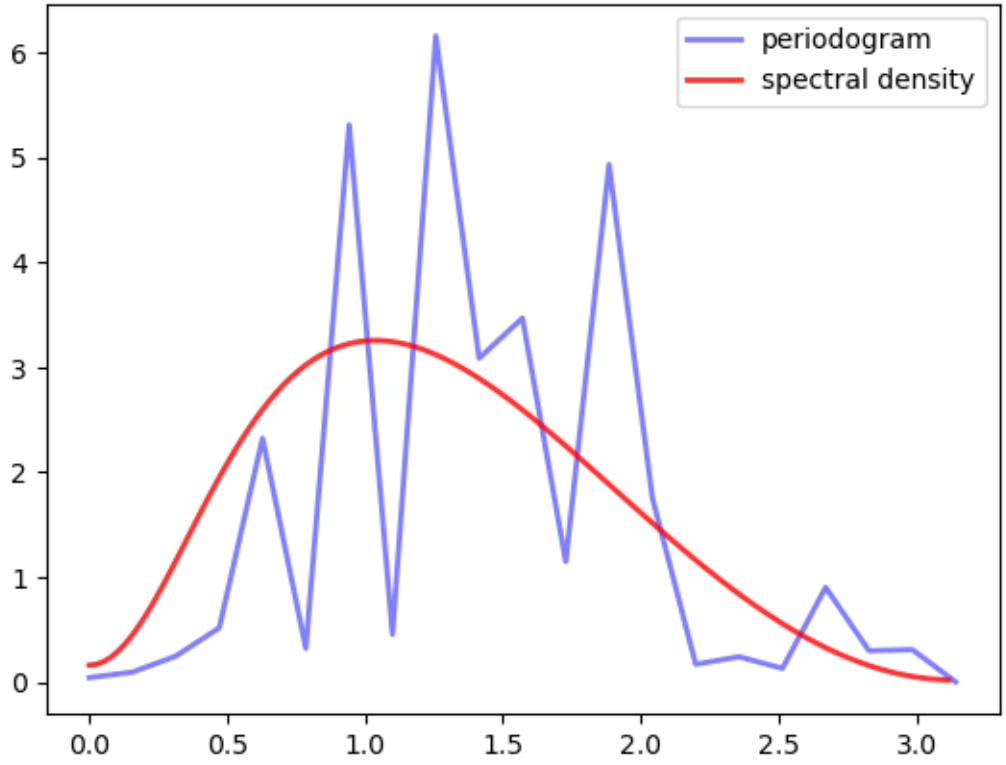
$$X_t = 0.5 X_{t-1} + \epsilon_t - 0.8 \epsilon_{t-2} \tag{30.2}$$

where $\{\epsilon_t\}$ is white noise with unit variance, and compares the periodogram to the actual spectral density

```
n = 40                          # Data size
ϕ, θ = 0.5, (0, -0.8)           # AR and MA parameters
lp = ARMA(ϕ, θ)
X = lp.simulation(ts_length=n)

fig, ax = plt.subplots()
x, y = periodogram(X)
ax.plot(x, y, 'b-', lw=2, alpha=0.5, label='periodogram')
x_sd, y_sd = lp.spectral_density(two_pi=False, res=120)
ax.plot(x_sd, y_sd, 'r-', lw=2, alpha=0.8, label='spectral density')
ax.legend()
plt.show()
```

```
/home/runner/miniconda3/envs/quantecon/lib/python3.12/site-packages/matplotlib/
 ↪cbook.py:1762: ComplexWarning: Casting complex values to real discards the␣
 ↪imaginary part
  return math.isfinite(val)
/home/runner/miniconda3/envs/quantecon/lib/python3.12/site-packages/matplotlib/
 ↪cbook.py:1398: ComplexWarning: Casting complex values to real discards the␣
 ↪imaginary part
  return np.asarray(x, float)
```



This estimate looks rather disappointing, but the data size is only 40, so perhaps it's not surprising that the estimate is poor.

However, if we try again with `n = 1200` the outcome is not much better

The periodogram is far too irregular relative to the underlying spectral density.

This brings us to our next topic.

30.3 Smoothing

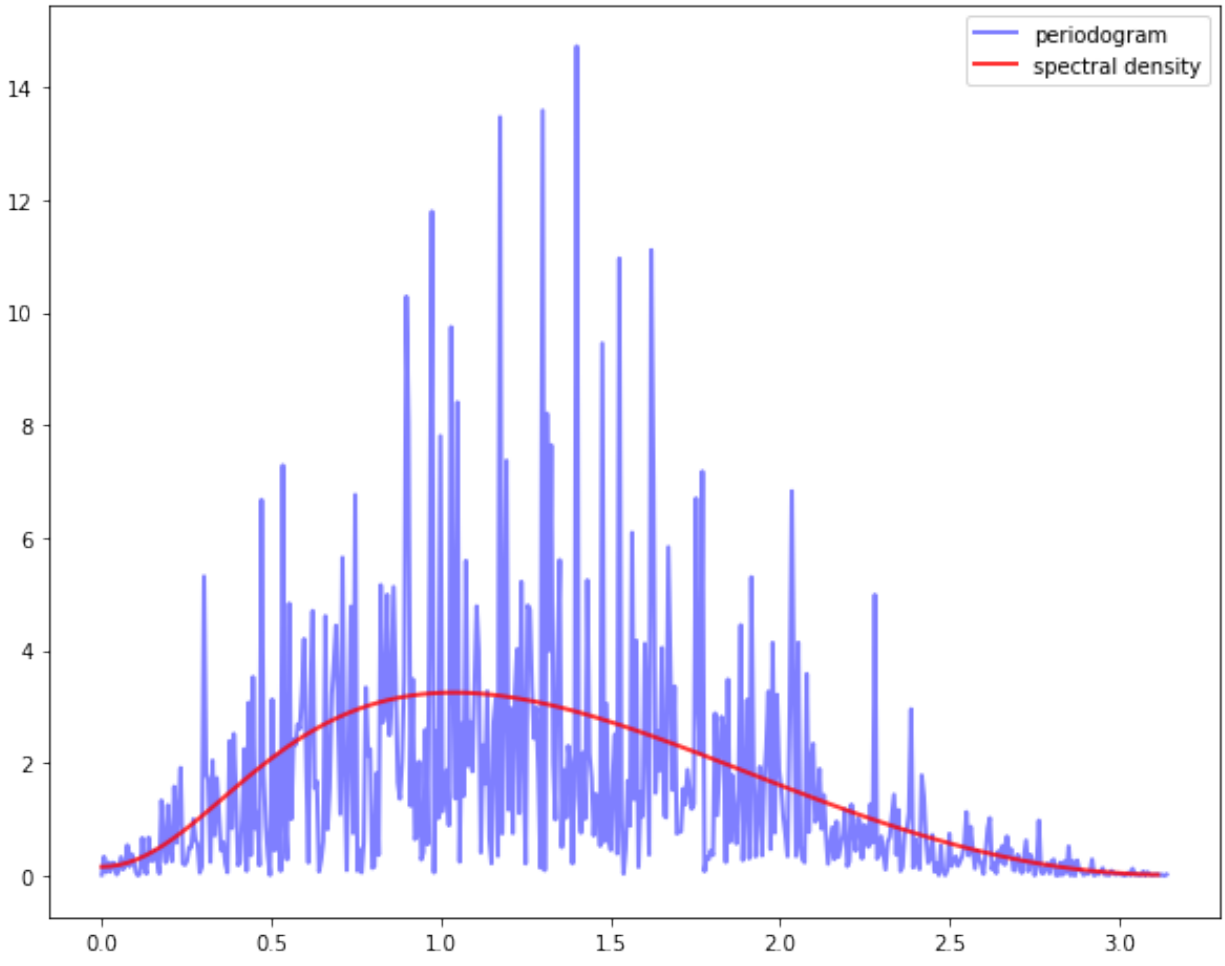
There are two related issues here.

One is that, given the way the fast Fourier transform is implemented, the number of points ω at which $I(\omega)$ is estimated increases in line with the amount of data.

In other words, although we have more data, we are also using it to estimate more values.

A second issue is that densities of all types are fundamentally hard to estimate without parametric assumptions.

Typically, nonparametric estimation of densities requires some degree of smoothing.


periodogram
spectral density
14
12
10
8
6
4
2
0
0.0
0.5
1.0
1.5
2.0
2.5
3.0

The standard way that smoothing is applied to periodograms is by taking local averages.

In other words, the value $I(\omega_j)$ is replaced with a weighted average of the adjacent values

$$I(\omega_{j-p}), I(\omega_{j-p+1}), \ldots, I(\omega_j), \ldots, I(\omega_{j+p})$$

This weighted average can be written as

$$I_S(\omega_j) := \sum_{\ell=-p}^{p} w(\ell) I(\omega_{j+\ell}) \tag{30.3}$$

where the weights $w(-p), \ldots, w(p)$ are a sequence of $2p+1$ nonnegative values summing to one.

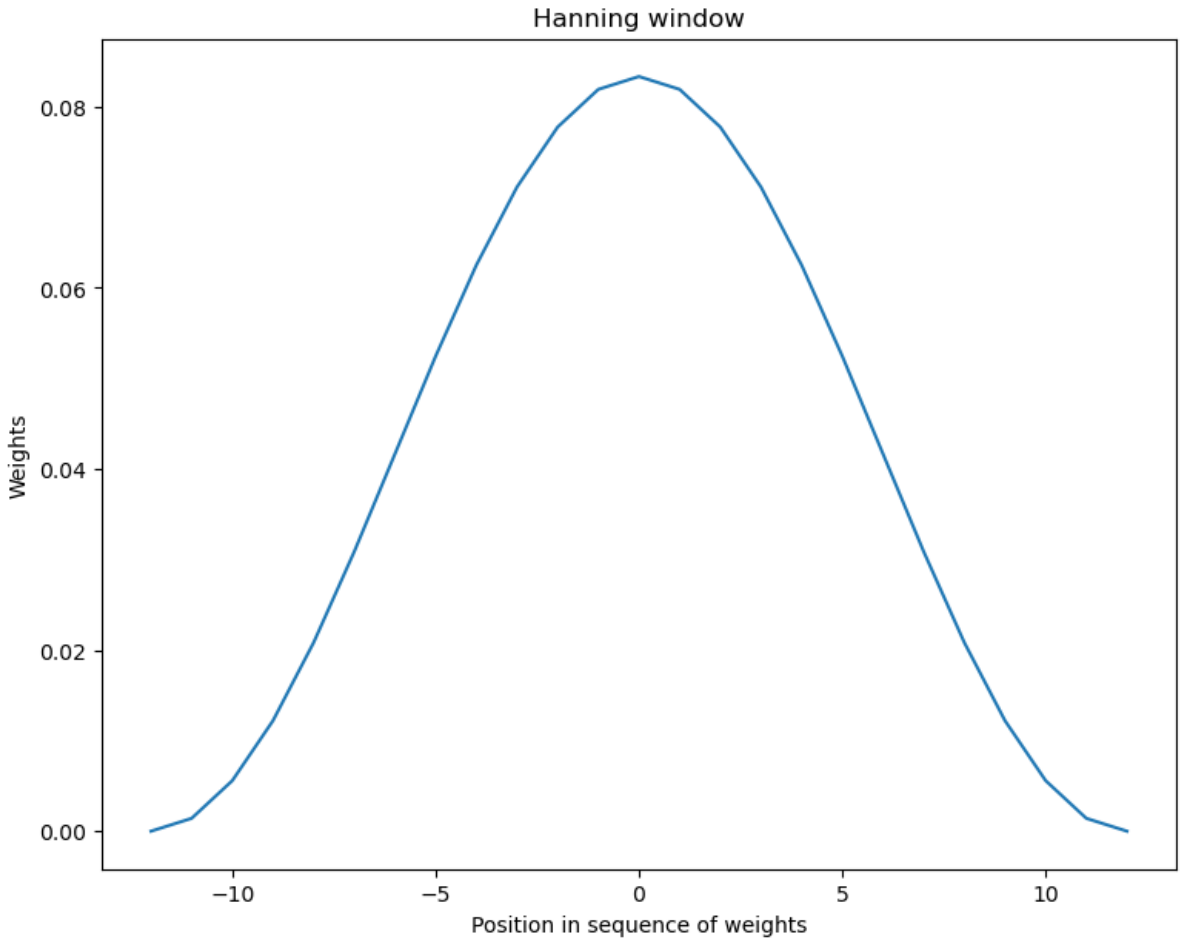
In general, larger values of p indicate more smoothing — more on this below.

The next figure shows the kind of sequence typically used.

Note the smaller weights towards the edges and larger weights in the center, so that more distant values from $I(\omega_j)$ have less weight than closer ones in the sum (30.3).

```
def hanning_window(M):
    w = [0.5 - 0.5 * np.cos(2 * np.pi * n/(M-1)) for n in range(M)]
    return w

window = hanning_window(25) / np.abs(sum(hanning_window(25)))
x = np.linspace(-12, 12, 25)
fig, ax = plt.subplots(figsize=(9, 7))
ax.plot(x, window)
ax.set_title("Hanning window")
ax.set_ylabel("Weights")
ax.set_xlabel("Position in sequence of weights")
plt.show()
```



30.3.1 Estimation with Smoothing

Our next step is to provide code that will not only estimate the periodogram but also provide smoothing as required.

Such functions have been written in estspec.py and are available once you've installed QuantEcon.py.

The GitHub listing displays three functions, `smooth()`, `periodogram()`, `ar_periodogram()`. We will discuss the first two here and the third one *below*.

The `periodogram()` function returns a periodogram, optionally smoothed via the `smooth()` function.

Regarding the `smooth()` function, since smoothing adds a nontrivial amount of computation, we have applied a fairly terse array-centric method based around `np.convolve`.

Readers are left either to explore or simply to use this code according to their interests.

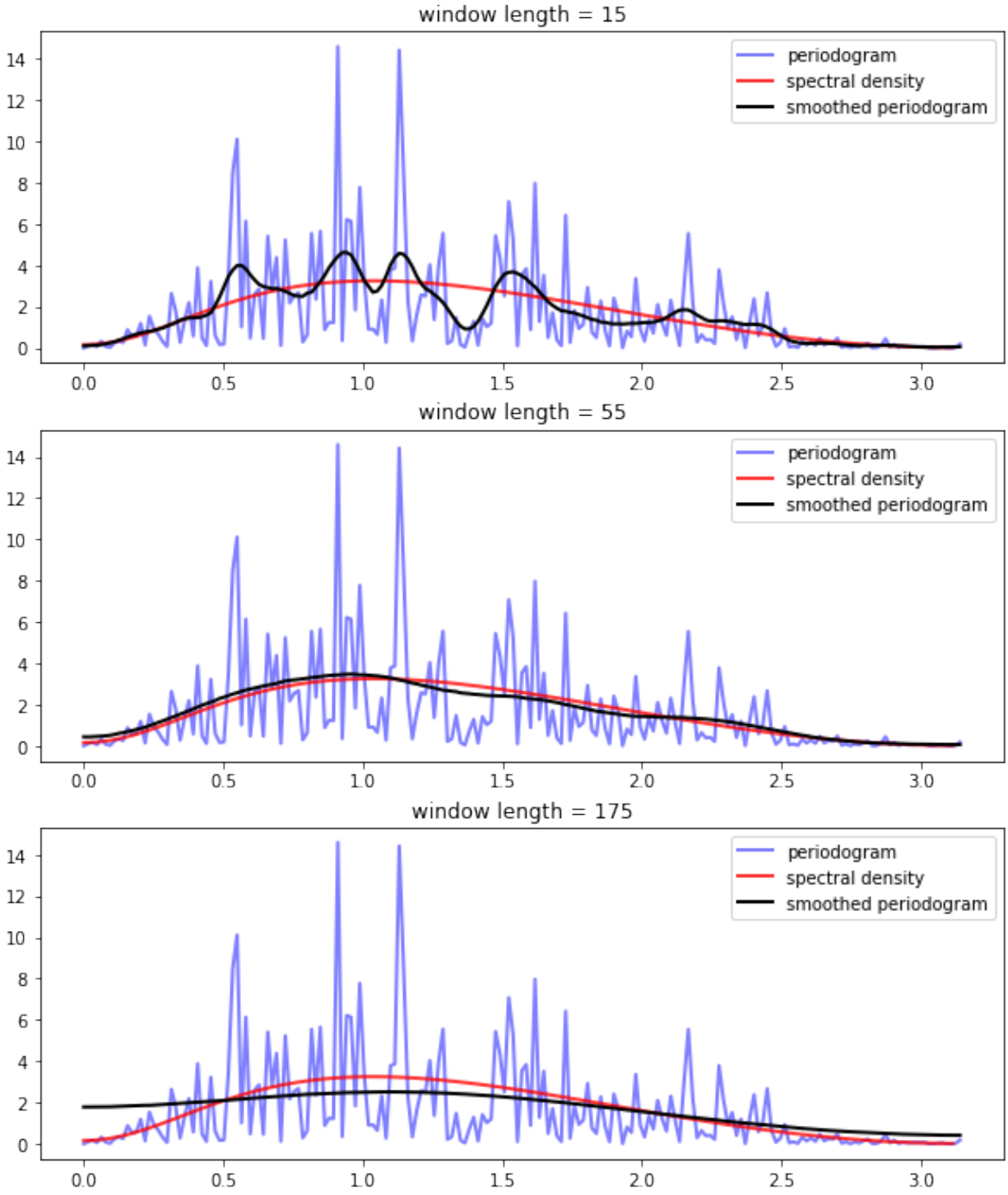
The next three figures each show smoothed and unsmoothed periodograms, as well as the population or "true" spectral density.

(The model is the same as before — see equation (30.2) — and there are 400 observations)

From the top figure to bottom, the window length is varied from small to large.

In looking at the figure, we can see that for this model and data size, the window length chosen in the middle figure provides the best fit.

Relative to this value, the first window length provides insufficient smoothing, while the third gives too much smoothing.


window length = 15
periodogram
spectral density
smoothed periodogram
window length = 55
periodogram
spectral density
smoothed periodogram
window length = 175
periodogram
spectral density
smoothed periodogram

Of course in real estimation problems, the true spectral density is not visible and the choice of appropriate smoothing will have to be made based on judgement/priors or some other theory.

30.3.2 Pre-Filtering and Smoothing

In the code listing, we showed three functions from the file `estspec.py`.

The third function in the file (`ar_periodogram()`) adds a pre-processing step to periodogram smoothing.

First, we describe the basic idea, and after that we give the code.

The essential idea is to

1. Transform the data in order to make estimation of the spectral density more efficient.
2. Compute the periodogram associated with the transformed data.
3. Reverse the effect of the transformation on the periodogram, so that it now estimates the spectral density of the original process.

Step 1 is called *pre-filtering* or *pre-whitening*, while step 3 is called *recoloring*.

The first step is called pre-whitening because the transformation is usually designed to turn the data into something closer to white noise.

Why would this be desirable in terms of spectral density estimation?

The reason is that we are smoothing our estimated periodogram based on estimated values at nearby points — recall (30.3).

The underlying assumption that makes this a good idea is that the true spectral density is relatively regular — the value of $I(\omega)$ is close to that of $I(\omega')$ when ω is close to ω'.

This will not be true in all cases, but it is certainly true for white noise.

For white noise, I is as regular as possible — *it is a constant function*.

In this case, values of $I(\omega')$ at points ω' near to ω provided the maximum possible amount of information about the value $I(\omega)$.

Another way to put this is that if I is relatively constant, then we can use a large amount of smoothing without introducing too much bias.

30.3.3 The AR(1) Setting

Let's examine this idea more carefully in a particular setting — where the data are assumed to be generated by an AR(1) process.

(More general ARMA settings can be handled using similar techniques to those described below)

Suppose in particular that $\{X_t\}$ is covariance stationary and AR(1), with

$$X_{t+1} = \mu + \phi X_t + \epsilon_{t+1} \tag{30.4}$$

where μ and $\phi \in (-1, 1)$ are unknown parameters and $\{\epsilon_t\}$ is white noise.

It follows that if we regress X_{t+1} on X_t and an intercept, the residuals will approximate white noise.

Let

- g be the spectral density of $\{\epsilon_t\}$ — a constant function, as discussed above
- I_0 be the periodogram estimated from the residuals — an estimate of g

- f be the spectral density of $\{X_t\}$ — the object we are trying to estimate

In view of *an earlier result* we obtained while discussing ARMA processes, f and g are related by

$$f(\omega) = \left| \frac{1}{1 - \phi e^{i\omega}} \right|^2 g(\omega) \tag{30.5}$$

This suggests that the recoloring step, which constructs an estimate I of f from I_0, should set

$$I(\omega) = \left| \frac{1}{1 - \hat{\phi} e^{i\omega}} \right|^2 I_0(\omega)$$

where $\hat{\phi}$ is the OLS estimate of ϕ.

The code for `ar_periodogram()` — the third function in `estspec.py` — does exactly this. (See the code here).

The next figure shows realizations of the two kinds of smoothed periodograms

1. "standard smoothed periodogram", the ordinary smoothed periodogram, and
2. "AR smoothed periodogram", the pre-whitened and recolored one generated by `ar_periodogram()`

The periodograms are calculated from time series drawn from (30.4) with $\mu = 0$ and $\phi = -0.9$.

Each time series is of length 150.

The difference between the three subfigures is just randomness — each one uses a different draw of the time series.

In all cases, periodograms are fit with the "hamming" window and window length of 65.

Overall, the fit of the AR smoothed periodogram is much better, in the sense of being closer to the true spectral density.

30.4 Exercises

Exercise 30.4.1

Replicate *this figure* (modulo randomness).

The model is as in equation (30.2) and there are 400 observations.

For the smoothed periodogram, the window type is "hamming".

Solution to Exercise 30.4.1

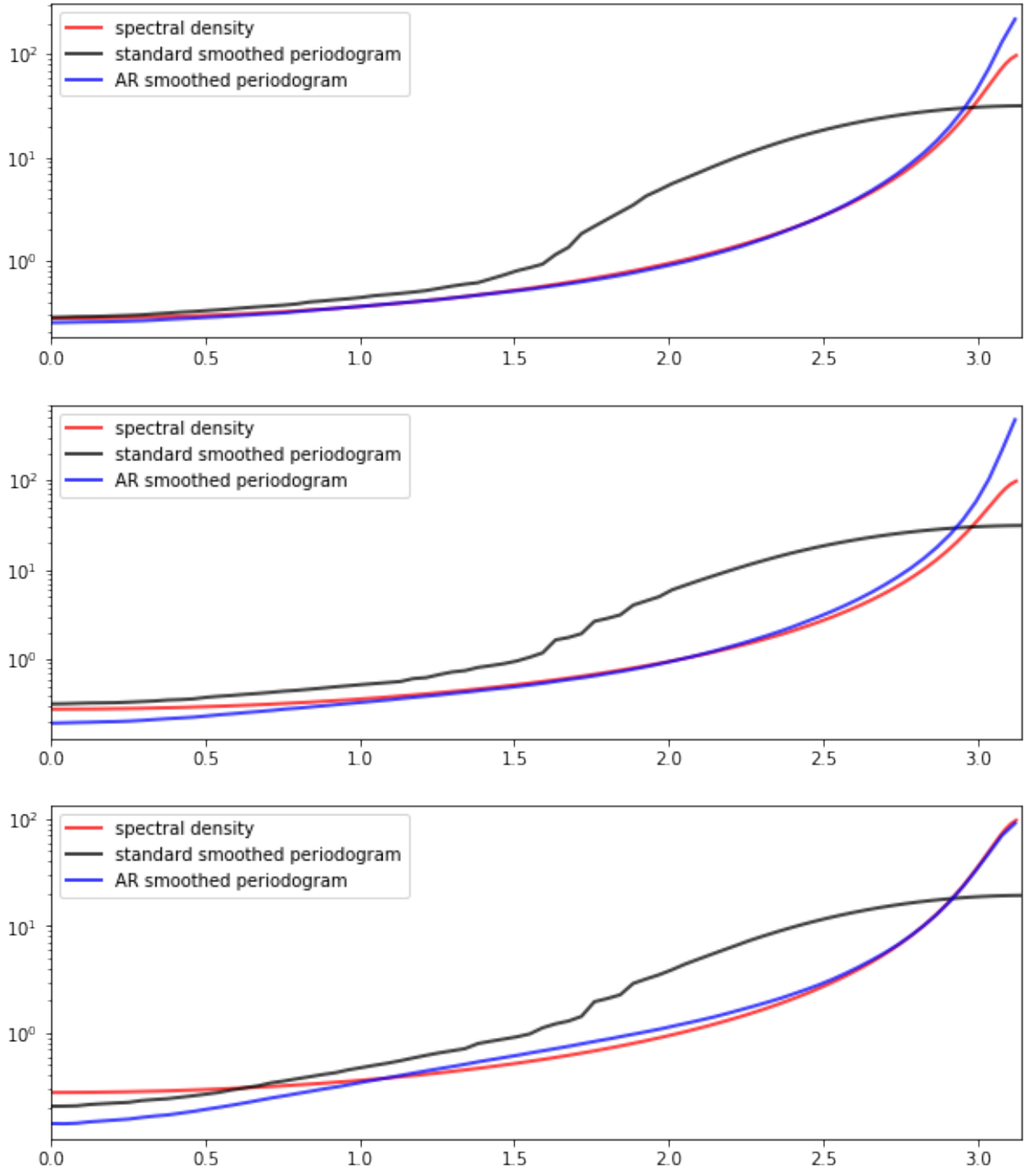
```
## Data
n = 400
ϕ = 0.5
θ = 0, -0.8
lp = ARMA(ϕ, θ)
X = lp.simulation(ts_length=n)

fig, ax = plt.subplots(3, 1, figsize=(10, 12))

for i, wl in enumerate((15, 55, 175)):  # Window lengths

    x, y = periodogram(X)
    ax[i].plot(x, y, 'b-', lw=2, alpha=0.5, label='periodogram')
```

(continues on next page)

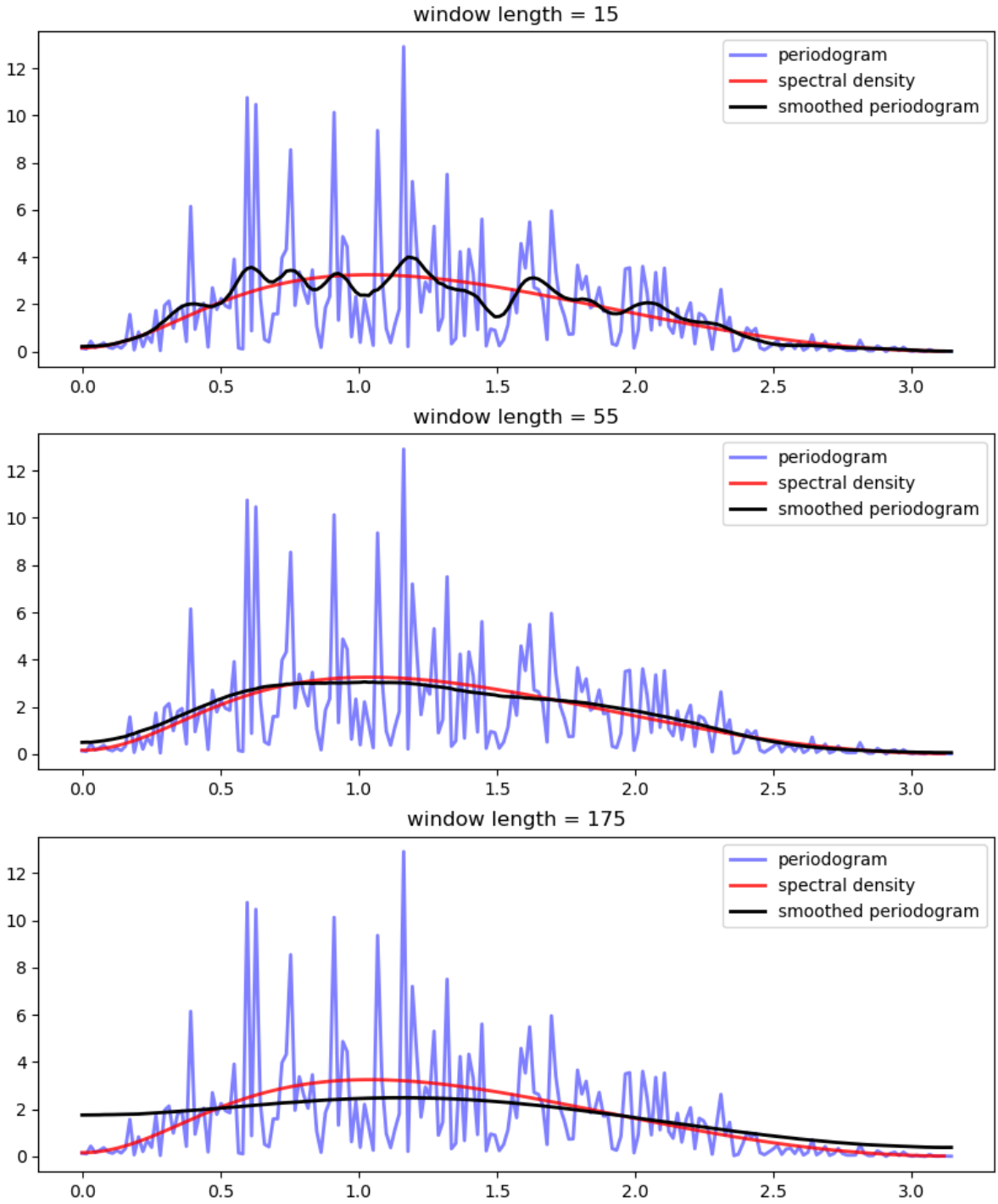

spectral density
standard smoothed periodogram
AR smoothed periodogram
10^2
10^1
10^0
0.0
0.5
1.0
1.5
2.0
2.5
3.0
spectral density
standard smoothed periodogram
AR smoothed periodogram
10^2
10^1
10^0
0.0
0.5
1.0
1.5
2.0
2.5
3.0
spectral density
standard smoothed periodogram
AR smoothed periodogram
10^2
10^1
10^0
0.0
0.5
1.0
1.5
2.0
2.5
3.0

(continued from previous page)

```
    x_sd, y_sd = lp.spectral_density(two_pi=False, res=120)
    ax[i].plot(x_sd, y_sd, 'r-', lw=2, alpha=0.8, label='spectral density')

    x, y_smoothed = periodogram(X, window='hamming', window_len=wl)
    ax[i].plot(x, y_smoothed, 'k-', lw=2, label='smoothed periodogram')

    ax[i].legend()
    ax[i].set_title(f'window length = {wl}')
plt.show()
```



Exercise 30.4.2

Replicate *this figure* (modulo randomness).

The model is as in equation (30.4), with $\mu = 0$, $\phi = -0.9$ and 150 observations in each time series.

All periodograms are fit with the "hamming" window and window length of 65.

Solution to Exercise 30.4.2

```
lp = ARMA(-0.9)
wl = 65


fig, ax = plt.subplots(3, 1, figsize=(10,12))

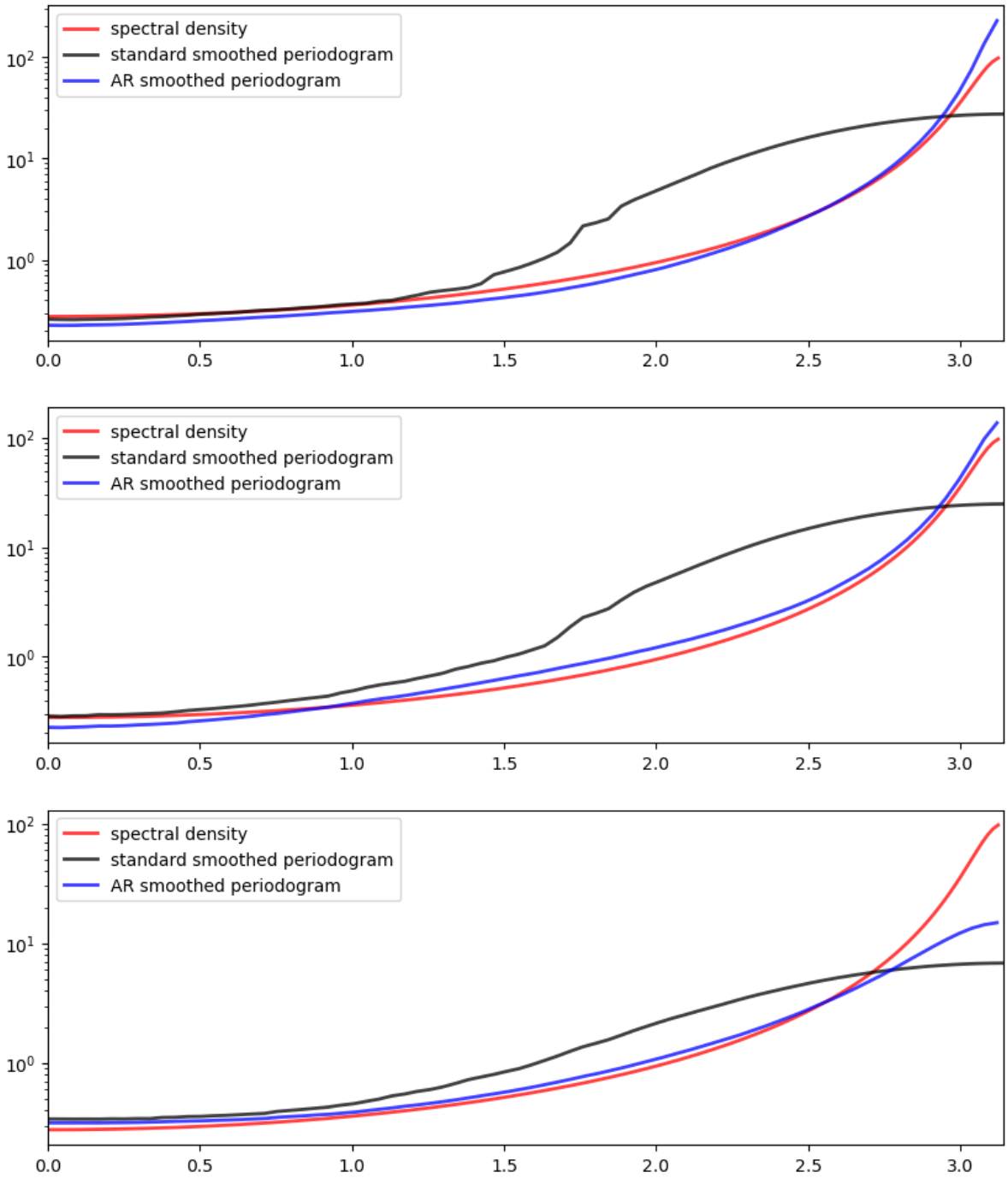
for i in range(3):
    X = lp.simulation(ts_length=150)
    ax[i].set_xlim(0, np.pi)

    x_sd, y_sd = lp.spectral_density(two_pi=False, res=180)
    ax[i].semilogy(x_sd, y_sd, 'r-', lw=2, alpha=0.75,
        label='spectral density')

    x, y_smoothed = periodogram(X, window='hamming', window_len=wl)
    ax[i].semilogy(x, y_smoothed, 'k-', lw=2, alpha=0.75,
        label='standard smoothed periodogram')

    x, y_ar = ar_periodogram(X, window='hamming', window_len=wl)
    ax[i].semilogy(x, y_ar, 'b-', lw=2, alpha=0.75,
        label='AR smoothed periodogram')

    ax[i].legend(loc='upper left')
plt.show()
```


spectral density
standard smoothed periodogram
AR smoothed periodogram
10^2
10^1
10^0
0.0
0.5
1.0
1.5
2.0
2.5
3.0
spectral density
standard smoothed periodogram
AR smoothed periodogram
10^2
10^1
10^0
0.0
0.5
1.0
1.5
2.0
2.5
3.0
spectral density
standard smoothed periodogram
AR smoothed periodogram
10^2
10^1
10^0
0.0
0.5
1.0
1.5
2.0
2.5
3.0

CHAPTER

THIRTYONE

ADDITIVE AND MULTIPLICATIVE FUNCTIONALS

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install --upgrade quantecon
```

31.1 Overview

Many economic time series display persistent growth that prevents them from being asymptotically stationary and ergodic.

For example, outputs, prices, and dividends typically display irregular but persistent growth.

Asymptotic stationarity and ergodicity are key assumptions needed to make it possible to learn by applying statistical methods.

But there are good ways to model time series that have persistent growth that still enable statistical learning based on a law of large numbers for an asymptotically stationary and ergodic process.

Thus, [Hansen, 2012] described two classes of time series models that accommodate growth.

They are

1. **additive functionals** that display random "arithmetic growth"
2. **multiplicative functionals** that display random "geometric growth"

These two classes of processes are closely connected.

If a process $\{y_t\}$ is an additive functional and $\phi_t = \exp(y_t)$, then $\{\phi_t\}$ is a multiplicative functional.

In this lecture, we describe both additive functionals and multiplicative functionals.

We also describe and compute decompositions of additive and multiplicative processes into four components:

1. a **constant**
2. a **trend** component
3. an asymptotically **stationary** component
4. a **martingale**

We describe how to construct, simulate, and interpret these components.

More details about these concepts and algorithms can be found in Hansen [Hansen, 2012] and Hansen and Sargent [Hansen and Sargent, 2024].

Let's start with some imports:

```
import numpy as np
import scipy.linalg as la
import quantecon as qe
import matplotlib.pyplot as plt
from scipy.stats import norm, lognorm
```

31.2 A Particular Additive Functional

[Hansen, 2012] describes a general class of additive functionals.

This lecture focuses on a subclass of these: a scalar process $\{y_t\}_{t=0}^{\infty}$ whose increments are driven by a Gaussian vector autoregression.

Our special additive functional displays interesting time series behavior while also being easy to construct, simulate, and analyze by using linear state-space tools.

We construct our additive functional from two pieces, the first of which is a **first-order vector autoregression** (VAR)

$$x_{t+1} = Ax_t + Bz_{t+1} \tag{31.1}$$

Here

- x_t is an $n \times 1$ vector,
- A is an $n \times n$ stable matrix (all eigenvalues lie within the open unit circle),
- $z_{t+1} \sim N(0, I)$ is an $m \times 1$ IID shock,
- B is an $n \times m$ matrix, and
- $x_0 \sim N(\mu_0, \Sigma_0)$ is a random initial condition for x

The second piece is an equation that expresses increments of $\{y_t\}_{t=0}^{\infty}$ as linear functions of

- a scalar constant ν,
- the vector x_t, and
- the same Gaussian vector z_{t+1} that appears in the VAR (31.1)

In particular,

$$y_{t+1} - y_t = \nu + Dx_t + Fz_{t+1} \tag{31.2}$$

Here $y_0 \sim N(\mu_{y0}, \Sigma_{y0})$ is a random initial condition for y.

The nonstationary random process $\{y_t\}_{t=0}^{\infty}$ displays systematic but random *arithmetic growth*.

31.2.1 Linear State-Space Representation

A convenient way to represent our additive functional is to use a linear state space system.

To do this, we set up state and observation vectors

$$\hat{x}_t = \begin{bmatrix} 1 \\ x_t \\ y_t \end{bmatrix} \quad \text{and} \quad \hat{y}_t = \begin{bmatrix} x_t \\ y_t \end{bmatrix}$$

Next we construct a linear system

$$\begin{bmatrix} 1 \\ x_{t+1} \\ y_{t+1} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & A & 0 \\ \nu & D & 1 \end{bmatrix} \begin{bmatrix} 1 \\ x_t \\ y_t \end{bmatrix} + \begin{bmatrix} 0 \\ B \\ F \end{bmatrix} z_{t+1}$$

$$\begin{bmatrix} x_t \\ y_t \end{bmatrix} = \begin{bmatrix} 0 & I & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ x_t \\ y_t \end{bmatrix}$$

This can be written as

$$\begin{aligned} \hat{x}_{t+1} &= \hat{A}\hat{x}_t + \hat{B}z_{t+1} \\ \hat{y}_t &= \hat{D}\hat{x}_t \end{aligned}$$

which is a standard linear state space system.

To study it, we could map it into an instance of LinearStateSpace from QuantEcon.py.

But here we will use a different set of code for simulation, for reasons described below.

31.3 Dynamics

Let's run some simulations to build intuition.

In doing so we'll assume that z_{t+1} is scalar and that $\tilde{x}_t$ follows a 4th-order scalar autoregression.

$$\tilde{x}_{t+1} = \phi_1 \tilde{x}_t + \phi_2 \tilde{x}_{t-1} + \phi_3 \tilde{x}_{t-2} + \phi_4 \tilde{x}_{t-3} + \sigma z_{t+1} \tag{31.3}$$

in which the zeros z of the polynomial

$$\phi(z) = (1 - \phi_1 z - \phi_2 z^2 - \phi_3 z^3 - \phi_4 z^4)$$

are strictly greater than unity in absolute value.

(Being a zero of $\phi(z)$ means that $\phi(z) = 0$)

Let the increment in $\{y_t\}$ obey

$$y_{t+1} - y_t = \nu + \tilde{x}_t + \sigma z_{t+1}$$

with an initial condition for y_0.

While (31.3) is not a first order system like (31.1), we know that it can be mapped into a first order system.

- For an example of such a mapping, see this example.

In fact, this whole model can be mapped into the additive functional system definition in (31.1) – (31.2) by appropriate selection of the matrices A, B, D, F.

You can try writing these matrices down now as an exercise — correct expressions appear in the code below.

31.3.1 Simulation

When simulating we embed our variables into a bigger system.

This system also constructs the components of the decompositions of y_t and of $\exp(y_t)$ proposed by Hansen [Hansen, 2012].

All of these objects are computed using the code below

```
class AMF_LSS_VAR:
    """
    This class transforms an additive (multiplicative)
    functional into a QuantEcon linear state space system.
    """

    def __init__(self, A, B, D, F=None, ν=None):
        # Unpack required elements
        self.nx, self.nk = B.shape
        self.A, self.B = A, B

        # Checking the dimension of D (extended from the scalar case)
        if len(D.shape) > 1 and D.shape[0] != 1:
            self.nm = D.shape[0]
            self.D = D
        elif len(D.shape) > 1 and D.shape[0] == 1:
            self.nm = 1
            self.D = D
        else:
            self.nm = 1
            self.D = np.expand_dims(D, 0)

        # Create space for additive decomposition
        self.add_decomp = None
        self.mult_decomp = None

        # Set F
        if not np.any(F):
            self.F = np.zeros((self.nk, 1))
        else:
            self.F = F

        # Set ν
        if not np.any(ν):
            self.ν = np.zeros((self.nm, 1))
        elif type(ν) == float:
            self.ν = np.asarray([[ν]])
        elif len(ν.shape) == 1:
            self.ν = np.expand_dims(ν, 1)
        else:
            self.ν = ν

        if self.ν.shape[0] != self.D.shape[0]:
            raise ValueError("The dimension of ν is inconsistent with D!")

        # Construct BIG state space representation
        self.lss = self.construct_ss()

    def construct_ss(self):
```

(continues on next page)

(continued from previous page)

```
        """
        This creates the state space representation that can be passed
        into the quantecon LSS class.
        """
        # Pull out useful info
        nx, nk, nm = self.nx, self.nk, self.nm
        A, B, D, F, ν = self.A, self.B, self.D, self.F, self.ν
        if self.add_decomp:
            ν, H, g = self.add_decomp
        else:
            ν, H, g = self.additive_decomp()

        # Auxiliary blocks with 0's and 1's to fill out the lss matrices
        nx0c = np.zeros((nx, 1))
        nx0r = np.zeros(nx)
        nx1 = np.ones(nx)
        nk0 = np.zeros(nk)
        ny0c = np.zeros((nm, 1))
        ny0r = np.zeros(nm)
        ny1m = np.eye(nm)
        ny0m = np.zeros((nm, nm))
        nyx0m = np.zeros_like(D)

        # Build A matrix for LSS
        # Order of states is: [1, t, xt, yt, mt]
        A1 = np.hstack([1, 0, nx0r, ny0r, ny0r])          # Transition for 1
        A2 = np.hstack([1, 1, nx0r, ny0r, ny0r])          # Transition for t
        # Transition for x_{t+1}
        A3 = np.hstack([nx0c, nx0c, A, nyx0m.T, nyx0m.T])
        # Transition for y_{t+1}
        A4 = np.hstack([ν, ny0c, D, ny1m, ny0m])
        # Transition for m_{t+1}
        A5 = np.hstack([ny0c, ny0c, nyx0m, ny0m, ny1m])
        Abar = np.vstack([A1, A2, A3, A4, A5])

        # Build B matrix for LSS
        Bbar = np.vstack([nk0, nk0, B, F, H])

        # Build G matrix for LSS
        # Order of observation is: [xt, yt, mt, st, tt]
        # Selector for x_{t}
        G1 = np.hstack([nx0c, nx0c, np.eye(nx), nyx0m.T, nyx0m.T])
        G2 = np.hstack([ny0c, ny0c, nyx0m, ny1m, ny0m])    # Selector for y_{t}
        # Selector for martingale
        G3 = np.hstack([ny0c, ny0c, nyx0m, ny0m, ny1m])
        G4 = np.hstack([ny0c, ny0c, -g, ny0m, ny0m])  # Selector for stationary
        G5 = np.hstack([ny0c, ν, nyx0m, ny0m, ny0m])  # Selector for trend
        Gbar = np.vstack([G1, G2, G3, G4, G5])

        # Build H matrix for LSS
        Hbar = np.zeros((Gbar.shape[0], nk))

        # Build LSS type
        x0 = np.hstack([1, 0, nx0r, ny0r, ny0r])
        S0 = np.zeros((len(x0), len(x0)))
        lss = qe.LinearStateSpace(Abar, Bbar, Gbar, Hbar, mu_0=x0, Sigma_0=S0)
```

(continues on next page)

(continued from previous page)

```
        return lss

    def additive_decomp(self):
        """
        Return values for the martingale decomposition
            - ν      : unconditional mean difference in Y
            - H      : coefficient for the (linear) martingale component (κ_a)
            - g      : coefficient for the stationary component g(x)
            - Y_0    : it should be the function of X_0 (for now set it to 0.0)
        """
        I = np.identity(self.nx)
        A_res = la.solve(I - self.A, I)
        g = self.D @ A_res
        H = self.F + self.D @ A_res @ self.B

        return self.ν, H, g

    def multiplicative_decomp(self):
        """
        Return values for the multiplicative decomposition (Example 5.4.4.)
            - ν_tilde  : eigenvalue
            - H        : vector for the Jensen term
        """
        ν, H, g = self.additive_decomp()
        ν_tilde = ν + (.5)*np.expand_dims(np.diag(H @ H.T), 1)

        return ν_tilde, H, g

    def loglikelihood_path(self, x, y):
        A, B, D, F = self.A, self.B, self.D, self.F
        k, T = y.shape
        FF = F @ F.T
        FFinv = la.inv(FF)
        temp = y[:, 1:] - y[:, :-1] - D @ x[:, :-1]
        obs =  temp * FFinv * temp
        obssum = np.cumsum(obs)
        scalar = (np.log(la.det(FF)) + k*np.log(2*np.pi))*np.arange(1, T)

        return -(.5)*(obssum + scalar)

    def loglikelihood(self, x, y):
        llh = self.loglikelihood_path(x, y)

        return llh[-1]
```

Plotting

The code below adds some functions that generate plots for instances of the `AMF_LSS_VAR` *class*.

```
def plot_given_paths(amf, T, ypath, mpath, spath, tpath,
                     mbounds, sbounds, horline=0, show_trend=True):

    # Allocate space
    trange = np.arange(T)

    # Create figure
    fig, ax = plt.subplots(2, 2, sharey=True, figsize=(15, 8))

    # Plot all paths together
    ax[0, 0].plot(trange, ypath[0, :], label="$y_t$", color="k")
    ax[0, 0].plot(trange, mpath[0, :], label="$m_t$", color="m")
    ax[0, 0].plot(trange, spath[0, :], label="$s_t$", color="g")
    if show_trend:
        ax[0, 0].plot(trange, tpath[0, :], label="$t_t$", color="r")
    ax[0, 0].axhline(horline, color="k", linestyle="-.")
    ax[0, 0].set_title("One Path of All Variables")
    ax[0, 0].legend(loc="upper left")

    # Plot Martingale Component
    ax[0, 1].plot(trange, mpath[0, :], "m")
    ax[0, 1].plot(trange, mpath.T, alpha=0.45, color="m")
    ub = mbounds[1, :]
    lb = mbounds[0, :]

    ax[0, 1].fill_between(trange, lb, ub, alpha=0.25, color="m")
    ax[0, 1].set_title("Martingale Components for Many Paths")
    ax[0, 1].axhline(horline, color="k", linestyle="-.")

    # Plot Stationary Component
    ax[1, 0].plot(spath[0, :], color="g")
    ax[1, 0].plot(spath.T, alpha=0.25, color="g")
    ub = sbounds[1, :]
    lb = sbounds[0, :]
    ax[1, 0].fill_between(trange, lb, ub, alpha=0.25, color="g")
    ax[1, 0].axhline(horline, color="k", linestyle="-.")
    ax[1, 0].set_title("Stationary Components for Many Paths")

    # Plot Trend Component
    if show_trend:
        ax[1, 1].plot(tpath.T, color="r")
    ax[1, 1].set_title("Trend Components for Many Paths")
    ax[1, 1].axhline(horline, color="k", linestyle="-.")

    return fig

def plot_additive(amf, T, npaths=25, show_trend=True):
    """
    Plots for the additive decomposition.
    Acts on an instance amf of the AMF_LSS_VAR class

    """
    # Pull out right sizes so we know how to increment
```

(continues on next page)

(continued from previous page)

```
    nx, nk, nm = amf.nx, amf.nk, amf.nm

    # Allocate space (nm is the number of additive functionals -
    # we want npaths for each)
    mpath = np.empty((nm*npaths, T))
    mbounds = np.empty((nm*2, T))
    spath = np.empty((nm*npaths, T))
    sbounds = np.empty((nm*2, T))
    tpath = np.empty((nm*npaths, T))
    ypath = np.empty((nm*npaths, T))

    # Simulate for as long as we wanted
    moment_generator = amf.lss.moment_sequence()
    # Pull out population moments
    for t in range (T):
        tmoms = next(moment_generator)
        ymeans = tmoms[1]
        yvar = tmoms[3]

        # Lower and upper bounds - for each additive functional
        for ii in range(nm):
            li, ui = ii*2, (ii+1)*2
            mscale = np.sqrt(yvar[nx+nm+ii, nx+nm+ii])
            sscale = np.sqrt(yvar[nx+2*nm+ii, nx+2*nm+ii])
            if mscale == 0.0:
                mscale = 1e-12   # avoids a RuntimeWarning from calculating ppf
            if sscale == 0.0:    # of normal distribution with std dev = 0.
                sscale = 1e-12   # sets std dev to small value instead

            madd_dist = norm(ymeans[nx+nm+ii], mscale)
            sadd_dist = norm(ymeans[nx+2*nm+ii], sscale)

            mbounds[li:ui, t] = madd_dist.ppf([0.01, .99])
            sbounds[li:ui, t] = sadd_dist.ppf([0.01, .99])

    # Pull out paths
    for n in range(npaths):
        x, y = amf.lss.simulate(T)
        for ii in range(nm):
            ypath[npaths*ii+n, :] = y[nx+ii, :]
            mpath[npaths*ii+n, :] = y[nx+nm + ii, :]
            spath[npaths*ii+n, :] = y[nx+2*nm + ii, :]
            tpath[npaths*ii+n, :] = y[nx+3*nm + ii, :]

    add_figs = []

    for ii in range(nm):
        li, ui = npaths*(ii), npaths*(ii+1)
        LI, UI = 2*(ii), 2*(ii+1)
        add_figs.append(plot_given_paths(amf, T,
                                         ypath[li:ui,:],
                                         mpath[li:ui,:],
                                         spath[li:ui,:],
                                         tpath[li:ui,:],
                                         mbounds[LI:UI,:],
                                         sbounds[LI:UI,:],
```

(continues on next page)

(continued from previous page)

```
                                                show_trend=show_trend))

        add_figs[ii].suptitle(f'Additive decomposition of $y_{ii+1}$',
                              fontsize=14)

    return add_figs


def plot_multiplicative(amf, T, npaths=25, show_trend=True):
    """
    Plots for the multiplicative decomposition

    """
    # Pull out right sizes so we know how to increment
    nx, nk, nm = amf.nx, amf.nk, amf.nm
    # Matrices for the multiplicative decomposition
    ν_tilde, H, g = amf.multiplicative_decomp()

    # Allocate space (nm is the number of functionals -
    # we want npaths for each)
    mpath_mult = np.empty((nm*npaths, T))
    mbounds_mult = np.empty((nm*2, T))
    spath_mult = np.empty((nm*npaths, T))
    sbounds_mult = np.empty((nm*2, T))
    tpath_mult = np.empty((nm*npaths, T))
    ypath_mult = np.empty((nm*npaths, T))

    # Simulate for as long as we wanted
    moment_generator = amf.lss.moment_sequence()
    # Pull out population moments
    for t in range(T):
        tmoms = next(moment_generator)
        ymeans = tmoms[1]
        yvar = tmoms[3]

        # Lower and upper bounds - for each multiplicative functional
        for ii in range(nm):
            li, ui = ii*2, (ii+1)*2
            Mdist = lognorm(np.sqrt(yvar[nx+nm+ii, nx+nm+ii]).item(),
                            scale=np.exp(ymeans[nx+nm+ii] \
                                                - t * (.5)
                                                * np.expand_dims(
                                                    np.diag(H @ H.T),
                                                    1
                                                    )[ii]
                                                ).item()
                                            )
            Sdist = lognorm(np.sqrt(yvar[nx+2*nm+ii, nx+2*nm+ii]).item(),
                            scale = np.exp(-ymeans[nx+2*nm+ii]).item())
            mbounds_mult[li:ui, t] = Mdist.ppf([.01, .99])
            sbounds_mult[li:ui, t] = Sdist.ppf([.01, .99])

    # Pull out paths
    for n in range(npaths):
        x, y = amf.lss.simulate(T)
        for ii in range(nm):
```

(continues on next page)

(continued from previous page)

```
            ypath_mult[npaths*ii+n, :] = np.exp(y[nx+ii, :])
            mpath_mult[npaths*ii+n, :] = np.exp(y[nx+nm + ii, :] \
                                                - np.arange(T)*(.5)
                                                * np.expand_dims(np.diag(H
                                                                         @ H.T),
                                                                         1)[ii]
                                                )
            spath_mult[npaths*ii+n, :] = 1/np.exp(-y[nx+2*nm + ii, :])
            tpath_mult[npaths*ii+n, :] = np.exp(y[nx+3*nm + ii, :]
                                                + np.arange(T)*(.5)
                                                * np.expand_dims(np.diag(H
                                                                         @ H.T),
                                                                         1)[ii]
                                                )

    mult_figs = []

    for ii in range(nm):
        li, ui = npaths*(ii), npaths*(ii+1)
        LI, UI = 2*(ii), 2*(ii+1)

        mult_figs.append(plot_given_paths(amf,T,
                                                ypath_mult[li:ui,:],
                                                mpath_mult[li:ui,:],
                                                spath_mult[li:ui,:],
                                                tpath_mult[li:ui,:],
                                                mbounds_mult[LI:UI,:],
                                                sbounds_mult[LI:UI,:],
                                                1,
                                                show_trend=show_trend))
        mult_figs[ii].suptitle(f'Multiplicative decomposition of \
                                    $y_{ii+1}$', fontsize=14)

    return mult_figs

def plot_martingale_paths(amf, T, mpath, mbounds, horline=1, show_trend=False):
    # Allocate space
    trange = np.arange(T)

    # Create figure
    fig, ax = plt.subplots(1, 1, figsize=(10, 6))

    # Plot Martingale Component
    ub = mbounds[1, :]
    lb = mbounds[0, :]
    ax.fill_between(trange, lb, ub, color="#ffccff")
    ax.axhline(horline, color="k", linestyle="-.")
    ax.plot(trange, mpath.T, linewidth=0.25, color="#4c4c4c")

    return fig

def plot_martingales(amf, T, npaths=25):

    # Pull out right sizes so we know how to increment
    nx, nk, nm = amf.nx, amf.nk, amf.nm
    # Matrices for the multiplicative decomposition
```

(continues on next page)

(continued from previous page)

```
    v_tilde, H, g = amf.multiplicative_decomp()

    # Allocate space (nm is the number of functionals -
    # we want npaths for each)
    mpath_mult = np.empty((nm*npaths, T))
    mbounds_mult = np.empty((nm*2, T))

    # Simulate for as long as we wanted
    moment_generator = amf.lss.moment_sequence()
    # Pull out population moments
    for t in range (T):
        tmoms = next(moment_generator)
        ymeans = tmoms[1]
        yvar = tmoms[3]

        # Lower and upper bounds - for each functional
        for ii in range(nm):
            li, ui = ii*2, (ii+1)*2
            Mdist = lognorm(np.sqrt(yvar[nx+nm+ii, nx+nm+ii]).item(),
                            scale= np.exp(ymeans[nx+nm+ii] \
                                                    - t * (.5)
                                                    * np.expand_dims(
                                                        np.diag(H @ H.T),
                                                        1)[ii]

                                                ).item()
                            )
            mbounds_mult[li:ui, t] = Mdist.ppf([.01, .99])

    # Pull out paths
    for n in range(npaths):
        x, y = amf.lss.simulate(T)
        for ii in range(nm):
            mpath_mult[npaths*ii+n, :] = np.exp(y[nx+nm + ii, :] \
                                                - np.arange(T) * (.5)
                                                * np.expand_dims(np.diag(H
                                                                    @ H.T),
                                                                1)[ii]
                                                )

    mart_figs = []

    for ii in range(nm):
        li, ui = npaths*(ii), npaths*(ii+1)
        LI, UI = 2*(ii), 2*(ii+1)
        mart_figs.append(plot_martingale_paths(amf, T, mpath_mult[li:ui, :],
                                                    mbounds_mult[LI:UI, :],
                                                    horline=1))
        mart_figs[ii].suptitle(f'Martingale components for many paths of \
                                $y_{ii+1}$', fontsize=14)

    return mart_figs
```

For now, we just plot y_t and x_t, postponing until later a description of exactly how we compute them.

```
ϕ_1, ϕ_2, ϕ_3, ϕ_4 = 0.5, -0.2, 0, 0.5
σ = 0.01
ν = 0.01    # Growth rate

# A matrix should be n x n
A = np.array([[ϕ_1, ϕ_2, ϕ_3, ϕ_4],
              [  1,   0,   0,   0],
              [  0,   1,   0,   0],
              [  0,   0,   1,   0]])

# B matrix should be n x k
B = np.array([[σ, 0, 0, 0]]).T

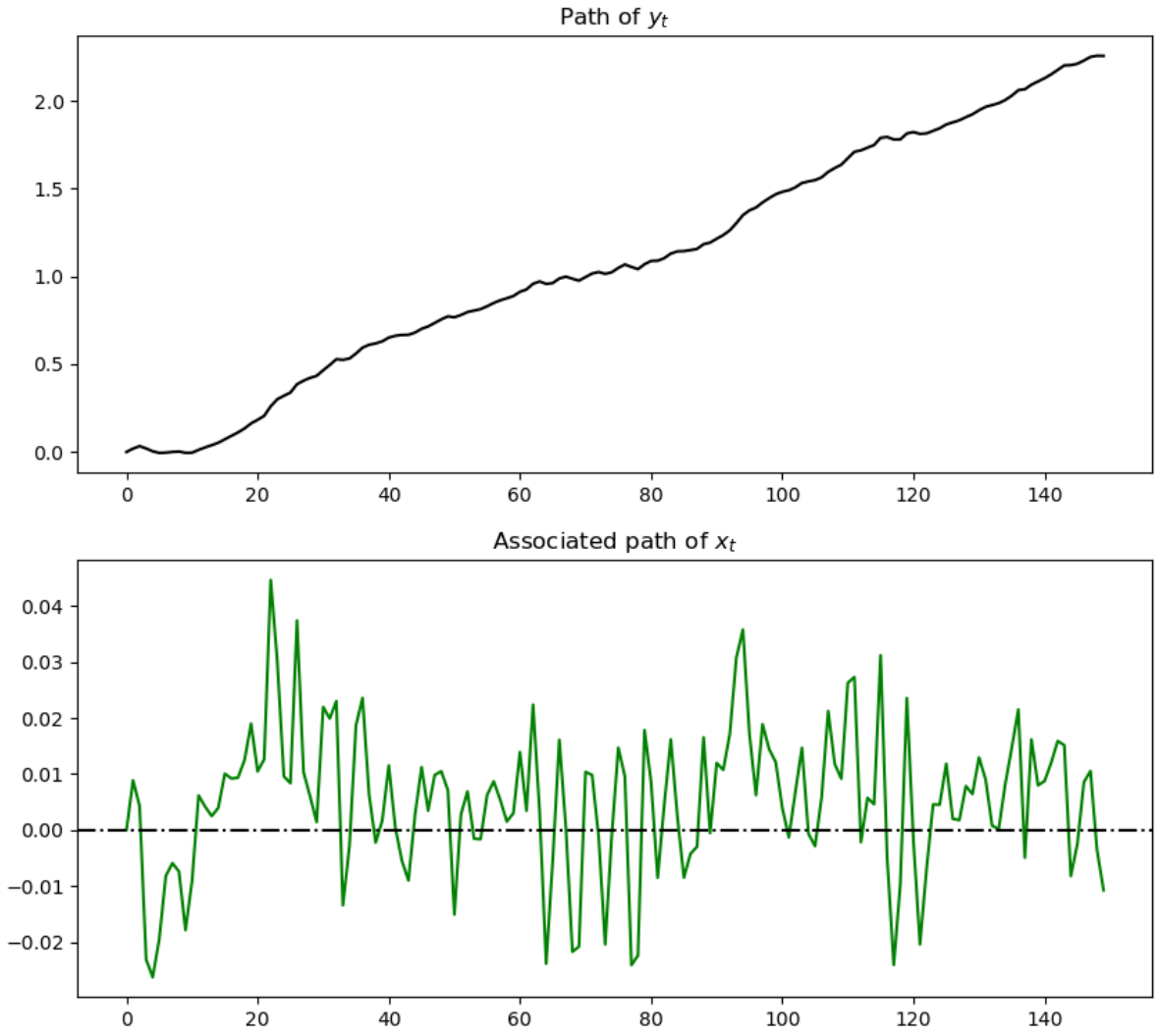
D = np.array([1, 0, 0, 0]) @ A
F = np.array([1, 0, 0, 0]) @ B

amf = AMF_LSS_VAR(A, B, D, F, ν=ν)

T = 150
x, y = amf.lss.simulate(T)

fig, ax = plt.subplots(2, 1, figsize=(10, 9))

ax[0].plot(np.arange(T), y[amf.nx, :], color='k')
ax[0].set_title('Path of $y_t$')
ax[1].plot(np.arange(T), y[0, :], color='g')
ax[1].axhline(0, color='k', linestyle='-.')
ax[1].set_title('Associated path of $x_t$')
plt.show()
```



Notice the irregular but persistent growth in y_t.

31.3.2 Decomposition

Hansen and Sargent [Hansen and Sargent, 2024] describe how to construct a decomposition of an additive functional into four parts:

- a constant inherited from initial values x_0 and y_0
- a linear trend
- a martingale
- an (asymptotically) stationary component

To attain this decomposition for the particular class of additive functionals defined by (31.1) and (31.2), we first construct the matrices

$$H := F + D(I - A)^{-1}B$$
$$g := D(I - A)^{-1}$$

Then the Hansen [Hansen, 2012], [Hansen and Sargent, 2024] decomposition is

$$y_t = \underbrace{t\nu}_{\text{trend component}} + \overbrace{\sum_{j=1}^{t} Hz_j}^{\text{Martingale component}} - \underbrace{gx_t}_{\text{stationary component}} + \overbrace{gx_0 + y_0}^{\text{initial conditions}}$$

At this stage, you should pause and verify that $y_{t+1} - y_t$ satisfies (31.2).

It is convenient for us to introduce the following notation:

- $\tau_t = \nu t$, a linear, deterministic trend
- $m_t = \sum_{j=1}^{t} Hz_j$, a martingale with time $t+1$ increment Hz_{t+1}
- $s_t = gx_t$, an (asymptotically) stationary component

We want to characterize and simulate components τ_t, m_t, s_t of the decomposition.

A convenient way to do this is to construct an appropriate instance of a linear state space system by using LinearStateSpace from QuantEcon.py.

This will allow us to use the routines in LinearStateSpace to study dynamics.

To start, observe that, under the dynamics in (31.1) and (31.2) and with the definitions just given,

$$\begin{bmatrix} 1 \\ t+1 \\ x_{t+1} \\ y_{t+1} \\ m_{t+1} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & A & 0 & 0 \\ \nu & 0 & D & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ t \\ x_t \\ y_t \\ m_t \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ B \\ F \\ H \end{bmatrix} z_{t+1}$$

and

$$\begin{bmatrix} x_t \\ y_t \\ \tau_t \\ m_t \\ s_t \end{bmatrix} = \begin{bmatrix} 0 & 0 & I & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & \nu & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & -g & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ t \\ x_t \\ y_t \\ m_t \end{bmatrix}$$

With

$$\tilde{x} := \begin{bmatrix} 1 \\ t \\ x_t \\ y_t \\ m_t \end{bmatrix} \quad \text{and} \quad \tilde{y} := \begin{bmatrix} x_t \\ y_t \\ \tau_t \\ m_t \\ s_t \end{bmatrix}$$

we can write this as the linear state space system

$$\begin{aligned} \tilde{x}_{t+1} &= \tilde{A}\tilde{x}_t + \tilde{B}z_{t+1} \\ \tilde{y}_t &= \tilde{D}\tilde{x}_t \end{aligned}$$

By picking out components of $\tilde{y}_t$, we can track all variables of interest.

31.4 Code

The class `AMF_LSS_VAR` mentioned *above* does all that we want to study our additive functional.

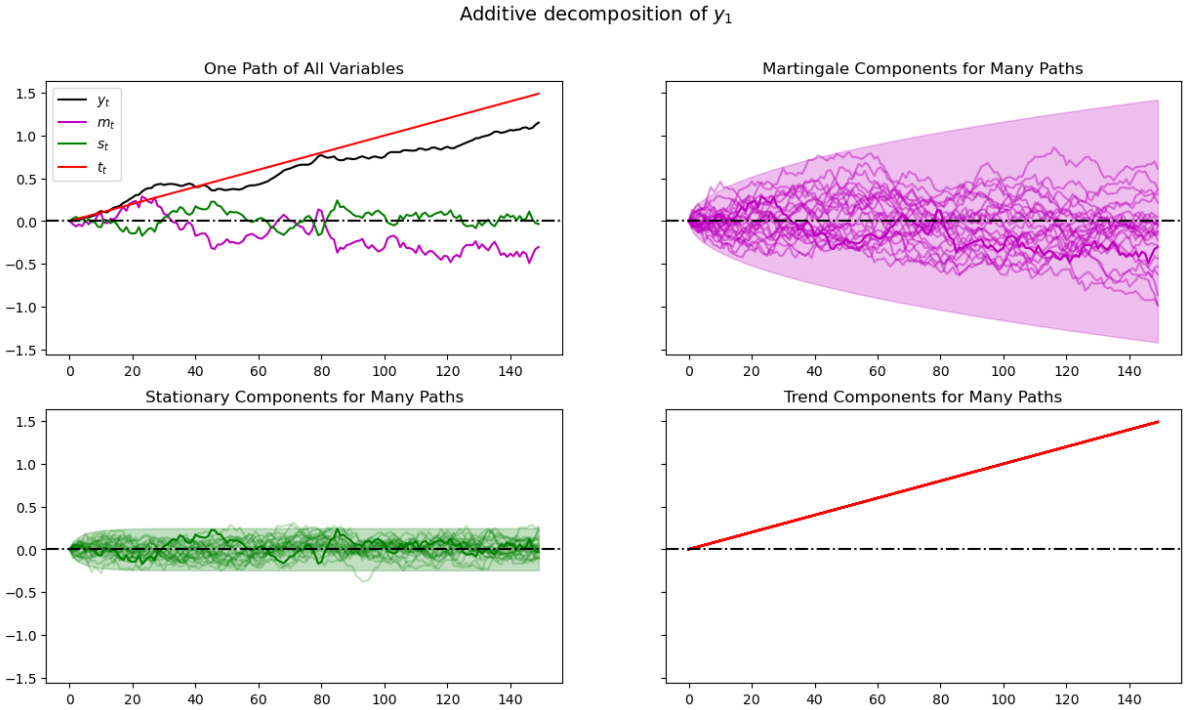
In fact, `AMF_LSS_VAR` does more because it allows us to study an associated multiplicative functional as well.

(A hint that it does more is the name of the class – here AMF stands for "additive and multiplicative functional" – the code computes and displays objects associated with multiplicative functionals too.)

Let's use this code (embedded above) to explore the *example process described above*.

If you run *the code that first simulated that example* again and then the method call you will generate (modulo randomness) the plot

```
plot_additive(amf, T)
plt.show()
```



When we plot multiple realizations of a component in the 2nd, 3rd, and 4th panels, we also plot the population 95% probability coverage sets computed using the LinearStateSpace class.

We have chosen to simulate many paths, all starting from the *same* non-random initial conditions x_0, y_0 (you can tell this from the shape of the 95% probability coverage shaded areas).

Notice tell-tale signs of these probability coverage shaded areas

- the purple one for the martingale component m_t grows with $\sqrt{t}$
- the green one for the stationary component s_t converges to a constant band

31.4.1 Associated Multiplicative Functional

Where $\{y_t\}$ is our additive functional, let $M_t = \exp(y_t)$.

As mentioned above, the process $\{M_t\}$ is called a **multiplicative functional**.

Corresponding to the additive decomposition described above we have a multiplicative decomposition of M_t

$$\frac{M_t}{M_0} = \exp(t\nu) \exp\Big(\sum_{j=1}^{t} H \cdot Z_j\Big) \exp\Big(D(I-A)^{-1}x_0 - D(I-A)^{-1}x_t\Big)$$

or

$$\frac{M_t}{M_0} = \exp\left(\tilde{\nu} t\right) \left(\frac{\widetilde{M}_t}{\widetilde{M}_0}\right) \left(\frac{\tilde{e}(X_0)}{\tilde{e}(x_t)}\right)$$

where

$$\tilde{\nu} = \nu + \frac{H \cdot H}{2}, \quad \widetilde{M}_t = \exp\left(\sum_{j=1}^{t}\left(H \cdot z_j - \frac{H \cdot H}{2}\right)\right), \quad \widetilde{M}_0 = 1$$

and

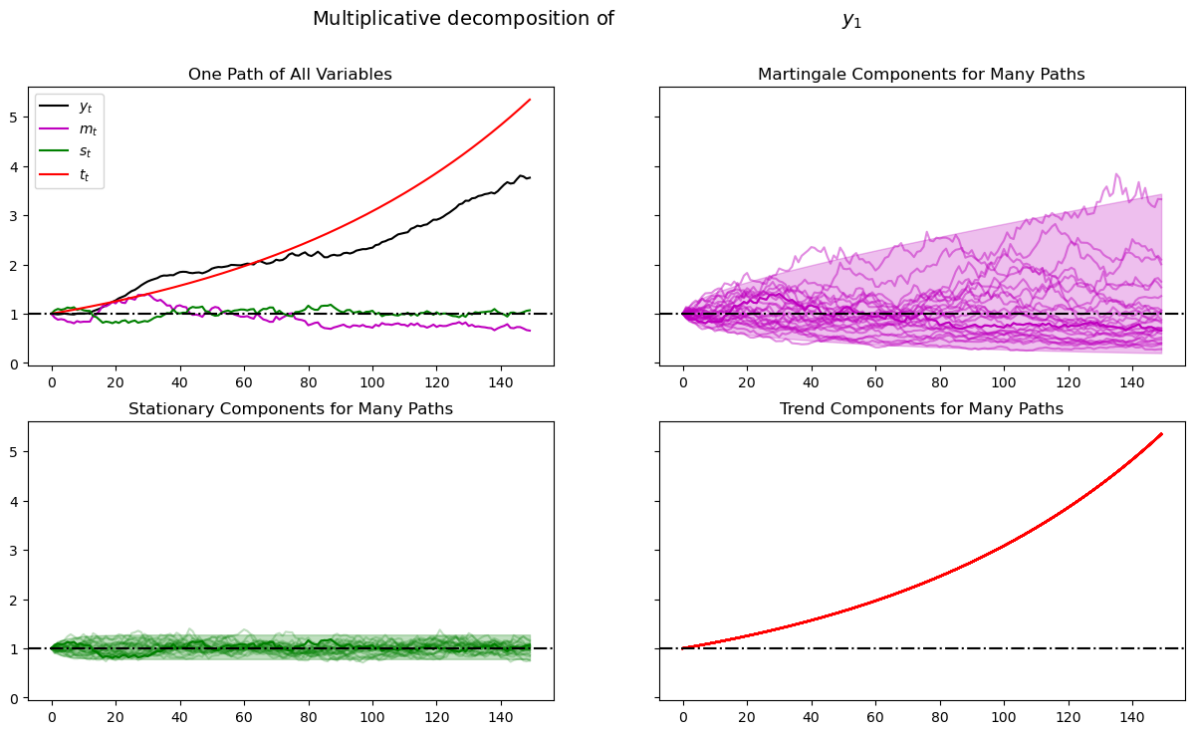
$$\tilde{e}(x) = \exp[g(x)] = \exp[D(I-A)^{-1}x]$$

An instance of class `AMF_LSS_VAR` (*above*) includes this associated multiplicative functional as an attribute.

Let's plot this multiplicative functional for our example.

If you run *the code that first simulated that example* again and then the method call in the cell below you'll obtain the graph in the next cell.

```
plot_multiplicative(amf, T)
plt.show()
```



As before, when we plotted multiple realizations of a component in the 2nd, 3rd, and 4th panels, we also plotted population 95% confidence bands computed using the LinearStateSpace class.

Comparing this figure and the last also helps show how geometric growth differs from arithmetic growth.

The top right panel of the above graph shows a panel of martingales associated with the panel of $M_t = \exp(y_t)$ that we have generated for a limited horizon T.

It is interesting to how the martingale behaves as $T \to +\infty$.

Let's see what happens when we set $T = 12000$ instead of 150.

31.4.2 Peculiar Large Sample Property

Hansen and Sargent [Hansen and Sargent, 2024] (ch. 8) describe the following two properties of the martingale component $\widetilde{M}_t$ of the multiplicative decomposition

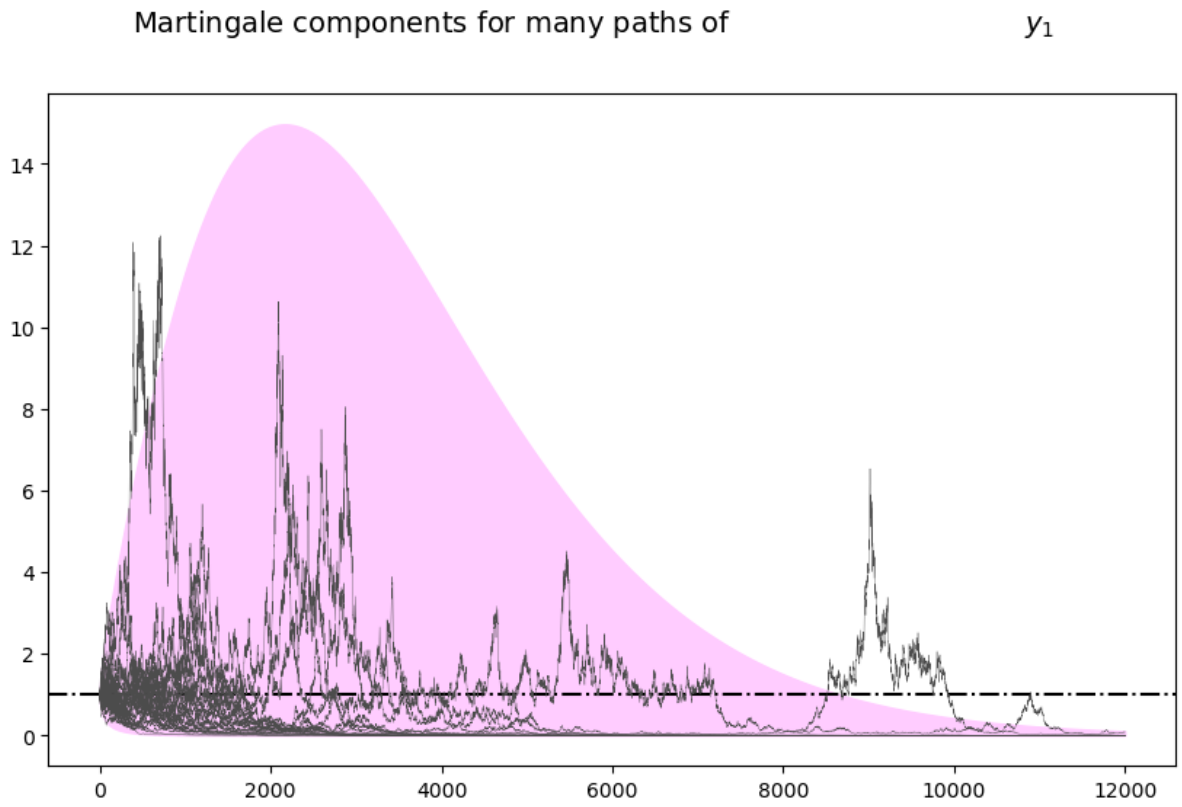
- while $E_0 \widetilde{M}_t = 1$ for all $t \geq 0$, nevertheless ...
- as $t \to +\infty$, $\widetilde{M}_t$ converges to zero almost surely

The first property follows from the fact that $\widetilde{M}_t$ is a multiplicative martingale with initial condition $\widetilde{M}_0 = 1$.

The second is a **peculiar property** noted and proved by Hansen and Sargent [Hansen and Sargent, 2024].

The following simulation of many paths of $\widetilde{M}_t$ illustrates both properties

```
np.random.seed(10021987)
plot_martingales(amf, 12000)
plt.show()
```



The dotted line in the above graph is the mean $E\tilde{M}_t = 1$ of the martingale.

It remains constant at unity, illustrating the first property.

The purple 95 percent frequency coverage interval collapses around zero, illustrating the second property.

31.5 More About the Multiplicative Martingale

Let's drill down and study probability distribution of the multiplicative martingale $\{\widetilde{M}_t\}_{t=0}^{\infty}$ in more detail.

As we have seen, it has representation

$$\widetilde{M}_t = \exp\left(\sum_{j=1}^{t}\left(H \cdot z_j - \frac{H \cdot H}{2}\right)\right), \quad \widetilde{M}_0 = 1$$

where $H = [F + D(I - A)^{-1}B]$.

It follows that $\log \widetilde{M}_t \sim \mathcal{N}(-\frac{tH \cdot H}{2}, tH \cdot H)$ and that consequently $\widetilde{M}_t$ is log normal.

31.5.1 Simulating a Multiplicative Martingale Again

Next, we want a program to simulate the likelihood ratio process $\{\tilde{M}_t\}_{t=0}^{\infty}$.

In particular, we want to simulate 5000 sample paths of length T for the case in which x is a scalar and $[A, B, D, F] = [0.8, 0.001, 1.0, 0.01]$ and $\nu = 0.005$.

After accomplishing this, we want to display and study histograms of $\tilde{M}_T^i$ for various values of T.

Here is code that accomplishes these tasks.

31.5.2 Sample Paths

Let's write a program to simulate sample paths of $\{x_t, y_t\}_{t=0}^{\infty}$.

We'll do this by formulating the additive functional as a linear state space model and putting the LinearStateSpace class to work.

```
class AMF_LSS_VAR:
    """
    This class is written to transform a scalar additive functional
    into a linear state space system.
    """
    def __init__(self, A, B, D, F=0.0, ν=0.0):
        # Unpack required elements
        self.A, self.B, self.D, self.F, self.ν = A, B, D, F, ν

        # Create space for additive decomposition
        self.add_decomp = None
        self.mult_decomp = None

        # Construct BIG state space representation
        self.lss = self.construct_ss()

    def construct_ss(self):
```

(continues on next page)

(continued from previous page)

```
        """
        This creates the state space representation that can be passed
        into the quantecon LSS class.
        """
        # Pull out useful info
        A, B, D, F, ν = self.A, self.B, self.D, self.F, self.ν
        nx, nk, nm = 1, 1, 1
        if self.add_decomp:
            ν, H, g = self.add_decomp
        else:
            ν, H, g = self.additive_decomp()

        # Build A matrix for LSS
        # Order of states is: [1, t, xt, yt, mt]
        A1 = np.hstack([1, 0, 0, 0, 0])       # Transition for 1
        A2 = np.hstack([1, 1, 0, 0, 0])       # Transition for t
        A3 = np.hstack([0, 0, A, 0, 0])       # Transition for x_{t+1}
        A4 = np.hstack([ν, 0, D, 1, 0])       # Transition for y_{t+1}
        A5 = np.hstack([0, 0, 0, 0, 1])       # Transition for m_{t+1}
        Abar = np.vstack([A1, A2, A3, A4, A5])

        # Build B matrix for LSS
        Bbar = np.vstack([0, 0, B, F, H])

        # Build G matrix for LSS
        # Order of observation is: [xt, yt, mt, st, tt]
        G1 = np.hstack([0, 0, 1, 0, 0])            # Selector for x_{t}
        G2 = np.hstack([0, 0, 0, 1, 0])            # Selector for y_{t}
        G3 = np.hstack([0, 0, 0, 0, 1])            # Selector for martingale
        G4 = np.hstack([0, 0, -g, 0, 0])           # Selector for stationary
        G5 = np.hstack([0, ν, 0, 0, 0])            # Selector for trend
        Gbar = np.vstack([G1, G2, G3, G4, G5])

        # Build H matrix for LSS
        Hbar = np.zeros((1, 1))

        # Build LSS type
        x0 = np.hstack([1, 0, 0, 0, 0])
        S0 = np.zeros((5, 5))
        lss = qe.LinearStateSpace(Abar, Bbar, Gbar, Hbar,
                                  mu_0=x0, Sigma_0=S0)

        return lss

    def additive_decomp(self):
        """
        Return values for the martingale decomposition (Proposition 4.3.3.)
           - ν     : unconditional mean difference in Y
           - H     : coefficient for the (linear) martingale component (kappa_a)
           - g     : coefficient for the stationary component g(x)
           - Y_0   : it should be the function of X_0 (for now set it to 0.0)
        """
        A_res = 1 / (1 - self.A)
        g = self.D * A_res
        H = self.F + self.D * A_res * self.B
```

(continues on next page)

(continued from previous page)

```
        return self.ν, H, g

    def multiplicative_decomp(self):
        """
        Return values for the multiplicative decomposition (Example 5.4.4.)
            - ν_tilde  : eigenvalue
            - H        : vector for the Jensen term
        """
        ν, H, g = self.additive_decomp()
        ν_tilde = ν + (.5) * H**2

        return ν_tilde, H, g

    def loglikelihood_path(self, x, y):
        A, B, D, F = self.A, self.B, self.D, self.F
        T = y.T.size
        FF = F**2
        FFinv = 1 / FF
        temp = y[1:] - y[:-1] - D * x[:-1]
        obs = temp * FFinv * temp
        obssum = np.cumsum(obs)
        scalar = (np.log(FF) + np.log(2 * np.pi)) * np.arange(1, T)

        return (-0.5) * (obssum + scalar)

    def loglikelihood(self, x, y):
        llh = self.loglikelihood_path(x, y)

        return llh[-1]
```

The heavy lifting is done inside the `AMF_LSS_VAR` class.

The following code adds some simple functions that make it straightforward to generate sample paths from an instance of `AMF_LSS_VAR`.

```
def simulate_xy(amf, T):
    "Simulate individual paths."
    foo, bar = amf.lss.simulate(T)
    x = bar[0, :]
    y = bar[1, :]

    return x, y

def simulate_paths(amf, T=150, I=5000):
    "Simulate multiple independent paths."

    # Allocate space
    storeX = np.empty((I, T))
    storeY = np.empty((I, T))

    for i in range(I):
        # Do specific simulation
        x, y = simulate_xy(amf, T)

        # Fill in our storage matrices
        storeX[i, :] = x
```

(continues on next page)

(continued from previous page)

```
        storeY[i, :] = y

    return storeX, storeY

def population_means(amf, T=150):
    # Allocate Space
    xmean = np.empty(T)
    ymean = np.empty(T)

    # Pull out moment generator
    moment_generator = amf.lss.moment_sequence()

    for tt in range (T):
        tmoms = next(moment_generator)
        ymeans = tmoms[1]
        xmean[tt] = ymeans[0]
        ymean[tt] = ymeans[1]

    return xmean, ymean
```

Now that we have these functions in our toolkit, let's apply them to run some simulations.

```
def simulate_martingale_components(amf, T=1000, I=5000):
    # Get the multiplicative decomposition
    ν, H, g = amf.multiplicative_decomp()

    # Allocate space
    add_mart_comp = np.empty((I, T))

    # Simulate and pull out additive martingale component
    for i in range(I):
        foo, bar = amf.lss.simulate(T)

        # Martingale component is third component
        add_mart_comp[i, :] = bar[2, :]

    mul_mart_comp = np.exp(add_mart_comp - (np.arange(T) * H**2)/2)

    return add_mart_comp, mul_mart_comp


# Build model
amf_2 = AMF_LSS_VAR(0.8, 0.001, 1.0, 0.01,.005)

amc, mmc = simulate_martingale_components(amf_2, 1000, 5000)

amcT = amc[:, -1]
mmcT = mmc[:, -1]

print("The (min, mean, max) of additive Martingale component in period T is")
print(f"\t ({np.min(amcT)}, {np.mean(amcT)}, {np.max(amcT)})")

print("The (min, mean, max) of multiplicative Martingale component \
in period T is")
print(f"\t ({np.min(mmcT)}, {np.mean(mmcT)}, {np.max(mmcT)})")
```

```
The (min, mean, max) of additive Martingale component in period T is
        (-1.8379907335579106, 0.011040789361757435, 1.4697384727035145)
The (min, mean, max) of multiplicative Martingale component in period T is
        (0.14222026893384476, 1.006753060146832, 3.8858858377907133)
```

Let's plot the probability density functions for $\log \widetilde{M}_t$ for $t = 100, 500, 1000, 10000, 100000$.

Then let's use the plots to investigate how these densities evolve through time.

We will plot the densities of $\log \widetilde{M}_t$ for different values of t.

Note: `scipy.stats.lognorm` expects you to pass the standard deviation first $(tH \cdot H)$ and then the exponent of the mean as a keyword argument `scale` (`scale=np.exp(-t * H2 / 2)`).

- See the documentation here.

This is peculiar, so make sure you are careful in working with the log normal distribution.

Here is some code that tackles these tasks

```
def Mtilde_t_density(amf, t, xmin=1e-8, xmax=5.0, npts=5000):

    # Pull out the multiplicative decomposition
    vtilde, H, g = amf.multiplicative_decomp()
    H2 = H*H

    # The distribution
    mdist = lognorm(np.sqrt(t*H2), scale=np.exp(-t*H2/2))
    x = np.linspace(xmin, xmax, npts)
    pdf = mdist.pdf(x)

    return x, pdf


def logMtilde_t_density(amf, t, xmin=-15.0, xmax=15.0, npts=5000):

    # Pull out the multiplicative decomposition
    vtilde, H, g = amf.multiplicative_decomp()
    H2 = H*H

    # The distribution
    lmdist = norm(-t*H2/2, np.sqrt(t*H2))
    x = np.linspace(xmin, xmax, npts)
    pdf = lmdist.pdf(x)

    return x, pdf


times_to_plot = [10, 100, 500, 1000, 2500, 5000]
dens_to_plot = map(lambda t: Mtilde_t_density(amf_2, t, xmin=1e-8, xmax=6.0),
                   times_to_plot)
ldens_to_plot = map(lambda t: logMtilde_t_density(amf_2, t, xmin=-10.0,
                    xmax=10.0), times_to_plot)

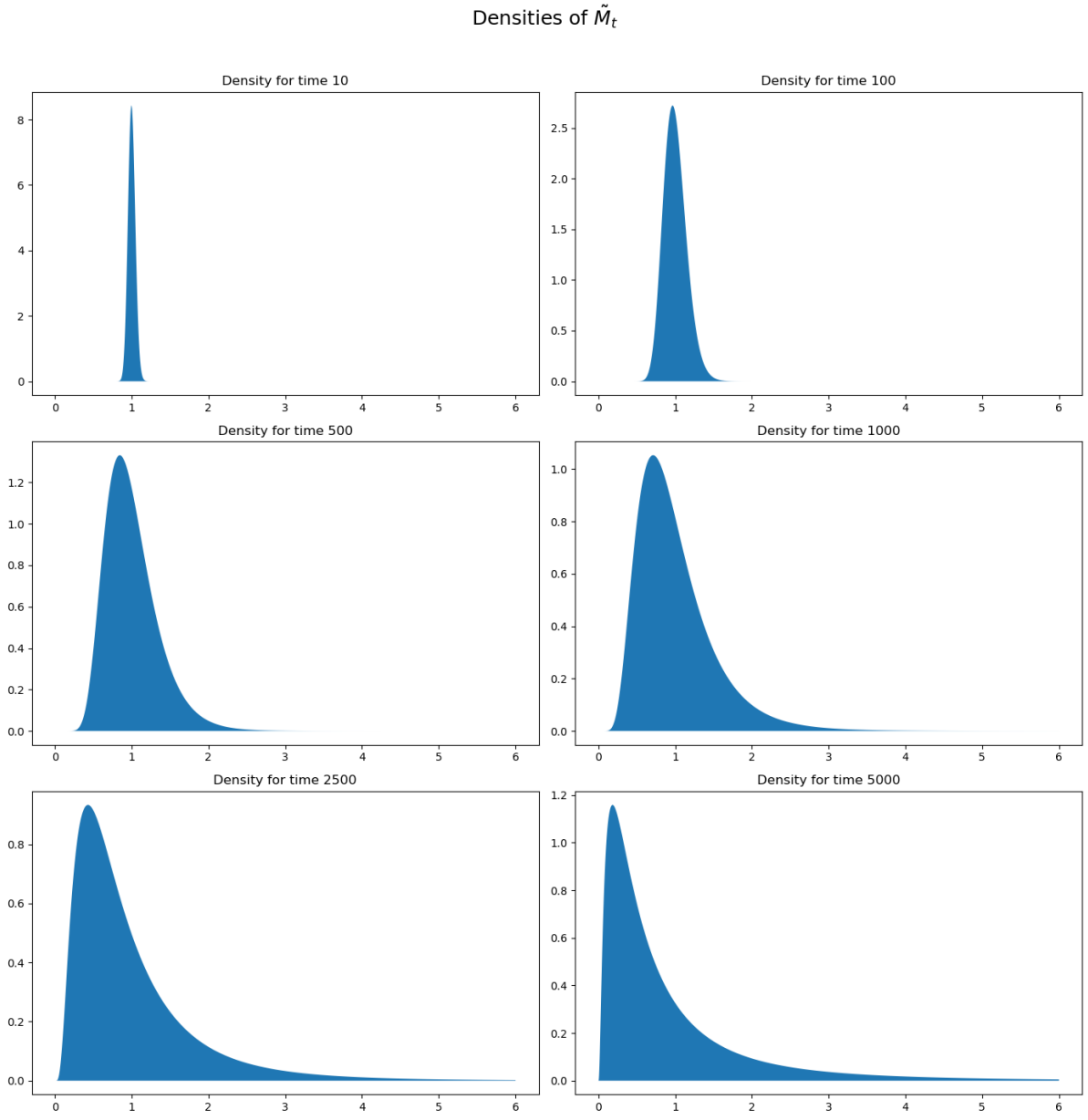
fig, ax = plt.subplots(3, 2, figsize=(14, 14))
ax = ax.flatten()
```

(continues on next page)

(continued from previous page)

```
fig.suptitle(r"Densities of $\tilde{M}_t$", fontsize=18, y=1.02)
for (it, dens_t) in enumerate(dens_to_plot):
    x, pdf = dens_t
    ax[it].set_title(f"Density for time {times_to_plot[it]}")
    ax[it].fill_between(x, np.zeros_like(pdf), pdf)

plt.tight_layout()
plt.show()
```



These probability density functions help us understand mechanics underlying the **peculiar property** of our multiplicative martingale

- As T grows, most of the probability mass shifts leftward toward zero.

- For example, note that most mass is near 1 for $T = 10$ or $T = 100$ but most of it is near 0 for $T = 5000$.
- As T grows, the tail of the density of $\widetilde{M}_T$ lengthens toward the right.
- Enough mass moves toward the right tail to keep $E\widetilde{M}_T = 1$ even as most mass in the distribution of $\widetilde{M}_T$ collapses around 0.

31.5.3 Multiplicative Martingale as Likelihood Ratio Process

This lecture studies **likelihood processes** and **likelihood ratio processes**.

A **likelihood ratio process** is a multiplicative martingale with mean unity.

Likelihood ratio processes exhibit the peculiar property that naturally also appears here.

CHAPTER

THIRTYTWO

CLASSICAL CONTROL WITH LINEAR ALGEBRA

32.1 Overview

In an earlier lecture Linear Quadratic Dynamic Programming Problems, we have studied how to solve a special class of dynamic optimization and prediction problems by applying the method of dynamic programming. In this class of problems

- the objective function is **quadratic** in **states** and **controls**.
- the one-step transition function is **linear**.
- shocks are IID Gaussian or martingale differences.

In this lecture and a companion lecture *Classical Filtering with Linear Algebra*, we study the classical theory of linear-quadratic (LQ) optimal control problems.

The classical approach does not use the two closely related methods – dynamic programming and Kalman filtering – that we describe in other lectures, namely, Linear Quadratic Dynamic Programming Problems and A First Look at the Kalman Filter.

Instead, they use either

- z-transform and lag operator methods, or
- matrix decompositions applied to linear systems of first-order conditions for optimum problems.

In this lecture and the sequel *Classical Filtering with Linear Algebra*, we mostly rely on elementary linear algebra.

The main tool from linear algebra we'll put to work here is LU decomposition.

We'll begin with discrete horizon problems.

Then we'll view infinite horizon problems as appropriate limits of these finite horizon problems.

Later, we will examine the close connection between LQ control and least-squares prediction and filtering problems.

These classes of problems are connected in the sense that to solve each, essentially the same mathematics is used.

Let's start with some standard imports:

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

32.1.1 References

Useful references include [Whittle, 1963], [Hansen and Sargent, 1980], [Orfanidis, 1988], [Athanasios and Pillai, 1991], and [Muth, 1960].

32.2 A Control Problem

Let L be the **lag operator**, so that, for sequence $\{x_t\}$ we have $Lx_t = x_{t-1}$.

More generally, let $L^k x_t = x_{t-k}$ with $L^0 x_t = x_t$ and

$$d(L) = d_0 + d_1 L + ... + d_m L^m$$

where $d_0, d_1, ..., d_m$ is a given scalar sequence.

Consider the discrete-time control problem

$$\max_{\{y_t\}} \lim_{N\to\infty} \sum_{t=0}^{N} \beta^t \left\{ a_t y_t - \frac{1}{2} h y_t^2 - \frac{1}{2} \left[d(L) y_t \right]^2 \right\}, \tag{32.1}$$

where

- h is a positive parameter and $\beta \in (0,1)$ is a discount factor.
- $\{a_t\}_{t\geq 0}$ is a sequence of exponential order less than $\beta^{-1/2}$, by which we mean $\lim_{t\to\infty} \beta^{\frac{t}{2}} a_t = 0$.

Maximization in (32.1) is subject to initial conditions for $y_{-1}, y_{-2} \cdots, y_{-m}$.

Maximization is over infinite sequences $\{y_t\}_{t\geq 0}$.

32.2.1 Example

The formulation of the LQ problem given above is broad enough to encompass many useful models.

As a simple illustration, recall that in LQ Control: Foundations we consider a monopolist facing stochastic demand shocks and adjustment costs.

Let's consider a deterministic version of this problem, where the monopolist maximizes the discounted sum

$$\sum_{t=0}^{\infty} \beta^t \pi_t$$

and

$$\pi_t = p_t q_t - c q_t - \gamma (q_{t+1} - q_t)^2 \quad \text{with} \quad p_t = \alpha_0 - \alpha_1 q_t + d_t$$

In this expression, q_t is output, c is average cost of production, and d_t is a demand shock.

The term $\gamma (q_{t+1} - q_t)^2$ represents adjustment costs.

You will be able to confirm that the objective function can be rewritten as (32.1) when

- $a_t := \alpha_0 + d_t - c$
- $h := 2\alpha_1$
- $d(L) := \sqrt{2\gamma}(I - L)$

Further examples of this problem for factor demand, economic growth, and government policy problems are given in ch. IX of [Sargent, 1987].

32.3 Finite Horizon Theory

We first study a finite N version of the problem.

Later we will study an infinite horizon problem solution as a limiting version of a finite horizon problem.

(This will require being careful because the limits as $N \to \infty$ of the necessary and sufficient conditions for maximizing finite N versions of (32.1) are not sufficient for maximizing (32.1))

We begin by

1. fixing $N > m$,
2. differentiating the finite version of (32.1) with respect to $y_0, y_1, \ldots, y_N$, and
3. setting these derivatives to zero.

For $t = 0, \ldots, N-m$ these first-order necessary conditions are the *Euler equations*.

For $t = N-m+1, \ldots, N$, the first-order conditions are a set of *terminal conditions*.

Consider the term

$$
\begin{aligned}
J &= \sum_{t=0}^{N} \beta^t [d(L) y_t][d(L) y_t] \\
&= \sum_{t=0}^{N} \beta^t \left(d_0 y_t + d_1 y_{t-1} + \cdots + d_m y_{t-m}\right)\left(d_0 y_t + d_1 y_{t-1} + \cdots + d_m y_{t-m}\right)
\end{aligned}
$$

Differentiating J with respect to y_t for $t = 0,\ 1,\ \ldots,\ N-m$ gives

$$
\begin{aligned}
\frac{\partial J}{\partial y_t} &= 2\beta^t d_0 d(L) y_t + 2\beta^{t+1} d_1 d(L) y_{t+1} + \cdots + 2\beta^{t+m} d_m d(L) y_{t+m} \\
&= 2\beta^t \left(d_0 + d_1 \beta L^{-1} + d_2 \beta^2 L^{-2} + \cdots + d_m \beta^m L^{-m}\right) d(L) y_t
\end{aligned}
$$

We can write this more succinctly as

$$
\frac{\partial J}{\partial y_t} = 2\beta^t d(\beta L^{-1})\, d(L) y_t \tag{32.2}
$$

Differentiating J with respect to y_t for $t = N-m+1, \ldots, N$ gives

$$
\begin{aligned}
\frac{\partial J}{\partial y_N} &= 2\beta^N d_0 d(L) y_N \\
\frac{\partial J}{\partial y_{N-1}} &= 2\beta^{N-1}\left[d_0 + \beta d_1 L^{-1}\right] d(L) y_{N-1} \\
&\vdots \qquad \vdots \\
\frac{\partial J}{\partial y_{N-m+1}} &= 2\beta^{N-m+1}\left[d_0 + \beta L^{-1} d_1 + \cdots + \beta^{m-1} L^{-m+1} d_{m-1}\right] d(L) y_{N-m+1}
\end{aligned} \tag{32.3}
$$

With these preliminaries under our belts, we are ready to differentiate (32.1).

Differentiating (32.1) with respect to y_t for $t = 0, \ldots, N-m$ gives the Euler equations

$$
\left[h + d\left(\beta L^{-1}\right) d(L)\right] y_t = a_t, \quad t = 0,\ 1,\ \ldots, N-m \tag{32.4}
$$

The system of equations (32.4) forms a $2 \times m$ order linear *difference equation* that must hold for the values of t indicated.

Differentiating (32.1) with respect to y_t for $t = N - m + 1, \ldots, N$ gives the terminal conditions

$$\begin{aligned} \beta^N (a_N - h y_N - d_0 \, d(L) y_N) &= 0 \\ \beta^{N-1} \left(a_{N-1} - h y_{N-1} - \left(d_0 + \beta \, d_1 \, L^{-1} \right) d(L) \, y_{N-1} \right) &= 0 \\ \vdots \qquad\qquad \vdots \qquad\qquad &= 0 \\ \beta^{N-m+1} \left(a_{N-m+1} - h y_{N-m+1} - (d_0 + \beta L^{-1} d_1 + \cdots + \beta^{m-1} L^{-m+1} d_{m-1}) d(L) y_{N-m+1} \right) &= 0 \end{aligned} \tag{32.5}$$

In the finite N problem, we want simultaneously to solve (32.4) subject to the m initial conditions $y_{-1}, \ldots, y_{-m}$ and the m terminal conditions (32.5).

These conditions uniquely pin down the solution of the finite N problem.

That is, for the finite N problem, conditions (32.4) and (32.5) are necessary and sufficient for a maximum, by concavity of the objective function.

Next, we describe how to obtain the solution using matrix methods.

32.3.1 Matrix Methods

Let's look at how linear algebra can be used to tackle and shed light on the finite horizon LQ control problem.

A Single Lag Term

Let's begin with the special case in which $m = 1$.

We want to solve the system of $N + 1$ linear equations

$$\begin{aligned} [h + d \, (\beta L^{-1}) \, d \, (L)] y_t &= a_t, \quad t = 0, \; 1, \; \ldots, \; N - 1 \\ \beta^N [a_N - h \, y_N - d_0 \, d \, (L) y_N] &= 0 \end{aligned} \tag{32.6}$$

where $d(L) = d_0 + d_1 L$.

These equations are to be solved for $y_0, y_1, \ldots, y_N$ as functions of $a_0, a_1, \ldots, a_N$ and y_{-1}.

Let

$$\phi(L) = \phi_0 + \phi_1 L + \beta \phi_1 L^{-1} = h + d(\beta L^{-1}) d(L) = (h + d_0^2 + d_1^2) + d_1 d_0 L + d_1 d_0 \beta L^{-1}$$

Then we can represent (32.6) as the matrix equation

$$\begin{bmatrix} (\phi_0 - d_1^2) & \phi_1 & 0 & 0 & \ldots & \ldots & 0 \\ \beta \phi_1 & \phi_0 & \phi_1 & 0 & \ldots & \ldots & 0 \\ 0 & \beta \phi_1 & \phi_0 & \phi_1 & \ldots & \ldots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & \ldots & \ldots & \ldots & \beta \phi_1 & \phi_0 & \phi_1 \\ 0 & \ldots & \ldots & \ldots & 0 & \beta \phi_1 & \phi_0 \end{bmatrix} \begin{bmatrix} y_N \\ y_{N-1} \\ y_{N-2} \\ \vdots \\ y_1 \\ y_0 \end{bmatrix} = \begin{bmatrix} a_N \\ a_{N-1} \\ a_{N-2} \\ \vdots \\ a_1 \\ a_0 - \phi_1 y_{-1} \end{bmatrix} \tag{32.7}$$

or

$$W \bar{y} = \bar{a} \tag{32.8}$$

Notice how we have chosen to arrange the y_t's in reverse time order.

The matrix W on the left side of (32.7) is "almost" a Toeplitz matrix (where each descending diagonal is constant).

There are two sources of deviation from the form of a Toeplitz matrix

1. The first element differs from the remaining diagonal elements, reflecting the terminal condition.
2. The sub-diagonal elements equal β time the super-diagonal elements.

The solution of (32.8) can be expressed in the form

$$\bar{y} = W^{-1}\bar{a} \tag{32.9}$$

which represents each element y_t of $\bar{y}$ as a function of the entire vector $\bar{a}$.

That is, y_t is a function of past, present, and future values of a's, as well as of the initial condition y_{-1}.

An Alternative Representation

An alternative way to express the solution to (32.7) or (32.8) is in so-called **feedback-feedforward** form.

The idea here is to find a solution expressing y_t as a function of *past* y's and *current* and *future* a's.

To achieve this solution, one can use an LU decomposition of W.

There always exists a decomposition of W of the form $W = LU$ where

- L is an $(N+1) \times (N+1)$ lower triangular matrix.
- U is an $(N+1) \times (N+1)$ upper triangular matrix.

The factorization can be normalized so that the diagonal elements of U are unity.

Using the LU representation in (32.9), we obtain

$$U\bar{y} = L^{-1}\bar{a} \tag{32.10}$$

Since L^{-1} is lower triangular, this representation expresses y_t as a function of

- lagged y's (via the term $U\bar{y}$), and
- current and future a's (via the term $L^{-1}\bar{a}$)

Because there are zeros everywhere in the matrix on the left of (32.7) except on the diagonal, super-diagonal, and sub-diagonal, the LU decomposition takes

- L to be zero except in the diagonal and the leading sub-diagonal.
- U to be zero except on the diagonal and the super-diagonal.

Thus, (32.10) has the form

$$\begin{bmatrix} 1 & U_{12} & 0 & 0 & \dots & 0 & 0 \\ 0 & 1 & U_{23} & 0 & \dots & 0 & 0 \\ 0 & 0 & 1 & U_{34} & \dots & 0 & 0 \\ 0 & 0 & 0 & 1 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & 1 & U_{N,N+1} \\ 0 & 0 & 0 & 0 & \dots & 0 & 1 \end{bmatrix} \begin{bmatrix} y_N \\ y_{N-1} \\ y_{N-2} \\ y_{N-3} \\ \vdots \\ y_1 \\ y_0 \end{bmatrix} =$$

$$\begin{bmatrix} L_{11}^{-1} & 0 & 0 & \dots & 0 \\ L_{21}^{-1} & L_{22}^{-1} & 0 & \dots & 0 \\ L_{31}^{-1} & L_{32}^{-1} & L_{33}^{-1} & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ L_{N,1}^{-1} & L_{N,2}^{-1} & L_{N,3}^{-1} & \dots & 0 \\ L_{N+1,1}^{-1} & L_{N+1,2}^{-1} & L_{N+1,3}^{-1} & \dots & L_{N+1\,N+1}^{-1} \end{bmatrix} \begin{bmatrix} a_N \\ a_{N-1} \\ a_{N-2} \\ \vdots \\ a_1 \\ a_0 - \phi_1 y_{-1} \end{bmatrix}$$

where L_{ij}^{-1} is the (i,j) element of L^{-1} and U_{ij} is the (i,j) element of U.

Note how the left side for a given t involves y_t and one lagged value y_{t-1} while the right side involves all future values of the forcing process $a_t, a_{t+1}, \ldots, a_N$.

Additional Lag Terms

We briefly indicate how this approach extends to the problem with $m > 1$.

Assume that $\beta = 1$ and let D_{m+1} be the $(m+1) \times (m+1)$ symmetric matrix whose elements are determined from the following formula:

$$D_{jk} = d_0 d_{k-j} + d_1 d_{k-j+1} + \ldots + d_{j-1} d_{k-1}, \qquad k \geq j$$

Let I_{m+1} be the $(m+1) \times (m+1)$ identity matrix.

Let ϕ_j be the coefficients in the expansion $\phi(L) = h + d(L^{-1})d(L)$.

Then the first order conditions (32.4) and (32.5) can be expressed as:

$$(D_{m+1} + hI_{m+1}) \begin{bmatrix} y_N \\ y_{N-1} \\ \vdots \\ y_{N-m} \end{bmatrix} = \begin{bmatrix} a_N \\ a_{N-1} \\ \vdots \\ a_{N-m} \end{bmatrix} + M \begin{bmatrix} y_{N-m+1} \\ y_{N-m-2} \\ \vdots \\ y_{N-2m} \end{bmatrix}$$

where M is $(m+1) \times m$ and

$$M_{ij} = \begin{cases} D_{i-j,\, m+1} \text{ for } i > j \\ 0 \text{ for } i \leq j \end{cases}$$

$$\begin{aligned} \phi_m y_{N-1} + \phi_{m-1} y_{N-2} + \ldots + \phi_0 y_{N-m-1} + \phi_1 y_{N-m-2} + \\ \ldots + \phi_m y_{N-2m-1} = a_{N-m-1} \\ \phi_m y_{N-2} + \phi_{m-1} y_{N-3} + \ldots + \phi_0 y_{N-m-2} + \phi_1 y_{N-m-3} + \\ \ldots + \phi_m y_{N-2m-2} = a_{N-m-2} \\ \vdots \\ \phi_m y_{m+1} + \phi_{m-1} y_m + + \ldots + \phi_0 y_1 + \phi_1 y_0 + \phi_m y_{-m+1} = a_1 \\ \phi_m y_m + \phi_{m-1} y_{m-1} + \phi_{m-2} + \ldots + \phi_0 y_0 + \phi_1 y_{-1} + \ldots + \phi_m y_{-m} = a_0 \end{aligned}$$

As before, we can express this equation as $W\bar{y} = \bar{a}$.

The matrix on the left of this equation is "almost" Toeplitz, the exception being the leading $m \times m$ submatrix in the upper left-hand corner.

We can represent the solution in feedback-feedforward form by obtaining a decomposition $LU = W$, and obtain

$$U\bar{y} = L^{-1}\bar{a} \tag{32.11}$$

$$\sum_{j=0}^{t} U_{-t+N+1,\, -t+N+j+1}\, y_{t-j} = \sum_{j=0}^{N-t} L_{-t+N+1,\, -t+N+1-j}\, \bar{a}_{t+j}\,, \qquad t = 0, 1, \ldots, N$$

where $L^{-1}_{t,s}$ is the element in the (t, s) position of L, and similarly for U.

The left side of equation (32.11) is the "feedback" part of the optimal control law for y_t, while the right-hand side is the "feedforward" part.

We note that there is a different control law for each t.

Thus, in the finite horizon case, the optimal control law is time-dependent.

It is natural to suspect that as $N \to \infty$, (32.11) becomes equivalent to the solution of our infinite horizon problem, which below we shall show can be expressed as

$$c(L)y_t = c(\beta L^{-1})^{-1}a_t \ ,$$

so that as $N \to \infty$ we expect that for each fixed $t, U^{-1}_{t,t-j} \to c_j$ and $L_{t,t+j}$ approaches the coefficient on L^{-j} in the expansion of $c(\beta L^{-1})^{-1}$.

This suspicion is true under general conditions that we shall study later.

For now, we note that by creating the matrix W for large N and factoring it into the LU form, good approximations to $c(L)$ and $c(\beta L^{-1})^{-1}$ can be obtained.

32.4 Infinite Horizon Limit

For the infinite horizon problem, we propose to discover first-order necessary conditions by taking the limits of (32.4) and (32.5) as $N \to \infty$.

This approach is valid, and the limits of (32.4) and (32.5) as N approaches infinity are first-order necessary conditions for a maximum.

However, for the infinite horizon problem with $\beta < 1$, the limits of (32.4) and (32.5) are, in general, not sufficient for a maximum.

That is, the limits of (32.5) do not provide enough information uniquely to determine the solution of the Euler equation (32.4) that maximizes (32.1).

As we shall see below, a side condition on the path of y_t that together with (32.4) is sufficient for an optimum is

$$\sum_{t=0}^{\infty} \beta^t \, h y_t^2 < \infty \tag{32.12}$$

All paths that satisfy the Euler equations, except the one that we shall select below, violate this condition and, therefore, evidently lead to (much) lower values of (32.1) than does the optimal path selected by the solution procedure below.

Consider the *characteristic equation* associated with the Euler equation

$$h + d\left(\beta z^{-1}\right) d\left(z\right) = 0 \tag{32.13}$$

Notice that if $\tilde{z}$ is a root of equation (32.13), then so is $\beta \tilde{z}^{-1}$.

Thus, the roots of (32.13) come in "β-reciprocal" pairs.

Assume that the roots of (32.13) are distinct.

Let the roots be, in descending order according to their moduli, $z_1, z_2, \ldots, z_{2m}$.

From the reciprocal pairs property and the assumption of distinct roots, it follows that $|z_j| > \sqrt{\beta}$ for $j \leq m$ and $|z_j| < \sqrt{\beta}$ for $j > m$.

It also follows that $z_{2m-j} = \beta z_{j+1}^{-1}, j = 0, 1, \ldots, m-1$.

Therefore, the characteristic polynomial on the left side of (32.13) can be expressed as

$$\begin{aligned} h + d(\beta z^{-1})d(z) &= z^{-m} z_0 (z - z_1) \cdots (z - z_m)(z - z_{m+1}) \cdots (z - z_{2m}) \\ &= z^{-m} z_0 (z - z_1)(z - z_2) \cdots (z - z_m)(z - \beta z_m^{-1}) \cdots (z - \beta z_2^{-1})(z - \beta z_1^{-1}) \end{aligned} \tag{32.14}$$

where z_0 is a constant.

In (32.14), we substitute $(z - z_j) = -z_j(1 - \frac{1}{z_j}z)$ and $(z - \beta z_j^{-1}) = z(1 - \frac{\beta}{z_j}z^{-1})$ for $j = 1, \ldots, m$ to get

$$h + d(\beta z^{-1})d(z) = (-1)^m(z_0 z_1 \cdots z_m)(1 - \frac{1}{z_1}z)\cdots(1 - \frac{1}{z_m}z)(1 - \frac{1}{z_1}\beta z^{-1})\cdots(1 - \frac{1}{z_m}\beta z^{-1})$$

Now define $c(z) = \sum_{j=0}^{m} c_j z^j$ as

$$c(z) = \left[(-1)^m z_0 z_1 \cdots z_m\right]^{1/2}(1 - \frac{z}{z_1})(1 - \frac{z}{z_2})\cdots(1 - \frac{z}{z_m}) \tag{32.15}$$

Notice that (32.14) can be written

$$h + d(\beta z^{-1})\, d(z) = c(\beta z^{-1})\, c(z) \tag{32.16}$$

It is useful to write (32.15) as

$$c(z) = c_0(1 - \lambda_1 z)\ldots(1 - \lambda_m z) \tag{32.17}$$

where

$$c_0 = \left[(-1)^m z_0 z_1 \cdots z_m\right]^{1/2}; \quad \lambda_j = \frac{1}{z_j}, \ \ j = 1, \ldots, m$$

Since $|z_j| > \sqrt{\beta}$ for $j = 1, \ldots, m$ it follows that $|\lambda_j| < 1/\sqrt{\beta}$ for $j = 1, \ldots, m$.

Using (32.17), we can express the factorization (32.16) as

$$h + d(\beta z^{-1})d(z) = c_0^2(1 - \lambda_1 z)\cdots(1 - \lambda_m z)(1 - \lambda_1\beta z^{-1})\cdots(1 - \lambda_m\beta z^{-1})$$

In sum, we have constructed a factorization (32.16) of the characteristic polynomial for the Euler equation in which the zeros of $c(z)$ exceed $\beta^{1/2}$ in modulus, and the zeros of $c(\beta z^{-1})$ are less than $\beta^{1/2}$ in modulus.

Using (32.16), we now write the Euler equation as

$$c(\beta L^{-1})\, c(L)\, y_t = a_t$$

The unique solution of the Euler equation that satisfies condition (32.12) is

$$c(L)\, y_t = c(\beta L^{-1})^{-1} a_t \tag{32.18}$$

This can be established by using an argument paralleling that in chapter IX of [Sargent, 1987].

To exhibit the solution in a form paralleling that of [Sargent, 1987], we use (32.17) to write (32.18) as

$$(1 - \lambda_1 L)\cdots(1 - \lambda_m L)y_t = \frac{c_0^{-2} a_t}{(1 - \beta\lambda_1 L^{-1})\cdots(1 - \beta\lambda_m L^{-1})} \tag{32.19}$$

Using partial fractions, we can write the characteristic polynomial on the right side of (32.19) as

$$\sum_{j=1}^{m} \frac{A_j}{1 - \lambda_j \beta L^{-1}} \quad \text{where} \quad A_j := \frac{c_0^{-2}}{\prod_{i \neq j}(1 - \frac{\lambda_i}{\lambda_j})}$$

Then (32.19) can be written

$$(1 - \lambda_1 L)\cdots(1 - \lambda_m L)y_t = \sum_{j=1}^{m} \frac{A_j}{1 - \lambda_j \beta L^{-1}} a_t$$

or

$$(1 - \lambda_1 L) \cdots (1 - \lambda_m L) y_t = \sum_{j=1}^{m} A_j \sum_{k=0}^{\infty} (\lambda_j \beta)^k \, a_{t+k} \tag{32.20}$$

Equation (32.20) expresses the optimum sequence for y_t in terms of m lagged y's, and m weighted infinite geometric sums of future a_t's.

Furthermore, (32.20) is the unique solution of the Euler equation that satisfies the initial conditions and condition (32.12).

In effect, condition (32.12) compels us to solve the "unstable" roots of $h + d(\beta z^{-1}) d(z)$ forward (see [Sargent, 1987]).

The step of factoring the polynomial $h + d(\beta z^{-1})\, d(z)$ into $c\,(\beta z^{-1}) c\,(z)$, where the zeros of $c\,(z)$ all have modulus exceeding $\sqrt{\beta}$, is central to solving the problem.

We note two features of the solution (32.20)

- Since $|\lambda_j| < 1/\sqrt{\beta}$ for all j, it follows that $(\lambda_j\, \beta) < \sqrt{\beta}$.
- The assumption that $\{a_t\}$ is of exponential order less than $1/\sqrt{\beta}$ is sufficient to guarantee that the geometric sums of future a_t's on the right side of (32.20) converge.

We immediately see that those sums will converge under the weaker condition that $\{a_t\}$ is of exponential order less than ϕ^{-1} where $\phi = \max \{\beta \lambda_i, i = 1, \ldots, m\}$.

Note that with a_t identically zero, (32.20) implies that in general $|y_t|$ eventually grows exponentially at a rate given by $\max_i |\lambda_i|$.

The condition $\max_i |\lambda_i| < 1/\sqrt{\beta}$ guarantees that condition (32.12) is satisfied.

In fact, $\max_i |\lambda_i| < 1/\sqrt{\beta}$ is a necessary condition for (32.12) to hold.

Were (32.12) not satisfied, the objective function would diverge to $-\infty$, implying that the y_t path could not be optimal.

For example, with $a_t = 0$, for all t, it is easy to describe a naive (nonoptimal) policy for $\{y_t, t \geq 0\}$ that gives a finite value of (32.17).

We can simply let $y_t = 0$ for $t \geq 0$.

This policy involves at most m nonzero values of $h y_t^2$ and $[d(L) y_t]^2$, and so yields a finite value of (32.1).

Therefore it is easy to dominate a path that violates (32.12).

32.5 Undiscounted Problems

It is worthwhile focusing on a special case of the LQ problems above: the undiscounted problem that emerges when $\beta = 1$.

In this case, the Euler equation is

$$\left(h + d(L^{-1}) d(L)\right) y_t = a_t$$

The factorization of the characteristic polynomial (32.16) becomes

$$\left(h + d\,(z^{-1}) d(z)\right) = c\,(z^{-1})\, c\,(z)$$

where

$$
\begin{aligned}
c\left(z\right) &= c_0(1-\lambda_1 z)\ldots(1-\lambda_m z) \\
c_0 &= \left[(-1)^m z_0 z_1 \ldots z_m\right] \\
|\lambda_j| &< 1 \text{ for } j = 1,\ldots,m \\
\lambda_j &= \frac{1}{z_j} \text{ for } j = 1,\ldots,m \\
z_0 &= \text{ constant}
\end{aligned}
$$

The solution of the problem becomes

$$
(1-\lambda_1 L)\cdots(1-\lambda_m L)y_t = \sum_{j=1}^{m} A_j \sum_{k=0}^{\infty} \lambda_j^k a_{t+k}
$$

32.5.1 Transforming Discounted to Undiscounted Problem

Discounted problems can always be converted into undiscounted problems via a simple transformation.

Consider problem (32.1) with $0 < \beta < 1$.

Define the transformed variables

$$
\tilde{a}_t = \beta^{t/2} a_t,\ \tilde{y}_t = \beta^{t/2} y_t \tag{32.21}
$$

Then notice that $\beta^t \left[d\left(L\right) y_t\right]^2 = [\tilde{d}\left(L\right)\tilde{y}_t]^2$ with $\tilde{d}\left(L\right) = \sum_{j=0}^{m} \tilde{d}_j L^j$ and $\tilde{d}_j = \beta^{j/2} d_j$.

Then the original criterion function (32.1) is equivalent to

$$
\lim_{N\to\infty} \sum_{t=0}^{N} \{\tilde{a}_t\, \tilde{y}_t - \frac{1}{2} h\, \tilde{y}_t^2 - \frac{1}{2}[\tilde{d}\left(L\right)\tilde{y}_t]^2\} \tag{32.22}
$$

which is to be maximized over sequences $\{\tilde{y}_t,\ t = 0,\ldots\}$ subject to $\tilde{y}_{-1},\cdots,\tilde{y}_{-m}$ given and $\{\tilde{a}_t,\ t = 1,\ldots\}$ a known bounded sequence.

The Euler equation for this problem is $[h + \tilde{d}\left(L^{-1}\right)\tilde{d}\left(L\right)]\,\tilde{y}_t = \tilde{a}_t$.

The solution is

$$
(1-\tilde{\lambda}_1 L)\cdots(1-\tilde{\lambda}_m L)\,\tilde{y}_t = \sum_{j=1}^{m} \tilde{A}_j \sum_{k=0}^{\infty} \tilde{\lambda}_j^k\, \tilde{a}_{t+k}
$$

or

$$
\tilde{y}_t = \tilde{f}_1\, \tilde{y}_{t-1} + \cdots + \tilde{f}_m\, \tilde{y}_{t-m} + \sum_{j=1}^{m} \tilde{A}_j \sum_{k=0}^{\infty} \tilde{\lambda}_j^k\, \tilde{a}_{t+k}, \tag{32.23}
$$

where $\tilde{c}\left(z^{-1}\right)\tilde{c}\left(z\right) = h + \tilde{d}\left(z^{-1}\right)\tilde{d}\left(z\right)$, and where

$$
\left[(-1)^m\, \tilde{z}_0 \tilde{z}_1 \ldots \tilde{z}_m\right]^{1/2}(1-\tilde{\lambda}_1\, z)\ldots(1-\tilde{\lambda}_m\, z) = \tilde{c}\left(z\right),\ \text{where}\ |\tilde{\lambda}_j| < 1
$$

We leave it to the reader to show that (32.23) implies the equivalent form of the solution

$$
y_t = f_1\, y_{t-1} + \cdots + f_m\, y_{t-m} + \sum_{j=1}^{m} A_j \sum_{k=0}^{\infty} (\lambda_j\, \beta)^k\, a_{t+k}
$$

where

$$
f_j = \tilde{f}_j\, \beta^{-j/2},\ A_j = \tilde{A}_j,\ \lambda_j = \tilde{\lambda}_j\, \beta^{-1/2} \tag{32.24}
$$

The transformations (32.21) and the inverse formulas (32.24) allow us to solve a discounted problem by first solving a related undiscounted problem.

32.6 Implementation

Here's the code that computes solutions to the LQ problem using the methods described above.

```
import numpy as np
import scipy.stats as spst
import scipy.linalg as la

class LQFilter:

    def __init__(self, d, h, y_m, r=None, h_eps=None, β=None):
        """

        Parameters
        ----------
            d : list or numpy.array (1-D or a 2-D column vector)
                    The order of the coefficients: [d_0, d_1, ..., d_m]
            h : scalar
                    Parameter of the objective function (corresponding to the
                    quadratic term)
            y_m : list or numpy.array (1-D or a 2-D column vector)
                    Initial conditions for y
            r : list or numpy.array (1-D or a 2-D column vector)
                    The order of the coefficients: [r_0, r_1, ..., r_k]
                    (optional, if not defined -> deterministic problem)
            β : scalar
                    Discount factor (optional, default value is one)
        """

        self.h = h
        self.d = np.asarray(d)
        self.m = self.d.shape[0] - 1

        self.y_m = np.asarray(y_m)

        if self.m == self.y_m.shape[0]:
            self.y_m = self.y_m.reshape(self.m, 1)
        else:
            raise ValueError("y_m must be of length m = {self.m:d}")

        #-----------------------------------------
        # Define the coefficients of ϕ upfront
        #-----------------------------------------
        ϕ = np.zeros(2 * self.m + 1)
        for i in range(- self.m, self.m + 1):
            ϕ[self.m - i] = np.sum(np.diag(self.d.reshape(self.m + 1, 1) \
                                           @ self.d.reshape(1, self.m + 1),
                                           k=-i
                                           )
                                   )
        ϕ[self.m] = ϕ[self.m] + self.h
        self.ϕ = ϕ

        #-----------------------------------------------------
        # If r is given calculate the vector ϕ_r
        #-----------------------------------------------------
        if r is None:
```

(continues on next page)

(continued from previous page)

```
            pass
        else:
            self.r = np.asarray(r)
            self.k = self.r.shape[0] - 1
            ϕ_r = np.zeros(2 * self.k + 1)
            for i in range(- self.k, self.k + 1):
                ϕ_r[self.k - i] = np.sum(np.diag(self.r.reshape(self.k + 1, 1) \
                                                 @ self.r.reshape(1, self.k + 1),
                                                 k=-i
                                                 )
                                         )
            if h_eps is None:
                self.ϕ_r = ϕ_r
            else:
                ϕ_r[self.k] = ϕ_r[self.k] + h_eps
                self.ϕ_r = ϕ_r

        #-----------------------------------------------------
        # If β is given, define the transformed variables
        #-----------------------------------------------------
        if β is None:
            self.β = 1
        else:
            self.β = β
            self.d = self.β**(np.arange(self.m + 1)/2) * self.d
            self.y_m = self.y_m * (self.β**(- np.arange(1, self.m + 1)/2)) \
                                .reshape(self.m, 1)

    def construct_W_and_Wm(self, N):
        """
        This constructs the matrices W and W_m for a given number of periods N
        """

        m = self.m
        d = self.d

        W = np.zeros((N + 1, N + 1))
        W_m = np.zeros((N + 1, m))

        #---------------------------------------
        # Terminal conditions
        #---------------------------------------

        D_m1 = np.zeros((m + 1, m + 1))
        M = np.zeros((m + 1, m))

        # (1) Constuct the D_{m+1} matrix using the formula

        for j in range(m + 1):
            for k in range(j, m + 1):
                D_m1[j, k] = d[:j + 1] @ d[k - j: k + 1]

        # Make the matrix symmetric
        D_m1 = D_m1 + D_m1.T - np.diag(np.diag(D_m1))

        # (2) Construct the M matrix using the entries of D_m1
```

(continues on next page)

(continued from previous page)

```
        for j in range(m):
            for i in range(j + 1, m + 1):
                M[i, j] = D_m1[i - j - 1, m]

        #-----------------------------------------------
        # Euler equations for t = 0, 1, ..., N-(m+1)
        #-----------------------------------------------
        ϕ = self.ϕ

        W[:(m + 1), :(m + 1)] = D_m1 + self.h * np.eye(m + 1)
        W[:(m + 1), (m + 1):(2 * m + 1)] = M

        for i, row in enumerate(np.arange(m + 1, N + 1 - m)):
            W[row, (i + 1):(2 * m + 2 + i)] = ϕ

        for i in range(1, m + 1):
            W[N - m + i, -(2 * m + 1 - i):] = ϕ[:-i]

        for i in range(m):
            W_m[N - i, :(m - i)] = ϕ[(m + 1 + i):]

        return W, W_m

    def roots_of_characteristic(self):
        """
        This function calculates z_0 and the 2m roots of the characteristic
        equation associated with the Euler equation (1.7)

        Note:
        ------
        numpy.poly1d(roots, True) defines a polynomial using its roots that can
        be evaluated at any point. If x_1, x_2, ... , x_m are the roots then
            p(x) = (x - x_1)(x - x_2)...(x - x_m)
        """
        m = self.m
        ϕ = self.ϕ

        # Calculate the roots of the 2m-polynomial
        roots = np.roots(ϕ)
        # Sort the roots according to their length (in descending order)
        roots_sorted = roots[np.argsort(abs(roots))[::-1]]

        z_0 = ϕ.sum() / np.poly1d(roots, True)(1)
        z_1_to_m = roots_sorted[:m]     # We need only those outside the unit circle

        λ = 1 / z_1_to_m

        return z_1_to_m, z_0, λ

    def coeffs_of_c(self):
        '''
        This function computes the coefficients {c_j, j = 0, 1, ..., m} for
                c(z) = sum_{j = 0}^{m} c_j z^j

        Based on the expression (1.9). The order is
```

(continues on next page)

(continued from previous page)

```
            c_coeffs = [c_0, c_1, ..., c_{m-1}, c_m]
        '''
        z_1_to_m, z_0 = self.roots_of_characteristic()[:2]

        c_0 = (z_0 * np.prod(z_1_to_m).real * (- 1)**self.m)**(.5)
        c_coeffs = np.poly1d(z_1_to_m, True).c * z_0 / c_0

        return c_coeffs[::-1]

    def solution(self):
        """
        This function calculates {λ_j, j=1,...,m} and {A_j, j=1,...,m}
        of the expression (1.15)
        """
        λ = self.roots_of_characteristic()[2]
        c_0 = self.coeffs_of_c()[-1]

        A = np.zeros(self.m, dtype=complex)
        for j in range(self.m):
            denom = 1 - λ/λ[j]
            A[j] = c_0**(-2) / np.prod(denom[np.arange(self.m) != j])

        return λ, A

    def construct_V(self, N):
        '''
        This function constructs the covariance matrix for x^N (see section 6)
        for a given period N
        '''
        V = np.zeros((N, N))
        ϕ_r = self.ϕ_r

        for i in range(N):
            for j in range(N):
                if abs(i-j) <= self.k:
                    V[i, j] = ϕ_r[self.k + abs(i-j)]

        return V

    def simulate_a(self, N):
        """
        Assuming that the u's are normal, this method draws a random path
        for x^N
        """
        V = self.construct_V(N + 1)
        d = spst.multivariate_normal(np.zeros(N + 1), V)

        return d.rvs()

    def predict(self, a_hist, t):
        """
        This function implements the prediction formula discussed in section 6 (1.59)
        It takes a realization for a^N, and the period in which the prediction is
        formed

        Output:  E[abar | a_t, a_{t-1}, ..., a_1, a_0]
```

(continues on next page)

(continued from previous page)

```
        """

        N = np.asarray(a_hist).shape[0] - 1
        a_hist = np.asarray(a_hist).reshape(N + 1, 1)
        V = self.construct_V(N + 1)

        aux_matrix = np.zeros((N + 1, N + 1))
        aux_matrix[:(t + 1), :(t + 1)] = np.eye(t + 1)
        L = la.cholesky(V).T
        Ea_hist = la.inv(L) @ aux_matrix @ L @ a_hist

        return Ea_hist

    def optimal_y(self, a_hist, t=None):
        """
        - if t is NOT given it takes a_hist (list or numpy.array) as a
          deterministic a_t
        - if t is given, it solves the combined control prediction problem
          (section 7)(by default, t == None -> deterministic)

        for a given sequence of a_t (either deterministic or a particular
        realization), it calculates the optimal y_t sequence using the method
        of the lecture

        Note:
        ------
        scipy.linalg.lu normalizes L, U so that L has unit diagonal elements
        To make things consistent with the lecture, we need an auxiliary
        diagonal matrix D which renormalizes L and U
        """

        N = np.asarray(a_hist).shape[0] - 1
        W, W_m = self.construct_W_and_Wm(N)

        L, U = la.lu(W, permute_l=True)
        D = np.diag(1 / np.diag(U))
        U = D @ U
        L = L @ np.diag(1 / np.diag(D))

        J = np.fliplr(np.eye(N + 1))

        if t is None:   # If the problem is deterministic

            a_hist = J @ np.asarray(a_hist).reshape(N + 1, 1)

            #--------------------------------------------
            # Transform the 'a' sequence if β is given
            #--------------------------------------------
            if self.β != 1:
                a_hist =  a_hist * (self.β**(np.arange(N + 1) / 2))[::-1] \
                                 .reshape(N + 1, 1)

            a_bar = a_hist - W_m @ self.y_m             # a_bar from the lecture
            Uy = np.linalg.solve(L, a_bar)              # U @ y_bar = L^{-1}
            y_bar = np.linalg.solve(U, Uy)              # y_bar = U^{-1}L^{-1}
```

(continues on next page)

(continued from previous page)

```
            # Reverse the order of y_bar with the matrix J
            J = np.fliplr(np.eye(N + self.m + 1))
            # y_hist : concatenated y_m and y_bar
            y_hist = J @ np.vstack([y_bar, self.y_m])

            #-----------------------------------------
            # Transform the optimal sequence back if β is given
            #-----------------------------------------
            if self.β != 1:
                y_hist = y_hist * (self.β**(- np.arange(-self.m, N + 1)/2)) \
                                  .reshape(N + 1 + self.m, 1)

            return y_hist, L, U, y_bar

        else:           # If the problem is stochastic and we look at it

            Ea_hist = self.predict(a_hist, t).reshape(N + 1, 1)
            Ea_hist = J @ Ea_hist

            a_bar = Ea_hist - W_m @ self.y_m            # a_bar from the lecture
            Uy = np.linalg.solve(L, a_bar)              # U @ y_bar = L^{-1}
            y_bar = np.linalg.solve(U, Uy)              # y_bar = U^{-1}L^{-1}

            # Reverse the order of y_bar with the matrix J
            J = np.fliplr(np.eye(N + self.m + 1))
            # y_hist : concatenated y_m and y_bar
            y_hist = J @ np.vstack([y_bar, self.y_m])

            return y_hist, L, U, y_bar
```

32.6.1 Example

In this application, we'll have one lag, with

$$d(L)y_t = \gamma(I - L)y_t = \gamma(y_t - y_{t-1})$$

Suppose for the moment that $\gamma = 0$.

Then the intertemporal component of the LQ problem disappears, and the agent simply wants to maximize $a_t y_t - h y_t^2/2$ in each period.

This means that the agent chooses $y_t = a_t/h$.

In the following we'll set $h = 1$, so that the agent just wants to track the $\{a_t\}$ process.

However, as we increase γ, the agent gives greater weight to a smooth time path.

Hence $\{y_t\}$ evolves as a smoothed version of $\{a_t\}$.

The $\{a_t\}$ sequence we'll choose as a stationary cyclic process plus some white noise.

Here's some code that generates a plot when $\gamma = 0.8$

```
# Set seed and generate a_t sequence
np.random.seed(123)
n = 100
```

(continues on next page)

(continued from previous page)

```
a_seq = np.sin(np.linspace(0, 5 * np.pi, n)) + 2 + 0.1 * np.random.randn(n)

def plot_simulation(y=0.8, m=1, h=1, y_m=2):

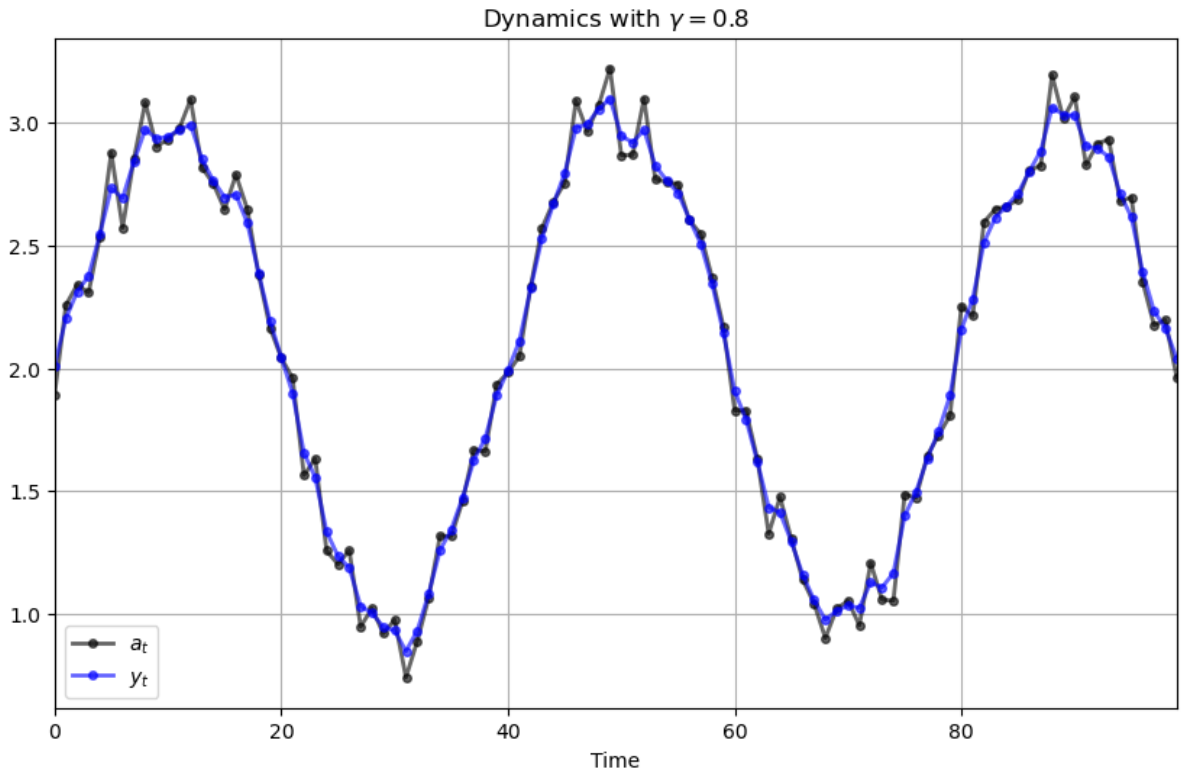
    d = y * np.asarray([1, -1])
    y_m = np.asarray(y_m).reshape(m, 1)

    testlq = LQFilter(d, h, y_m)
    y_hist, L, U, y = testlq.optimal_y(a_seq)
    y = y[::-1]  # Reverse y

    # Plot simulation results

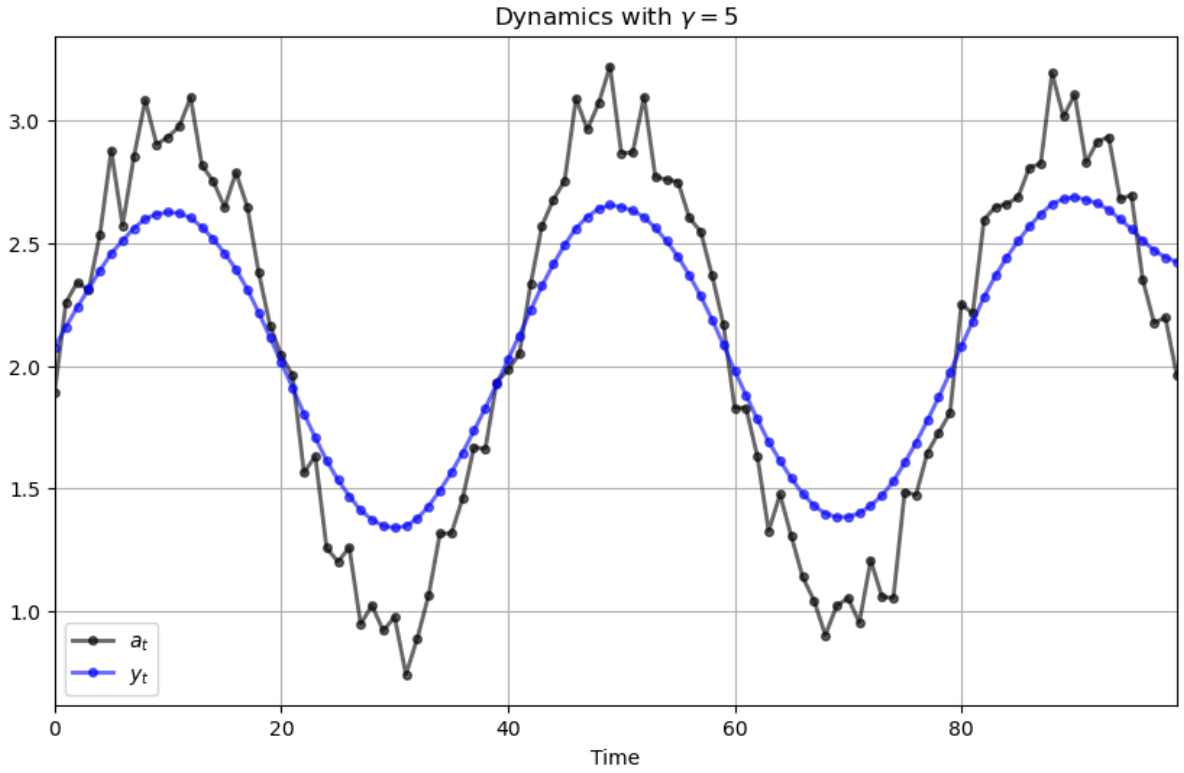
    fig, ax = plt.subplots(figsize=(10, 6))
    p_args = {'lw' : 2, 'alpha' : 0.6}
    time = range(len(y))
    ax.plot(time, a_seq / h, 'k-o', ms=4, lw=2, alpha=0.6, label='$a_t$')
    ax.plot(time, y, 'b-o', ms=4, lw=2, alpha=0.6, label='$y_t$')
    ax.set(title=rf'Dynamics with $\gamma = {y}$',
           xlabel='Time',
           xlim=(0, max(time))
          )
    ax.legend()
    ax.grid()
    plt.show()

plot_simulation()
```



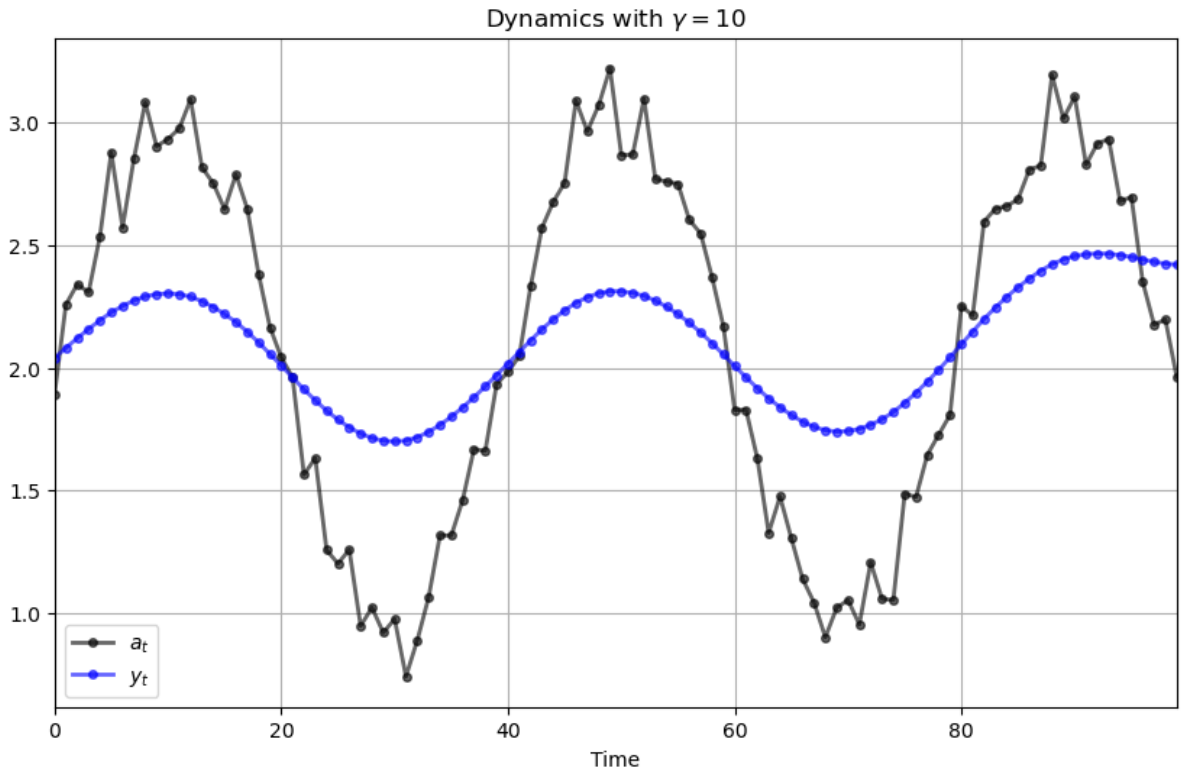
Here's what happens when we change γ to 5.0

```
plot_simulation(γ=5)
```



And here's $\gamma = 10$

```
plot_simulation(γ=10)
```



32.7 Exercises

Exercise 32.7.1

Consider solving a discounted version ($\beta < 1$) of problem (32.1), as follows.

Convert (32.1) to the undiscounted problem (32.22).

Let the solution of (32.22) in feedback form be

$$(1 - \tilde{\lambda}_1 L) \cdots (1 - \tilde{\lambda}_m L)\tilde{y}_t = \sum_{j=1}^{m} \tilde{A}_j \sum_{k=0}^{\infty} \tilde{\lambda}_j^k \tilde{a}_{t+k}$$

or

$$\tilde{y}_t = \tilde{f}_1 \tilde{y}_{t-1} + \cdots + \tilde{f}_m \tilde{y}_{t-m} + \sum_{j=1}^{m} \tilde{A}_j \sum_{k=0}^{\infty} \tilde{\lambda}_j^k \tilde{a}_{t+k} \tag{32.25}$$

Here

- $h + \tilde{d}(z^{-1})\tilde{d}(z) = \tilde{c}(z^{-1})\tilde{c}(z)$
- $\tilde{c}(z) = [(-1)^m \tilde{z}_0 \tilde{z}_1 \cdots \tilde{z}_m]^{1/2}(1 - \tilde{\lambda}_1 z) \cdots (1 - \tilde{\lambda}_m z)$

where the $\tilde{z}_j$ are the zeros of $h + \tilde{d}(z^{-1})\,\tilde{d}(z)$.

Prove that (32.25) implies that the solution for y_t in feedback form is

$$y_t = f_1 y_{t-1} + \ldots + f_m y_{t-m} + \sum_{j=1}^{m} A_j \sum_{k=0}^{\infty} \beta^k \lambda_j^k a_{t+k}$$

where $f_j = \tilde{f}_j \beta^{-j/2}$, $A_j = \tilde{A}_j$, and $\lambda_j = \tilde{\lambda}_j \beta^{-1/2}$.

Exercise 32.7.2

Solve the optimal control problem, maximize

$$\sum_{t=0}^{2} \left\{ a_t y_t - \frac{1}{2}[(1-2L)y_t]^2 \right\}$$

subject to y_{-1} given, and $\{a_t\}$ a known bounded sequence.

Express the solution in the "feedback form" (32.20), giving numerical values for the coefficients.

Make sure that the boundary conditions (32.5) are satisfied.

Note: This problem differs from the problem in the text in one important way: instead of $h > 0$ in (32.1), $h = 0$. This has an important influence on the solution.

Exercise 32.7.3

Solve the infinite time-optimal control problem to maximize

$$\lim_{N \to \infty} \sum_{t=0}^{N} -\frac{1}{2}[(1-2L)y_t]^2,$$

subject to y_{-1} given. Prove that the solution is

$$y_t = 2y_{t-1} = 2^{t+1} y_{-1} \qquad t > 0$$

Exercise 32.7.4

Solve the infinite time problem, to maximize

$$\lim_{N \to \infty} \sum_{t=0}^{N} (.0000001)\, y_t^2 - \frac{1}{2}[(1-2L)y_t]^2$$

subject to y_{-1} given. Prove that the solution $y_t = 2y_{t-1}$ violates condition (32.12), and so is not optimal.

Prove that the optimal solution is approximately $y_t = .5y_{t-1}$.

CHAPTER

THIRTYTHREE

CLASSICAL PREDICTION AND FILTERING WITH LINEAR ALGEBRA

33.1 Overview

This is a sequel to the earlier lecture *Classical Control with Linear Algebra*.

That lecture used linear algebra – in particular, the LU decomposition – to formulate and solve a class of linear-quadratic optimal control problems.

In this lecture, we'll be using a closely related decomposition, the Cholesky decomposition, to solve linear prediction and filtering problems.

We exploit the useful fact that there is an intimate connection between two superficially different classes of problems:

- deterministic linear-quadratic (LQ) optimal control problems
- linear least squares prediction and filtering problems

The first class of problems involves no randomness, while the second is all about randomness.

Nevertheless, essentially the same mathematics solves both types of problem.

This connection, which is often termed "duality," is present whether one uses "classical" or "recursive" solution procedures.

In fact, we saw duality at work earlier when we formulated control and prediction problems recursively in lectures LQ dynamic programming problems, A first look at the Kalman filter, and The permanent income model.

A useful consequence of duality is that

- With every LQ control problem, there is implicitly affiliated a linear least squares prediction or filtering problem.
- With every linear least squares prediction or filtering problem there is implicitly affiliated a LQ control problem.

An understanding of these connections has repeatedly proved useful in cracking interesting applied problems.

For example, Sargent [Sargent, 1987] [chs. IX, XIV] and Hansen and Sargent [Hansen and Sargent, 1980] formulated and solved control and filtering problems using z-transform methods.

In this lecture, we begin to investigate these ideas by using mostly elementary linear algebra.

This is the main purpose and focus of the lecture.

However, after showing matrix algebra formulas, we'll summarize classic infinite-horizon formulas built on z-transform and lag operator methods.

And we'll occasionally refer to some of these formulas from the infinite dimensional problems as we present the finite time formulas and associated linear algebra.

We'll start with the following standard import:

```
import numpy as np
```

33.1.1 References

Useful references include [Whittle, 1963], [Hansen and Sargent, 1980], [Orfanidis, 1988], [Athanasios and Pillai, 1991], and [Muth, 1960].

33.2 Finite Dimensional Prediction

Let $(x_1, x_2, \ldots, x_T)' = x$ be a $T \times 1$ vector of random variables with mean $\mathbb{E}x = 0$ and covariance matrix $\mathbb{E}xx' = V$.

Here V is a $T \times T$ positive definite matrix.

The i, j component $Ex_i x_j$ of V is the **inner product** between x_i and x_j.

We regard the random variables as being ordered in time so that x_t is thought of as the value of some economic variable at time t.

For example, x_t could be generated by the random process described by the Wold representation presented in equation (33.16) in the section below on infinite dimensional prediction and filtering.

In that case, V_{ij} is given by the coefficient on $z^{|i-j|}$ in the expansion of $g_x(z) = d(z)\, d(z^{-1}) + h$, which equals $h + \sum_{k=0}^{\infty} d_k d_{k+|i-j|}$.

We want to construct j step ahead linear least squares predictors of the form

$$\hat{\mathbb{E}}\left[x_T | x_{T-j}, x_{T-j+1}, \ldots, x_1\right]$$

where $\hat{\mathbb{E}}$ is the linear least squares projection operator.

(Sometimes $\hat{\mathbb{E}}$ is called the wide-sense expectations operator)

To find linear least squares predictors it is helpful first to construct a $T \times 1$ vector ε of random variables that form an orthonormal basis for the vector of random variables x.

The key insight here comes from noting that because the covariance matrix V of x is a positive definite and symmetric, there exists a (Cholesky) decomposition of V such that

$$V = L^{-1}(L^{-1})'$$

and

$$L\, V\, L' = I$$

where L and L^{-1} are both lower triangular.

Form the $T \times 1$ random vector $\varepsilon = Lx$.

The random vector ε is an orthonormal basis for x because

- L is nonsingular
- $\mathbb{E}\, \varepsilon\, \varepsilon' = L \mathbb{E} x x' L' = I$
- $x = L^{-1} \varepsilon$

It is enlightening to write out and interpret the equations $Lx = \varepsilon$ and $L^{-1}\varepsilon = x$.

First, we'll write $Lx = \varepsilon$

$$\begin{aligned} L_{11}x_1 &= \varepsilon_1 \\ L_{21}x_1 + L_{22}x_2 &= \varepsilon_2 \\ &\vdots \\ L_{T1}\, x_1 \, \ldots + L_{TT}x_T &= \varepsilon_T \end{aligned} \tag{33.1}$$

or

$$\sum_{j=0}^{t-1} L_{t,t-j}\, x_{t-j} = \varepsilon_t, \quad t = 1,\, 2, \ldots T \tag{33.2}$$

Next, we write $L^{-1}\varepsilon = x$

$$\begin{aligned} x_1 &= L_{11}^{-1}\varepsilon_1 \\ x_2 &= L_{22}^{-1}\varepsilon_2 + L_{21}^{-1}\varepsilon_1 \\ &\vdots \\ x_T &= L_{TT}^{-1}\varepsilon_T + L_{T,T-1}^{-1}\varepsilon_{T-1} \, \ldots + L_{T,1}^{-1}\varepsilon_1 \end{aligned} \quad , \tag{33.3}$$

or

$$x_t = \sum_{j=0}^{t-1} L_{t,t-j}^{-1}\, \varepsilon_{t-j} \tag{33.4}$$

where $L_{i,j}^{-1}$ denotes the i, j element of L^{-1}.

From (33.2), it follows that ε_t is in the linear subspace spanned by $x_t,\, x_{t-1}, \ldots,\, x_1$.

From (33.4) it follows that that x_t is in the linear subspace spanned by $\varepsilon_t,\, \varepsilon_{t-1}, \ldots, \varepsilon_1$.

Equation (33.2) forms a sequence of **autoregressions** that for $t = 1, \ldots, T$ express x_t as linear functions of $x_s, s = 1, \ldots, t-1$ and a random variable $(L_{t,t})^{-1}\varepsilon_t$ that is orthogonal to each componenent of $x_s, s = 1, \ldots, t-1$.

(Here $(L_{t,t})^{-1}$ denotes the reciprocal of $L_{t,t}$ while $L_{t,t}^{-1}$ denotes the t, t element of L^{-1}).

The equivalence of the subspaces spanned by $\varepsilon_t, \ldots, \varepsilon_1$ and $x_t, \ldots, x_1$ means that for $t - 1 \geq m \geq 1$

$$\hat{\mathbb{E}}[x_t \mid x_{t-m},\, x_{t-m-1}, \ldots, x_1] = \hat{\mathbb{E}}[x_t \mid \varepsilon_{t-m}, \varepsilon_{t-m-1}, \ldots, \varepsilon_1] \tag{33.5}$$

To proceed, it is useful to drill down and note that for $t - 1 \geq m \geq 1$ we can rewrite (33.4) in the form of the **moving average representation**

$$x_t = \sum_{j=0}^{m-1} L_{t,t-j}^{-1}\, \varepsilon_{t-j} + \sum_{j=m}^{t-1} L_{t,t-j}^{-1}\, \varepsilon_{t-j} \tag{33.6}$$

Representation (33.6) is an orthogonal decomposition of x_t into a part $\sum_{j=m}^{t-1} L_{t,t-j}^{-1}\, \varepsilon_{t-j}$ that lies in the space spanned by $[x_{t-m},\, x_{t-m+1},\, \ldots, x_1]$ and an orthogonal component $\sum_{j=m}^{t-1} L_{t,t-j}^{-1}\, \varepsilon_{t-j}$ that does not lie in that space but instead in a linear space knowns as its **orthogonal complement**.

It follows that

$$\hat{\mathbb{E}}[x_t \mid x_{t-m},\, x_{t-m-1}, \ldots, x_1] = \sum_{j=0}^{m-1} L_{t,t-j}^{-1}\, \varepsilon_{t-j}$$

33.2.1 Implementation

Here's the code that computes solutions to LQ control and filtering problems using the methods described here and in *Classical Control with Linear Algebra*.

```
import numpy as np
import scipy.stats as spst
import scipy.linalg as la

class LQFilter:

    def __init__(self, d, h, y_m, r=None, h_eps=None, β=None):
        """

        Parameters
        ----------
            d : list or numpy.array (1-D or a 2-D column vector)
                    The order of the coefficients: [d_0, d_1, ..., d_m]
            h : scalar
                    Parameter of the objective function (corresponding to the
                    quadratic term)
            y_m : list or numpy.array (1-D or a 2-D column vector)
                    Initial conditions for y
            r : list or numpy.array (1-D or a 2-D column vector)
                    The order of the coefficients: [r_0, r_1, ..., r_k]
                    (optional, if not defined -> deterministic problem)
            β : scalar
                    Discount factor (optional, default value is one)
        """

        self.h = h
        self.d = np.asarray(d)
        self.m = self.d.shape[0] - 1

        self.y_m = np.asarray(y_m)

        if self.m == self.y_m.shape[0]:
            self.y_m = self.y_m.reshape(self.m, 1)
        else:
            raise ValueError("y_m must be of length m = {self.m:d}")

        #-----------------------------------------
        # Define the coefficients of ϕ upfront
        #-----------------------------------------
        ϕ = np.zeros(2 * self.m + 1)
        for i in range(- self.m, self.m + 1):
            ϕ[self.m - i] = np.sum(np.diag(self.d.reshape(self.m + 1, 1) \
                                           @ self.d.reshape(1, self.m + 1),
                                           k=-i
                                           )
                                   )
        ϕ[self.m] = ϕ[self.m] + self.h
        self.ϕ = ϕ

        #-----------------------------------------------------
        # If r is given calculate the vector ϕ_r
        #-----------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
        if r is None:
            pass
        else:
            self.r = np.asarray(r)
            self.k = self.r.shape[0] - 1
            ϕ_r = np.zeros(2 * self.k + 1)
            for i in range(- self.k, self.k + 1):
                ϕ_r[self.k - i] = np.sum(np.diag(self.r.reshape(self.k + 1, 1) \
                                                 @ self.r.reshape(1, self.k + 1),
                                                 k=-i
                                                 )
                                         )
            if h_eps is None:
                self.ϕ_r = ϕ_r
            else:
                ϕ_r[self.k] = ϕ_r[self.k] + h_eps
                self.ϕ_r = ϕ_r

        #-----------------------------------------------------
        # If β is given, define the transformed variables
        #-----------------------------------------------------
        if β is None:
            self.β = 1
        else:
            self.β = β
            self.d = self.β**(np.arange(self.m + 1)/2) * self.d
            self.y_m = self.y_m * (self.β**(- np.arange(1, self.m + 1)/2)) \
                                  .reshape(self.m, 1)

    def construct_W_and_Wm(self, N):
        """
        This constructs the matrices W and W_m for a given number of periods N
        """

        m = self.m
        d = self.d

        W = np.zeros((N + 1, N + 1))
        W_m = np.zeros((N + 1, m))

        #---------------------------------------
        # Terminal conditions
        #---------------------------------------

        D_m1 = np.zeros((m + 1, m + 1))
        M = np.zeros((m + 1, m))

        # (1) Constuct the D_{m+1} matrix using the formula

        for j in range(m + 1):
            for k in range(j, m + 1):
                D_m1[j, k] = d[:j + 1] @ d[k - j: k + 1]

        # Make the matrix symmetric
        D_m1 = D_m1 + D_m1.T - np.diag(np.diag(D_m1))
```

(continues on next page)

(continued from previous page)

```
        # (2) Construct the M matrix using the entries of D_m1

        for j in range(m):
            for i in range(j + 1, m + 1):
                M[i, j] = D_m1[i - j - 1, m]

        #-----------------------------------------------
        # Euler equations for t = 0, 1, ..., N-(m+1)
        #-----------------------------------------------
        ϕ = self.ϕ

        W[:(m + 1), :(m + 1)] = D_m1 + self.h * np.eye(m + 1)
        W[:(m + 1), (m + 1):(2 * m + 1)] = M

        for i, row in enumerate(np.arange(m + 1, N + 1 - m)):
            W[row, (i + 1):(2 * m + 2 + i)] = ϕ

        for i in range(1, m + 1):
            W[N - m + i, -(2 * m + 1 - i):] = ϕ[:-i]

        for i in range(m):
            W_m[N - i, :(m - i)] = ϕ[(m + 1 + i):]

        return W, W_m

    def roots_of_characteristic(self):
        """
        This function calculates z_0 and the 2m roots of the characteristic
        equation associated with the Euler equation (1.7)

        Note:
        ------
        numpy.poly1d(roots, True) defines a polynomial using its roots that can
        be evaluated at any point. If x_1, x_2, ... , x_m are the roots then
            p(x) = (x - x_1)(x - x_2)...(x - x_m)
        """
        m = self.m
        ϕ = self.ϕ

        # Calculate the roots of the 2m-polynomial
        roots = np.roots(ϕ)
        # Sort the roots according to their length (in descending order)
        roots_sorted = roots[np.argsort(abs(roots))[::-1]]

        z_0 = ϕ.sum() / np.poly1d(roots, True)(1)
        z_1_to_m = roots_sorted[:m]     # We need only those outside the unit circle

        λ = 1 / z_1_to_m

        return z_1_to_m, z_0, λ

    def coeffs_of_c(self):
        '''
        This function computes the coefficients {c_j, j = 0, 1, ..., m} for
                c(z) = sum_{j = 0}^{m} c_j z^j
```

(continues on next page)

(continued from previous page)

```
        Based on the expression (1.9). The order is
            c_coeffs = [c_0, c_1, ..., c_{m-1}, c_m]
        '''
        z_1_to_m, z_0 = self.roots_of_characteristic()[:2]

        c_0 = (z_0 * np.prod(z_1_to_m).real * (- 1)**self.m)**(.5)
        c_coeffs = np.poly1d(z_1_to_m, True).c * z_0 / c_0

        return c_coeffs[::-1]

    def solution(self):
        """
        This function calculates {λ_j, j=1,...,m} and {A_j, j=1,...,m}
        of the expression (1.15)
        """
        λ = self.roots_of_characteristic()[2]
        c_0 = self.coeffs_of_c()[-1]

        A = np.zeros(self.m, dtype=complex)
        for j in range(self.m):
            denom = 1 - λ/λ[j]
            A[j] = c_0**(-2) / np.prod(denom[np.arange(self.m) != j])

        return λ, A

    def construct_V(self, N):
        '''
        This function constructs the covariance matrix for x^N (see section 6)
        for a given period N
        '''
        V = np.zeros((N, N))
        ϕ_r = self.ϕ_r

        for i in range(N):
            for j in range(N):
                if abs(i-j) <= self.k:
                    V[i, j] = ϕ_r[self.k + abs(i-j)]

        return V

    def simulate_a(self, N):
        """
        Assuming that the u's are normal, this method draws a random path
        for x^N
        """
        V = self.construct_V(N + 1)
        d = spst.multivariate_normal(np.zeros(N + 1), V)

        return d.rvs()

    def predict(self, a_hist, t):
        """
        This function implements the prediction formula discussed in section 6 (1.59)
        It takes a realization for a^N, and the period in which the prediction is
        formed
```

(continues on next page)

(continued from previous page)

```
        Output:  E[abar | a_t, a_{t-1}, ..., a_1, a_0]
        """

        N = np.asarray(a_hist).shape[0] - 1
        a_hist = np.asarray(a_hist).reshape(N + 1, 1)
        V = self.construct_V(N + 1)

        aux_matrix = np.zeros((N + 1, N + 1))
        aux_matrix[:(t + 1), :(t + 1)] = np.eye(t + 1)
        L = la.cholesky(V).T
        Ea_hist = la.inv(L) @ aux_matrix @ L @ a_hist

        return Ea_hist

    def optimal_y(self, a_hist, t=None):
        """
        - if t is NOT given it takes a_hist (list or numpy.array) as a
          deterministic a_t
        - if t is given, it solves the combined control prediction problem
          (section 7)(by default, t == None -> deterministic)

        for a given sequence of a_t (either deterministic or a particular
        realization), it calculates the optimal y_t sequence using the method
        of the lecture

        Note:
        ------
        scipy.linalg.lu normalizes L, U so that L has unit diagonal elements
        To make things consistent with the lecture, we need an auxiliary
        diagonal matrix D which renormalizes L and U
        """

        N = np.asarray(a_hist).shape[0] - 1
        W, W_m = self.construct_W_and_Wm(N)

        L, U = la.lu(W, permute_l=True)
        D = np.diag(1 / np.diag(U))
        U = D @ U
        L = L @ np.diag(1 / np.diag(D))

        J = np.fliplr(np.eye(N + 1))

        if t is None:   # If the problem is deterministic

            a_hist = J @ np.asarray(a_hist).reshape(N + 1, 1)

            #--------------------------------------------
            # Transform the 'a' sequence if β is given
            #--------------------------------------------
            if self.β != 1:
                a_hist =  a_hist * (self.β**(np.arange(N + 1) / 2))[::-1] \
                                 .reshape(N + 1, 1)

            a_bar = a_hist - W_m @ self.y_m             # a_bar from the lecture
            Uy = np.linalg.solve(L, a_bar)              # U @ y_bar = L^{-1}
            y_bar = np.linalg.solve(U, Uy)              # y_bar = U^{-1}L^{-1}
```

(continues on next page)

(continued from previous page)

```
            # Reverse the order of y_bar with the matrix J
            J = np.fliplr(np.eye(N + self.m + 1))
            # y_hist : concatenated y_m and y_bar
            y_hist = J @ np.vstack([y_bar, self.y_m])

            #-----------------------------------------
            # Transform the optimal sequence back if β is given
            #-----------------------------------------
            if self.β != 1:
                y_hist = y_hist * (self.β**(- np.arange(-self.m, N + 1)/2)) \
                                  .reshape(N + 1 + self.m, 1)

            return y_hist, L, U, y_bar

        else:           # If the problem is stochastic and we look at it

            Ea_hist = self.predict(a_hist, t).reshape(N + 1, 1)
            Ea_hist = J @ Ea_hist

            a_bar = Ea_hist - W_m @ self.y_m            # a_bar from the lecture
            Uy = np.linalg.solve(L, a_bar)              # U @ y_bar = L^{-1}
            y_bar = np.linalg.solve(U, Uy)              # y_bar = U^{-1}L^{-1}

            # Reverse the order of y_bar with the matrix J
            J = np.fliplr(np.eye(N + self.m + 1))
            # y_hist : concatenated y_m and y_bar
            y_hist = J @ np.vstack([y_bar, self.y_m])

            return y_hist, L, U, y_bar
```

Let's use this code to tackle two interesting examples.

33.2.2 Example 1

Consider a stochastic process with moving average representation

$$x_t = (1 - 2L)\varepsilon_t$$

where ε_t is a serially uncorrelated random process with mean zero and variance unity.

If we were to use the tools associated with infinite dimensional prediction and filtering to be described below, we would use the Wiener-Kolmogorov formula (33.21) to compute the linear least squares forecasts $\mathbb{E}[x_{t+j} \mid x_t, x_{t-1}, ...]$, for $j = 1, 2$.

But we can do everything we want by instead using our finite dimensional tools and setting $d = r$, generating an instance of LQFilter, then invoking pertinent methods of LQFilter.

```
m = 1
y_m = np.asarray([.0]).reshape(m, 1)
d = np.asarray([1, -2])
r = np.asarray([1, -2])
h = 0.0
example = LQFilter(d, h, y_m, r=d)
```

The Wold representation is computed by `example.coeffs_of_c()`.

Let's check that it "flips roots" as required

```
example.coeffs_of_c()
```

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

```
example.roots_of_characteristic()
```

```
(array([2.]), -2.0, array([0.5]))
```

Now let's form the covariance matrix of a time series vector of length N and put it in V.

Then we'll take a Cholesky decomposition of $V = L^{-1}L^{-1}$ and use it to form the vector of "moving average representations" $x = L^{-1}\varepsilon$ and the vector of "autoregressive representations" $Lx = \varepsilon$.

```
V = example.construct_V(N=5)
print(V)
```

```
[[ 5. -2.  0.  0.  0.]
 [-2.  5. -2.  0.  0.]
 [ 0. -2.  5. -2.  0.]
 [ 0.  0. -2.  5. -2.]
 [ 0.  0.  0. -2.  5.]]
```

Notice how the lower rows of the "moving average representations" are converging to the appropriate infinite history Wold representation to be described below when we study infinite horizon-prediction and filtering

```
Li = np.linalg.cholesky(V)
print(Li)
```

```
[[ 2.23606798  0.          0.          0.          0.        ]
 [-0.89442719  2.04939015  0.          0.          0.        ]
 [ 0.         -0.97590007  2.01186954  0.          0.        ]
 [ 0.          0.         -0.99410024  2.00293902  0.        ]
 [ 0.          0.          0.         -0.99853265  2.000733  ]]
```

Notice how the lower rows of the "autoregressive representations" are converging to the appropriate infinite-history autoregressive representation to be described below when we study infinite horizon-prediction and filtering

```
L = np.linalg.inv(Li)
print(L)
```

```
[[0.4472136  0.         0.         0.         0.        ]
 [0.19518001 0.48795004 0.         0.         0.        ]
 [0.09467621 0.23669053 0.49705012 0.         0.        ]
 [0.04698977 0.11747443 0.2466963  0.49926632 0.        ]
 [0.02345182 0.05862954 0.12312203 0.24917554 0.49981682]]
```

33.2.3 Example 2

Consider a stochastic process X_t with moving average representation

$$X_t = (1 - \sqrt{2}L^2)\varepsilon_t$$

where ε_t is a serially uncorrelated random process with mean zero and variance unity.

Let's find a Wold moving average representation for x_t that will prevail in the infinite-history context to be studied in detail below.

To do this, we'll use the Wiener-Kolomogorov formula (33.21) presented below to compute the linear least squares forecasts $\hat{\mathbb{E}}\left[X_{t+j} \mid X_{t-1}, ...\right]$ for $j = 1, 2, 3$.

We proceed in the same way as in example 1

```
m = 2
y_m = np.asarray([.0, .0]).reshape(m, 1)
d = np.asarray([1, 0, -np.sqrt(2)])
r = np.asarray([1, 0, -np.sqrt(2)])
h = 0.0
example = LQFilter(d, h, y_m, r=d)
example.coeffs_of_c()
```

```
array([ 1.41421356, -0.        , -1.        ])
```

```
example.roots_of_characteristic()
```

```
(array([ 1.18920712, -1.18920712]),
 -1.4142135623731122,
 array([ 0.84089642, -0.84089642]))
```

```
V = example.construct_V(N=8)
print(V)
```

```
[[ 3.          0.         -1.41421356  0.          0.          0.
   0.          0.        ]
 [ 0.          3.          0.         -1.41421356  0.          0.
   0.          0.        ]
 [-1.41421356  0.          3.          0.         -1.41421356  0.
   0.          0.        ]
 [ 0.         -1.41421356  0.          3.          0.         -1.41421356
   0.          0.        ]
 [ 0.          0.         -1.41421356  0.          3.          0.
  -1.41421356  0.        ]
 [ 0.          0.          0.         -1.41421356  0.          3.
   0.         -1.41421356]
 [ 0.          0.          0.          0.         -1.41421356  0.
   3.          0.        ]
 [ 0.          0.          0.          0.          0.         -1.41421356
   0.          3.        ]]
```

```
Li = np.linalg.cholesky(V)
print(Li[-3:, :])
```

```
[[ 0.          0.          0.         -0.9258201   0.          1.46385011
   0.          0.        ]
 [ 0.          0.          0.          0.         -0.96609178  0.
   1.43759058  0.        ]
 [ 0.          0.          0.          0.          0.         -0.96609178
   0.          1.43759058]]
```

```
L = np.linalg.inv(Li)
print(L)
```

```
[[0.57735027 0.         0.         0.         0.         0.
  0.         0.        ]
 [0.         0.57735027 0.         0.         0.         0.
  0.         0.        ]
 [0.3086067  0.         0.65465367 0.         0.         0.
  0.         0.        ]
 [0.         0.3086067  0.         0.65465367 0.         0.
  0.         0.        ]
 [0.19518001 0.         0.41403934 0.         0.68313005 0.
  0.         0.        ]
 [0.         0.19518001 0.         0.41403934 0.         0.68313005
  0.         0.        ]
 [0.13116517 0.         0.27824334 0.         0.45907809 0.
  0.69560834 0.        ]
 [0.         0.13116517 0.         0.27824334 0.         0.45907809
  0.         0.69560834]]
```

33.2.4 Prediction

It immediately follows from the "orthogonality principle" of least squares (see [Athanasios and Pillai, 1991] or [Sargent, 1987] [ch. X]) that

$$\begin{aligned}\hat{\mathbb{E}}[x_t \mid x_{t-m},\, x_{t-m+1}, \dots x_1] &= \sum_{j=m}^{t-1} L^{-1}_{t,t-j}\, \varepsilon_{t-j} \\ &= [L^{-1}_{t,1}\, L^{-1}_{t,2},\, \dots,\, L^{-1}_{t,t-m}\, 0\, 0 \dots 0] L\, x\end{aligned} \tag{33.7}$$

This can be interpreted as a finite-dimensional version of the Wiener-Kolmogorov m-step ahead prediction formula.

We can use (33.7) to represent the linear least squares projection of the vector x conditioned on the first s observations $[x_s, x_{s-1} \dots, x_1]$.

We have

$$\hat{\mathbb{E}}[x \mid x_s, x_{s-1}, \dots, x_1] = L^{-1} \begin{bmatrix} I_s & 0 \\ 0 & 0_{(t-s)} \end{bmatrix} Lx \tag{33.8}$$

This formula will be convenient in representing the solution of control problems under uncertainty.

Equation (33.4) can be recognized as a finite dimensional version of a moving average representation.

Equation (33.2) can be viewed as a finite dimension version of an autoregressive representation.

Notice that even if the x_t process is covariance stationary, so that V is such that V_{ij} depends only on $|i-j|$, the coefficients in the moving average representation are time-dependent, there being a different moving average for each t.

If x_t is a covariance stationary process, the last row of L^{-1} converges to the coefficients in the Wold moving average representation for $\{x_t\}$ as $T \to \infty$.

Further, if x_t is covariance stationary, for fixed k and $j > 0$, $L^{-1}_{T,T-j}$ converges to $L^{-1}_{T-k,T-k-j}$ as $T \to \infty$.

That is, the "bottom" rows of L^{-1} converge to each other and to the Wold moving average coefficients as $T \to \infty$.

This last observation gives one simple and widely-used practical way of forming a finite T approximation to a Wold moving average representation.

First, form the covariance matrix $\mathbb{E} xx' = V$, then obtain the Cholesky decomposition $L^{-1}L^{-1'}$ of V, which can be accomplished quickly on a computer.

The last row of L^{-1} gives the approximate Wold moving average coefficients.

This method can readily be generalized to multivariate systems.

33.3 Combined Finite Dimensional Control and Prediction

Consider the finite-dimensional control problem, maximize

$$\mathbb{E} \sum_{t=0}^{N} \left\{ a_t y_t - \frac{1}{2} h y_t^2 - \frac{1}{2} [d(L) y_t]^2 \right\}, \quad h > 0$$

where $d(L) = d_0 + d_1 L + ... + d_m L^m$, L is the lag operator, $\bar{a} = [a_N, a_{N-1} \dots, a_1, a_0]'$ a random vector with mean zero and $\mathbb{E}\, \bar{a}\bar{a}' = V$.

The variables $y_{-1}, \dots, y_{-m}$ are given.

Maximization is over choices of $y_0, y_1 \dots, y_N$, where y_t is required to be a linear function of $\{y_{t-s-1}, t + m - 1 \geq 0;\ a_{t-s}, t \geq s \geq 0\}$.

We saw in the lecture *Classical Control with Linear Algebra* that the solution of this problem under certainty could be represented in the feedback-feedforward form

$$U\bar{y} = L^{-1}\bar{a} + K \begin{bmatrix} y_{-1} \\ \vdots \\ y_{-m} \end{bmatrix}$$

for some $(N+1) \times m$ matrix K.

Using a version of formula (33.7), we can express $\hat{\mathbb{E}}[\bar{a} \mid a_s,\ a_{s-1}, \dots, a_0]$ as

$$\hat{\mathbb{E}}[\bar{a} \mid a_s,\ a_{s-1}, \dots, a_0] = \tilde{U}^{-1} \begin{bmatrix} 0 & 0 \\ 0 & I_{(s+1)} \end{bmatrix} \tilde{U}\bar{a}$$

where $I_{(s+1)}$ is the $(s+1) \times (s+1)$ identity matrix, and $V = \tilde{U}^{-1}\tilde{U}^{-1'}$, where $\tilde{U}$ is the *upper* triangular Cholesky factor of the covariance matrix V.

(We have reversed the time axis in dating the a's relative to earlier)

The time axis can be reversed in representation (33.8) by replacing L with L^T.

The optimal decision rule to use at time $0 \leq t \leq N$ is then given by the $(N - t + 1)^{\text{th}}$ row of

$$U\bar{y} = L^{-1}\tilde{U}^{-1} \begin{bmatrix} 0 & 0 \\ 0 & I_{(t+1)} \end{bmatrix} \tilde{U}\bar{a} + K \begin{bmatrix} y_{-1} \\ \vdots \\ y_{-m} \end{bmatrix}$$

33.4 Infinite Horizon Prediction and Filtering Problems

It is instructive to compare the finite-horizon formulas based on linear algebra decompositions of finite-dimensional covariance matrices with classic formulas for infinite horizon and infinite history prediction and control problems.

These classic infinite horizon formulas used the mathematics of z-transforms and lag operators.

We'll meet interesting lag operator and z-transform counterparts to our finite horizon matrix formulas.

We pose two related prediction and filtering problems.

We let Y_t be a univariate m^{th} order moving average, covariance stationary stochastic process,

$$Y_t = d(L)u_t \tag{33.9}$$

where $d(L) = \sum_{j=0}^{m} d_j L^j$, and u_t is a serially uncorrelated stationary random process satisfying

$$\begin{aligned} \mathbb{E}u_t &= 0 \\ \mathbb{E}u_t u_s &= \begin{cases} 1 & \text{if } t = s \\ 0 & \text{otherwise} \end{cases} \end{aligned} \tag{33.10}$$

We impose no conditions on the zeros of $d(z)$.

A second covariance stationary process is X_t given by

$$X_t = Y_t + \varepsilon_t \tag{33.11}$$

where ε_t is a serially uncorrelated stationary random process with $\mathbb{E}\varepsilon_t = 0$ and $\mathbb{E}\varepsilon_t\varepsilon_s = 0$ for all distinct t and s.

We also assume that $\mathbb{E}\varepsilon_t u_s = 0$ for all t and s.

The **linear least squares prediction problem** is to find the L_2 random variable $\hat{X}_{t+j}$ among linear combinations of $\{X_t,\ X_{t-1}, ...\}$ that minimizes $\mathbb{E}(\hat{X}_{t+j} - X_{t+j})^2$.

That is, the problem is to find a $\gamma_j(L) = \sum_{k=0}^{\infty} \gamma_{jk} L^k$ such that $\sum_{k=0}^{\infty} |\gamma_{jk}|^2 < \infty$ and $\mathbb{E}[\gamma_j(L)X_t - X_{t+j}]^2$ is minimized.

The **linear least squares filtering problem** is to find a $b(L) = \sum_{j=0}^{\infty} b_j L^j$ such that $\sum_{j=0}^{\infty} |b_j|^2 < \infty$ and $\mathbb{E}[b(L)X_t - Y_t]^2$ is minimized.

Interesting versions of these problems related to the permanent income theory were studied by [Muth, 1960].

33.4.1 Problem Formulation

These problems are solved as follows.

The covariograms of Y and X and their cross covariogram are, respectively,

$$\begin{aligned} C_X(\tau) &= \mathbb{E}X_t X_{t-\tau} \\ C_Y(\tau) &= \mathbb{E}Y_t Y_{t-\tau} \qquad \tau = 0, \pm 1, \pm 2, ... \\ C_{Y,X}(\tau) &= \mathbb{E}Y_t X_{t-\tau} \end{aligned} \tag{33.12}$$

The covariance and cross-covariance generating functions are defined as

$$\begin{aligned} g_X(z) &= \sum_{\tau=-\infty}^{\infty} C_X(\tau) z^\tau \\ g_Y(z) &= \sum_{\tau=-\infty}^{\infty} C_Y(\tau) z^\tau \\ g_{YX}(z) &= \sum_{\tau=-\infty}^{\infty} C_{YX}(\tau) z^\tau \end{aligned} \tag{33.13}$$

The generating functions can be computed by using the following facts.

Let v_{1t} and v_{2t} be two mutually and serially uncorrelated white noises with unit variances.

That is, $\mathbb{E}v_{1t}^2 = \mathbb{E}v_{2t}^2 = 1, \mathbb{E}v_{1t} = \mathbb{E}v_{2t} = 0, \mathbb{E}v_{1t}v_{2s} = 0$ for all t and s, $\mathbb{E}v_{1t}v_{1t-j} = \mathbb{E}v_{2t}v_{2t-j} = 0$ for all $j \neq 0$.

Let x_t and y_t be two random processes given by

$$\begin{aligned} y_t &= A(L)v_{1t} + B(L)v_{2t} \\ x_t &= C(L)v_{1t} + D(L)v_{2t} \end{aligned}$$

Then, as shown for example in [Sargent, 1987] [ch. XI], it is true that

$$\begin{aligned} g_y(z) &= A(z)A(z^{-1}) + B(z)B(z^{-1}) \\ g_x(z) &= C(z)C(z^{-1}) + D(z)D(z^{-1}) \\ g_{yx}(z) &= A(z)C(z^{-1}) + B(z)D(z^{-1}) \end{aligned} \tag{33.14}$$

Applying these formulas to (33.9) – (33.12), we have

$$\begin{aligned} g_Y(z) &= d(z)d(z^{-1}) \\ g_X(z) &= d(z)d(z^{-1}) + h \\ g_{YX}(z) &= d(z)d(z^{-1}) \end{aligned} \tag{33.15}$$

The key step in obtaining solutions to our problems is to factor the covariance generating function $g_X(z)$ of X.

The solutions of our problems are given by formulas due to Wiener and Kolmogorov.

These formulas utilize the Wold moving average representation of the X_t process,

$$X_t = c\left(L\right)\eta_t \tag{33.16}$$

where $c(L) = \sum_{j=0}^{m} c_j L^j$, with

$$c_0\eta_t = X_t - \hat{\mathbb{E}}[X_t | X_{t-1}, X_{t-2}, ...] \tag{33.17}$$

Here $\hat{\mathbb{E}}$ is the linear least squares projection operator.

Equation (33.17) is the condition that $c_0\eta_t$ can be the one-step-ahead error in predicting X_t from its own past values.

Condition (33.17) requires that η_t lie in the closed linear space spanned by $[X_t,\ X_{t-1}, ...]$.

This will be true if and only if the zeros of $c(z)$ do not lie inside the unit circle.

It is an implication of (33.17) that η_t is a serially uncorrelated random process and that normalization can be imposed so that $\mathbb{E}\eta_t^2 = 1$.

Consequently, an implication of (33.16) is that the covariance generating function of X_t can be expressed as

$$g_X(z) = c\left(z\right)c\left(z^{-1}\right) \tag{33.18}$$

It remains to discuss how $c(L)$ is to be computed.

Combining (33.14) and (33.18) gives

$$d(z)\,d(z^{-1}) + h = c\,(z)\,c\,(z^{-1}) \tag{33.19}$$

Therefore, we have already shown constructively how to factor the covariance generating function $g_X(z) = d(z)\,d\,(z^{-1}) + h$.

We now introduce the **annihilation operator**:

$$\left[\sum_{j=-\infty}^{\infty} f_j\,L^j\right]_+ \equiv \sum_{j=0}^{\infty} f_j\,L^j \tag{33.20}$$

In words, $[\quad]_+$ means "ignore negative powers of L".

We have defined the solution of the prediction problem as $\hat{\mathbb{E}}[X_{t+j}|X_t,\,X_{t-1},...] = \gamma_j\,(L)X_t$.

Assuming that the roots of $c(z) = 0$ all lie outside the unit circle, the Wiener-Kolmogorov formula for $\gamma_j(L)$ holds:

$$\gamma_j\,(L) = \left[\frac{c(L)}{L^j}\right]_+ c\,(L)^{-1} \tag{33.21}$$

We have defined the solution of the filtering problem as $\hat{\mathbb{E}}[Y_t \mid X_t, X_{t-1}, ...] = b(L)X_t$.

The Wiener-Kolomogorov formula for $b(L)$ is

$$b(L) = \left[\frac{g_{YX}(L)}{c(L^{-1})}\right]_+ c(L)^{-1}$$

or

$$b(L) = \left[\frac{d(L)d(L^{-1})}{c(L^{-1})}\right]_+ c(L)^{-1} \tag{33.22}$$

Formulas (33.21) and (33.22) are discussed in detail in [Whittle, 1983] and [Sargent, 1987].

The interested reader can there find several examples of the use of these formulas in economics Some classic examples using these formulas are due to [Muth, 1960].

As an example of the usefulness of formula (33.22), we let X_t be a stochastic process with Wold moving average representation

$$X_t = c(L)\eta_t$$

where $\mathbb{E}\eta_t^2 = 1,$ and $c_0\eta_t = X_t - \hat{\mathbb{E}}[X_t|X_{t-1}, ...], c(L) = \sum_{j=0}^{m} c_j L.$

Suppose that at time t, we wish to predict a geometric sum of future X's, namely

$$y_t \equiv \sum_{j=0}^{\infty} \delta^j X_{t+j} = \frac{1}{1-\delta L^{-1}} X_t$$

given knowledge of $X_t, X_{t-1}, ...$

We shall use (33.22) to obtain the answer.

Using the standard formulas (33.14), we have that

$$g_{yx}(z) = (1-\delta z^{-1})c(z)c(z^{-1})$$
$$g_x(z) = c(z)c(z^{-1})$$

Then (33.22) becomes

$$b(L) = \left[\frac{c(L)}{1-\delta L^{-1}}\right]_+ c(L)^{-1} \tag{33.23}$$

In order to evaluate the term in the annihilation operator, we use the following result from [Hansen and Sargent, 1980].

Proposition Let

- $g(z) = \sum_{j=0}^{\infty} g_j z^j$ where $\sum_{j=0}^{\infty} |g_j|^2 < +\infty$.
- $h\left(z^{-1}\right) = \left(1-\delta_1 z^{-1}\right) \ldots \left(1-\delta_n z^{-1}\right)$, where $|\delta_j| < 1$, for $j = 1, \ldots, n$.

Then

$$\left[\frac{g(z)}{h(z^{-1})}\right]_+ = \frac{g(z)}{h(z^{-1})} - \sum_{j=1}^{n} \frac{\delta_j g(\delta_j)}{\prod_{\substack{k=1 \\ k\neq j}}^{n} (\delta_j - \delta_k)} \left(\frac{1}{z-\delta_j}\right) \tag{33.24}$$

and, alternatively,

$$\left[\frac{g(z)}{h(z^{-1})}\right]_+ = \sum_{j=1}^{n} B_j \left(\frac{z g(z) - \delta_j g(\delta_j)}{z-\delta_j}\right) \tag{33.25}$$

where $B_j = 1/\prod_{\substack{k=1 \\ k+j}}^{n} (1-\delta_k/\delta_j)$.

Applying formula (33.25) of the proposition to evaluating (33.23) with $g(z) = c(z)$ and $h(z^{-1}) = 1-\delta z^{-1}$ gives

$$b(L) = \left[\frac{Lc(L) - \delta c(\delta)}{L-\delta}\right] c(L)^{-1}$$

or

$$b(L) = \left[\frac{1-\delta c(\delta) L^{-1} c(L)^{-1}}{1-\delta L^{-1}}\right]$$

Thus, we have

$$\hat{\mathbb{E}}\left[\sum_{j=0}^{\infty} \delta^j X_{t+j} | X_t,\, x_{t-1}, \ldots\right] = \left[\frac{1-\delta c(\delta) L^{-1} c(L)^{-1}}{1-\delta L^{-1}}\right] X_t \tag{33.26}$$

This formula is useful in solving stochastic versions of problem 1 of lecture *Classical Control with Linear Algebra* in which the randomness emerges because $\{a_t\}$ is a stochastic process.

The problem is to maximize

$$\mathbb{E}_0 \lim_{N\to\infty} \sum_{t-0}^{N} \beta^t \left[a_t\, y_t - \frac{1}{2}\, h y_t^2 - \frac{1}{2}\, [d(L) y_t]^2\right] \tag{33.27}$$

where $\mathbb{E}_t$ is mathematical expectation conditioned on information known at t, and where $\{a_t\}$ is a covariance stationary stochastic process with Wold moving average representation

$$a_t = c(L)\, \eta_t$$

where

$$c(L) = \sum_{j=0}^{\tilde{n}} c_j L^j$$

and

$$\eta_t = a_t - \hat{\mathbb{E}}[a_t | a_{t-1}, ...]$$

The problem is to maximize (33.27) with respect to a contingency plan expressing y_t as a function of information known at t, which is assumed to be $(y_{t-1},\ y_{t-2}, \dots, a_t,\ a_{t-1}, ...)$.

The solution of this problem can be achieved in two steps.

First, ignoring the uncertainty, we can solve the problem assuming that $\{a_t\}$ is a known sequence.

The solution is, from above,

$$c(L) y_t = c(\beta L^{-1})^{-1} a_t$$

or

$$(1 - \lambda_1 L) \dots (1 - \lambda_m L) y_t = \sum_{j=1}^{m} A_j \sum_{k=0}^{\infty} (\lambda_j \beta)^k \, a_{t+k} \tag{33.28}$$

Second, the solution of the problem under uncertainty is obtained by replacing the terms on the right-hand side of the above expressions with their linear least squares predictors.

Using (33.26) and (33.28), we have the following solution

$$(1 - \lambda_1 L) \dots (1 - \lambda_m L) y_t = \sum_{j=1}^{m} A_j \left[\frac{1 - \beta \lambda_j \, c(\beta \lambda_j) L^{-1} c(L)^{-1}}{1 - \beta \lambda_j L^{-1}} \right] a_t$$

Blaschke factors

The following is a useful piece of mathematics underlying "root flipping".

Let $\pi(z) = \sum_{j=0}^{m} \pi_j z^j$ and let $z_1, \dots, z_k$ be the zeros of $\pi(z)$ that are inside the unit circle, $k < m$.

Then define

$$\theta(z) = \pi(z) \left(\frac{(z_1 z - 1)}{(z - z_1)} \right) \left(\frac{(z_2 z - 1)}{(z - z_2)} \right) \dots \left(\frac{(z_k z - 1)}{(z - z_k)} \right)$$

The term multiplying $\pi(z)$ is termed a "Blaschke factor".

Then it can be proved directly that

$$\theta(z^{-1}) \theta(z) = \pi(z^{-1}) \pi(z)$$

and that the zeros of $\theta(z)$ are not inside the unit circle.

33.5 Exercises

Exercise 33.5.1

Let $Y_t = (1 - 2L) u_t$ where u_t is a mean zero white noise with $\mathbb{E} u_t^2 = 1$. Let

$$X_t = Y_t + \varepsilon_t$$

where ε_t is a serially uncorrelated white noise with $\mathbb{E} \varepsilon_t^2 = 9$, and $\mathbb{E} \varepsilon_t u_s = 0$ for all t and s.

Find the Wold moving average representation for X_t.

Find a formula for the A_{1j}'s in

$$\mathbb{E}\widehat{X}_{t+1} \mid X_t, X_{t-1}, ... = \sum_{j=0}^{\infty} A_{1j} X_{t-j}$$

Find a formula for the A_{2j}'s in

$$\hat{\mathbb{E}} X_{t+2} \mid X_t, X_{t-1}, ... = \sum_{j=0}^{\infty} A_{2j} X_{t-j}$$

Exercise 33.5.2

Multivariable Prediction: Let Y_t be an $(n \times 1)$ vector stochastic process with moving average representation

$$Y_t = D(L) U_t$$

where $D(L) = \sum_{j=0}^{m} D_j L^J$, D_j an $n \times n$ matrix, U_t an $(n \times 1)$ vector white noise with $\mathbb{E}U_t = 0$ for all t, $\mathbb{E}U_t U_s' = 0$ for all $s \neq t$, and $\mathbb{E}U_t U_t' = I$ for all t.

Let ε_t be an $n \times 1$ vector white noise with mean 0 and contemporaneous covariance matrix H, where H is a positive definite matrix.

Let $X_t = Y_t + \varepsilon_t$.

Define the covariograms as $C_X(\tau) = \mathbb{E}X_t X_{t-\tau}', C_Y(\tau) = \mathbb{E}Y_t Y_{t-\tau}', C_{YX}(\tau) = \mathbb{E}Y_t X_{t-\tau}'$.

Then define the matrix covariance generating function, as in (32.21), only interpret all the objects in (32.21) as matrices.

Show that the covariance generating functions are given by

$$\begin{aligned} g_y(z) &= D(z)D(z^{-1})' \\ g_X(z) &= D(z)D(z^{-1})' + H \\ g_{YX}(z) &= D(z)D(z^{-1})' \end{aligned}$$

A factorization of $g_X(z)$ can be found (see [Rozanov, 1967] or [Whittle, 1983]) of the form

$$D(z)D(z^{-1})' + H = C(z)C(z^{-1})', \quad C(z) = \sum_{j=0}^{m} C_j z^j$$

where the zeros of $|C(z)|$ do not lie inside the unit circle.

A vector Wold moving average representation of X_t is then

$$X_t = C(L)\eta_t$$

where η_t is an $(n \times 1)$ vector white noise that is "fundamental" for X_t.

That is, $X_t - \hat{\mathbb{E}}\left[X_t \mid X_{t-1}, X_{t-2} ...\right] = C_0 \eta_t$.

The optimum predictor of X_{t+j} is

$$\hat{\mathbb{E}}\left[X_{t+j} \mid X_t, X_{t-1}, ...\right] = \left[\frac{C(L)}{L^j}\right]_+ \eta_t$$

If $C(L)$ is invertible, i.e., if the zeros of $\det C(z)$ lie strictly outside the unit circle, then this formula can be written

$$\hat{\mathbb{E}}\left[X_{t+j} \mid X_t, X_{t-1}, ...\right] = \left[\frac{C(L)}{L^J}\right]_+ C(L)^{-1} X_t$$

CHAPTER

THIRTYFOUR

KNOWING THE FORECASTS OF OTHERS

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install --upgrade quantecon
!conda install -y -c plotly plotly plotly-orca
```

34.1 Introduction

Robert E. Lucas, Jr. [Robert E. Lucas, 1975], Kenneth Kasa [Kasa, 2000], and Robert Townsend [Townsend, 1983] showed that putting decision makers into environments in which they want to infer persistent hidden state variables from equilibrium prices and quantities can elongate and amplify impulse responses to aggregate shocks.

This provides a promising way to think about amplification mechanisms in business cycle models.

Townsend [Townsend, 1983] noted that living in such environments makes decision makers want to forecast forecasts of others.

This theme has been pursued for situations in which decision makers' imperfect information forces them to pursue an infinite recursion that involves forming beliefs about the beliefs of others (e.g., [Allen *et al.*, 2002]).

Lucas [Robert E. Lucas, 1975] side stepped having decision makers forecast the forecasts of other decision makers by assuming that they simply **pool their information** before forecasting.

A **pooling equilibrium** like Lucas's plays a prominent role in this lecture.

Because he didn't assume such pooling, [Townsend, 1983] confronted the forecasting the forecasts of others problem.

To formulate the problem recursively required that Townsend define a decision maker's **state** vector.

Townsend concluded that his original model required an intractable infinite dimensional state space.

Therefore, he constructed a more manageable approximating model in which a hidden Markov component of a demand shock is revealed to all firms after a fixed, finite number of periods.

In this lecture, we illustrate again the theme that **finding the state is an art** by showing how to formulate Townsend's original model in terms of a low-dimensional state space.

We show that Townsend's model shares equilibrium prices and quantities with those that prevail in a pooling equilibrium.

That finding emerged from a line of research about Townsend's model that built on [Pearlman *et al.*, 1986] and that culminated in [Pearlman and Sargent, 2005] .

Rather than directly deploying the [Pearlman *et al.*, 1986] machinery here, we shall instead implement a sneaky **guess-and-verify** tactic.

- We first compute a pooling equilibrium and represent it as an instance of a linear state-space system provided by the Python class `quantecon.LinearStateSpace`.

- Leaving the state-transition equation for the pooling equilibrium unaltered, we alter the observation vector for a firm to match what it is in Townsend's original model. So rather than directly observing the signal received by firms in the other industry, a firm sees the equilibrium price of the good produced by the other industry.
- We compute a population linear least squares regression of the noisy signal at time t that firms in the other industry would receive in a pooling equilibrium on time t information that a firm receives in Townsend's original model.
- The R^2 in this regression equals 1.
- That verifies that a firm's information set in Townsend's original model equals its information set in a pooling equilibrium.
- Therefore, equilibrium prices and quantities in Townsend's original model equal those in a pooling equilibrium.

34.1.1 A Sequence of Models

We proceed by describing a sequence of models of two industries that are linked in a single way:

- shocks to the demand curves for their products have a common component.

The models are simplified versions of Townsend's [Townsend, 1983].

Townsend's is a model of a rational expectations equilibrium in which firms want to **forecast forecasts of others**.

In Townsend's model, firms condition their forecasts on observed endogenous variables whose equilibrium laws of motion are determined by their own forecasting functions.

We shall assemble model components progressively in ways that can help us to appreciate the structure of the **pooling equilibrium** that ultimately interests us.

While keeping all other aspects of the model the same, we shall study consequences of alternative assumptions about what decision makers observe.

Technically, this lecture deploys concepts and tools that appear in First Look at Kalman Filter and Rational Expectations Equilibrium.

34.2 The Setting

We cast all variables in terms of deviations from means.

Therefore, we omit constants from inverse demand curves and other functions.

Firms in industry $i = 1, 2$ use a single factor of production, capital k_t^i, to produce output of a single good, y_t^i.

Firms bear quadratic costs of adjusting their capital stocks.

A representative firm in industry i has production function $y_t^i = fk_t^i$, $f > 0$.

The firm acts as a price taker with respect to output price P_t^i, and maximizes

$$E_0^i \sum_{t=0}^{\infty} \beta^t \left\{ P_t^i f k_t^i - .5h(k_{t+1}^i - k_t^i)^2 \right\}, \quad h > 0. \tag{34.1}$$

Demand in industry i is described by the inverse demand curve

$$P_t^i = -bY_t^i + \theta_t + \epsilon_t^i, \quad b > 0, \tag{34.2}$$

where P_t^i is the price of good i at t, $Y_t^i = fK_t^i$ is output in market i, θ_t is a persistent component of a demand shock that is common across the two industries, and ϵ_t^i is an industry specific component of the demand shock that is i.i.d. and whose time t marginal distribution is $\mathcal{N}(0, \sigma_\epsilon^2)$.

We assume that θ_t is governed by

$$\theta_{t+1} = \rho\theta_t + v_t \tag{34.3}$$

where $\{v_t\}$ is an i.i.d. sequence of Gaussian shocks, each with mean zero and variance σ_v^2.

To simplify notation, we'll study a special case by setting $h = f = 1$.

Costs of adjusting their capital stocks impart to firms an incentive to forecast the price of the good that they sell.

Throughout, we use the **rational expectations** equilibrium concept presented in this lecture Rational Expectations Equilibrium.

We let capital letters denote market wide objects and lower case letters denote objects chosen by a representative firm.

In each industry, a competitive equilibrium prevails.

To rationalize the big K, little k connection, we can think of there being a continuum of firms in industry i, with each firm being indexed by $\omega \in [0, 1]$ and $K^i = \int_0^1 k^i(\omega)d\omega$.

In equilibrium, $k_t^i = K_t^i$, but we must distinguish between k_t^i and K_t^i when we pose the firm's optimization problem.

34.3 Tactics

We shall compute equilibrium laws of motion for capital in industry i under a sequence of assumptions about what a representative firm observes.

Successive members of this sequence make a representative firm's information more and more obscure.

We begin with the most information, then gradually withdraw information in a way that approaches and eventually reaches the Townsend-like information structure that we are ultimately interested in.

Thus, we shall compute equilibria under the following alternative information structures:

- **Perfect foresight:** future values of θ_t, ϵ_t^i are observed in industry i.
- **Observed history of stochastic** θ_t: $\{\theta_t, \epsilon_t^i\}$ are realizations from a stochastic process; current and past values of each are observed at time t but future values are not.
- **One noise-ridden observation on** θ_t: values of $\{\theta_t, \epsilon_t^i\}$ separately are never observed. However, at time t, a history w^t of scalar noise-ridden observations on θ_t is observed at time t.
- **Two noise-ridden observations on** θ_t: values of $\{\theta_t, \epsilon_t^i\}$ separately are never observed. However, at time t, a history w^t of *two* noise-ridden observations on θ_t is observed at time t.

Successive computations build one on previous ones.

We proceed by first finding an equilibrium under perfect foresight.

To compute an equilibrium with current and past but not future values of θ_t observed, we use a *certainty equivalence principle* to justify modifying the perfect foresight equilibrium by replacing future values of $\theta_s, \epsilon_s^i, s \geq t$ with mathematical expectations conditioned on θ_t.

This provides the equilibrium when θ_t is observed at t but future θ_{t+j} and ϵ_{t+j}^i are not observed.

To find an equilibrium when a history w^t observations of a **single** noise-ridden θ_t is observed, we again apply a certainty equivalence principle and replace future values of the random variables $\theta_s, \epsilon_s^i, s \geq t$ with their mathematical expectations conditioned on w^t.

To find an equilibrium when a history w^t of **two** noisy signals on θ_t is observed, we replace future values of the random variables $\theta_s, \epsilon_s^i, s \geq t$ with their mathematical expectations conditioned on history w^t.

We call the equilibrium with two noise-ridden observations on θ_t a **pooling equilibrium**.

- It corresponds to an arrangement in which at the beginning of each period firms in industries 1 and 2 somehow get together and share information about current values of their noisy signals on θ.

We want ultimately to compare outcomes in a pooling equilibrium with an equilibrium under the following alternative information structure for a firm in industry i that originally interested Townsend [Townsend, 1983]:

- **Firm** i**'s noise-ridden signal on** θ_t **and the price in industry** $-i$, a firm in industry i observes a history w^t of *one* noise-ridden signal on θ_t and a history of industry $-i$'s price is observed. (Here $-i$ means ``not i".)

With this information structure, a representative firm i sees the price as well as the aggregate endogenous state variable Y_t^i in its own industry.

That allows it to infer the total demand shock $\theta_t + \epsilon_t^i$.

However, at time t, the firm sees only P_t^{-i} and does not see Y_t^{-i}, so that a firm in industry i does not **directly** observe $\theta_t + \epsilon_t^{-i}$.

Nevertheless, it will turn out that equilibrium prices and quantities in this equilibrium equal their counterparts in a pooling equilibrium because firms in industry i are able to infer the noisy signal about the demand shock received by firms in industry $-i$.

We shall verify this assertion by using a guess and verify tactic that involves running a least squares regression and inspecting its R^2.[1]

34.4 Equilibrium Conditions

It is convenient to solve a firm's problem without uncertainty by forming the Lagrangian:

$$J = \sum_{t=0}^{\infty} \beta^t \left\{ P_t^i k_t^i - .5(\mu_t^i)^2 + \phi_t^i \left[k_t^i + \mu_t^i - k_{t+1}^i \right] \right\}$$

where $\{\phi_t^i\}$ is a sequence of Lagrange multipliers on the transition law $k_{t+1}^i = k_t^i + \mu_t^i$.

First order conditions for the nonstochastic problem are

$$\begin{aligned} \phi_t^i &= \beta \phi_{t+1}^i + \beta P_{t+1}^i \\ \mu_t^i &= \phi_t^i. \end{aligned} \tag{34.4}$$

Substituting the demand function (34.2) for P_t^i, imposing the condition $k_t^i = K_t^i$ that makes representative firm be representative, and using definition (34.6) of g_t^i, the Euler equation (34.4) lagged by one period can be expressed as $-bk_t^i + \theta_t + \epsilon_t^i + (k_{t+1}^i - k_t^i) - g_t^i = 0$ or

$$k_{t+1}^i = (b+1)k_t^i - \theta_t - \epsilon_t^i + g_t^i \tag{34.5}$$

where we define g_t^i by

$$g_t^i = \beta^{-1}(k_t^i - k_{t-1}^i) \tag{34.6}$$

We can write Euler equation (34.4) as:

$$g_t^i = P_t^i + \beta g_{t+1}^i \tag{34.7}$$

In addition, we have the law of motion for θ_t, (34.3), and the demand equation (34.2).

[1] [Pearlman and Sargent, 2005] verified this assertion using a different tactic, namely, by constructing analytic formulas for an equilibrium under the incomplete information structure and confirming that they match the pooling equilibrium formulas derived here.

In summary, with perfect foresight, equilibrium conditions for industry i comprise the following system of difference equations:

$$\begin{aligned} k^i_{t+1} &= (1+b)k^i_t - \epsilon^i_t - \theta_t + g^i_t \\ \theta_{t+1} &= \rho\theta_t + v_t \\ g^i_{t+1} &= \beta^{-1}(g^i_t - P^i_t) \\ P^i_t &= -bk^i_t + \epsilon^i_t + \theta_t \end{aligned} \tag{34.8}$$

Without perfect foresight, the same system prevails except that the following equation replaces the third equation of (34.8):

$$g^i_{t+1,t} = \beta^{-1}(g^i_t - P^i_t)$$

where $x_{t+1,t}$ denotes the mathematical expectation of x_{t+1} conditional on information at time t.

34.4.1 Equilibrium under perfect foresight

Our first step is to compute the equilibrium law of motion for k^i_t under perfect foresight.

Let L be the lag operator.[2]

Equations (34.7) and (34.5) imply the second order difference equation in k^i_t:[3]

$$\left[(L^{-1} - (1+b))(1-\beta L^{-1}) + b\right] k^i_t = \beta L^{-1}\epsilon^i_t + \beta L^{-1}\theta_t. \tag{34.9}$$

Factor the polynomial in L on the left side as:

$$-\beta[L^{-2} - (\beta^{-1} + (1+b))L^{-1} + \beta^{-1}] = \tilde{\lambda}^{-1}(L^{-1} - \tilde{\lambda})(1 - \tilde{\lambda}\beta L^{-1})$$

where $|\tilde{\lambda}| < 1$ is the smaller root and λ is the larger root of $(\lambda - 1)(\lambda - 1/\beta) = b\lambda$.

Therefore, (34.9) can be expressed as

$$\tilde{\lambda}^{-1}(L^{-1} - \tilde{\lambda})(1 - \tilde{\lambda}\beta L^{-1})k^i_t = \beta L^{-1}\epsilon^i_t + \beta L^{-1}\theta_t.$$

Solving the stable root backwards and the unstable root forwards gives

$$k^i_{t+1} = \tilde{\lambda}k^i_t + \frac{\tilde{\lambda}\beta}{1 - \tilde{\lambda}\beta L^{-1}}(\epsilon^i_{t+1} + \theta_{t+1}).$$

Recall that we have already set $k^i = K^i$ at the appropriate point in the argument, namely, *after* having derived the first-order necessary conditions for a representative firm in industry i.

Thus, under perfect foresight the equilibrium capital stock in industry i satisfies

$$k^i_{t+1} = \tilde{\lambda}k^i_t + \sum_{j=1}^{\infty}(\tilde{\lambda}\beta)^j(\epsilon^i_{t+j} + \theta_{t+j}). \tag{34.10}$$

Next, we shall investigate consequences of replacing future values of $(\epsilon^i_{t+j} + \theta_{t+j})$ in equation (34.10) with alternative forecasting schemes.

In particular, we shall compute equilibrium laws of motion for capital under alternative assumptions about information available to firms in market i.

[2] See [Sargent, 1987], especially chapters IX and XIV, for principles that guide solving some roots backwards and others forwards.

[3] As noted by [Sargent, 1987], this difference equation is the Euler equation for a planning problem that maximizes the discounted sum of consumer plus producer surplus.

34.5 Equilibrium with θ_t stochastic but observed at t

If future θ's are unknown at t, it is appropriate to replace all random variables on the right side of (34.10) with their conditional expectations based on the information available to decision makers in market i.

For now, we assume that this information set is $I_t^p = \begin{bmatrix}\theta^t & \epsilon^{it}\end{bmatrix}$, where z^t represents the semi-infinite history of variable z_s up to time t.

Later we shall give firms less information.

To obtain an appropriate counterpart to (34.10) under our current assumption about information, we apply a certainty equivalence principle.

In particular, it is appropriate to take (34.10) and replace each term $(\epsilon_{t+j}^i + \theta_{t+j})$ on the right side with $E[(\epsilon_{t+j}^i + \theta_{t+j})|\theta^t]$.

After using (34.3) and the i.i.d. assumption about $\{\epsilon_t^i\}$, this gives

$$k_{t+1}^i = \tilde{\lambda} k_t^i + \frac{\tilde{\lambda}\beta\rho}{1 - \tilde{\lambda}\beta\rho}\theta_t$$

or

$$k_{t+1}^i = \tilde{\lambda} k_t^i + \frac{\rho}{\lambda - \rho}\theta_t \tag{34.11}$$

where $\lambda \equiv (\beta\tilde{\lambda})^{-1}$.

For our purposes, it is convenient to represent the equilibrium $\{k_t^i\}_t$ process recursively as

$$\begin{aligned} k_{t+1}^i &= \tilde{\lambda} k_t^i + \frac{1}{\lambda - \rho}\hat{\theta}_{t+1} \\ \hat{\theta}_{t+1} &= \rho\theta_t \\ \theta_{t+1} &= \rho\theta_t + v_t. \end{aligned} \tag{34.12}$$

34.5.1 Filtering

One noisy signal

We get closer to the original Townsend model that interests us by now assuming that firms in market i do not observe θ_t.

Instead they observe a history w^t of noisy signals at time t.

In particular, assume that

$$\begin{aligned} w_t &= \theta_t + e_t \\ \theta_{t+1} &= \rho\theta_t + v_t \end{aligned} \tag{34.13}$$

where e_t and v_t are mutually independent i.i.d. Gaussian shock processes with means of zero and variances σ_e^2 and σ_v^2, respectively.

Define

$$\hat{\theta}_{t+1} = E(\theta_{t+1}|w^t)$$

where $w^t = [w_t, w_{t-1}, \dots, w_0]$ denotes the history of the w_s process up to and including t.

Associated with the state-space representation (34.13) is the time-invariant *innovations representation*

$$\begin{aligned} \hat{\theta}_{t+1} &= \rho\hat{\theta}_t + \kappa a_t \\ w_t &= \hat{\theta}_t + a_t \end{aligned} \tag{34.14}$$

where $a_t \equiv w_t - E(w_t|w^{t-1})$ is the *innovations* process in w_t and the Kalman gain κ is

$$\kappa = \frac{\rho p}{p + \sigma_e^2} \tag{34.15}$$

and where p satisfies the Riccati equation

$$p = \sigma_v^2 + \frac{p\rho^2\sigma_e^2}{\sigma_e^2 + p}. \tag{34.16}$$

State-reconstruction error

Define the state *reconstruction error* $\tilde{\theta}_t$ by

$$\tilde{\theta}_t = \theta_t - \hat{\theta}_t.$$

Then $p = E\tilde{\theta}_t^2$.

Equations (34.13) and (34.14) imply

$$\tilde{\theta}_{t+1} = (\rho - \kappa)\tilde{\theta}_t + v_t - ke_t. \tag{34.17}$$

Notice that we can express $\hat{\theta}_{t+1}$ as

$$\hat{\theta}_{t+1} = [\rho\theta_t + v_t] + [\kappa e_t - (\rho - \kappa)\tilde{\theta}_t - v_t], \tag{34.18}$$

where the first term in braces equals θ_{t+1} and the second term in braces equals $-\tilde{\theta}_{t+1}$.

We can express (34.11) as

$$k_{t+1}^i = \tilde{\lambda}k_t^i + \frac{1}{\lambda - \rho}E\theta_{t+1}|\theta^t. \tag{34.19}$$

An application of a certainty equivalence principle asserts that when only w^t is observed, a corresponding equilibrium $\{k_t^i\}$ process can be found by replacing the information set θ^t with w^t in (34.19).

Making this substitution and using (34.18) leads to

$$k_{t+1}^i = \tilde{\lambda}k_t^i + \frac{\rho}{\lambda - \rho}\theta_t + \frac{\kappa}{\lambda - \rho}e_t - \frac{\rho - \kappa}{\lambda - \rho}\tilde{\theta}_t. \tag{34.20}$$

Simplifying equation (34.18), we also have

$$\hat{\theta}_{t+1} = \rho\theta_t + \kappa e_t - (\rho - \kappa)\tilde{\theta}_t. \tag{34.21}$$

Equations (34.20), (34.21) describe an equilibrium when w^t is observed.

34.5.2 A new state variable

Relative to (34.11), the equilibrium acquires a **new state variable**, namely, the θ–reconstruction error, $\tilde{\theta}_t$.

For a subsequent argument, by using (34.15), it is convenient to write (34.20) as

$$k_{t+1}^i = \tilde{\lambda}k_t^i + \frac{\rho}{\lambda - \rho}\theta_t + \frac{1}{\lambda - \rho}\frac{p\rho}{p + \sigma_e^2}e_t - \frac{1}{\lambda - \rho}\frac{\rho\sigma_e^2}{p + \sigma_e^2}\tilde{\theta}_t \tag{34.22}$$

In summary, when decision makers in market i observe a semi-infinite history w^t of noisy signals w_t on θ_t at t, we an equilibrium law of motion for k_t^i can be represented as

$$\begin{aligned} k_{t+1}^i &= \tilde{\lambda} k_t^i + \frac{1}{\lambda - \rho}\hat{\theta}_{t+1} \\ \hat{\theta}_{t+1} &= \rho\theta_t + \frac{\rho p}{p + \sigma_e^2} e_t - \frac{\rho\sigma_e^2}{p + \sigma_e^2}\tilde{\theta}_t \\ \tilde{\theta}_{t+1} &= \frac{\rho\sigma_e^2}{p + \sigma_e^2}\tilde{\theta}_t - \frac{p\rho}{p + \sigma_e^2} e_t + v_t \\ \theta_{t+1} &= \rho\theta_t + v_t. \end{aligned} \tag{34.23}$$

34.5.3 Two Noisy Signals

We now construct a **pooling equilibrium** by assuming that at time t a firm in industry i receives a vector w_t of *two* noisy signals on θ_t:

$$\begin{aligned} \theta_{t+1} &= \rho\theta_t + v_t \\ w_t &= \begin{bmatrix} 1 \\ 1 \end{bmatrix} \theta_t + \begin{bmatrix} e_{1t} \\ e_{2t} \end{bmatrix} \end{aligned}$$

To justify that we are constructing is a **pooling equilibrium** we can assume that

$$\begin{bmatrix} e_{1t} \\ e_{2t} \end{bmatrix} = \begin{bmatrix} \epsilon_t^1 \\ \epsilon_t^2 \end{bmatrix}$$

so that a firm in industry i observes the noisy signals on that θ_t presented to firms in both industries i and $-i$.

The pertinent innovations representation now becomes

$$\begin{aligned} \hat{\theta}_{t+1} &= \rho\hat{\theta}_t + \kappa a_t \\ w_t &= \begin{bmatrix} 1 \\ 1 \end{bmatrix} \hat{\theta}_t + a_t \end{aligned} \tag{34.24}$$

where $a_t \equiv w_t - E[w_t | w^{t-1}]$ is a (2×1) vector of innovations in w_t and κ is now a (1×2) vector of Kalman gains.

Formulas for the Kalman filter imply that

$$\kappa = \frac{\rho p}{2p + \sigma_e^2} \begin{bmatrix} 1 & 1 \end{bmatrix} \tag{34.25}$$

where $p = E\tilde{\theta}_t \tilde{\theta}_t^T$ now satisfies the Riccati equation

$$p = \sigma_v^2 + \frac{p\rho^2\sigma_e^2}{2p + \sigma_e^2}. \tag{34.26}$$

Thus, when a representative firm in industry i observes *two* noisy signals on θ_t, we can express the equilibrium law of motion for capital recursively as

$$\begin{aligned} k_{t+1}^i &= \tilde{\lambda} k_t^i + \frac{1}{\lambda - \rho}\hat{\theta}_{t+1} \\ \hat{\theta}_{t+1} &= \rho\theta_t + \frac{\rho p}{2p + \sigma_e^2}(e_{1t} + e_{2t}) - \frac{\rho\sigma_e^2}{2p + \sigma_e^2}\tilde{\theta}_t \\ \tilde{\theta}_{t+1} &= \frac{\rho\sigma_e^2}{2p + \sigma_e^2}\tilde{\theta}_t - \frac{p\rho}{2p + \sigma_e^2}(e_{1t} + e_{2t}) + v_t \\ \theta_{t+1} &= \rho\theta_t + v_t. \end{aligned} \tag{34.27}$$

Below, by using a guess-and-verify tactic, we shall show that outcomes in this **pooling equilibrium** equal those in an equilibrium under the alternative information structure that interested Townsend [Townsend, 1983] but that originally seemed too challenging to compute.[4]

34.6 Guess-and-Verify Tactic

As a preliminary step we shall take our recursive representation (34.23) of an equilibrium in industry i with one noisy signal on θ_t and perform the following steps:

- Compute λ and $\tilde{\lambda}$ by posing a root-finding problem and solving it with `numpy.roots`
- Compute p by forming the appropriate discrete Riccati equation and then solving it using `quantecon.solve_discrete_riccati`
- Add a *measurement equation* for $P_t^i = bk_t^i + \theta_t + e_t$, $\theta_t + e_t$, and e_t to system (34.23).
- Write the resulting system in state-space form and encode it using `quantecon.LinearStateSpace`
- Use methods of the `quantecon.LinearStateSpace` to compute impulse response functions of k_t^i with respect to shocks v_t, e_t.

After analyzing the one-noisy-signal structure in this way, by making appropriate modifications we shall analyze the two-noisy-signal structure.

We proceed to analyze first the one-noisy-signal structure and then the two-noisy-signal structure.

34.7 Equilibrium with One Noisy Signal on θ_t

34.7.1 Step 1: Solve for $\tilde{\lambda}$ and λ

1. Cast $(\lambda - 1)\left(\lambda - \frac{1}{\beta}\right) = b\lambda$ as $p(\lambda) = 0$ where p is a polynomial function of λ.
2. Use `numpy.roots` to solve for the roots of p
3. Verify $\lambda \approx \frac{1}{\beta\tilde{\lambda}}$

Note that $p(\lambda) = \lambda^2 - \left(1 + b + \frac{1}{\beta}\right)\lambda + \frac{1}{\beta}$.

34.7.2 Step 2: Solve for p

1. Cast $p = \sigma_v^2 + \frac{p\rho^2\sigma_e^2}{2p+\sigma_e^2}$ as a discrete matrix Riccati equation.
2. Use `quantecon.solve_discrete_riccati` to solve for p
3. Verify $p \approx \sigma_v^2 + \frac{p\rho^2\sigma_e^2}{2p+\sigma_e^2}$

[4] [Pearlman and Sargent, 2005] verify the same claim by applying machinery of [Pearlman *et al.*, 1986].

Note that:

$$
\begin{aligned}
A &= \begin{bmatrix} \rho \end{bmatrix} \\
B &= \begin{bmatrix} \sqrt{2} \end{bmatrix} \\
R &= \begin{bmatrix} \sigma_e^2 \end{bmatrix} \\
Q &= \begin{bmatrix} \sigma_v^2 \end{bmatrix} \\
N &= \begin{bmatrix} 0 \end{bmatrix}
\end{aligned}
$$

34.7.3 Step 3: Represent the system using `quantecon.LinearStateSpace`

We use the following representation for constructing the `quantecon.LinearStateSpace` instance.

$$
\underbrace{\begin{bmatrix} e_{t+1} \\ k^i_{t+1} \\ \tilde{\theta}_{t+1} \\ P_{t+1} \\ \theta_{t+1} \\ v_{t+1} \end{bmatrix}}_{x_{t+1}} = \underbrace{\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{\kappa}{\lambda-\rho} & \tilde{\lambda} & \frac{-1}{\lambda-\rho}\frac{\kappa\sigma_e^2}{p} & 0 & \frac{\rho}{\lambda-\rho} & 0 \\ -\kappa & 0 & \frac{\kappa\sigma_e^2}{p} & 0 & 0 & 1 \\ \frac{b\kappa}{\lambda-\rho} & b\tilde{\lambda} & \frac{-b}{\lambda-\rho}\frac{\kappa\sigma_e^2}{p} & 0 & \frac{b\rho}{\lambda-\rho}+\rho & 1 \\ 0 & 0 & 0 & 0 & \rho & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} e_t \\ k^i_t \\ \tilde{\theta}_t \\ P_t \\ \theta_t \\ v_t \end{bmatrix}}_{x_t} + \underbrace{\begin{bmatrix} \sigma_e & 0 \\ 0 & 0 \\ 0 & 0 \\ \sigma_e & 0 \\ 0 & 0 \\ 0 & \sigma_v \end{bmatrix}}_{C} \begin{bmatrix} z_{1,t+1} \\ z_{2,t+1} \end{bmatrix}
$$

$$
\underbrace{\begin{bmatrix} P_t \\ e_t + \theta_t \\ e_t \end{bmatrix}}_{y_t} = \underbrace{\begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}}_{G} \underbrace{\begin{bmatrix} e_t \\ k^i_t \\ \tilde{\theta}_t \\ P_t \\ \theta_t \\ v_t \end{bmatrix}}_{x_t} + \underbrace{\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}}_{H} w_{t+1}
$$

$$
\begin{bmatrix} z_{1,t+1} \\ z_{2,t+1} \\ w_{t+1} \end{bmatrix} \sim \mathcal{N}(0, I)
$$

$$
\kappa = \frac{\rho p}{p + \sigma_e^2}
$$

This representation includes extraneous variables such as P_t in the state vector.

We formulate things in this way because it allows us easily to compute covariances of these variables with other components of the state vector (step 5 above) by using the `stationary_distributions` method of the `LinearStateSpace` class.

```
import numpy as np
import quantecon as qe
import plotly.graph_objects as go
import plotly.offline as pyo
from statsmodels.regression.linear_model import OLS
from IPython.display import display, Latex, Image

pyo.init_notebook_mode(connected=True)
```

```
β = 0.9  # Discount factor
ρ = 0.8  # Persistence parameter for the hidden state
```

(continues on next page)

(continued from previous page)

```
b = 1.5  # Demand curve parameter
σ_v = 0.5  # Standard deviation of shock to θ_t
σ_e = 0.6  # Standard deviation of shocks to w_t
```

```
# Compute λ
poly = np.array([1, -(1 + β + b) / β, 1 / β])
roots_poly = np.roots(poly)
λ_tilde = roots_poly.min()
λ = roots_poly.max()
```

```
# Verify that λ = (βλ_tilde) ^ (-1)
tol = 1e-12
np.max(np.abs(λ - 1 / (β * λ_tilde))) < tol
```

```
True
```

```
A_ricc = np.array([[ρ]])
B_ricc = np.array([[1.]])
R_ricc = np.array([[σ_e ** 2]])
Q_ricc = np.array([[σ_v ** 2]])
N_ricc = np.zeros((1, 1))
p = qe.solve_discrete_riccati(A_ricc, B_ricc, Q_ricc, R_ricc, N_ricc).item()

p_one = p  # Save for comparison later
```

```
# Verify that p = σ_v ^ 2 + p * ρ ^ 2 - (ρ * p) ^ 2 / (p + σ_e ** 2)
tol = 1e-12
np.abs(p - (σ_v ** 2 + p * ρ ** 2 - (ρ * p) ** 2 / (p + σ_e ** 2))) < tol
```

```
True
```

```
κ = ρ * p / (p + σ_e ** 2)
κ_prod = κ * σ_e ** 2 / p

κ_one = κ  # Save for comparison later

A_lss = np.array([[0., 0., 0., 0., 0., 0.],
                  [κ / (λ - ρ), λ_tilde, -κ_prod / (λ - ρ), 0., ρ / (λ - ρ), 0.],
                  [-κ, 0., κ_prod, 0., 0., 1.],
                  [b * κ / (λ - ρ) , b * λ_tilde, -b * κ_prod / (λ - ρ), 0., b * ρ /␣
 ↪(λ - ρ) + ρ, 1.],
                  [0., 0., 0., 0., ρ, 1.],
                  [0., 0., 0., 0., 0., 0.]])

C_lss = np.array([[σ_e, 0.],
                  [0.,  0.],
                  [0.,  0.],
                  [σ_e, 0.],
                  [0., 0.],
                  [0., σ_v]])
```

(continues on next page)

(continued from previous page)

```
G_lss = np.array([[0., 0., 0., 1., 0., 0.],
                  [1., 0., 0., 0., 1., 0.],
                  [1., 0., 0., 0., 0., 0.]])
```

```
mu_0 = np.array([0., 0., 0., 0., 0., 0.])

lss = qe.LinearStateSpace(A_lss, C_lss, G_lss, mu_0=mu_0)
```

```
ts_length = 100_000
x, y = lss.simulate(ts_length, random_state=1)
```

```
# Verify that two ways of computing P_t match
np.max(np.abs(np.array([[1., b, 0., 0., 1., 0.]]) @ x - x[3])) < 1e-12
```

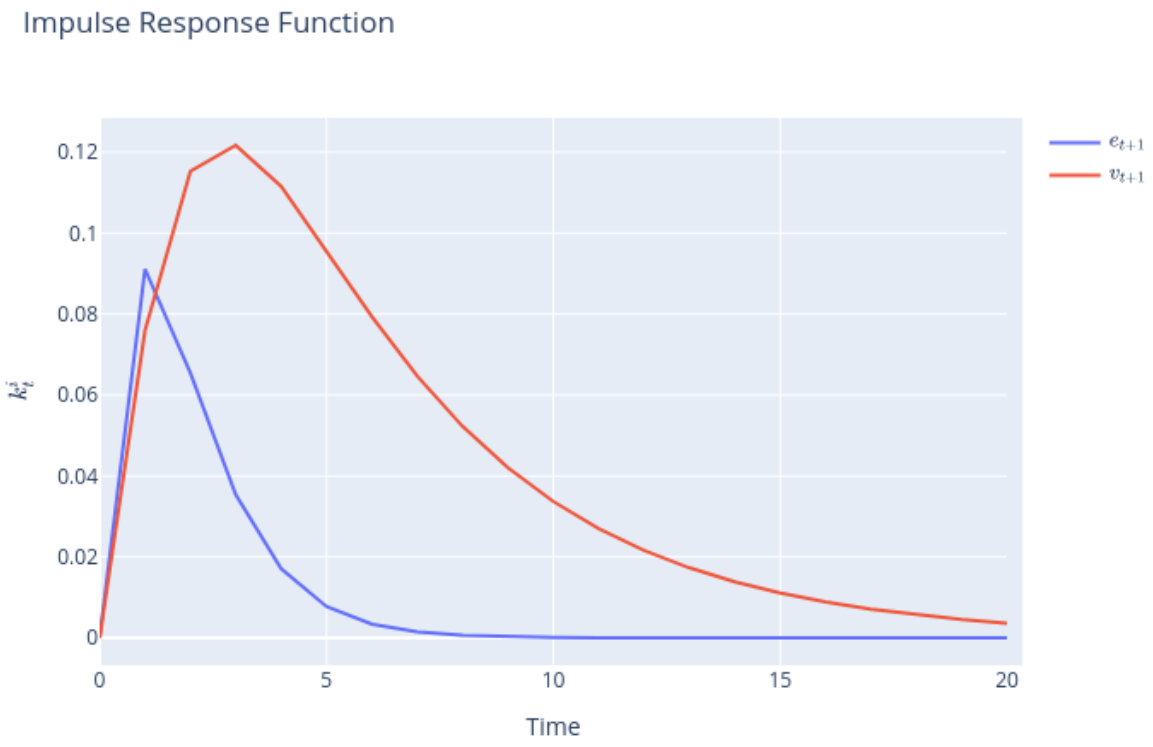
```
True
```

34.7.4 Step 4: Compute impulse response functions

To compute impulse response functions of k_t^i, we use the `impulse_response` method of the `quantecon.LinearStateSpace` class and plot outcomes.

```
xcoef, ycoef = lss.impulse_response(j=21)
data = np.array([xcoef])[0, :, 1, :]

fig = go.Figure(data=go.Scatter(y=data[:-1, 0], name=r'$e_{t+1}$'))
fig.add_trace(go.Scatter(y=data[1:, 1], name=r'$v_{t+1}$'))
fig.update_layout(title=r'Impulse Response Function',
                  xaxis_title='Time',
                  yaxis_title=r'$k^{i}_{t}$')
fig1 = fig
# Export to PNG file
Image(fig1.to_image(format="png"))
# fig1.show() will provide interactive plot when running
# notebook locally
```



34.7.5 Step 5: Compute stationary covariance matrices and population regressions

We compute stationary covariance matrices by calling the `stationary_distributions` method of the `quantecon.LinearStateSpace` class.

By appropriately decomposing the covariance matrix of the state vector, we obtain ingredients of pertinent population regression coefficients.

Define

$$\Sigma_x = \begin{bmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{bmatrix}$$

where Σ_{11} is the covariance matrix of dependent variables and Σ_{22} is the covariance matrix of independent variables.

Regression coefficients are $\beta = \Sigma_{21}\Sigma_{22}^{-1}$.

To verify an instance of a law of large numbers computation, we construct a long simulation of the state vector and for the resulting sample compute the ordinary least-squares estimator of β that we shall compare with corresponding population regression coefficients.

```
_, _, Σ_x, Σ_y, Σ_yx = lss.stationary_distributions()

Σ_11 = Σ_x[0, 0]
Σ_12 = Σ_x[0, 1:4]
Σ_21 = Σ_x[1:4, 0]
Σ_22 = Σ_x[1:4, 1:4]
```

(continues on next page)

(continued from previous page)

```
reg_coeffs = Σ_12 @ np.linalg.inv(Σ_22)

print('Regression coefficients (e_t on k_t, P_t, \\tilde{\\theta_t})')
print('------------------------------')
print(r'k_t:', reg_coeffs[0])
print(r'\tilde{\theta_t}:', reg_coeffs[1])
print(r'P_t:', reg_coeffs[2])
```

```
Regression coefficients (e_t on k_t, P_t, \tilde{\theta_t})
------------------------------
k_t: -3.275556845219769
\tilde{\theta_t}: -0.9649461170475457
P_t: 0.9649461170475457
```

```
# Compute R squared
R_squared = reg_coeffs @ Σ_x[1:4, 1:4] @ reg_coeffs  / Σ_x[0, 0]
R_squared
```

```
0.9649461170475461
```

```
# Verify that the computed coefficients are close to least squares estimates
model = OLS(x[0], x[1:4].T)
reg_res = model.fit()
np.max(np.abs(reg_coeffs - reg_res.params)) < 1e-2
```

```
True
```

```
# Verify that R_squared matches least squares estimate
np.abs(reg_res.rsquared - R_squared) < 1e-2
```

```
True
```

```
# Verify that θ_t + e_t can be recovered
model = OLS(y[1], x[1:4].T)
reg_res = model.fit()
np.abs(reg_res.rsquared - 1.) < 1e-6
```

```
True
```

34.8 Equilibrium with Two Noisy Signals on θ_t

Steps 1, 4, and 5 are identical to those for the one-noisy-signal structure.

Step 2 requires a straightforward modification.

For step 3, we construct the following state-space representation so that we can get our hands on all of the random processes that we require in order to compute a regression of the noisy signal about θ from the other industry that a firm receives directly in a pooling equilibrium against information that a firm would receive in Townsend's original model.

For this purpose, we include equilibrium goods prices from both industries in the state vector:

$$\underbrace{\begin{bmatrix} e_{1,t+1} \\ e_{2,t+1} \\ k^i_{t+1} \\ \tilde{\theta}_{t+1} \\ P^1_{t+1} \\ P^2_{t+1} \\ \theta_{t+1} \\ v_{t+1} \end{bmatrix}}_{x_{t+1}} = \underbrace{\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{\kappa}{\lambda-\rho} & \frac{\kappa}{\lambda-\rho} & \tilde{\lambda} & \frac{-1}{\lambda-\rho}\frac{\kappa\sigma_e^2}{p} & 0 & 0 & \frac{\rho}{\lambda-\rho} & 0 \\ -\kappa & -\kappa & 0 & \frac{\kappa\sigma_e^2}{p} & 0 & 0 & 0 & 1 \\ \frac{b\kappa}{\lambda-\rho} & \frac{b\kappa}{\lambda-\rho} & b\tilde{\lambda} & \frac{-b}{\lambda-\rho}\frac{\kappa\sigma_e^2}{p} & 0 & 0 & \frac{b\rho}{\lambda-\rho}+\rho & 1 \\ \frac{b\kappa}{\lambda-\rho} & \frac{b\kappa}{\lambda-\rho} & b\tilde{\lambda} & \frac{-b}{\lambda-\rho}\frac{\kappa\sigma_e^2}{p} & 0 & 0 & \frac{b\rho}{\lambda-\rho}+\rho & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & \rho & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} e_{1,t} \\ e_{2,t} \\ k^i_t \\ \tilde{\theta}_t \\ P^1_t \\ P^2_t \\ \theta_t \\ v_t \end{bmatrix}}_{x_t} + \underbrace{\begin{bmatrix} \sigma_e & 0 & 0 \\ 0 & \sigma_e & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ \sigma_e & 0 & 0 \\ 0 & \sigma_e & 0 \\ 0 & 0 & 0 \\ 0 & 0 & \sigma_v \end{bmatrix}}_{C} \begin{bmatrix} z_{1,t+1} \\ z_{2,t+1} \\ z_{3,t+1} \end{bmatrix}$$

$$\underbrace{\begin{bmatrix} P^1_t \\ P^2_t \\ e_{1,t}+\theta_t \\ e_{2,t}+\theta_t \\ e_{1,t} \\ e_{2,t} \end{bmatrix}}_{y_t} = \underbrace{\begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}}_{G} \underbrace{\begin{bmatrix} e_{1,t} \\ e_{2,t} \\ k^i_t \\ \tilde{\theta}_t \\ P^1_t \\ P^2_t \\ \theta_t \\ v_t \end{bmatrix}}_{x_t} + \underbrace{\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}}_{H} w_{t+1}$$

$$\begin{bmatrix} z_{1,t+1} \\ z_{2,t+1} \\ z_{3,t+1} \\ w_{t+1} \end{bmatrix} \sim \mathcal{N}(0, I)$$

$$\kappa = \frac{\rho p}{2p + \sigma_e^2}$$

```
A_ricc = np.array([[ρ]])
B_ricc = np.array([[np.sqrt(2)]])
R_ricc = np.array([[σ_e ** 2]])
Q_ricc = np.array([[σ_v ** 2]])
N_ricc = np.zeros((1, 1))
p = qe.solve_discrete_riccati(A_ricc, B_ricc, Q_ricc, R_ricc, N_ricc).item()

p_two = p  # Save for comparison later
```

```
# Verify that p = σ_v^2 + (pρ^2σ_e^2) / (2p + σ_e^2)
tol = 1e-12
np.abs(p - (σ_v ** 2 + p * ρ ** 2 * σ_e ** 2 / (2 * p + σ_e ** 2))) < tol
```

```
True
```

```
κ = ρ * p / (2 * p + σ_e ** 2)
κ_prod = κ * σ_e ** 2 / p

κ_two = κ  # Save for comparison later

A_lss = np.array([[0., 0., 0., 0., 0., 0., 0., 0.],
                  [0., 0., 0., 0., 0., 0., 0., 0.],
                  [κ / (λ - ρ), κ / (λ - ρ), λ_tilde, -κ_prod / (λ - ρ), 0., 0., ρ /␣
 ↪(λ - ρ), 0.],
                  [-κ, -κ, 0., κ_prod, 0., 0., 0., 1.],
                  [b * κ / (λ - ρ), b * κ / (λ - ρ), b * λ_tilde, -b * κ_prod / (λ -␣
 ↪ρ), 0., 0., b * ρ / (λ - ρ) + ρ, 1.],
                  [b * κ / (λ - ρ), b * κ / (λ - ρ), b * λ_tilde, -b * κ_prod / (λ -␣
 ↪ρ), 0., 0., b * ρ / (λ - ρ) + ρ, 1.],
                  [0., 0., 0., 0., 0., 0., ρ, 1.],
                  [0., 0., 0., 0., 0., 0., 0., 0.]])

C_lss = np.array([[σ_e, 0., 0.],
                  [0., σ_e, 0.],
                  [0., 0.,  0.],
                  [0., 0.,  0.],
                  [σ_e, 0., 0.],
                  [0., σ_e, 0.],
                  [0., 0., 0.],
                  [0., 0., σ_v]])

G_lss = np.array([[0., 0., 0., 0., 1., 0., 0., 0.],
                  [0., 0, 0, 0., 0., 1., 0., 0.],
                  [1., 0., 0., 0., 0., 0., 1., 0.],
                  [0., 1., 0., 0., 0., 0., 1., 0.],
                  [1., 0., 0., 0., 0., 0., 0., 0.],
                  [0., 1., 0., 0., 0., 0., 0., 0.]])
```

```
mu_0 = np.array([0., 0., 0., 0., 0., 0., 0., 0.])

lss = qe.LinearStateSpace(A_lss, C_lss, G_lss, mu_0=mu_0)
```

```
ts_length = 100_000
x, y = lss.simulate(ts_length, random_state=1)
```

```
xcoef, ycoef = lss.impulse_response(j=20)
```

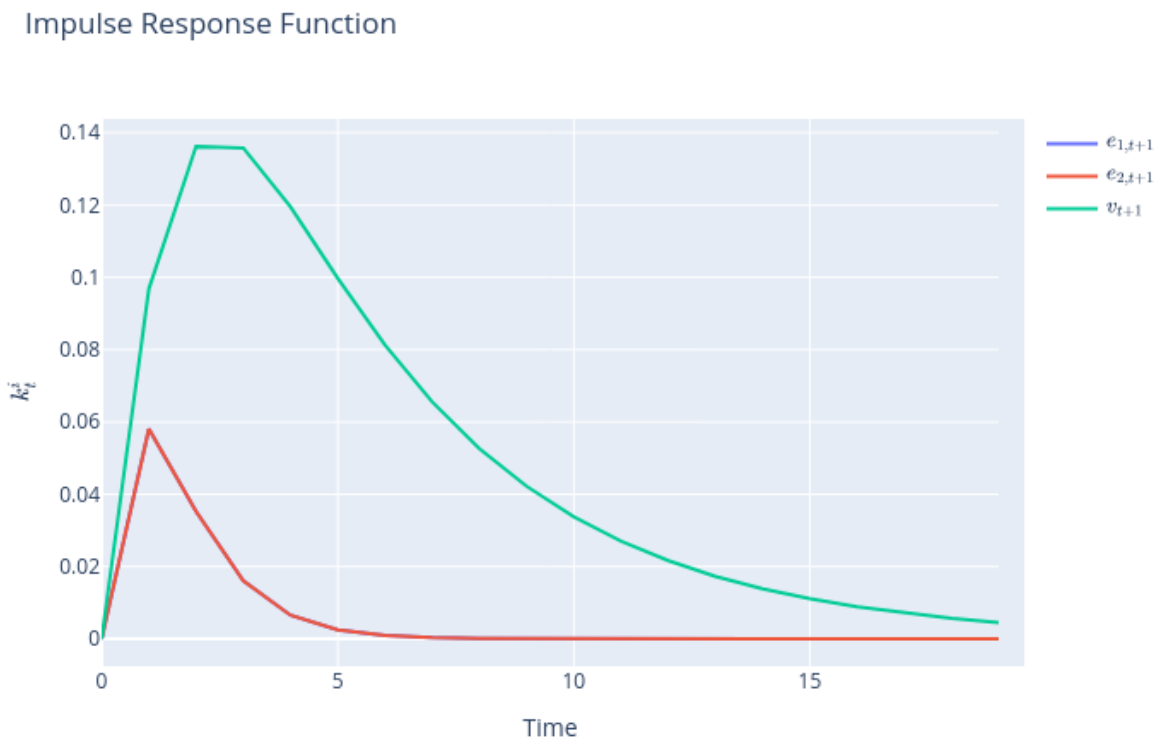
```
data = np.array([xcoef])[0, :, 2, :]

fig = go.Figure(data=go.Scatter(y=data[:-1, 0], name=r'$e_{1,t+1}$'))
fig.add_trace(go.Scatter(y=data[:-1, 1], name=r'$e_{2,t+1}$'))
fig.add_trace(go.Scatter(y=data[1:, 2], name=r'$v_{t+1}$'))
fig.update_layout(title=r'Impulse Response Function',
                  xaxis_title='Time',
                  yaxis_title=r'$k^{i}_{t}$')
fig2=fig
# Export to PNG file
Image(fig2.to_image(format="png"))
# fig2.show() will provide interactive plot when running
```

(continues on next page)

(continued from previous page)

```
# notebook locally
```




```
_, _, Σ_x, Σ_y, Σ_yx = lss.stationary_distributions()

Σ_11 = Σ_x[1, 1]
Σ_12 = Σ_x[1, 2:5]
Σ_21 = Σ_x[2:5, 1]
Σ_22 = Σ_x[2:5, 2:5]

reg_coeffs = Σ_12 @ np.linalg.inv(Σ_22)

print('Regression coefficients (e_{2,t} on k_t, P^{1}_t, \\tilde{\\theta_t})')
print('------------------------------')
print(r'k_t:', reg_coeffs[0])
print(r'\tilde{\theta_t}:', reg_coeffs[1])
print(r'P_t:', reg_coeffs[2])
```

```
Regression coefficients (e_{2,t} on k_t, P^{1}_t, \tilde{\theta_t})
------------------------------
k_t: 0.0
\tilde{\theta_t}: 0.0
P_t: 0.0
```

```
# Compute R squared
```

(continues on next page)

(continued from previous page)

```
R_squared = reg_coeffs @ Σ_x[2:5, 2:5] @ reg_coeffs  / Σ_x[1, 1]
R_squared
```

```
0.0
```

```
# Verify that the computed coefficients are close to least squares estimates
model = OLS(x[1], x[2:5].T)
reg_res = model.fit()
np.max(np.abs(reg_coeffs - reg_res.params)) < 1e-2
```

```
True
```

```
# Verify that R_squared matches least squares estimate
np.abs(reg_res.rsquared - R_squared) < 1e-2
```

```
True
```

```
_, _, Σ_x, Σ_y, Σ_yx = lss.stationary_distributions()

Σ_11 = Σ_x[1, 1]
Σ_12 = Σ_x[1, 2:6]
Σ_21 = Σ_x[2:6, 1]
Σ_22 = Σ_x[2:6, 2:6]

reg_coeffs = Σ_12 @ np.linalg.inv(Σ_22)

print('Regression coefficients (e_{2,t} on k_t, P^{1}_t, P^{2}_t, \\tilde{\\theta_t})
↪')
print('------------------------------')
print(r'k_t:', reg_coeffs[0])
print(r'\tilde{\theta_t}:', reg_coeffs[1])
print(r'P^{1}_t:', reg_coeffs[2])
print(r'P^{2}_t:', reg_coeffs[3])
```

```
Regression coefficients (e_{2,t} on k_t, P^{1}_t, P^{2}_t, \tilde{\theta_t})
------------------------------
k_t: -3.1373589171035627
\tilde{\theta_t}: -0.9242343967443672
P^{1}_t: -0.037882801627816154
P^{2}_t: 0.9621171983721835
```

```
# Compute R squared
R_squared = reg_coeffs @ Σ_x[2:6, 2:6] @ reg_coeffs  / Σ_x[1, 1]
R_squared
```

```
0.9621171983721837
```

34.9 Key Step

Now we come to the key step for verifying that equilibrium outcomes for prices and quantities are identical in the pooling equilibrium original model that led Townsend to deduce an infinite-dimensional state space.

We accomplish this by computing a population linear least squares regression of the noisy signal that firms in the other industry receive in a pooling equilibrium on time t information that a firm would receive in Townsend's original model.

Let's compute the regression and stare at the R^2:

```
# Verify that θ_t + e^{2}_t can be recovered

# θ_t + e^{2}_t on k^{i}_t, P^{1}_t, P^{2}_t, \\tilde{\\theta_t}


model = OLS(y[1], x[2:6].T)
reg_res = model.fit()
np.abs(reg_res.rsquared - 1.) < 1e-6
```

```
True
```

```
reg_res.rsquared
```

```
1.0
```

The R^2 in this regression equals 1.

That verifies that a firm's information set in Townsend's original model equals its information set in a pooling equilibrium.

Therefore, equilibrium prices and quantities in Townsend's original model equal those in a pooling equilibrium.

34.10 An observed common shock benchmark

For purposes of comparison, it is useful to construct a model in which demand disturbance in both industries still both share have a common persistent component θ_t, but in which the persistent component θ is observed each period.

In this case, firms share the same information immediately and have no need to deploy signal-extraction techniques.

Thus, consider a version of our model in which histories of both ϵ_t^i and θ_t are observed by a representative firm.

In this case, the firm's optimal decision rule is described by

$$k_{t+1}^i = \tilde{\lambda} k_t^i + \frac{1}{\lambda - \rho} \hat{\theta}_{t+1}$$

where $\hat{\theta}_{t+1} = E_t \theta_{t+1}$ is given by

$$\hat{\theta}_{t+1} = \rho \theta_t$$

Thus, the firm's decision rule can be expressed

$$k_{t+1}^i = \tilde{\lambda} k_t^i + \frac{\rho}{\lambda - \rho}\theta_t$$

Consequently, when a history $\theta_s, s \leq t$ is observed without noise, the following state space system prevails:

$$\begin{bmatrix} \theta_{t+1} \\ k_{t+1}^i \end{bmatrix} = \begin{bmatrix} \rho & 0 \\ \frac{\rho}{\lambda-\rho} & \tilde{\lambda} \end{bmatrix} \begin{bmatrix} \theta_t \\ k_t^i \end{bmatrix} + \begin{bmatrix} \sigma_v \\ 0 \end{bmatrix} z_{1,t+1}$$
$$\begin{bmatrix} \theta_t \\ k_t^i \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \theta_t \\ k_t^i \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} z_{1,t+1}$$

where $z_{t,t+1}$ is a scalar iid standardized Gaussian process.

As usual, the system can be written as

$$x_{t+1} = Ax_t + Cz_{t+1}$$
$$y_t = Gx_t + Hw_{t+1}$$

In order once again to use the quantecon class `quantecon.LinearStateSpace`, let's form pertinent state-space matrices

```
Ao_lss = np.array([[ρ, 0.],
                   [ρ / (λ - ρ), λ_tilde]])

Co_lss = np.array([[σ_v], [0.]])

Go_lss = np.identity(2)
```

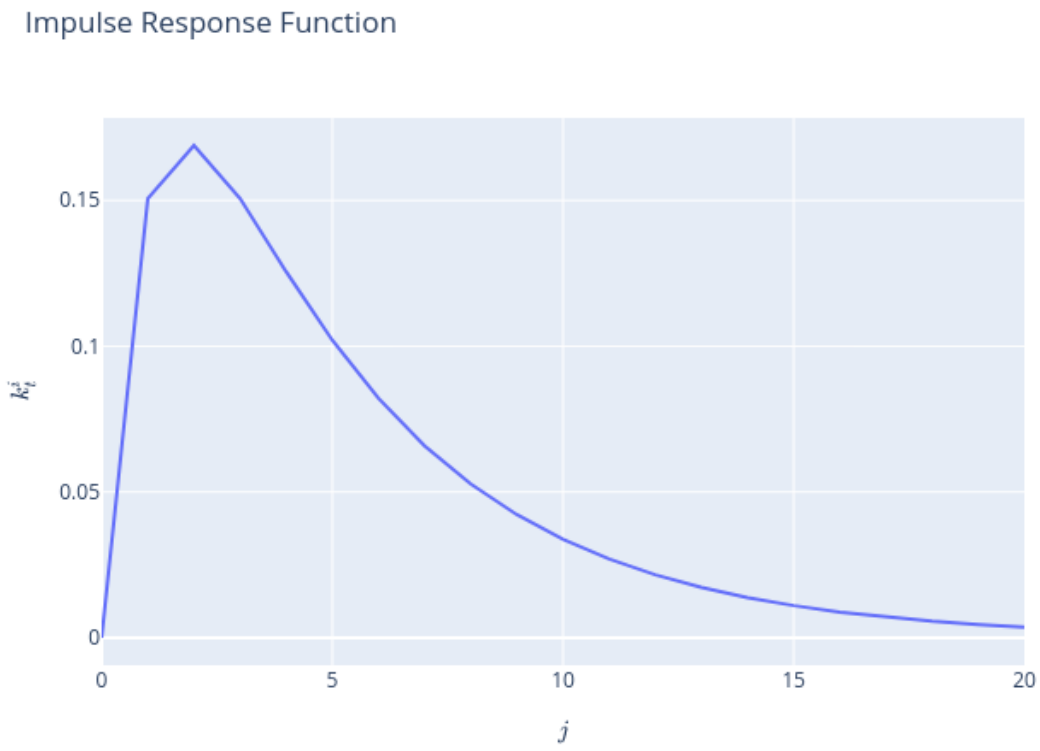
```
muo_0 = np.array([0., 0.])

lsso = qe.LinearStateSpace(Ao_lss, Co_lss, Go_lss, mu_0=muo_0)
```

Now let's form and plot an impulse response function of k_t^i to shocks v_t to θ_{t+1}

```
xcoef, ycoef = lsso.impulse_response(j=21)
data = np.array([ycoef])[0, :, 1, :]

fig = go.Figure(data=go.Scatter(y=data[:-1, 0], name=r'$z_{t+1}$'))
fig.update_layout(title=r'Impulse Response Function',
                  xaxis_title= r'lag $j$',
                  yaxis_title=r'$k^{i}_{t}$')
fig3 = fig
# Export to PNG file
Image(fig3.to_image(format="png"))
# fig1.show() will provide interactive plot when running
# notebook locally
```

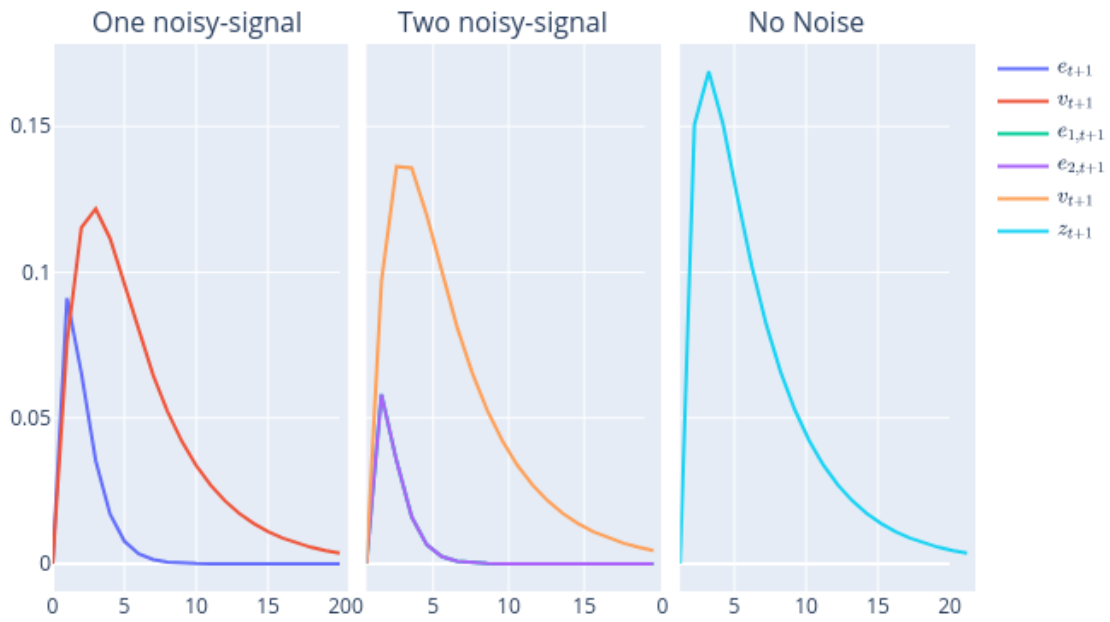


34.11 Comparison of All Signal Structures

It is enlightening side by side to plot impulse response functions for capital for the two noisy-signal information structures and the noiseless signal on θ that we have just presented.

Please remember that the two-signal structure corresponds to the **pooling equilibrium** and also **Townsend's original model**.

```
fig_comb = go.Figure(data=[
                        *fig1.data,
                        *fig2.update_traces(xaxis='x2', yaxis='y2').data,
                        *fig3.update_traces(xaxis='x3', yaxis='y3').data
                    ]).set_subplots(1, 3,
                            subplot_titles=("One noisy-signal",
                                            "Two noisy-signal",
                                            "No Noise"),
                            horizontal_spacing=0.02,
                            shared_yaxes=True)
# Export to PNG file
Image(fig_comb.to_image(format="png"))
# fig_comb.show() # will provide interactive plot when running
# notebook locally
```



The three panels in the graph above show that

- responses of k_t^i to shocks v_t to the hidden Markov demand state θ_t process are **largest** in the no-noisy-signal structure in which the firm observes θ_t at time t
- responses of k_t^i to shocks v_t to the hidden Markov demand state θ_t process are **smaller** in the two-noisy-signal structure
- responses of k_t^i to shocks v_t to the hidden Markov demand state θ_t process are **smallest** in the one-noisy-signal structure

With respect to the iid demand shocks e_t the graphs show that

- responses of k_t^i to shocks e_t to the hidden Markov demand state θ_t process are **smallest** (i.e., nonexistent) in the no-noisy-signal structure in which the firm observes θ_t at time t
- responses of k_t^i to shocks e_t to the hidden Markov demand state θ_t process are **larger** in the two-noisy-signal structure
- responses of k_t^i to idiosyncratic *own-market* noise-shocks e_t are **largest** in the one-noisy-signal structure

Among other things, these findings indicate that time series correlations and coherences between outputs in the two industries are higher in the two-noisy-signals or **pooling** model than they are in the one-noisy signal model.

The enhanced influence of the shocks v_t to the hidden Markov demand state θ_t process that emerges from the two-noisy-signal model relative to the one-noisy-signal model is a symptom of a lower equilibrium hidden-state reconstruction error variance in the two-signal model:

```
display(Latex('$\\textbf{Reconstruction error variances}$'))
display(Latex(f'One-noise structure: {round(p_one, 6)}'))
display(Latex(f'Two-noise structure: {round(p_two, 6)}'))
```

Reconstruction error variances

$$One - noisestructure : 0.36618$$

$$Two - noisestructure : 0.324062$$

Kalman gains for the two structures are

```
display(Latex('$\\textbf{Kalman Gains}$'))
display(Latex(f'One noisy-signal structure: {round(κ_one, 6)}'))
display(Latex(f'Two noisy-signals structure: {round(κ_two, 6)}'))
```

Kalman Gains

$$Onenoisy - signalstructure : 0.403404$$

$$Twonoisy - signalsstructure : 0.25716$$

Another lesson that comes from the preceding three-panel graph is that the presence of iid noise ϵ_t^i in industry i generates a response in k_t^{-i} in the two-noisy-signal structure, but not in the one-noisy-signal structure.

34.12 Notes on History of the Problem

To truncate what he saw as an intractable, infinite dimensional state space, Townsend constructed an approximating model in which the common hidden Markov demand shock is revealed to all firms after a fixed number of periods.

Thus,

- Townsend wanted to assume that at time t firms in industry i observe $k_t^i, Y_t^i, P_t^i, (P^{-i})^t$, where $(P^{-i})^t$ is the history of prices in the other market up to time t.
- Because that turned out to be too challenging, Townsend made a sensible alternative assumption that eased his calculations: that after a large number S of periods, firms in industry i observe the hidden Markov component of the demand shock θ_{t-S}.

Townsend argued that the more manageable model could do a good job of approximating the intractable model in which the Markov component of the demand shock remains unobserved for ever.

By applying technical machinery of [Pearlman *et al.*, 1986], [Pearlman and Sargent, 2005] showed that there is a recursive representation of the equilibrium of the perpetually and symmetrically uninformed model that Townsend wanted to solve [Townsend, 1983].

A reader of [Pearlman and Sargent, 2005] will notice that their representation of the equilibrium of Townsend's model exactly matches that of the **pooling equilibrium** presented here.

We have structured our notation in this lecture to faciliate comparison of the **pooling equilibrium** constructed here with the equilibrium of Townsend's model reported in [Pearlman and Sargent, 2005].

The computational method of [Pearlman and Sargent, 2005] is recursive: it enlists the Kalman filter and invariant subspace methods for solving systems of Euler equations[5] .

[5] See [Anderson *et al.*, 1996] for an account of invariant subspace methods.

As [Singleton, 1987], [Kasa, 2000], and [Sargent, 1991] also found, the equilibrium is fully revealing: observed prices tell participants in industry i all of the information held by participants in market $-i$ ($-i$ means not i).

This means that higher-order beliefs play no role: observing equilibrium prices in effect lets decision makers pool their information sets[6] .

The disappearance of higher order beliefs means that decision makers in this model do not really face a problem of forecasting the forecasts of others.

Because those forecasts are the same as their own, they know them.

34.12.1 Further historical remarks

Sargent [Sargent, 1991] proposed a way to compute an equilibrium without making Townsend's approximation.

Extending the reasoning of [Muth, 1960], Sargent noticed that it is possible to summarize the relevant history with a low dimensional object, namely, a small number of current and lagged forecasting errors.

Positing an equilibrium in a space of perceived laws of motion for endogenous variables that takes the form of a vector autoregressive, moving average, Sargent described an equilibrium as a fixed point of a mapping from the perceived law of motion to the actual law of motion of that form.

Sargent worked in the time domain and proceeded to guess and verify the appropriate orders of the autoregressive and moving average pieces of the equilibrium representation.

By working in the frequency domain [Kasa, 2000] showed how to discover the appropriate orders of the autoregressive and moving average parts, and also how to compute an equilibrium.

The [Pearlman and Sargent, 2005] recursive computational method, which stays in the time domain, also discovered appropriate orders of the autoregressive and moving average pieces.

In addition, by displaying equilibrium representations in the form of [Pearlman *et al.*, 1986], [Pearlman and Sargent, 2005] showed how the moving average piece is linked to the innovation process of the hidden persistent component of the demand shock.

That scalar innovation process is the additional state variable contributed by the problem of extracting a signal from equilibrium prices that decision makers face in Townsend's model.

---

[6] See [Allen *et al.*, 2002] for a discussion of information assumptions needed to create a situation in which higher order beliefs appear in equilibrium decision rules. A way to read our findings in light of [Allen *et al.*, 2002] is that, relative to the number of signals agents observe, Townsend's section 8 model has too few random shocks to get higher order beliefs to play a role.

Part VII

Asset Pricing and Finance

CHAPTER

THIRTYFIVE

ASSET PRICING II: THE LUCAS ASSET PRICING MODEL

35.1 Overview

As stated in an earlier lecture, an asset is a claim on a stream of prospective payments.

What is the correct price to pay for such a claim?

The elegant asset pricing model of Lucas [Lucas, 1978] attempts to answer this question in an equilibrium setting with risk-averse agents.

While we mentioned some consequences of Lucas' model earlier, it is now time to work through the model more carefully and try to understand where the fundamental asset pricing equation comes from.

A side benefit of studying Lucas' model is that it provides a beautiful illustration of model building in general and equilibrium pricing in competitive models in particular.

Another difference to our first asset pricing lecture is that the state space and shock will be continuous rather than discrete.

Let's start with some imports:

```
import numpy as np
from numba import njit, prange
from scipy.stats import lognorm
import matplotlib.pyplot as plt
```

35.2 The Lucas Model

Lucas studied a pure exchange economy with a representative consumer (or household), where

- *Pure exchange* means that all endowments are exogenous.
- *Representative* consumer means that either
  - there is a single consumer (sometimes also referred to as a household), or
  - all consumers have identical endowments and preferences

Either way, the assumption of a representative agent means that prices adjust to eradicate desires to trade.

This makes it very easy to compute competitive equilibrium prices.

35.2.1 Basic Setup

Let's review the setup.

Assets

There is a single "productive unit" that costlessly generates a sequence of consumption goods $\{y_t\}_{t=0}^{\infty}$.

Another way to view $\{y_t\}_{t=0}^{\infty}$ is as a *consumption endowment* for this economy.

We will assume that this endowment is Markovian, following the exogenous process

$$y_{t+1} = G(y_t, \xi_{t+1})$$

Here $\{\xi_t\}$ is an IID shock sequence with known distribution ϕ and $y_t \geq 0$.

An asset is a claim on all or part of this endowment stream.

The consumption goods $\{y_t\}_{t=0}^{\infty}$ are nonstorable, so holding assets is the only way to transfer wealth into the future.

For the purposes of intuition, it's common to think of the productive unit as a "tree" that produces fruit.

Based on this idea, a "Lucas tree" is a claim on the consumption endowment.

Consumers

A representative consumer ranks consumption streams $\{c_t\}$ according to the time separable utility functional

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t u(c_t) \tag{35.1}$$

Here

- $\beta \in (0, 1)$ is a fixed discount factor.
- u is a strictly increasing, strictly concave, continuously differentiable period utility function.
- $\mathbb{E}$ is a mathematical expectation.

35.2.2 Pricing a Lucas Tree

What is an appropriate price for a claim on the consumption endowment?

We'll price an *ex-dividend* claim, meaning that

- the seller retains this period's dividend
- the buyer pays p_t today to purchase a claim on
  - y_{t+1} and
  - the right to sell the claim tomorrow at price p_{t+1}

Since this is a competitive model, the first step is to pin down consumer behavior, taking prices as given.

Next, we'll impose equilibrium constraints and try to back out prices.

In the consumer problem, the consumer's control variable is the share π_t of the claim held in each period.

Thus, the consumer problem is to maximize (35.1) subject to

$$c_t + \pi_{t+1} p_t \leq \pi_t y_t + \pi_t p_t$$

along with $c_t \geq 0$ and $0 \leq \pi_t \leq 1$ at each t.

The decision to hold share π_t is actually made at time $t-1$.

But this value is inherited as a state variable at time t, which explains the choice of subscript.

The Dynamic Program

We can write the consumer problem as a dynamic programming problem.

Our first observation is that prices depend on current information, and current information is really just the endowment process up until the current period.

In fact, the endowment process is Markovian, so that the only relevant information is the current state $y \in \mathbb{R}_+$ (dropping the time subscript).

This leads us to guess an equilibrium where price is a function p of y.

Remarks on the solution method

- Since this is a competitive (read: price taking) model, the consumer will take this function p as given.
- In this way, we determine consumer behavior given p and then use equilibrium conditions to recover p.
- This is the standard way to solve competitive equilibrium models.

Using the assumption that price is a given function p of y, we write the value function and constraint as

$$v(\pi, y) = \max_{c, \pi'} \left\{ u(c) + \beta \int v(\pi', G(y, z)) \phi(dz) \right\}$$

subject to

$$c + \pi' p(y) \leq \pi y + \pi p(y) \tag{35.2}$$

We can invoke the fact that utility is increasing to claim equality in (35.2) and hence eliminate the constraint, obtaining

$$v(\pi, y) = \max_{\pi'} \left\{ u[\pi(y + p(y)) - \pi' p(y)] + \beta \int v(\pi', G(y, z)) \phi(dz) \right\} \tag{35.3}$$

The solution to this dynamic programming problem is an optimal policy expressing either π' or c as a function of the state (π, y).

- Each one determines the other, since $c(\pi, y) = \pi(y + p(y)) - \pi'(\pi, y) p(y)$

Next Steps

What we need to do now is determine equilibrium prices.

It seems that to obtain these, we will have to

1. Solve this two-dimensional dynamic programming problem for the optimal policy.
2. Impose equilibrium constraints.
3. Solve out for the price function $p(y)$ directly.

However, as Lucas showed, there is a related but more straightforward way to do this.

Equilibrium Constraints

Since the consumption good is not storable, in equilibrium we must have $c_t = y_t$ for all t.

In addition, since there is one representative consumer (alternatively, since all consumers are identical), there should be no trade in equilibrium.

In particular, the representative consumer owns the whole tree in every period, so $\pi_t = 1$ for all t.

Prices must adjust to satisfy these two constraints.

The Equilibrium Price Function

Now observe that the first-order condition for (35.3) can be written as

$$u'(c)p(y) = \beta \int v_1'(\pi', G(y,z))\phi(dz)$$

where v_1' is the derivative of v with respect to its first argument.

To obtain v_1' we can simply differentiate the right-hand side of (35.3) with respect to π, yielding

$$v_1'(\pi, y) = u'(c)(y + p(y))$$

Next, we impose the equilibrium constraints while combining the last two equations to get

$$p(y) = \beta \int \frac{u'[G(y,z)]}{u'(y)} [G(y,z) + p(G(y,z))]\phi(dz) \tag{35.4}$$

In sequential rather than functional notation, we can also write this as

$$p_t = \mathbb{E}_t \left[\beta \frac{u'(c_{t+1})}{u'(c_t)} (y_{t+1} + p_{t+1}) \right] \tag{35.5}$$

This is the famous consumption-based asset pricing equation.

Before discussing it further we want to solve out for prices.

35.2.3 Solving the Model

Equation (35.4) is a *functional equation* in the unknown function p.

The solution is an equilibrium price function p^*.

Let's look at how to obtain it.

Setting up the Problem

Instead of solving for it directly we'll follow Lucas' indirect approach, first setting

$$f(y) := u'(y)p(y) \tag{35.6}$$

so that (35.4) becomes

$$f(y) = h(y) + \beta \int f[G(y,z)]\phi(dz) \tag{35.7}$$

Here $h(y) := \beta \int u'[G(y,z)]G(y,z)\phi(dz)$ is a function that depends only on the primitives.

Equation (35.7) is a functional equation in f.

The plan is to solve out for f and convert back to p via (35.6).

To solve (35.7) we'll use a standard method: convert it to a fixed point problem.

First, we introduce the operator T mapping f into Tf as defined by

$$(Tf)(y) = h(y) + \beta \int f[G(y,z)]\phi(dz) \tag{35.8}$$

In what follows, we refer to T as the Lucas operator.

The reason we do this is that a solution to (35.7) now corresponds to a function f^* satisfying $(Tf^*)(y) = f^*(y)$ for all y.

In other words, a solution is a *fixed point* of T.

This means that we can use fixed point theory to obtain and compute the solution.

A Little Fixed Point Theory

Let $cb\mathbb{R}_+$ be the set of continuous bounded functions $f \colon \mathbb{R}_+ \to \mathbb{R}_+$.

We now show that

1. T has exactly one fixed point f^* in $cb\mathbb{R}_+$.
2. For any $f \in cb\mathbb{R}_+$, the sequence $T^k f$ converges uniformly to f^*.

Note: If you find the mathematics heavy going you can take 1–2 as given and skip to the *next section*

Recall the Banach contraction mapping theorem.

It tells us that the previous statements will be true if we can find an $\alpha < 1$ such that

$$\|Tf - Tg\| \leq \alpha \|f - g\|, \qquad \forall \, f, g \in cb\mathbb{R}_+ \tag{35.9}$$

Here $\|h\| := \sup_{x \in \mathbb{R}_+} |h(x)|$.

To see that (35.9) is valid, pick any $f, g \in cb\mathbb{R}_+$ and any $y \in \mathbb{R}_+$.

Observe that, since integrals get larger when absolute values are moved to the inside,

$$\begin{aligned} |Tf(y) - Tg(y)| &= \left| \beta \int f[G(y,z)]\phi(dz) - \beta \int g[G(y,z)]\phi(dz) \right| \\ &\leq \beta \int |f[G(y,z)] - g[G(y,z)]| \, \phi(dz) \\ &\leq \beta \int \|f - g\| \phi(dz) \\ &= \beta \|f - g\| \end{aligned}$$

Since the right-hand side is an upper bound, taking the sup over all y on the left-hand side gives (35.9) with $\alpha := \beta$.

35.2.4 Computation – An Example

The preceding discussion tells that we can compute f^* by picking any arbitrary $f \in cb\mathbb{R}_+$ and then iterating with T.

The equilibrium price function p^* can then be recovered by $p^*(y) = f^*(y)/u'(y)$.

Let's try this when $\ln y_{t+1} = \alpha \ln y_t + \sigma \epsilon_{t+1}$ where $\{\epsilon_t\}$ is IID and standard normal.

Utility will take the isoelastic form $u(c) = c^{1-\gamma}/(1-\gamma)$, where $\gamma > 0$ is the coefficient of relative risk aversion.

We will set up a `LucasTree` class to hold parameters of the model

```
class LucasTree:
    """
    Class to store parameters of the Lucas tree model.

    """

    def __init__(self,
                 γ=2,            # CRRA utility parameter
                 β=0.95,         # Discount factor
                 α=0.90,         # Correlation coefficient
                 σ=0.1,          # Volatility coefficient
                 grid_size=100):

        self.γ, self.β, self.α, self.σ = γ, β, α, σ

        # Set the grid interval to contain most of the mass of the
        # stationary distribution of the consumption endowment
        ssd = self.σ / np.sqrt(1 - self.α**2)
        grid_min, grid_max = np.exp(-4 * ssd), np.exp(4 * ssd)
        self.grid = np.linspace(grid_min, grid_max, grid_size)
        self.grid_size = grid_size

        # Set up distribution for shocks
        self.ϕ = lognorm(σ)
        self.draws = self.ϕ.rvs(500)

        self.h = np.empty(self.grid_size)
        for i, y in enumerate(self.grid):
            self.h[i] = β * np.mean((y**α * self.draws)**(1 - γ))
```

The following function takes an instance of the `LucasTree` and generates a jitted version of the Lucas operator

```
def operator_factory(tree, parallel_flag=True):

    """
    Returns approximate Lucas operator, which computes and returns the
    updated function Tf on the grid points.

    tree is an instance of the LucasTree class

    """

    grid, h = tree.grid, tree.h
    α, β = tree.α, tree.β
    z_vec = tree.draws
```

(continues on next page)

(continued from previous page)

```
    @njit(parallel=parallel_flag)
    def T(f):
        """
        The Lucas operator
        """

        # Turn f into a function
        Af = lambda x: np.interp(x, grid, f)

        Tf = np.empty_like(f)
        # Apply the T operator to f using Monte Carlo integration
        for i in prange(len(grid)):
            y = grid[i]
            Tf[i] = h[i] + β * np.mean(Af(y**α * z_vec))

        return Tf

    return T
```

To solve the model, we write a function that iterates using the Lucas operator to find the fixed point.

```
def solve_model(tree, tol=1e-6, max_iter=500):
    """
    Compute the equilibrium price function associated with Lucas
    tree

    * tree is an instance of LucasTree

    """
    # Simplify notation
    grid, grid_size = tree.grid, tree.grid_size
    γ = tree.γ

    T = operator_factory(tree)

    i = 0
    f = np.ones_like(grid)  # Initial guess of f
    error = tol + 1
    while error > tol and i < max_iter:
        Tf = T(f)
        error = np.max(np.abs(Tf - f))
        f = Tf
        i += 1

    price = f * grid**γ  # Back out price vector

    return price
```

Solving the model and plotting the resulting price function

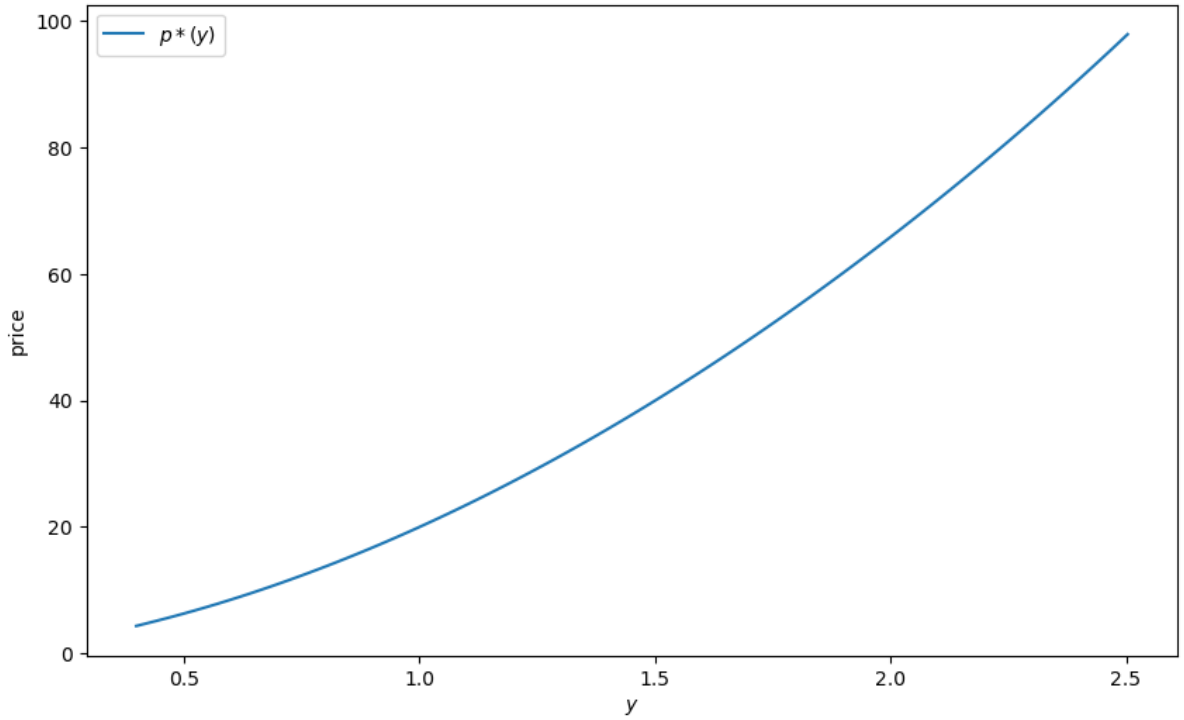
```
tree = LucasTree()
price_vals = solve_model(tree)

fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(tree.grid, price_vals, label='$p*(y)$')
```

(continues on next page)

(continued from previous page)

```
ax.set_xlabel('$y$')
ax.set_ylabel('price')
ax.legend()
plt.show()
```



We see that the price is increasing, even if we remove all serial correlation from the endowment process.

The reason is that a larger current endowment reduces current marginal utility.

The price must therefore rise to induce the household to consume the entire endowment (and hence satisfy the resource constraint).

What happens with a more patient consumer?

Here the orange line corresponds to the previous parameters and the green line is price when $\beta = 0.98$.

We see that when consumers are more patient the asset becomes more valuable, and the price of the Lucas tree shifts up.

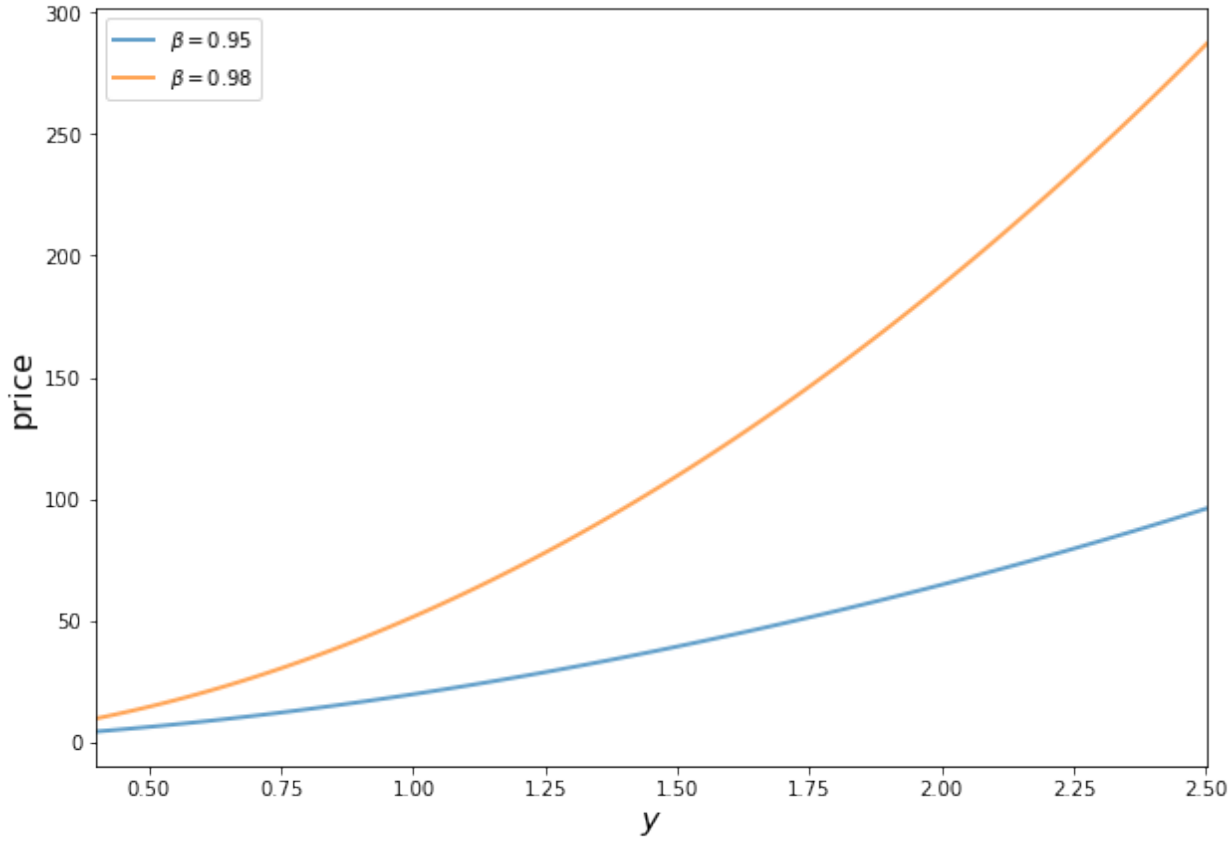
Exercise 1 asks you to replicate this figure.

35.3 Exercises

Exercise 35.3.1

Replicate *the figure* to show how discount factors affect prices.

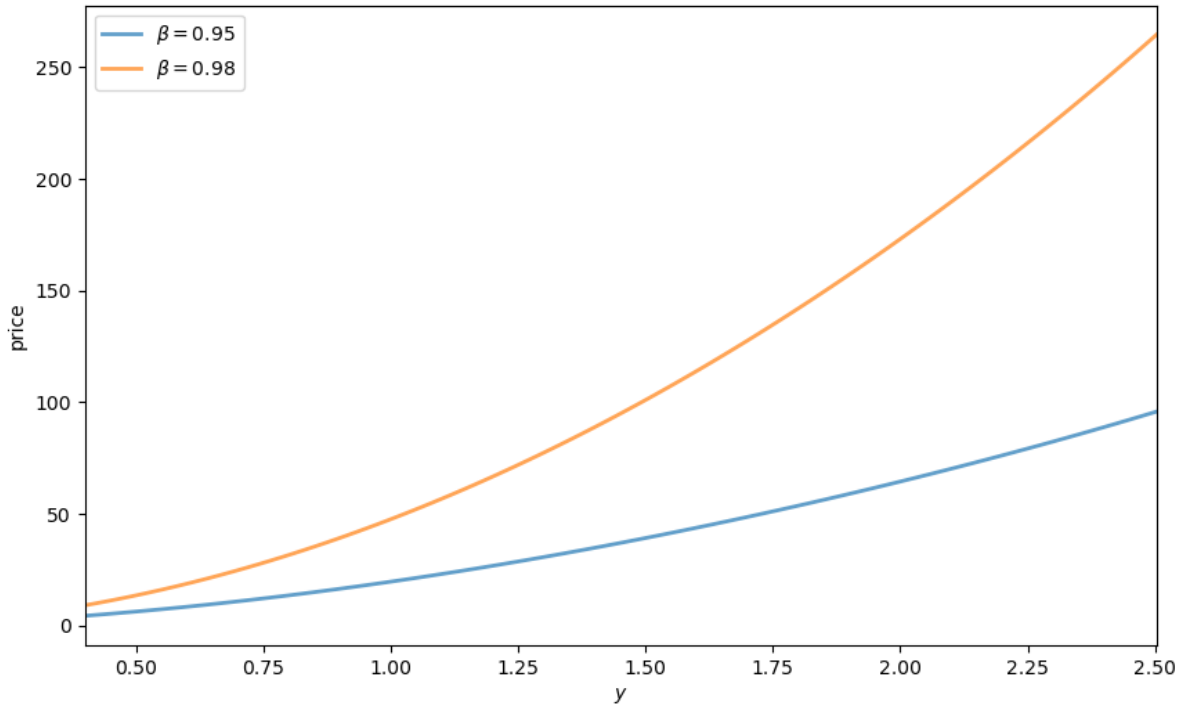
Solution to Exercise 35.3.1


$\beta = 0.95$
$\beta = 0.98$
price
y
300
250
200
150
100
50
0
0.50
0.75
1.00
1.25
1.50
1.75
2.00
2.25
2.50

```
fig, ax = plt.subplots(figsize=(10, 6))

for β in (.95, 0.98):
    tree = LucasTree(β=β)
    grid = tree.grid
    price_vals = solve_model(tree)
    label = rf'$\beta = {β}$'
    ax.plot(grid, price_vals, lw=2, alpha=0.7, label=label)

ax.legend(loc='upper left')
ax.set(xlabel='$y$', ylabel='price', xlim=(min(grid), max(grid)))
plt.show()
```



CHAPTER

THIRTYSIX

ELEMENTARY ASSET PRICING THEORY

36.1 Overview

This lecture is about some implications of asset-pricing theories that are based on the equation $EmR = 1$, where R is the gross return on an asset, m is a stochastic discount factor, and E is a mathematical expectation with respect to a joint probability distribution of R and m.

Instances of this equation occur in many models.

Note: Chapter 1 of [Ljungqvist and Sargent, 2018] describes the role that this equation plays in a diverse set of models in macroeconomics, monetary economics, and public finance.

We aim to convey insights about empirical implications of this equation brought out in the work of Lars Peter Hansen [Hansen and Richard, 1987] and Lars Peter Hansen and Ravi Jagannathan [Hansen and Jagannathan, 1991].

By following their footsteps, from that single equation we'll derive

- a mean-variance frontier
- a single-factor model of excess returns

To do this, we use two ideas:

- the equation $EmR = 1$ that is implied by an application of a *law of one price*
- a Cauchy-Schwartz inequality

In particular, we'll apply a Cauchy-Schwartz inequality to a population linear least squares regression equation that is implied by $EmR = 1$.

We'll also describe how practitioners have implemented the model using

- cross sections of returns on many assets
- time series of returns on various assets

For background and basic concepts about linear least squares projections, see our lecture orthogonal projections and their applications.

As a sequel to the material here, please see our lecture two modifications of mean-variance portfolio theory.

36.2 Key Equation

We begin with a **key asset pricing equation**:

$$EmR^i = 1 \tag{36.1}$$

for $i = 1, \ldots, I$ and where

$$\begin{aligned} m &= \text{ stochastic discount factor} \\ R^i &= \text{random gross return on asset } i \\ E &\sim \text{ mathematical expectation} \end{aligned}$$

The random gross return R^i for every asset i and the scalar stochastic discount factor m live in a common probability space.

[Hansen and Richard, 1987] and [Hansen and Jagannathan, 1991] explain how **existence** of a scalar stochastic discount factor that verifies equation (36.1) is implied by a **law of one price** that requires that all portfolios of assets that bring the same payouts have the same price.

They also explain how the **absence of an arbitrage** opportunity implies that the stochastic discount factor $m \geq 0$.

In order to say something about the **uniqueness** of a stochastic discount factor, we would have to impose more theoretical structure than we do in this lecture.

For example, in **complete markets** models like those illustrated in this lecture equilibrium capital structures with incomplete markets, the stochastic discount factor is unique.

In **incomplete markets** models like those illustrated in this lecture the Aiyagari model, the stochastic discount factor is not unique.

36.3 Implications of Key Equation

We combine key equation (36.1) with a remark of Lars Peter Hansen that "asset pricing theory is all about covariances".

Note: Lars Hansen's remark is a concise summary of ideas in [Hansen and Richard, 1987] and [Hansen and Jagannathan, 1991]. Important foundations of these ideas were set down by [Ross, 1976], [Ross, 1978], [Harrison and Kreps, 1979], [Kreps, 1981], and [Chamberlain and Rothschild, 1983].

This remark of Lars Hansen refers to the fact that interesting restrictions can be deduced by recognizing that EmR^i is a component of the covariance between m and R^i and then using that fact to rearrange equation (36.1).

Let's do this step by step.

First note that the definition of a covariance $\text{cov}\left(m, R^i\right) = E(m - Em)(R^i - ER^i)$ implies that

$$EmR^i = EmER^i + \text{cov}\left(m, R^i\right)$$

Substituting this result into equation (36.1) gives

$$1 = EmER^i + \text{cov}\left(m, R^i\right) \tag{36.2}$$

Next note that for a risk-free asset with non-random gross return R^f, equation (36.1) becomes

$$1 = ER^f m = R^f Em.$$

This is true because we can pull the constant R^f outside the mathematical expectation.

It follows that the gross return on a risk-free asset is

$$R^f = 1/E(m)$$

Using this formula for R^f in equation (36.2) and rearranging, it follows that

$$R^f = ER^i + \text{cov}\left(m, R^i\right) R^f$$

which can be rearranged to become

$$ER^i = R^f - \text{cov}\left(m, R^i\right) R^f.$$

It follows that we can express an **excess return** $ER^i - R^f$ on asset i relative to the risk-free rate as

$$ER^i - R^f = -\text{cov}\left(m, R^i\right) R^f \tag{36.3}$$

Equation (36.3) can be rearranged to display important parts of asset pricing theory.

36.4 Expected Return - Beta Representation

We can obtain the celebrated **expected-return-Beta -representation** for gross return R^i by simply rearranging excess return equation (36.3) to become

$$ER^i = R^f + \left(\underbrace{\frac{\text{cov}\left(R^i, m\right)}{\text{var}(m)}}_{\beta_{i,m}=\text{regression coefficient}} \right) \left(\underbrace{-\frac{\text{var}(m)}{E(m)}}_{\lambda_m=\text{price of risk}} \right)$$

or

$$ER^i = R^f + \beta_{i,m} \lambda_m \tag{36.4}$$

Here

- $\beta_{i,m}$ is a (population) least squares regression coefficient of gross return R^i on stochastic discount factor m
- λ_m is minus the variance of m divided by the mean of m, an object that is sometimes called a **price of risk**.

Because $\lambda_m < 0$, equation (36.4) asserts that

- assets whose returns are **positively** correlated with the stochastic discount factor (SDF) m have expected returns **lower** than the risk-free rate R^f
- assets whose returns are **negatively** correlated with the SDF m have expected returns **higher** than the risk-free rate R^f

These patterns will be discussed more below.

In particular, we'll see that returns that are **perfectly** negatively correlated with the SDF m have a special status:

- they are on a **mean-variance frontier**

Before we dive into that more, we'll pause to look at an example of an SDF.

To interpret representation (36.4), the following widely used example helps.

Example

Let c_t be the logarithm of the consumption of a *representative consumer* or just a single consumer for whom we have consumption data.

A popular model of m is

$$m_{t+1} = \beta \frac{U'(C_{t+1})}{U'(C_t)}$$

where C_t is consumption at time t, $\beta = \exp(-\rho)$ is a discount **factor** with ρ being the discount **rate**, and $U(\cdot)$ is a concave, twice-diffential utility function.

For a constant relative risk aversion (CRRA) utility function $U(C) = \frac{C^{1-\gamma}}{1-\gamma}$ utility function $U'(C) = C^{-\gamma}$.

In this case, letting $c_t = \log(C_t)$, we can write m_{t+1} as

$$m_{t+1} = \exp(-\rho) \exp(-\gamma(c_{t+1} - c_t))$$

where $\rho > 0$, $\gamma > 0$.

A popular model for the growth of log of consumption is

$$c_{t+1} - c_t = \mu + \sigma_c \epsilon_{t+1}$$

where $\epsilon_{t+1} \sim \mathcal{N}(0, 1)$.

Here $\{c_t\}$ is a random walk with drift μ, a good approximation to US per capital consumption growth.

Again here

- $\gamma > 0$ is a coefficient of relative risk aversion
- $\rho > 0$ is a fixed intertemporal discount rate

So we have

$$m_{t+1} = \exp(-\rho) \exp(-\gamma\mu - \gamma\sigma_c \epsilon_{t+1})$$

In this case

$$Em_{t+1} = \exp(-\rho) \exp\left(-\gamma\mu + \frac{\sigma_c^2 \gamma^2}{2}\right)$$

and

$$\text{var}(m_{t+1}) = E(m)[\exp(\sigma_c^2 \gamma^2) - 1)]$$

When $\gamma > 0$, it is true that

- when consumption growth is **high**, m is **low**
- when consumption growth is **low**, m is **high**

According to representation (36.4), an asset with a gross return R^i that is expected to be **high** when consumption growth is **low** has $\beta_{i,m}$ positive and a **low** expected return.

- because it has a high gross return when consumption growth is low, it is a good hedge against consumption risk. That justifies its low average return.

An asset with an R^i that is **low** when consumption growth is **low** has $\beta_{i,m}$ negative and a **high** expected return.

- because it has a low gross return when consumption growth is low, it is a poor hedge against consumption risk. That justifies its high average return.

36.5 Mean-Variance Frontier

Now we'll derive the celebrated **mean-variance frontier**.

We do this using a method deployed by Lars Peter Hansen and Scott Richard [Hansen and Richard, 1987].

Note: Methods of Hansen and Richard are described and used extensively by [Cochrane, 2005].

Their idea was rearrange the key equation (36.1), namely, $EmR^i = 1$, and then to apply a Cauchy-Schwarz inequality.

A convenient way to remember the Cauchy-Schwartz inequality in our context is that it says that an R^2 in any regression has to be less than or equal to 1.

(Please note that here R^2 denotes the coefficient of determination in a regression, not a return on an asset!)

Let's apply that idea to deduce

$$1 = E\left(mR^i\right) = E(m)E\left(R^i\right) + \rho_{m,R^i}\sigma(m)\sigma\left(R^i\right) \tag{36.5}$$

where the correlation coefficient ρ_{m,R^i} is defined as

$$\rho_{m,R^i} \equiv \frac{\text{cov}\left(m, R^i\right)}{\sigma(m)\sigma\left(R^i\right)}$$

and where $\sigma(\cdot)$ denotes the standard deviation of the variable in parentheses

Equation (36.5) implies

$$ER^i = R^f - \rho_{m,R^i}\frac{\sigma(m)}{E(m)}\sigma\left(R^i\right)$$

Because $\rho_{m,R^i} \in [-1, 1]$, it follows that $|\rho_{m,R^i}| \leq 1$ and that

$$\left|ER^i - R^f\right| \leqslant \frac{\sigma(m)}{E(m)}\sigma\left(R^i\right) \tag{36.6}$$

Inequality (36.6) delineates a **mean-variance frontier**

(Actually, it looks more like a **mean-standard-deviation frontier**)

Evidently, points on the frontier correspond to gross returns that are perfectly correlated (either positively or negatively) with the stochastic discount factor m.

We summarize this observation as

$$\rho_{m,R^i} = \begin{cases} +1 & \implies R^i \text{ is on lower frontier} \\ -1 & \implies R^i \text{ is on an upper frontier} \end{cases}$$

Now let's use matplotlib to draw a mean variance frontier.

In drawing a frontier, we'll set $\sigma(m) = .25$ and $Em = .99$, values roughly consistent with what many studies calibrate from quarterly US data.

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

# Define the function to plot
def y(x, alpha, beta):
```

(continues on next page)

(continued from previous page)

```
    return alpha + beta*x
def z(x, alpha, beta):
    return alpha - beta*x

sigmam = .25
Em = .99

# Set the values of alpha and beta
alpha = 1/Em
beta = sigmam/Em

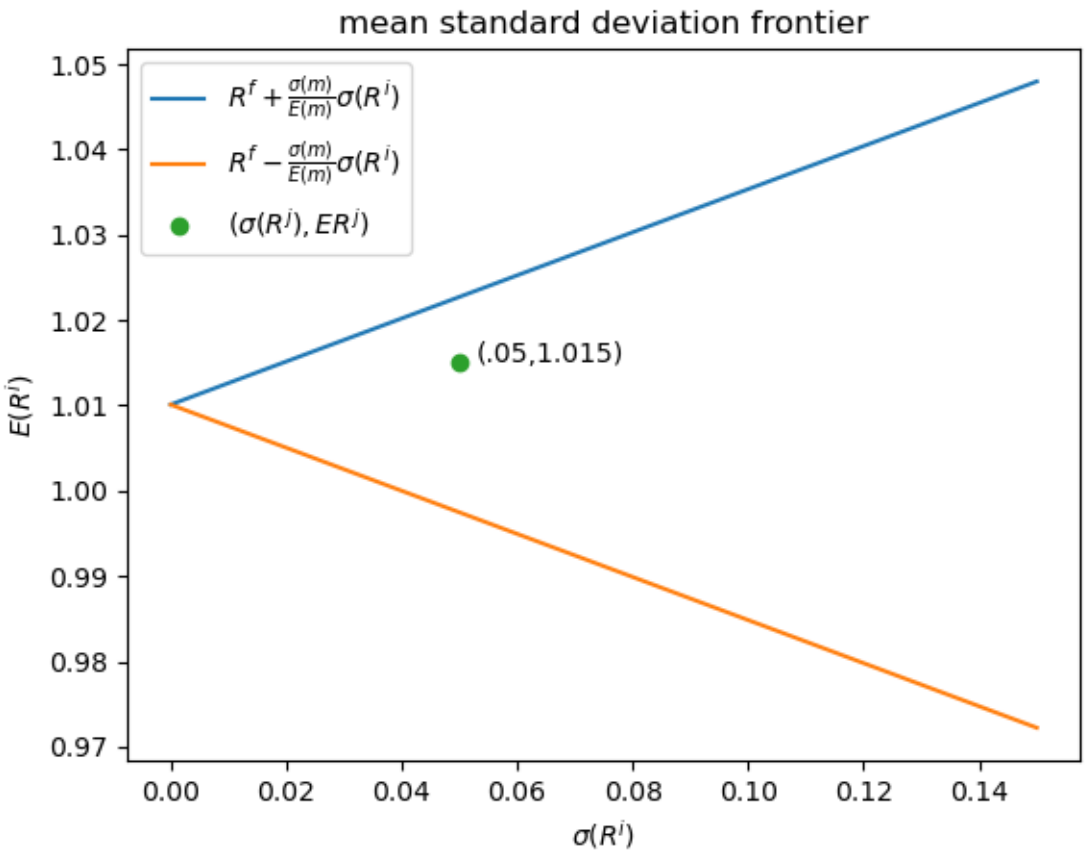
# Create a range of values for x
x = np.linspace(0, .15, 100)

# Calculate the values of y and z
y_values = y(x, alpha, beta)
z_values = z(x, alpha, beta)

# Create a figure and axes object
fig, ax = plt.subplots()

# Plot y
ax.plot(x, y_values, label=r'$R^f + \frac{\sigma(m)}{E(m)} \sigma(R^i)$')
ax.plot(x, z_values, label=r'$R^f - \frac{\sigma(m)}{E(m)} \sigma(R^i)$')

plt.title('mean standard deviation frontier')
plt.xlabel(r"$\sigma(R^i)$")
plt.ylabel(r"$E (R^i) $")
plt.text(.053, 1.015, "(.05,1.015)")
ax.plot(.05, 1.015, 'o', label=r"$(\sigma(R^j), E R^j)$")
# Add a legend and show the plot
ax.legend()
plt.show()
```



The figure shows two straight lines, the blue upper one being the locus of $(\sigma(R^i), E(R^i)$ pairs that are on the **mean-variance frontier** or **mean-standard-deviation frontier**.

The green dot refers to a return R^j that is **not** on the frontier and that has moments $(\sigma(R^j), ER^j) = (.05, 1.015)$.

It is described by the statistical model

$$R^j = R^i + \epsilon^j$$

where R^i is a return that is on the frontier and ϵ^j is a random variable that has mean zero and that is orthogonal to R^i.

Then $ER^j = ER^i$ and, as a consequence of R^j not being on the frontier,

$$\sigma^2(R^j) = \sigma^2(R^i) + \sigma^2(\epsilon^j)$$

The length of a horizontal line from the point $\sigma(R^j), E(R^j) = .05, 1.015$ to the frontier equals

$$\sqrt{\sigma^2(R^i) + \sigma^2(\epsilon^j)} - \sigma(R^i)$$

This is a measure of the part of the risk in R^j that is not priced because it is uncorrelated with the stochastic discount factor and so can be diversified away (i.e., averaged out to zero by holding a diversified portfolio).

36.6 Sharpe Ratios and the Price of Risk

An asset's **Sharpe ratio** is defined as

$$\frac{E(R^i) - R^f}{\sigma(R^i)}$$

The above figure reminds us that all assets R^i whose returns are on the mean-standard deviation frontier satisfy

$$\frac{E(R^i) - R^f}{\sigma(R^i)} = \frac{\sigma(m)}{Em}$$

The ratio $\frac{\sigma(m)}{Em}$ is often called the **market price of risk**.

Evidently it equals the maximum Sharpe ratio for any asset or portfolio of assets.

36.7 Mathematical Structure of Frontier

The mathematical structure of the mean-variance frontier described by inequality (36.6) implies that

- all returns on the frontier are perfectly correlated.

  Thus,

  - Let R^m, R^{mv} be two returns on the frontier.
  - Then for some scalar a, a return R^{mv} on the mean-variance frontier satisfies the affine equation $R^{mv} = R^f + a\left(R^m - R^f\right)$. This is an **exact** equation with no **residual**.

- each return R^{mv} that is on the mean-variance frontier is perfectly (negatively) correlated with m

  - $\left(\rho_{m,R^{mv}} = -1\right) \Rightarrow \begin{cases} m = a + bR^{mv} \\ R^{mv} = e + dm \end{cases}$ for some scalars a, b, e, d,

  Therefore, **any return on the mean-variance frontier is a legitimate stochastic discount factor**

- for any mean-variance-efficient return R^{mv} that is on the frontier but that is **not** R^f, there exists a **single-beta representation** for any return R^i that takes the form:

$$ER^i = R^f + \beta_{i,R^{mv}}\left[E\left(R^{mv}\right) - R^f\right] \tag{36.7}$$

- the regression coefficient $\beta_{i,R^{mv}}$ is often called asset i's **beta**
- The special case of a single-beta representation (36.7) with $R^i = R^{mv}$ is

  $ER^{mv} = R^f + 1 \cdot \left[E\left(R^{mv}\right) - R^f\right]$

36.8 Multi-factor Models

The single-beta representation (36.7) is a special case of the multi-factor model

$$ER^i = \gamma + \beta_{i,a}\lambda_a + \beta_{i,b}\lambda_b + \cdots$$

where λ_j is the price of being exposed to risk factor f_t^j and $\beta_{i,j}$ is asset i's exposure to that risk factor.

To uncover the $\beta_{i,j}$'s, one takes data on time series of the risk factors f_t^j that are being priced and specifies the following least squares regression

$$\begin{aligned} R_t^i = a_i + \beta_{i,a} f_t^a + \beta_{i,b} f_t^b + ... + \epsilon_t^i, \quad t = 1, 2, ..., T \\ \epsilon_t^i \perp f_t^j, i = 1, 2, ..., I; j = a, b, ... \end{aligned} \tag{36.8}$$

Special cases are:

- a popular **single-factor** model specifies the single factor f_t to be the return on the market portfolio
- another popular **single-factor** model called the **consumption-based model** specifies the factor to be $m_{t+1} = \beta \frac{u'(c_{t+1})}{u'(c_t)}$, where c_t is a representative consumer's time t consumption.

As a reminder, model objects are interpreted as follows:

- $\beta_{i,a}$ is the exposure of return R^i to risk factor f_a
- λ_a is the price of exposure to risk factor f_a

36.9 Empirical Implementations

We briefly describe empirical implementations of multi-factor generalizations of the single-factor model described above.

Two representations of a multi-factor model play importnt roles in empirical applications.

One is the time series regression (36.8)

The other representation entails a **cross-section regression** of **average returns** ER^i for assets $i = 1, 2, ..., I$ on **prices of risk** λ_j for $j = a, b, c, ...$

Here is the cross-section regression specification for a multi-factor model:

$$ER^i = \gamma + \beta_{i,a} \lambda_a + \beta_{i,b} \lambda_b + \cdots$$

Testing strategies:

Time-series and cross-section regressions play roles in both **estimating** and **testing** beta representation models.

The basic idea is to implement the following two steps.

Step 1:

- Estimate $a_i, \beta_{i,a}, \beta_{i,b}, \cdots$ by running a **time series regression:** R_t^i on a constant and $f_t^a, f_t^b, ...$

Step 2:

- take the $\beta_{i,j}$'s estimated in step one as regressors together with data on average returns ER^i over some period and then estimate the **cross-section regression**

$$\underbrace{E\left(R^i\right)}_{\text{average return over time series}} = \gamma + \underbrace{\beta_{i,a}}_{\text{regressor}} \underbrace{\lambda_a}_{\text{regressioncoefficient}} + \underbrace{\beta_{i,b}}_{\text{regressor}} \underbrace{\lambda_b}_{\text{regressioncoefficient}} + \cdots + \underbrace{\alpha_i}_{\text{pricing errors}}, i = 1, ..., I; \quad \underbrace{\alpha_i \perp \beta_{i,j}, j = a, b, ...}_{\text{least squares orthogonality conditio}}$$

- Here $\perp$ means **orthogonal to**
- estimate $\gamma, \lambda_a, \lambda_b, ...$ by an appropriate regression technique, recognizing that the regressors have been generated by a step 1 regression.

Note that presumably the risk-free return $ER^f = \gamma$.

For excess returns $R^{ei} = R^i - R^f$ we have

$$ER^{ei} = \beta_{i,a} \lambda_a + \beta_{i,b} \lambda_b + \cdots + \alpha_i, i = 1, ..., I$$

In the following exercises, we illustrate aspects of these empirical strategies on artificial data.

Our basic tools are random number generator that we shall use to create artificial samples that conform to the theory and least squares regressions that let us watch aspects of the theory at work.

These exercises will further convince us that asset pricing theory is mostly about covariances and least squares regressions.

36.10 Exercises

Let's start with some imports.

```
import numpy as np
import statsmodels.api as sm
import matplotlib.pyplot as plt
```

Lots of our calculations will involve computing population and sample OLS regressions.

So we define a function for simple univariate OLS regression that calls the `OLS` routine from `statsmodels`.

```
def simple_ols(X, Y, constant=False):

    if constant:
        X = sm.add_constant(X)

    model = sm.OLS(Y, X)
    res = model.fit()

    β_hat = res.params[-1]
    σ_hat = np.sqrt(res.resid @ res.resid / res.df_resid)

    return β_hat, σ_hat
```

Exercise 36.10.1

Look at the equation,

$$R_t^i - R^f = \beta_{i,R^m}(R_t^m - R^f) + \sigma_i \varepsilon_{i,t}.$$

Verify that this equation is a regression equation.

Solution to Exercise 36.10.1

To verify that it is a **regression equation** we must show that the residual is orthogonal to the regressor.

Our assumptions about mutual orthogonality imply that

$$E\left[\epsilon_{i,t}\right] = 0, \quad E\left[\epsilon_{i,t} u_t\right] = 0$$

It follows that

$$\begin{aligned} E\left[\sigma_i \epsilon_{i,t}\left(R_t^m - R^f\right)\right] &= E\left[\sigma_i \epsilon_{i,t}\left(\xi + \lambda u_t\right)\right] \\ &= \sigma_i \xi E\left[\epsilon_{i,t}\right] + \sigma_i \lambda E\left[\epsilon_{i,t} u_t\right] \\ &= 0 \end{aligned}$$

Exercise 36.10.2

Give a formula for the regression coefficient β_{i,R^m}.

Solution to Exercise 36.10.2

The regression coefficient β_{i,R^m} is

$$\beta_{i,R^m} = \frac{Cov\left(R_t^i - R^f, R_t^m - R^f\right)}{Var\left(R_t^m - R^f\right)}$$

Exercise 36.10.3

As in many sciences, it is useful to distinguish a **direct problem** from an **inverse problem**.

- A direct problem involves simulating a particular model with known parameter values.
- An inverse problem involves using data to **estimate** or **choose** a particular parameter vector from a manifold of models indexed by a set of parameter vectors.

Please assume the parameter values provided below and then simulate 2000 observations from the theory specified above for 5 assets, $i = 1, \ldots, 5$.

$$\begin{aligned}
E\left[R^f\right] &= 0.02 \\
\sigma_f &= 0.00 \\
\xi &= 0.06 \\
\lambda &= 0.04 \\
\beta_{1,R^m} &= 0.2 \\
\sigma_1 &= 0.04 \\
\beta_{2,R^m} &= .4 \\
\sigma_2 &= 0.04 \\
\beta_{3,R^m} &= .6 \\
\sigma_3 &= 0.04 \\
\beta_{4,R^m} &= .8 \\
\sigma_4 &= 0.04 \\
\beta_{5,R^m} &= 1.0 \\
\sigma_5 &= 0.04
\end{aligned}$$

More Exercises

Now come some even more fun parts!

Our theory implies that there exist values of two scalars, a and b, such that a legitimate stochastic discount factor is:

$$m_t = a + bR_t^m$$

The parameters a, b must satisfy the following equations:

$$\begin{aligned}
E[(a + bR_t^m)R_t^m)] &= 1 \\
E[(a + bR_t^m)R_t^f)] &= 1
\end{aligned}$$

Solution to Exercise 36.10.3

Direct Problem:

```
# Code for the direct problem

# assign the parameter values
ERf = 0.02
σf = 0.00 # Zejin: Hi tom, here is where you manipulate σf
ξ = 0.06
λ = 0.08
βi = np.array([0.2, .4, .6, .8, 1.0])
σi = np.array([0.04, 0.04, 0.04, 0.04, 0.04])
```

```
# in this cell we set the number of assets and number of observations
# we first set T to a large number to verify our computation results
T = 2000
N = 5
```

```
# simulate i.i.d. random shocks
e = np.random.normal(size=T)
u = np.random.normal(size=T)
ϵ = np.random.normal(size=(N, T))
```

```
# simulate the return on a risk-free asset
Rf = ERf + σf * e

# simulate the return on the market portfolio
excess_Rm = ξ + λ * u
Rm = Rf + excess_Rm

# simulate the return on asset i
Ri = np.empty((N, T))
for i in range(N):
    Ri[i, :] = Rf + βi[i] * excess_Rm + σi[i] * ϵ[i, :]
```

Now that we have a panel of data, we'd like to solve the inverse problem by assuming the theory specified above and estimating the coefficients given above.

```
# Code for the inverse problem
```

Inverse Problem:

We will solve the inverse problem by simple OLS regressions.

1. estimate $E\left[R^f\right]$ and σ_f

```
ERf_hat, σf_hat = simple_ols(np.ones(T), Rf)
```

```
ERf_hat, σf_hat
```

```
(0.020000000000000046, 4.5114090308141905e-17)
```

Let's compare these with the *true* population parameter values.

```
ERf, σf
```

```
(0.02, 0.0)
```

2. ξ and λ

```
ξ_hat, λ_hat = simple_ols(np.ones(T), Rm - Rf)
```

```
ξ_hat, λ_hat
```

```
(0.060902213658939275, 0.08010231766993321)
```

```
ξ, λ
```

```
(0.06, 0.08)
```

3. β_{i,R^m} and σ_i

```
βi_hat = np.empty(N)
σi_hat = np.empty(N)

for i in range(N):
    βi_hat[i], σi_hat[i] = simple_ols(Rm - Rf, Ri[i, :] - Rf)
```

```
βi_hat, σi_hat
```

```
(array([0.20377995, 0.39909004, 0.59731331, 0.80367115, 1.00292618]),
 array([0.0402678 , 0.03989453, 0.04010159, 0.04023966, 0.03941183]))
```

```
βi, σi
```

```
(array([0.2, 0.4, 0.6, 0.8, 1. ]), array([0.04, 0.04, 0.04, 0.04, 0.04]))
```

Q: How close did your estimates come to the parameters we specified?

Exercise 36.10.4

Using the equations above, find a system of two **linear** equations that you can solve for a and b as functions of the parameters $(\lambda, \xi, E[R_f])$.

Write a function that can solve these equations.

Please check the **condition number** of a key matrix that must be inverted to determine a, b

---

Solution to Exercise 36.10.4

The system of two linear equations is shown below:

$$a((E(R^f)+\xi)+b((E(R^f)+\xi)^2+\lambda^2+\sigma_f^2)=1$$
$$aE(R^f)+b(E(R^f)^2+\xi E(R^f)+\sigma_f^2)=1$$

```
# Code here
def solve_ab(ERf, σf, λ, ξ):

    M = np.empty((2, 2))
    M[0, 0] = ERf + ξ
    M[0, 1] = (ERf + ξ) ** 2 + λ ** 2 + σf ** 2
    M[1, 0] = ERf
    M[1, 1] = ERf ** 2 + ξ * ERf + σf ** 2

    a, b = np.linalg.solve(M, np.ones(2))
    condM = np.linalg.cond(M)

    return a, b, condM
```

Let's try to solve a and b using the actual model parameters.

```
a, b, condM = solve_ab(ERf, σf, λ, ξ)
```

```
a, b, condM
```

```
(87.49999999999999, -468.7499999999999, 54.406619883717504)
```

Exercise 36.10.5

Using the estimates of the parameters that you generated above, compute the implied stochastic discount factor.

Solution to Exercise 36.10.5

Now let's pass $\hat{E}(R^f), \hat{\sigma}^f, \hat{\lambda}, \hat{\xi}$ to the function `solve_ab`.

```
a_hat, b_hat, M_hat = solve_ab(ERf_hat, σf_hat, λ_hat, ξ_hat)
```

```
a_hat, b_hat, M_hat
```

```
(88.39488080872451, -474.58381016107285, 55.43212472804429)
```

CHAPTER

THIRTYSEVEN

TWO MODIFICATIONS OF MEAN-VARIANCE PORTFOLIO THEORY

37.1 Overview

This lecture describes extensions to the classical mean-variance portfolio theory summarized in our lecture Elementary Asset Pricing Theory.

The classic theory described there assumes that a decision maker completely trusts the statistical model that he posits to govern the joint distribution of returns on a list of available assets.

Both extensions described here put distrust of that statistical model into the mind of the decision maker.

One is a model of Black and Litterman [Black and Litterman, 1992] that imputes to the decision maker distrust of historically estimated mean returns but still complete trust of estimated covariances of returns.

The second model also imputes to the decision maker doubts about his statistical model, but now by saying that, because of that distrust, the decision maker uses a version of robust control theory described in this lecture Robustness.

The famous **Black-Litterman** (1992) [Black and Litterman, 1992] portfolio choice model was motivated by the finding that with high frequency or moderately high frequency data, means are more difficult to estimate than variances.

A model of **robust portfolio choice** that we'll describe below also begins from the same starting point.

To begin, we'll take for granted that means are more difficult to estimate that covariances and will focus on how Black and Litterman, on the one hand, an robust control theorists, on the other, would recommend modifying the **mean-variance portfolio choice model** to take that into account.

At the end of this lecture, we shall use some rates of convergence results and some simulations to verify how means are more difficult to estimate than variances.

Among the ideas in play in this lecture will be

- Mean-variance portfolio theory
- Bayesian approaches to estimating linear regressions
- A risk-sensitivity operator and its connection to robust control theory

In summary, we'll describe two ways to modify the classic mean-variance portfolio choice model in ways designed to make its recommendations more plausible.

Both of the adjustments that we describe are designed to confront a widely recognized embarrassment to mean-variance portfolio theory, namely, that it usually implies taking very extreme long-short portfolio positions.

The two approaches build on a common and widespread hunch – that because it is much easier statistically to estimate covariances of excess returns than it is to estimate their means, it makes sense to adjust investors' subjective beliefs about mean returns in order to render more plausible decisions.

Let's start with some imports:

```
import numpy as np
import scipy.stats as stat
import matplotlib.pyplot as plt
from numba import jit
```

37.2 Mean-Variance Portfolio Choice

A risk-free security earns one-period net return r_f.

An $n \times 1$ vector of risky securities earns an $n \times 1$ vector $\vec{r} - r_f \mathbf{1}$ of *excess returns*, where $\mathbf{1}$ is an $n \times 1$ vector of ones.

The excess return vector is multivariate normal with mean μ and covariance matrix Σ, which we express either as

$$\vec{r} - r_f \mathbf{1} \sim \mathcal{N}(\mu, \Sigma)$$

or

$$\vec{r} - r_f \mathbf{1} = \mu + C\epsilon$$

where $\epsilon \sim \mathcal{N}(0, I)$ is an $n \times 1$ random vector.

Let w be an $n \times 1$ vector of portfolio weights.

A portfolio consisting w earns returns

$$w'(\vec{r} - r_f \mathbf{1}) \sim \mathcal{N}(w'\mu, w'\Sigma w)$$

The **mean-variance portfolio choice problem** is to choose w to maximize

$$U(\mu, \Sigma; w) = w'\mu - \frac{\delta}{2} w'\Sigma w \tag{37.1}$$

where $\delta > 0$ is a risk-aversion parameter. The first-order condition for maximizing (37.1) with respect to the vector w is

$$\mu = \delta \Sigma w$$

which implies the following design of a risky portfolio:

$$w = (\delta\Sigma)^{-1}\mu \tag{37.2}$$

37.3 Estimating Mean and Variance

The key inputs into the portfolio choice model (37.2) are

- estimates of the parameters μ, Σ of the random excess return vector$(\vec{r} - r_f \mathbf{1})$
- the risk-aversion parameter δ

A standard way of estimating μ is maximum-likelihood or least squares; that amounts to estimating μ by a sample mean of excess returns and estimating Σ by a sample covariance matrix.

37.4 Black-Litterman Starting Point

When estimates of μ and Σ from historical sample means and covariances have been combined with **plausible** values of the risk-aversion parameter δ to compute an optimal portfolio from formula (37.2), a typical outcome has been w's with **extreme long and short positions**.

A common reaction to these outcomes is that they are so implausible that a portfolio manager cannot recommend them to a customer.

```
np.random.seed(12)

N = 10                                                  # Number of assets
T = 200                                                 # Sample size

# random market portfolio (sum is normalized to 1)
w_m = np.random.rand(N)
w_m = w_m / (w_m.sum())

# True risk premia and variance of excess return (constructed
# so that the Sharpe ratio is 1)
μ = (np.random.randn(N) + 5)  /100      # Mean excess return (risk premium)
S = np.random.randn(N, N)         # Random matrix for the covariance matrix
V = S @ S.T           # Turn the random matrix into symmetric psd
# Make sure that the Sharpe ratio is one
Σ = V * (w_m @ μ)**2 / (w_m @ V @ w_m)

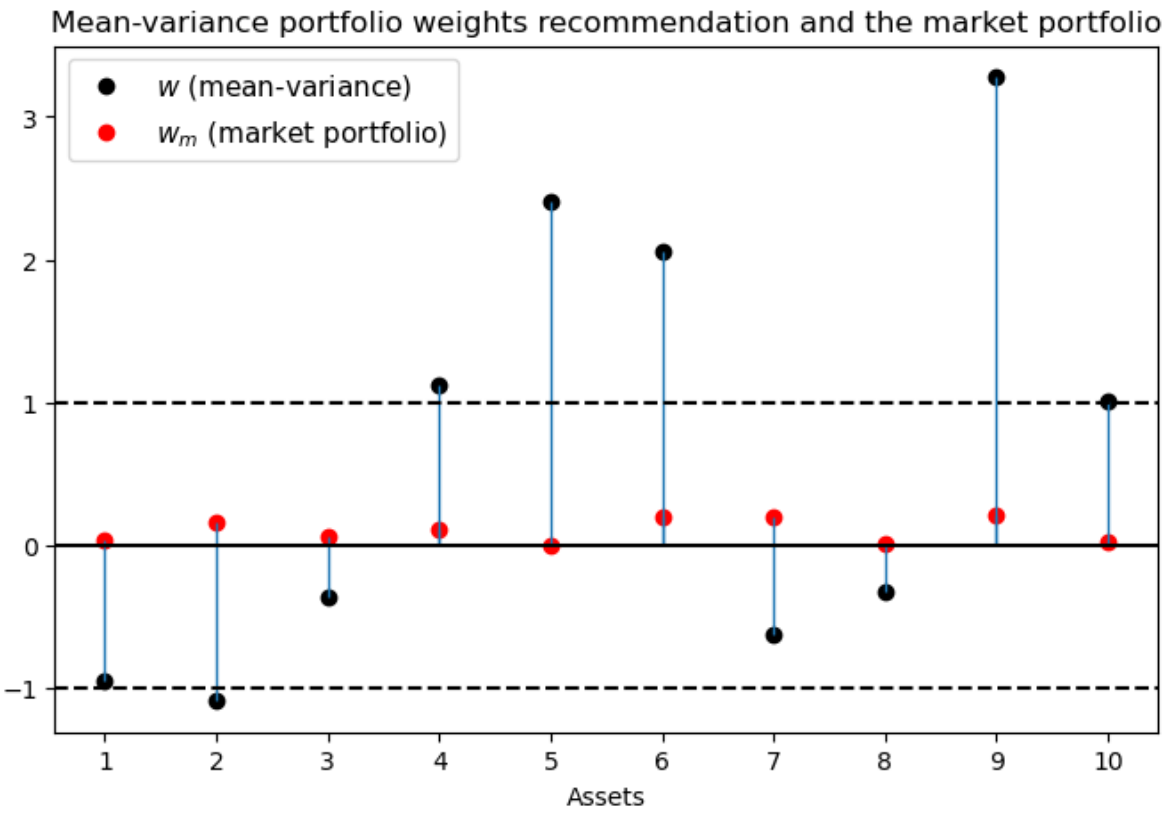
# Risk aversion of market portfolio holder
δ = 1 / np.sqrt(w_m @ Σ @ w_m)

# Generate a sample of excess returns
excess_return = stat.multivariate_normal(μ, Σ)
sample = excess_return.rvs(T)

# Estimate μ and Σ
μ_est = sample.mean(0).reshape(N, 1)
Σ_est = np.cov(sample.T)

w = np.linalg.solve(δ * Σ_est, μ_est)

fig, ax = plt.subplots(figsize=(8, 5))
ax.set_title('Mean-variance portfolio weights recommendation and the market portfolio
 ↪')
ax.plot(np.arange(N)+1, w, 'o', c='k', label='$w$ (mean-variance)')
ax.plot(np.arange(N)+1, w_m, 'o', c='r', label='$w_m$ (market portfolio)')
ax.vlines(np.arange(N)+1, 0, w, lw=1)
ax.vlines(np.arange(N)+1, 0, w_m, lw=1)
ax.axhline(0, c='k')
ax.axhline(-1, c='k', ls='--')
ax.axhline(1, c='k', ls='--')
ax.set_xlabel('Assets')
ax.xaxis.set_ticks(np.arange(1, N+1, 1))
plt.legend(numpoints=1, fontsize=11)
plt.show()
```



Black and Litterman's responded to this situation in the following way:

- They continue to accept (37.2) as a good model for choosing an optimal portfolio w.
- They want to continue to allow the customer to express his or her risk tolerance by setting δ.
- Leaving Σ at its maximum-likelihood value, they push μ away from its maximum-likelihood value in a way designed to make portfolio choices that are more plausible in terms of conforming to what most people actually do.

In particular, given Σ and a plausible value of δ, Black and Litterman reverse engineered a vector μ_{BL} of mean excess returns that makes the w implied by formula (37.2) equal the **actual** market portfolio w_m, so that

$$w_m = (\delta\Sigma)^{-1}\mu_{BL}$$

37.5 Details

Let's define

$$w_m'\mu \equiv (r_m - r_f)$$

as the (scalar) excess return on the market portfolio w_m.

Define

$$\sigma^2 = w_m'\Sigma w_m$$

as the variance of the excess return on the market portfolio w_m.

Define

$$\textbf{SR}_m = \frac{r_m - r_f}{\sigma}$$

as the **Sharpe-ratio** on the market portfolio w_m.

Let δ_m be the value of the risk aversion parameter that induces an investor to hold the market portfolio in light of the optimal portfolio choice rule (37.2).

Evidently, portfolio rule (37.2) then implies that $r_m - r_f = \delta_m \sigma^2$ or

$$\delta_m = \frac{r_m - r_f}{\sigma^2}$$

or

$$\delta_m = \frac{\textbf{SR}_m}{\sigma}$$

Following the Black-Litterman philosophy, our first step will be to back a value of δ_m from

- an estimate of the Sharpe-ratio, and
- our maximum likelihood estimate of σ drawn from our estimates or w_m and Σ

The second key Black-Litterman step is then to use this value of δ together with the maximum likelihood estimate of Σ to deduce a $\mu_{\textbf{BL}}$ that verifies portfolio rule (37.2) at the market portfolio $w = w_m$

$$\mu_m = \delta_m \Sigma w_m$$

The starting point of the Black-Litterman portfolio choice model is thus a pair (δ_m, μ_m) that tells the customer to hold the market portfolio.

```
# Observed mean excess market return
r_m = w_m @ μ_est

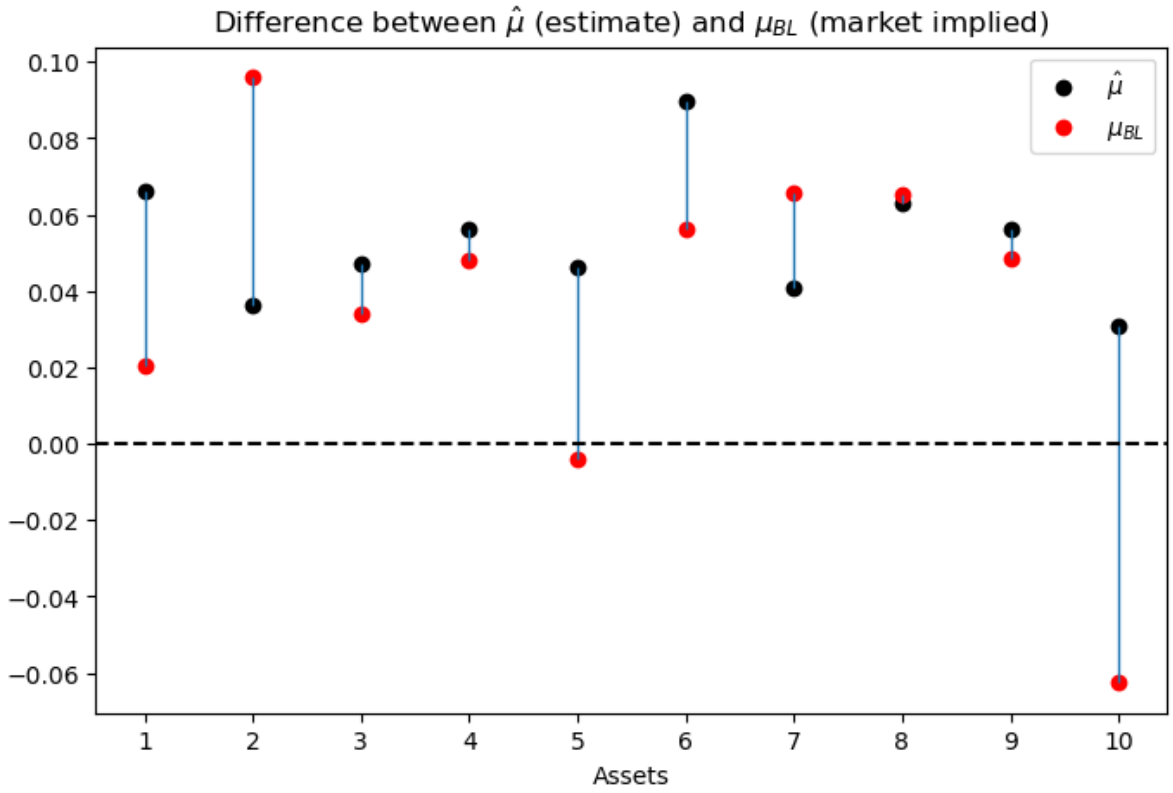
# Estimated variance of the market portfolio
σ_m = w_m @ Σ_est @ w_m

# Sharpe-ratio
sr_m = r_m / np.sqrt(σ_m)

# Risk aversion of market portfolio holder
d_m = r_m / σ_m

# Derive "view" which would induce the market portfolio
μ_m = (d_m * Σ_est @ w_m).reshape(N, 1)

x = np.arange(N) + 1
fig, ax = plt.subplots(figsize=(8, 5))
ax.set_title(r'Difference between $\hat{\mu}$ (estimate) and $\mu_{BL}$ (market␣
 ↪implied)')
ax.plot(x, μ_est, 'o', c='k', label=r'$\hat{\mu}$')
ax.plot(x, μ_m, 'o', c='r', label=r'$\mu_{BL}$')
ax.vlines(x, μ_m, μ_est, lw=1)
ax.axhline(0, c='k', ls='--')
ax.set_xlabel('Assets')
ax.xaxis.set_ticks(np.arange(1, N+1, 1))
plt.legend(numpoints=1)
plt.show()
```



37.6 Adding Views

Black and Litterman start with a baseline customer who asserts that he or she shares the **market's views**, which means that he or she believes that excess returns are governed by

$$\vec{r} - r_f \mathbf{1} \sim \mathcal{N}(\mu_{BL}, \Sigma) \tag{37.3}$$

Black and Litterman would advise that customer to hold the market portfolio of risky securities.

Black and Litterman then imagine a consumer who would like to express a view that differs from the market's.

The consumer wants appropriately to mix his view with the market's before using (37.2) to choose a portfolio.

Suppose that the customer's view is expressed by a hunch that rather than (37.3), excess returns are governed by

$$\vec{r} - r_f \mathbf{1} \sim \mathcal{N}(\hat{\mu}, \tau\Sigma)$$

where $\tau > 0$ is a scalar parameter that determines how the decision maker wants to mix his view $\hat{\mu}$ with the market's view $\mu_{\mathbf{BL}}$.

Black and Litterman would then use a formula like the following one to mix the views $\hat{\mu}$ and $\mu_{\mathbf{BL}}$

$$\tilde{\mu} = (\Sigma^{-1} + (\tau\Sigma)^{-1})^{-1}(\Sigma^{-1}\mu_{BL} + (\tau\Sigma)^{-1}\hat{\mu}) \tag{37.4}$$

Black and Litterman would then advise the customer to hold the portfolio associated with these views implied by rule (37.2):

$$\tilde{w} = (\delta\Sigma)^{-1}\tilde{\mu}$$

This portfolio $\tilde{w}$ will deviate from the portfolio w_{BL} in amounts that depend on the mixing parameter τ.

If $\hat{\mu}$ is the maximum likelihood estimator and τ is chosen heavily to weight this view, then the customer's portfolio will involve big short-long positions.

```
def black_litterman(λ, μ1, μ2, Σ1, Σ2):
    """
    This function calculates the Black-Litterman mixture
    mean excess return and covariance matrix
    """
    Σ1_inv = np.linalg.inv(Σ1)
    Σ2_inv = np.linalg.inv(Σ2)

    μ_tilde = np.linalg.solve(Σ1_inv + λ * Σ2_inv,
                              Σ1_inv @ μ1 + λ * Σ2_inv @ μ2)
    return μ_tilde

τ = 1
μ_tilde = black_litterman(1, μ_m, μ_est, Σ_est, τ * Σ_est)

# The Black-Litterman recommendation for the portfolio weights
w_tilde = np.linalg.solve(δ * Σ_est, μ_tilde)
```

```
def BL_plot(τ):
    μ_tilde = black_litterman(1, μ_m, μ_est, Σ_est, τ * Σ_est)
    w_tilde = np.linalg.solve(δ * Σ_est, μ_tilde)

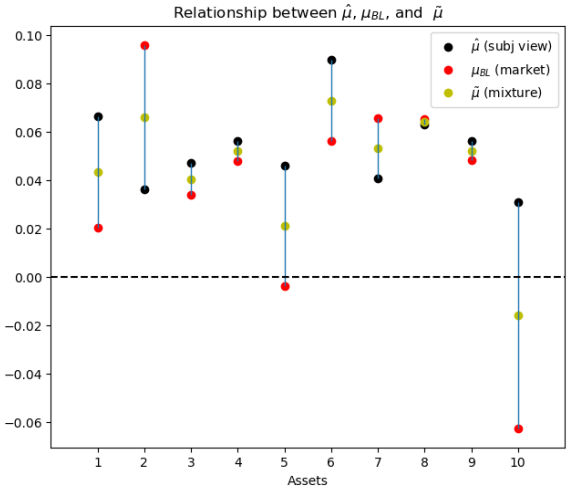
    fig, ax = plt.subplots(1, 2, figsize=(16, 6))
    ax[0].plot(np.arange(N)+1, μ_est, 'o', c='k',
               label=r'$\hat{\mu}$ (subj view)')
    ax[0].plot(np.arange(N)+1, μ_m, 'o', c='r',
               label=r'$\mu_{BL}$ (market)')
    ax[0].plot(np.arange(N)+1, μ_tilde, 'o', c='y',
               label=r'$\tilde{\mu}$ (mixture)')
    ax[0].vlines(np.arange(N)+1, μ_m, μ_est, lw=1)
    ax[0].axhline(0, c='k', ls='--')
    ax[0].set(xlim=(0, N+1), xlabel='Assets',
              title=r'Relationship between $\hat{\mu}$, $\mu_{BL}$, and  $ \tilde{\mu}
↪$')
    ax[0].xaxis.set_ticks(np.arange(1, N+1, 1))
    ax[0].legend(numpoints=1)

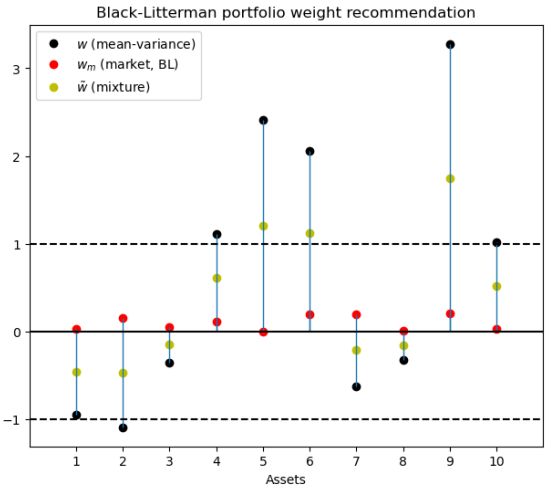
    ax[1].set_title('Black-Litterman portfolio weight recommendation')
    ax[1].plot(np.arange(N)+1, w, 'o', c='k', label=r'$w$ (mean-variance)')
    ax[1].plot(np.arange(N)+1, w_m, 'o', c='r', label=r'$w_{m}$ (market, BL)')
    ax[1].plot(np.arange(N)+1, w_tilde, 'o', c='y',
               label=r'$\tilde{w}$ (mixture)')
    ax[1].vlines(np.arange(N)+1, 0, w, lw=1)
    ax[1].vlines(np.arange(N)+1, 0, w_m, lw=1)
    ax[1].axhline(0, c='k')
    ax[1].axhline(-1, c='k', ls='--')
    ax[1].axhline(1, c='k', ls='--')
    ax[1].set(xlim=(0, N+1), xlabel='Assets',
              title='Black-Litterman portfolio weight recommendation')
    ax[1].xaxis.set_ticks(np.arange(1, N+1, 1))
    ax[1].legend(numpoints=1)
    plt.show()
```

(continues on next page)

(continued from previous page)

```
BL_plot(τ)
```





37.7 Bayesian Interpretation

Consider the following Bayesian interpretation of the Black-Litterman recommendation.

The prior belief over the mean excess returns is consistent with the market portfolio and is given by

$$\mu \sim \mathcal{N}(\mu_{BL}, \Sigma)$$

Given a particular realization of the mean excess returns μ one observes the average excess returns $\hat{\mu}$ on the market according to the distribution

$$\hat{\mu} \mid \mu, \Sigma \sim \mathcal{N}(\mu, \tau\Sigma)$$

where τ is typically small capturing the idea that the variation in the mean is smaller than the variation of the individual random variable.

Given the realized excess returns one should then update the prior over the mean excess returns according to Bayes rule.

The corresponding posterior over mean excess returns is normally distributed with mean

$$(\Sigma^{-1} + (\tau\Sigma)^{-1})^{-1}(\Sigma^{-1}\mu_{BL} + (\tau\Sigma)^{-1}\hat{\mu})$$

The covariance matrix is

$$(\Sigma^{-1} + (\tau\Sigma)^{-1})^{-1}$$

Hence, the Black-Litterman recommendation is consistent with the Bayes update of the prior over the mean excess returns in light of the realized average excess returns on the market.

37.8 Curve Decolletage

Consider two independent "competing" views on the excess market returns

$$\vec{r}_e \sim \mathcal{N}(\mu_{BL}, \Sigma)$$

and

$$\vec{r}_e \sim \mathcal{N}(\hat{\mu}, \tau\Sigma)$$

A special feature of the multivariate normal random variable Z is that its density function depends only on the (Euclidiean) length of its realization z.

Formally, let the k-dimensional random vector be

$$Z \sim \mathcal{N}(\mu, \Sigma)$$

then

$$\bar{Z} \equiv \Sigma(Z - \mu) \sim \mathcal{N}(\mathbf{0}, I)$$

and so the points where the density takes the same value can be described by the ellipse

$$\bar{z} \cdot \bar{z} = (z - \mu)' \Sigma^{-1} (z - \mu) = \bar{d} \tag{37.5}$$

where $\bar{d} \in \mathbb{R}_+$ denotes the (transformation) of a particular density value.

The curves defined by equation (37.5) can be labeled as iso-likelihood ellipses

Remark: More generally there is a class of density functions that possesses this feature, i.e.

$$\exists g : \mathbb{R}_+ \mapsto \mathbb{R}_+ \quad \text{and} \quad c \geq 0, \quad \text{s.t. the density} \quad f \quad \text{of} \quad Z \quad \text{has the form} \quad f(z) = cg(z \cdot z)$$

This property is called **spherical symmetry** (see p 81. in Leamer (1978) [Leamer, 1978]).

In our specific example, we can use the pair $(\bar{d}_1, \bar{d}_2)$ as being two "likelihood" values for which the corresponding iso-likelihood ellipses in the excess return space are given by

$$\begin{aligned}(\vec{r}_e - \mu_{BL})' \Sigma^{-1} (\vec{r}_e - \mu_{BL}) &= \bar{d}_1 \\ (\vec{r}_e - \hat{\mu})' \left(\tau\Sigma\right)^{-1} (\vec{r}_e - \hat{\mu}) &= \bar{d}_2\end{aligned}$$

Notice that for particular $\bar{d}_1$ and $\bar{d}_2$ values the two ellipses have a tangency point.

These tangency points, indexed by the pairs $(\bar{d}_1, \bar{d}_2)$, characterize points $\vec{r}_e$ from which there exists no deviation where one can increase the likelihood of one view without decreasing the likelihood of the other view.

The pairs $(\bar{d}_1, \bar{d}_2)$ for which there is such a point outlines a curve in the excess return space. This curve is reminiscent of the Pareto curve in an Edgeworth-box setting.

Dickey (1975) [Dickey, 1975] calls it a *curve decolletage*.

Leamer (1978) [Leamer, 1978] calls it an *information contract curve* and describes it by the following program: maximize the likelihood of one view, say the Black-Litterman recommendation while keeping the likelihood of the other view at least at a prespecified constant $\bar{d}_2$

$$\begin{aligned}\bar{d}_1(\bar{d}_2) \equiv \max_{\vec{r}_e} \; & (\vec{r}_e - \mu_{BL})' \Sigma^{-1} (\vec{r}_e - \mu_{BL}) \\ \text{subject to} \quad & (\vec{r}_e - \hat{\mu})' (\tau\Sigma)^{-1} (\vec{r}_e - \hat{\mu}) \geq \bar{d}_2\end{aligned}$$

Denoting the multiplier on the constraint by λ, the first-order condition is

$$2(\vec{r}_e - \mu_{BL})'\Sigma^{-1} + \lambda 2(\vec{r}_e - \hat{\mu})'(\tau\Sigma)^{-1} = \mathbf{0}$$

which defines the *information contract curve* between μ_{BL} and $\hat{\mu}$

$$\vec{r}_e = (\Sigma^{-1} + \lambda(\tau\Sigma)^{-1})^{-1}(\Sigma^{-1}\mu_{BL} + \lambda(\tau\Sigma)^{-1}\hat{\mu}) \tag{37.6}$$

Note that if $\lambda = 1$, (37.6) is equivalent with (37.4) and it identifies one point on the information contract curve.

Furthermore, because λ is a function of the minimum likelihood $\bar{d}_2$ on the RHS of the constraint, by varying $\bar{d}_2$ (or λ), we can trace out the whole curve as the figure below illustrates.

```
np.random.seed(1987102)

N = 2                                                  # Number of assets
T = 200                                                # Sample size
τ = 0.8

# Random market portfolio (sum is normalized to 1)
w_m = np.random.rand(N)
w_m = w_m / (w_m.sum())

μ = (np.random.randn(N) + 5) / 100
S = np.random.randn(N, N)
V = S @ S.T
Σ = V * (w_m @ μ)**2 / (w_m @ V @ w_m)

excess_return = stat.multivariate_normal(μ, Σ)
sample = excess_return.rvs(T)

μ_est = sample.mean(0).reshape(N, 1)
Σ_est = np.cov(sample.T)

σ_m = w_m @ Σ_est @ w_m
d_m = (w_m @ μ_est) / σ_m
μ_m = (d_m * Σ_est @ w_m).reshape(N, 1)

N_r1, N_r2 = 100, 100
r1 = np.linspace(-0.04, .1, N_r1)
r2 = np.linspace(-0.02, .15, N_r2)

λ_grid = np.linspace(.001, 20, 100)
curve = np.asarray([black_litterman(λ, μ_m, μ_est, Σ_est,
                                    τ * Σ_est).flatten() for λ in λ_grid])

λ = 1
```

```
def decolletage(λ):
    dist_r_BL = stat.multivariate_normal(μ_m.squeeze(), Σ_est)
    dist_r_hat = stat.multivariate_normal(μ_est.squeeze(), τ * Σ_est)

    X, Y = np.meshgrid(r1, r2)
    XY = np.stack((X, Y), axis=-1)
    Z_BL = dist_r_BL.pdf(XY)
    Z_hat = dist_r_hat.pdf(XY)
```

(continues on next page)

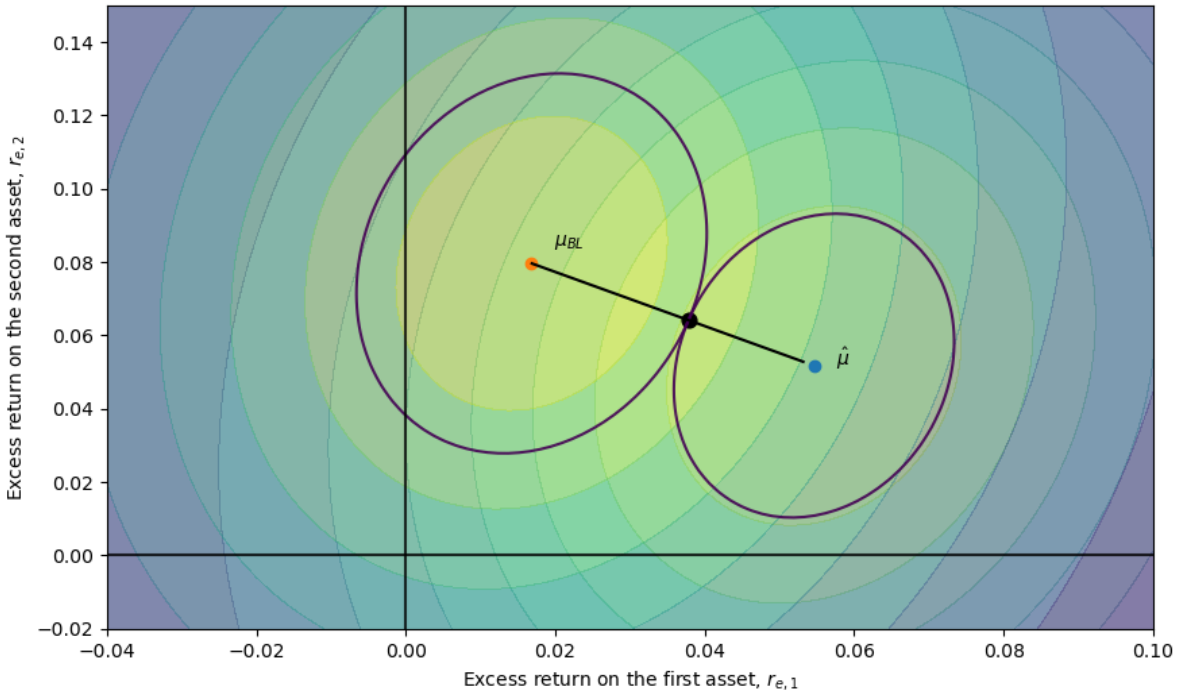
(continued from previous page)

```
    μ_tilde = black_litterman(λ, μ_m, μ_est, Σ_est, τ * Σ_est).flatten()

    fig, ax = plt.subplots(figsize=(10, 6))
    ax.contourf(X, Y, Z_hat, cmap='viridis', alpha =.4)
    ax.contourf(X, Y, Z_BL, cmap='viridis', alpha =.4)
    ax.contour(X, Y, Z_BL, [dist_r_BL.pdf(μ_tilde)], cmap='viridis', alpha=.9)
    ax.contour(X, Y, Z_hat, [dist_r_hat.pdf(μ_tilde)], cmap='viridis', alpha=.9)
    ax.scatter(μ_est[0], μ_est[1])
    ax.scatter(μ_m[0], μ_m[1])
    ax.scatter(μ_tilde[0], μ_tilde[1], c='k', s=20*3)

    ax.plot(curve[:, 0], curve[:, 1], c='k')
    ax.axhline(0, c='k', alpha=.8)
    ax.axvline(0, c='k', alpha=.8)
    ax.set_xlabel(r'Excess return on the first asset, $r_{e, 1}$')
    ax.set_ylabel(r'Excess return on the second asset, $r_{e, 2}$')
    ax.text(μ_est[0] + 0.003, μ_est[1], r'$\hat{\mu}$')
    ax.text(μ_m[0] + 0.003, μ_m[1] + 0.005, r'$\mu_{BL}$')
    plt.show()

decolletage(λ)
```



Note that the line that connects the two points $\hat{\mu}$ and μ_{BL} is linear, which comes from the fact that the covariance matrices of the two competing distributions (views) are proportional to each other.

To illustrate the fact that this is not necessarily the case, consider another example using the same parameter values, except that the "second view" constituting the constraint has covariance matrix τI instead of $\tau\Sigma$.

This leads to the following figure, on which the curve connecting $\hat{\mu}$ and μ_{BL} are bending

```
λ_grid = np.linspace(.001, 20000, 1000)
```

(continues on next page)

(continued from previous page)

```
curve = np.asarray([black_litterman(λ, μ_m, μ_est, Σ_est,
                                    τ * np.eye(N)).flatten() for λ in λ_grid])
λ = 200
```

```
def decolletage(λ):
    dist_r_BL = stat.multivariate_normal(μ_m.squeeze(), Σ_est)
    dist_r_hat = stat.multivariate_normal(μ_est.squeeze(), τ * np.eye(N))

    X, Y = np.meshgrid(r1, r2)
    XY = np.stack((X, Y), axis=-1)
    Z_BL = dist_r_BL.pdf(XY)
    Z_hat = dist_r_hat.pdf(XY)

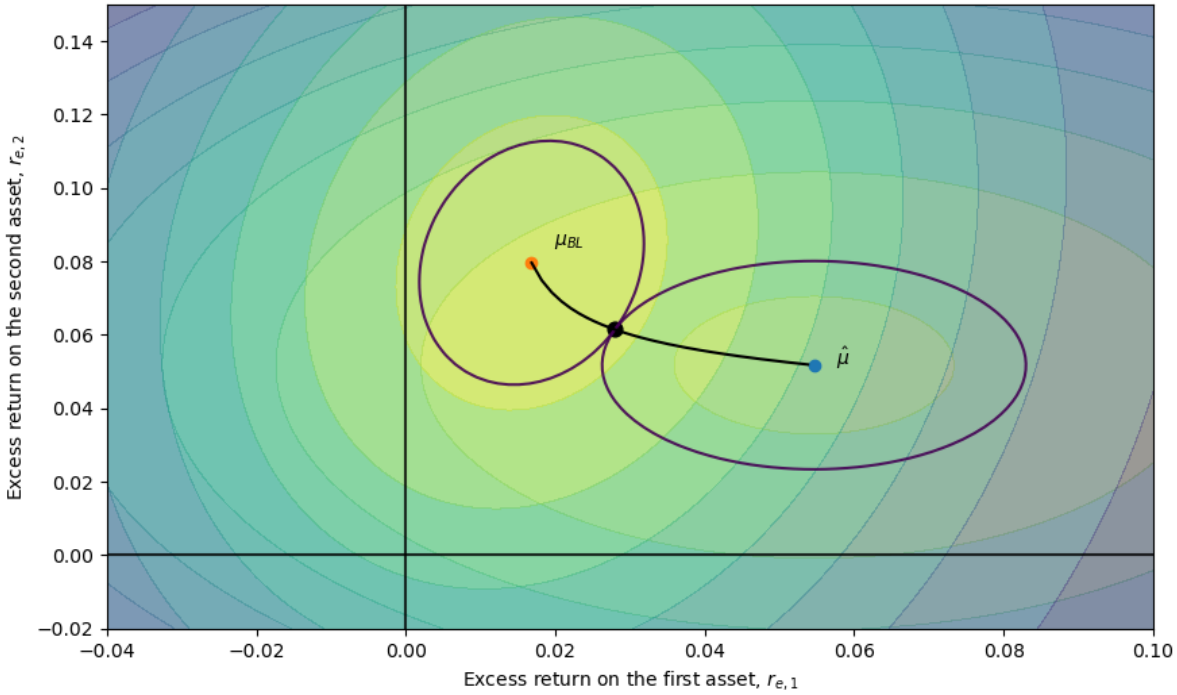
    μ_tilde = black_litterman(λ, μ_m, μ_est, Σ_est, τ * np.eye(N)).flatten()

    fig, ax = plt.subplots(figsize=(10, 6))
    ax.contourf(X, Y, Z_hat, cmap='viridis', alpha=.4)
    ax.contourf(X, Y, Z_BL, cmap='viridis', alpha=.4)
    ax.contour(X, Y, Z_BL, [dist_r_BL.pdf(μ_tilde)], cmap='viridis', alpha=.9)
    ax.contour(X, Y, Z_hat, [dist_r_hat.pdf(μ_tilde)], cmap='viridis', alpha=.9)
    ax.scatter(μ_est[0], μ_est[1])
    ax.scatter(μ_m[0], μ_m[1])

    ax.scatter(μ_tilde[0], μ_tilde[1], c='k', s=20*3)

    ax.plot(curve[:, 0], curve[:, 1], c='k')
    ax.axhline(0, c='k', alpha=.8)
    ax.axvline(0, c='k', alpha=.8)
    ax.set_xlabel(r'Excess return on the first asset, $r_{e, 1}$')
    ax.set_ylabel(r'Excess return on the second asset, $r_{e, 2}$')
    ax.text(μ_est[0] + 0.003, μ_est[1], r'$\hat{\mu}$')
    ax.text(μ_m[0] + 0.003, μ_m[1] + 0.005, r'$\mu_{BL}$')
    plt.show()

decolletage(λ)
```



37.9 Black-Litterman Recommendation as Regularization

First, consider the OLS regression

$$\min_{\beta} \|X\beta - y\|^2$$

which yields the solution

$$\hat{\beta}_{OLS} = (X'X)^{-1}X'y$$

A common performance measure of estimators is the *mean squared error (MSE)*.

An estimator is "good" if its MSE is relatively small. Suppose that β_0 is the "true" value of the coefficient, then the MSE of the OLS estimator is

$$\text{mse}(\hat{\beta}_{OLS}, \beta_0) := \mathbb{E}\|\hat{\beta}_{OLS} - \beta_0\|^2 = \underbrace{\mathbb{E}\|\hat{\beta}_{OLS} - \mathbb{E}\beta_{OLS}\|^2}_{\text{variance}} + \underbrace{\|\mathbb{E}\hat{\beta}_{OLS} - \beta_0\|^2}_{\text{bias}}$$

From this decomposition, one can see that in order for the MSE to be small, both the bias and the variance terms must be small.

For example, consider the case when X is a T-vector of ones (where T is the sample size), so $\hat{\beta}_{OLS}$ is simply the sample average, while $\beta_0 \in \mathbb{R}$ is defined by the true mean of y.

In this example the MSE is

$$\text{mse}(\hat{\beta}_{OLS}, \beta_0) = \underbrace{\frac{1}{T^2}\mathbb{E}\left(\sum_{t=1}^{T}(y_t - \beta_0)\right)^2}_{\text{variance}} + \underbrace{0}_{\text{bias}}$$

However, because there is a trade-off between the estimator's bias and variance, there are cases when by permitting a small bias we can substantially reduce the variance so overall the MSE gets smaller.

A typical scenario when this proves to be useful is when the number of coefficients to be estimated is large relative to the sample size.

In these cases, one approach to handle the bias-variance trade-off is the so called *Tikhonov regularization*.

A general form with regularization matrix Γ can be written as

$$\min_{\beta} \left\{ \|X\beta - y\|^2 + \|\Gamma(\beta - \tilde{\beta})\|^2 \right\}$$

which yields the solution

$$\hat{\beta}_{Reg} = (X'X + \Gamma'\Gamma)^{-1}(X'y + \Gamma'\Gamma\tilde{\beta})$$

Substituting the value of $\hat{\beta}_{OLS}$ yields

$$\hat{\beta}_{Reg} = (X'X + \Gamma'\Gamma)^{-1}(X'X\hat{\beta}_{OLS} + \Gamma'\Gamma\tilde{\beta})$$

Often, the regularization matrix takes the form $\Gamma = \lambda I$ with $\lambda > 0$ and $\tilde{\beta} = \mathbf{0}$.

Then the Tikhonov regularization is equivalent to what is called *ridge regression* in statistics.

To illustrate how this estimator addresses the bias-variance trade-off, we compute the MSE of the ridge estimator

$$\text{mse}(\hat{\beta}_{\text{ridge}}, \beta_0) = \underbrace{\frac{1}{(T+\lambda)^2} \mathbb{E}\left(\sum_{t=1}^{T} (y_t - \beta_0) \right)^2}_{\text{variance}} + \underbrace{\left(\frac{\lambda}{T+\lambda} \right)^2 \beta_0^2}_{\text{bias}}$$

The ridge regression shrinks the coefficients of the estimated vector towards zero relative to the OLS estimates thus reducing the variance term at the cost of introducing a "small" bias.

However, there is nothing special about the zero vector.

When $\tilde{\beta} \neq \mathbf{0}$ shrinkage occurs in the direction of $\tilde{\beta}$.

Now, we can give a regularization interpretation of the Black-Litterman portfolio recommendation.

To this end, first simplify the equation (37.4) that characterizes the Black-Litterman recommendation

$$\begin{aligned}
\tilde{\mu} &= (\Sigma^{-1} + (\tau\Sigma)^{-1})^{-1}(\Sigma^{-1}\mu_{BL} + (\tau\Sigma)^{-1}\hat{\mu}) \\
&= (1 + \tau^{-1})^{-1}\Sigma\Sigma^{-1}(\mu_{BL} + \tau^{-1}\hat{\mu}) \\
&= (1 + \tau^{-1})^{-1}(\mu_{BL} + \tau^{-1}\hat{\mu})
\end{aligned}$$

In our case, $\hat{\mu}$ is the estimated mean excess returns of securities. This could be written as a vector autoregression where

- y is the stacked vector of observed excess returns of size $(NT \times 1)$ - N securities and T observations.
- $X = \sqrt{T^{-1}}(I_N \otimes \iota_T)$ where I_N is the identity matrix and ι_T is a column vector of ones.

Correspondingly, the OLS regression of y on X would yield the mean excess returns as coefficients.

With $\Gamma = \sqrt{\tau T^{-1}}(I_N \otimes \iota_T)$ we can write the regularized version of the mean excess return estimation

$$\begin{aligned}
\hat{\beta}_{Reg} &= (X'X + \Gamma'\Gamma)^{-1}(X'X\hat{\beta}_{OLS} + \Gamma'\Gamma\tilde{\beta}) \\
&= (1 + \tau)^{-1}X'X(X'X)^{-1}(\hat{\beta}_{OLS} + \tau\tilde{\beta}) \\
&= (1 + \tau)^{-1}(\hat{\beta}_{OLS} + \tau\tilde{\beta}) \\
&= (1 + \tau^{-1})^{-1}(\tau^{-1}\hat{\beta}_{OLS} + \tilde{\beta})
\end{aligned}$$

Given that $\hat{\beta}_{OLS} = \hat{\mu}$ and $\tilde{\beta} = \mu_{BL}$ in the Black-Litterman model, we have the following interpretation of the model's recommendation.

The estimated (personal) view of the mean excess returns, $\widehat{\mu}$ that would lead to extreme short-long positions are "shrunk" towards the conservative market view, μ_{BL}, that leads to the more conservative market portfolio.

So the Black-Litterman procedure results in a recommendation that is a compromise between the conservative market portfolio and the more extreme portfolio that is implied by estimated "personal" views.

37.10 A Robust Control Operator

The Black-Litterman approach is partly inspired by the econometric insight that it is easier to estimate covariances of excess returns than the means.

That is what gave Black and Litterman license to adjust investors' perception of mean excess returns while not tampering with the covariance matrix of excess returns.

The robust control theory is another approach that also hinges on adjusting mean excess returns but not covariances.

Associated with a robust control problem is what Hansen and Sargent [Hansen and Sargent, 2001], [Hansen and Sargent, 2008] call a T operator.

Let's define the T operator as it applies to the problem at hand.

Let x be an $n \times 1$ Gaussian random vector with mean vector μ and covariance matrix $\Sigma = CC'$. This means that x can be represented as

$$x = \mu + C\epsilon$$

where $\epsilon \sim \mathcal{N}(0, I)$.

Let $\phi(\epsilon)$ denote the associated standardized Gaussian density.

Let $m(\epsilon, \mu)$ be a **likelihood ratio**, meaning that it satisfies

- $m(\epsilon, \mu) > 0$
- $\int m(\epsilon, \mu)\phi(\epsilon)d\epsilon = 1$

That is, $m(\epsilon, \mu)$ is a non-negative random variable with mean 1.

Multiplying $\phi(\epsilon)$ by the likelihood ratio $m(\epsilon, \mu)$ produces a distorted distribution for ϵ, namely

$$\tilde{\phi}(\epsilon) = m(\epsilon, \mu)\phi(\epsilon)$$

The next concept that we need is the **entropy** of the distorted distribution $\tilde{\phi}$ with respect to ϕ.

Entropy is defined as

$$\text{ent} = \int \log m(\epsilon, \mu) m(\epsilon, \mu)\phi(\epsilon)d\epsilon$$

or

$$\text{ent} = \int \log m(\epsilon, \mu)\tilde{\phi}(\epsilon)d\epsilon$$

That is, relative entropy is the expected value of the likelihood ratio m where the expectation is taken with respect to the twisted density $\tilde{\phi}$.

Relative entropy is non-negative. It is a measure of the discrepancy between two probability distributions.

As such, it plays an important role in governing the behavior of statistical tests designed to discriminate one probability distribution from another.

We are ready to define the T operator.

Let $V(x)$ be a value function.

Define

$$\begin{aligned}\mathsf{T}\left(V(x)\right) &= \min_{m(\epsilon,\mu)} \int m(\epsilon,\mu)[V(\mu + C\epsilon) + \theta \log m(\epsilon,\mu)]\phi(\epsilon)d\epsilon \\ &= -\log\theta \int \exp\left(\frac{-V(\mu + C\epsilon)}{\theta}\right)\phi(\epsilon)d\epsilon\end{aligned}$$

This asserts that T is an indirect utility function for a minimization problem in which an **adversary** chooses a distorted probability distribution $\tilde{\phi}$ to lower expected utility, subject to a penalty term that gets bigger the larger is relative entropy.

Here the penalty parameter

$$\theta \in [\underline{\theta}, +\infty]$$

is a robustness parameter when it is $+\infty$, there is no scope for the minimizing agent to distort the distribution, so no robustness to alternative distributions is acquired.

As θ is lowered, more robustness is achieved.

Note: The T operator is sometimes called a *risk-sensitivity* operator.

We shall apply T to the special case of a linear value function $w'(\vec{r} - r_f 1)$ where $\vec{r} - r_f 1 \sim \mathcal{N}(\mu, \Sigma)$ or $\vec{r} - r_f \mathbf{1} = \mu + C\epsilon$ and $\epsilon \sim \mathcal{N}(0, I)$.

The associated worst-case distribution of ϵ is Gaussian with mean $v = -\theta^{-1}C'w$ and covariance matrix I

(When the value function is affine, the worst-case distribution distorts the mean vector of ϵ but not the covariance matrix of ϵ).

For utility function argument $w'(\vec{r} - r_f 1)$

$$\mathsf{T}(\vec{r} - r_f \mathbf{1}) = w'\mu + \zeta - \frac{1}{2\theta} w'\Sigma w$$

and entropy is

$$\frac{v'v}{2} = \frac{1}{2\theta^2} w'CC'w$$

37.11 A Robust Mean-Variance Portfolio Model

According to criterion (37.1), the mean-variance portfolio choice problem chooses w to maximize

$$E[w(\vec{r} - r_f \mathbf{1})]] - \text{var}[w(\vec{r} - r_f \mathbf{1})]$$

which equals

$$w'\mu - \frac{\delta}{2} w'\Sigma w$$

A robust decision maker can be modeled as replacing the mean return $E[w(\vec{r} - r_f \mathbf{1})]$ with the risk-sensitive criterion

$$\mathsf{T}[w(\vec{r} - r_f \mathbf{1})] = w'\mu - \frac{1}{2\theta} w'\Sigma w$$

that comes from replacing the mean μ of $\vec{r} - r_f\mathbf{1}$ with the worst-case mean

$$\mu - \theta^{-1}\Sigma w$$

Notice how the worst-case mean vector depends on the portfolio w.

The operator T is the indirect utility function that emerges from solving a problem in which an agent who chooses probabilities does so in order to minimize the expected utility of a maximizing agent (in our case, the maximizing agent chooses portfolio weights w).

The robust version of the mean-variance portfolio choice problem is then to choose a portfolio w that maximizes

$$\mathsf{T}[w(\vec{r} - r_f\mathbf{1})] - \frac{\delta}{2}w'\Sigma w$$

or

$$w'(\mu - \theta^{-1}\Sigma w) - \frac{\delta}{2}w'\Sigma w \tag{37.7}$$

The minimizer of (37.7) is

$$w_{\text{rob}} = \frac{1}{\delta + \gamma}\Sigma^{-1}\mu$$

where $\gamma \equiv \theta^{-1}$ is sometimes called the risk-sensitivity parameter.

An increase in the risk-sensitivity parameter γ shrinks the portfolio weights toward zero in the same way that an increase in risk aversion does.

37.12 Appendix

We want to illustrate the "folk theorem" that with high or moderate frequency data, it is more difficult to estimate means than variances.

In order to operationalize this statement, we take two analog estimators:

- sample average: $\bar{X}_N = \frac{1}{N}\sum_{i=1}^N X_i$
- sample variance: $S_N = \frac{1}{N-1}\sum_{t=1}^N (X_i - \bar{X}_N)^2$

to estimate the unconditional mean and unconditional variance of the random variable X, respectively.

To measure the "difficulty of estimation", we use *mean squared error* (MSE), that is the average squared difference between the estimator and the true value.

Assuming that the process $\{X_i\}$is ergodic, both analog estimators are known to converge to their true values as the sample size N goes to infinity.

More precisely for all $\varepsilon > 0$

$$\lim_{N\to\infty} \; P\left\{\left|\bar{X}_N - \mathbb{E}X\right| > \varepsilon\right\} = 0$$

and

$$\lim_{N\to\infty} \; P\left\{|S_N - \mathbb{V}X| > \varepsilon\right\} = 0$$

A necessary condition for these convergence results is that the associated MSEs vanish as N goes to infinity, or in other words,

$$\text{MSE}(\bar{X}_N, \mathbb{E}X) = o(1) \qquad \text{and} \qquad \text{MSE}(S_N, \mathbb{V}X) = o(1)$$

Even if the MSEs converge to zero, the associated rates might be different. Looking at the limit of the *relative MSE* (as the sample size grows to infinity)

$$\frac{\text{MSE}(S_N, \mathbb{V}X)}{\text{MSE}(\bar{X}_N, \mathbb{E}X)} = \frac{o(1)}{o(1)} \underset{N\to\infty}{\to} B$$

can inform us about the relative (asymptotic) rates.

We will show that in general, with dependent data, the limit B depends on the sampling frequency.

In particular, we find that the rate of convergence of the variance estimator is less sensitive to increased sampling frequency than the rate of convergence of the mean estimator.

Hence, we can expect the relative asymptotic rate, B, to get smaller with higher frequency data, illustrating that "it is more difficult to estimate means than variances".

That is, we need significantly more data to obtain a given precision of the mean estimate than for our variance estimate.

37.13 Special Case – IID Sample

We start our analysis with the benchmark case of IID data.

Consider a sample of size N generated by the following IID process,

$$X_i \sim \mathcal{N}(\mu, \sigma^2)$$

Taking $\bar{X}_N$ to estimate the mean, the MSE is

$$\text{MSE}(\bar{X}_N, \mu) = \frac{\sigma^2}{N}$$

Taking S_N to estimate the variance, the MSE is

$$\text{MSE}(S_N, \sigma^2) = \frac{2\sigma^4}{N-1}$$

Both estimators are unbiased and hence the MSEs reflect the corresponding variances of the estimators.

Furthermore, both MSEs are $o(1)$ with a (multiplicative) factor of difference in their rates of convergence:

$$\frac{\text{MSE}(S_N, \sigma^2)}{\text{MSE}(\bar{X}_N, \mu)} = \frac{N2\sigma^2}{N-1} \quad \underset{N\to\infty}{\to} \quad 2\sigma^2$$

We are interested in how this (asymptotic) relative rate of convergence changes as increasing sampling frequency puts dependence into the data.

37.14 Dependence and Sampling Frequency

To investigate how sampling frequency affects relative rates of convergence, we assume that the data are generated by a mean-reverting continuous time process of the form

$$dX_t = -\kappa(X_t - \mu)dt + \sigma dW_t$$

where $\mu$is the unconditional mean, $\kappa > 0$ is a persistence parameter, and $\{W_t\}$ is a standardized Brownian motion.

Observations arising from this system in particular discrete periods $\mathcal{T}(h) \equiv \{nh : n \in \mathbb{Z}\}$with$h > 0$ can be described by the following process

$$X_{t+1} = (1 - \exp(-\kappa h))\mu + \exp(-\kappa h)X_t + \epsilon_{t,h}$$

where

$$\epsilon_{t,h} \sim \mathcal{N}(0, \Sigma_h) \quad \text{with} \quad \Sigma_h = \frac{\sigma^2(1 - \exp(-2\kappa h))}{2\kappa}$$

We call h the *frequency* parameter, whereas n represents the number of *lags* between observations.

Hence, the effective distance between two observations X_t and X_{t+n} in the discrete time notation is equal to $h \cdot n$ in terms of the underlying continuous time process.

Straightforward calculations show that the autocorrelation function for the stochastic process $\{X_t\}_{t\in\mathcal{T}(h)}$ is

$$\Gamma_h(n) \equiv \text{corr}(X_{t+hn}, X_t) = \exp(-\kappa hn)$$

and the auto-covariance function is

$$\gamma_h(n) \equiv \text{cov}(X_{t+hn}, X_t) = \frac{\exp(-\kappa hn)\sigma^2}{2\kappa}.$$

It follows that if $n = 0$, the unconditional variance is given by $\gamma_h(0) = \frac{\sigma^2}{2\kappa}$ irrespective of the sampling frequency.

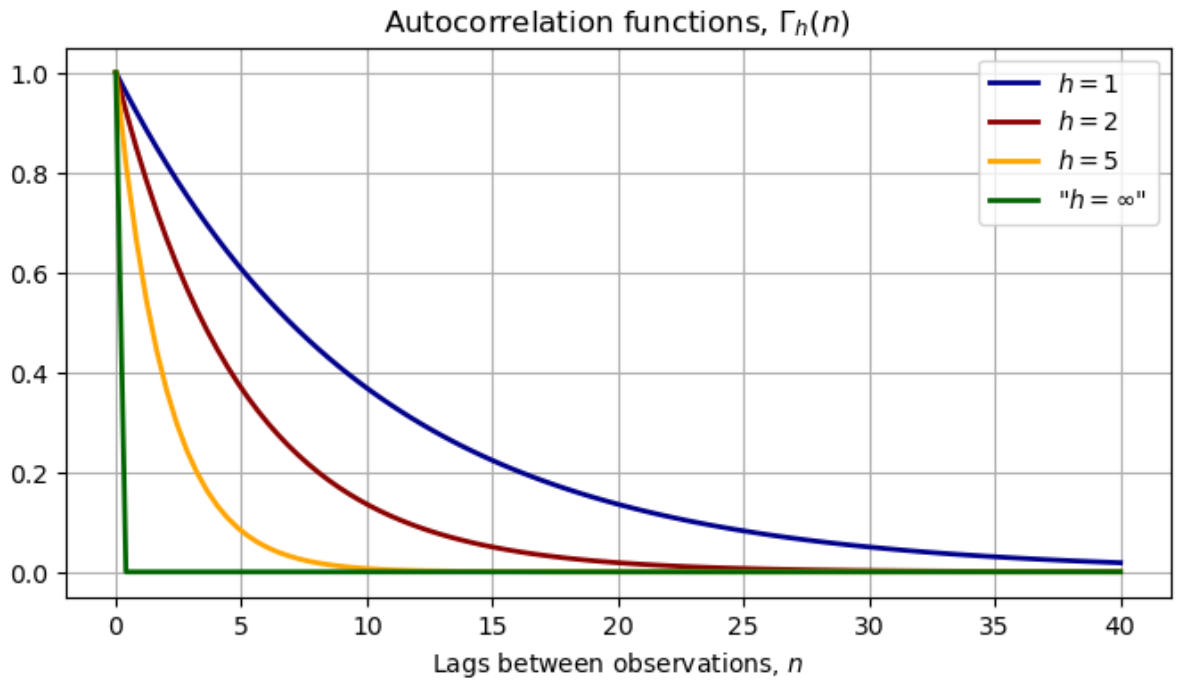
The following figure illustrates how the dependence between the observations is related to the sampling frequency

- For any given h, the autocorrelation converges to zero as we increase the distance – n– between the observations. This represents the "weak dependence" of the X process.
- Moreover, for a fixed lag length, n, the dependence vanishes as the sampling frequency goes to infinity. In fact, letting h go to ∞ gives back the case of IID data.

```
μ = .0
κ = .1
σ = .5
var_uncond = σ**2 / (2 * κ)

n_grid = np.linspace(0, 40, 100)
autocorr_h1 = np.exp(-κ * n_grid * 1)
autocorr_h2 = np.exp(-κ * n_grid * 2)
autocorr_h5 = np.exp(-κ * n_grid * 5)
autocorr_h1000 = np.exp(-κ * n_grid * 1e8)

fig, ax = plt.subplots(figsize=(8, 4))
ax.plot(n_grid, autocorr_h1, label=r'$h=1$', c='darkblue', lw=2)
ax.plot(n_grid, autocorr_h2, label=r'$h=2$', c='darkred', lw=2)
ax.plot(n_grid, autocorr_h5, label=r'$h=5$', c='orange', lw=2)
ax.plot(n_grid, autocorr_h1000, label=r'"$h=\infty$"', c='darkgreen', lw=2)
ax.legend()
ax.grid()
ax.set(title=r'Autocorrelation functions, $\Gamma_h(n)$',
       xlabel=r'Lags between observations, $n$')
plt.show()
```



37.15 Frequency and the Mean Estimator

Consider again the AR(1) process generated by discrete sampling with frequency h. Assume that we have a sample of size N and we would like to estimate the unconditional mean – in our case the true mean is μ.

Again, the sample average is an unbiased estimator of the unconditional mean

$$\mathbb{E}[\bar{X}_N] = \frac{1}{N}\sum_{i=1}^{N}\mathbb{E}[X_i] = \mathbb{E}[X_0] = \mu$$

The variance of the sample mean is given by

$$\begin{aligned}
\mathbb{V}\left(\bar{X}_N\right) &= \mathbb{V}\left(\frac{1}{N}\sum_{i=1}^{N} X_i\right) \\
&= \frac{1}{N^2}\left(\sum_{i=1}^{N}\mathbb{V}(X_i) + 2\sum_{i=1}^{N-1}\sum_{s=i+1}^{N}\operatorname{cov}(X_i, X_s)\right) \\
&= \frac{1}{N^2}\left(N\gamma(0) + 2\sum_{i=1}^{N-1} i\cdot\gamma\left(h\cdot(N-i)\right)\right) \\
&= \frac{1}{N^2}\left(N\frac{\sigma^2}{2\kappa} + 2\sum_{i=1}^{N-1} i\cdot\exp(-\kappa h(N-i))\frac{\sigma^2}{2\kappa}\right)
\end{aligned}$$

It is explicit in the above equation that time dependence in the data inflates the variance of the mean estimator through the covariance terms.

Moreover, as we can see, a higher sampling frequency—smaller h—makes all the covariance terms larger, everything else being fixed.

This implies a relatively slower rate of convergence of the sample average for high-frequency data.

Intuitively, stronger dependence across observations for high-frequency data reduces the "information content" of each observation relative to the IID case.

We can upper bound the variance term in the following way

$$\begin{aligned}\mathbb{V}(\bar{X}_N) &= \frac{1}{N^2}\left(N\sigma^2 + 2\sum_{i=1}^{N-1} i \cdot \exp(-\kappa h(N-i))\sigma^2\right) \\ &\leq \frac{\sigma^2}{2\kappa N}\left(1 + 2\sum_{i=1}^{N-1} \cdot \exp(-\kappa h(i))\right) \\ &= \underbrace{\frac{\sigma^2}{2\kappa N}}_{\text{IID case}}\left(1 + 2\frac{1-\exp(-\kappa h)^{N-1}}{1-\exp(-\kappa h)}\right)\end{aligned}$$

Asymptotically, the term $\exp(-\kappa h)^{N-1}$ vanishes and the dependence in the data inflates the benchmark IID variance by a factor of

$$\left(1 + 2\frac{1}{1-\exp(-\kappa h)}\right)$$

This long run factor is larger the higher is the frequency (the smaller is h).

Therefore, we expect the asymptotic relative MSEs, B, to change with time-dependent data. We just saw that the mean estimator's rate is roughly changing by a factor of

$$\left(1 + 2\frac{1}{1-\exp(-\kappa h)}\right)$$

Unfortunately, the variance estimator's MSE is harder to derive.

Nonetheless, we can approximate it by using (large sample) simulations, thus getting an idea about how the asymptotic relative MSEs changes in the sampling frequency h relative to the IID case that we compute in closed form.

```
@jit
def sample_generator(h, N, M):
    ϕ = (1 - np.exp(-κ * h)) * μ
    ρ = np.exp(-κ * h)
    s = σ**2 * (1 - np.exp(-2 * κ * h)) / (2 * κ)

    mean_uncond = μ
    std_uncond = np.sqrt(σ**2 / (2 * κ))

    ε_path = np.random.normal(0, np.sqrt(s), (M, N))

    y_path = np.zeros((M, N + 1))
    y_path[:, 0] = np.random.normal(mean_uncond, std_uncond, M)

    for i in range(N):
        y_path[:, i + 1] = ϕ + ρ * y_path[:, i] + ε_path[:, i]

    return y_path
```

```
# Generate large sample for different frequencies
N_app, M_app = 1000, 30000        # Sample size, number of simulations
h_grid = np.linspace(.1, 80, 30)

var_est_store = []
```

(continues on next page)

(continued from previous page)

```
mean_est_store = []
labels = []

for h in h_grid:
    labels.append(h)
    sample = sample_generator(h, N_app, M_app)
    mean_est_store.append(np.mean(sample, 1))
    var_est_store.append(np.var(sample, 1))

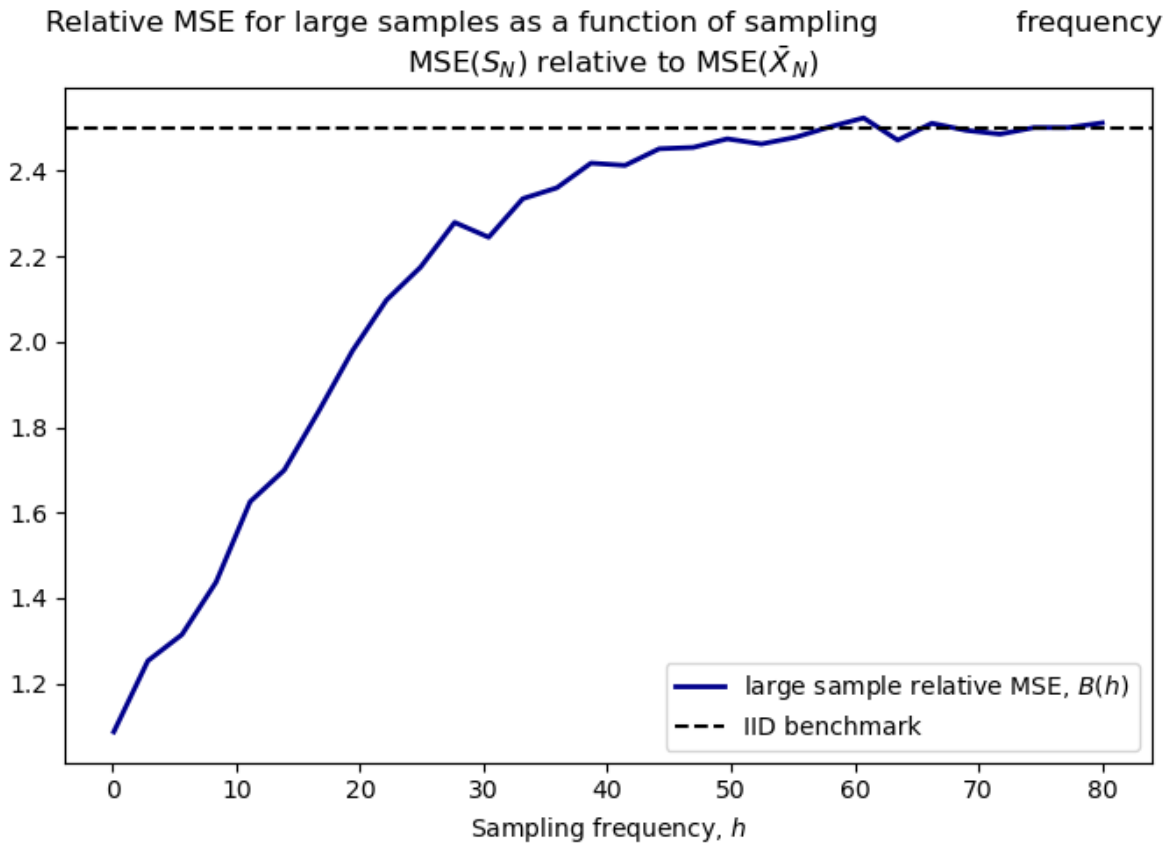
var_est_store = np.array(var_est_store)
mean_est_store = np.array(mean_est_store)

# Save mse of estimators
mse_mean = np.var(mean_est_store, 1) + (np.mean(mean_est_store, 1) - μ)**2
mse_var = np.var(var_est_store, 1) \
          + (np.mean(var_est_store, 1) - var_uncond)**2

benchmark_rate = 2 * var_uncond       # IID case

# Relative MSE for large samples
rate_h = mse_var / mse_mean

fig, ax = plt.subplots(figsize=(8, 5))
ax.plot(h_grid, rate_h, c='darkblue', lw=2,
        label=r'large sample relative MSE, $B(h)$')
ax.axhline(benchmark_rate, c='k', ls='--', label=r'IID benchmark')
ax.set_title('Relative MSE for large samples as a function of sampling \
              frequency \n MSE($S_N$) relative to MSE($\\bar X_N$)')
ax.set_xlabel('Sampling frequency, $h$')
ax.legend()
plt.show()
```



The above figure illustrates the relationship between the asymptotic relative MSEs and the sampling frequency

- We can see that with low-frequency data – large values of h – the ratio of asymptotic rates approaches the IID case.
- As h gets smaller – the higher the frequency – the relative performance of the variance estimator is better in the sense that the ratio of asymptotic rates gets smaller. That is, as the time dependence gets more pronounced, the rate of convergence of the mean estimator's MSE deteriorates more than that of the variance estimator.

CHAPTER

THIRTYEIGHT

IRRELEVANCE OF CAPITAL STRUCTURES WITH COMPLETE MARKETS

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install --upgrade quantecon
!conda install -y -c plotly plotly plotly-orca
```

38.1 Introduction

This is a prolegomenon to another lecture *Equilibrium Capital Structures with Incomplete Markets* about a model with incomplete markets authored by Bisin, Clementi, and Gottardi [Bisin *et al.*, 2018].

We adopt specifications of preferences and technologies very close to Bisin, Clemente, and Gottardi's but unlike them assume that there are complete markets in one-period Arrow securities.

This simplification of BCG's setup helps us by

- creating a benchmark economy to compare with outcomes in BCG's incomplete markets economy
- creating a good guess for initial values of some equilibrium objects to be computed in BCG's incomplete markets economy via an iterative algorithm
- illustrating classic complete markets outcomes that include
  - indeterminacy of consumers' portfolio choices
  - indeterminacy of firms' financial structures that underlies a Modigliani-Miller theorem [Modigliani and Miller, 1958]
- introducing `Big K, little k` issues in a simple context that will recur in the BCG incomplete markets environment

A Big K, little k analysis also played roles in this quantecon lecture as well as here and *here*.

38.1.1 Setup

The economy lasts for two periods, $t = 0, 1$.

There are two types of consumers named $i = 1, 2$.

A scalar random variable ϵ with probability density $g(\epsilon)$ affects both

- the return in period 1 from investing $k \geq 0$ in physical capital in period 0.
- exogenous period 1 endowments of the consumption good for agents of types $i = 1$ and $i = 2$.

Type $i = 1$ and $i = 2$ agents' period 1 endowments are correlated with the return on physical capital in different ways.

We discuss two arrangements:

- a command economy in which a benevolent planner chooses k and allocates goods to the two types of consumers in each period and each random second period state
- a competitive equilibrium with markets in claims on physical capital and a complete set (possibly a continuum) of one-period Arrow securities that pay period 1 consumption goods contingent on the realization of random variable ϵ.

38.1.2 Endowments

There is a single consumption good in period 0 and at each random state ϵ in period 1.

Economy-wide endowments in periods 0 and 1 are

$$
\begin{aligned}
w_0 \\
w_1(\epsilon) \text{ in state } \epsilon
\end{aligned}
$$

Soon we'll explain how aggregate endowments are divided between type $i = 1$ and type $i = 2$ consumers.

We don't need to do that in order to describe a social planning problem.

38.1.3 Technology:

Where $\alpha \in (0, 1)$ and $A > 0$

$$
\begin{aligned}
c_0^1 + c_0^2 + k &= w_0^1 + w_0^2 \\
c_1^1(\epsilon) + c_1^2(\epsilon) &= w_1^1(\epsilon) + w_1^2(\epsilon) + e^\epsilon A k^\alpha, \quad k \geq 0
\end{aligned}
$$

38.1.4 Preferences:

A consumer of type i orders period 0 consumption c_0^i and state ϵ, period 1 consumption $c_1^i(\epsilon)$ by

$$
u^i = u(c_0^i) + \beta \int u(c_1^i(\epsilon)) g(\epsilon) d\epsilon, \quad i = 1, 2
$$

$\beta \in (0, 1)$ and the one-period utility function is

$$
u(c) = \begin{cases} \frac{c^{1-\gamma}}{1-\gamma} & \text{if } \gamma \neq 1 \\ \log c & \text{if } \gamma = 1 \end{cases}
$$

38.1.5 Parameterizations

Following BCG, we shall employ the following parameterizations:

$$\begin{aligned}\epsilon &\sim \mathcal{N}(\mu, \sigma^2) \\ u(c) &= \frac{c^{1-\gamma}}{1-\gamma} \\ w_1^i(\epsilon) &= e^{-\chi_i \mu - .5\chi_i^2\sigma^2 + \chi_i \epsilon}, \quad \chi_i \in [0,1]\end{aligned}$$

Sometimes instead of asuming $\epsilon \sim g(\epsilon) = \mathcal{N}(0, \sigma^2)$, we'll assume that $g(\cdot)$ is a probability mass function that serves as a discrete approximation to a standardized normal density.

38.1.6 Pareto criterion and planning problem

The planner's objective function is

$$\text{obj} = \phi_1 u^1 + \phi_2 u^2, \quad \phi_i \geq 0, \quad \phi_1 + \phi_2 = 1$$

where $\phi_i \geq 0$ is a Pareto weight that the planner attaches to a consumer of type i.

We form the following Lagrangian for the planner's problem:

$$\begin{aligned}L = &\sum_{i=1}^{2} \phi_i \left[u(c_0^i) + \beta \int u(c_1^i(\epsilon)) g(\epsilon) d\epsilon\right] \\ &+ \lambda_0 \left[w_0^1 + w_0^2 - k - c_0^1 - c_0^2\right] \\ &+ \beta \int \lambda_1(\epsilon) \left[w_1^1(\epsilon) + w_1^2(\epsilon) + e^\epsilon A k^\alpha - c_1^1(\epsilon) - c_1^2(\epsilon)\right] g(\epsilon) d\epsilon\end{aligned}$$

First-order necessary optimality conditions for the planning problem are:

$$\begin{aligned}c_0^1: &\quad \phi_1 u'(c_0^1) - \lambda_0 = 0 \\ c_0^2: &\quad \phi_2 u'(c_0^2) - \lambda_0 = 0 \\ c_1^1(\epsilon): &\quad \phi_1 \beta u'(c_1^1(\epsilon)) g(\epsilon) - \beta \lambda_1(\epsilon) g(\epsilon) = 0 \\ c_1^2(\epsilon): &\quad \phi_2 \beta u'(c_1^2(\epsilon)) g(\epsilon) - \beta \lambda_1(\epsilon) g(\epsilon) = 0 \\ k: &\quad -\lambda_0 + \beta \alpha A k^{\alpha-1} \int \lambda_1(\epsilon) e^\epsilon g(\epsilon) d\epsilon = 0\end{aligned}$$

The first four equations imply that

$$\begin{aligned}\frac{u'(c_1^1(\epsilon))}{u'(c_0^1))} = \frac{u'(c_1^2(\epsilon))}{u'(c_0^2))} = \frac{\lambda_1(\epsilon)}{\lambda_0} \\ \frac{u'(c_0^1)}{u'(c_0^2)} = \frac{u'(c_1^1(\epsilon))}{u'(c_1^2(\epsilon))} = \frac{\phi_2}{\phi_1}\end{aligned}$$

These together with the fifth first-order condition for the planner imply the following equation that determines an optimal choice of capital

$$1 = \beta \alpha A k^{\alpha-1} \int \frac{u'(c_1^i(\epsilon))}{u'(c_0^i)} e^\epsilon g(\epsilon) d\epsilon$$

for $i = 1, 2$.

38.1.7 Helpful observations and bookkeeping

Evidently,

$$u'(c) = c^{-\gamma}$$

and

$$\frac{u'(c^1)}{u'(c^2)} = \left(\frac{c^1}{c^2}\right)^{-\gamma} = \frac{\phi_2}{\phi_1}$$

where it is to be understood that this equation holds for $c^1 = c_0^1$ and $c^2 = c_0^2$ and also for $c^1 = c^1(\epsilon)$ and $c^2 = c^2(\epsilon)$ for all ϵ.

With the same understanding, it follows that

$$\left(\frac{c^1}{c^2}\right) = \left(\frac{\phi_2}{\phi_1}\right)^{-\gamma^{-1}}$$

Let $c = c^1 + c^2$.

It follows from the preceding equation that

$$\begin{aligned} c^1 &= \eta c \\ c^2 &= (1-\eta)c \end{aligned}$$

where $\eta \in [0,1]$ is a function of ϕ_1 and γ.

Consequently, we can write the planner's first-order condition for k as

$$1 = \beta\alpha A k^{\alpha-1} \int \left(\frac{w_1(\epsilon) + Ak^\alpha e^\epsilon}{w_0 - k}\right)^{-\gamma} e^\epsilon g(\epsilon) d\epsilon$$

which is one equation to be solved for $k \geq 0$.

Anticipating a `Big K, little k` idea widely used in macroeconomics, to be discussed in detail below, let K be the value of k that solves the preceding equation so that

$$1 = \beta\alpha A K^{\alpha-1} \int \left(\frac{w_1(\epsilon) + AK^\alpha e^\epsilon}{w_0 - K}\right)^{-\gamma} g(\epsilon) e^\epsilon d\epsilon \tag{38.1}$$

The associated optimal consumption allocation is

$$\begin{aligned} C_0 &= w_0 - K \\ C_1(\epsilon) &= w_1(\epsilon) + AK^\alpha e^\epsilon \\ c_0^1 &= \eta C_0 \\ c_0^2 &= (1-\eta) C_0 \\ c_1^1(\epsilon) &= \eta C_1(\epsilon) \\ c_1^2(\epsilon) &= (1-\eta) C_1(\epsilon) \end{aligned}$$

where $\eta \in [0,1]$ is the consumption share parameter mentioned above that is a function of the Pareto weight ϕ_1 and the utility curvature parameter γ.

Remarks

The relative Pareto weight parameter η does not appear in equation (38.1) that determines K.

Neither does it influence C_0 or $C_1(\epsilon)$, which depend solely on K.

The role of η is to determine how to allocate total consumption between the two types of consumers.

Thus, the planner's choice of K does not interact with how it wants to allocate consumption.

38.2 Competitive equilibrium

We now describe a competitive equilibrium for an economy that has specifications of consumer preferences, technology, and aggregate endowments that are identical to those in the preceding planning problem.

While prices do not appear in the planning problem – only quantities do – prices play an important role in a competitive equilibrium.

To understand how the planning economy is related to a competitive equilibrium, we now turn to the `Big K, little k` distinction.

38.2.1 Measures of agents and firms

We follow BCG in assuming that there are unit measures of

- consumers of type $i = 1$
- consumers of type $i = 2$
- firms with access to the production technology that converts k units of time 0 good into $Ak^\alpha e^\epsilon$ units of the time 1 good in random state ϵ

Thus, let $\omega \in [0, 1]$ index a particular consumer of type i.

Then define Big C^i as

$$C^i = \int_0^1 c^i(\omega) d\,\omega$$

In the same spirit, let $\zeta \in [0, 1]$ index a particular firm. Then define Big K as

$$K = \int_0^1 k(\zeta) d\,\zeta$$

The assumption that there are continua of our three types of agents plays an important role making each individual agent into a powerless **price taker**:

- an individual consumer chooses its own (infinesimal) part $c^i(\omega)$ of C^i taking prices as given
- an individual firm chooses its own (infinitesmimal) part $k(\zeta)$ of K taking prices as
- equilibrium prices depend on the `Big K, Big C` objects K and C

Nevertheless, in equilibrium, $K = k, C^i = c^i$

The assumption about measures of agents is thus a powerful device for making a host of competitive agents take as given equilibrium prices that are determined by the independent decisions of hosts of agents who behave just like they do.

Ownership

Consumers of type i own the following exogenous quantities of the consumption good in periods 0 and 1:

$$\begin{aligned} w_0^i, \quad & i = 1, 2 \\ w_1^i(\epsilon) \quad & i = 1, 2 \end{aligned}$$

where

$$\sum_i w_0^i = w_0$$

$$\sum_i w_1^i(\epsilon) = w_1(\epsilon)$$

Consumers also own shares in a firm that operates the technology for converting nonnegative amounts of the time 0 consumption good one-for-one into a capital good k that produces $Ak^\alpha e^\epsilon$ units of the time 1 consumption good in time 1 state ϵ.

Consumers of types $i = 1, 2$ are endowed with θ_0^i shares of a firm and

$$\theta_0^1 + \theta_0^2 = 1$$

Asset markets

At time 0, consumers trade the following assets with other consumers and with firms:

- equities (also known as stocks) issued by firms
- one-period Arrow securities that pay one unit of consumption at time 1 when the shock ϵ assumes a particular value

Later, we'll allow the firm to issue bonds too, but not now.

38.2.2 Objects appearing in a competitive equilibrium

Let

- $a^i(\epsilon)$ be consumer i 's purchases of claims on time 1 consumption in state ϵ
- $q(\epsilon)$ be a pricing kernel for one-period Arrow securities
- $\theta_0^i \geq 0$ be consumer i's intial share of the firm, $\sum_i \theta_0^i = 1$
- θ^i be the fraction of a firm's shares purchased by consumer i at time $t = 0$
- V be the value of the representative firm
- $\tilde{V}$ be the value of equity issued by the representative firm
- K, C_0 be two scalars and $C_1(\epsilon)$ a function that we use to construct a guess about an equilibrium pricing kernel for Arrow securities

We proceed to describe constrained optimum problems faced by consumers and a representative firm in a competitive equilibrium.

38.2.3 A representative firm's problem

A representative firm takes Arrow security prices $q(\epsilon)$ as given.

The firm purchases capital $k \geq 0$ from consumers at time 0 and finances itself by issuing equity at time 0.

The firm produces time 1 goods $Ak^\alpha e^\epsilon$ in state ϵ and pays all of these `earnings` to owners of its equity.

The value of a firm's equity at time 0 can be computed by multiplying its state-contingent earnings by their Arrow securities prices and then adding over all contingencies:

$$\tilde{V} = \int Ak^\alpha e^\epsilon q(\epsilon) d\epsilon$$

Owners of a firm want it to choose k to maximize

$$V = -k + \int Ak^\alpha e^\epsilon q(\epsilon) d\epsilon$$

The firm's first-order necessary condition for an optimal k is

$$-1 + \alpha A k^{\alpha-1} \int e^\epsilon q(\epsilon) d\epsilon = 0$$

The time 0 value of a representative firm is

$$V = -k + \tilde{V}$$

The right side equals the value of equity minus the cost of the time 0 goods that it purchases and uses as capital.

38.2.4 A consumer's problem

We now pose a consumer's problem in a competitive equilibrium.

As a price taker, each consumer faces a given Arrow securities pricing kernel $q(\epsilon)$, a given value of a firm V that has chosen capital stock k, a price of equity $\tilde{V}$, and prospective next period random dividends $Ak^\alpha e^\epsilon$.

If we evaluate consumer i's time 1 budget constraint at zero consumption $c_1^i(\epsilon) = 0$ and solve for $-a^i(\epsilon)$ we obtain

$$-\bar{a}^i(\epsilon; \theta^i) = w_1^i(\epsilon) + \theta^i A k^\alpha e^\epsilon \tag{38.2}$$

The quantity $-\bar{a}^i(\epsilon; \theta^i)$ is the maximum amount that it is feasible for consumer i to repay to his Arrow security creditors at time 1 in state ϵ.

Notice that $-\bar{a}^i(\epsilon; \theta^i)$ defined in (38.2) depends on

- his endowment $w_1^i(\epsilon)$ at time 1 in state ϵ
- his share θ^i of a representive firm's dividends

These constitute two sources of **collateral** that back the consumer's issues of Arrow securities that pay off in state ϵ

Consumer i chooses a scalar c_0^i and a function $c_1^i(\epsilon)$ to maximize

$$u(c_0^i) + \beta \int u(c_1^i(\epsilon)) g(\epsilon) d\epsilon$$

subject to time 0 and time 1 budget constraints

$$\begin{aligned} c_0^i &\leq w_0^i + \theta_0^i V - \int q(\epsilon) a^i(\epsilon) d\epsilon - \theta^i \tilde{V} \\ c_1^i(\epsilon) &\leq w_1^i(\epsilon) + \theta^i A k^\alpha e^\epsilon + a^i(\epsilon) \end{aligned}$$

Attach Lagrange multiplier λ_0^i to the budget constraint at time 0 and scaled Lagrange multiplier $\beta\lambda_1^i(\epsilon)g(\epsilon)$ to the budget constraint at time 1 and state ϵ, then form the Lagrangian

$$\begin{aligned} L^i &= u(c_0^i) + \beta \int u(c_1^i(\epsilon))g(\epsilon)d\epsilon \\ &\quad + \lambda_0^i[w_0^i + \theta_0^i - \int q(\epsilon)a^i(\epsilon)d\epsilon - \theta^i\tilde{V} - c_0^i] \\ &\quad + \beta \int \lambda_1^i(\epsilon)[w_1^i(\epsilon) + \theta^i Ak^\alpha e^\epsilon + a^i(\epsilon)c_1^i(\epsilon)]g(\epsilon)d\epsilon \end{aligned}$$

Off corners, first-order necessary conditions for an optimum with respect to c_0^i, $c_1^i(\epsilon)$, and $a^i(\epsilon)$ are

$$\begin{aligned} c_0^i: &\quad u'(c_0^i) - \lambda_0^i = 0 \\ c_1^i(\epsilon): &\quad \beta u'(c_1^i(\epsilon))g(\epsilon) - \beta\lambda_1^i(\epsilon)g(\epsilon) = 0 \\ a^i(\epsilon): &\quad -\lambda_0^i q(\epsilon) + \beta\lambda_1^i(\epsilon) = 0 \end{aligned}$$

These equations imply that consumer i adjusts its consumption plan to satisfy

$$q(\epsilon) = \beta\left(\frac{u'(c_1^i(\epsilon))}{u'(c_0^i)}\right)g(\epsilon) \tag{38.3}$$

To deduce a restriction on equilibrium prices, we solve the period 1 budget constraint to express $a^i(\epsilon)$ as

$$a^i(\epsilon) = c_1^i(\epsilon) - w_1^i(\epsilon) - \theta^i Ak^\alpha e^\epsilon$$

then substitute the expression on the right side into the time 0 budget constraint and rearrange to get the single intertemporal budget constraint

$$w_0^i + \theta_0^i V + \int w_1^i(\epsilon)q(\epsilon)d\epsilon + \theta^i\left[Ak^\alpha \int e^\epsilon q(\epsilon)d\epsilon - \tilde{V}\right] \geq c_0^i + \int c_1^i(\epsilon)q(\epsilon)d\epsilon \tag{38.4}$$

The right side of inequality (38.4) is the present value of consumer i's consumption while the left side is the present value of consumer i's endowment when consumer i buys θ^i shares of equity.

From inequality (38.4), we deduce two findings.

1. No arbitrage profits condition:

Unless

$$\tilde{V} = Ak^\alpha \int e^\epsilon q(\epsilon)d\epsilon \tag{38.5}$$

an **arbitrage** opportunity would be open.

If

$$\tilde{V} > Ak^\alpha \int e^\epsilon q(\epsilon)d\epsilon$$

the consumer could afford an arbitrarily high present value of consumption by setting θ^i to an arbitrarily large **negative** number.

If

$$\tilde{V} < Ak^\alpha \int e^\epsilon q(\epsilon)d\epsilon$$

the consumer could afford an arbitrarily high present value of consumption by setting θ^i to be arbitrarily large **positive** number.

Since resources are finite, there can exist no such arbitrage opportunity in a competitive equilibrium.

Therefore, it must be true that the following no arbitrage condition prevails:

$$\tilde{V} = \int Ak^{\alpha} e^{\epsilon} q(\epsilon; K) d\epsilon \tag{38.6}$$

Equation (38.6) asserts that the value of equity equals the value of the state-contingent dividends $Ak^{\alpha}e^{\epsilon}$ evaluated at the Arrow security prices $q(\epsilon; K)$ that we have expressed as a function of K.

We'll say more about this equation later.

2. Indeterminacy of portfolio

When the no-arbitrage pricing equation (38.6) prevails, a consumer of type i's choice θ^i of equity is indeterminate.

Consumer of type i can offset any choice of θ^i by setting an appropriate schedule $a^i(\epsilon)$ for purchasing state-contingent securities.

38.2.5 Computing competitive equilibrium prices and quantities

Having computed an allocation that solves the planning problem, we can readily compute a competitive equilibrium via the following steps that, as we'll see, relies heavily on the `Big K, little k`, `Big C, little c` logic mentioned earlier:

- a competitive equilbrium allocation equals the allocation chosen by the planner
- competitive equilibrium prices and the value of a firm's equity are encoded in shadow prices from the planning problem that depend on Big K and Big C.

To substantiate that this procedure is valid, we proceed as follows.

With K in hand, we make the following guess for competitive equilibrium Arrow securities prices

$$q(\epsilon; K) = \beta \left(\frac{u'\left(w_1(\epsilon) + AK^{\alpha} e^{\epsilon}\right)}{u'(w_0 - K)} \right)^{-\gamma} \tag{38.7}$$

To confirm the guess, we begin by considering its consequences for the firm's choice of k.

With Arrow securities prices (38.7), the firm's first-order necessary condition for choosing k becomes

$$-1 + \alpha A k^{\alpha - 1} \int e^{\epsilon} q(\epsilon; K) d\epsilon = 0 \tag{38.8}$$

which can be verified to be satisfied if the firm sets

$$k = K$$

because by setting $k = K$ equation (38.8) becomes equivalent with the planner's first-order condition (38.1) for setting K.

To pose a consumer's problem in a competitive equilibrium, we require not only the above guess for the Arrow securities pricing kernel $q(\epsilon)$ but the value of equity $\tilde{V}$:

$$\tilde{V} = \int AK^{\alpha} e^{\epsilon} q(\epsilon; K) d\epsilon \tag{38.9}$$

Let $\tilde{V}$ be the value of equity implied by Arrow securities price function (38.7) and formula (38.9).

At the Arrow securities prices $q(\epsilon)$ given by (38.7) and equity value $\tilde{V}$ given by (38.9), consumer $i = 1, 2$ choose consumption allocations and portolios that satisfy the first-order necessary conditions

$$\beta \left(\frac{u'(c_1^i(\epsilon))}{u'(c_0^i)} \right) g(\epsilon) = q(\epsilon; K)$$

It can be verified directly that the following choices satisfy these equations

$$
\begin{aligned}
c_0^1 + c_0^2 &= C_0 = w_0 - K \\
c_0^1(\epsilon) + c_0^2(\epsilon) &= C_1(\epsilon) = w_1(\epsilon) + Ak^\alpha e^\epsilon \\
\frac{c_1^2(\epsilon)}{c_1^1(\epsilon)} &= \frac{c_0^2}{c_0^1} = \frac{1-\eta}{\eta}
\end{aligned}
$$

for an $\eta \in (0,1)$ that depends on consumers' endowments $[w_0^1, w_0^2, w_1^1(\epsilon), w_1^2(\epsilon), \theta_0^1, \theta_0^2]$.

Remark: Multiple arrangements of endowments $[w_0^1, w_0^2, w_1^1(\epsilon), w_1^2(\epsilon), \theta_0^1, \theta_0^2]$ associated with the same distribution of wealth η. Can you explain why?

Hint: Think about the portfolio indeterminacy finding above.

38.2.6 Modigliani-Miller theorem

We now allow a firm to issue both bonds and equity.

Payouts from equity and bonds, respectively, are

$$
\begin{aligned}
d^e(k, b; \epsilon) &= \max\left\{e^\epsilon A k^\alpha - b, 0\right\} \\
d^b(k, b; \epsilon) &= \min\left\{\frac{e^\epsilon A k^\alpha}{b}, 1\right\}
\end{aligned}
$$

Thus, one unit of the bond pays one unit of consumption at time 1 in state ϵ if $Ak^\alpha e^\epsilon - b \geq 0$, which is true when $\epsilon \geq \epsilon^* = \log \frac{b}{Ak^\alpha}$, and pays $\frac{Ak^\alpha e^\epsilon}{b}$ units of time 1 consumption in state ϵ when $\epsilon < \epsilon^*$.

The value of the firm is now the sum of equity plus the value of bonds, which we denote

$$
\tilde{V} + bp(k, b)
$$

where $p(k, b)$ is the price of one unit of the bond when a firm with k units of physical capital issues b bonds.

We continue to assume that there are complete markets in Arrow securities with pricing kernel $q(\epsilon)$.

A version of the no-arbitrage-in-equilibrium argument that we presented earlier implies that the value of equity and the price of bonds are

$$
\begin{aligned}
\tilde{V} &= Ak^\alpha \int_{\epsilon^*}^{\infty} e^\epsilon q(\epsilon) d\epsilon - b \int_{\epsilon^*}^{\infty} q(\epsilon) d\epsilon \\
p(k, b) &= \frac{Ak^\alpha}{b} \int_{-\infty}^{\epsilon^*} e^\epsilon q(\epsilon) d\epsilon + \int_{\epsilon^*}^{\infty} q(\epsilon) d\epsilon
\end{aligned}
$$

Consequently, the value of the firm is

$$
\tilde{V} + p(k, b)b = Ak^\alpha \int_{-\infty}^{\infty} e^\epsilon q(\epsilon) d\epsilon,
$$

which is the same expression that we obtained above when we assumed that the firm issued only equity.

We thus obtain a version of the celebrated Modigliani-Miller theorem [Modigliani and Miller, 1958] about firms' finance:

Modigliani-Miller theorem:

- The value of a firm is independent the mix of equity and bonds that it uses to finance its physical capital.

- The firms's decision about how much physical capital to purchase does not depend on whether it finances those purchases by issuing bonds or equity
- The firm's choice of whether to finance itself by issuing equity or bonds is indeterminant

Please note the role of the assumption of complete markets in Arrow securities in substantiating these claims.

In *Equilibrium Capital Structures with Incomplete Markets*, we will assume that markets are (very) incomplete – we'll shut down markets in almost all Arrow securities.

That will pull the rug from underneath the Modigliani-Miller theorem.

38.3 Code

We create a class object `BCG_complete_markets` to compute equilibrium allocations of the complete market BCG model given a list of parameter values.

It consists of 4 functions that do the following things:

- `opt_k` computes the planner's optimal capital K
  - First, create a grid for capital.
  - Then for each value of capital stock in the grid, compute the left side of the planner's first-order necessary condition for k, that is,

$$\beta\alpha AK^{\alpha-1}\int\left(\frac{w_1(\epsilon)+AK^{\alpha}e^{\epsilon}}{w_0-K}\right)^{-\gamma}e^{\epsilon}g(\epsilon)d\epsilon-1=0$$

  - Find k that solves this equation.
- `q` computes Arrow security prices as a function of the productivity shock ϵ and capital K:

$$q(\epsilon;K)=\beta\left(\frac{u'\left(w_1(\epsilon)+AK^{\alpha}e^{\epsilon}\right)}{u'(w_0-K)}\right)$$

- `V` solves for the firm value given capital k:

$$V=-k+\int Ak^{\alpha}e^{\epsilon}q(\epsilon;K)d\epsilon$$

- `opt_c` computes optimal consumptions c_0^i, and $c^i(\epsilon)$:
  - The function first computes weight η using the budget constraint for agent 1:

$$w_0^1+\theta_0^1V+\int w_1^1(\epsilon)q(\epsilon)d\epsilon=c_0^1+\int c_1^1(\epsilon)q(\epsilon)d\epsilon=\eta\left(C_0+\int C_1(\epsilon)q(\epsilon)d\epsilon\right)$$

    where

$$\begin{aligned}C_0&=w_0-K\\C_1(\epsilon)&=w_1(\epsilon)+AK^{\alpha}e^{\epsilon}\end{aligned}$$

  - It computes consumption for each agent as

$$\begin{aligned}c_0^1&=\eta C_0\\c_0^2&=(1-\eta)C_0\\c_1^1(\epsilon)&=\eta C_1(\epsilon)\\c_1^2(\epsilon)&=(1-\eta)C_1(\epsilon)\end{aligned}$$

The list of parameters includes:

- χ_1, χ_2: Correlation parameters for agents 1 and 2. Default values are 0 and 0.9, respectively.
- w_0^1, w_0^2: Initial endowments. Default values are 1.
- θ_0^1, θ_0^2: Consumers' initial shares of a representative firm. Default values are 0.5.
- ψ: CRRA risk parameter. Default value is 3.
- α: Returns to scale production function parameter. Default value is 0.6.
- A: Productivity of technology. Default value is 2.5.
- μ, σ: Mean and standard deviation of the log of the shock. Default values are -0.025 and 0.4, respectively.
- β: time preference discount factor. Default value is .96.
- `nb_points_integ`: number of points used for integration through Gauss-Hermite quadrature: default value is 10

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm
from numba import njit, prange
from quantecon.optimize import root_finding
```

```
#=========== Class: BCG for complete markets ===========#
class BCG_complete_markets:

    # init method or constructor
    def __init__(self,
                 χ1 = 0,
                 χ2 = 0.9,
                 w10 = 1,
                 w20 = 1,
                 θ10 = 0.5,
                 θ20 = 0.5,
                 ψ = 3,
                 α = 0.6,
                 A = 2.5,
                 μ = -0.025,
                 σ = 0.4,
                 β = 0.96,
                 nb_points_integ = 10):

        #=========== Setup ===========#
        # Risk parameters
        self.χ1 = χ1
        self.χ2 = χ2

        # Other parameters
        self.ψ = ψ
        self.α = α
        self.A = A
        self.μ = μ
        self.σ = σ
        self.β = β

        # Utility
```

(continues on next page)

(continued from previous page)

```
        self.u = lambda c: (c**(1-ψ)) / (1-ψ)

        # Production
        self.f = njit(lambda k: A * (k ** α))
        self.Y = lambda ϵ, k: np.exp(ϵ) * self.f(k)

        # Initial endowments
        self.w10 = w10
        self.w20 = w20
        self.w0 = w10 + w20

        # Initial holdings
        self.θ10 = θ10
        self.θ20 = θ20

        # Endowments at t=1
        w11 = njit(lambda ϵ: np.exp(-χ1*μ - 0.5*(χ1**2)*(σ**2) + χ1*ϵ))
        w21 = njit(lambda ϵ: np.exp(-χ2*μ - 0.5*(χ2**2)*(σ**2) + χ2*ϵ))
        self.w11 = w11
        self.w21 = w21

        self.w1 = njit(lambda ϵ: w11(ϵ) + w21(ϵ))

        # Normal PDF
        self.g = lambda x: norm.pdf(x, loc=μ, scale=σ)

        # Integration
        x, self.weights = np.polynomial.hermite.hermgauss(nb_points_integ)
        self.points_integral = np.sqrt(2) * σ * x + μ

        self.k_foc = k_foc_factory(self)

    #=========== Optimal k ===========#
    # Function: solve for optimal k
    def opt_k(self, plot=False):
        w0 = self.w0

        # Grid for k
        kgrid = np.linspace(1e-4, w0-1e-4, 100)

        # get FONC values for each k in the grid
        kfoc_list = [];
        for k in kgrid:
            kfoc = self.k_foc(k, self.χ1, self.χ2)
            kfoc_list.append(kfoc)

        # Plot FONC for k
        if plot:
            fig, ax = plt.subplots(figsize=(8,7))
            ax.plot(kgrid, kfoc_list, color='blue', label=r'FONC for k')
            ax.axhline(0, color='red', linestyle='--')
            ax.legend()
            ax.set_xlabel(r'k')
            plt.show()

        # Find k that solves the FONC
```

(continues on next page)

(continued from previous page)

```
        kk = root_finding.newton_secant(self.k_foc, 1e-2, args=(self.χ1, self.χ2)).
↪root

        return kk

    #=========== Arrow security price ===========#
    # Function: Compute Arrow security price
    def q(self,ϵ,k):
        β = self.β
        ψ = self.ψ
        w0 = self.w0
        w1 = self.w1
        fk = self.f(k)
        g = self.g

        return β * ((w1(ϵ) + np.exp(ϵ)*fk) / (w0 - k))**(-ψ)


    #=========== Firm value V ===========#
    # Function: compute firm value V
    def V(self, k):
        q = self.q
        fk = self.f(k)
        weights = self.weights
        integ = lambda ϵ: np.exp(ϵ) * fk * q(ϵ, k)

        return -k + np.sum(weights * integ(self.points_integral)) / np.sqrt(np.pi)

    #=========== Optimal c ===========#
    # Function: Compute optimal consumption choices c
    def opt_c(self, k=None, plot=False):
        w1 = self.w1
        w0 = self.w0
        w10 = self.w10
        w11 = self.w11
        θ10 = self.θ10
        Y = self.Y
        q = self.q
        V = self.V
        weights = self.weights

        if k is None:
            k = self.opt_k()

        # Solve for the ratio of consumption  from the intertemporal B.C.
        fk = self.f(k)

        c1 = lambda ϵ: (w1(ϵ) + np.exp(ϵ)*fk)*q(ϵ,k)
        denom = np.sum(weights * c1(self.points_integral)) / np.sqrt(np.pi) + (w0 - k)

        w11q = lambda ϵ: w11(ϵ)*q(ϵ,k)
        num = w10 + θ10 * V(k) + np.sum(weights * w11q(self.points_integral)) / np.
↪sqrt(np.pi)

        η = num / denom
```

(continues on next page)

(continued from previous page)

```
        # Consumption choices
        c10 = η * (w0 - k)
        c20 = (1-η) * (w0 - k)
        c11 = lambda ϵ: η * (w1(ϵ)+Y(ϵ,k))
        c21 = lambda ϵ: (1-η) * (w1(ϵ)+Y(ϵ,k))

        return c10, c20, c11, c21


def k_foc_factory(model):
    ψ = model.ψ
    f = model.f
    β = model.β
    α = model.α
    A = model.A
    ψ = model.ψ
    w0 = model.w0
    μ = model.μ
    σ = model.σ

    weights = model.weights
    points_integral = model.points_integral

    w11 = njit(lambda ϵ, χ1, : np.exp(-χ1*μ - 0.5*(χ1**2)*(σ**2) + χ1*ϵ))
    w21 = njit(lambda ϵ, χ2: np.exp(-χ2*μ - 0.5*(χ2**2)*(σ**2) + χ2*ϵ))
    w1 = njit(lambda ϵ, χ1, χ2: w11(ϵ, χ1) + w21(ϵ, χ2))

    @njit
    def integrand(ϵ, χ1, χ2, k=1e-4):
        fk = f(k)
        return (w1(ϵ, χ1, χ2) + np.exp(ϵ) * fk) ** (-ψ) * np.exp(ϵ)

    @njit
    def k_foc(k, χ1, χ2):
        int_k = np.sum(weights * integrand(points_integral, χ1, χ2, k=k)) / np.
↪sqrt(np.pi)

        mul = β * α * A * k ** (α - 1) / ((w0 - k) ** (-ψ))
        val = mul * int_k - 1

        return val

    return k_foc
```

38.3.1 Examples

Below we provide some examples of how to use `BCG_complete markets`.

1st example

In the first example, we set up instances of BCG complete markets models.

We can use either default parameter values or set parameter values as we want.

The two instances of the BCG complete markets model, `mdl1` and `mdl2`, represent the model with default parameter settings and with agent 2's income correlation altered to be $\chi_2 = -0.9$, respectively.

```
# Example: BCG model for complete markets
mdl1 = BCG_complete_markets()
mdl2 = BCG_complete_markets(χ2=-0.9)
```

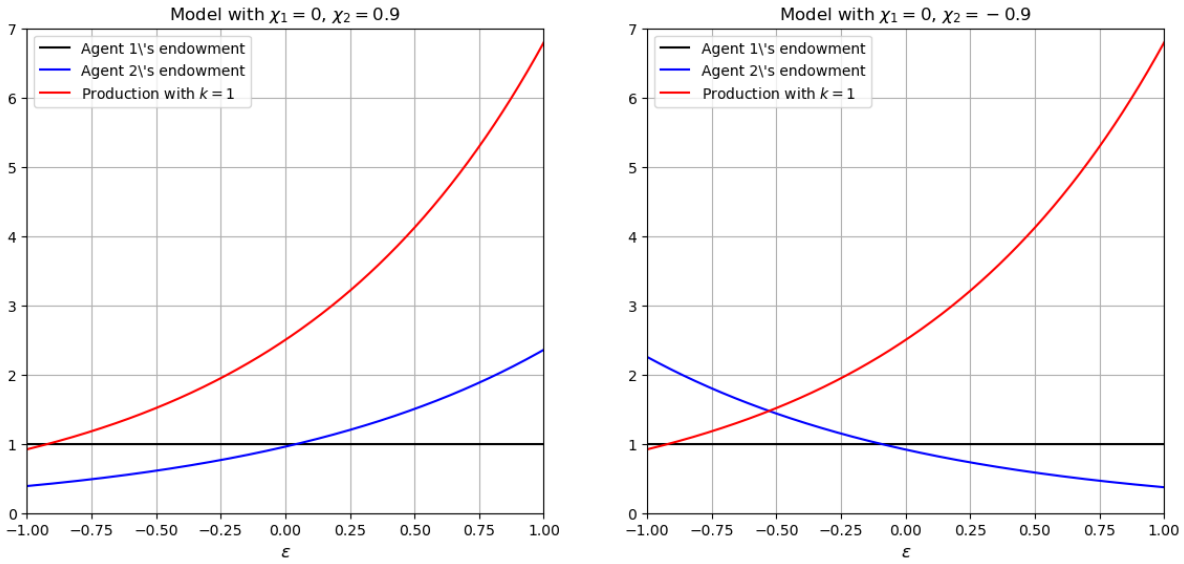
Let's plot the agents' time-1 endowments with respect to shocks to see the difference in the two models:

```
#==== Figure 1: HH endowments and firm productivity ====#
# Realizations of innovation from -3 to 3
epsgrid = np.linspace(-1,1,1000)


fig, ax = plt.subplots(1,2,figsize=(14,6))
ax[0].plot(epsgrid, mdl1.w11(epsgrid), color='black', label=r'Agent 1\'s endowment')
ax[0].plot(epsgrid, mdl1.w21(epsgrid), color='blue', label=r'Agent 2\'s endowment')
ax[0].plot(epsgrid, mdl1.Y(epsgrid,1), color='red', label=r'Production with $k=1$')
ax[0].set_xlim([-1,1])
ax[0].set_ylim([0,7])
ax[0].set_xlabel(r'$\epsilon$',fontsize=12)
ax[0].set_title(r'Model with $\chi_1 = 0$, $\chi_2 = 0.9$')
ax[0].legend()
ax[0].grid()

ax[1].plot(epsgrid, mdl2.w11(epsgrid), color='black', label=r'Agent 1\'s endowment')
ax[1].plot(epsgrid, mdl2.w21(epsgrid), color='blue', label=r'Agent 2\'s endowment')
ax[1].plot(epsgrid, mdl2.Y(epsgrid,1), color='red', label=r'Production with $k=1$')
ax[1].set_xlim([-1,1])
ax[1].set_ylim([0,7])
ax[1].set_xlabel(r'$\epsilon$',fontsize=12)
ax[1].set_title(r'Model with $\chi_1 = 0$, $\chi_2 = -0.9$')
ax[1].legend()
ax[1].grid()

plt.show()
```



Let's also compare the optimal capital stock, k, and optimal time-0 consumption of agent 2, c_0^2, for the two models:

```
# Print optimal k
kk_1 = mdl1.opt_k()
kk_2 = mdl2.opt_k()

print('The optimal k for model 1: {:.5f}'.format(kk_1))
print('The optimal k for model 2: {:.5f}'.format(kk_2))

# Print optimal time-0 consumption for agent 2
c20_1 = mdl1.opt_c(k=kk_1)[1]
c20_2 = mdl2.opt_c(k=kk_2)[1]

print('The optimal c20 for model 1: {:.5f}'.format(c20_1))
print('The optimal c20 for model 2: {:.5f}'.format(c20_2))
```

```
The optimal k for model 1: 0.14235
The optimal k for model 2: 0.13791
```

```
The optimal c20 for model 1: 0.90205
The optimal c20 for model 2: 0.92862
```

2nd example

In the second example, we illustrate how the optimal choice of k is influenced by the correlation parameter χ_i.

We will need to install the `plotly` package for 3D illustration. See https://plotly.com/python/getting-started/ for further instructions.

```
# Mesh grid of χ
N = 30
χ1grid, χ2grid = np.meshgrid(np.linspace(-1,1,N),
                             np.linspace(-1,1,N))
```

(continues on next page)

(continued from previous page)

```
k_foc = k_foc_factory(mdl1)

# Create grid for k
kgrid = np.zeros_like(χ1grid)

w0 = mdl1.w0

@njit(parallel=True)
def fill_k_grid(kgrid):
    # Loop: Compute optimal k and
    for i in prange(N):
        for j in prange(N):
            X1 = χ1grid[i, j]
            X2 = χ2grid[i, j]
            k = root_finding.newton_secant(k_foc, 1e-2, args=(X1, X2)).root
            kgrid[i, j] = k
```

```
%%time
fill_k_grid(kgrid)
```

```
CPU times: user 4.69 s, sys: 117 ms, total: 4.81 s
Wall time: 4.81 s
```

```
%%time
# Second-run
fill_k_grid(kgrid)
```

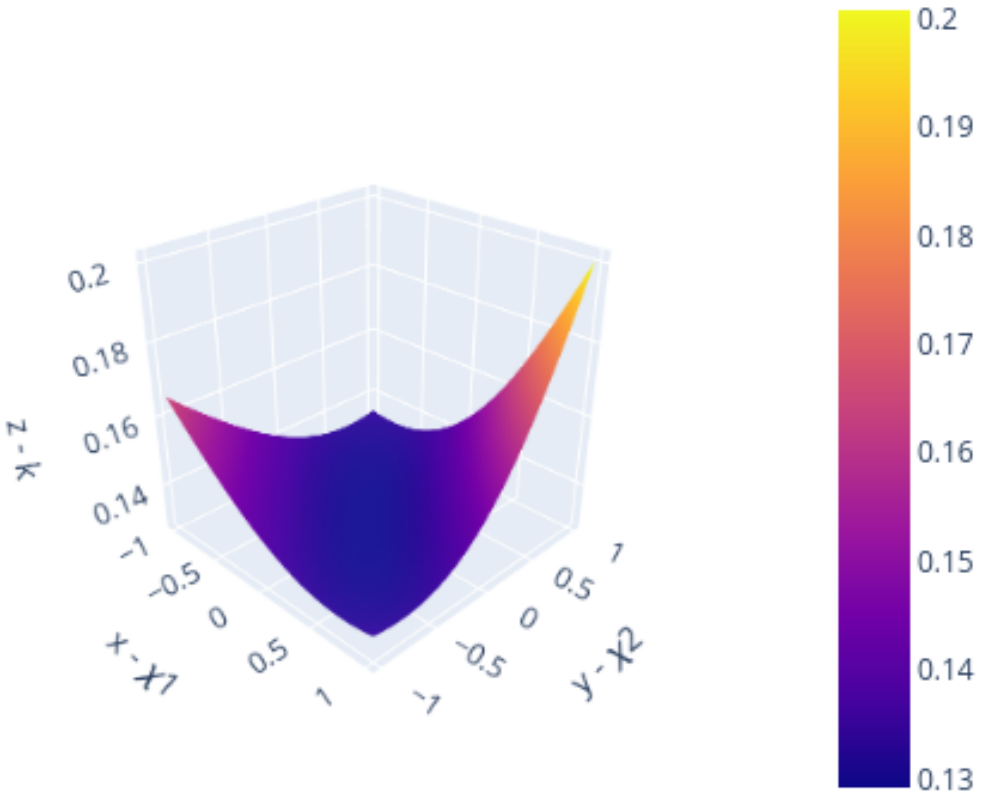
```
CPU times: user 8.08 ms, sys: 0 ns, total: 8.08 ms
Wall time: 2.09 ms
```

```
#=== Example: Plot optimal k with different correlations ===#

from IPython.display import Image
# Import plotly
import plotly.graph_objs as go

# Plot optimal k
fig = go.Figure(data=[go.Surface(x=χ1grid, y=χ2grid, z=kgrid)])
fig.update_layout(scene = dict(xaxis_title='x - χ1',
                               yaxis_title='y - χ2',
                               zaxis_title='z - k',
                               aspectratio=dict(x=1,y=1,z=1)))
fig.update_layout(width=500,
                  height=500,
                  margin=dict(l=50, r=50, b=65, t=90))
fig.update_layout(scene_camera=dict(eye=dict(x=2, y=-2, z=1.5)))

# Export to PNG file
Image(fig.to_image(format="png"))
# fig.show() will provide interactive plot when running
# notebook locally
```


0.2
0.18
0.16
0.14
z - x
-1
-0.5
0
0.5
1
x - x1
-1
-0.5
0
0.5
1
y - x2
0.2
0.19
0.18
0.17
0.16
0.15
0.14
0.13

CHAPTER

THIRTYNINE

EQUILIBRIUM CAPITAL STRUCTURES WITH INCOMPLETE MARKETS

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install --upgrade quantecon
!conda install -y -c plotly plotly plotly-orca
```

39.1 Introduction

This is an extension of an earlier lecture *Irrelevance of Capital Structure with Complete Markets* about a **complete markets** model.

In contrast to that lecture, this one describes an instance of a model authored by Bisin, Clementi, and Gottardi [Bisin *et al.*, 2018] in which financial markets are **incomplete**.

Instead of being able to trade equities and a full set of one-period Arrow securities as they can in *Irrelevance of Capital Structure with Complete Markets*, here consumers and firms trade only equity and a bond.

It is useful to watch how outcomes differ in the two settings.

In the complete markets economy in *Irrelevance of Capital Structure with Complete Markets*

- there is a unique stochastic discount factor that prices all assets
- consumers' portfolio choices are indeterminate
- firms' financial structures are indeterminate, so the model embodies an instance of a Modigliani-Miller irrelevance theorem [Modigliani and Miller, 1958]
- the aggregate of all firms' financial structures are indeterminate, a consequence of there being redundant assets

In the incomplete markets economy studied here

- there is a not a unique equilibrium stochastic discount factor
- different stochastic discount factors price different assets
- consumers' portfolio choices are determinate
- while **individual** firms' financial structures are indeterminate, thus conforming to part of a Modigliani-Miller theorem, [Modigliani and Miller, 1958], the **aggregate** of all firms' financial structures **is** determinate.

A `Big K, little k` analysis played an important role in the previous lecture *Irrelevance of Capital Structure with Complete Markets*.

A more subtle version of a `Big K, little k` features in the BCG incomplete markets environment here.

We use it to convey the heart of what BCG call a **rational conjectures** equilibrium in which conjectures are about equilibrium pricing functions in regions of the state space that an average consumer or firm does not visit in equilibrium.

Note that the absence of complete markets means that now we cannot compute competitive equilibrium prices and allocations by first solving the simple planning problem that we did in *Irrelevance of Capital Structure with Complete Markets*.

Instead, we compute an equilibrium by solving a system of simultaneous inequalities.

(Here we do not address the interesting question of whether there is a *different* planning problem that we could use to compute a competitive equlibrium allocation.)

39.1.1 Setup

We adopt specifications of preferences and technologies used by Bisin, Clemente, and Gottardi (2018) [Bisin *et al.*, 2018] and in our earlier lecture on a complete markets version of their model.

The economy lasts for two periods, $t = 0, 1$.

There are two types of consumers named $i = 1, 2$.

A scalar random variable ϵ affects both

- a representative firm's physical return $f(k)e^{\epsilon}$ in period 1 from investing $k \geq 0$ in capital in period 0.
- period 1 endowments $w_1^i(\epsilon)$ of the consumption good for agents $i = 1$ and $i = 2$.

39.1.2 Ownership

A consumer of type i is endowed with w_0^i units of the time 0 good and $w_1^i(\epsilon)$ of the time 1 good when the random variable takes value ϵ.

At the start of period 0, a consumer of type i also owns θ_0^i shares of a representative firm.

39.1.3 Measures of agents and firms

As in the companion lecture *Irrelevance of Capital Structure with Complete Markets* that studies a complete markets version of the model, we follow BCG in assuming that there are unit measures of

- consumers of type $i = 1$
- consumers of type $i = 2$
- firms with access to a production technology that converts k units of time 0 good into $Ak^{\alpha}e^{\epsilon}$ units of the time 1 good in random state ϵ

Thus, let $\omega \in [0, 1]$ index a particular consumer of type i.

Then define Big C^i as

$$C^i = \int_0^1 c^i(\omega) d\,\omega$$

with components

$$C_0^i = \int_0^1 c_0^i(\omega) d\,\omega$$

$$C_1^i(\epsilon) = \int_0^1 c_1^i(\epsilon; \omega) d\,\omega$$

In the same spirit, let $\zeta \in [0,1]$ index a particular firm and let firm ζ purchase $k(\zeta)$ units of capital and issue $b(\zeta)$ bonds.

Then define Big K and Big B as

$$K = \int_0^1 k(\zeta) d\,\zeta, \quad B = \int_0^1 b(\zeta) d\,\zeta$$

The assumption that there are equal measures of our three types of agents justifies our assumption that each individual agent is a powerless **price taker**:

- an individual consumer chooses its own (infinitesimal) part $c^i(\omega)$ of C^i taking prices as given
- an individual firm chooses its own (infinitesmimal) part $k(\zeta)$ of K and $b(\zeta)$ of B taking pricing functions as given
- However, equilibrium prices depend on the `Big K, Big B, Big C` objects K, B, and C

The assumption about measures of agents is a powerful device for making a host of competitive agents take as given the equilibrium prices that turn out to be determined by the decisions of hosts of agents who are just like them.

We call an equilibrium **symmetric** if

- all type i consumers choose the same consumption profiles so that $c^i(\omega) = C^i$ for all $\omega \in [0,1]$
- all firms choose the same levels of k and b so that $k(\zeta) = K$, $b(\zeta) = B$ for all $\zeta \in [0,1]$

In this lecture, we restrict ourselves to describing symmetric equilibria.

39.1.4 Endowments

Per capital economy-wide endowments in periods 0 and 1 are

$$\begin{aligned} w_0 &= w_0^1 + w_0^2 \\ w_1(\epsilon) &= w_1^1(\epsilon) + w_1^2(\epsilon) \text{ in state } \epsilon \end{aligned}$$

39.1.5 Feasibility:

Where $\alpha \in (0,1)$ and $A > 0$

$$\begin{aligned} C_0^1 + C_0^2 &= w_0^1 + w_0^2 - K \\ C_1^1(\epsilon) + C_1^2(\epsilon) &= w_1^1(\epsilon) + w_1^2(\epsilon) + e^\epsilon \int_0^1 f(k(\zeta)) d\zeta, \quad k \geq 0 \end{aligned}$$

where $f(k) = Ak^\alpha, A > 0, \alpha \in (0,1)$.

39.1.6 Parameterizations

Following BCG, we shall employ the following parameterizations:

$$\begin{aligned} \epsilon &\sim \mathcal{N}(\mu, \sigma^2) \\ u(c) &= \frac{c^{1-\gamma}}{1-\gamma} \\ w_1^i(\epsilon) &= e^{-\chi_i \mu - .5\chi_i^2\sigma^2 + \chi_i \epsilon}, \quad \chi_i \in [0,1] \end{aligned}$$

Sometimes instead of asuming $\epsilon \sim g(\epsilon) = \mathcal{N}(0, \sigma^2)$, we'll assume that $g(\cdot)$ is a probability mass function that serves as a discrete approximation to a standardized normal density.

39.1.7 Preferences:

A consumer of type i orders period 0 consumption c_0^i and state ϵ-period 1 consumption $c^i(\epsilon)$ by

$$u^i = u(c_0^i) + \beta \int u(c_1^i(\epsilon)) g(\epsilon) d\epsilon, \quad i = 1, 2$$

$\beta \in (0, 1)$ and the one-period utility function is

$$u(c) = \begin{cases} \frac{c^{1-\gamma}}{1-\gamma} & \text{if } \gamma \neq 1 \\ \log c & \text{if } \gamma = 1 \end{cases}$$

39.1.8 Risk-sharing motives

The two types of agents' period 1 endowments have different correlations with the physical return on capital.

Endowment differences give agents incentives to trade risks that in the complete market version of the model showed up in their demands for equity and in their demands and supplies of one-period Arrow securities.

In the incomplete-markets setting under study here, these differences show up in differences in the two types of consumers' demands for a typical firm's bonds and equity, the only two assets that agents can now trade.

39.2 Asset Markets

Markets are incomplete: *ex cathedra* we the model builders declare that only equities and bonds issued by representative firms can be traded.

Let θ^i and ξ^i be a consumer of type i's post-trade holdings of equity and bonds, respectively.

A firm issues bonds promising to pay b units of consumption at time $t = 1$ and purchases k units of physical capital at time $t = 0$.

When $e^\epsilon A k^\alpha < b$ at time 1, the firm defaults and its output is divided equally among bondholders.

Evidently, when the productivity shock $\epsilon < \epsilon^* = \log\left(\frac{b}{Ak^\alpha}\right)$, the firm defaults on its debt

Payoffs to equity and debt at date 1 as functions of the productivity shock ϵ are thus

$$\begin{aligned} d^e(k, b; \epsilon) &= \max\left\{e^\epsilon A k^\alpha - b, 0\right\} \\ d^b(k, b; \epsilon) &= \min\left\{\frac{e^\epsilon A k^\alpha}{b}, 1\right\} \end{aligned} \tag{39.1}$$

A firm faces a bond price function $p(k, b)$ when it issues b bonds and purchases k units of physical capital.

A firm's equity is worth $q(k, b)$ when it issues b bonds and purchases k units of physical capital.

A firm regards an equity-pricing function $q(k, b)$ and a bond pricing function $p(k, b)$ as exogenous in the sense that they are not affected by its choices of k and b.

Consumers face equilibrium prices $\check{q}$ and $\check{p}$ for bonds and equities, where $\check{q}$ and $\check{p}$ are both scalars.

Consumers are price takers and only need to know the scalars $\check{q}, \check{p}$.

Firms are *price function* takers and must know the functions $q(k, b), p(k, b)$ in order completely to pose their optimum problems.

39.2.1 Consumers

Each consumer of type i is endowed with w_0^i of the time 0 consumption good, $w_1^i(\epsilon)$ of the time 1, state ϵ consumption good and also owns a fraction $\theta_0^i \in (0,1)$ of the initial value of a representative firm, where $\theta_0^1 + \theta_0^2 = 1$.

The initial value of a representative firm is V (an object to be determined in a rational expectations equilibrium).

Consumer i buys θ^i shares of equity and buys bonds worth $\check{p}\xi^i$ where $\check{p}$ is the bond price.

Being a price-taker, a consumer takes V, $\check{q}$, $\check{p}$, and K, B as given.

Consumers know that equilibrium payoff functions for bonds and equities take the form

$$d^e(K,B;\epsilon) = \max\{e^\epsilon AK^\alpha - B, 0\}$$
$$d^b(K,B;\epsilon) = \min\left\{\frac{e^\epsilon AK^\alpha}{B}, 1\right\}$$

Consumer i's optimization problem is

$$\begin{aligned}
\max_{c_0^i,\theta^i,\xi^i,c_1^i(\epsilon)} \quad & u(c_0^i) + \beta \int u(c^i(\epsilon))g(\epsilon)\, d\epsilon \\
\text{subject to} \quad & c_0^i = w_0^i + \theta_0^i V - \check{q}\theta^i - \check{p}\xi^i, \\
& c_1^i(\epsilon) = w_1^i(\epsilon) + \theta^i d^e(K,B;\epsilon) + \xi^i d^b(K,B;\epsilon) \;\forall\; \epsilon, \\
& \theta^i \geq 0, \xi^i \geq 0.
\end{aligned}$$

The last two inequalities impose that the consumer cannot short sell either equity or bonds.

In a rational expectations equilibrium, $\check{q} = q(K,B)$ and $\check{p} = p(K,B)$

We form consumer i's Lagrangian:

$$\begin{aligned}
L^i :=& u(c_0^i) + \beta \int u(c^i(\epsilon))g(\epsilon)\, d\epsilon \\
& + \lambda_0^i[w_0^i + \theta_0 V - \check{q}\theta^i - \check{p}\xi^i - c_0^i] \\
& + \beta \int \lambda_1^i(\epsilon)\left[w_1^i(\epsilon) + \theta^i d^e(K,B;\epsilon) + \xi^i d^b(K,B;\epsilon) - c_1^i(\epsilon)\right] g(\epsilon)\, d\epsilon
\end{aligned}$$

Consumer i's first-order necessary conditions for an optimum include:

$$\begin{aligned}
c_0^i: \quad & u'(c_0^i) = \lambda_0^i \\
c_1^i(\epsilon): \quad & u'(c_1^i(\epsilon)) = \lambda_1^i(\epsilon) \\
\theta^i: \quad & \beta \int \lambda_1^i(\epsilon) d^e(K,B;\epsilon) g(\epsilon)\, d\epsilon \leq \lambda_0^i \check{q} \quad (= \text{ if } \theta^i > 0) \\
\xi^i: \quad & \beta \int \lambda_1^i(\epsilon) d^b(K,B;\epsilon) g(\epsilon)\, d\epsilon \leq \lambda_0^i \check{p} \quad (= \text{ if } b^i > 0)
\end{aligned}$$

We can combine and rearrange consumer i's first-order conditions to become:

$$\check{q} \geq \beta \int \frac{u'(c_1^i(\epsilon))}{u'(c_0^i)} d^e(K,B;\epsilon) g(\epsilon)\, d\epsilon \quad (= \text{ if } \theta^i > 0)$$
$$\check{p} \geq \beta \int \frac{u'(c_1^i(\epsilon))}{u'(c_0^i)} d^b(K,B;\epsilon) g(\epsilon)\, d\epsilon \quad (= \text{ if } b^i > 0)$$

These inequalities imply that in a symmetric rational expectations equilibrium consumption allocations and prices satisfy

$$\check{q} = \max_i \beta \int \frac{u'(c_1^i(\epsilon))}{u'(c_0^i)} d^e(K,B;\epsilon) g(\epsilon)\, d\epsilon$$
$$\check{p} = \max_i \beta \int \frac{u'(c_1^i(\epsilon))}{u'(c_0^i)} d^b(K,B;\epsilon) g(\epsilon)\, d\epsilon$$

39.2.2 Pricing functions

When individual firms solve their optimization problems, they take big C^i's as fixed objects that they don't influence.

A representative firm faces a price function $q(k, b)$ for its equity and a price function $p(k, b)$ per unit of bonds that satisfy

$$q(k,b) = \max_i \beta \int \frac{u'(C_1^i(\epsilon))}{u'(C_0^i)} d^e(k,b;\epsilon) g(\epsilon)\, d\epsilon$$
$$p(k,b) = \max_i \beta \int \frac{u'(C_1^i(\epsilon))}{u'(C_0^i)} d^b(k,b;\epsilon) g(\epsilon)\, d\epsilon$$

where the payoff functions are described by equations (39.1).

Notice the appearance of big C^i's on the right sides of these two equations that define equilibrium pricing functions.

The two price functions describe outcomes not only for equilibrium choices K, B of capital k and debt b, but also for any **out-of-equilibrium** pairs $(k, b) \neq (K, B)$.

The firm is assumed to know both price functions.

This means that the firm understands that its choice of k, b influences how markets price its equity and debt.

This package of assumptions is sometimes called **rational conjectures** (about price functions).

BCG give credit to Makowski for emphasizing and clarifying how rational conjectures are components of rational expectations equilibria.

39.2.3 Firms

The firm chooses capital k and debt b to maximize its market value:

$$V \equiv \max_{k,b} -k + q(k,b) + p(k,b)b$$

Attributing value maximization to the firm is a good idea because in equilibrium consumers of both types *want* a firm to maximize its value.

In the special quantitative examples studied here

- consumers of types $i = 1, 2$ both hold equity
- only consumers of type $i = 2$ hold debt; consumers of type $i = 1$ hold none.

These outcomes occur because we follow BCG and set parameters so that a type 2 consumer's stochastic endowment of the consumption good in period 1 is more correlated with the firm's output than is a type 1 consumer's.

This gives consumers of type 2 a motive to hedge their second period endowment risk by holding bonds (they also choose to hold some equity).

These outcomes mean that the pricing functions end up satisfying

$$q(k,b) = \beta \int \frac{u'(C_1^1(\epsilon))}{u'(C_0^1)} d^e(k,b;\epsilon) g(\epsilon)\, d\epsilon = \beta \int \frac{u'(C_1^2(\epsilon))}{u'(C_0^2)} d^e(k,b;\epsilon) g(\epsilon)\, d\epsilon$$
$$p(k,b) = \beta \int \frac{u'(C_1^2(\epsilon))}{u'(C_0^2)} d^b(k,b;\epsilon) g(\epsilon)\, d\epsilon$$

Recall that $\epsilon^*(k, b) \equiv \log\left(\frac{b}{Ak^\alpha}\right)$ is a firm's default threshold.

We can rewrite the pricing functions as:

$$q(k,b) = \beta \int_{\epsilon^*}^{\infty} \frac{u'(C_1^i(\epsilon))}{u'(C_0^i)} \left(e^{\epsilon} A k^{\alpha} - b\right) g(\epsilon)\, d\epsilon, \quad i = 1, 2$$

$$p(k,b) = \beta \int_{-\infty}^{\epsilon^*} \frac{u'(C_1^2(\epsilon))}{u'(C_0^2)} \left(\frac{e^{\epsilon} A k^{\alpha}}{b}\right) g(\epsilon)\, d\epsilon + \beta \int_{\epsilon^*}^{\infty} \frac{u'(C_1^2(\epsilon))}{u'(C_0^2)} g(\epsilon)\, d\epsilon$$

Firm's optimization problem

The firm's optimization problem is

$$V \equiv \max_{k,b} \{-k + q(k,b) + p(k,b)b\}$$

The firm's first-order necessary conditions with respect to k and b, respectively, are

$$\begin{aligned} k: \quad & -1 + \frac{\partial q(k,b)}{\partial k} + b\frac{\partial p(q,b)}{\partial k} = 0 \\ b: \quad & \frac{\partial q(k,b)}{\partial b} + p(k,b) + b\frac{\partial p(k,b)}{\partial b} = 0 \end{aligned}$$

We use the Leibniz integral rule several times to arrive at the following derivatives:

$$\frac{\partial q(k,b)}{\partial k} = \beta\alpha A k^{\alpha-1} \int_{\epsilon^*}^{\infty} \frac{u'(C_1^i(\epsilon))}{u'(C_0^i)} e^{\epsilon} g(\epsilon) d\epsilon, \quad i = 1, 2$$

$$\frac{\partial q(k,b)}{\partial b} = -\beta \int_{\epsilon^*}^{\infty} \frac{u'(C_1^i(\epsilon))}{u'(C_0^i)} g(\epsilon) d\epsilon, \quad i = 1, 2$$

$$\frac{\partial p(k,b)}{\partial k} = \beta\alpha \frac{A k^{\alpha-1}}{b} \int_{-\infty}^{\epsilon^*} \frac{u'(C_1^2(\epsilon))}{u'(C_0^2)} g(\epsilon) d\epsilon$$

$$\frac{\partial p(k,b)}{\partial b} = -\beta \frac{A k^{\alpha}}{b^2} \int_{-\infty}^{\epsilon^*} \frac{u'(C_1^2(\epsilon))}{u'(C_0^2)} e^{\epsilon} g(\epsilon) d\epsilon$$

Special case: We confine ourselves to a special case in which both types of consumer hold positive equities so that $\frac{\partial q(k,b)}{\partial k}$ and $\frac{\partial q(k,b)}{\partial b}$ are related to rates of intertemporal substitution for both agents.

Substituting these partial derivatives into the above first-order conditions for k and b, respectively, we obtain the following versions of those first order conditions:

$$k: \quad -1 + \beta\alpha A k^{\alpha-1} \int_{-\infty}^{\infty} \frac{u'(C_1^2(\epsilon))}{u'(C_0^2)} e^{\epsilon} g(\epsilon) d\epsilon = 0 \tag{39.2}$$

$$b: \quad \int_{\epsilon^*}^{\infty} \left(\frac{u'(C_1^1(\epsilon))}{u'(C_0^1)}\right) g(\epsilon)\, d\epsilon = \int_{\epsilon^*}^{\infty} \left(\frac{u'(C_1^2(\epsilon))}{u'(C_0^2)}\right) g(\epsilon)\, d\epsilon \tag{39.3}$$

where again recall that $\epsilon^*(k,b) \equiv \log\left(\frac{b}{Ak^{\alpha}}\right)$.

Taking $C_0^i, C_1^i(\epsilon)$ as given, these are two equations that we want to solve for the firm's optimal decisions k, b.

39.3 Equilibrium verification

On page 5 of BCG (2018), the authors say

If the price conjectures corresponding to the plan chosen by firms in equilibrium are correct, that is equal to the market prices $\check{q}$ and $\check{p}$, it is immediate to verify that the rationality of the conjecture coincides with the agents' Euler equations.

Here BCG are describing how they go about verifying that when they set little k, little b from the firm's first-order conditions equal to the big K, big B at the big C's that appear in the pricing functions, then

- consumers' Euler equations are satisfied if little c's are equated to Big C's
- firms' first-order necessary conditions for k, b are satisfied.
- $\check{q} = q(K, B)$ and $\check{p} = p(K, B)$.

39.4 Pseudo Code

Before displaying our Python code for computing a BCG incomplete markets equilibrium, we'll sketch some pseudo code that describes its logical flow.

Here goes:

1. Set upper and lower bounds for firm value as V_h and V_l, for capital as k_h and k_l, and for debt as b_h and b_l.
2. Conjecture firm value $V = \frac{1}{2}(V_h + V_l)$
3. Conjecture debt level $b = \frac{1}{2}(b_h + b_l)$.
4. Conjecture capital $k = \frac{1}{2}(k_h + k_l)$.
5. Compute the default threshold $\epsilon^* \equiv \log\left(\frac{b}{Ak^\alpha}\right)$.
6. (In this step we abuse notation by freezing V, k, b and in effect temporarily treating them as Big K, B values. Thus, in this step 6 little k, b are frozen at guessed at value of K, B.) Fixing the values of V, b and k, compute optimal choices of consumption c^i with consumers' FOCs. Assume that only agent 2 holds debt: $\xi^2 = b$ and that both agents hold equity: $0 < \theta^i < 1$ for $i = 1, 2$.
7. Set high and low bounds for equity holdings for agent 1 as θ_h^1 and θ_l^1. Guess $\theta^1 = \frac{1}{2}(\theta_h^1 + \theta_l^1)$, and $\theta^2 = 1 - \theta^1$. While $|\theta_h^1 - \theta_l^1|$ is large:
   - Compute agent 1's valuation of the equity claim with a fixed-point iteration:

     $q_1 = \beta \int \frac{u'(c_1^1(\epsilon))}{u'(c_0^1)} d^e(k, b; \epsilon) g(\epsilon)\, d\epsilon$

     where

     $c_1^1(\epsilon) = w_1^1(\epsilon) + \theta^1 d^e(k, b; \epsilon)$

     and

     $c_0^1 = w_0^1 + \theta_0^1 V - q_1 \theta^1$
   - Compute agent 2's valuation of the bond claim with a fixed-point iteration:

     $p = \beta \int \frac{u'(c_1^2(\epsilon))}{u'(c_0^2)} d^b(k, b; \epsilon) g(\epsilon)\, d\epsilon$

     where

     $c_1^2(\epsilon) = w_1^2(\epsilon) + \theta^2 d^e(k, b; \epsilon) + b$

     and

$c_0^2 = w_0^2 + \theta_0^2 V - q_1 \theta^2 - pb$

- Compute agent 2's valuation of the equity claim with a fixed-point iteration:

  $q_2 = \beta \int \frac{u'(c_1^2(\epsilon))}{u'(c_0^2)} d^e(k, b; \epsilon) g(\epsilon)\, d\epsilon$

  where

  $c_1^2(\epsilon) = w_1^2(\epsilon) + \theta^2 d^e(k, b; \epsilon) + b$

  and

  $c_0^2 = w_0^2 + \theta_0^2 V - q_2 \theta^2 - pb$

- If $q_1 > q_2$, Set $\theta_l = \theta^1$; otherwise, set $\theta_h = \theta^1$.
- Repeat steps 6Aa through 6Ad until $|\theta_h^1 - \theta_l^1|$ is small.

8. Set bond price as p and equity price as $q = \max(q_1, q_2)$.
9. Compute optimal choices of consumption:

$$\begin{aligned} c_0^1 &= w_0^1 + \theta_0^1 V - q\theta^1 \\ c_0^2 &= w_0^2 + \theta_0^2 V - q\theta^2 - pb \\ c_1^1(\epsilon) &= w_1^1(\epsilon) + \theta^1 d^e(k, b; \epsilon) \\ c_1^2(\epsilon) &= w_1^2(\epsilon) + \theta^2 d^e(k, b; \epsilon) + b \end{aligned}$$

10. (Here we confess to abusing notation again, but now in a different way. In step 7, we interpret frozen c^is as Big C^i. We do this to solve the firm's problem.) Fixing the values of c_0^i and $c_1^i(\epsilon)$, compute optimal choices of capital k and debt level b using the firm's first order necessary conditions.
11. Compute deviations from the firm's FONC for capital k as:

    $kfoc = \beta\alpha A k^{\alpha-1} \left(\int \frac{u'(c_1^2(\epsilon))}{u'(c_0^2)} e^\epsilon g(\epsilon)\, d\epsilon \right) - 1$

    - If $kfoc > 0$, Set $k_l = k$; otherwise, set $k_h = k$.
    - Repeat steps 4 through 7A until $|k_h - k_l|$ is small.

12. Compute deviations from the firm's FONC for debt level b as:

    $bfoc = \beta \left[\int_{\epsilon^*}^{\infty} \left(\frac{u'(c_1^1(\epsilon))}{u'(c_0^1)} \right) g(\epsilon)\, d\epsilon - \int_{\epsilon^*}^{\infty} \left(\frac{u'(c_1^2(\epsilon))}{u'(c_0^2)} \right) g(\epsilon)\, d\epsilon \right]$

    - If $bfoc > 0$, Set $b_h = b$; otherwise, set $b_l = b$.
    - Repeat steps 3 through 7B until $|b_h - b_l|$ is small.

13. Given prices q and p from step 6, and the firm choices of k and b from step 7, compute the synthetic firm value:

    $V_x = -k + q + pb$

    - If $V_x > V$, then set $V_l = V$; otherwise, set $V_h = V$.
    - Repeat steps 1 through 8 until $|V_x - V|$ is small.

14. Ultimately, the algorithm returns equilibrium capital k^*, debt b^* and firm value V^*, as well as the following equilibrium values:

    - Equity holdings $\theta^{1,*} = \theta^1(k^*, b^*)$
    - Prices $q^* = q(k^*, b^*),\ p^* = p(k^*, b^*)$
    - Consumption plans $C_0^{1,*} = c_0^1(k^*, b^*),\ C_0^{2,*} = c_0^2(k^*, b^*),\ C_1^{1,*}(\epsilon) = c_1^1(k^*, b^*; \epsilon),\ C_1^{1,*}(\epsilon) = c_1^2(k^*, b^*; \epsilon)$.

39.5 Code

We create a Python class `BCG_incomplete_markets` to compute the equilibrium allocations of the incomplete market BCG model, given a set of parameter values.

The class includes the following methods, i.e., functions:

- `solve_eq`: solves the BCG model and returns the equilibrium values of capital k, debt b and firm value V, as well as
  - agent 1's equity holdings $\theta^{1,*}$
  - prices q^*, p^*
  - consumption plans $C_0^{1,*}, C_0^{2,*}, C_1^{1,*}(\epsilon), C_1^{2,*}(\epsilon)$.
- `eq_valuation`: inputs equilibrium consumpion plans C^* and outputs the following valuations for each pair of (k, b) in the grid:
  - the firm $V(k, b)$
  - the equity $q(k, b)$
  - the bond $p(k, b)$.

Parameters include:

- χ_1, χ_2: correlation parameter for agent 1 and 2. Default values are respectively 0 and 0.9.
- w_0^1, w_0^2: initial endowments. Default values are respectively 0.9 and 1.1.
- θ_0^1, θ_0^2: initial holding of the firm. Default values are 0.5.
- ψ: risk parameter. Default value is 3.
- α: Production function parameter. Default value is 0.6.
- A: Productivity of the firm. Default value is 2.5.
- μ, σ: Mean and standard deviation of the shock distribution. Default values are respectively -0.025 and 0.4
- β: Discount factor. Default value is 0.96.
- bound: Bound for truncated normal distribution. Default value is 3.

```
import numpy as np
from scipy.stats import truncnorm
from scipy.integrate import quad
from numba import njit
```

```
class BCG_incomplete_markets:

    # init method or constructor
    def __init__(self,
                 χ1 = 0,
                 χ2 = 0.9,
                 w10 = 0.9,
                 w20 = 1.1,
                 θ10 = 0.5,
                 θ20 = 0.5,
                 ψ1 = 3,
                 ψ2 = 3,
                 α = 0.6,
```

(continues on next page)

(continued from previous page)

```
                A = 2.5,
                μ = -0.025,
                σ = 0.4,
                β = 0.96,
                bound = 3,
                Vl = 0,
                Vh = 0.5,
                kbot = 0.01,
                #ktop = (β*A)**(1/(1-α)),
                ktop = 0.25,
                bbot = 0.1,
                btop = 0.8):

        #=========== Setup ===========#
        # Risk parameters
        self.χ1 = χ1
        self.χ2 = χ2

        # Other parameters
        self.ψ1 = ψ1
        self.ψ2 = ψ2
        self.α = α
        self.A = A
        self.μ = μ
        self.σ = σ
        self.β = β
        self.bound = bound

        # Bounds for firm value, capital, and debt
        self.Vl = Vl
        self.Vh = Vh
        self.kbot = kbot
        #self.kbot = (β*A)**(1/(1-α))
        self.ktop = ktop
        self.bbot = bbot
        self.btop = btop

        # Utility
        self.u = njit(lambda c: (c**(1-ψ)) / (1-ψ))

        # Initial endowments
        self.w10 = w10
        self.w20 = w20
        self.w0 = w10 + w20

        # Initial holdings
        self.θ10 = θ10
        self.θ20 = θ20

        # Endowments at t=1
        self.w11 = njit(lambda ϵ: np.exp(-χ1*μ - 0.5*(χ1**2)*(σ**2) + χ1*ϵ))
        self.w21 = njit(lambda ϵ: np.exp(-χ2*μ - 0.5*(χ2**2)*(σ**2) + χ2*ϵ))
        self.w1 = njit(lambda ϵ: self.w11(ϵ) + self.w21(ϵ))

        # Truncated normal
        ta, tb = (-bound - μ) / σ, (bound - μ) / σ
```

(continues on next page)

(continued from previous page)

```
        rv = truncnorm(ta, tb, loc=μ, scale=σ)
        ϵ_range = np.linspace(ta, tb, 1000000)
        pdf_range = rv.pdf(ϵ_range)
        self.g = njit(lambda ϵ: np.interp(ϵ, ϵ_range, pdf_range))


    #*************************************************************
    # Function: Solve for equilibrium of the BCG model
    #*************************************************************
    def solve_eq(self, print_crit=True):

        # Load parameters
        ψ1 = self.ψ1
        ψ2 = self.ψ2
        α = self.α
        A = self.A
        β = self.β
        bound = self.bound
        Vl = self.Vl
        Vh = self.Vh
        kbot = self.kbot
        ktop = self.ktop
        bbot = self.bbot
        btop = self.btop
        w10 = self.w10
        w20 = self.w20
        θ10 = self.θ10
        θ20 = self.θ20
        w11 = self.w11
        w21 = self.w21
        g = self.g

        # We need to find a fixed point on the value of the firm
        V_crit = 1

        Y = njit(lambda ϵ, fk: np.exp(ϵ)*fk)
        intqq1 = njit(lambda ϵ, fk, θ1, ψ1, b: (w11(ϵ) + θ1*(Y(ϵ, fk) - b))**(-
↪ψ1)*(Y(ϵ, fk) - b)*g(ϵ))
        intp1 = njit(lambda ϵ, fk, ψ2, b: (Y(ϵ, fk)/b)*(w21(ϵ) + Y(ϵ, fk))**(-
↪ψ2)*g(ϵ))
        intp2 = njit(lambda ϵ, fk, θ2, ψ2, b: (w21(ϵ) + θ2*(Y(ϵ, fk)-b) + b)**(-
↪ψ2)*g(ϵ))
        intqq2 = njit(lambda ϵ, fk, θ2, ψ2, b: (w21(ϵ) + θ2*(Y(ϵ, fk)-b) + b)**(-
↪ψ2)*(Y(ϵ, fk) - b)*g(ϵ))
        intk1 = njit(lambda ϵ, fk, ψ2: (w21(ϵ) + Y(ϵ, fk))**(-ψ2)*np.exp(ϵ)*g(ϵ))
        intk2 = njit(lambda ϵ, fk, θ2, ψ2, b: (w21(ϵ) + θ2*(Y(ϵ, fk)-b) + b)**(-
↪ψ2)*np.exp(ϵ)*g(ϵ))
        intB1 = njit(lambda ϵ, fk, θ1, ψ1, b: (w11(ϵ) + θ1*(Y(ϵ, fk) - b))**(-
↪ψ1)*g(ϵ))
        intB2 = njit(lambda ϵ, fk, θ2, ψ2, b: (w21(ϵ) + θ2*(Y(ϵ, fk) - b) + b)**(-
↪ψ2)*g(ϵ))

        while V_crit>1e-4:

            # We begin by adding the guess for the value of the firm to endowment
            V = (Vl+Vh)/2
```

(continues on next page)

(continued from previous page)

```
            ww10 = w10 + θ10*V
            ww20 = w20 + θ20*V

            # Figure out the optimal level of debt
            bl = bbot
            bh = btop
            b_crit=1

            while b_crit>1e-5:

                # Setting the conjecture for debt
                b = (bl+bh)/2

                # Figure out the optimal level of capital
                kl = kbot
                kh = ktop
                k_crit=1

                while k_crit>1e-5:

                    # Setting the conjecture for capital
                    k = (kl+kh)/2

                    # Production
                    fk = A*(k**α)
#                     Y = lambda ξ: np.exp(ξ)*fk

                    # Compute integration threshold
                    epstar = np.log(b/fk)


                    #*************************************************************
                    # Compute the prices and allocations consistent with consumers'
                    # Euler equations
                    #*************************************************************

                    # We impose the following:
                    # Agent 1 buys equity
                    # Agent 2 buys equity and all debt
                    # Agents trade such that prices converge

                    #========
                    # Agent 1
                    #========
                    # Holdings
                    ξ1 = 0
                    θ1a = 0.3
                    θ1b = 1

                    while abs(θ1b - θ1a) > 0.001:

                        θ1 = (θ1a + θ1b) / 2

                        # qq1 is the equity price consistent with agent-1 Euler␣
↪Equation
                        ## Note: Price is in the date-0 budget constraint of the agent
```

(continues on next page)

(continued from previous page)

```
                        ## First, compute the constant term that is not influenced by␣
 ↪q
                        ## that is, ∫E[u'(c^{1}_{1})d^{e}(k,B)]
#                         intqq1 = lambda ω: (w11(ω) + ξ1*(Y(ω, fk) - b))**(-ψ1)*(Y(ω,
 ↪ fk) - b)*g(ω)
#                         const_qq1 = β * quad(intqq1,epstar,bound)[0]

                        const_qq1 = β * quad(intqq1,epstar,bound, args=(fk, θ1, ψ1,␣
 ↪b))[0]



                        ## Second, iterate to get the equity price q
                        qq1l = 0
                        qq1h = ww10
                        diff = 1
                        while diff > 1e-7:
                            qq1 = (qq1l+qq1h)/2
                            rhs = const_qq1/((ww10-qq1*θ1)**(-ψ1));
                            if (rhs > qq1):
                                qq1l = qq1
                            else:
                                qq1h = qq1
                            diff = abs(qq1l-qq1h)

                        #========
                        # Agent 2
                        #========
                        ξ2 = b - ξ1
                        θ2 = 1 - θ1

                        # p is the bond price consistent with agent-2 Euler Equation
                        ## Note: Price is in the date-0 budget constraint of the agent

                        ## First, compute the constant term that is not influenced by␣
 ↪p
                        ## that is, ∫E[u'(c^{2}_{1})d^{b}(k,B)]
#                         intp1 = lambda ω: (Y(ω, fk)/b)*(w21(ω) + Y(ω, fk))**(-
 ↪ω2)*g(ω)
#                         intp2 = lambda ω: (w21(ω) + ξ2*(Y(ω, fk)-b) + b)**(-ω2)*g(ω)
#                         const_p = β * (quad(intp1,-bound,epstar)[0] + quad(intp2,
 ↪epstar,bound)[0])
                        const_p = β * (quad(intp1,-bound,epstar, args=(fk, ψ2, b))[0]\
                                       + quad(intp2,epstar,bound, args=(fk, θ2, ψ2,␣
 ↪b))[0])

                        ## iterate to get the bond price p
                        pl = 0
                        ph = ww20/b
                        diff = 1
                        while diff > 1e-7:
                            p = (pl+ph)/2
                            rhs = const_p/((ww20-qq1*θ2-p*b)**(-ψ2))
                            if (rhs > p):
                                pl = p
                            else:
```

(continues on next page)

(continued from previous page)

```
                                ph = p
                            diff = abs(pl-ph)

                        # qq2 is the equity price consistent with agent-2 Euler␣
 ↪Equation
#                         intqq2 = lambda ϵ: (w21(ϵ) + θ2*(Y(ϵ, fk)-b) + b)**(-
 ↪θ2)*(Y(ϵ, fk) - b)*g(ϵ)
                        const_qq2 = β * quad(intqq2,epstar,bound, args=(fk, θ2, ψ2,␣
 ↪b))[0]
                        qq2l = 0
                        qq2h = ww20
                        diff = 1
                        while diff > 1e-7:
                            qq2 = (qq2l+qq2h)/2
                            rhs = const_qq2/((ww20-qq2*θ2-p*b)**(-ψ2));
                            if (rhs > qq2):
                                qq2l = qq2
                            else:
                                qq2h = qq2
                            diff = abs(qq2l-qq2h)

                        # q be the maximum valuation for the equity among agents
                        ## This will be the equity price based on Makowski's criterion
                        q = max(qq1,qq2)

                        #================
                        # Update holdings
                        #================
                        if qq1 > qq2:
                            θ1a = θ1
                        else:
                            θ1b = θ1

                    #================
                    # Get consumption
                    #================
                    c10 = ww10 - q*θ1
                    c11 = lambda ϵ: w11(ϵ) + θ1*max(Y(ϵ, fk)-b,0)
                    c20 = ww20 - q*(1-θ1) - p*b
                    c21 = lambda ϵ: w21(ϵ) + (1-θ1)*max(Y(ϵ, fk)-b,0) + min(Y(ϵ, fk),
 ↪b)



                    #************************************************
                    # Compute the first order conditions for the firm
                    #************************************************

                    #===========
                    # Equity FOC
                    #===========
                    # Only agent 2's IMRS is relevent
#                     intk1 = lambda ϵ: (w21(ϵ) + Y(ϵ, fk))**(-θ2)*np.exp(ϵ)*g(ϵ)
#                     intk2 = lambda ϵ: (w21(ϵ) + θ2*(Y(ϵ, fk)-b) + b)**(-θ2)*np.
 ↪exp(ϵ)*g(ϵ)
#                     kfoc_num = quad(intk1,-bound,epstar)[0] + quad(intk2,epstar,
 ↪bound)[0]
```

(continues on next page)

(continued from previous page)

```
                    kfoc_num = quad(intk1,-bound,epstar, args=(fk, ψ2))[0] +␣
↪quad(intk2,epstar,bound, args=(fk, θ2, ψ2, b))[0]
                    kfoc_denom = (ww20- q*θ2 - p*b)**(-ψ2)
                    kfoc = β*α*A*(k**(α-1))*(kfoc_num/kfoc_denom) - 1

                    if (kfoc > 0):
                        kl = k
                    else:
                        kh = k
                    k_crit = abs(kh-kl)

                    if print_crit:
                        print("critical value of k: {:.5f}".format(k_crit))


                #=========
                # Bond FOC
                #=========
#                 intB1 = lambda ℓ: (w11(ℓ) + ℓ1*(Y(ℓ, fk) - b))**(-ℓ1)*g(ℓ)
#                 intB2 = lambda ℓ: (w21(ℓ) + ℓ2*(Y(ℓ, fk) - b) + b)**(-ℓ2)*g(ℓ)

#                 bfoc1 = quad(intB1,epstar,bound)[0] / (ww10 - q*ℓ1)**(-ℓ1)
#                 bfoc2 = quad(intB2,epstar,bound)[0] / (ww20 - q*ℓ2 - p*b)**(-ℓ2)

                bfoc1 = quad(intB1,epstar,bound, args=(fk, θ1, ψ1, b))[0] / (ww10 -␣
↪q*θ1)**(-ψ1)
                bfoc2 = quad(intB2,epstar,bound, args=(fk, θ2, ψ2, b))[0] / (ww20 -␣
↪q*θ2 - p*b)**(-ψ2)
                bfoc = bfoc1 - bfoc2

                if (bfoc > 0):
                    bh = b
                else:
                    bl = b
                b_crit = abs(bh-bl)

                if print_crit:
                    print("#=== critical value of b: {:.5f}".format(b_crit))

            # Compute the value of the firm
            value_x = -k + q + p*b
            if (value_x > V):
                 Vl = V
            else:
                 Vh = V
            V_crit = abs(value_x-V)

            if print_crit:
                print("#====== critical value of V: {:.5f}".format(V_crit))

            print('k,b,p,q,kfoc,bfoc,epstar,V,V_crit')
            formattedList = ["%.3f" % member for member in [k,
                                                            b,
                                                            p,
                                                            q,
                                                            kfoc,
```

(continues on next page)

(continued from previous page)

```
                                                        bfoc,
                                                        epstar,
                                                        V,
                                                        V_crit]]
            print(formattedList)

        #*********************************
        # Equilibrium values
        #*********************************

        # Return the results
        kss = k
        bss = b
        Vss = V
        qss = q
        pss = p
        c10ss = c10
        c11ss = c11
        c20ss = c20
        c21ss = c21
        θ1ss = θ1


        # Print the results
        print('finished')
       # print('k,b,p,q,kfoc,bfoc,epstar,V,V_crit')
        #formattedList = ["%.3f" % member for member in [kss,
        #                                                bss,
        #                                                pss,
        #                                                qss,
        #                                                kfoc,
        #                                                bfoc,
        #                                                epstar,
        #                                                Vss,
        #                                                V_crit]]
        #print(formattedList)

        return kss,bss,Vss,qss,pss,c10ss,c11ss,c20ss,c21ss,θ1ss


    #************************************************************
    # Function: Equity and bond valuations by different agents
    #************************************************************
    def valuations_by_agent(self,
                            c10, c11, c20, c21,
                            k, b):

        # Load parameters
        ψ1 = self.ψ1
        ψ2 = self.ψ2
        α = self.α
        A = self.A
        β = self.β
        bound = self.bound
        Vl = self.Vl
        Vh = self.Vh
```

(continues on next page)

(continued from previous page)

```
        kbot = self.kbot
        ktop = self.ktop
        bbot = self.bbot
        btop = self.btop
        w10 = self.w10
        w20 = self.w20
        θ10 = self.θ10
        θ20 = self.θ20
        w11 = self.w11
        w21 = self.w21
        g = self.g

        # Get functions for IMRS/state price density
        IMRS1 = lambda ϵ: β * (c11(ϵ)/c10)**(-ψ1)*g(ϵ)
        IMRS2 = lambda ϵ: β * (c21(ϵ)/c20)**(-ψ2)*g(ϵ)

        # Production
        fk = A*(k**α)
        Y = lambda ϵ: np.exp(ϵ)*fk

        # Compute integration threshold
        epstar = np.log(b/fk)

        # Compute equity valuation with agent 1's IMRS
        intQ1 = lambda ϵ: IMRS1(ϵ)*(Y(ϵ) - b)
        Q1 = quad(intQ1, epstar, bound)[0]

        # Compute bond valuation with agent 1's IMRS
        intP1 = lambda ϵ: IMRS1(ϵ)*Y(ϵ)/b
        P1 = quad(intP1, -bound, epstar)[0] + quad(IMRS1, epstar, bound)[0]

        # Compute equity valuation with agent 2's IMRS
        intQ2 = lambda ϵ: IMRS2(ϵ)*(Y(ϵ) - b)
        Q2 = quad(intQ2, epstar, bound)[0]

        # Compute bond valuation with agent 2's IMRS
        intP2 = lambda ϵ: IMRS2(ϵ)*Y(ϵ)/b
        P2 = quad(intP2, -bound, epstar)[0] + quad(IMRS2, epstar, bound)[0]

        return Q1,Q2,P1,P2


    #*************************************************************
    # Function: equilibrium valuations for firm, equity, bond
    #*************************************************************
    def eq_valuation(self, c10, c11, c20, c21, N=30):

        # Load parameters
        ψ1 = self.ψ1
        ψ2 = self.ψ2
        α = self.α
        A = self.A
        β = self.β
        bound = self.bound
        Vl = self.Vl
        Vh = self.Vh
```

(continues on next page)

(continued from previous page)

```
        kbot = self.kbot
        ktop = self.ktop
        bbot = self.bbot
        btop = self.btop
        w10 = self.w10
        w20 = self.w20
        θ10 = self.θ10
        θ20 = self.θ20
        w11 = self.w11
        w21 = self.w21
        g = self.g

        # Create grids
        kgrid, bgrid = np.meshgrid(np.linspace(kbot,ktop,N),
                                   np.linspace(bbot,btop,N))
        Vgrid = np.zeros_like(kgrid)
        Qgrid = np.zeros_like(kgrid)
        Pgrid = np.zeros_like(kgrid)

        # Loop: firm value
        for i in range(N):
            for j in range(N):

                # Get capital and debt
                k = kgrid[i,j]
                b = bgrid[i,j]

                # Valuations by each agent
                Q1,Q2,P1,P2 = self.valuations_by_agent(c10,
                                                       c11,
                                                       c20,
                                                       c21,
                                                       k,
                                                       b)

                # The prices will be the maximum of the valuations
                Q = max(Q1,Q2)
                P = max(P1,P2)

                # Compute firm value
                V = -k + Q + P*b
                Vgrid[i,j] = V
                Qgrid[i,j] = Q
                Pgrid[i,j] = P

        return kgrid, bgrid, Vgrid, Qgrid, Pgrid
```

39.6 Examples

Below we show some examples computed with the class `BCG_incomplete markets`.

39.6.1 First example

In the first example, we set up an instance of the BCG incomplete markets model with default parameter values.

```
mdl = BCG_incomplete_markets()
kss,bss,Vss,qss,pss,c10ss,c11ss,c20ss,c21ss,θ1ss = mdl.solve_eq(print_crit=False)
```

```
print(-kss+qss+pss*bss)
print(Vss)
print(θ1ss)
```

```
0.10073912888808995
0.100830078125
0.98564453125
```

Python reports to us that the equilibrium firm value is $V = 0.101$, with capital $k = 0.151$ and debt $b = 0.484$.

Let's verify some things that have to be true if our algorithm has truly found an equilibrium.

Thus, let's see if the firm is actually maximizing its firm value given the equilibrium pricing function $q(k, b)$ for equity and $p(k, b)$ for bonds.

```
kgrid, bgrid, Vgrid, Qgrid, Pgrid = mdl.eq_valuation(c10ss, c11ss, c20ss, c21ss,N=30)

print('Maximum valuation of the firm value in the (k,B) grid: {:.5f}'.format(Vgrid.
↪max()))
print('Equilibrium firm value: {:.5f}'.format(Vss))
```

```
Maximum valuation of the firm value in the (k,B) grid: 0.10074
Equilibrium firm value: 0.10083
```

Up to the approximation involved in using a discrete grid, these numbers give us comfort that the firm does indeed seem to be maximizing its value at the top of the value hill on the (k, b) plane that it faces.

Below we will plot the firm's value as a function of k, b.

We'll also plot the equilibrium price functions $q(k, b)$ and $p(k, b)$.

```
from IPython.display import Image
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d
import plotly.graph_objs as go

# Firm Valuation
fig = go.Figure(data=[go.Scatter3d(x=[kss],
                                   y=[bss],
                                   z=[Vss],
                                   mode='markers',
                                   marker=dict(size=3, color='red')),
```

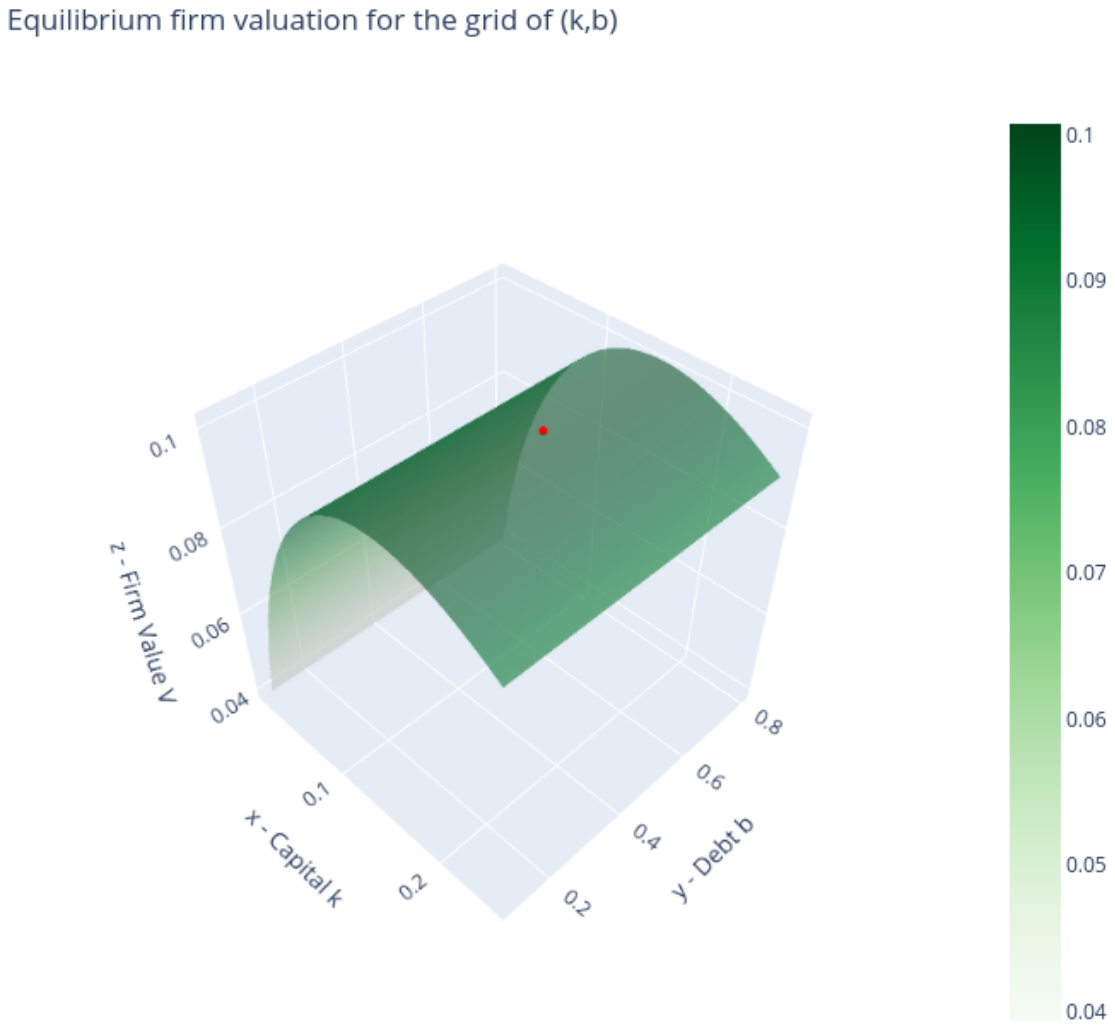
(continues on next page)

(continued from previous page)

```
                        go.Surface(x=kgrid,
                                   y=bgrid,
                                   z=Vgrid,
                                   colorscale='Greens',opacity=0.6)])

fig.update_layout(scene = dict(
                    xaxis_title='x - Capital k',
                    yaxis_title='y - Debt b',
                    zaxis_title='z - Firm Value V',
                    aspectratio = dict(x=1,y=1,z=1)),
                  width=700,
                  height=700,
                  margin=dict(l=50, r=50, b=65, t=90))
fig.update_layout(scene_camera=dict(eye=dict(x=1.5, y=-1.5, z=2)))
fig.update_layout(title='Equilibrium firm valuation for the grid of (k,b)')

# Export to PNG file
Image(fig.to_image(format="png"))
# fig.show() will provide interactive plot when running
# code locally
```



A Modigliani-Miller theorem?

The red dot in the above graph is **both** an equilibrium (b, k) chosen by a representative firm **and** the equilibrium B, K pair chosen by the aggregate of all firms.

Thus, **in equilibrium** it is true that

$$(b, k) = (B, K)$$

But an individual firm named $\zeta \in [0, 1]$ neither knows nor cares whether it sets $(b(\zeta), k(\zeta)) = (B, K)$.

Indeed the above graph has a ridge of $b(\zeta)$'s that also maximize the firm's value so long as it sets $k(\zeta) = K$.

Here it is important that the measure of firms that deviate from setting b at the red dot is very small – measure zero – so that B remains at the red dot even while one firm ζ deviates.

So within this equilibrium, there is a *qualified* Modigliani-Miller theorem that asserts that firm ζ's value is independent of how it mixes its financing between equity and bonds (so long as it is not what other firms do on average).

Thus, while an individual firm ζ's financial structure is indeterminate, the **market's** financial structure is determinant and sits at the red dot in the above graph.

This contrasts sharply with the *unqualified* Modigliani-Miller theorem descibed in the complete markets model in the lecture *Irrelevance of Capital Structure with Complete Markets*.

There the **market's** financial structure was indeterminate.

These subtle distinctions bear more thought and exploration.

So we will do some calculations to ferret out a sense in which the equilibrium $(k, b) = (K, B)$ outcome at the red dot in the above graph is **stable**.

In particular, we'll explore the consequences of some choices of $b = B$ that deviate from the red dot and ask whether firm ζ would want to remain at that b.

In more detail, here is what we'll do:

1. Obtain equilibrium values of capital and debt as $k^* = K$ and $b^* = B$, the red dot above.
2. Now fix k^* and let $b^{**} = b^* - e$ for some $e > 0$. Conjecture that big $K = k^*$ but big $B = b^{**}$.
3. Take K and B and compute intertermporal marginal rates of substitution (IMRS's) as we did before.
4. Taking the **new** IMRS to the firm's problem. Plot 3D surface for the valuations of the firm with this **new** IMRS.
5. Check if the value at k^*, b^{**} is at the top of this new 3D surface.
6. Repeat these calculations for $b^{**} = b^* + e$.

To conduct the above procedures, we create a function `off_eq_check` that inputs the BCG model instance parameters, equilibrium capital $K = k^*$ and debt $B = b^*$, and a perturbation of debt e.

The function outputs the fixed point firm values V^{**}, prices q^{**}, p^{**}, and consumption choices c^{**}.

Importantly, we relax the condition that only agent 2 holds bonds.

Now **both** agents can hold bonds, i.e., $0 \leq \xi^1 \leq B$ and $\xi^1 + \xi^2 = B$.

That implies the consumers' budget constraints are:

$$
\begin{aligned}
c_0^1 &= w_0^1 + \theta_0^1 V - q\theta^1 - p\xi^1 \\
c_0^2 &= w_0^2 + \theta_0^2 V - q\theta^2 - p\xi^2 \\
c_1^1(\epsilon) &= w_1^1(\epsilon) + \theta^1 d^e(k, b; \epsilon) + \xi^1 \\
c_1^2(\epsilon) &= w_1^2(\epsilon) + \theta^2 d^e(k, b; \epsilon) + \xi^2
\end{aligned}
$$

The function also outputs agent 1's bond holdings ξ_1.

```
def off_eq_check(mdl,kss,bss,e=0.1):
    # Big K and big B
    k = kss
    b = bss + e

    # Load parameters
    ψ1 = mdl.ψ1
    ψ2 = mdl.ψ2
    α = mdl.α
    A = mdl.A
    β = mdl.β
    bound = mdl.bound
```

(continues on next page)

(continued from previous page)

```
    Vl = mdl.Vl
    Vh = mdl.Vh
    kbot = mdl.kbot
    ktop = mdl.ktop
    bbot = mdl.bbot
    btop = mdl.btop
    w10 = mdl.w10
    w20 = mdl.w20
    θ10 = mdl.θ10
    θ20 = mdl.θ20
    w11 = mdl.w11
    w21 = mdl.w21
    g = mdl.g

    Y = njit(lambda ϵ, fk: np.exp(ϵ)*fk)
    intqq1 = njit(lambda ϵ, fk, θ1, ψ1, ξ1, b: (w11(ϵ) + θ1*(Y(ϵ, fk) - b) + ξ1)**(-
↪ψ1)*(Y(ϵ, fk) - b)*g(ϵ))
    intpp1a = njit(lambda ϵ, fk, ψ1, ξ1, b: (Y(ϵ, fk)/b)*(w11(ϵ) + Y(ϵ, fk)/b*ξ1)**(-
↪ψ1)*g(ϵ))
    intpp1b = njit(lambda ϵ, fk, θ1, ψ1, ξ1, b: (w11(ϵ) + θ1*(Y(ϵ, fk)-b) + ξ1)**(-
↪ψ1)*g(ϵ))
    intpp2a = njit(lambda ϵ, fk, ψ2, ξ2, b: (Y(ϵ, fk)/b)*(w21(ϵ) + Y(ϵ, fk)/b*ξ2)**(-
↪ψ2)*g(ϵ))
    intpp2b = njit(lambda ϵ, fk, θ2, ψ2, ξ2, b: (w21(ϵ) + θ2*(Y(ϵ, fk)-b) + ξ2)**(-
↪ψ2)*g(ϵ))
    intqq2 = njit(lambda ϵ, fk, θ2, ψ2, b: (w21(ϵ) + θ2*(Y(ϵ, fk)-b) + b)**(-ψ2)*(Y(ϵ,
↪ fk) - b)*g(ϵ))


    # Loop: Find fixed points V, q and p
    V_crit = 1
    while V_crit>1e-5:

        # We begin by adding the guess for the value of the firm to endowment
        V = (Vl+Vh)/2
        ww10 = w10 + θ10*V
        ww20 = w20 + θ20*V

        # Production
        fk = A*(k**α)
#         Y = lambda ?: np.exp(?)*fk

        # Compute integration threshold
        epstar = np.log(b/fk)


        #************************************************************
        # Compute the prices and allocations consistent with consumers'
        # Euler equations
        #************************************************************

        # We impose the following:
        # Agent 1 buys equity
        # Agent 2 buys equity and all debt
        # Agents trade such that prices converge
```

(continues on next page)

(continued from previous page)

```
        #========
        # Agent 1
        #========
        # Holdings
        ξ1a = 0
        ξ1b = b/2
        p = 0.3

        while abs(ξ1b - ξ1a) > 0.001:

            ξ1 = (ξ1a + ξ1b) / 2
            θ1a = 0.3
            θ1b = 1

            while abs(θ1b - θ1a) > (0.001/b):

                θ1 = (θ1a + θ1b) / 2

                # qq1 is the equity price consistent with agent-1 Euler Equation
                ## Note: Price is in the date-0 budget constraint of the agent

                ## First, compute the constant term that is not influenced by q
                ## that is, E[u'(c^{1}_{1})d^{e}(k,B)]
#                 intqq1 = lambda ω: (w11(ω) + ω1*(Y(ω, fk) - b) + ω1)**(-ψ1)*(Y(ω,␣
 ↪fk) - b)*g(ω)
#                 const_qq1 = β * quad(intqq1,epstar,bound)[0]
                const_qq1 = β * quad(intqq1,epstar,bound, args=(fk, θ1, ψ1, ξ1, b))[0]

                ## Second, iterate to get the equity price q
                qq1l = 0
                qq1h = ww10
                diff = 1
                while diff > 1e-7:
                    qq1 = (qq1l+qq1h)/2
                    rhs = const_qq1/((ww10-qq1*θ1-p*ξ1)**(-ψ1));
                    if (rhs > qq1):
                        qq1l = qq1
                    else:
                        qq1h = qq1
                    diff = abs(qq1l-qq1h)

                # pp1 is the bond price consistent with agent-2 Euler Equation
                ## Note: Price is in the date-0 budget constraint of the agent

                ## First, compute the constant term that is not influenced by p
                ## that is, E[u'(c^{1}_{1})d^{b}(k,B)]
#                 intpp1a = lambda ω: (Y(ω, fk)/b)*(w11(ω) + Y(ω, fk)/b*ω1)**(-
 ↪ω1)*g(ω)
#                 intpp1b = lambda ω: (w11(ω) + ω1*(Y(ω, fk)-b) + ω1)**(-ω1)*g(ω)
#                 const_pp1 = β * (quad(intpp1a,-bound,epstar)[0] + quad(intpp1b,
 ↪epstar,bound)[0])
                const_pp1 = β * (quad(intpp1a,-bound,epstar, args=(fk, ψ1, ξ1, b))[0]␣
 ↪\
                                 + quad(intpp1b,epstar,bound, args=(fk, θ1, ψ1, ξ1,␣
 ↪b))[0])
```

(continues on next page)

(continued from previous page)

```
                ## iterate to get the bond price p
                pp1l = 0
                pp1h = ww10/b
                diff = 1
                while diff > 1e-7:
                    pp1 = (pp1l+pp1h)/2
                    rhs = const_pp1/((ww10-qq1*θ1-pp1*ξ1)**(-ψ1))
                    if (rhs > pp1):
                        pp1l = pp1
                    else:
                        pp1h = pp1
                    diff = abs(pp1l-pp1h)

                #========
                # Agent 2
                #========
                ξ2 = b - ξ1
                θ2 = 1 - θ1

                # pp2 is the bond price consistent with agent-2 Euler Equation
                ## Note: Price is in the date-0 budget constraint of the agent

                ## First, compute the constant term that is not influenced by p
                ## that is, E[u'(c^{2}_{1})d^{b}(k,B)]
#                 intpp2a = lambda z: (Y(z, fk)/b)*(w21(z) + Y(z, fk)/b*z2)**(-
 ↪z2)*g(z)
#                 intpp2b = lambda z: (w21(z) + z2*(Y(z, fk)-b) + z2)**(-z2)*g(z)
#                 const_pp2 = β * (quad(intpp2a,-bound,epstar)[0] + quad(intpp2b,
 ↪epstar,bound)[0])
                const_pp2 = β * (quad(intpp2a,-bound,epstar, args=(fk, ψ2, ξ2, b))[0]␣
 ↪\
                                 + quad(intpp2b,epstar,bound, args=(fk, θ2, ψ2, ξ2,␣
 ↪b))[0])

                ## iterate to get the bond price p
                pp2l = 0
                pp2h = ww20/b
                diff = 1
                while diff > 1e-7:
                    pp2 = (pp2l+pp2h)/2
                    rhs = const_pp2/((ww20-qq1*θ2-pp2*ξ2)**(-ψ2))
                    if (rhs > pp2):
                        pp2l = pp2
                    else:
                        pp2h = pp2
                    diff = abs(pp2l-pp2h)

                # p be the maximum valuation for the bond among agents
                ## This will be the equity price based on Makowski's criterion
                p = max(pp1,pp2)


                # qq2 is the equity price consistent with agent-2 Euler Equation
#                 intqq2 = lambda z: (w21(z) + z2*(Y(z, fk)-b) + b)**(-z2)*(Y(z, fk) -
 ↪ b)*g(z)
#                 const_qq2 = β * quad(intqq2,epstar,bound)[0]
```

(continues on next page)

(continued from previous page)

```
                const_qq2 = β * quad(intqq2,epstar,bound, args=(fk, θ2, ψ2, b))[0]
                qq2l = 0
                qq2h = ww20
                diff = 1
                while diff > 1e-7:
                    qq2 = (qq2l+qq2h)/2
                    rhs = const_qq2/((ww20-qq2*θ2-p*ξ2)**(-ψ2));
                    if (rhs > qq2):
                        qq2l = qq2
                    else:
                        qq2h = qq2
                    diff = abs(qq2l-qq2h)

                # q be the maximum valuation for the equity among agents
                ## This will be the equity price based on Makowski's criterion
                q = max(qq1,qq2)

                #================
                # Update holdings
                #================
                if qq1 > qq2:
                    θ1a = θ1
                else:
                    θ1b = θ1

                #print(p,q,θ1,ξ1)

            if pp1 > pp2:
                ξ1a = ξ1
            else:
                ξ1b = ξ1


        #================
        # Get consumption
        #================
        c10 = ww10 - q*θ1 - p*ξ1
        c11 = lambda ϵ: w11(ϵ) + θ1*max(Y(ϵ, fk)-b,0) + ξ1*min(Y(ϵ, fk)/b,1)
        c20 = ww20 - q*(1-θ1) - p*(b-ξ1)
        c21 = lambda ϵ: w21(ϵ) + (1-θ1)*max(Y(ϵ, fk)-b,0) + (b-ξ1)*min(Y(ϵ, fk)/b,1)

        # Compute the value of the firm
        value_x = -k + q + p*b
        if (value_x > V):
            Vl = V
        else:
            Vh = V
        V_crit = abs(value_x-V)

    return V,k,b,p,q,c10,c11,c20,c21,ξ1
```

Here is our strategy for checking *stability* of an equilibrium.

We use `off_eq_check` to obtain consumption plans for both agents at the conjectured big K and big B.

Then we input consumption plans into the function `eq_valuation` from the BCG model class and plot the agents' valuations associated with different choices of k and b.

Our hunch is that (k^*, b^{**}) is **not** at the top of the firm valuation 3D surface so that the firm is **not** maximizing its value if it chooses $k = K = k^*$ and $b = B = b^{**}$.

That indicates that (k^*, b^{**}) is not an equilibrium capital structure for the firm.

We first check the case in which $b^{**} = b^* - e$ where $e = 0.1$:

```
#====================== Experiment 1 ======================#
Ve1,ke1,be1,pe1,qe1,c10e1,c11e1,c20e1,c21e1,ξ1e1 = off_eq_check(mdl,
                                                               kss,
                                                               bss,
                                                               e=-0.1)

# Firm Valuation
kgride1, bgride1, Vgride1, Qgride1, Pgride1 = mdl.eq_valuation(c10e1, c11e1, c20e1,␣
 ↪c21e1,N=20)

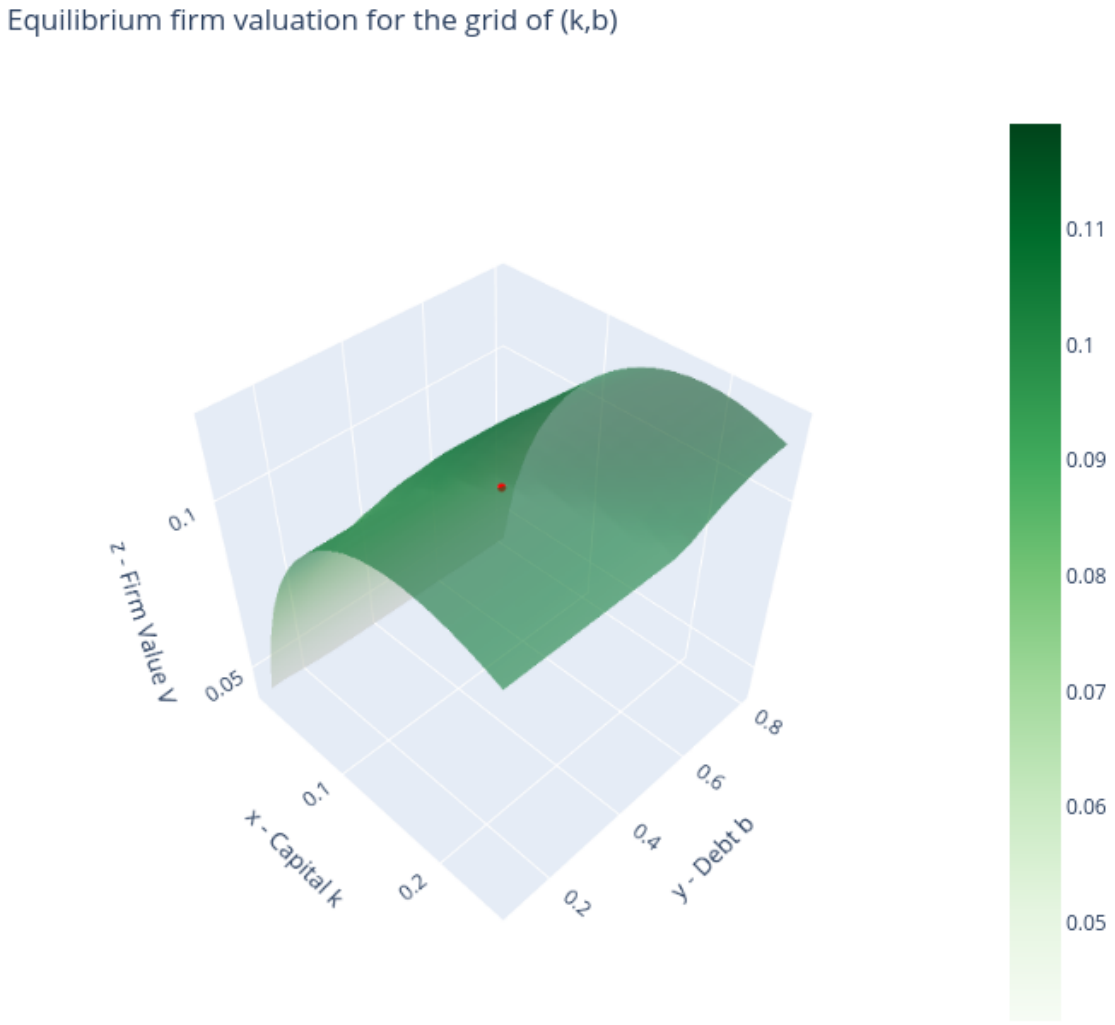
print('Maximum valuation of the firm value in the (k,b) grid: {:.4f}'.format(Vgride1.
 ↪max()))
print('Equilibrium firm value: {:.4f}'.format(Ve1))

fig = go.Figure(data=[go.Scatter3d(x=[ke1],
                                   y=[be1],
                                   z=[Ve1],
                                   mode='markers',
                                   marker=dict(size=3, color='red')),
                      go.Surface(x=kgride1,
                                 y=bgride1,
                                 z=Vgride1,
                                 colorscale='Greens',opacity=0.6)])

fig.update_layout(scene = dict(
                    xaxis_title='x - Capital k',
                    yaxis_title='y - Debt b',
                    zaxis_title='z - Firm Value V',
                    aspectratio = dict(x=1,y=1,z=1)),
                  width=700,
                  height=700,
                  margin=dict(l=50, r=50, b=65, t=90))
fig.update_layout(scene_camera=dict(eye=dict(x=1.5, y=-1.5, z=2)))
fig.update_layout(title='Equilibrium firm valuation for the grid of (k,b)')


# Export to PNG file
Image(fig.to_image(format="png"))
# fig.show() will provide interactive plot when running
# code locally
```

```
Maximum valuation of the firm value in the (k,b) grid: 0.1191
Equilibrium firm value: 0.1118
```



In the above 3D surface of prospective firm valuations, the perturbed choice $(k^*, b^* - e)$, represented by the red dot, is not at the top.

The firm could issue more debts and attain a higher firm valuation from the market.

Therefore, $(k^*, b^* - e)$ would not be an equilibrium.

Next, we check for $b^{**} = b^* + e$.

```
#====================== Experiment 2 ======================#
Ve2,ke2,be2,pe2,qe2,c10e2,c11e2,c20e2,c21e2,ξ1e2 = off_eq_check(mdl,
                                                               kss,
                                                               bss,
                                                               e=0.1)

# Firm Valuation
kgride2, bgride2, Vgride2, Qgride2, Pgride2 = mdl.eq_valuation(c10e2, c11e2, c20e2,␣
```

(continues on next page)

(continued from previous page)

```
 ↪c21e2,N=20)

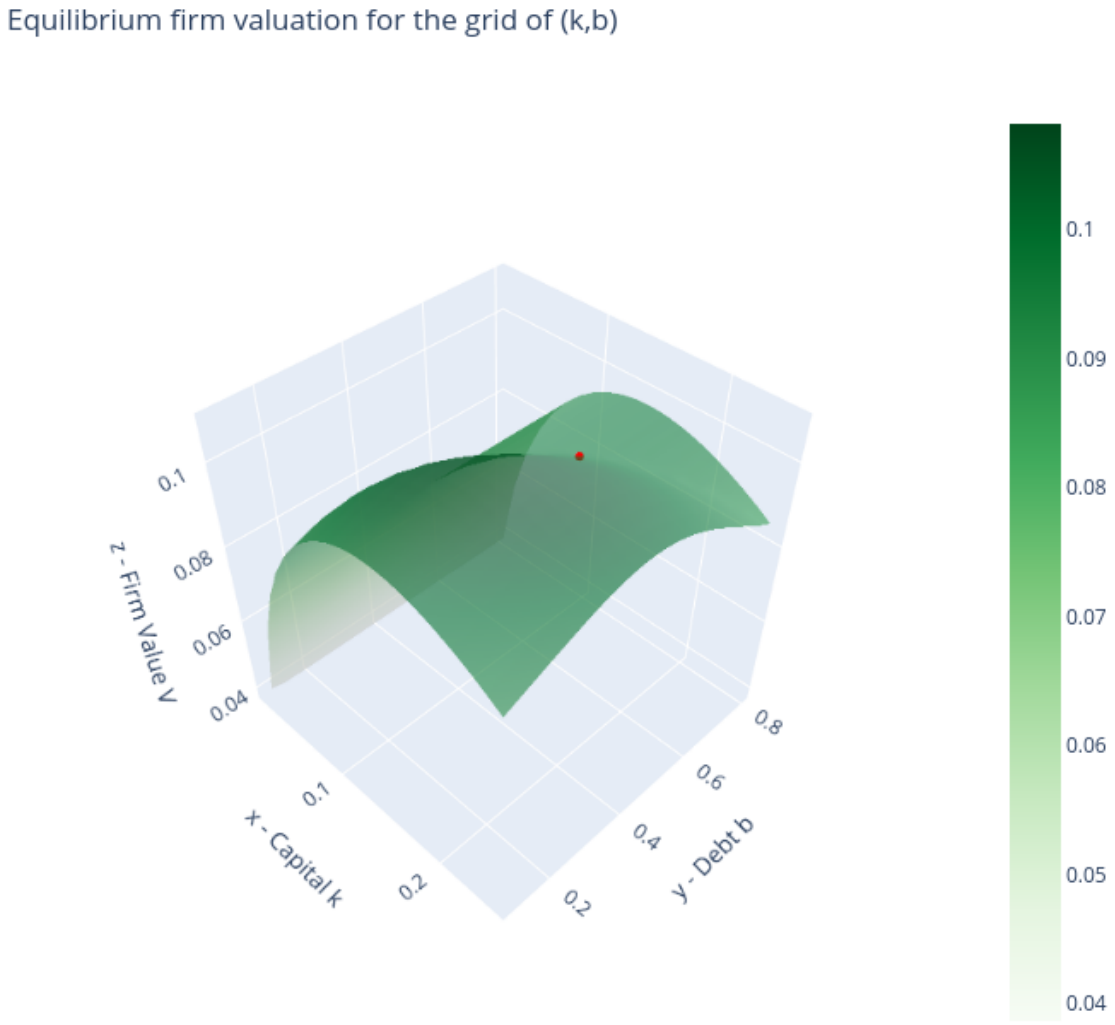
print('Maximum valuation of the firm value in the (k,b) grid: {:.4f}'.format(Vgride2.
 ↪max()))
print('Equilibrium firm value: {:.4f}'.format(Ve2))

fig = go.Figure(data=[go.Scatter3d(x=[ke2],
                                   y=[be2],
                                   z=[Ve2],
                                   mode='markers',
                                   marker=dict(size=3, color='red')),
                      go.Surface(x=kgride2,
                                 y=bgride2,
                                 z=Vgride2,
                                 colorscale='Greens',opacity=0.6)])

fig.update_layout(scene = dict(
                    xaxis_title='x - Capital k',
                    yaxis_title='y - Debt b',
                    zaxis_title='z - Firm Value V',
                    aspectratio = dict(x=1,y=1,z=1)),
                  width=700,
                  height=700,
                  margin=dict(l=50, r=50, b=65, t=90))
fig.update_layout(scene_camera=dict(eye=dict(x=1.5, y=-1.5, z=2)))
fig.update_layout(title='Equilibrium firm valuation for the grid of (k,b)')

# Export to PNG file
Image(fig.to_image(format="png"))
# fig.show() will provide interactive plot when running
# code locally
```

```
Maximum valuation of the firm value in the (k,b) grid: 0.1082
Equilibrium firm value: 0.0974
```



In contrast to $(k^*, b^* - e)$, the 3D surface for $(k^*, b^* + e)$ now indicates that a firm would want o *decrease* its debt issuance to attain a higher valuation.

That incentive to deviate means that $(k^*, b^* + e)$ is not an equilibrium capital structure for the firm.

Interestingly, if consumers were to anticipate that firms would over-issue debt, i.e. $B > b^*$, then both types of consumer would want to hold corporate debt.

For example, $\xi^1 > 0$:

```
print('Bond holdings of agent 1: {:.3f}'.format(ξ1e2))
```

```
Bond holdings of agent 1: 0.039
```

Our two *stability experiments* suggest that the equilibrium capital structure (k^*, b^*) is locally unique even though **at the equilibrium** an individual firm would be willing to deviate from the representative firms' equilibrium debt choice.

These experiments thus refine our discussion of the *qualified* Modigliani-Miller theorem that prevails in this example economy.

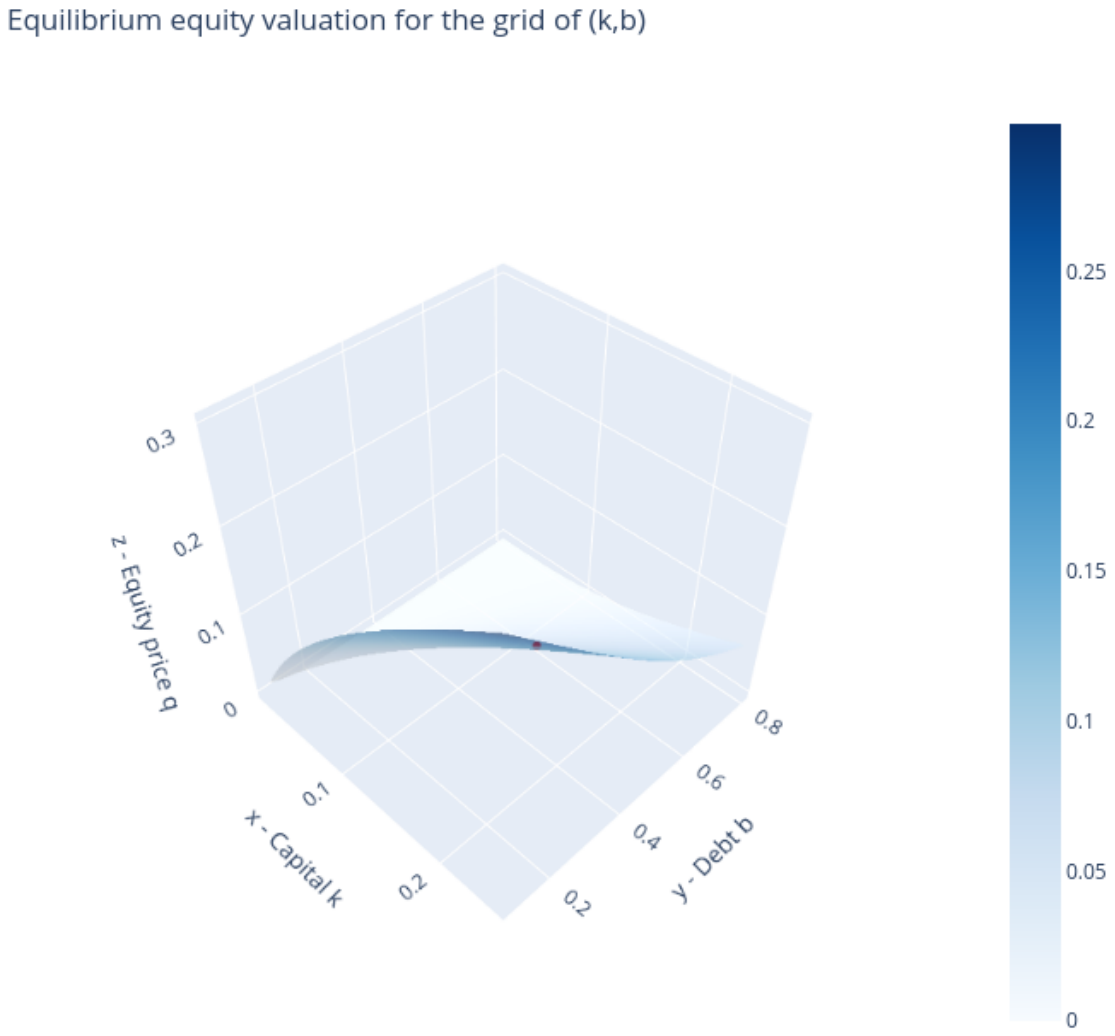
Equilibrium equity and bond price functions

It is also interesting to look at the equilibrium price functions $q(k, b)$ and $p(k, b)$ faced by firms in our rational expectations equilibrium.

```
# Equity Valuation
fig = go.Figure(data=[go.Scatter3d(x=[kss],
                                   y=[bss],
                                   z=[qss],
                                   mode='markers',
                                   marker=dict(size=3, color='red')),
                      go.Surface(x=kgrid,
                                 y=bgrid,
                                 z=Qgrid,
                                 colorscale='Blues',opacity=0.6)])

fig.update_layout(scene = dict(
                    xaxis_title='x - Capital k',
                    yaxis_title='y - Debt b',
                    zaxis_title='z - Equity price q',
                    aspectratio = dict(x=1,y=1,z=1)),
                  width=700,
                  height=700,
                  margin=dict(l=50, r=50, b=65, t=90))
fig.update_layout(scene_camera=dict(eye=dict(x=1.5, y=-1.5, z=2)))
fig.update_layout(title='Equilibrium equity valuation for the grid of (k,b)')


# Export to PNG file
Image(fig.to_image(format="png"))
# fig.show() will provide interactive plot when running
# code locally
```




```
# Bond Valuation
fig = go.Figure(data=[go.Scatter3d(x=[kss],
                                   y=[bss],
                                   z=[pss],
                                   mode='markers',
                                   marker=dict(size=3, color='red')),
                      go.Surface(x=kgrid,
                                 y=bgrid,
                                 z=Pgrid,
                                 colorscale='Oranges',opacity=0.6)])

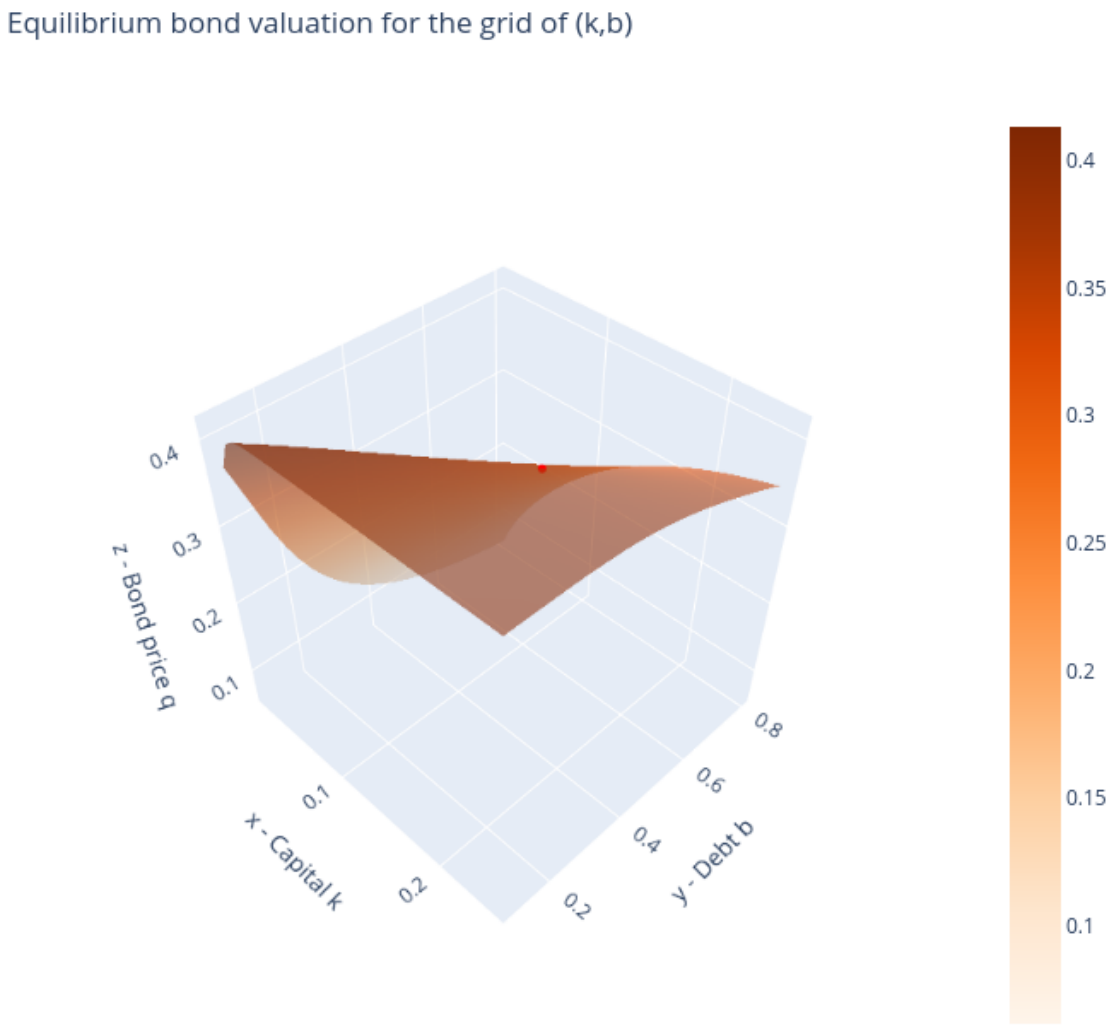
fig.update_layout(scene = dict(
                    xaxis_title='x - Capital k',
                    yaxis_title='y - Debt b',
                    zaxis_title='z - Bond price q',
                    aspectratio = dict(x=1,y=1,z=1)),
```

(continues on next page)

(continued from previous page)

```
                  width=700,
                  height=700,
                  margin=dict(l=50, r=50, b=65, t=90))
fig.update_layout(scene_camera=dict(eye=dict(x=1.5, y=-1.5, z=2)))
fig.update_layout(title='Equilibrium bond valuation for the grid of (k,b)')


# Export to PNG file
Image(fig.to_image(format="png"))
# fig.show() will provide interactive plot when running
# code locally
```



39.6.2 Comments on equilibrium pricing functions

The equilibrium pricing functions displayed above merit study and reflection.

They reveal the countervailing effects on a firm's valuations of bonds and equities that lie beneath the Modigliani-Miller ridge apparent in our earlier graph of an individual firm ζ's value as a function of $k(\zeta), b(\zeta)$.

39.6.3 Another example economy

We illustrate how the fraction of initial endowments held by agent 2, $w_0^2/(w_0^1+w_0^2)$ affects an equilibrium capital structure $(k, b) = (K, B)$ well as associated equilibrium allocations.

We are interested in how agents 1 and 2 value equity and bond.

$$Q^i = \beta \int \frac{u'(C_1^{i,*}(\epsilon))}{u'(C_0^{i,*})} d^e(k^*, b^*; \epsilon) g(\epsilon)\, d\epsilon$$

$$P^i = \beta \int \frac{u'(C_1^{i,*}(\epsilon))}{u'(C_0^{i,*})} d^b(k^*, b^*; \epsilon) g(\epsilon)\, d\epsilon$$

The function `valuations_by_agent` is used in calculating these valuations.

```
# Lists for storage
wlist = []
klist = []
blist = []
qlist = []
plist = []
Vlist = []
tlist = []
q1list = []
q2list = []
p1list = []
p2list = []

# For loop: optimization for each endowment combination
for i in range(10):
    print(i)

    # Save fraction
    w10 = 0.9 - 0.05*i
    w20 = 1.1 + 0.05*i
    wlist.append(w20/(w10+w20))

    # Create the instance
    mdl = BCG_incomplete_markets(w10 = w10, w20 = w20, ktop = 0.5, btop = 2.5)

    # Solve for equilibrium
    kss,bss,Vss,qss,pss,c10ss,c11ss,c20ss,c21ss,θ1ss = mdl.solve_eq(print_crit=False)

    # Store the equilibrium results
    klist.append(kss)
    blist.append(bss)
    qlist.append(qss)
    plist.append(pss)
    Vlist.append(Vss)
```

(continues on next page)

(continued from previous page)

```
    tlist.append(θ1ss)

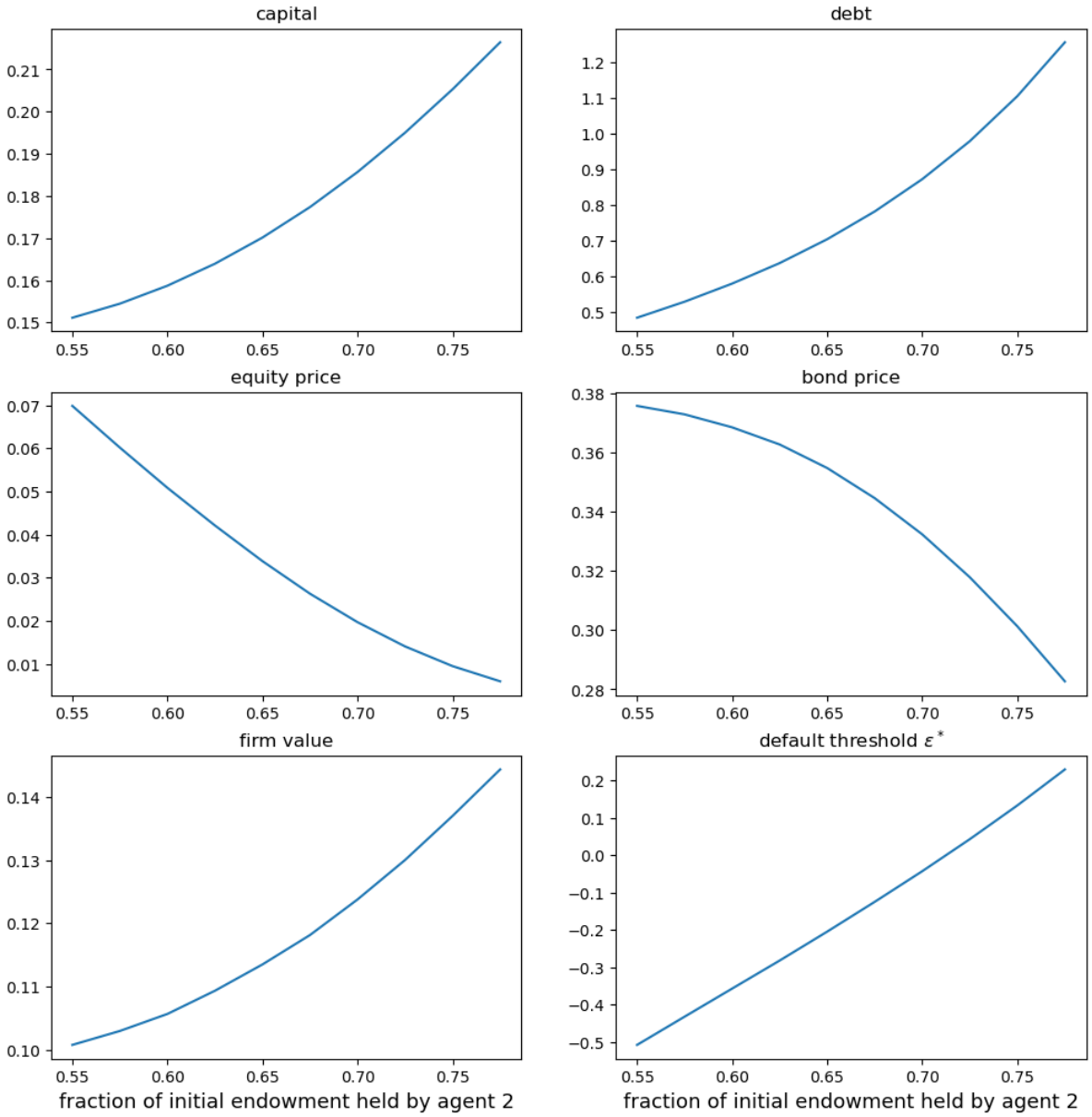
    # Evaluations of equity and bond by each agent
    Q1,Q2,P1,P2 = mdl.valuations_by_agent(c10ss, c11ss, c20ss, c21ss, kss, bss)

    # Save the valuations
    q1list.append(Q1)
    q2list.append(Q2)
    p1list.append(P1)
    p2list.append(P2)
```

```
# Plot
fig, ax = plt.subplots(3,2,figsize=(12,12))
ax[0,0].plot(wlist,klist)
ax[0,0].set_title('capital')
ax[0,1].plot(wlist,blist)
ax[0,1].set_title('debt')
ax[1,0].plot(wlist,qlist)
ax[1,0].set_title('equity price')
ax[1,1].plot(wlist,plist)
ax[1,1].set_title('bond price')
ax[2,0].plot(wlist,Vlist)
ax[2,0].set_title('firm value')
ax[2,0].set_xlabel('fraction of initial endowment held by agent 2',fontsize=13)

# Create a list of Default thresholds
A = mdl.A
α = mdl.α
epslist = []
for i in range(len(wlist)):
    bb = blist[i]
    kk = klist[i]
    eps = np.log(bb/(A*kk**α))
    epslist.append(eps)

# Plot (cont.)
ax[2,1].plot(wlist,epslist)
ax[2,1].set_title(r'default threshold $\epsilon^*$')
ax[2,1].set_xlabel('fraction of initial endowment held by agent 2',fontsize=13)
plt.show()
```



39.7 A picture worth a thousand words

Please stare at the above panels.

They describe how equilibrium prices and quantities respond to alterations in the structure of society's *hedging desires* across economies with different allocations of the initial endowment to our two types of agents.

Now let's see how the two types of agents value bonds and equities, keeping in mind that the type that values the asset highest determines the equilibrium price (and thus the pertinent set of Big C's).

```
# Comparing the prices
fig, ax = plt.subplots(1,3,figsize=(16,6))
```

(continues on next page)

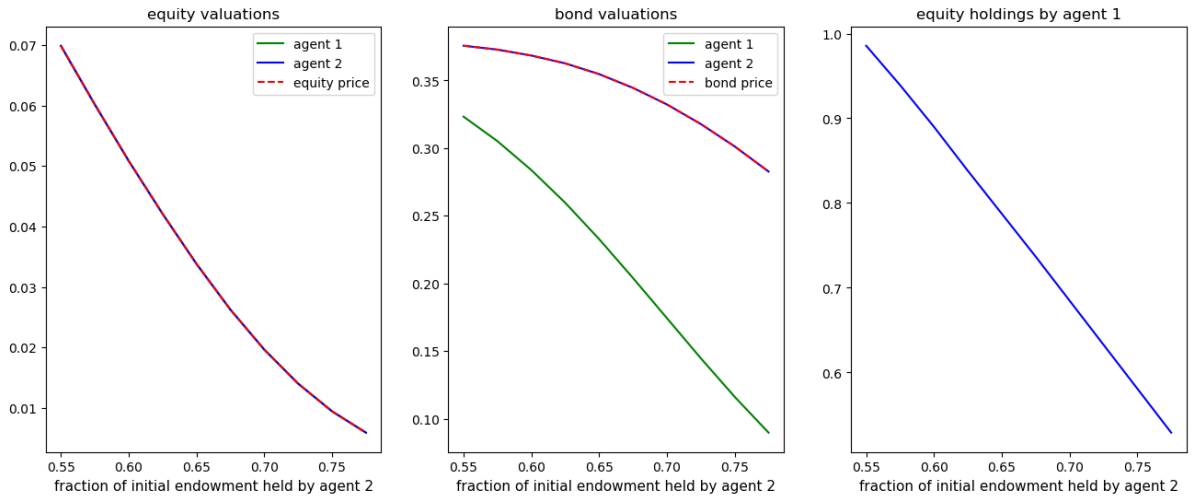
(continued from previous page)

```
ax[0].plot(wlist,q1list,label='agent 1',color='green')
ax[0].plot(wlist,q2list,label='agent 2',color='blue')
ax[0].plot(wlist,qlist,label='equity price',color='red',linestyle='--')
ax[0].legend()
ax[0].set_title('equity valuations')
ax[0].set_xlabel('fraction of initial endowment held by agent 2',fontsize=11)

ax[1].plot(wlist,p1list,label='agent 1',color='green')
ax[1].plot(wlist,p2list,label='agent 2',color='blue')
ax[1].plot(wlist,plist,label='bond price',color='red',linestyle='--')
ax[1].legend()
ax[1].set_title('bond valuations')
ax[1].set_xlabel('fraction of initial endowment held by agent 2',fontsize=11)

ax[2].plot(wlist,tlist,color='blue')
ax[2].set_title('equity holdings by agent 1')
ax[2].set_xlabel('fraction of initial endowment held by agent 2',fontsize=11)

plt.show()
```



It is rewarding to stare at the above plots too.

In equilibrium, equity valuations are the same across the two types of agents but bond valuations are not.

Agents of type 2 value bonds more highly (they want more hedging).

Taken together with our earlier plot of equity holdings, these graphs confirm our earlier conjecture that while both type of agents hold equities, only agents of type 2 holds bonds.

Part VIII

Dynamic Programming Squared

CHAPTER

FORTY

OPTIMAL UNEMPLOYMENT INSURANCE

40.1 Overview

This lecture describes a model of optimal unemployment insurance created by Shavell and Weiss (1979) [Shavell and Weiss, 1979].

We use recursive techniques of Hopenhayn and Nicolini (1997) [Hopenhayn and Nicolini, 1997] to compute optimal insurance plans for Shavell and Weiss's model.

Hopenhayn and Nicolini's model is a generalization of Shavell and Weiss's along dimensions that we'll soon describe.

40.2 Shavell and Weiss's Model

An unemployed worker orders stochastic processes of consumption and search effort $\{c_t, a_t\}_{t=0}^{\infty}$ according to

$$E \sum_{t=0}^{\infty} \beta^t \left[u(c_t) - a_t \right] \tag{40.1}$$

where $\beta \in (0, 1)$ and $u(c)$ is strictly increasing, twice differentiable, and strictly concave.

We assume that $u(0)$ is well defined.

We require that $c_t \geq 0$ and $a_t \geq 0$.

All jobs are alike and pay wage $w > 0$ units of the consumption good each period forever.

An unemployed worker searches with effort a and with probability $p(a)$ receives a permanent job at the beginning of the next period.

Furthermore, $a = 0$ when the worker is employed.

The probability of finding a job is $p(a)$.

p is an increasing, strictly concave, and twice differentiable function of a that satisfies $p(a) \in [0, 1]$ for $a \geq 0$, $p(0) = 0$.

Note: When we compute examples below, we'll use assume the same $p(a)$ function as [Hopenhayn and Nicolini, 1997], namely, $p(a) = 1 - \exp(-ra)$, where r is a parameter that we'll calibrate to hit the same target that [Hopenhayn and Nicolini, 1997] did, namely, an empirical hazard rate of leaving unemployment.

The consumption good is nonstorable.

An unemployed worker has no savings and cannot borrow or lend.

The unemployed worker's only source of consumption smoothing over time and across states is an **insurance agency** or **planner**.

Once a worker has found a job, he is beyond the planner's grasp.

- This is Shavell and Weiss's assumption, but not Hopenhayn and Nicolini's.
- Hopenhayn and Nicolini allow the unemployment insurance agency to impose history-dependent taxes on previously unemployed workers.
- Since there is no incentive problem after the worker has found a job, it is optimal for the agency to provide an employed worker with a constant level of consumption.
- Hence, Hopenhayn and Nicolini's insurance agency imposes a permanent per-period history-dependent tax on a previously unemployed but presently employed worker.

40.2.1 Autarky

As a benchmark, we first study the fate of an unemployed worker who has no access to unemployment insurance.

Because employment is an absorbing state for the worker, we work backward from that state.

Let V^e be the expected sum of discounted one-period utilities of an employed worker.

Once the worker is employed, $a = 0$, making his period utility be $u(c) - a = u(w)$ forever.

Therefore,

$$V^e = \frac{u(w)}{(1-\beta)}. \tag{40.2}$$

Now let V^u be the expected discounted present value of utility for an unemployed worker who chooses consumption, effort pair (c, a) optimally.

Value V^u satisfies the Bellman equation

$$V^u = \max_{a \geq 0} \left\{ u(0) - a + \beta \left[p(a) V^e + (1 - p(a)) V^u \right] \right\}. \tag{40.3}$$

The first-order condition for a maximum is

$$\beta p'(a) \left[V^e - V^u \right] \leq 1, \tag{40.4}$$

with equality if $a > 0$.

Since there is no state variable in this infinite horizon problem, there is a time-invariant optimal search intensity a and an associated value of being unemployed V^u.

Let $V_{\text{aut}} = V^u$ solve Bellman equation (40.3).

Equations (40.3) and (40.4) form the basis for an iterative algorithm for computing $V^u = V_{\text{aut}}$.

- Let V_j^u be the estimate of V_{aut} at the jth iteration.
- Use this value in equation (40.4) and solve for an estimate of effort a_j.
- Use this value in a version of equation (40.3) with V_j^u on the right side to compute V_{j+1}^u.
- Iterate to convergence.

40.2.2 Full Information

Another benchmark model helps set the stage for the model with private information that we ultimately want to study.

We temporarily assume that an unemployment insurance agency has full information about the unemployed worker.

We assume that the insurance agency can control both the consumption and the search effort of an unemployed worker.

The agency wants to design an unemployment insurance contract to give the unemployed worker expected discounted utility $V > V_{\text{aut}}$.

The agency, i.e., the planner, wants to deliver value V efficiently, meaning in a way that minimizes an expected present value discounted costs, using β as the discount factor.

We formulate the optimal insurance problem recursively.

Let $C(V)$ be the expected discounted cost of giving the worker expected discounted utility V.

The cost function is strictly convex because a higher V implies a lower marginal utility of the worker; that is, additional expected utils can be awarded to the worker only at an increasing marginal cost in terms of the consumption good.

Given V, the planner assigns first-period pair (c, a) and promised continuation value V^u next period if the worker is unlucky and does not find a job this period.

The planner sets (c, a, V^u) as functions of V and to satisfy the following Bellman equation for associated cost function $C(V)$:

$$C(V) = \min_{c,a,V^u} \left\{ c + \beta[1 - p(a)]C(V^u) \right\}, \tag{40.5}$$

where minimization is subject to the promise-keeping constraint

$$V \leq u(c) - a + \beta \left\{ p(a)V^e + [1 - p(a)]V^u \right\}. \tag{40.6}$$

Here V^e is given by equation (40.2), which reflects the assumption that once the worker is employed, he is beyond the reach of the unemployment insurance agency.

The right side of Bellman equation (40.5) is attained by policy functions $c = c(V), a = a(V)$, and $V^u = V^u(V)$.

The promise-keeping constraint, equation (40.6), asserts that the 3-tuple (c, a, V^u) attains at least V.

Let θ be a Lagrange multiplier on constraint (40.6).

At an interior solution, first-order conditions with respect to c, a, and V^u, respectively, are

$$\begin{aligned} \theta &= \frac{1}{u'(c)}, \\ C(V^u) &= \theta \left[\frac{1}{\beta p'(a)} - (V^e - V^u) \right], \\ C'(V^u) &= \theta. \end{aligned} \tag{40.7}$$

The envelope condition $C'(V) = \theta$ and the third equation of (40.7) imply that $C'(V^u) = C'(V)$.

Strict convexity of C then implies that $V^u = V$.

Applied repeatedly over time, $V^u = V$ makes the continuation value remain constant during the entire spell of unemployment.

The first equation of (40.7) determines c, and the second equation of (40.7) determines a, both as functions of promised value V.

That $V^u = V$ then implies that c and a are held constant during the unemployment spell.

Thus, the unemployed worker's consumption c and search effort a are both fully smoothed during the unemployment spell.

But the worker's consumption is not smoothed across states of employment and unemployment unless $V = V^e$.

40.2.3 Incentive Problem

The preceding efficient insurance scheme assumes that the insurance agency controls both c and a.

The insurance agency cannot simply provide c and then allow the worker to choose a.

Here is why.

The agency delivers a value V^u higher than the autarky value V_{aut} by doing two things.

It **increases** the unemployed worker's consumption c and **decreases** his search effort a.

The prescribed search effort is **higher** than what the worker would choose if he were to be guaranteed consumption level c while he remains unemployed.

This follows from the first two equations of (40.7) and the fact that the insurance scheme is costly, $C(V^u) > 0$, which imply $[\beta p'(a)]^{-1} > (V^e - V^u)$.

Now look at the worker's first-order condition (40.4) under autarky.

It implies that if search effort $a > 0$, then $[\beta p'(a)]^{-1} = [V^e - V^u]$, which is inconsistent with the inequality $[\beta p'(a)]^{-1} > (V^e - V^u)$ that prevails when $a > 0$ when the agency controls both a and c.

If he were free to choose a, the worker would therefore want to fulfill (40.4), either at equality so long as $a > 0$, or by setting $a = 0$ otherwise.

Starting from the a associated with the full-information social insurance scheme in which the agency controls both c and a, the worker would establish the desired equality in (40.4) by *lowering* a, thereby decreasing the term $[\beta p'(a)]^{-1}$ (which also lowers $(V^e - V^u)$ when the value of being unemployed V^u increases).

If an equality can be established before a reaches zero, this would be the worker's preferred search effort; otherwise the worker would find it optimal to accept the insurance payment, set $a = 0$, and never work again.

Thus, since the worker does not take the cost of the insurance scheme into account, he would choose a search effort below the socially optimal, full-information level.

The full-information contract thus relies on the agency's ability to control *both* the unemployed worker's consumption *and* his search effort.

40.3 Private Information

Following [Shavell and Weiss, 1979] and [Hopenhayn and Nicolini, 1997], now assume that the unemployment insurance agency cannot observe or control a, though it can observe and control c.

The worker is free to choose a, which puts expression (40.4), the worker's first-order condition under autarky, back in the picture.

- We are assuming that the worker's best response to the unemployment insurance arrangement is completely characterized by the first-order condition (40.4), an instance of the so-called first-order approach to incentive problems.

Given a contract, the individual will choose search effort according to first-order condition (40.4).

This fact motivates the insurance agency to design an unemployment insurance contract that respects this restriction.

Thus, the contract design problem is now to minimize the right side of equation (40.5) subject to expression (40.6) and the incentive constraint (40.4).

Since the restrictions (40.4) and (40.6) are not linear and generally do not define a convex set, it becomes challenging to provide conditions under which the solution to the dynamic programming problem results in a convex function $C(V)$.

- Sometimes this complication can be handled by convexifying the constraint set by introducing lotteries.
- A common finding is that optimal plans do not involve lotteries, because convexity of the constraint set is a sufficient but not necessary condition for convexity of the cost function.
- In order to characterize the optimal solution, we follow Hopenhayn and Nicolini (1997) [Hopenhayn and Nicolini, 1997] by hopefully proceeding under the assumption that $C(V)$ is strictly convex.

Let η be the multiplier on constraint (40.4), while θ continues to denote the multiplier on constraint (40.6).

But now we replace the weak inequality in (40.6) by an equality.

- We do this because the unemployment insurance agency cannot award a higher utility than V because that might violate an incentive-compatibility constraint for exerting the proper search effort in earlier periods.

At an interior solution, first-order conditions with respect to c, a, and V^u, respectively, are

$$\begin{aligned} \theta &= \frac{1}{u'(c)}\,, \\ C(V^u) &= \theta\left[\frac{1}{\beta p'(a)} - (V^e - V^u)\right] - \eta\frac{p''(a)}{p'(a)}(V^e - V^u) \\ &= -\eta\frac{p''(a)}{p'(a)}(V^e - V^u)\,, \\ C'(V^u) &= \theta - \eta\frac{p'(a)}{1-p(a)}\,, \end{aligned} \tag{40.8}$$

where the second equality in the second equation in (40.8) follows from strict equality of the incentive constraint (40.4) when $a > 0$.

As long as the insurance scheme is associated with costs, so that $C(V^u) > 0$, the first-order condition in the second equation of (40.8) implies that the multiplier η is strictly positive.

The first-order condition in the second equation of the third equality in (40.8) and the envelope condition $C'(V) = \theta$ together allow us to conclude that $C'(V^u) < C'(V)$.

Convexity of C then implies that $V^u < V$.

After we have also used the first equation of (40.8), it follows that in order to provide the proper incentives, the consumption of the unemployed worker must decrease as the duration of the unemployment spell lengthens.

It also follows from (40.4) at equality that search effort a rises as V^u falls, i.e., it rises with the duration of unemployment.

The of benefits on the duration of unemployment is designed to provide the worker an incentive to search.

To understand this, from the third equation of (40.8), notice how the conclusion that consumption falls with the duration of unemployment depends on the assumption that more search effort raises the prospect of finding a job, i.e., that $p'(a) > 0$.

If $p'(a) = 0$, then the third equation of (40.8) and the strict convexity of C imply that $V^u = V$.

Thus, when $p'(a) = 0$, there is no reason for the planner to make consumption fall with the duration of unemployment.

40.3.1 Computational Details

It is useful to note that there are natural lower and upper bounds to the set of continuation values V^u.

The lower bound is the expected lifetime utility in autarky, V_{aut}.

To compute an upper bound, represent condition (40.4) as

$$V^u \geq V^e - [\beta p'(a)]^{-1},$$

with equality if $a > 0$.

If there is zero search effort, then $V^u \geq V^e - [\beta p'(0)]^{-1}$.

Therefore, to rule out zero search effort we require

$$V^u < V^e - [\beta p'(0)]^{-1}.$$

(Remember that $p''(a) < 0$.)

This step gives our upper bound for V^u.

To formulate the Bellman equation numerically, we suggest using the constraints to eliminate c and a as choice variables, thereby reducing the Bellman equation to a minimization over the one choice variable V^u.

First express the promise-keeping constraint (40.6) at equality as

$$u(c) = V + a - \beta\{p(a)V^e + [1 - p(a)]V^u\}$$

so that consumption is

$$c = u^{-1}\left(V + a - \beta[p(a)V^e + (1 - p(a))V^u]\right). \tag{40.9}$$

Similarly, solving the inequality (40.4) for a leads to

$$a = \max\left\{0, p'^{-1}\left(\frac{1}{\beta(V^e - V^u)}\right)\right\}. \tag{40.10}$$

When we specialize (40.10) to the functional form for $p(a)$ used by Hopenhayn and Nicolini, we obtain

$$a = \max\left\{0, \frac{\log[r\beta(V^e - V^u)]}{r}\right\}. \tag{40.11}$$

Formulas (40.9) and (40.11) express (c, a) as functions of V and the continuation value V^u.

Using these functions allows us to write the Bellman equation in $C(V)$ as

$$C(V) = \min_{V^u}\{c + \beta[1 - p(a)]C(V^u)\} \tag{40.12}$$

where c and a are given by equations (40.9) and (40.11).

40.3.2 Python Computations

We'll approximate the planner's optimal cost function with cubic splines.

To do this, we'll load some useful modules

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

We first create a class to set up a particular parametrization.

```
class params_instance:

    def __init__(self,
                 r,
                 β = 0.999,
                 σ = 0.500,
                 w = 100,
                 n_grid = 50):

        self.β,self.σ,self.w,self.r = β,σ,w,r
        self.n_grid = n_grid
        uw = self.w**(1-self.σ)/(1-self.σ)  #Utility from consuming all wage
        self.Ve = uw/(1-β)
```

40.3.3 Parameter Values

For the other parameters appearing in the above Python code, we'll calibrate parameter r that pins down the function $p(a) = 1 - \exp(-ra)$ to match an observerd hazard rate – the probability that an unemployed worker finds a job each – in US data.

In particular, we seek an r so that in autarky `p(a(r)) = 0.1`, where `a` is the optimal search effort.

First, we create some helper functions.

```
# The probability of finding a job given search effort, a and parameter r.
def p(a,r):
    return 1-np.exp(-r*a)

def invp_prime(x,r):
    return -np.log(x/r)/r

def p_prime(a,r):
    return r*np.exp(-r*a)

# The utiliy function
def u(self,c):
    return (c**(1-self.σ))/(1-self.σ)

def u_inv(self,x):
    return ((1-self.σ)*x)**(1/(1-self.σ))
```

Recall that under autarky the value for an unemployed worker satisfies the Bellman equation

$$V^u = \max_a \{u(0) - a + \beta \left[p_r(a)V^e + (1 - p_r(a))V^u\right]\} \tag{40.13}$$

At the optimal choice of a, we have first-order necessary condition:

$$\beta p_r'(a)[V^e - V^u] \leq 1 \tag{40.14}$$

with equality when a >0.

Given a value of parameter $\bar{r}$, we can solve the autarky problem as follows:

1. Guess $V^u \in \mathbb{R}^+$
2. Given V^u, use the FOC (40.14) to calculate the implied optimal search effort a

3. Evaluate the difference between the LHS and RHS of the Bellman equation (40.13)
4. Update guess for V^u accordingly, then return to 2) and repeat until the Bellman equation is satisfied.

For a given r and guess V^u, the function `Vu_error` calculates the error in the Bellman equation under the optimal search intensity.

We'll soon use this as an input to computing V^u.

```
# The error in the Bellman equation that requires equality at
# the optimal choices.
def Vu_error(self,Vu,r):
    β= self.β
    Ve = self.Ve

    a = invp_prime(1/(β*(Ve-Vu)),r)
    error = u(self,0) -a + β*(p(a,r)*Ve + (1-p(a,r))*Vu) - Vu
    return error
```

Since the calibration exercise is to match the hazard rate under autarky to the data, we must find a parameter r to match `p(a,r) = 0.1`.

The function below `r_error` calculates, for a given guess of r the difference between the model implied equilibrium hazard rate and 0.1.

We'll use this to compute a calibrated r^*.

```
# The error of our p(a^*) relative to our calibration target
def r_error(self,r):
    β = self.β
    Ve = self.Ve

    Vu_star = sp.optimize.fsolve(Vu_error_Λ,15000,args = (r))
    a_star = invp_prime(1/(β*(Ve-Vu_star)),r) # Assuming a>0
    return    p(a_star,r) - 0.1
```

Now, let us create an instance of the model with our parametrization

```
params = params_instance(r = 1e-2)
# Create some lambda functions useful for fsolve function
Vu_error_Λ =  lambda Vu,r: Vu_error(params,Vu,r)
r_error_Λ =  lambda r: r_error(params,r)
```

We want to compute an r that is consistent with the hazard rate 0.1 in autarky.

To do so, we will use a bisection strategy.

```
r_calibrated = sp.optimize.brentq(r_error_Λ,1e-10,1-1e-10)
print(f"Parameter to match 0.1 hazard rate: r = {r_calibrated}")

Vu_aut = sp.optimize.fsolve(Vu_error_Λ,15000,args = (r_calibrated))[0]
a_aut = invp_prime(1/(params.β*(params.Ve-Vu_aut)),r_calibrated)

print(f"Check p at r: {p(a_aut,r_calibrated)}")
```

```
Parameter to match 0.1 hazard rate: r = 0.0003431409393866592
Check p at r: 0.10000000000001996
```

```
/tmp/ipykernel_9649/2412693371.py:6: RuntimeWarning: The iteration is not making
 ↪good progress, as measured by the
  improvement from the last five Jacobian evaluations.
  Vu_star = sp.optimize.fsolve(Vu_error_Λ,15000,args = (r))
```

Now that we have calibrated our the parameter r, we can continue with solving the model with private information.

40.3.4 Computation under Private Information

Our approach to solving the full model follows ideas of Judd (1998) [Judd, 1998], who uses a polynomial to approximate the value function and a numerical optimizer to perform the optimization at each iteration.

Note: For further details of the Judd (1998) [Judd, 1998] method, see [Ljungqvist and Sargent, 2018], Section 5.7.

We will use cubic splines to interpolate across a pre-set grid of points to approximate the value function.

Our strategy involves finding a function $C(V)$ – the expected cost of giving the worker value V – that satisfies the Bellman equation:

$$C(V) = \min_{c,a,V^u} \{c + \beta [1 - p(a)] C(V^u)\} \tag{40.15}$$

Notice that in equations (40.9) and (40.11), we have analytical solutions of c and a in terms of promised value V and V^u (and other parameters).

We can substitute these equations for c and a and obtain the functional equation (40.12).

```
def calc_c(self,Vu,V,a):
    '''
    Calculates the optimal consumption choice coming from the constraint of the
 ↪insurer's problem
    (which is also a Bellman equation)
    '''
    β,Ve,r = self.β,self.Ve,self.r

    c = u_inv(self,V + a - β*(p(a,r)*Ve + (1-p(a,r))*Vu))
    return c

def calc_a(self,Vu):
    '''
    Calculates the optimal effort choice coming from the worker's effort optimality
 ↪condition.
    '''

    r,β,Ve = self.r,self.β,self.Ve

    a_temp = np.log(r*β*(Ve - Vu))/r
    a = max(0,a_temp)
    return a
```

With these analytical solutions for optimal c and a in hand, we can reduce the minimization to (40.12) in the single variable V^u.

With this in hand, we have our algorithm.

40.3.5 Algorithm

1. Fix a set of grid points $grid_V$ for V and Vu_{grid} for V^u
2. Guess a function $C_0(V)$ that is evaluated at a grid $grid_V$.
3. For each point in $grid_V$ find the V^u that minimizes the expression on right side of (40.12). We find the minimum by evaluating the right side of (40.12) at each point in Vu_{grid} and then finding the minimum using cubic splines.
4. Evaluating the minimum across all points in $grid_V$ gives you another function $C_1(V)$.
5. If $C_0(V)$ and $C_1(V)$ are sufficiently different, then repeat steps 3-4 again. Otherwise, we are done.
6. Thus, the iterations are $C_{j+1}(V) = \min_{c,a,V^u}\{c - \beta[1-p(a)]C_j(V)\}$.

The function `iterate_C` below executes step 3 in the above algorithm.

```
# Operator iterate_C that calculates the next iteration of the cost function.
def iterate_C(self,C_old,Vu_grid):

    '''
    We solve the model by minimising the value function across a grid of possible␣
↪promised values.
    '''
    β,r,n_grid = self.β,self.r,self.n_grid

    C_new = np.zeros(n_grid)
    cons_star = np.zeros(n_grid)
    a_star = np.zeros(n_grid)
    V_star = np.zeros(n_grid)

    C_new2 = np.zeros(n_grid)
    V_star2 = np.zeros(n_grid)

    for V_i in range(n_grid):
        C_Vi_temp = np.zeros(n_grid)
        cons_Vi_temp = np.zeros(n_grid)
        a_Vi_temp = np.zeros(n_grid)

        for Vu_i in range(n_grid):
            a_i = calc_a(self,Vu_grid[Vu_i])
            c_i = calc_c(self,Vu_grid[Vu_i],Vu_grid[V_i],a_i)

            C_Vi_temp[Vu_i] = c_i + β*(1-p(a_i,r))*C_old[Vu_i]
            cons_Vi_temp[Vu_i] = c_i
            a_Vi_temp[Vu_i] = a_i

        # Interpolate across the grid to get better approximation of the minimum
        C_Vi_temp_interp = sp.interpolate.interp1d(Vu_grid,C_Vi_temp, kind = 'cubic')
        cons_Vi_temp_interp = sp.interpolate.interp1d(Vu_grid,cons_Vi_temp, kind =
↪'cubic')
        a_Vi_temp_interp = sp.interpolate.interp1d(Vu_grid,a_Vi_temp, kind = 'cubic')

        res = sp.optimize.minimize_scalar(C_Vi_temp_interp,method='bounded',bounds =␣
↪(Vu_min,Vu_max))
        V_star[V_i] = res.x
        C_new[V_i] = res.fun

        # Save the associated consumpton and search policy functions as well
```

(continues on next page)

(continued from previous page)

```
        cons_star[V_i] = cons_Vi_temp_interp(V_star[V_i])
        a_star[V_i] = a_Vi_temp_interp(V_star[V_i])

    return C_new,V_star,cons_star,a_star
```

The following code executes steps 4 and 5 in the Algorithm until convergence to a function $C^*(V)$.

```
def solve_incomplete_info_model(self,Vu_grid,Vu_aut,tol = 1e-6,max_iter = 10000):
    iter = 0
    error = 1

    C_init = np.ones(self.n_grid)*0
    C_old = np.copy(C_init)

    while iter<max_iter and error >tol:
        C_new,V_new,cons_star,a_star = iterate_C(self,C_old,Vu_grid)
        error = np.max(np.abs(C_new - C_old))

        #Only print the iterations every 50 steps
        if iter % 50 ==0:
            print(f"Iteration: {iter}, error:{error}")
        C_old = np.copy(C_new)
        iter+=1

    return C_new,V_new,cons_star,a_star
```

40.4 Outcomes

Using the above functions, we create another instance of the parameters with our calibrated parameter r.

```
##? Create another instance with the correct r now
params = params_instance(r = r_calibrated)

#Set up grid
Vu_min = Vu_aut
Vu_max = params.Ve - 1/(params.β*p_prime(0,params.r))
Vu_grid = np.linspace(Vu_min,Vu_max,params.n_grid)

#Solve model
C_star,V_star,cons_star,a_star = solve_incomplete_info_model(params,Vu_grid,Vu_aut,
 ↪tol = 1e-6,max_iter = 10000) #,cons_star,a_star

# Since we have the policy functions in grid form, we will interpolate them to be␣
 ↪able to
# evaluate any promised value
cons_star_interp = sp.interpolate.interp1d(Vu_grid,cons_star)
a_star_interp = sp.interpolate.interp1d(Vu_grid,a_star)
V_star_interp = sp.interpolate.interp1d(Vu_grid,V_star)
```

```
Iteration: 0, error:72.95964854907824
```

```
Iteration: 50, error:12.222761762480786
```

```
Iteration: 100, error:0.12875960366727668
```

```
Iteration: 150, error:0.0009402349710398994
```

```
Iteration: 200, error:6.115462838351959e-06
```

40.4.1 Replacement Ratios and Continuation Values

Let's graph the replacement ratio (c/w) and search effort a as functions of the duration of unemployment.

We'll do this for three levels of V_0, the lowest being the autarky value V_{aut}.

We accomplish this by using the optimal policy functions `V_star`, `cons_star` and `a_star` computed above as well the following iterative procedure:

```
# Replacement ratio and effort as a function of unemployment duration
T_max = 52
Vu_t = np.empty((T_max,3))
cons_t = np.empty((T_max-1,3))
a_t = np.empty((T_max-1,3))

# Calculate the replacement ratios depending on different initial
# promised values
Vu_0_hold = np.array([Vu_aut,16942,17000])
```

```
for i,Vu_0, in enumerate(Vu_0_hold):
    Vu_t[0,i] = Vu_0
    for t in range(1,T_max):
        cons_t[t-1,i] = cons_star_interp(Vu_t[t-1,i])
        a_t[t-1,i] = a_star_interp(Vu_t[t-1,i])
        Vu_t[t,i] = V_star_interp(Vu_t[t-1,i])
```

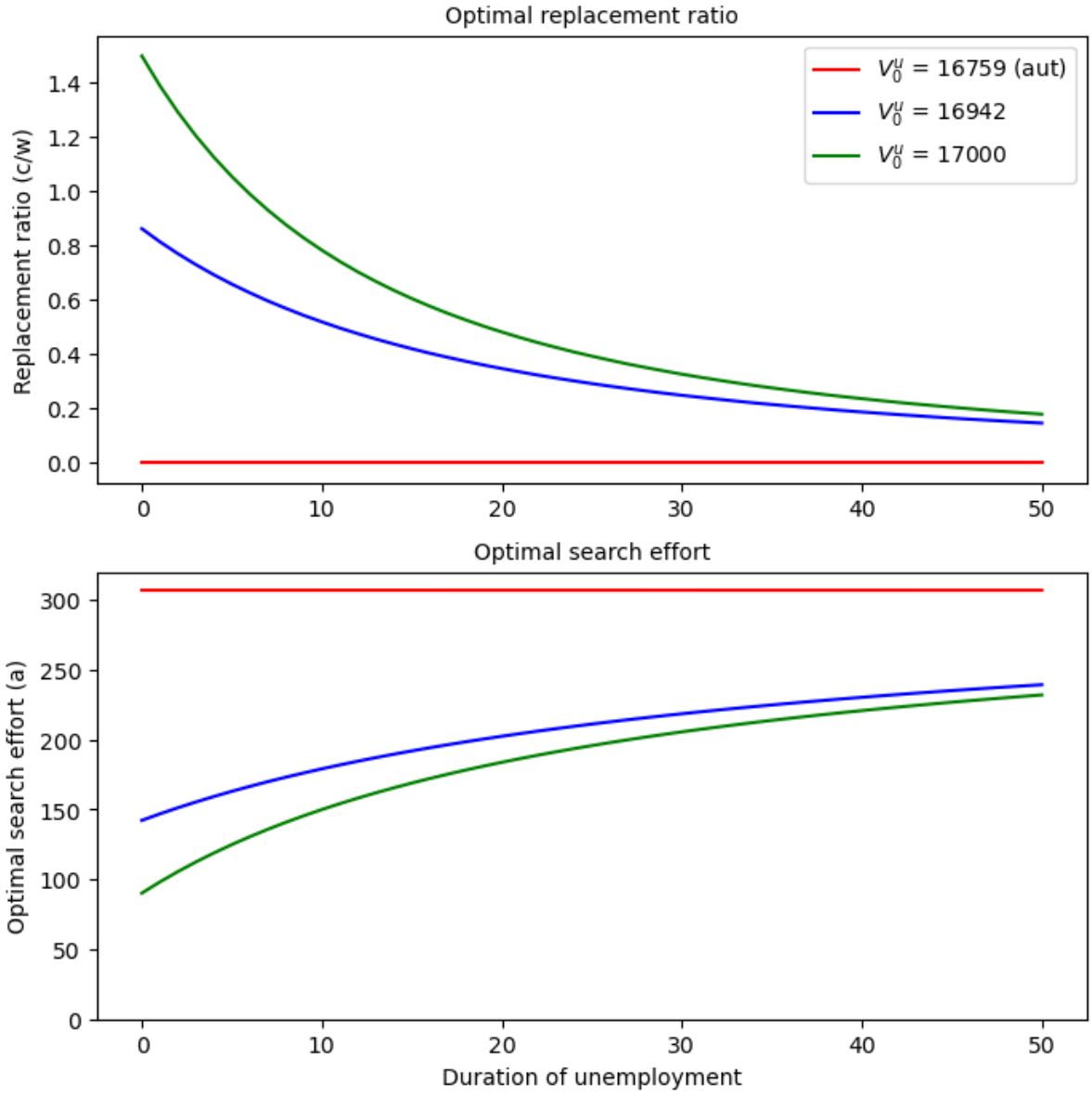
```
fontSize = 10
plt.rc('font', size=fontSize)          # controls default text sizes
plt.rc('axes', titlesize=fontSize)     # fontsize of the axes title
plt.rc('axes', labelsize=fontSize)    # fontsize of the x and y labels
plt.rc('xtick', labelsize=fontSize)    # fontsize of the tick labels
plt.rc('ytick', labelsize=fontSize)    # fontsize of the tick labels
plt.rc('legend', fontsize=fontSize)    # legend fontsize

f1 = plt.figure(figsize = (8,8))
plt.subplot(2,1,1)
plt.plot(range(T_max-1),cons_t[:,0]/params.w,label = '$V^u_0$ = 16759 (aut)',color =
 ↪'red')
plt.plot(range(T_max-1),cons_t[:,1]/params.w,label = '$V^u_0$ = 16942',color = 'blue')
plt.plot(range(T_max-1),cons_t[:,2]/params.w,label = '$V^u_0$ = 17000',color = 'green
 ↪')
plt.ylabel("Replacement ratio (c/w)")
plt.legend()
plt.title("Optimal replacement ratio")
```

(continues on next page)

(continued from previous page)

```
plt.subplot(2,1,2)
plt.plot(range(T_max-1),a_t[:,0],color = 'red')
plt.plot(range(T_max-1),a_t[:,1],color = 'blue')
plt.plot(range(T_max-1),a_t[:,2],color = 'green')
plt.ylim(0,320)
plt.ylabel("Optimal search effort (a)")
plt.xlabel("Duration of unemployment")
plt.title("Optimal search effort")
plt.show()
```



For an initial promised value $V^u = V_{\text{aut}}$, the planner chooses the autarky level of 0 for the replacement ratio and instructs the worker to search at the autarky search intensity, regardless of the duration of unemployment

But for $V^u > V_{\text{aut}}$, the planner makes the replacement ratio decline and search effort increase with the duration of

unemployment.

40.4.2 Interpretations

The downward slope of the replacement ratio when $V^u > V_{\text{aut}}$ is a consequence of the planner's limited information about the worker's search effort.

By providing the worker with a duration-dependent schedule of replacement ratios, the planner induces the worker in effect to reveal his/her search effort to the planner.

We saw earlier that with full information, the planner would smooth consumption over an unemployment spell by keeping the replacement ratio constant.

With private information, the planner can't observe the worker's search effort and therefore makes the replacement ratio fall.

Evidently, search effort rise as the duration of unemployment increases, especially early in an unemployment spell.

There is a **carrot-and-stick** aspect to the replacement rate and search effort schedules:

- the **carrot** occurs in the forms of high compensation and low search effort early in an unemployment spell.
- the **stick** occurs in the low compensation and high effort later in the spell.

We shall encounter a related carrot-and-stick feature in our other lectures about dynamic programming squared.

The planner offers declining benefits and induces increased search effort as the duration of an unemployment spell rises in order to provide an unemployed worker with proper incentives, not to punish an unlucky worker who has been unemployed for a long time.

The planner believes that a worker who has been unemployed a long time is unlucky, not that he has done anything wrong (e.g.,that he has not lived up to the contract).

Indeed, the contract is designed to induce the unemployed workers to search in the way the planner expects.

The falling consumption and rising search effort of the unlucky ones with long unemployment spells are simply costs that have to be paid in order to provide proper incentives.

CHAPTER

FORTYONE

STACKELBERG PLANS

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install --upgrade quantecon
```

41.1 Overview

This lecture formulates and computes a plan that a **Stackelberg leader** uses to manipulate forward-looking decisions of a **Stackelberg follower** that depend on continuation sequences of decisions made once and for all by the Stackelberg leader at time 0.

To facilitate computation and interpretation, we formulate things in a context that allows us to apply dynamic programming for linear-quadratic models.

Technically, our calculations are closely related to ones described this lecture.

From the beginning, we carry along a linear-quadratic model of duopoly in which firms face adjustment costs that make them want to forecast actions of other firms that influence future prices.

Let's start with some standard imports:

```
import numpy as np
import numpy.linalg as la
import quantecon as qe
from quantecon import LQ
import matplotlib.pyplot as plt
```

41.2 Duopoly

Time is discrete and is indexed by $t = 0, 1, \ldots$.

Two firms produce a single good whose demand is governed by the linear inverse demand curve

$$p_t = a_0 - a_1(q_{1t} + q_{2t})$$

where q_{it} is output of firm i at time t and a_0 and a_1 are both positive.

q_{10}, q_{20} are given numbers that serve as initial conditions at time 0.

By incurring a cost equal to

$$\gamma v_{it}^2, \quad \gamma > 0,$$

firm i can change its output according to

$$q_{it+1} = q_{it} + v_{it}$$

Firm i's profits at time t equal

$$\pi_{it} = p_t q_{it} - \gamma v_{it}^2$$

Firm i wants to maximize the present value of its profits

$$\sum_{t=0}^{\infty} \beta^t \pi_{it}$$

where $\beta \in (0, 1)$ is a time discount factor.

41.2.1 Stackelberg Leader and Follower

Each firm $i = 1, 2$ chooses a sequence $\vec{q}_i \equiv \{q_{it+1}\}_{t=0}^{\infty}$ once and for all at time 0.

We let firm 2 be a **Stackelberg leader** and firm 1 be a **Stackelberg follower**.

The leader firm 2 goes first and chooses $\{q_{2t+1}\}_{t=0}^{\infty}$ once and for all at time 0.

Knowing that firm 2 has chosen $\{q_{2t+1}\}_{t=0}^{\infty}$, the follower firm 1 goes second and chooses $\{q_{1t+1}\}_{t=0}^{\infty}$ once and for all at time 0.

In choosing $\vec{q}_2$, firm 2 takes into account that firm 1 will base its choice of $\vec{q}_1$ on firm 2's choice of $\vec{q}_2$.

41.2.2 Statement of Leader's and Follower's Problems

We can express firm 1's problem as

$$\max_{\vec{q}_1} \Pi_1(\vec{q}_1; \vec{q}_2)$$

where the appearance behind the semi-colon indicates that $\vec{q}_2$ is given.

Firm 1's problem induces the best response mapping

$$\vec{q}_1 = B(\vec{q}_2)$$

(Here B maps a sequence into a sequence)

The Stackelberg leader's problem is

$$\max_{\vec{q}_2} \Pi_2(B(\vec{q}_2), \vec{q}_2)$$

whose maximizer is a sequence $\vec{q}_2$ that depends on the initial conditions q_{10}, q_{20} and the parameters of the model a_0, a_1, γ.

This formulation captures key features of the model

- Both firms make once-and-for-all choices at time 0.
- This is true even though both firms are choosing sequences of quantities that are indexed by **time**.
- The Stackelberg leader chooses first **within time** 0, knowing that the Stackelberg follower will choose second **within time** 0.

While our abstract formulation reveals the timing protocol and equilibrium concept well, it obscures details that must be addressed when we want to compute and interpret a Stackelberg plan and the follower's best response to it.

To gain insights about these things, we study them in more detail.

41.2.3 Firms' Problems

Firm 1 acts as if firm 2's sequence $\{q_{2t+1}\}_{t=0}^{\infty}$ is given and beyond its control.

Firm 2 knows that firm 1 chooses second and takes this into account in choosing $\{q_{2t+1}\}_{t=0}^{\infty}$.

In the spirit of *working backward*, we study firm 1's problem first, taking $\{q_{2t+1}\}_{t=0}^{\infty}$ as given.

We can formulate firm 1's optimum problem in terms of the Lagrangian

$$L = \sum_{t=0}^{\infty} \beta^t \{a_0 q_{1t} - a_1 q_{1t}^2 - a_1 q_{1t} q_{2t} - \gamma v_{1t}^2 + \lambda_t [q_{1t} + v_{1t} - q_{1t+1}]\}$$

Firm 1 seeks a maximum with respect to $\{q_{1t+1}, v_{1t}\}_{t=0}^{\infty}$ and a minimum with respect to $\{\lambda_t\}_{t=0}^{\infty}$.

We approach this problem using methods described in [Ljungqvist and Sargent, 2018], chapter 2, appendix A and [Sargent, 1987], chapter IX.

First-order conditions for this problem are

$$\begin{aligned}
\frac{\partial L}{\partial q_{1t}} &= a_0 - 2a_1 q_{1t} - a_1 q_{2t} + \lambda_t - \beta^{-1}\lambda_{t-1} = 0, \quad t \geq 1 \\
\frac{\partial L}{\partial v_{1t}} &= -2\gamma v_{1t} + \lambda_t = 0, \quad t \geq 0
\end{aligned}$$

These first-order conditions and the constraint $q_{1t+1} = q_{1t} + v_{1t}$ can be rearranged to take the form

$$\begin{aligned}
v_{1t} &= \beta v_{1t+1} + \frac{\beta a_0}{2\gamma} - \frac{\beta a_1}{\gamma} q_{1t+1} - \frac{\beta a_1}{2\gamma} q_{2t+1} \\
q_{t+1} &= q_{1t} + v_{1t}
\end{aligned}$$

We can substitute the second equation into the first equation to obtain

$$(q_{1t+1} - q_{1t}) = \beta(q_{1t+2} - q_{1t+1}) + c_0 - c_1 q_{1t+1} - c_2 q_{2t+1}$$

where $c_0 = \frac{\beta a_0}{2\gamma}, c_1 = \frac{\beta a_1}{\gamma}, c_2 = \frac{\beta a_1}{2\gamma}$.

This equation can in turn be rearranged to become

$$-q_{1t} + (1 + \beta + c_1) q_{1t+1} - \beta q_{1t+2} = c_0 - c_2 q_{2t+1} \tag{41.1}$$

Equation (41.1) is a second-order difference equation in the sequence $\vec{q}_1$ whose solution we want.

It satisfies **two boundary conditions:**

- an initial condition that $q_{1,0}$, which is given
- a terminal condition requiring that $\lim_{T \to +\infty} \beta^T q_{1t}^2 < +\infty$

Using the lag operators described in [Sargent, 1987], chapter IX, difference equation (41.1) can be written as

$$\beta(1 - \frac{1 + \beta + c_1}{\beta} L + \beta^{-1} L^2) q_{1t+2} = -c_0 + c_2 q_{2t+1}$$

The polynomial in the lag operator on the left side can be **factored** as

$$(1 - \frac{1 + \beta + c_1}{\beta} L + \beta^{-1} L^2) = (1 - \delta_1 L)(1 - \delta_2 L) \tag{41.2}$$

where $0 < \delta_1 < 1 < \frac{1}{\sqrt{\beta}} < \delta_2$.

Because $\delta_2 > \frac{1}{\sqrt{\beta}}$ the operator $(1-\delta_2 L)$ contributes an **unstable** component if solved **backwards** but a **stable** component if solved **forwards**.

Mechanically, write

$$(1 - \delta_2 L) = -\delta_2 L(1 - \delta_2^{-1} L^{-1})$$

and compute the following inverse operator

$$\left[-\delta_2 L(1 - \delta_2^{-1} L^{-1})\right]^{-1} = -\delta_2 (1 - \delta_2{}^{-1})^{-1} L^{-1}$$

Operating on both sides of equation (41.2) with β^{-1} times this inverse operator gives the follower's decision rule for setting q_{1t+1} in the **feedback-feedforward** form

$$q_{1t+1} = \delta_1 q_{1t} - c_0 \delta_2^{-1} \beta^{-1} \frac{1}{1 - \delta_2^{-1}} + c_2 \delta_2^{-1} \beta^{-1} \sum_{j=0}^{\infty} \delta_2^j q_{2t+j+1}, \quad t \geq 0 \tag{41.3}$$

The problem of the Stackelberg leader firm 2 is to choose the sequence $\{q_{2t+1}\}_{t=0}^{\infty}$ to maximize its discounted profits

$$\sum_{t=0}^{\infty} \beta^t \{(a_0 - a_1(q_{1t} + q_{2t}))q_{2t} - \gamma(q_{2t+1} - q_{2t})^2\}$$

subject to the sequence of constraints (41.3) for $t \geq 0$.

We can put a sequence $\{\theta_t\}_{t=0}^{\infty}$ of Lagrange multipliers on the sequence of equations (41.3) and formulate the following Lagrangian for the Stackelberg leader firm 2's problem

$$\begin{aligned} \tilde{L} = & \sum_{t=0}^{\infty} \beta^t \{(a_0 - a_1(q_{1t} + q_{2t}))q_{2t} - \gamma(q_{2t+1} - q_{2t})^2\} \\ & + \sum_{t=0}^{\infty} \beta^t \theta_t \{\delta_1 q_{1t} - c_0 \delta_2^{-1} \beta^{-1} \frac{1}{1 - \delta_2^{-1}} + c_2 \delta_2^{-1} \beta^{-1} \sum_{j=0}^{\infty} \delta_2^{-j} q_{2t+j+1} - q_{1t+1}\} \end{aligned} \tag{41.4}$$

subject to initial conditions for q_{1t}, q_{2t} at $t = 0$.

Remarks: We have formulated the Stackelberg problem in a space of sequences.

The max-min problem associated with firm 2's Lagrangian (41.4) is unpleasant because the time t component of firm 2's payoff function depends on the entire future of its choices of $\{q_{2t+j}\}_{j=0}^{\infty}$.

This renders a direct attack on the problem in the space of sequences cumbersome.

Therefore, below we will formulate the Stackelberg leader's problem recursively.

We'll proceed by putting our duopoly model into a broader class of models with the same general structure.

41.3 Stackelberg Problem

We formulate a class of linear-quadratic Stackelberg leader-follower problems of which our duopoly model is an instance.

We use the optimal linear regulator (a.k.a. the linear-quadratic dynamic programming problem described in LQ Dynamic Programming problems) to represent a Stackelberg leader's problem recursively.

Let z_t be an $n_z \times 1$ vector of **natural state variables**.

Let x_t be an $n_x \times 1$ vector of endogenous forward-looking variables that are physically free to jump at t.

In our duopoly example $x_t = v_{1t}$, the time t decision of the Stackelberg **follower**.

Let u_t be a vector of decisions chosen by the Stackelberg leader at t.

The z_t vector is inherited from the past.

But x_t is a decision made by the Stackelberg follower at time t that is the follower's best response to the choice of an entire sequence of decisions made by the Stackelberg leader at time $t = 0$.

Let

$$y_t = \begin{bmatrix} z_t \\ x_t \end{bmatrix}$$

Represent the Stackelberg leader's one-period loss function as

$$r(y, u) = y' R y + u' Q u$$

Subject to an initial condition for z_0, but not for x_0, the Stackelberg leader wants to maximize

$$-\sum_{t=0}^{\infty} \beta^t r(y_t, u_t) \tag{41.5}$$

The Stackelberg leader faces the model

$$\begin{bmatrix} I & 0 \\ G_{21} & G_{22} \end{bmatrix} \begin{bmatrix} z_{t+1} \\ x_{t+1} \end{bmatrix} = \begin{bmatrix} \hat{A}_{11} & \hat{A}_{12} \\ \hat{A}_{21} & \hat{A}_{22} \end{bmatrix} \begin{bmatrix} z_t \\ x_t \end{bmatrix} + \hat{B} u_t \tag{41.6}$$

We assume that the matrix $\begin{bmatrix} I & 0 \\ G_{21} & G_{22} \end{bmatrix}$ on the left side of equation (41.6) is invertible, so that we can multiply both sides by its inverse to obtain

$$\begin{bmatrix} z_{t+1} \\ x_{t+1} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} z_t \\ x_t \end{bmatrix} + B u_t \tag{41.7}$$

or

$$y_{t+1} = A y_t + B u_t \tag{41.8}$$

41.3.1 Interpretation of Second Block of Equations

The Stackelberg follower's best response mapping is summarized by the second block of equations of (41.7).

In particular, these equations are the first-order conditions of the Stackelberg follower's optimization problem (i.e., its Euler equations).

These Euler equations summarize the forward-looking aspect of the follower's behavior and express how its time t decision depends on the leader's actions at times $s \geq t$.

When combined with a stability condition to be imposed below, the Euler equations summarize the follower's best response to the sequence of actions by the leader.

The Stackelberg leader maximizes (41.5) by choosing sequences $\{u_t, x_t, z_{t+1}\}_{t=0}^{\infty}$ subject to (41.8) and an initial condition for z_0.

Note that we have an initial condition for z_0 but not for x_0.

x_0 is among the variables to be chosen at time 0 by the Stackelberg leader.

The Stackelberg leader uses its understanding of the responses restricted by (41.8) to manipulate the follower's decisions.

41.3.2 More Mechanical Details

For any vector a_t, define $\vec{a}_t = [a_t, a_{t+1} ...]$.

Define a feasible set of $(\vec{y}_1, \vec{u}_0)$ sequences

$$\Omega(y_0) = \{(\vec{y}_1, \vec{u}_0) : y_{t+1} = Ay_t + Bu_t, \forall t \geq 0\}$$

Please remember that the follower's system of Euler equations is embedded in the system of dynamic equations $y_{t+1} = Ay_t + Bu_t$.

Note that the definition of $\Omega(y_0)$ treats y_0 as given.

Although it is taken as given in $\Omega(y_0)$, eventually, the x_0 component of y_0 is to be chosen by the Stackelberg leader.

41.3.3 Two Subproblems

Once again we use backward induction.

We express the Stackelberg problem in terms of **two subproblems**.

Subproblem 1 is solved by a **continuation Stackelberg leader** at each date $t \geq 0$.

Subproblem 2 is solved by the **Stackelberg leader** at $t = 0$.

The two subproblems are designed

- to respect the timing protocol in which the follower chooses $\vec{q}_1$ after seeing $\vec{q}_2$ chosen by the leader
- to make the leader choose $\vec{q}_2$ while respecting that $\vec{q}_1$ will be the follower's best response to $\vec{q}_2$
- to represent the leader's problem recursively by artfully choosing the leader's state variables and the control variables available to the leader

Subproblem 1

$$v(y_0) = \max_{(\vec{y}_1, \vec{u}_0) \in \Omega(y_0)} -\sum_{t=0}^{\infty} \beta^t r(y_t, u_t)$$

Subproblem 2

$$w(z_0) = \max_{x_0} v(y_0)$$

Subproblem 1 takes the vector of forward-looking variables x_0 as given.

Subproblem 2 optimizes over x_0.

The value function $w(z_0)$ tells the value of the Stackelberg plan as a function of the vector of natural state variables z_0 at time 0.

41.4 Two Bellman Equations

We now describe Bellman equations for $v(y)$ and $w(z_0)$.

Subproblem 1

The value function $v(y)$ in subproblem 1 satisfies the Bellman equation

$$v(y) = \max_{u, y^*} \{-r(y, u) + \beta v(y^*)\} \tag{41.9}$$

where the maximization is subject to

$$y^* = Ay + Bu$$

and y^* denotes next period's value.

Substituting $v(y) = -y'Py$ into Bellman equation (41.9) gives

$$-y'Py = \max_{u,y^*} \{-y'Ry - u'Qu - \beta y^{*\prime} P y^*\}$$

which as in lecture linear regulator gives rise to the algebraic matrix Riccati equation

$$P = R + \beta A'PA - \beta^2 A'PB(Q + \beta B'PB)^{-1}B'PA$$

and the optimal decision rule coefficient vector

$$F = \beta(Q + \beta B'PB)^{-1}B'PA$$

where the optimal decision rule is

$$u_t = -Fy_t$$

Subproblem 2

We find an optimal x_0 by equating to zero the gradient of $v(y_0)$ with respect to x_0:

$$-2P_{21}z_0 - 2P_{22}x_0 = 0,$$

which implies that

$$x_0 = -P_{22}^{-1}P_{21}z_0 \tag{41.10}$$

41.5 Stackelberg Plan for Duopoly

Now let's map our duopoly model into the above setup.

We formulate a state vector

$$y_t = \begin{bmatrix} z_t \\ x_t \end{bmatrix}$$

where for our duopoly model

$$z_t = \begin{bmatrix} 1 \\ q_{2t} \\ q_{1t} \end{bmatrix}, \quad x_t = v_{1t},$$

where $x_t = v_{1t}$ is the time t decision of the follower firm 1, u_t is the time t decision of the leader firm 2 and

$$v_{1t} = q_{1t+1} - q_{1t}, \quad u_t = q_{2t+1} - q_{2t}.$$

For our duopoly model, initial conditions for the natural state variables in z_t are

$$z_0 = \begin{bmatrix} 1 \\ q_{20} \\ q_{10} \end{bmatrix}$$

while $x_0 = v_{10} = q_{11} - q_{10}$ is a choice variable for the Stackelberg leader firm 2, one that will ultimately be chosen according an optimal rule prescribed by (41.10) for subproblem 2 above.

That the Stackelberg leader firm 2 chooses $x_0 = v_{10}$ is subtle.

Of course, $x_0 = v_{10}$ emerges from the feedback-feedforward solution (41.3) of firm 1's system of Euler equations, so that it is actually firm 1 that sets x_0.

But firm 2 manipulates firm 1's choice through firm 2's choice of the sequence $\vec{q}_{2,1} = \{q_{2t+1}\}_{t=0}^{\infty}$.

41.5.1 Calculations to Prepare Duopoly Model

Now we'll proceed to cast our duopoly model within the framework of the more general linear-quadratic structure described above.

That will allow us to compute a Stackelberg plan simply by enlisting a Riccati equation to solve a linear-quadratic dynamic program.

As emphasized above, firm 1 acts as if firm 2's decisions $\{q_{2t+1}, v_{2t}\}_{t=0}^{\infty}$ are given and beyond its control.

41.5.2 Firm 1's Problem

We again formulate firm 1's optimum problem in terms of the Lagrangian

$$L = \sum_{t=0}^{\infty} \beta^t \{a_0 q_{1t} - a_1 q_{1t}^2 - a_1 q_{1t} q_{2t} - \gamma v_{1t}^2 + \lambda_t [q_{1t} + v_{1t} - q_{1t+1}]\}$$

Firm 1 seeks a maximum with respect to $\{q_{1t+1}, v_{1t}\}_{t=0}^{\infty}$ and a minimum with respect to $\{\lambda_t\}_{t=0}^{\infty}$.

First-order conditions for this problem are

$$\begin{aligned}
\frac{\partial L}{\partial q_{1t}} &= a_0 - 2a_1 q_{1t} - a_1 q_{2t} + \lambda_t - \beta^{-1}\lambda_{t-1} = 0, \quad t \geq 1 \\
\frac{\partial L}{\partial v_{1t}} &= -2\gamma v_{1t} + \lambda_t = 0, \quad t \geq 0
\end{aligned}$$

These first-order order conditions and the constraint $q_{1t+1} = q_{1t} + v_{1t}$ can be rearranged to take the form

$$\begin{aligned}
v_{1t} &= \beta v_{1t+1} + \frac{\beta a_0}{2\gamma} - \frac{\beta a_1}{\gamma} q_{1t+1} - \frac{\beta a_1}{2\gamma} q_{2t+1} \\
q_{t+1} &= q_{1t} + v_{1t}
\end{aligned}$$

We use these two equations as components of the following linear system that confronts a Stackelberg continuation leader at time t

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \frac{\beta a_0}{2\gamma} & -\frac{\beta a_1}{2\gamma} & -\frac{\beta a_1}{\gamma} & \beta \end{bmatrix} \begin{bmatrix} 1 \\ q_{2t+1} \\ q_{1t+1} \\ v_{1t+1} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ q_{2t} \\ q_{1t} \\ v_{1t} \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} v_{2t}$$

Time t revenues of firm 2 are $\pi_{2t} = a_0 q_{2t} - a_1 q_{2t}^2 - a_1 q_{1t} q_{2t}$ which evidently equal

$$z_t' R_1 z_t \equiv \begin{bmatrix} 1 \\ q_{2t} \\ q_{1t} \end{bmatrix}' \begin{bmatrix} 0 & \frac{a_0}{2} & 0 \\ \frac{a_0}{2} & -a_1 & -\frac{a_1}{2} \\ 0 & -\frac{a_1}{2} & 0 \end{bmatrix} \begin{bmatrix} 1 \\ q_{2t} \\ q_{1t} \end{bmatrix}$$

If we set $Q = \gamma$, then firm 2's period t profits can then be written

$$y_t' R y_t - Q v_{2t}^2$$

where

$$y_t = \begin{bmatrix} z_t \\ x_t \end{bmatrix}$$

with $x_t = v_{1t}$ and

$$R = \begin{bmatrix} R_1 & 0 \\ 0 & 0 \end{bmatrix}$$

We'll report results of implementing this code soon.

But first, we want to represent the Stackelberg leader's optimal choices recursively.

It is important to do this for several reasons:

- properly to interpret a representation of the Stackelberg leader's choice as a sequence of history-dependent functions
- to formulate a recursive version of the follower's choice problem

First, let's get a recursive representation of the Stackelberg leader's choice of $\vec{q}_2$ for our duopoly model.

41.6 Recursive Representation of Stackelberg Plan

In order to attain an appropriate representation of the Stackelberg leader's history-dependent plan, we will employ what amounts to a version of the **Big K, little k** device often used in macroeconomics by distinguishing z_t, which depends partly on decisions x_t of the followers, from another vector $\check{z}_t$, which does not.

We will use $\check{z}_t$ and its history $\check{z}^t = [\check{z}_t, \check{z}_{t-1}, \ldots, \check{z}_0]$ to describe the sequence of the Stackelberg leader's decisions that the Stackelberg follower takes as given.

Thus, we let $\check{y}_t' = \begin{bmatrix} \check{z}_t' & \check{x}_t' \end{bmatrix}$ with initial condition $\check{z}_0 = z_0$ given.

That we distinguish $\check{z}_t$ from z_t is part and parcel of the **Big K, little k** device in this instance.

We have demonstrated that a Stackelberg plan for $\{u_t\}_{t=0}^{\infty}$ has a recursive representation

$$\begin{aligned} \check{x}_0 &= -P_{22}^{-1} P_{21} z_0 \\ u_t &= -F \check{y}_t, \quad t \geq 0 \\ \check{y}_{t+1} &= (A - BF) \check{y}_t, \quad t \geq 0 \end{aligned}$$

From this representation, we can deduce the sequence of functions $\sigma = \{\sigma_t(\check{z}^t)\}_{t=0}^{\infty}$ that comprise a Stackelberg plan.

For convenience, let $\check{A} \equiv A - BF$ and partition $\check{A}$ conformably to the partition $y_t = \begin{bmatrix} \check{z}_t \\ \check{x}_t \end{bmatrix}$ as

$$\begin{bmatrix} \check{A}_{11} & \check{A}_{12} \\ \check{A}_{21} & \check{A}_{22} \end{bmatrix}$$

Let $H_0^0 \equiv -P_{22}^{-1} P_{21}$ so that $\check{x}_0 = H_0^0 \check{z}_0$.

Then iterations on $\check{y}_{t+1} = \check{A} \check{y}_t$ starting from initial condition $\check{y}_0 = \begin{bmatrix} \check{z}_0 \\ H_0^0 \check{z}_0 \end{bmatrix}$ imply that for $t \geq 1$

$$\check{x}_t = \sum_{j=1}^{t} H_j^t \check{z}_{t-j}$$

where

$$\begin{aligned} H_1^t &= \check{A}_{21} \\ H_2^t &= \check{A}_{22} \check{A}_{21} \\ &\vdots \quad \vdots \\ H_{t-1}^t &= \check{A}_{22}^{t-2} \check{A}_{21} \\ H_t^t &= \check{A}_{22}^{t-1} (\check{A}_{21} + \check{A}_{22} H_0^0) \end{aligned}$$

An optimal decision rule for the Stackelberg leader's choice of u_t is

$$u_t = -F\check{y}_t \equiv -\begin{bmatrix} F_z & F_x \end{bmatrix} \begin{bmatrix} \check{z}_t \\ x_t \end{bmatrix}$$

or

$$u_t = -F_z\check{z}_t - F_x \sum_{j=1}^{t} H_j^t z_{t-j} = \sigma_t(\check{z}^t) \tag{41.11}$$

Representation (41.11) confirms that whenever $F_x \neq 0$, the typical situation, the time t component σ_t of a Stackelberg plan is **history-dependent**, meaning that the Stackelberg leader's choice u_t depends not just on $\check{z}_t$ but on components of $\check{z}^{t-1}$.

41.6.1 Comments and Interpretations

Because we set $\check{z}_0 = z_0$, it will turn out that $z_t = \check{z}_t$ for all $t \geq 0$.

Then why did we distinguish $\check{z}_t$ from z_t?

The answer is that if we want to present to the Stackelberg **follower** a history-dependent representation of the Stackelberg **leader's** sequence $\vec{q}_2$, we must use representation (41.11) cast in terms of the history $\check{z}^t$ and **not** a corresponding representation cast in terms of z^t.

41.7 Dynamic Programming and Time Consistency of Follower's Problem

Given the sequence $\vec{q}_2$ chosen by the Stackelberg leader in our duopoly model, it turns out that the Stackelberg **follower's** problem is recursive in the *natural* state variables that confront a follower at any time $t \geq 0$.

This means that the follower's plan is time consistent.

To verify these claims, we'll formulate a recursive version of a follower's problem that builds on our recursive representation of the Stackelberg leader's plan and our use of the **Big K, little k** idea.

41.7.1 Recursive Formulation of a Follower's Problem

We now use what amounts to another "Big K, little k" trick (see rational expectations equilibrium) to formulate a recursive version of a follower's problem cast in terms of an ordinary Bellman equation.

Firm 1, the follower, faces $\{q_{2t}\}_{t=0}^{\infty}$ as a given quantity sequence chosen by the leader and believes that its output price at t satisfies

$$p_t = a_0 - a_1(q_{1t} + q_{2t}), \quad t \geq 0$$

Our challenge is to represent $\{q_{2t}\}_{t=0}^{\infty}$ as a given sequence.

To do so, recall that under the Stackelberg plan, firm 2 sets output according to the q_{2t} component of

$$y_{t+1} = \begin{bmatrix} 1 \\ q_{2t} \\ q_{1t} \\ x_t \end{bmatrix}$$

which is governed by

$$y_{t+1} = (A - BF)y_t$$

To obtain a recursive representation of a $\{q_{2t}\}$ sequence that is exogenous to firm 1, we define a state $\tilde{y}_t$

$$\tilde{y}_t = \begin{bmatrix} 1 \\ q_{2t} \\ \tilde{q}_{1t} \\ \tilde{x}_t \end{bmatrix}$$

that evolves according to

$$\tilde{y}_{t+1} = (A - BF)\tilde{y}_t$$

subject to the initial condition $\tilde{q}_{10} = q_{10}$ and $\tilde{x}_0 = x_0$ where $x_0 = -P_{22}^{-1}P_{21}$ as stated above.

Firm 1's state vector is

$$X_t = \begin{bmatrix} \tilde{y}_t \\ q_{1t} \end{bmatrix}$$

It follows that the follower firm 1 faces law of motion

$$\begin{bmatrix} \tilde{y}_{t+1} \\ q_{1t+1} \end{bmatrix} = \begin{bmatrix} A - BF & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \tilde{y}_t \\ q_{1t} \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} x_t \tag{41.12}$$

This specification assures that from the point of the view of firm 1, q_{2t} is an exogenous process.

Here

- $\tilde{q}_{1t}, \tilde{x}_t$ play the role of **Big K**
- q_{1t}, x_t play the role of **little k**

The time t component of firm 1's objective is

$$\tilde{X}_t' \tilde{R} x_t - x_t^2 \tilde{Q} = \begin{bmatrix} 1 \\ q_{2t} \\ \tilde{q}_{1t} \\ \tilde{x}_t \\ q_{1t} \end{bmatrix}' \begin{bmatrix} 0 & 0 & 0 & 0 & \frac{a_0}{2} \\ 0 & 0 & 0 & 0 & -\frac{a_1}{2} \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ \frac{a_0}{2} & -\frac{a_1}{2} & 0 & 0 & -a_1 \end{bmatrix} \begin{bmatrix} 1 \\ q_{2t} \\ \tilde{q}_{1t} \\ \tilde{x}_t \\ q_{1t} \end{bmatrix} - \gamma x_t^2$$

Firm 1's optimal decision rule is

$$x_t = -\tilde{F} X_t$$

and its state evolves according to

$$\tilde{X}_{t+1} = (\tilde{A} - \tilde{B}\tilde{F})X_t$$

under its optimal decision rule.

Later we shall compute $\tilde{F}$ and verify that when we set

$$X_0 = \begin{bmatrix} 1 \\ q_{20} \\ q_{10} \\ x_0 \\ q_{10} \end{bmatrix}$$

we recover

$$x_0 = -\tilde{F}\tilde{X}_0,$$

which will verify that we have properly set up a recursive representation of the follower's problem facing the Stackelberg leader's $\vec{q}_2$.

41.7.2 Time Consistency of Follower's Plan

The follower can solve its problem using dynamic programming because its problem is recursive in what for it are the **natural state variables**, namely

$$\begin{bmatrix} 1 \\ q_{2t} \\ \tilde{q}_{1t} \\ \tilde{x}_t \end{bmatrix}$$

It follows that the follower's plan is time consistent.

41.8 Computing Stackelberg Plan

Here is our code to compute a Stackelberg plan via the linear-quadratic dynamic program describe above.

Let's use it to compute the Stackelberg plan.

```
# Parameters
a0 = 10
a1 = 2
β = 0.96
γ = 120
n = 300
tol0 = 1e-8
tol1 = 1e-16
tol2 = 1e-2

βs = np.ones(n)
βs[1:] = β
βs = βs.cumprod()
```

```
# In LQ form
Alhs = np.eye(4)

# Euler equation coefficients
Alhs[3, :] = β * a0 / (2 * γ), -β * a1 / (2 * γ), -β * a1 / γ, β

Arhs = np.eye(4)
Arhs[2, 3] = 1

Alhsinv = la.inv(Alhs)

A = Alhsinv @ Arhs

B = Alhsinv @ np.array([[0, 1, 0, 0]]).T

R = np.array([[0,      -a0 / 2,      0, 0],
              [-a0 / 2,     a1, a1 / 2, 0],
              [0,       a1 / 2,      0, 0],
              [0,            0,      0, 0]])

Q = np.array([[γ]])

# Solve using QE's LQ class
```

(continues on next page)

(continued from previous page)

```
# LQ solves minimization problems which is why the sign of R and Q was changed
lq = LQ(Q, R, A, B, beta=β)
P, F, d = lq.stationary_values(method='doubling')

P22 = P[3:, 3:]
P21 = P[3:, :3]
P22inv = la.inv(P22)
H_0_0 = -P22inv @ P21

# Simulate forward

π_leader = np.zeros(n)

z0 = np.array([[1, 1, 1]]).T
x0 = H_0_0 @ z0
y0 = np.vstack((z0, x0))

yt, ut = lq.compute_sequence(y0, ts_length=n)[:2]

π_matrix = (R + F. T @ Q @ F)

for t in range(n):
    π_leader[t] = -(yt[:, t].T @ π_matrix @ yt[:, t])

# Display policies
print("Computed policy for Continuation Stackelberg leader\n")
print(f"F = {F}")
```

```
Computed policy for Continuation Stackelberg leader

F = [[-1.58004454  0.29461313  0.67480938  6.53970594]]
```

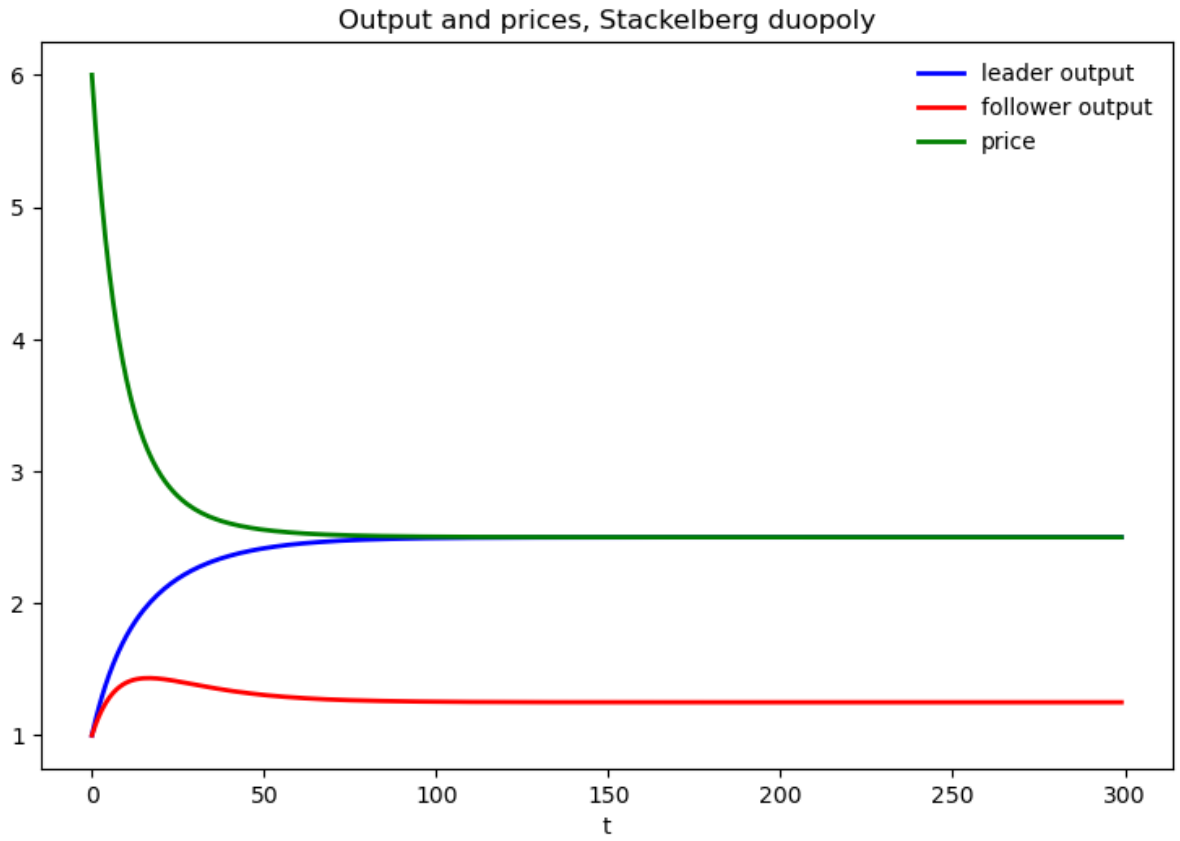
41.9 Time Series for Price and Quantities

Now let's use the code to compute and display outcomes as a Stackelberg plan unfolds.

The following code plots quantities chosen by the Stackelberg leader and follower, together with the equilibrium output price.

```
q_leader = yt[1, :-1]
q_follower = yt[2, :-1]
q = q_leader + q_follower       # Total output, Stackelberg
p = a0 - a1 * q                 # Price, Stackelberg

fig, ax = plt.subplots(figsize=(9, 5.8))
ax.plot(range(n), q_leader, 'b-', lw=2, label='leader output')
ax.plot(range(n), q_follower, 'r-', lw=2, label='follower output')
ax.plot(range(n), p, 'g-', lw=2, label='price')
ax.set_title('Output and prices, Stackelberg duopoly')
ax.legend(frameon=False)
ax.set_xlabel('t')
plt.show()
```



41.9.1 Value of Stackelberg Leader

We'll compute the value $w(x_0)$ attained by the Stackelberg leader, where x_0 is given by the maximizer (41.10) of sub-problem 2.

We'll compute it two ways and get the same answer.

In addition to being a useful check on the accuracy of our coding, computing things in these two ways helps us think about the structure of the problem.

```
v_leader_forward = np.sum(βs * π_leader)
v_leader_direct = -yt[:, 0].T @ P @ yt[:, 0]

# Display values
print("Computed values for the Stackelberg leader at t=0:\n")
print(f"v_leader_forward(forward sim) = {v_leader_forward:.4f}")
print(f"v_leader_direct (direct) = {v_leader_direct:.4f}")
```

```
Computed values for the Stackelberg leader at t=0:

v_leader_forward(forward sim) = 150.0316
v_leader_direct (direct) = 150.0324
```

```
# Manually checks whether P is approximately a fixed point
P_next = (R + F.T @ Q @ F + β * (A - B @ F).T @ P @ (A - B @ F))
(P - P_next < tol0).all()
```

```
True
```

```
# Manually checks whether two different ways of computing the
# value function give approximately the same answer
v_expanded = -((y0.T @ R @ y0 + ut[:, 0].T @ Q @ ut[:, 0] +
                β * (y0.T @ (A - B @ F).T @ P @ (A - B @ F) @ y0)))
(v_leader_direct - v_expanded < tol0)[0, 0]
```

```
True
```

41.10 Time Inconsistency of Stackelberg Plan

In the code below we compare two values

- the continuation value $v(y_t) = -y_t' P y_t$ earned by a **continuation Stackelberg leader** who inherits state y_t at t
- the value $w(\hat{x}_t)$ of a **reborn Stackelberg leader** who, at date t along the Stackelberg plan, inherits state z_t at t but who discards x_t from the time t continuation of the original Stackelberg plan and **resets** it to $\hat{x}_t = -P_{22}^{-1} P_{21} z_t$

The difference between these two values is a tell-tale sign of the time inconsistency of the Stackelberg plan

```
# Compute value function over time with a reset at time t
vt_leader = np.zeros(n)
vt_reset_leader = np.empty_like(vt_leader)

yt_reset = yt.copy()
yt_reset[-1, :] = (H_0_0 @ yt[:3, :])

for t in range(n):
    vt_leader[t] = -yt[:, t].T @ P @ yt[:, t]
    vt_reset_leader[t] = -yt_reset[:, t].T @ P @ yt_reset[:, t]
```

```
fig, axes = plt.subplots(3, 1, figsize=(10, 7))

axes[0].plot(range(n+1), (- F @ yt).flatten(), 'bo',
    label='Stackelberg leader', ms=2)
axes[0].plot(range(n+1), (- F @ yt_reset).flatten(), 'ro',
    label='reborn  at t Stackelberg leader', ms=2)
axes[0].set(title=r' $u_{t} = q_{2t+1} - q_t$', xlabel='t')
axes[0].legend()

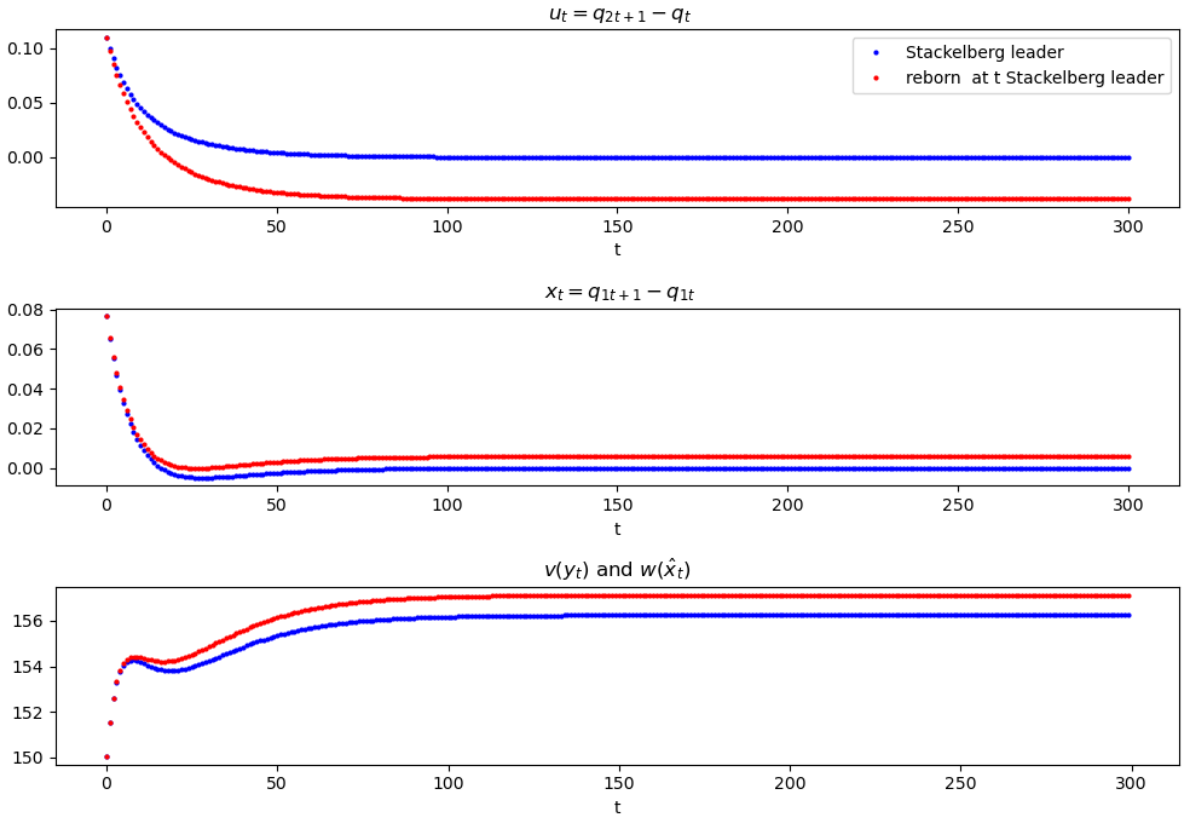
axes[1].plot(range(n+1), yt[3, :], 'bo', ms=2)
axes[1].plot(range(n+1), yt_reset[3, :], 'ro', ms=2)
axes[1].set(title=r' $x_{t} = q_{1t+1} - q_{1t}$', xlabel='t')

axes[2].plot(range(n), vt_leader, 'bo', ms=2)
axes[2].plot(range(n), vt_reset_leader, 'ro', ms=2)
axes[2].set(title=r'$v(y_{t})$ and $w(\hat x_t)$', xlabel='t')
```

(continues on next page)

(continued from previous page)

```
plt.tight_layout()
plt.show()
```



The figure above shows

- in the third panel that for $t \geq 1$ the **reborn at** t Stackelberg leader's's value $w(\hat{x}_0)$ exceeds the continuation value $v(y_t)$ of the time 0 Stackelberg leader
- in the first panel that for $t \geq 1$ the **reborn at** t Stackelberg leader wants to reduce his output below that prescribed by the time 0 Stackelberg leader
- in the second panel that for $t \geq 1$ the **reborn at** t Stackelberg leader wants to increase the output of the follower firm 2 below that prescribed by the time 0 Stackelberg leader

Taken together, these outcomes express the time inconsistency of the original time 0 Stackelberg leaders's plan.

41.11 Recursive Formulation of Follower's Problem

We now formulate and compute the recursive version of the follower's problem.

We check that the recursive **Big** K **, little** k formulation of the follower's problem produces the same output path $\vec{q}_1$ that we computed when we solved the Stackelberg problem

```
A_tilde = np.eye(5)
A_tilde[:4, :4] = A - B @ F
```

(continues on next page)

(continued from previous page)

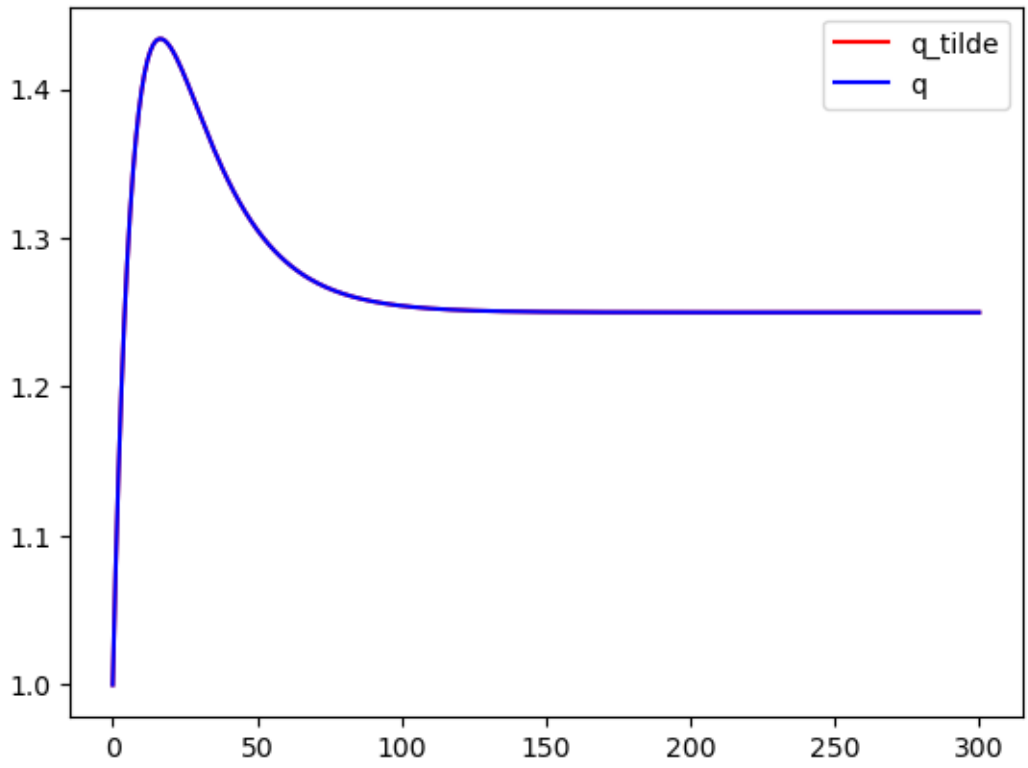
```
R_tilde = np.array([[0,             0, 0,    0, -a0 / 2],
                    [0,             0, 0,    0,  a1 / 2],
                    [0,             0, 0,    0,       0],
                    [0,             0, 0,    0,       0],
                    [-a0 / 2, a1 / 2, 0,    0,      a1]])

Q_tilde = Q
B_tilde = np.array([[0, 0, 0, 0, 1]]).T

lq_tilde = LQ(Q_tilde, R_tilde, A_tilde, B_tilde, beta=β)
P_tilde, F_tilde, d_tilde = lq_tilde.stationary_values(method='doubling')

y0_tilde = np.vstack((y0, y0[2]))
yt_tilde = lq_tilde.compute_sequence(y0_tilde, ts_length=n)[0]
```

```
# Checks that the recursive formulation of the follower's problem gives
# the same solution as the original Stackelberg problem
fig, ax = plt.subplots()
ax.plot(yt_tilde[4], 'r', label="q_tilde")
ax.plot(yt_tilde[2], 'b', label="q")
ax.legend()
plt.show()
```



Note: Variables with `_tilde` are obtained from solving the follower's problem – those without are from the Stackelberg problem

```
# Maximum absolute difference in quantities over time between
# the first and second solution methods
np.max(np.abs(yt_tilde[4] - yt_tilde[2]))
```

```
4.440892098500626e-16
```

```
# x0 == x0_tilde
yt[:, 0][-1] - (yt_tilde[:, 1] - yt_tilde[:, 0])[-1] < tol0
```

```
True
```

41.11.1 Explanation of Alignment

If we inspect coefficients in the decision rule $-\tilde{F}$, we should be able to spot why the follower chooses to set $x_t = \tilde{x}_t$ when it sets $x_t = -\tilde{F} X_t$ in the recursive formulation of the follower problem.

Can you spot what features of $\tilde{F}$ imply this?

Hint: Remember the components of X_t

```
# Policy function in the follower's problem
F_tilde.round(4)
```

```
array([[ 0.    , -0.    , -0.1032, -1.    ,  0.1032]])
```

```
# Value function in the Stackelberg problem
P
```

```
array([[  963.54083615,  -194.60534465,  -511.62197962, -5258.22585724],
       [ -194.60534465,    37.3535753 ,    81.97712513,   784.76471234],
       [ -511.62197962,    81.97712513,   247.34333344,  2517.05126111],
       [-5258.22585724,   784.76471234,  2517.05126111, 25556.16504097]])
```

```
# Value function in the follower's problem
P_tilde
```

```
array([[-1.81991134e+01,  2.58003020e+00,  1.56048755e+01,
         1.51229815e+02, -5.00000000e+00],
       [ 2.58003020e+00, -9.69465925e-01, -5.26007958e+00,
        -5.09764310e+01,  1.00000000e+00],
       [ 1.56048755e+01, -5.26007958e+00, -3.22759027e+01,
        -3.12791908e+02, -1.23823802e+01],
       [ 1.51229815e+02, -5.09764310e+01, -3.12791908e+02,
        -3.03132584e+03, -1.20000000e+02],
       [-5.00000000e+00,  1.00000000e+00, -1.23823802e+01,
        -1.20000000e+02,  1.43823802e+01]])
```

```
# Manually check that P is an approximate fixed point
(P  - ((R + F.T @ Q @ F) + β * (A - B @ F).T @ P @ (A - B @ F)) < tol0).all()
```

```
True
```

```
# Compute `P_guess` using `F_tilde_star`
F_tilde_star = -np.array([[0, 0, 0, 1, 0]])
P_guess = np.zeros((5, 5))

for i in range(1000):
    P_guess = ((R_tilde + F_tilde_star.T @ Q @ F_tilde_star) +
               β * (A_tilde - B_tilde @ F_tilde_star).T @ P_guess
               @ (A_tilde - B_tilde @ F_tilde_star))
```

```
# Value function in the follower's problem
-(y0_tilde.T @ P_tilde @ y0_tilde)[0, 0]
```

```
112.65590740578115
```

```
# Value function with `P_guess`
-(y0_tilde.T @ P_guess @ y0_tilde)[0, 0]
```

```
112.65590740578136
```

```
# Compute policy using policy iteration algorithm
F_iter = (β * la.inv(Q + β * B_tilde.T @ P_guess @ B_tilde)
          @ B_tilde.T @ P_guess @ A_tilde)

for i in range(100):
    # Compute P_iter
    P_iter = np.zeros((5, 5))
    for j in range(1000):
        P_iter = ((R_tilde + F_iter.T @ Q @ F_iter) + β
                  * (A_tilde - B_tilde @ F_iter).T @ P_iter
                  @ (A_tilde - B_tilde @ F_iter))

    # Update F_iter
    F_iter = (β * la.inv(Q + β * B_tilde.T @ P_iter @ B_tilde)
              @ B_tilde.T @ P_iter @ A_tilde)

dist_vec = (P_iter - ((R_tilde + F_iter.T @ Q @ F_iter)
            + β * (A_tilde - B_tilde @ F_iter).T @ P_iter
            @ (A_tilde - B_tilde @ F_iter)))

if np.max(np.abs(dist_vec)) < 1e-8:
    dist_vec2 = (F_iter - (β * la.inv(Q + β * B_tilde.T @ P_iter @ B_tilde)
                 @ B_tilde.T @ P_iter @ A_tilde))

    if np.max(np.abs(dist_vec2)) < 1e-8:
        F_iter
    else:
        print("The policy didn't converge: try increasing the number of \
```

(continues on next page)

(continued from previous page)

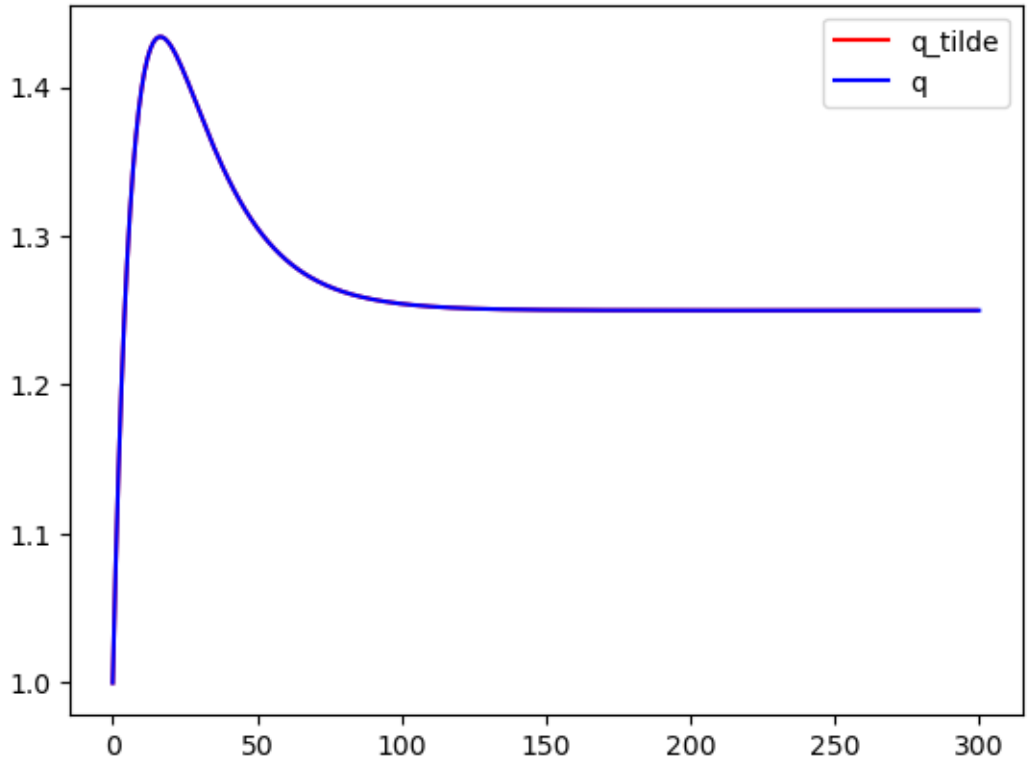
```
            outer loop iterations")
else:
    print("`P_iter` didn't converge: try increasing the number of inner \
        loop iterations")
```

```
# Simulate the system using `F_tilde_star` and check that it gives the
# same result as the original solution

yt_tilde_star = np.zeros((n, 5))
yt_tilde_star[0, :] = y0_tilde.flatten()

for t in range(n-1):
    yt_tilde_star[t+1, :] = (A_tilde - B_tilde @ F_tilde_star) \
        @ yt_tilde_star[t, :]

fig, ax = plt.subplots()
ax.plot(yt_tilde_star[:, 4], 'r', label="q_tilde")
ax.plot(yt_tilde[2], 'b', label="q")
ax.legend()
plt.show()
```




```
# Maximum absolute difference
np.max(np.abs(yt_tilde_star[:, 4] - yt_tilde[2, :-1]))
```

```
0.0
```

41.12 Markov Perfect Equilibrium

The **state** vector is

$$z_t = \begin{bmatrix} 1 \\ q_{2t} \\ q_{1t} \end{bmatrix}$$

and the state transition dynamics are

$$z_{t+1} = Az_t + B_1 v_{1t} + B_2 v_{2t}$$

where A is a 3×3 identity matrix and

$$B_1 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \quad B_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

The Markov perfect decision rules are

$$v_{1t} = -F_1 z_t, \quad v_{2t} = -F_2 z_t$$

and in the Markov perfect equilibrium, the state evolves according to

$$z_{t+1} = (A - B_1 F_1 - B_2 F_2) z_t$$

```
# In LQ form
A = np.eye(3)
B1 = np.array([[0], [0], [1]])
B2 = np.array([[0], [1], [0]])

R1 = np.array([[0,             0, -a0 / 2],
               [0,             0,  a1 / 2],
               [-a0 / 2, a1 / 2,       a1]])

R2 = np.array([[0,       -a0 / 2,       0],
               [-a0 / 2,       a1, a1 / 2],
               [0,         a1 / 2,       0]])

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, tol=tol1)

# Simulate forward
AF = A - B1 @ F1 - B2 @ F2
z = np.empty((3, n))
z[:, 0] = 1, 1, 1
for t in range(n-1):
    z[:, t+1] = AF @ z[:, t]

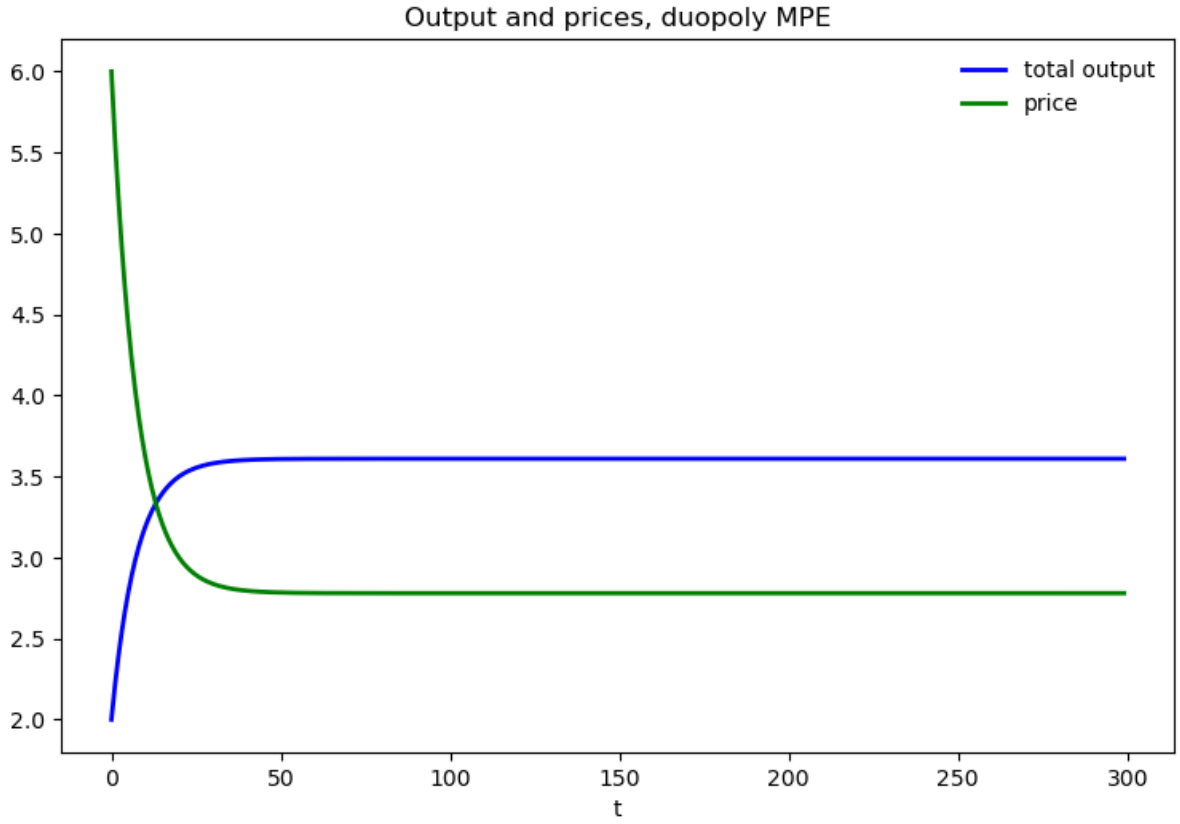
# Display policies
print("Computed policies for firm 1 and firm 2:\n")
print(f"F1 = {F1}")
print(f"F2 = {F2}")
```

```
Computed policies for firm 1 and firm 2:

F1 = [[-0.22701363  0.03129874  0.09447113]]
F2 = [[-0.22701363  0.09447113  0.03129874]]
```

```
q1 = z[1, :]
q2 = z[2, :]
q = q1 + q2       # Total output, MPE
p = a0 - a1 * q   # Price, MPE

fig, ax = plt.subplots(figsize=(9, 5.8))
ax.plot(range(n), q, 'b-', lw=2, label='total output')
ax.plot(range(n), p, 'g-', lw=2, label='price')
ax.set_title('Output and prices, duopoly MPE')
ax.legend(frameon=False)
ax.set_xlabel('t')
plt.show()
```




```
# Computes the maximum difference between the two quantities of the two firms
np.max(np.abs(q1 - q2))
```

```
8.881784197001252e-16
```

```
# Compute values
u1 = (- F1 @ z).flatten()
u2 = (- F2 @ z).flatten()

π_1 = p * q1 - γ * (u1) ** 2
π_2 = p * q2 - γ * (u2) ** 2

v1_forward = np.sum(βs * π_1)
v2_forward = np.sum(βs * π_2)

v1_direct = (- z[:, 0].T @ P1 @ z[:, 0])
v2_direct = (- z[:, 0].T @ P2 @ z[:, 0])

# Display values
print("Computed values for firm 1 and firm 2:\n")
print(f"v1(forward sim) = {v1_forward:.4f}; v1 (direct) = {v1_direct:.4f}")
print(f"v2 (forward sim) = {v2_forward:.4f}; v2 (direct) = {v2_direct:.4f}")
```

```
Computed values for firm 1 and firm 2:

v1(forward sim) = 133.3303; v1 (direct) = 133.3296
v2 (forward sim) = 133.3303; v2 (direct) = 133.3296
```

```
# Sanity check
Λ1 = A - B2 @ F2
lq1 = qe.LQ(Q1, R1, Λ1, B1, beta=β)
P1_ih, F1_ih, d = lq1.stationary_values()

v2_direct_alt = - z[:, 0].T @ lq1.P @ z[:, 0] + lq1.d

(np.abs(v2_direct - v2_direct_alt) < tol2).all()
```

```
True
```

41.13 Comparing Markov Perfect Equilibrium and Stackelberg Outcome

It is enlightening to compare equilbrium values for firms 1 and 2 under two alternative settings:

- A Markov perfect equilibrium like that described in this lecture
- A Stackelberg equilbrium

The following code performs the required computations, then plots the continuation values.

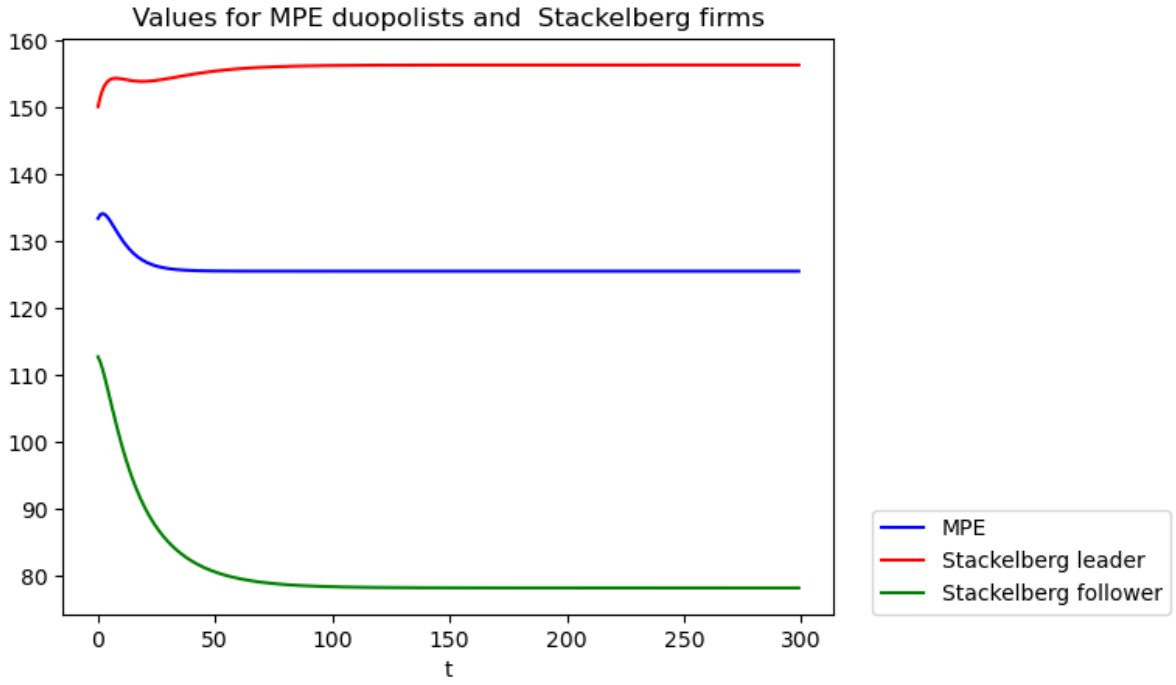
```
vt_MPE = np.zeros(n)
vt_follower = np.zeros(n)

for t in range(n):
    vt_MPE[t] = -z[:, t].T @ P1 @ z[:, t]
    vt_follower[t] = -yt_tilde[:, t].T @ P_tilde @ yt_tilde[:, t]
```

(continues on next page)

(continued from previous page)

```
fig, ax = plt.subplots()
ax.plot(vt_MPE, 'b', label='MPE')
ax.plot(vt_leader, 'r', label='Stackelberg leader')
ax.plot(vt_follower, 'g', label='Stackelberg follower')
ax.set_title(r'Values for MPE duopolists and  Stackelberg firms')
ax.set_xlabel('t')
ax.legend(loc=(1.05, 0))
plt.show()
```




```
# Display values
print("Computed values:\n")
print(f"vt_leader(y0) = {vt_leader[0]:.4f}")
print(f"vt_follower(y0) = {vt_follower[0]:.4f}")
print(f"vt_MPE(y0) = {vt_MPE[0]:.4f}")
```

```
Computed values:

vt_leader(y0) = 150.0324
vt_follower(y0) = 112.6559
vt_MPE(y0) = 133.3296
```

```
# Compute the difference in total value between the Stackelberg and the MPE
vt_leader[0] + vt_follower[0] - 2 * vt_MPE[0]
```

```
-3.9709425620890784
```

CHAPTER

FORTYTWO

MACHINE LEARNING A RAMSEY PLAN

42.1 Introduction

This lecture uses what we call a `machine learning` approach to compute a Ramsey plan for a version of a model of Calvo [Calvo, 1978].

We use another approach to compute a Ramsey plan for Calvo's model in another quantecon lecture *Time Inconsistency of Ramsey Plans*.

The *Time Inconsistency of Ramsey Plans* lecture uses an analytic approach based on `dynamic programming squared` to guide computations.

Dynamic programming squared provides information about the structure of mathematical objects in terms of which a Ramsey plan can be represented recursively.

Using that information paves the way to computing a Ramsey plan efficiently.

Included in the structural information that dynamic programming squared provides in quantecon lecture *Time Inconsistency of Ramsey Plans* are

- a **state** variable that confronts a continuation Ramsey planner, and
- two **Bellman equations**
  - one that describes the behavior of the representative agent
  - another that describes decision problems of a Ramsey planner and of a continuation Ramsey planner

In this lecture, we approach the Ramsey planner in a less sophisticated way that proceeds without knowing the mathematical structure imparted by dynamic programming squared.

We simply choose a pair of infinite sequences of real numbers that maximizes a Ramsey planner's objective function.

The pair consists of

- a sequence $\vec{\theta}$ of inflation rates
- a sequence $\vec{\mu}$ of money growh rates

Because it fails to take advantage of the structure recognized by dynamic programming squared and, relative to the dynamic programming squared approach, proliferates parameters, we take the liberty of calling this a **machine learning** approach.

This is similar to what other machine learning algorithms also do.

Comparing the calculations in this lecture with those in our sister lecture *Time Inconsistency of Ramsey Plans* provides us with a laboratory that can help us appreciate promises and limits of machine learning approaches more generally.

In this lecture, we'll actually deploy two machine learning approaches.

- the first is really lazy
  - it writes a Python function that computes the Ramsey planner's objective as a function of a money growth rate sequence and hands it over to a `gradient descent` optimizer
- the second is less lazy
  - it exerts enough mental effort required to express the Ramsey planner's objective as an affine quadratic form in $\vec{\mu}$, computes first-order conditions for an optimum, arranges them into a system of simultaneous linear equations for $\vec{\mu}$ and then $\vec{\theta}$, then solves them.

Each of these machine learning (ML) approaches recovers the same Ramsey plan that we compute in quantecon lecture *Time Inconsistency of Ramsey Plans* by using dynamic programming squared.

However, the recursive structure of the Ramsey plan lies hidden within some of the objects calculated by our ML approaches.

To ferret out that structure, we have to ask the right questions.

We pose some of those questions at the end of this lecture and answer them by running some linear regressions on components of $\vec{\mu}, \vec{\theta}$, and another vector that we'll define later.

Human intelligence, not the `artificial intelligence` deployed in our machine learning approach, is a key input into choosing which regressions to run.

42.2 The Model

We study a linear-quadratic version of a model that Guillermo Calvo [Calvo, 1978] used to illustrate the **time inconsistency** of optimal government plans.

Calvo's model focuses on intertemporal tradeoffs between

- utility accruing from a representative agent's anticipations of future deflation that lower the agent's cost of holding real money balances and prompt him to increase his *liquidity*, as measured by his stock of real money balances, and
- social costs associated with the distorting taxes that a government levies to acquire the paper money that it destroys in order to generate prospective deflation

The model features

- rational expectations
- costly government actions at all dates $t \geq 1$ that increase the representative agent's utilities at dates before t

The model combines ideas from papers by Cagan [Cagan, 1956], [Sargent and Wallace, 1973], and Calvo [Calvo, 1978].

42.3 Model Components

There is no uncertainty.

Let:

- p_t be the log of the price level
- m_t be the log of nominal money balances
- $\theta_t = p_{t+1} - p_t$ be the net rate of inflation between t and $t+1$
- $\mu_t = m_{t+1} - m_t$ be the net rate of growth of nominal balances

The demand for real balances is governed by a perfect foresight version of a Cagan [Cagan, 1956] demand function for real balances:

$$m_t - p_t = -\alpha(p_{t+1} - p_t)\ ,\ \alpha > 0 \tag{42.1}$$

for $t \geq 0$.

Equation (42.1) asserts that the representative agent's demand for real balances is inversely related to the representative agent's expected rate of inflation, which equals the actual rate of inflation because there is no uncertainty here.

(When there is no uncertainty, an assumption of **rational expectations** becomes equivalent to **perfect foresight**).

Subtracting the demand function (42.1) at time t from the demand function at $t+1$ gives:

$$\mu_t - \theta_t = -\alpha\theta_{t+1} + \alpha\theta_t$$

or

$$\theta_t = \frac{\alpha}{1+\alpha}\theta_{t+1} + \frac{1}{1+\alpha}\mu_t \tag{42.2}$$

Because $\alpha > 0$, $0 < \frac{\alpha}{1+\alpha} < 1$.

Definition 42.3.1

For scalar b_t, let L^2 be the space of sequences $\{b_t\}_{t=0}^{\infty}$ that satisfy

$$\sum_{t=0}^{\infty} b_t^2 < +\infty$$

We say that a sequence that belongs to L^2 is **square summable**.

When we assume that the sequence $\vec{\mu} = \{\mu_t\}_{t=0}^{\infty}$ is square summable and also require that the sequence $\vec{\theta} = \{\theta_t\}_{t=0}^{\infty}$ is square summable, the linear difference equation (42.2) can be solved forward to get:

$$\theta_t = \frac{1}{1+\alpha}\sum_{j=0}^{\infty}\left(\frac{\alpha}{1+\alpha}\right)^j \mu_{t+j}, \quad t \geq 0 \tag{42.3}$$

The government values a representative household's utility of real balances at time t according to the utility function

$$U(m_t - p_t) = u_0 + u_1(m_t - p_t) - \frac{u_2}{2}(m_t - p_t)^2, \quad u_0 > 0, u_1 > 0, u_2 > 0 \tag{42.4}$$

The money demand function (42.1) and the utility function (42.4) imply that

$$U(-\alpha\theta_t) = u_0 + u_1(-\alpha\theta_t) - \frac{u_2}{2}(-\alpha\theta_t)^2. \tag{42.5}$$

Note: The "bliss level" of real balances is $\frac{u_1}{u_2}$; the inflation rate that attains it is $-\frac{u_1}{u_2\alpha}$.

Via equation (42.3), a government plan $\vec{\mu} = \{\mu_t\}_{t=0}^{\infty}$ leads to a sequence of inflation rates $\vec{\theta} = \{\theta_t\}_{t=0}^{\infty}$.

We assume that the government incurs social costs $\frac{c}{2}\mu_t^2$ when it changes the stock of nominal money balances at rate μ_t at time t.

Therefore, the one-period welfare function of a benevolent government is

$$s(\theta_t, \mu_t) = U(-\alpha\theta_t) - \frac{c}{2}\mu_t^2.$$

The Ramsey planner's criterion is

$$V = \sum_{t=0}^{\infty} \beta^t s(\theta_t, \mu_t) \tag{42.6}$$

where $\beta \in (0, 1)$ is a discount factor.

The Ramsey planner chooses a vector of money growth rates $\vec{\mu}$ to maximize criterion (42.6) subject to equations (42.3) and that restriction

$$\vec{\theta} \in L^2 \tag{42.7}$$

Equations (42.3) and (42.7) imply that $\vec{\theta}$ is a function of $\vec{\mu}$.

In particular, the inflation rate θ_t satisfies

$$\theta_t = (1 - \lambda) \sum_{j=0}^{\infty} \lambda^j \mu_{t+j}, \quad t \geq 0 \tag{42.8}$$

where

$$\lambda = \frac{\alpha}{1 + \alpha}.$$

42.4 Parameters and Variables

Parameters:

- Demand for money parameter is $\alpha > 0$; we set its default value $\alpha = 1$
  - Induced demand function for money parameter is $\lambda = \frac{\alpha}{1+\alpha}$
- Utility function parameters are u_0, u_1, u_2 and $\beta \in (0, 1)$
- Cost parameter of tax distortions associated with setting $\mu_t \neq 0$ is c
- A horizon truncation parameter: a positive integer $T > 0$

Variables:

- $\theta_t = p_{t+1} - p_t$ where p_t is log of price level
- $\mu_t = m_{t+1} - m_t$ where m_t is log of money supply

42.4.1 Basic Objects

To prepare the way for our calculations, we'll remind ourselves of the mathematical objects in play.

- sequences of inflation rates and money creation rates:

$$(\vec{\theta}, \vec{\mu}) = \{\theta_t, \mu_t\}_{t=0}^{\infty}$$

- A planner's value function

$$V = \sum_{t=0}^{\infty} \beta^t (h_0 + h_1 \theta_t + h_2 \theta_t^2 - \frac{c}{2} \mu_t^2) \tag{42.9}$$

where we set h_0, h_1, h_2 to match

$$u_0 + u_1(-\alpha\theta_t) - \frac{u_2}{2}(-\alpha\theta_t)^2$$

with

$$h_0 + h_1\theta_t + h_2\theta_t^2$$

To make our parameters match as we want, we set

$$\begin{aligned} h_0 &= u_0 \\ h_1 &= -\alpha u_1 \\ h_2 &= -\frac{u_2\alpha^2}{2} \end{aligned}$$

A Ramsey planner chooses $\vec{\mu}$ to maximize the government's value function (42.9) subject to equations (42.8).

A solution $\vec{\mu}$ of this problem is called a **Ramsey plan**.

42.4.2 Timing protocol

Following Calvo [Calvo, 1978], we assume that the government chooses the money growth sequence $\vec{\mu}$ once and for all at, or before, time 0.

An optimal government plan under this timing protocol is an example of what is often called a **Ramsey plan**.

Notice that while the government is in effect choosing a bivariate **time series** $(\vec{m}u, \vec{\theta})$, the government's problem is **static** in the sense that it chooses treats that time-series as a single object to be chosen at a single point in time.

42.5 Approximation and Truncation parameter T

We anticipate that under a Ramsey plan the sequences $\{\theta_t\}$ and $\{\mu_t\}$ both converge to stationary values.

Thus, we guess that under the optimal policy $\lim_{t\to+\infty} \mu_t = \bar{\mu}$.

Convergence of μ_t to $\bar{\mu}$ together with formula (42.8) for the inflation rate then implies that $\lim_{t\to+\infty} \theta_t = \bar{\mu}$ as well.

We'll guess a time T large enough that μ_t has gotten very close to the limit $\bar{\mu}$.

Then we'll approximate $\vec{\mu}$ by a truncated vector with the property that

$$\mu_t = \bar{\mu} \quad \forall t \geq T$$

We'll approximate $\vec{\theta}$ with a truncated vector with the property that

$$\theta_t = \bar{\theta} \quad \forall t \geq T$$

Formula for truncated $\vec{\theta}$

In light of our approximation that $\mu_t = \bar{\mu}$ for all $t \geq T$, we seek a function that takes

$$\tilde{\mu} = \begin{bmatrix} \mu_0 & \mu_1 & \cdots & \mu_{T-1} & \bar{\mu} \end{bmatrix}$$

as an input and as an output gives

$$\tilde{\theta} = \begin{bmatrix} \theta_0 & \theta_1 & \cdots & \theta_{T-1} & \bar{\theta} \end{bmatrix}$$

where $\bar{\theta} = \bar{\mu}$ and θ_t satisfies

$$\theta_t = (1-\lambda)\sum_{j=0}^{T-1-t} \lambda^j \mu_{t+j} + \lambda^{T-t}\bar{\mu} \tag{42.10}$$

for $t = 0, 1, \ldots, T-1$.

Formula for V

Having specified a truncated vector $\tilde{\mu}$ and and having computed $\tilde{\theta}$ by using formula (42.10), we shall write a Python function that computes

$$\tilde{V} = \sum_{t=0}^{\infty} \beta^t (h_0 + h_1 \tilde{\theta}_t + h_2 \tilde{\theta}_t^2 - \frac{c}{2} \mu_t^2) \tag{42.11}$$

or more precisely

$$\tilde{V} = \sum_{t=0}^{T-1} \beta^t (h_0 + h_1 \tilde{\theta}_t + h_2 \tilde{\theta}_t^2 - \frac{c}{2} \mu_t^2) + \frac{\beta^T}{1-\beta} (h_0 + h_1 \bar{\mu} + h_2 \bar{\mu}^2 - \frac{c}{2} \bar{\mu}^2)$$

where $\tilde{\theta}_t, \ t = 0, 1, \ldots, T-1$ satisfies formula (1).

42.6 A Gradient Descent Algorithm

We now describe code that maximizes the criterion function (42.9) subject to equations (42.8) by choice of the truncated vector $\tilde{\mu}$.

We use a brute force or `machine learning` approach that just hands our problem off to code that minimizes V with respect to the components of $\tilde{\mu}$ by using gradient descent.

We hope that answers will agree with those found obtained by other more structured methods in this quantecon lecture *Time Inconsistency of Ramsey Plans*.

42.6.1 Implementation

We will implement the above in Python using JAX and Optax libraries.

We use the following imports in this lecture

```
!pip install --upgrade quantecon
!pip install --upgrade optax
!pip install --upgrade statsmodels
```

```
from quantecon import LQ
import numpy as np
import jax.numpy as jnp
from jax import jit, grad
import optax
import statsmodels.api as sm
import matplotlib.pyplot as plt
```

We'll eventually want to compare the results we obtain here to those that we obtain in those obtained in this quantecon lecture *Time Inconsistency of Ramsey Plans*.

To enable us to do that, we copy the class `ChangLQ` used in that lecture.

We hide the cell that copies the class, but readers can find details of the class in this quantecon lecture *Time Inconsistency of Ramsey Plans*.

Now we compute the value of V under this setup, and compare it against those obtained in this section *Outcomes under Three Timing Protocols* of the sister quantecon lecture *Time Inconsistency of Ramsey Plans*.

```
# Assume β=0.85, c=2, T=40.
T = 40
clq = ChangLQ(β=0.85, c=2, T=T)
```

```
@jit
def compute_θ(μ, α=1):
    λ = α / (1 + α)
    T = len(μ) - 1
    μbar = μ[-1]

    # Create an array of powers for λ
    λ_powers = λ ** jnp.arange(T + 1)

    # Compute the weighted sums for all t
    weighted_sums = jnp.array(
        [jnp.sum(λ_powers[:T-t] * μ[t:T]) for t in range(T)])

    # Compute θ values except for the last element
    θ = (1 - λ) * weighted_sums + λ**(T - jnp.arange(T)) * μbar

    # Set the last element
    θ = jnp.append(θ, μbar)

    return θ

@jit
def compute_hs(u0, u1, u2, α):
    h0 = u0
    h1 = -u1 * α
    h2 = -0.5 * u2 * α**2

    return h0, h1, h2

@jit
def compute_V(μ, β, c, α=1, u0=1, u1=0.5, u2=3):
    θ = compute_θ(μ, α)

    h0, h1, h2 = compute_hs(u0, u1, u2, α)

    T = len(μ) - 1
    t = np.arange(T)

    # Compute sum except for the last element
    V_sum = np.sum(β**t * (h0 + h1 * θ[:T] + h2 * θ[:T]**2 - 0.5 * c * μ[:T]**2))

    # Compute the final term
    V_final = (β**T / (1 - β)) * (h0 + h1 * μ[-1] + h2 * μ[-1]**2 - 0.5 * c * μ[-
↪1]**2)

    V = V_sum + V_final

    return V
```

```
V_val = compute_V(clq.μ_series, β=0.85, c=2)
```

(continues on next page)

(continued from previous page)

```
# Check the result with the ChangLQ class in previous lecture
print(f'deviation = {np.abs(V_val - clq.J_series[0])}') # good!
```

```
deviation = 1.430511474609375e-06
```

Now we want to maximize the function V by choice of μ.

We will use the `optax.adam` from the `optax` library.

```
def adam_optimizer(grad_func, init_params,
                   lr=0.1,
                   max_iter=10_000,
                   error_tol=1e-7):

    # Set initial parameters and optimizer
    params = init_params
    optimizer = optax.adam(learning_rate=lr)
    opt_state = optimizer.init(params)

    # Update parameters and gradients
    @jit
    def update(params, opt_state):
        grads = grad_func(params)
        updates, opt_state = optimizer.update(grads, opt_state)
        params = optax.apply_updates(params, updates)
        return params, opt_state, grads

    # Gradient descent loop
    for i in range(max_iter):
        params, opt_state, grads = update(params, opt_state)

        if jnp.linalg.norm(grads) < error_tol:
            print(f"Converged after {i} iterations.")
            break

        if i % 100 == 0:
            print(f"Iteration {i}, grad norm: {jnp.linalg.norm(grads)}")

    return params
```

Here we use automatic differentiation functionality in JAX with `grad`.

```
# Initial guess for μ
μ_init = jnp.zeros(T)

# Maximization instead of minimization
grad_V = jit(grad(
    lambda μ: -compute_V(μ, β=0.85, c=2)))
```

```
%%time

# Optimize μ
optimized_μ = adam_optimizer(grad_V, μ_init)

print(f"optimized μ = \n{optimized_μ}")
```

```
Iteration 0, grad norm: 0.8627105951309204
Iteration 100, grad norm: 0.0033030612394213676
Iteration 200, grad norm: 1.693110061751213e-05
Converged after 283 iterations.
optimized μ =
[-0.06450704 -0.09033978 -0.10068485 -0.10482768 -0.10648673 -0.10715112
 -0.1074172  -0.10752377 -0.10756643 -0.1075835  -0.10759035 -0.10759307
 -0.10759415 -0.10759456 -0.10759478 -0.10759483 -0.10759487 -0.10759489
 -0.10759491 -0.10759492 -0.1075949  -0.10759491 -0.10759491 -0.1075949
 -0.10759491 -0.10759491 -0.10759488 -0.10759489 -0.10759493 -0.10759494
 -0.10759492 -0.10759494 -0.10759488 -0.10759489 -0.10759488 -0.10759489
 -0.10759487 -0.1075949  -0.10759485 -0.10759484]
CPU times: user 1.11 s, sys: 59.8 ms, total: 1.17 s
Wall time: 670 ms
```

```
print(f"original μ = \n{clq.μ_series}")
```

```
original μ =
[-0.06450708 -0.09033982 -0.10068489 -0.10482772 -0.10648677 -0.10715115
 -0.10741722 -0.10752377 -0.10756644 -0.10758352 -0.10759037 -0.10759311
 -0.1075942  -0.10759464 -0.10759482 -0.10759489 -0.10759492 -0.10759493
 -0.10759493 -0.10759494 -0.10759494 -0.10759494 -0.10759494 -0.10759494
 -0.10759494 -0.10759494 -0.10759494 -0.10759494 -0.10759494 -0.10759494
 -0.10759494 -0.10759494 -0.10759494 -0.10759494 -0.10759494 -0.10759494
 -0.10759494 -0.10759494 -0.10759494 -0.10759494]
```

```
print(f'deviation = {np.linalg.norm(optimized_μ - clq.μ_series)}')
```

```
deviation = 2.7273603109279065e-07
```

```
compute_V(optimized_μ, β=0.85, c=2)
```

```
Array(6.8357825, dtype=float32)
```

```
compute_V(clq.μ_series, β=0.85, c=2)
```

```
Array(6.835783, dtype=float32)
```

42.6.2 Restricting $\mu_t = \bar{\mu}$ for all t

We take a brief detour to solve a restricted version of the Ramsey problem defined above.

First, recall that a Ramsey planner chooses $\vec{\mu}$ to maximize the government's value function (42.9) subject to equations (42.8).

We now define a distinct problem in which the planner chooses $\vec{\mu}$ to maximize the government's value function (42.9) subject to equation (42.8) and the additional restriction that $\mu_t = \bar{\mu}$ for all t.

The solution of this problem is a time-invariant μ_t that this quantecon lecture *Time Inconsistency of Ramsey Plans* calls μ^{CR}.

```
# Initial guess for single μ
μ_init = jnp.zeros(1)

# Maximization instead of minimization
grad_V = jit(grad(
    lambda μ: -compute_V(μ, β=0.85, c=2)))

# Optimize μ
optimized_μ_CR = adam_optimizer(grad_V, μ_init)

print(f"optimized μ = \n{optimized_μ_CR}")
```

```
Iteration 0, grad norm: 3.333333969116211
Iteration 100, grad norm: 0.0049784183502197266
Iteration 200, grad norm: 6.771087646484375e-05
Converged after 282 iterations.
```

```
optimized μ =
[-0.10000004]
```

Comparing it to μ^{CR} in *Time Inconsistency of Ramsey Plans*, we again obtained very close answers.

```
np.linalg.norm(clq.μ_CR - optimized_μ_CR)
```

```
3.7252903e-08
```

```
V_CR = compute_V(optimized_μ_CR, β=0.85, c=2)
V_CR
```

```
Array(6.8333354, dtype=float32)
```

```
compute_V(jnp.array([clq.μ_CR]), β=0.85, c=2)
```

```
Array(6.8333344, dtype=float32)
```

42.7 A More Structured ML Algorithm

By thinking about the mathematical structure of the Ramsey problem and using some linear algebra, we can simplify the problem that we hand over to a `machine learning` algorithm.

We start by recalling that the Ramsey problem that chooses $\vec{\mu}$ to maximize the government's value function (42.9)subject to equation (42.8).

This turns out to be an optimization problem with a quadratic objective function and linear constraints.

First-order conditions for this problem are a set of simultaneous linear equations in $\vec{\mu}$.

If we trust that the second-order conditions for a maximum are also satisfied (they are in our problem), we can compute the Ramsey plan by solving these equations for $\vec{\mu}$.

We'll apply this approach here and compare answers with what we obtained above with the gradient descent approach.

To remind us of the setting, remember that we have assumed that

$$\mu_t = \mu_T \ \forall t \geq T$$

and that

$$\theta_t = \theta_T = \mu_T \ \forall t \geq T$$

Again, define

$$\vec{\theta} = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_{T-1} \\ \theta_T \end{bmatrix}, \quad \vec{\mu} = \begin{bmatrix} \mu_0 \\ \mu_1 \\ \vdots \\ \mu_{T-1} \\ \mu_T \end{bmatrix}$$

Write the system of $T+1$ equations (42.10) that relate $\vec{\theta}$ to a choice of $\vec{\mu}$ as the single matrix equation

$$\frac{1}{(1-\lambda)} \begin{bmatrix} 1 & -\lambda & 0 & 0 & \cdots & 0 & 0 \\ 0 & 1 & -\lambda & 0 & \cdots & 0 & 0 \\ 0 & 0 & 1 & -\lambda & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & -\lambda & 0 \\ 0 & 0 & 0 & 0 & \cdots & 1 & -\lambda \\ 0 & 0 & 0 & 0 & \cdots & 0 & 1-\lambda \end{bmatrix} \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \vdots \\ \theta_{T-1} \\ \theta_T \end{bmatrix} = \begin{bmatrix} \mu_0 \\ \mu_1 \\ \mu_2 \\ \vdots \\ \mu_{T-1} \\ \mu_T \end{bmatrix}$$

or

$$A\vec{\theta} = \vec{\mu}$$

or

$$\vec{\theta} = B\vec{\mu}$$

where

$$B = A^{-1}$$

```
def construct_B(α, T):
    λ = α / (1 + α)

    A = (jnp.eye(T, T) - λ*jnp.eye(T, T, k=1))/(1-λ)
    A = A.at[-1, -1].set(A[-1, -1]*(1-λ))

    B = jnp.linalg.inv(A)
    return A, B
```

```
A, B = construct_B(α=clq.α, T=T)

print(f'A = \n {A}')
```

```
A =
 [[ 2. -1.  0. ...  0.  0.  0.]
 [ 0.  2. -1. ...  0.  0.  0.]
 [ 0.  0.  2. ...  0.  0.  0.]
 ...
 [ 0.  0.  0. ...  2. -1.  0.]
 [ 0.  0.  0. ...  0.  2. -1.]
 [ 0.  0.  0. ...  0.  0.  1.]]
```

```
# Compute θ using optimized_μ
θs = np.array(compute_θ(optimized_μ))
μs = np.array(optimized_μ)
```

```
np.allclose(θs, B @ clq.μ_series)
```

```
True
```

As before, the Ramsey planner's criterion is

$$V = \sum_{t=0}^{\infty} \beta^t (h_0 + h_1 \theta_t + h_2 \theta_t^2 - \frac{c}{2}\mu_t^2)$$

With our assumption above, criterion V can be rewritten as

$$\begin{aligned} V = &\sum_{t=0}^{T-1} \beta^t (h_0 + h_1 \theta_t + h_2 \theta_t^2 - \frac{c}{2}\mu_t^2) \\ &+ \frac{\beta^T}{1-\beta}(h_0 + h_1 \theta_T + h_2 \theta_T^2 - \frac{c}{2}\mu_T^2) \end{aligned}$$

To help us write V as a quadratic plus affine form, define

$$\vec{\beta} = \begin{bmatrix} 1 \\ \beta \\ \vdots \\ \beta^{T-1} \\ \frac{\beta^T}{1-\beta} \end{bmatrix}$$

Then we have:

$$h_1 \sum_{t=0}^{\infty} \beta^t \theta_t = h_1 \cdot \vec{\beta}^T \vec{\theta} = (h_1 \cdot B^T \vec{\beta})^T \vec{\mu} = g^T \vec{\mu}$$

where $g = h_1 \cdot B^T \vec{\beta}$ is a $(T+1) \times 1$ vector,

$$h_2 \sum_{t=0}^{\infty} \beta^t \theta_t^2 = \vec{\mu}^T (B^T (h_2 \cdot \vec{\beta} \cdot \mathbf{I}) B) \vec{\mu} = \vec{\mu}^T M \vec{\mu}$$

where $M = B^T (h_2 \cdot \vec{\beta} \cdot \mathbf{I}) B$ is a $(T+1) \times (T+1)$ matrix,

$$\frac{c}{2} \sum_{t=0}^{\infty} \beta^t \mu_t^2 = \vec{\mu}^T (\frac{c}{2} \cdot \vec{\beta} \cdot \mathbf{I}) \vec{\mu} = \vec{\mu}^T F \vec{\mu}$$

where $F = \frac{c}{2} \cdot \vec{\beta} \cdot \mathbf{I}$ is a $(T+1) \times (T+1)$ matrix

It follows that

$$\begin{aligned} J = V - h_0 &= \sum_{t=0}^{\infty} \beta^t (h_1 \theta_t + h_2 \theta_t^2 - \frac{c}{2}\mu_t^2) \\ &= g^T \vec{\mu} + \vec{\mu}^T M \vec{\mu} - \vec{\mu}^T F \vec{\mu} \\ &= g^T \vec{\mu} + \vec{\mu}^T (M - F) \vec{\mu} \\ &= g^T \vec{\mu} + \vec{\mu}^T G \vec{\mu} \end{aligned}$$

where $G = M - F$.

To compute the optimal government plan we want to maximize J with respect to $\vec{\mu}$.

We use linear algebra formulas for differentiating linear and quadratic forms to compute the gradient of J with respect to $\vec{\mu}$

$$\frac{\partial}{\partial \vec{\mu}} J = g + 2G\vec{\mu}.$$

Setting $\frac{\partial}{\partial \vec{\mu}} J = 0$, the maximizing μ is

$$\vec{\mu}^R = -\frac{1}{2} G^{-1} g$$

The associated optimal inflation sequence is

$$\vec{\theta}^R = B\vec{\mu}^R$$

42.7.1 Two implementations

With the more structured approach, we can update our gradient descent exercise with `compute_J`

```
def compute_J(μ, β, c, α=1, u0=1, u1=0.5, u2=3):
    T = len(μ) - 1

    h0, h1, h2 = compute_hs(u0, u1, u2, α)
    λ = α / (1 + α)

    _, B = construct_B(α, T+1)

    β_vec = jnp.hstack([β**jnp.arange(T),
                        (β**T/(1-β))])

    θ = B @ μ
    βθ_sum = jnp.sum((β_vec * h1) * θ)
    βθ_square_sum = β_vec * h2 * θ.T @ θ
    βμ_square_sum = 0.5 * c * β_vec * μ.T @ μ

    return βθ_sum + βθ_square_sum - βμ_square_sum
```

```
# Initial guess for μ
μ_init = jnp.zeros(T)

# Maximization instead of minimization
grad_J = jit(grad(
    lambda μ: -compute_J(μ, β=0.85, c=2)))
```

```
%%time

# Optimize μ
optimized_μ = adam_optimizer(grad_J, μ_init)

print(f"optimized μ = \n{optimized_μ}")
```

```
Iteration 0, grad norm: 0.8627105951309204
Iteration 100, grad norm: 0.003303033299744129
Iteration 200, grad norm: 1.6926183889154345e-05
Converged after 280 iterations.
optimized μ =
[-0.06450712 -0.09033988 -0.10068493 -0.10482774 -0.1064868  -0.1071512
 -0.10741723 -0.10752378 -0.10756644 -0.10758355 -0.10759039 -0.10759313
 -0.10759424 -0.10759471 -0.10759488 -0.10759497 -0.10759498 -0.10759498
 -0.10759498 -0.10759497 -0.10759494 -0.10759497 -0.10759496 -0.10759497
 -0.10759497 -0.10759498 -0.107595   -0.10759494 -0.10759496 -0.10759496
 -0.10759494 -0.10759493 -0.10759493 -0.10759494 -0.10759494 -0.10759495
 -0.10759497 -0.107595   -0.10759497 -0.10759495]
CPU times: user 437 ms, sys: 239 ms, total: 676 ms
Wall time: 247 ms
```

```
print(f"original μ = \n{clq.μ_series}")
```

```
original μ =
[-0.06450708 -0.09033982 -0.10068489 -0.10482772 -0.10648677 -0.10715115
 -0.10741722 -0.10752377 -0.10756644 -0.10758352 -0.10759037 -0.10759311
 -0.1075942  -0.10759464 -0.10759482 -0.10759489 -0.10759492 -0.10759493
 -0.10759493 -0.10759494 -0.10759494 -0.10759494 -0.10759494 -0.10759494
 -0.10759494 -0.10759494 -0.10759494 -0.10759494 -0.10759494 -0.10759494
 -0.10759494 -0.10759494 -0.10759494 -0.10759494 -0.10759494 -0.10759494
 -0.10759494 -0.10759494 -0.10759494 -0.10759494]
```

```
print(f'deviation = {np.linalg.norm(optimized_μ - clq.μ_series)}')
```

```
deviation = 2.3748542332668876e-07
```

```
V_R = compute_V(optimized_μ, β=0.85, c=2)
V_R
```

```
Array(6.8357825, dtype=float32)
```

We find that by exploiting more knowledge about the structure of the problem, we can significantly speed up our computation.

We can also derive a closed-form solution for $\vec{\mu}$

```
def compute_μ(β, c, T, α=1, u0=1, u1=0.5, u2=3):
    h0, h1, h2 = compute_hs(u0, u1, u2, α)

    _, B = construct_B(α, T+1)

    β_vec = jnp.hstack([β**jnp.arange(T),
                        (β**T/(1-β))])

    g = h1 * B.T @ β_vec
    M = B.T @ (h2 * jnp.diag(β_vec)) @ B
    F = c/2 * jnp.diag(β_vec)
    G = M - F
```

(continues on next page)

(continued from previous page)

```
    return jnp.linalg.solve(2*G, -g)

μ_closed = compute_μ(β=0.85, c=2, T=T-1)
print(f'closed-form μ = \n{μ_closed}')
```

```
closed-form μ =
[-0.0645071  -0.09033982 -0.1006849  -0.1048277  -0.10648677 -0.10715113
 -0.10741723 -0.10752378 -0.10756643 -0.10758351 -0.10759034 -0.10759313
 -0.10759421 -0.10759464 -0.10759482 -0.1075949  -0.10759489 -0.10759492
 -0.10759492 -0.10759491 -0.10759495 -0.10759494 -0.10759495 -0.10759493
 -0.10759491 -0.10759491 -0.10759494 -0.10759491 -0.10759491 -0.10759495
 -0.10759498 -0.10759492 -0.10759494 -0.10759485 -0.10759497 -0.10759495
 -0.10759493 -0.10759494 -0.10759498 -0.10759494]
```

```
print(f'deviation = {np.linalg.norm(μ_closed - clq.μ_series)}')
```

```
deviation = 1.47137171779832e-07
```

```
compute_V(μ_closed, β=0.85, c=2)
```

```
Array(6.835783, dtype=float32)
```

```
print(f'deviation = {np.linalg.norm(B @ μ_closed - θs)}')
```

```
deviation = 2.9181177296777605e-07
```

We can check the gradient of the analytical solution against the `JAX` computed version

```
def compute_grad(μ, β, c, α=1, u0=1, u1=0.5, u2=3):
    T = len(μ) - 1

    h0, h1, h2 = compute_hs(u0, u1, u2, α)

    _, B = construct_B(α, T+1)

    β_vec = jnp.hstack([β**jnp.arange(T),
                        (β**T/(1-β))])

    g = h1 * B.T @ β_vec
    M = (h2 * B.T @ jnp.diag(β_vec) @ B)
    F = c/2 * jnp.diag(β_vec)
    G = M - F
    return g + (2*G @ μ)

closed_grad = compute_grad(jnp.ones(T), β=0.85, c=2)
```

```
closed_grad
```

```
Array([-3.75      , -4.0625    , -3.8906252 , -3.5257816 , -3.1062894 ,
       -2.6950336 , -2.3181221 , -1.9840758 , -1.6933005 , -1.4427234 ,
       -1.2280239 , -1.0446749 , -0.8884009 , -0.7553544 , -0.64215815,
       -0.5458878 , -0.46403134, -0.39444   , -0.33528066, -0.28499195,
       -0.24224481, -0.20590894, -0.17502302, -0.14876978, -0.12645441,
       -0.10748631, -0.09136339, -0.0776589 , -0.06601007, -0.05610856,
       -0.04769228, -0.04053844, -0.03445768, -0.02928903, -0.02489567,
       -0.02116132, -0.01798713, -0.01528906, -0.0129957 , -0.07364222],      ␣
 ↪dtype=float32)
```

```
- grad_J(jnp.ones(T))
```

```
Array([-3.75      , -4.0625    , -3.890625  , -3.5257816 , -3.1062894 ,
       -2.6950336 , -2.3181224 , -1.9840759 , -1.6933005 , -1.4427235 ,
       -1.228024  , -1.0446749 , -0.8884009 , -0.7553544 , -0.6421581 ,
       -0.54588777, -0.46403137, -0.39444   , -0.33528066, -0.28499192,
       -0.24224481, -0.20590894, -0.175023  , -0.14876977, -0.12645441,
       -0.10748631, -0.0913634 , -0.0776589 , -0.06601007, -0.05610857,
       -0.04769228, -0.04053844, -0.03445768, -0.02928903, -0.02489568,
       -0.02116132, -0.01798712, -0.01528906, -0.0129957 , -0.07364222],      ␣
 ↪dtype=float32)
```

```
print(f'deviation = {np.linalg.norm(closed_grad - (- grad_J(jnp.ones(T))))}')
```

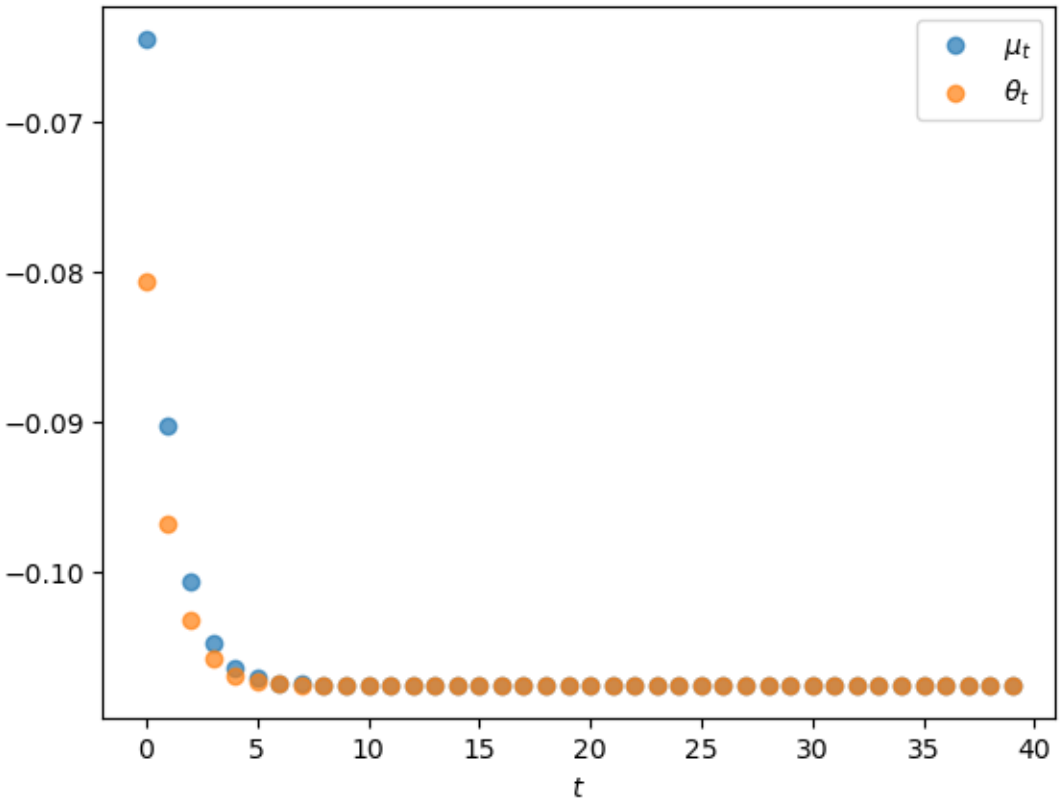
```
deviation = 4.074267394571507e-07
```

Let's plot the Ramsey plan's μ_t and θ_t for $t = 0, \ldots, T$ against t.

```
# Compute θ using optimized_μ
θs = np.array(compute_θ(optimized_μ))
μs = np.array(optimized_μ)

# Plot the two sequences
Ts = np.arange(T)

plt.scatter(Ts, μs, label=r'$\mu_t$', alpha=0.7)
plt.scatter(Ts, θs, label=r'$\theta_t$', alpha=0.7)
plt.xlabel(r'$t$')
plt.legend()
plt.show()
```



Note that while θ_t is less than $\mu_t$for low t's, it eventually converges to the limit $\bar{\mu}$ of μ_t as $t \to +\infty$.

This pattern reflects how formula (42.3) makes θ_t be a weighted average of future μ_t's.

42.8 Continuation Values

For subsquent analysis, it will be useful to compute a sequence $\{v_t\}_{t=0}^T$ of what we'll call `continuation values` along a Ramsey plan.

To do so, we'll start at date T and compute

$$v_T = \frac{1}{1-\beta} s(\bar{\mu}, \bar{\mu}).$$

Then starting from $t = T-1$, we'll iterate backwards on the recursion

$$v_t = s(\theta_t, \mu_t) + \beta v_{t+1}$$

for $t = T-1, T-2, \ldots, 0$.

```
# Define function for s and U in section 41.3
def s(θ, μ, u0, u1, u2, α, c):
    U = lambda x: u0 + u1 * x - (u2 / 2) * x**2
    return U(-α*θ) - (c / 2) * μ**2

# Calculate v_t sequence backward
def compute_vt(μ, β, c, u0=1, u1=0.5, u2=3, α=1):
```

(continues on next page)

(continued from previous page)

```
    T = len(μ)
    θ = compute_θ(μ, α)

    v_t = np.zeros(T)
    μ_bar = μ[-1]

    # Reduce parameters
    s_p = lambda θ, μ: s(θ, μ,
                         u0=u0, u1=u1, u2=u2, α=α, c=c)

    # Define v_T
    v_t[T-1] = (1 / (1 - β)) * s_p(μ_bar, μ_bar)

    # Backward iteration
    for t in reversed(range(T-1)):
        v_t[t] = s_p(θ[t], μ[t]) + β * v_t[t+1]

    return v_t

v_t = compute_vt(μs, β=0.85, c=2)
```

The initial continuation value v_0 should equal the optimized value of the Ramsey planner's criterion V defined in equation (42.6).

Indeed, we find that the deviation is very small:

```
print(f'deviation = {np.linalg.norm(v_t[0] - V_R)}')
```

```
deviation = 4.76837158203125e-07
```

We can also verify approximate equality by inspecting a graph of v_t against t for $t = 0, \ldots, T$ along with the value attained by a restricted Ramsey planner V^{CR} and the optimized value of the ordinary Ramsey planner V^R

```
# Plot the scatter plot
plt.scatter(Ts, v_t, label='$v_t$')

# Plot horizontal lines
plt.axhline(V_CR, color='C1', alpha=0.5)
plt.axhline(V_R, color='C2', alpha=0.5)

# Add labels
plt.text(max(Ts) + max(Ts)*0.07, V_CR, '$V^{CR}$', color='C1',
         va='center', clip_on=False, fontsize=15)
plt.text(max(Ts) + max(Ts)*0.07, V_R, '$V^R$', color='C2',
         va='center', clip_on=False, fontsize=15)
plt.xlabel(r'$t$')
plt.ylabel(r'$v_t$')

plt.tight_layout()
plt.show()
```

Figure Fig. 42.1 shows interesting patterns:

- The sequence of continuation values $\{v_t\}_{t=0}^T$ is monotonically decreasing

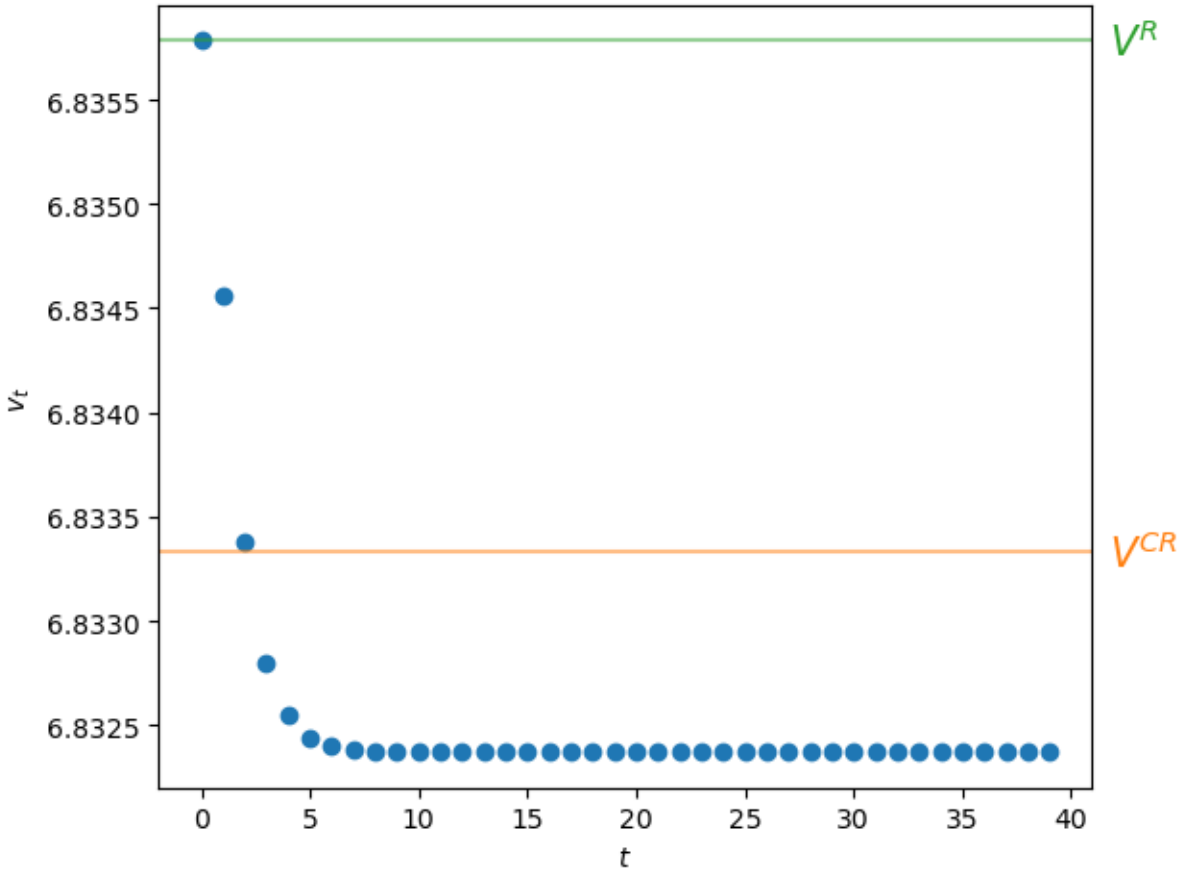


Fig. 42.1: Continuation values

- Evidently, $v_0 > V^{CR} > v_T$ so that
  - the value v_0 of the ordinary Ramsey plan exceeds the value V^{CR} of the special Ramsey plan in which the planner is constrained to set $\mu_t = \mu^{CR}$ for all t.
  - the continuation value v_T of the ordinary Ramsey plan for $t \geq T$ is constant and is less than the value V^{CR} of the special Ramsey plan in which the planner is constrained to set $\mu_t = \mu^{CR}$ for all t

Note: The continuation value v_T is what some researchers call the "value of a Ramsey plan under a time-less perspective." A more descriptive phrase is "the value of the worst continuation Ramsey plan."

42.9 Adding Some Human Intelligence

We have used our machine learning algorithms to compute a Ramsey plan.

By plotting it, we learned that the Ramsey planner makes $\vec{\mu}$ and $\vec{\theta}$ both vary over time.

- $\vec{\theta}$ and $\vec{\mu}$ both decline monotonically
- both of them converge from above to the same constant $\vec{\mu}$

Hidden from view, there is a recursive structure in the $\vec{\mu}, \vec{\theta}$ chosen by the Ramsey planner that we want to bring out.

To do so, we'll have to add some **human intelligence** to the **artificial intelligence** embodied in our machine learning approach.

To proceed, we'll compute least squares linear regressions of some components of $\vec{\theta}$ and $\vec{\mu}$ on others.

We hope that these regressions will reveal structure hidden within the $\vec{\mu}^R, \vec{\theta}^R$ sequences associated with a Ramsey plan.

It is worth pausing to think about roles being played here by **human** intelligence and **artificial** intelligence.

Artificial intelligence in the form of some Python code and a computer is running the regressions for us.

But we are free to regress anything on anything else.

Human intelligence tells us what regressions to run.

Additional inputs of human intelligence will be required fully to appreciate what those regressions reveal about the structure of a Ramsey plan.

Note: When we eventually get around to trying to understand the regressions below, it will worthwhile to study the reasoning that let Chang [Chang, 1998] to choose θ_t as his key state variable.

We begin by regressing μ_t on a constant and θ_t.

This might seem strange because, after all, equation (42.3) asserts that inflation at time t is determined $\{\mu_s\}_{s=t}^{\infty}$

Nevertheless, we'll run this regression anyway.

```
# First regression: μ_t on a constant and θ_t
X1_θ = sm.add_constant(θs)
model1 = sm.OLS(μs, X1_θ)
results1 = model1.fit()

# Print regression summary
print("Regression of μ_t on a constant and θ_t:")
print(results1.summary(slim=True))
```

```
Regression of μ_t on a constant and θ_t:
                            OLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:                       1.000
Model:                            OLS   Adj. R-squared:                  1.000
No. Observations:                  40   F-statistic:                 9.461e+12
Covariance Type:            nonrobust   Prob (F-statistic):          3.82e-218
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const          0.0645   5.54e-08   1.16e+06      0.000       0.065       0.065
x1             1.5995    5.2e-07   3.08e+06      0.000       1.600       1.600
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly␣
 ↪specified.
```

Our regression tells us that the affine function

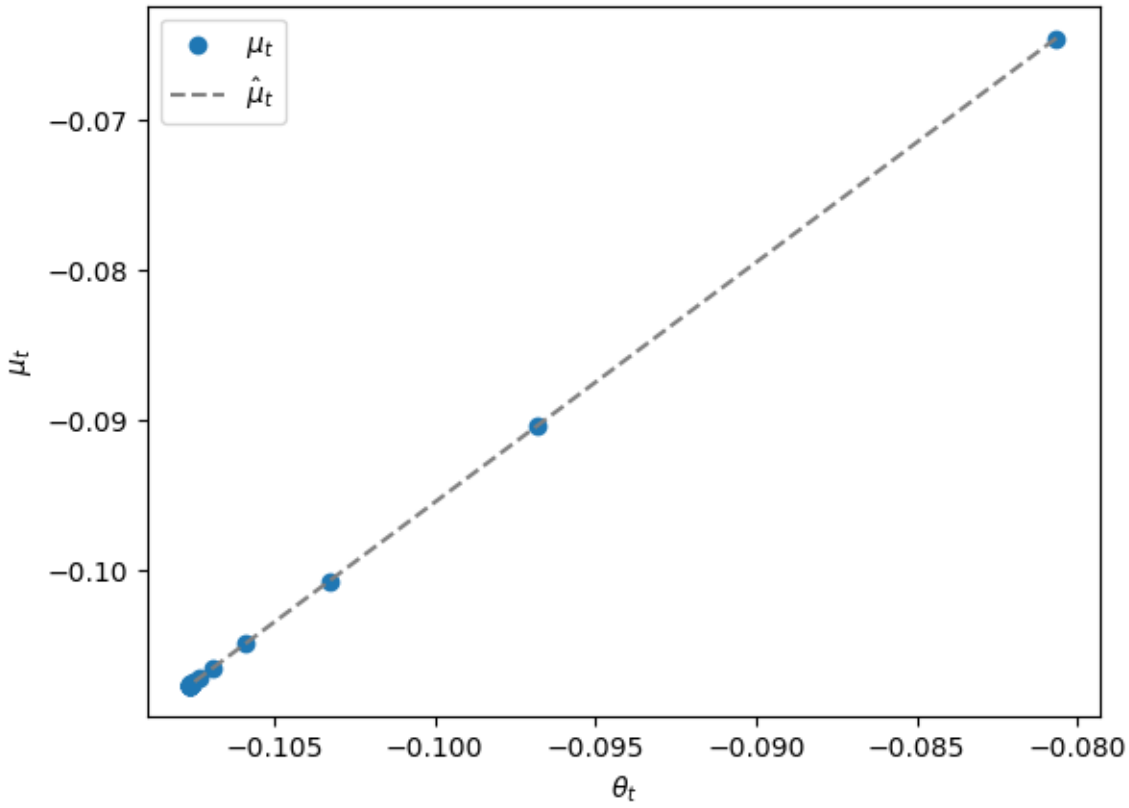
$$\mu_t = .0645 + 1.5995\theta_t$$

fits perfectly along the Ramsey outcome $\vec{\mu}, \vec{\theta}$.

Note: Of course, this means that a regression of θ_t on μ_t and a constant would also fit perfectly.

Let's plot the regression line $\mu_t = .0645 + 1.5995\theta_t$ and the points (θ_t, μ_t) that lie on it for $t = 0, \ldots, T$.

```
plt.scatter(θs, μs, label=r'$\mu_t$')
plt.plot(θs, results1.predict(X1_θ), 'grey', label=r'$\hat \mu_t$', linestyle='--')
plt.xlabel(r'$\theta_t$')
plt.ylabel(r'$\mu_t$')
plt.legend()
plt.show()
```



The time 0 pair (θ_0, μ_0) appears as the point on the upper right.

Points (θ_t, μ_t) for succeeding times appear further and further to the lower left and eventually converge to $(\bar{\mu}, \bar{\mu})$.

Next, we'll run a linear regression of θ_{t+1} against θ_t and a constant.

```
# Second regression: θ_{t+1} on a constant and θ_t
θ_t = np.array(θs[:-1])  # θ_t
θ_t1 = np.array(θs[1:])  # θ_{t+1}
X2_θ = sm.add_constant(θ_t)  # Add a constant term for the intercept
model2 = sm.OLS(θ_t1, X2_θ)
results2 = model2.fit()

# Print regression summary
print("\nRegression of θ_{t+1} on a constant and θ_t:")
print(results2.summary(slim=True))
```

```
Regression of θ_{t+1} on a constant and θ_t:
                            OLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:                       1.000
Model:                            OLS   Adj. R-squared:                  1.000
No. Observations:                  39   F-statistic:                 7.105e+11
Covariance Type:            nonrobust   Prob (F-statistic):          7.46e-192
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const         -0.0645   5.06e-08  -1.27e+06      0.000      -0.065      -0.065
```

(continues on next page)

(continued from previous page)

```
x1              0.4005    4.75e-07    8.43e+05       0.000        0.400        0.400
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly␣
 ↪specified.
```

We find that the regression line fits perfectly and thus discover the affine relationship

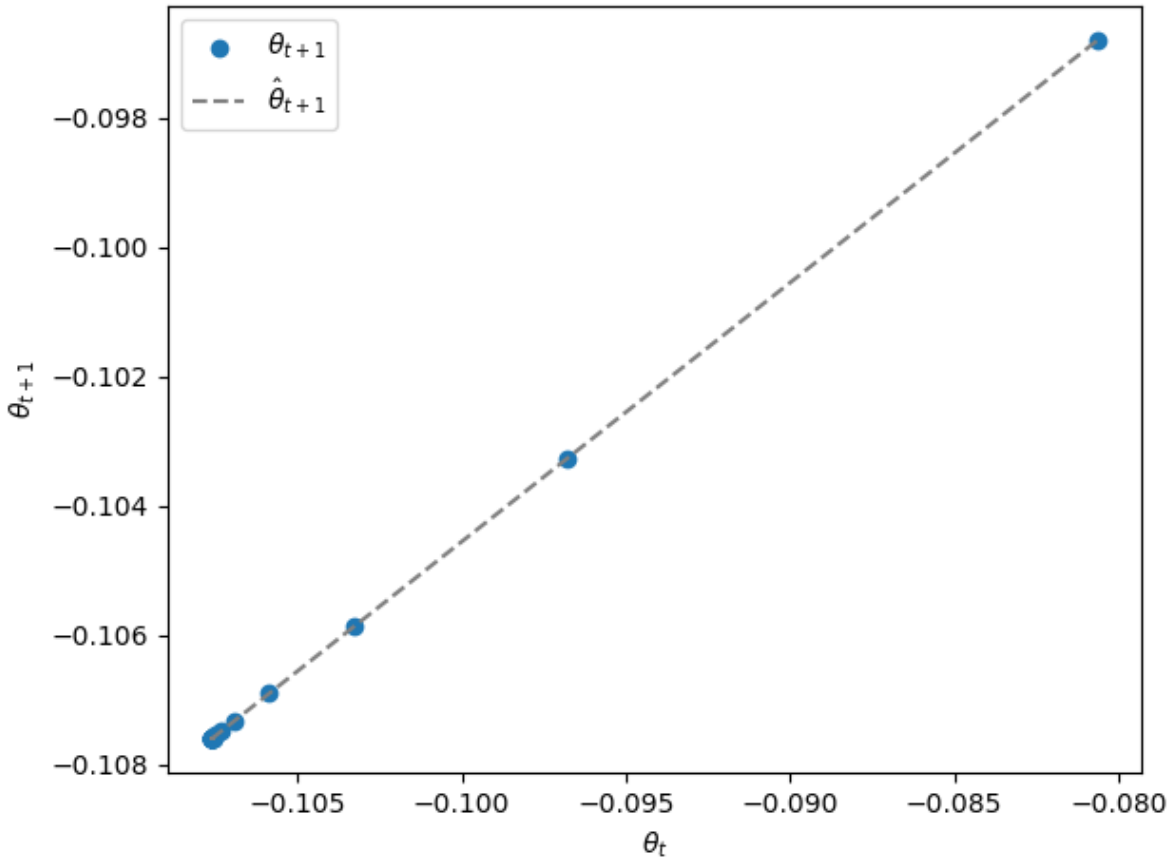
$$\theta_{t+1} = -.0645 + .4005\theta_t$$

that prevails along the Ramsey outcome for inflation.

Let's plot θ_t for $t = 0, 1, \ldots, T$ along the line.

```
plt.scatter(θ_t, θ_t1, label=r'$\theta_{t+1}$')
plt.plot(θ_t, results2.predict(X2_θ), color='grey', label=r'$\hat θ_{t+1}$',␣
 ↪linestyle='--')
plt.xlabel(r'$\theta_t$')
plt.ylabel(r'$\theta_{t+1}$')
plt.legend()

plt.tight_layout()
plt.show()
```



Points for succeeding times appear further and further to the lower left and eventually converge to $\bar{\mu}, \bar{\mu}$.

Next we ask Python to regress continuation value v_t against a constant, θ_t, and θ_t^2.

$$v_t = g_0 + g_1\theta_t + g_2\theta_t^2.$$

```
# Third regression: v_t on a constant, θ_t and θ^2_t
X3_θ = np.column_stack((np.ones(T), θs, θs**2))
model3 = sm.OLS(v_t, X3_θ)
results3 = model3.fit()


# Print regression summary
print("\nRegression of v_t on a constant, θ_t and θ^2_t:")
print(results3.summary(slim=True))
```

```
Regression of v_t on a constant, θ_t and θ^2_t:
                            OLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:                       1.000
Model:                            OLS   Adj. R-squared:                  1.000
No. Observations:                  40   F-statistic:                 5.481e+08
Covariance Type:            nonrobust   Prob (F-statistic):          5.94e-139
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const          6.8052   5.91e-06   1.15e+06      0.000       6.805       6.805
x1            -0.7581      0.000  -6032.921      0.000      -0.758      -0.758
x2            -4.6996      0.001  -7136.554      0.000      -4.701      -4.698
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly␣
 ↪specified.
[2] The condition number is large, 3.5e+04. This might indicate that there are
strong multicollinearity or other numerical problems.
```

The regression has an R^2 equal to 1 and so fits perfectly.

However, notice the warning about the high condition number.

As indicated in the printout, this is a consequence of θ_t and θ_t^2 being highly correlated along the Ramsey plan.

```
np.corrcoef(θs, θs**2)
```

```
array([[ 1.        , -0.99942156],
       [-0.99942156,  1.        ]])
```

Let's plot v_t against θ_t along with the nonlinear regression line.

```
θ_grid = np.linspace(min(θs), max(θs), 100)
X3_grid = np.column_stack((np.ones(len(θ_grid)), θ_grid, θ_grid**2))

plt.scatter(θs, v_t)
plt.plot(θ_grid, results3.predict(X3_grid), color='grey',
         label=r'$\hat v_t$', linestyle='--')
plt.axhline(V_CR, color='C1', alpha=0.5)
```

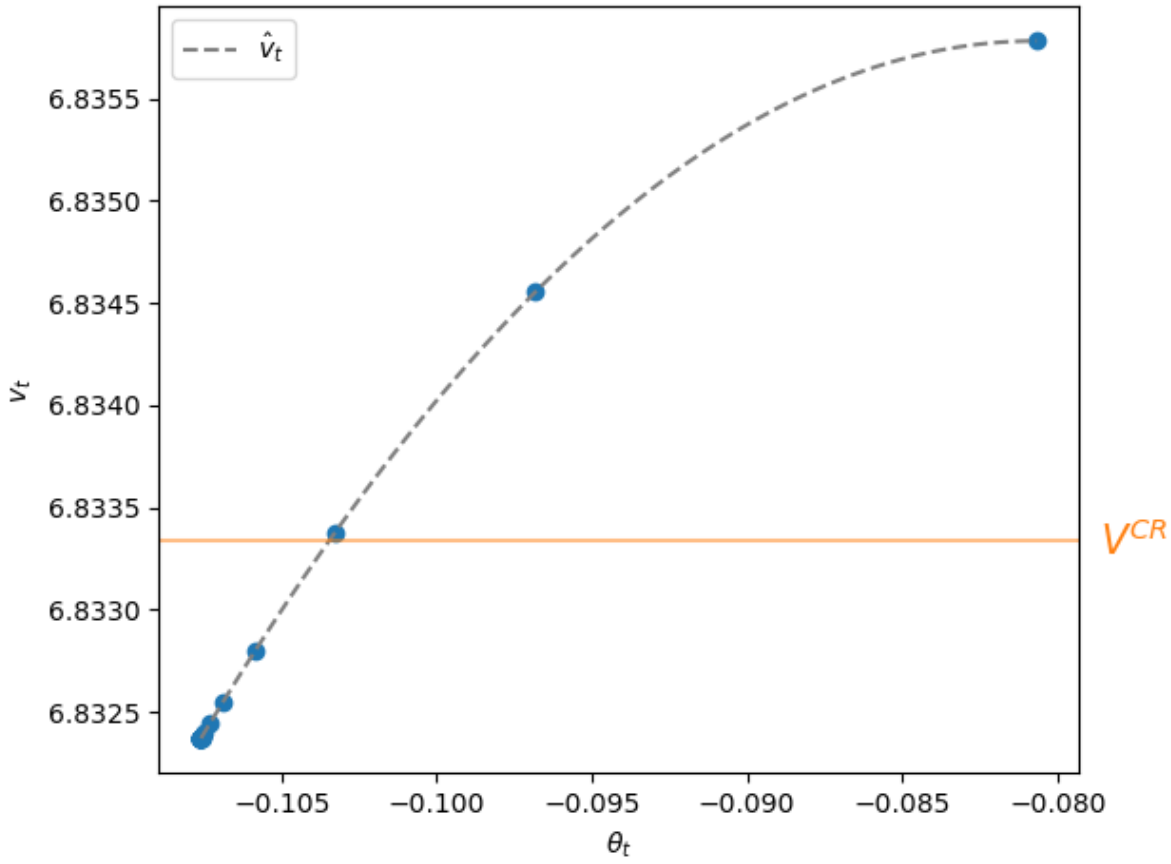
(continues on next page)

(continued from previous page)

```
plt.text(max(θ_grid) - max(θ_grid)*0.025, V_CR, '$V^{CR}$', color='C1',
         va='center', clip_on=False, fontsize=15)

plt.xlabel(r'$\theta_{t}$')
plt.ylabel(r'$v_t$')
plt.legend()

plt.tight_layout()
plt.show()
```



The highest continuation value v_0 at $t = 0$ appears at the peak of the function quadratic function $g_0 + g_1\theta_t + g_2\theta_t^2$.

Subsequent values of v_t for $t \geq 1$ appear to the lower left of the pair (θ_0, v_0) and converge monotonically from above to v_T at time T.

The value V^{CR} attained by the Ramsey plan that is restricted to be a constant $\mu_t = \mu^{CR}$ sequence appears as a horizontal line.

Evidently, continuation values $v_t > V^{CR}$ for $t = 0, 1, 2$ while $v_t < V^{CR}$ for $t \geq 3$.

42.10 What has Machine Learning Taught Us?

Our regressions tells us that along the Ramsey outcome $\vec{\mu}^R, \vec{\theta}^R$, the linear function

$$\mu_t = .0645 + 1.5995\theta_t$$

fits perfectly and that so do the regression lines

$$\theta_{t+1} = -.0645 + .4005\theta_t$$

$$v_t = 6.8052 - .7580\theta_t - 4.6991\theta_t^2.$$

Assembling these regressions, we have discovered run for our single Ramsey outcome path $\vec{\mu}^R, \vec{\theta}^R$ that along a Ramsey plan, the following relationships prevail:

$$\begin{aligned} \theta_0 &= \theta_0^R \\ \mu_t &= b_0 + b_1\theta_t \\ \theta_{t+1} &= d_0 + d_1\theta_t \end{aligned} \tag{42.12}$$

where the initial value θ_0^R was computed along with other components of $\vec{\mu}^R, \vec{\theta}^R$ when we computed the Ramsey plan, and where b_0, b_1, d_0, d_1 are parameters whose values we estimated with our regressions.

In addition, we learned that continuation values are described by the quadratic function

$$v_t = g_0 + g_1\theta_t + g_2\theta_t^2$$

We discovered these relationships by running some carefully chosen regressions and staring at the results, noticing that the R^2's of unity tell us that the fits are perfect.

We have learned much about the structure of the Ramsey problem.

However, by using the methods and ideas that we have deployed in this lecture, it is challenging to say more.

There are many other linear regressions among components of $\vec{\mu}^R, \theta^R$ that would also have given us perfect fits.

For example, we could have regressed θ_t on μ_t and obtained the same R^2 value.

Actually, wouldn't that direction of fit have made more sense?

After all, the Ramsey planner chooses $\vec{\mu}$, while $\vec{\theta}$ is an outcome that reflects the represenative agent's response to the Ramsey planner's choice of $\vec{\mu}$.

Isn't it more natural then to expect that we'd learn more about the structure of the Ramsey problem from a regression of components of $\vec{\theta}$ on components of $\vec{\mu}$?

To answer these questions, we'll have to deploy more economic theory.

We do that in this quantecon lecture *Time Inconsistency of Ramsey Plans*.

There, we'll discover that system (42.12) is actually a very good way to represent a Ramsey plan because it reveals many things about its structure.

Indeed, in that lecture, we show how to compute the Ramsey plan using **dynamic programming squared** and provide a Python class `ChangLQ` that performs the calculations.

We have deployed `ChangLQ` earlier in this lecture to compute a baseline Ramsey plan to which we have compared outcomes from our application of the cruder machine learning approaches studied here.

Let's use the code to compute the parameters d_0, d_1 for the decision rule for μ_t and the parameters d_0, d_1 in the updating rule for θ_{t+1} in representation (42.12).

First, we'll again use `ChangLQ` to compute these objects (along with a number of others).

```
clq = ChangLQ(β=0.85, c=2, T=T)
```

Now let's print out the decision rule for μ_t uncovered by applying dynamic programming squared.

```
print("decision rule for μ")
print(f'-(b_0, b_1) = ({-clq.b0:.6f}, {-clq.b1:.6f})')
```

```
decision rule for μ
-(b_0, b_1) = (0.064507, 1.599536)
```

Now let's print out the decision rule for θ_{t+1} uncovered by applying dynamic programming squared.

```
print("decision rule for θ(t+1) as function of θ(t)")
print(f'(d_0, d_1) =  ({clq.d0:.6f}, {clq.d1:.6f})')
```

```
decision rule for θ(t+1) as function of θ(t)
(d_0, d_1) =  (-0.064507, 0.400464)
```

Evidently, these agree with the relationships that we discovered by running regressions on the Ramsey outcomes $\vec{\mu}^R, \vec{\theta}^R$ that we constructed with either of our machine learning algorithms.

We have set the stage for this quantecon lecture *Time Inconsistency of Ramsey Plans*.

We close this lecture by giving a hint about an insight of Chang [Chang, 1998] that underlies much of quantecon lecture *Time Inconsistency of Ramsey Plans*.

Chang noticed how equation (42.3) shows that an equivalence class of continuation money growth sequences $\{\mu_{t+j}\}_{j=0}^{\infty}$ deliver the same θ_t.

Consequently, equations (42.1) and (42.3) indicate that θ_t intermediates how the government's choices of μ_{t+j}, $j = 0, 1, \ldots$ impinge on time t real balances $m_t - p_t = -\alpha\theta_t$.

In lecture *Time Inconsistency of Ramsey Plans*, we'll see how Chang [Chang, 1998] put this insight to work.

CHAPTER

FORTYTHREE

TIME INCONSISTENCY OF RAMSEY PLANS

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install --upgrade quantecon
```

43.1 Overview

This lecture describes a linear-quadratic version of a model that Guillermo Calvo [Calvo, 1978] used to analyze the **time inconsistency** of optimal government plans.

We use the model as a laboratory in which we explore consequences of different timing protocols for government decision making.

The model focuses on intertemporal tradeoffs between

- benefits that anticipations of future deflation generate by decreasing costs of holding real money balances and thereby increasing a representative agent's *liquidity*, as measured by his or her holdings of real money balances, and
- costs associated with the distorting taxes that the government must levy in order to acquire the paper money that it will destroy in order to generate anticipated deflation

Model features include

- rational expectations
- alternative possible timing protocols for government choices of a sequence of money growth rates
- costly government actions at all dates $t \geq 1$ that increase household utilities at dates before t
- alternative possible sets of Bellman equations, one set for each timing protocol
  - for example, in a timing protocol used to pose a **Ramsey plan**, a government chooses an infinite sequence of money supply growth rates once and for all at time 0.
  - in this timing protocol, there are two value functions and associated Bellman equations, one that expresses a representative private expectation of future inflation as a function of current and future government actions, another that describes the value function of a Ramsey planner
  - in other timing protocols, other Bellman equations and associated value functions will appear

A theme of this lecture is that timing protocols for government decisions affect outcomes.

We'll use ideas from papers by Cagan [Cagan, 1956], Calvo [Calvo, 1978], and Chang [Chang, 1998] as well as from chapter 19 of [Ljungqvist and Sargent, 2018].

In addition, we'll use ideas from linear-quadratic dynamic programming described in Linear Quadratic Control as applied to Ramsey problems in *Stackelberg plans*.

We specify model fundamentals in ways that allow us to use linear-quadratic discounted dynamic programming to compute an optimal government plan under each of our timing protocols.

A sister lecture *Machine Learning a Ramsey Plan* studies some of the same models but does not use dynamic programming.

Instead it uses a **machine learning** approach that does not explicitly recognize the recursive structure structure of the Ramsey problem that Chang [Chang, 1998] saw and that we exploit in this lecture.

In addition to what's in Anaconda, this lecture will use the following libraries:

```
!pip install --upgrade quantecon
```

We'll start with some imports:

```
import numpy as np
from quantecon import LQ
import matplotlib.pyplot as plt
from matplotlib.ticker import FormatStrFormatter
import pandas as pd
from IPython.display import display, Math
```

43.2 Model Components

There is no uncertainty.

Let:

- p_t be the log of the price level
- m_t be the log of nominal money balances
- $\theta_t = p_{t+1} - p_t$ be the net rate of inflation between t and $t+1$
- $\mu_t = m_{t+1} - m_t$ be the net rate of growth of nominal balances

The demand for real balances is governed by a discrete time version of Sargent and Wallace's [Sargent and Wallace, 1973] perfect foresight version of a Cagan [Cagan, 1956] demand function for real balances:

$$m_t - p_t = -\alpha(p_{t+1} - p_t)\ ,\ \alpha > 0 \tag{43.1}$$

for $t \geq 0$.

Equation (43.1) asserts that the demand for real balances is inversely related to the public's expected rate of inflation, which equals the actual rate of inflation because there is no uncertainty here.

Note: When there is no uncertainty, an assumption of **rational expectations** becomes equivalent to **perfect foresight**. [Sargent, 1977] presents a rational expectations version of the model when there is uncertainty.

Subtracting the demand function (43.1) at time t from the time $t+1$ version of this demand function gives

$$\mu_t - \theta_t = -\alpha\theta_{t+1} + \alpha\theta_t$$

or

$$\theta_t = \frac{\alpha}{1+\alpha}\theta_{t+1} + \frac{1}{1+\alpha}\mu_t \tag{43.2}$$

Because $\alpha > 0$, $0 < \frac{\alpha}{1+\alpha} < 1$.

Definition: For scalar b_t, let L^2 be the space of sequences $\{b_t\}_{t=0}^{\infty}$ satisfying

$$\sum_{t=0}^{\infty} b_t^2 < +\infty$$

We say that a sequence that belongs to L^2 is **square summable**.

When we assume that the sequence $\vec{\mu} = \{\mu_t\}_{t=0}^{\infty}$ is square summable and we require that the sequence $\vec{\theta} = \{\theta_t\}_{t=0}^{\infty}$ is square summable, the linear difference equation (43.2) can be solved forward to get:

$$\theta_t = \frac{1}{1+\alpha} \sum_{j=0}^{\infty} \left(\frac{\alpha}{1+\alpha} \right)^j \mu_{t+j} \tag{43.3}$$

Insight: Chang [Chang, 1998] noted that equations (43.1) and (43.3) show that θ_t intermediates how choices of μ_{t+j}, $j = 0, 1, \ldots$ impinge on time t real balances $m_t - p_t = -\alpha\theta_t$.

An equivalence class of continuation money growth sequences $\{\mu_{t+j}\}_{j=0}^{\infty}$ deliver the same θ_t.

We shall use this insight to simplify our analysis of alternative government policy problems.

That future rates of money creation influence earlier rates of inflation makes timing protocols matter for modeling optimal government policies.

We can represent restriction (43.3) as

$$\begin{bmatrix} 1 \\ \theta_{t+1} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & \frac{1+\alpha}{\alpha} \end{bmatrix} \begin{bmatrix} 1 \\ \theta_t \end{bmatrix} + \begin{bmatrix} 0 \\ -\frac{1}{\alpha} \end{bmatrix} \mu_t \tag{43.4}$$

or

$$x_{t+1} = Ax_t + B\mu_t \tag{43.5}$$

Even though θ_0 is to be determined by our model and so is not an initial condition, as it ordinarily would be in the state-space model described in our lecture on Linear Quadratic Control, we nevertheless write the model in the state-space form (43.5).

We use form (43.5) because we want to apply an approach described in our lecture on *Stackelberg plans*.

Notice that $\frac{1+\alpha}{\alpha} > 1$ is an eigenvalue of transition matrix A that threatens to destabilize the state-space system.

Indeed, for arbitrary, $\vec{\mu} = \{\mu_t\}_{t=0}^{\infty}$ sequences, $\vec{\theta} = \{\theta_t\}_{t=0}^{\infty}$ will not be square summable.

But the government planner will design a decision rule for μ_t that stabilizes the system and renders $\vec{\theta}$ square summable.

The government values a representative household's utility of real balances at time t according to the utility function

$$U(m_t - p_t) = u_0 + u_1(m_t - p_t) - \frac{u_2}{2}(m_t - p_t)^2, \quad u_0 > 0, u_1 > 0, u_2 > 0 \tag{43.6}$$

The money demand function (43.1) and the utility function (43.6) imply that

$$U(-\alpha\theta_t) = u_0 + u_1(-\alpha\theta_t) - \frac{u_2}{2}(-\alpha\theta_t)^2. \tag{43.7}$$

43.3 Friedman's Optimal Rate of Deflation

According to (43.7), the `bliss level` of real balances is $\frac{u_1}{u_2}$ and the inflation rate that attains it is

$$\theta_t = \theta^* = -\frac{u_1}{u_2\alpha} \tag{43.8}$$

Milton Friedman recommended that the government withdraw and destroy money at a rate that implies an inflation rate given by (43.8).

In our setting, that could be accomplished by setting

$$\mu_t = \mu^* = \theta^*, t \geq 0 \tag{43.9}$$

where θ^* is given by equation (43.8).

Milton Friedman assumed that the taxes that government imposes to collect money at rate μ_t do not distort economic decisions, e.g., they are lump-sum taxes.

43.4 Calvo's Distortion

The starting point of Calvo [Calvo, 1978] and Chang [Chang, 1998] is that lump sum taxes are not available.

Instead, the government acquires money by levying taxes that distort decisions and thereby impose costs on the representative consumer.

In the models of Calvo [Calvo, 1978] and Chang [Chang, 1998], the government takes those tax-distortion costs into account.

The government balances the **costs** of imposing the distorting taxes needed to acquire the money that it destroys in order to generate deflation against the **benefits** that expected deflation generates by raising the representative household's real money balances.

Let's see how the government does that.

Via equation (43.3), a government plan $\vec{\mu} = \{\mu_t\}_{t=0}^{\infty}$ leads to a sequence of inflation outcomes $\vec{\theta} = \{\theta_t\}_{t=0}^{\infty}$.

The government incurs social costs $\frac{c}{2}\mu_t^2$ at t when it changes the stock of nominal money balances at rate μ_t.

Therefore, the one-period welfare function of a benevolent government is:

$$s(\theta_t, \mu_t) := -r(x_t, \mu_t) = \begin{bmatrix} 1 \\ \theta_t \end{bmatrix}' \begin{bmatrix} u_0 & -\frac{u_1\alpha}{2} \\ -\frac{u_1\alpha}{2} & -\frac{u_2\alpha^2}{2} \end{bmatrix} \begin{bmatrix} 1 \\ \theta_t \end{bmatrix} - \frac{c}{2}\mu_t^2 = -x_t' R x_t - Q\mu_t^2 \tag{43.10}$$

The government's time 0 value is

$$v_0 = -\sum_{t=0}^{\infty} \beta^t r(x_t, \mu_t) = \sum_{t=0}^{\infty} \beta^t s(\theta_t, \mu_t) \tag{43.11}$$

where $\beta \in (0, 1)$ is a discount factor.

Note: We define $r(x_t, \mu_t) := -s(\theta_t, \mu_t)$ in order to represent the government's **maximization** problem in terms of our Python code for solving linear quadratic discounted dynamic programs. In first LQ control lecture and some other quantecon lectures, we formulated these as **loss minimization** problems.

The government's time t continuation value v_t is

$$v_t = \sum_{j=0}^{\infty} \beta^j s(\theta_{t+j}, \mu_{t+j}). \tag{43.12}$$

We can represent dependence of v_0 on $(\vec{\theta}, \vec{\mu})$ recursively via the difference equation

$$v_t = s(\theta_t, \mu_t) + \beta v_{t+1} \tag{43.13}$$

It is useful to evaluate (43.13) under a time-invariant money growth rate $\mu_t = \bar{\mu}$ that according to equation (43.3) would bring forth a constant inflation rate equal to $\bar{\mu}$.

Under that policy,

$$v_t = V(\bar{\mu}) = \frac{s(\bar{\mu}, \bar{\mu})}{1 - \beta} \tag{43.14}$$

for all $t \geq 0$.

Values of $V(\bar{\mu})$ computed according to formula (43.14) for three different values of $\bar{\mu}$ will play important roles below.

- $V(\mu^{MPE})$ is the value of attained by the government in a **Markov perfect equilibrium**
- $V(\mu^R_\infty)$ is the value that a continuation Ramsey planner attains at $t \to +\infty$
  - We shall discover that $V(\mu^R_\infty)$ is the worst continuation value attained along a Ramsey plan
- $V(\mu^{CR})$ is the value of attained by the government in a **constrained to constant μ equilibrium**

43.5 Structure

The following structure is induced by a representative agent's behavior as summarized by the demand function for money (43.1) that leads to equation (43.3), which tells how future settings of μ affect the current value of θ.

Equation (43.3) maps a **policy** sequence of money growth rates $\vec{\mu} = \{\mu_t\}_{t=0}^\infty \in L^2$ into an inflation sequence $\vec{\theta} = \{\theta_t\}_{t=0}^\infty \in L^2$.

These in turn induce a discounted value to a government sequence $\vec{v} = \{v_t\}_{t=0}^\infty \in L^2$ that satisfies recursion (43.13).

Thus, a triple of sequences $(\vec{\mu}, \vec{\theta}, \vec{v})$ depends on a sequence $\vec{\mu} \in L^2$.

At this point $\vec{\mu} \in L^2$ is an arbitrary exogenous policy.

A theory of government decisions will make $\vec{\mu}$ endogenous, i.e., a theoretical **output** instead of an **input**.

43.5.1 Intertemporal Aspects

Criterion function (43.11) and the constraint system (43.5) exhibit the following structure:

- Setting the money growth rate $\mu_t \neq 0$ imposes costs $\frac{c}{2}\mu_t^2$ at time t and at no other times; but
- The money growth rate μ_t affects the government's one-period utilities at all dates $s = 0, 1, \ldots, t$.

This structure sets the stage for the emergence of a time-inconsistent optimal government plan under a **Ramsey** timing protocol

- it is also called a **Stackelberg** timing protocol.

We'll study outcomes under a Ramsey timing protocol.

We'll also study outcomes under other timing protocols.

43.6 Three Timing Protocols

We consider three models of government policy making that differ in

- *what* a policymaker chooses, either a sequence $\vec{\mu}$ or just μ_t in a single period t.
- *when* a policymaker chooses, either once and for all at time 0, or at one or more times $t \geq 0$.
- what a policymaker *assumes* about how its choice of μ_t affects the representative agent's expectations about earlier and later inflation rates.

In two of our models, a single policymaker chooses a sequence $\{\mu_t\}_{t=0}^{\infty}$ once and for all, knowing how μ_t affects household one-period utilities at dates $s = 0, 1, \ldots, t-1$

- these two models thus employ a **Ramsey** or **Stackelberg** timing protocol.

In a third model, there is a sequence of policymaker indexed by $t \in \{0, 1, \ldots\}$, each of whom sets only μ_t.

- a time t policymaker cares only about v_t and ignores effects that its choice of μ_t has on v_s at dates $s = 0, 1, \ldots, t-1$.

The three models differ with respect to timing protocols, constraints on government choices, and government policymakers' beliefs about how their decisions affect the representative agent's beliefs about future government decisions.

The models are distinguished by their having either

- A single Ramsey planner that chooses a sequence $\{\mu_t\}_{t=0}^{\infty}$ once and for all at time 0; or
- A single Ramsey planner that chooses a sequence $\{\mu_t\}_{t=0}^{\infty}$ once and for all at time 0 subject to the constraint that $\mu_t = \mu$ for all $t \geq 0$; or
- A sequence of distinct policymakers indexed by $t = 0, 1, 2, \ldots$
  - a time t policymaker chooses μ_t only and forecasts that future government decisions are unaffected by its choice.

The first model describes a **Ramsey plan** chosen by a **Ramsey planner**

The second model describes a **Ramsey plan** chosen by a **Ramsey planner constrained to choose a time-invariant** μ

The third model describes a **Markov perfect equilibrium**

Note: In the quantecon lecture *Sustainable Plans for a Calvo Model*, we'll study outcomes under another timing protocol in which there is a sequence of separate policymakers. A time t policymaker chooses only μ_t but believes that its choice of μ_t shapes the representative agent's beliefs about future rates of money creation and inflation, and through them, future government actions. This is a model of a **credible government policy**, also called a **sustainable plan**. The relationship between outcomes in the first (Ramsey) timing protocol and the *Sustainable Plans for a Calvo Model* timing protocol and belief structure is the subject of a literature on **sustainable** or **credible** public policies (Chari and Kehoe [Chari and Kehoe, 1990] [Stokey, 1989], and Stokey [Stokey, 1991]).

43.7 Note on Dynamic Programming Squared

We'll begin with the timing protocol associated with a Ramsey plan and deploy an application of what we nickname **dynamic programming squared**.

The nickname refers to the feature that a value satisfying one Bellman equation appears as an argument in a value function associated with a second Bellman equation.

Thus, two Bellman equations appear:

- equation (43.1) expresses how θ_t depends on μ_t and θ_{t+1}
- equation (43.5) expresses how value v_t depends on (μ_t, θ_t) and v_{t+1}

A value θ from one Bellman equation appears as an argument of a second Bellman equation for another value v.

43.8 A Ramsey Planner

A Ramsey planner chooses $\{\mu_t, \theta_t\}_{t=0}^{\infty}$ to maximize (43.11) subject to the law of motion (43.5).

We split this problem into two stages, as in the lecture *Stackelberg plans* and [Ljungqvist and Sargent, 2018] Chapter 19.

In the first stage, we take the initial inflation rate θ_0 as given and pose an ordinary discounted dynamic programming problem that in our setting becomes an LQ discounted dynamic programming problem.

In the second stage, we choose an optimal initial inflation rate θ_0.

Define a feasible set of $\{x_{t+1}, \mu_t\}_{t=0}^{\infty}$ sequences, with each sequence belonging to L^2:

$$\Omega(x_0) = \{x_{t+1}, \mu_t\}_{t=0}^{\infty} : x_{t+1} = Ax_t + B\mu_t \ , \ \forall t \geq 0,$$

where we require that $\{x_{t+1}, \mu_t\}_{t=0}^{\infty} \in L^2 \times L^2$.

43.8.1 Subproblem 1

The value function

$$J(x_0) = \max_{\{x_{t+1}, \mu_t\}_{t=0}^{\infty} \in \Omega(x_0)} \sum_{t=0}^{\infty} \beta^t s(x_t, \mu_t) \tag{43.15}$$

satisfies the Bellman equation

$$J(x) = \max_{\mu, x'} \{s(x, \mu) + \beta J(x')\}$$

subject to:

$$x' = Ax + B\mu$$

As in the lecture *Stackelberg plans*, we can map this problem into a linear-quadratic control problem and deduce an optimal value function $J(x)$.

Guessing that $J(x) = -x'Px$ and substituting into the Bellman equation gives rise to the algebraic matrix Riccati equation:

$$P = R + \beta A'PA - \beta^2 A'PB(Q + \beta B'PB)^{-1}B'PA$$

and an optimal decision rule

$$\mu_t = -Fx_t$$

where

$$F = \beta(Q + \beta B'PB)^{-1}B'PA \tag{43.16}$$

The QuantEcon LQ class solves for F and P given inputs Q, R, A, B, and β.

The value function for a (continuation) Ramsey planner is

$$v_t = -\begin{bmatrix}1 & \theta_t\end{bmatrix}\begin{bmatrix}P_{11} & P_{12}\\ P_{21} & P_{22}\end{bmatrix}\begin{bmatrix}1\\ \theta_t\end{bmatrix}$$

or

$$v_t = -P_{11} - 2P_{21}\theta_t - P_{22}\theta_t^2$$

or

$$v_t = g_0 + g_1\theta_t + g_2\theta_t^2 \tag{43.17}$$

where

$$g_0 = -P_{11}, \quad g_1 = -2P_{21}, \quad g_2 = -P_{22}$$

The Ramsey plan for setting μ_t is

$$\mu_t = -\begin{bmatrix}F_1 & F_2\end{bmatrix}\begin{bmatrix}1\\ \theta_t\end{bmatrix}$$

or

$$\mu_t = b_0 + b_1\theta_t \tag{43.18}$$

where $b_0 = -F_1, b_1 = -F_2$ and F satisfies equation (43.16),

The Ramsey planner's decision rule for updating θ_{t+1} is

$$\theta_{t+1} = d_0 + d_1\theta_t \tag{43.19}$$

where $\begin{bmatrix} d_0 & d_1 \end{bmatrix}$ is the second row of the closed-loop matrix $A - BF$ for computed in subproblem 1 above.

The linear quadratic control problem (43.15) satisfies regularity conditions that guarantee that $A - BF$ is a stable matrix (i.e., its maximum eigenvalue is strictly less than 1 in absolute value).

Consequently, we are assured that

$$|d_1| < 1, \tag{43.20}$$

a stability condition that will play an important role.

It remains for us to describe how the Ramsey planner sets θ_0.

Subproblem 2 does that.

43.8.2 Subproblem 2

The value of the Ramsey problem is

$$V^R = \max_{x_0} J(x_0)$$

We abuse notation slightly by writing $J(x)$ as $J(\theta)$ and rewrite the above equation as

Note: Since $x = \begin{bmatrix}1\\ \theta\end{bmatrix}$, it follows that θ is the only component of x that can possibly vary.

$$V^R = \max_{\theta_0} J(\theta_0)$$

Evidently, V^R is the maximum value of v_0 defined in equation (43.11).

Value function $J(\theta_0)$ satisfies

$$J(\theta_0) = -\begin{bmatrix}1 & \theta_0\end{bmatrix}\begin{bmatrix}P_{11} & P_{12}\\ P_{21} & P_{22}\end{bmatrix}\begin{bmatrix}1\\ \theta_0\end{bmatrix} = -P_{11} - 2P_{21}\theta_0 - P_{22}\theta_0^2$$

The first-order necessary condition for maximizing $J(\theta_0)$ with respect to θ_0 is

$$-2P_{21} - 2P_{22}\theta_0 = 0$$

which implies

$$\theta_0 = \theta_0^R = -\frac{P_{21}}{P_{22}}$$

43.9 Representation of Ramsey Plan

The preceding calculations indicate that we can represent a Ramsey plan $\vec{\mu}$ recursively with the following system created in the spirit of Chang [Chang, 1998]:

$$\begin{aligned}\theta_0 &= \theta_0^R\\ \mu_t &= b_0 + b_1\theta_t\\ v_t &= g_0 + g_1\theta_t + g_2\theta_t^2\\ \theta_{t+1} &= d_0 + d_1\theta_t, \quad d_0 > 0, d_1 \in (0,1)\end{aligned} \tag{43.21}$$

where b_0, b_1, g_0, g_1, g_2 are positive parameters that we shall compute with Python code below.

From condition (43.20), we know that $|d_1| < 1$.

To interpret system (43.21), think of the sequence $\{\theta_t\}_{t=0}^{\infty}$ as a sequence of synthetic **promised inflation rates**.

For some purposes, we can think of these promised inflation rates just as computational devices for generating a sequence $\vec{\mu}$ of money growth rates that when substituted into equation (43.3) generate *actual* rates of inflation.

It can be verified that if we substitute a plan $\vec{\mu} = \{\mu_t\}_{t=0}^{\infty}$ that satisfies these equations into equation (43.3), we obtain the same sequence $\vec{\theta}$ generated by the system (43.21).

(Here an application of the Big K, little k trick is again at work.)

Thus, within the Ramsey plan, **promised inflation** equals **actual inflation**.

System (43.21) implies that under the Ramsey plan

$$\theta_t = d_0\left(\frac{1-d_1^t}{1-d_1}\right) + d_1^t\theta_0^R, \tag{43.22}$$

Because $d_1 \in (0,1)$, it follows from (43.22) that as $t \to \infty$, θ_t^R converges to

$$\lim_{t\to+\infty}\theta_t^R = \theta_\infty^R = \frac{d_0}{1-d_1}. \tag{43.23}$$

Furthermore, we shall see that θ_t^R converges to θ_∞^R from above.

Meanwhile, μ_t varies over time according to

$$\mu_t = b_0 + b_1 d_0 \left(\frac{1 - d_1^t}{1 - d_1} \right) + b_1 d_1^t \theta_0^R. \tag{43.24}$$

Variation of $\vec{\mu}^R, \vec{\theta}^R, \vec{v}^R$ over time are symptoms of time inconsistency.

- The Ramsey planner reaps immediate benefits from promising lower inflation later to be achieved by costly distorting taxes.
- These benefits are intermediated by reductions in expected inflation that precede the reductions in money creation rates that rationalize them, as indicated by equation (43.3).

43.10 Multiple roles of θ_t

The inflation rate θ_t plays three roles:

- In equation (43.3), θ_t is the actual rate of inflation between t and $t+1$.
- In equation (43.2) and (43.3), θ_t is also the public's expected rate of inflation between t and $t+1$.
- In system (43.21), θ_t is a promised rate of inflation chosen by the Ramsey planner at time 0.

That the same variable θ_t takes on these multiple roles brings insights about

- whether the government follows or leads the market,
- forward guidance, and
- inflation targeting.

43.11 Time inconsistency

As discussed in *Stackelberg plans* and *Optimal taxation with state-contingent debt*, a continuation Ramsey plan is not a Ramsey plan.

This is a concise way of characterizing the time inconsistency of a Ramsey plan.

In the present context, a symptom of time inconsistency is that the Ramsey plannner chooses to make μ_t a non-constant function of time t despite the fact that, other than time itself, there is no other state variable.

Thus, in our context, time-variation of $\vec{\mu}$ chosen by a Ramsey planner is the telltale sign of the Ramsey plan's **time inconsistency**.

43.12 Constrained-to-Constant-Growth-Rate Ramsey Plan

We can use brute force to create a government plan that **is** time consistent, i.e., that is a time-invariant function of time.

We simply constrain a planner to choose a time-invariant money growth rate $\bar{\mu}$ so that

$$\mu_t = \bar{\mu}, \quad \forall t \geq 0.$$

We assume that the government knows the perfect foresight outcome implied by equation (43.2) that $\theta_t = \bar{\mu}$ when $\mu_t = \bar{\mu}$ for all $t \geq 0$.

It follows that the value of such a plan is given by $V(\bar{\mu})$ defined inequation (43.14).

Then our restricted Ramsey planner chooses $\bar{\mu}$ to maximize $V(\bar{\mu})$.

We can express $V(\bar{\mu})$ as

$$V(\bar{\mu}) = (1-\beta)^{-1}\left[U(-\alpha\bar{\mu}) - \frac{c}{2}(\bar{\mu})^2\right] \tag{43.25}$$

With the quadratic form (43.6) for the utility function U, the maximizing $\bar{\mu}$ is

$$\mu^{CR} = \max_{\bar{\mu}} V(\bar{\mu}) = -\frac{\alpha u_1}{\alpha^2 u_2 + c} \tag{43.26}$$

The optimal value attained by a *constrained to constant* μ Ramsey planner is

$$V(\mu^{CR}) \equiv V^{CR} = (1-\beta)^{-1}\left[U(-\alpha\mu^{CR}) - \frac{c}{2}(\mu^{CR})^2\right] \tag{43.27}$$

Time-variation of $\vec{\mu}$ chosen by a Ramsey planner is the telltale sign of the Ramsey plan's **time inconsistency**.

Obviously, our constrained-to-constant μ Ramsey planner **must** must choose a plan that is time consistent.

43.13 Markov Perfect Governments

To generate an alternative model of time-consistent government decision making, we assume another timing protocol.

In this one, there is a sequence of government policymakers.

A time t government chooses μ_t and expects all future governments to set $\mu_{t+j} = \bar{\mu}$.

This assumption mirrors an assumption made in this QuantEcon lecture: Markov Perfect Equilibrium.

When it sets μ_t, the government at t believes that $\bar{\mu}$ is unaffected by its choice of μ_t.

According to equation (43.3), the time t rate of inflation is then

$$\theta_t = \frac{1}{1+\alpha}\mu_t + \frac{\alpha}{1+\alpha}\bar{\mu}, \tag{43.28}$$

which expresses inflation θ_t as a geometric weighted average of the money growth today μ_t and money growth from tomorrow onward $\bar{\mu}$.

Given $\bar{\mu}$, the time t government chooses μ_t to maximize:

$$H(\mu_t, \bar{\mu}) = U(-\alpha\theta_t) - \frac{c}{2}\mu_t^2 + \beta V(\bar{\mu}) \tag{43.29}$$

where $V(\bar{\mu})$ is given by formula (43.14) for the time 0 value v_0 of recursion (43.13) under a money supply growth rate that is forever constant at $\bar{\mu}$.

Substituting (43.28) into (43.29) and expanding gives:

$$\begin{aligned} H(\mu_t, \bar{\mu}) = u_0 + u_1\left(-\frac{\alpha^2}{1+\alpha}\bar{\mu} - \frac{\alpha}{1+\alpha}\mu_t\right) - \frac{u_2}{2}\left(-\frac{\alpha^2}{1+\alpha}\bar{\mu} - \frac{\alpha}{1+\alpha}\mu_t\right)^2 \\ - \frac{c}{2}\mu_t^2 + \beta V(\bar{\mu}) \end{aligned} \tag{43.30}$$

The first-order necessary condition for maximizing $H(\mu_t, \bar{\mu})$ with respect to μ_t is:

$$-\frac{\alpha}{1+\alpha}u_1 - u_2(-\frac{\alpha^2}{1+\alpha}\bar{\mu} - \frac{\alpha}{1+\alpha}\mu_t)(-\frac{\alpha}{1+\alpha}) - c\mu_t = 0$$

Rearranging we get the time t government's best response map

$$\mu_t = f(\bar{\mu})$$

where

$$f(\bar{\mu}) = \frac{-u_1}{\frac{1+\alpha}{\alpha}c + \frac{\alpha}{1+\alpha}u_2} - \frac{\alpha^2 u_2}{\left[\frac{1+\alpha}{\alpha}c + \frac{\alpha}{1+\alpha}u_2\right](1+\alpha)}\bar{\mu}$$

A **Markov Perfect Equilibrium** (MPE) outcome μ^{MPE} is a fixed point of the best response map:

$$\mu^{MPE} = f(\mu^{MPE})$$

Calculating μ^{MPE}, we find

$$\mu^{MPE} = \frac{-u_1}{\frac{1+\alpha}{\alpha}c + \frac{\alpha}{1+\alpha}u_2 + \frac{\alpha^2}{1+\alpha}u_2}$$

This can be simplified to

$$\mu^{MPE} = -\frac{\alpha u_1}{\alpha^2 u_2 + (1+\alpha)c}. \tag{43.31}$$

The value of a Markov perfect equilibrium is

$$V^{MPE} = \frac{s(\mu^{MPE}, \mu^{MPE})}{1-\beta} \tag{43.32}$$

or

$$V^{MPE} = V(\mu^{MPE})$$

where $V(\cdot)$ is given by formula (43.14).

Under the Markov perfect timing protocol

- a government takes $\bar{\mu}$ as given when it chooses μ_t
- we equate $\mu_t = \mu$ only *after* we have computed a time t government's first-order condition for μ_t.

43.14 Outcomes under Three Timing Protocols

We want to compare outcome sequences $\{\theta_t, \mu_t\}$ under three timing protocols associated with

- a standard Ramsey plan with its time-varying $\{\theta_t, \mu_t\}$ sequences
- a Markov perfect equilibrium, with its time-invariant $\{\theta_t, \mu_t\}$ sequences
- a nonstandard Ramsey plan in which the planner is restricted to choose a time-invariant $\mu_t = \mu$ for all $t \geq 0$.

We have computed closed form formulas for several of these outcomes, which we find it convenient to repeat here.

In particular, the constrained to constant inflation Ramsey inflation outcome is μ^{CR}, which according to equation (43.26) is

$$\theta^{CR} = -\frac{\alpha u_1}{\alpha^2 u_2 + c}$$

Equation (43.31) implies that the Markov perfect constant inflation rate is

$$\theta^{MPE} = -\frac{\alpha u_1}{\alpha^2 u_2 + (1+\alpha)c}$$

According to equation (43.8), the bliss level of inflation that we associated with a Friedman rule is

$$\theta^* = -\frac{u_1}{u_2 \alpha}$$

Proposition 1: When $c = 0$, $\theta^{MPE} = \theta^{CR} = \theta^*$ and $\theta_0^R = \theta_\infty^R$.

The first two equalities follow from the preceding three equations.

We'll illustrate the third equality that equates θ_0^R to θ_∞^R with some quantitative examples below.

Proposition 1 draws attention to how a positive tax distortion parameter c alters the optimal rate of deflation that Milton Friedman financed by imposing a lump sum tax.

We'll compute

- $(\vec{\theta}^R, \vec{\mu}^R)$: ordinary time-varying Ramsey sequences
- $(\theta^{MPE} = \mu^{MPE})$: Markov perfect equilibrium (MPE) fixed values
- $(\theta^{CR} = \mu^{CR})$: fixed values associated with a constrained to time-invariant μ Ramsey plan
- θ^*: bliss level of inflation prescribed by a Friedman rule

We will create a class ChangLQ that solves the models and stores their values

```
class ChangLQ:
    """
    Class to solve LQ Chang model
    """
    def __init__(self, β, c, α=1, u0=1, u1=0.5, u2=3, T=1000, θ_n=200):
        # Record parameters
        self.α, self.u0, self.u1, self.u2 = α, u0, u1, u2
        self.β, self.c, self.T, self.θ_n = β, c, T, θ_n

        self.setup_LQ_matrices()
        self.solve_LQ_problem()
        self.compute_policy_functions()
        self.simulate_ramsey_plan()
        self.compute_θ_range()
        self.compute_value_and_policy()

    def setup_LQ_matrices(self):
        # LQ Matrices
        self.R = -np.array([[self.u0, -self.u1 * self.α / 2],
                            [-self.u1 * self.α / 2,
                             -self.u2 * self.α**2 / 2]])
        self.Q = -np.array([[-self.c / 2]])
        self.A = np.array([[1, 0], [0, (1 + self.α) / self.α]])
        self.B = np.array([[0], [-1 / self.α]])

    def solve_LQ_problem(self):
        # Solve LQ Problem (Subproblem 1)
        lq = LQ(self.Q, self.R, self.A, self.B, beta=self.β)
        self.P, self.F, self.d = lq.stationary_values()

        # Compute g0, g1, and g2 (41.16)
        self.g0, self.g1, self.g2 = [-self.P[0, 0],
                                     -2 * self.P[1, 0], -self.P[1, 1]]

        # Compute b0 and b1 (41.17)
        [[self.b0, self.b1]] = self.F

        # Compute d0 and d1 (41.18)
        self.cl_mat = (self.A - self.B @ self.F)  # Closed loop matrix
        [[self.d0, self.d1]] = self.cl_mat[1:]
```

(continues on next page)

(continued from previous page)

```
        # Solve Subproblem 2
        self.θ_R = -self.P[0, 1] / self.P[1, 1]

        # Find the bliss level of θ
        self.θ_B = -self.u1 / (self.u2 * self.α)

    def compute_policy_functions(self):
        # Solve the Markov Perfect Equilibrium
        self.μ_MPE = -self.u1 / ((1 + self.α) / self.α * self.c
                                 + self.α / (1 + self.α)
                                 * self.u2 + self.α**2
                                 / (1 + self.α) * self.u2)
        self.θ_MPE = self.μ_MPE
        self.μ_CR = -self.α * self.u1 / (self.u2 * self.α**2 + self.c)
        self.θ_CR = self.μ_CR

        # Calculate value under MPE and CR economy
        self.J_θ = lambda θ_array: - np.array([1, θ_array]) \
                                   @ self.P @ np.array([1, θ_array]).T
        self.V_θ = lambda θ: (self.u0 + self.u1 * (-self.α * θ)
                              - self.u2 / 2 * (-self.α * θ)**2
                              - self.c / 2 * θ**2) / (1 - self.β)

        self.J_MPE = self.V_θ(self.μ_MPE)
        self.J_CR = self.V_θ(self.μ_CR)

    def simulate_ramsey_plan(self):
        # Simulate Ramsey plan for large number of periods
        θ_series = np.vstack((np.ones((1, self.T)), np.zeros((1, self.T))))
        μ_series = np.zeros(self.T)
        J_series = np.zeros(self.T)
        θ_series[1, 0] = self.θ_R
        [μ_series[0]] = -self.F.dot(θ_series[:, 0])
        J_series[0] = self.J_θ(θ_series[1, 0])

        for i in range(1, self.T):
            θ_series[:, i] = self.cl_mat @ θ_series[:, i-1]
            [μ_series[i]] = -self.F @ θ_series[:, i]
            J_series[i] = self.J_θ(θ_series[1, i])

        self.J_series = J_series
        self.μ_series = μ_series
        self.θ_series = θ_series

    def compute_θ_range(self):
        # Find the range of θ in Ramsey plan
        θ_LB = min(min(self.θ_series[1, :]), self.θ_B)
        θ_UB = max(max(self.θ_series[1, :]), self.θ_MPE)
        θ_range = θ_UB - θ_LB
        self.θ_LB = θ_LB - 0.05 * θ_range
        self.θ_UB = θ_UB + 0.05 * θ_range
        self.θ_range = θ_range

    def compute_value_and_policy(self):
        # Create the θ_space
```

(continues on next page)

(continued from previous page)

```
        self.θ_space = np.linspace(self.θ_LB, self.θ_UB, 200)

        # Find value function and policy functions over range of θ
        self.J_space = np.array([self.J_θ(θ) for θ in self.θ_space])
        self.μ_space = -self.F @ np.vstack((np.ones(200), self.θ_space))
        x_prime = self.cl_mat @ np.vstack((np.ones(200), self.θ_space))
        self.θ_prime = x_prime[1, :]
        self.CR_space = np.array([self.V_θ(θ) for θ in self.θ_space])

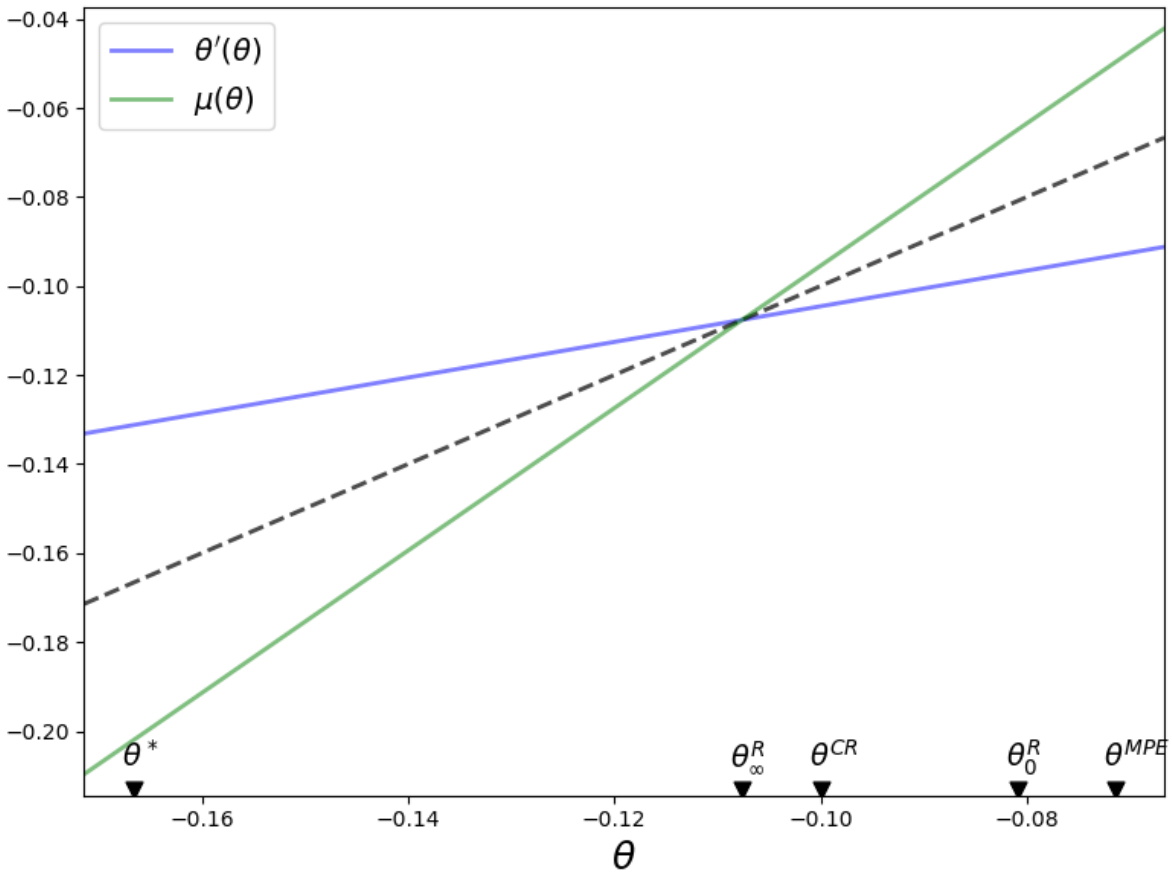
        self.μ_space = self.μ_space[0, :]

        # Calculate J_range, J_LB, and J_UB
        self.J_range = np.ptp(self.J_space)
        self.J_LB = np.min(self.J_space) - 0.05 * self.J_range
        self.J_UB = np.max(self.J_space) + 0.05 * self.J_range
```

Let's create an instance of ChangLQ with the following parameters:

```
clq = ChangLQ(β=0.85, c=2)
```

The following code plots policy functions for a continuation Ramsey planner.



The dotted line in the above graph is the 45-degree line.

The blue line shows the choice of $\theta_{t+1} = \theta'$ chosen by a continuation Ramsey planner who inherits $\theta_t = \theta$.

The green line shows a continuation Ramsey planner's choice of $\mu_t = \mu$ as a function of an inherited $\theta_t = \theta$.

Dynamics under the Ramsey plan are confined to $\theta \in [\theta_\infty^R, \theta_0^R]$.

The blue and green lines intersect each other and the 45-degree line at $\theta = \theta_\infty^R$.

Notice that for $\theta \in (\theta_\infty^R, \theta_0^R]$

- $\theta' < \theta$ because the blue line is below the 45-degree line
- $\mu > \theta$ because the green line is above the 45-degree line

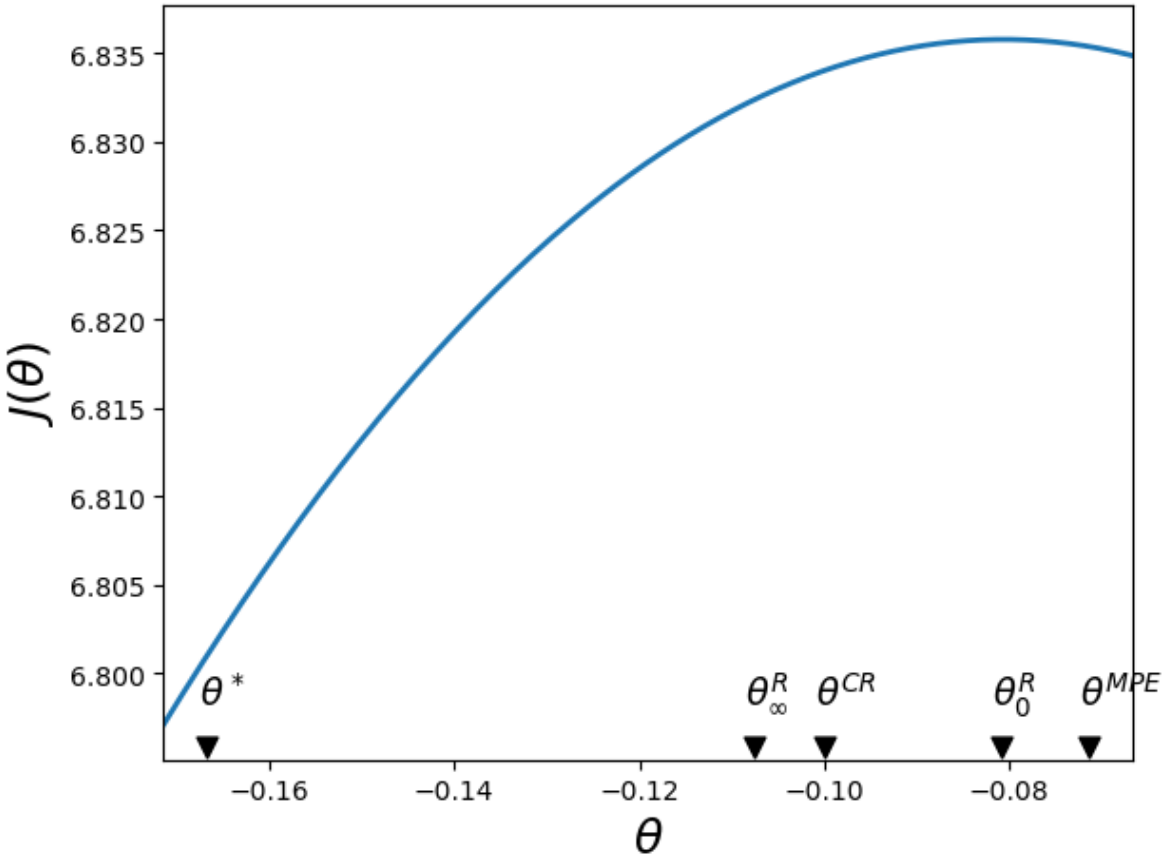
It follows that under the Ramsey plan $\{\theta_t\}$ and $\{\mu_t\}$ both converge monotonically from above to θ_∞^R.

The next code plots the Ramsey planner's value function $J(\theta)$.

We know that $J(\theta)$ is maximized at θ_0^R, the best time 0 promised inflation rate.

The figure also plots the limiting value θ_∞^R, the limiting value of promised inflation rate θ_t under the Ramsey plan as $t \to +\infty$.

The figure also indicates an MPE inflation rate θ^{MPE}, the inflation θ^{CR} under a Ramsey plan constrained to a constant money creation rate, and a bliss inflation θ^*.



In the above graph, notice that $\theta^* < \theta_\infty^R < \theta^{CR} < \theta_0^R < \theta^{MPE}$:

- $\theta_0^R < \theta^{MPE}$: the initial Ramsey inflation rate exceeds the MPE inflation rate
- $\theta_\infty^R < \theta^{CR} < \theta_0^R$: the initial Ramsey deflation rate, and the associated tax distortion cost $c\mu_0^2$ is less than the limiting Ramsey inflation rate θ_∞^R and the associated tax distortion cost μ_∞^2
- $\theta^* < \theta_\infty^R$: the limiting Ramsey inflation rate exceeds the bliss level of inflation

In some subsequent calculations, we'll use our Python code to study how gaps between these outcome vary depending on parameters such as the cost parameter c and the discount factor β.

43.15 Ramsey Planner's Value Function

The next code plots the Ramsey Planner's value function $J(\theta)$ as well as the value function of a constrained Ramsey planner who must choose a constant μ.

A time-invariant μ implies a time-invariant θ, we take the liberty of labeling this value function $V(\theta)$.

We'll use the code to plot $J(\theta)$ and $V(\theta)$ for several values of the discount factor β and the cost parameter c that multiplies μ_t^2 in the Ramsey planner's one-period payoff function.

In all of the graphs below, we disarm the Proposition 1 equivalence results by setting $c > 0$.

The graphs reveal interesting relationships among θ's associated with various timing protocols:

- $J(\theta) \geq V(\theta)$
- $J(\theta_\infty^R) = V(\theta_\infty^R)$

Before doing anything else, let's write code to verify our claim that $J(\theta_\infty^R) = V(\theta_\infty^R)$.

Here is the code.

```
θ_inf = clq.θ_series[1, -1]
np.allclose(clq.J_θ(θ_inf),
            clq.V_θ(θ_inf))
```

```
True
```

So we have verified our claim that $J(\theta_\infty^R) = V(\theta_\infty^R)$.

Since $J(\theta_\infty^R) = V(\theta_\infty^R)$ occurs at a tangency point at which $J(\theta)$ is increasing in θ, it follows that

$$V(\theta_\infty^R) \leq J(\theta^{CR}) \tag{43.33}$$

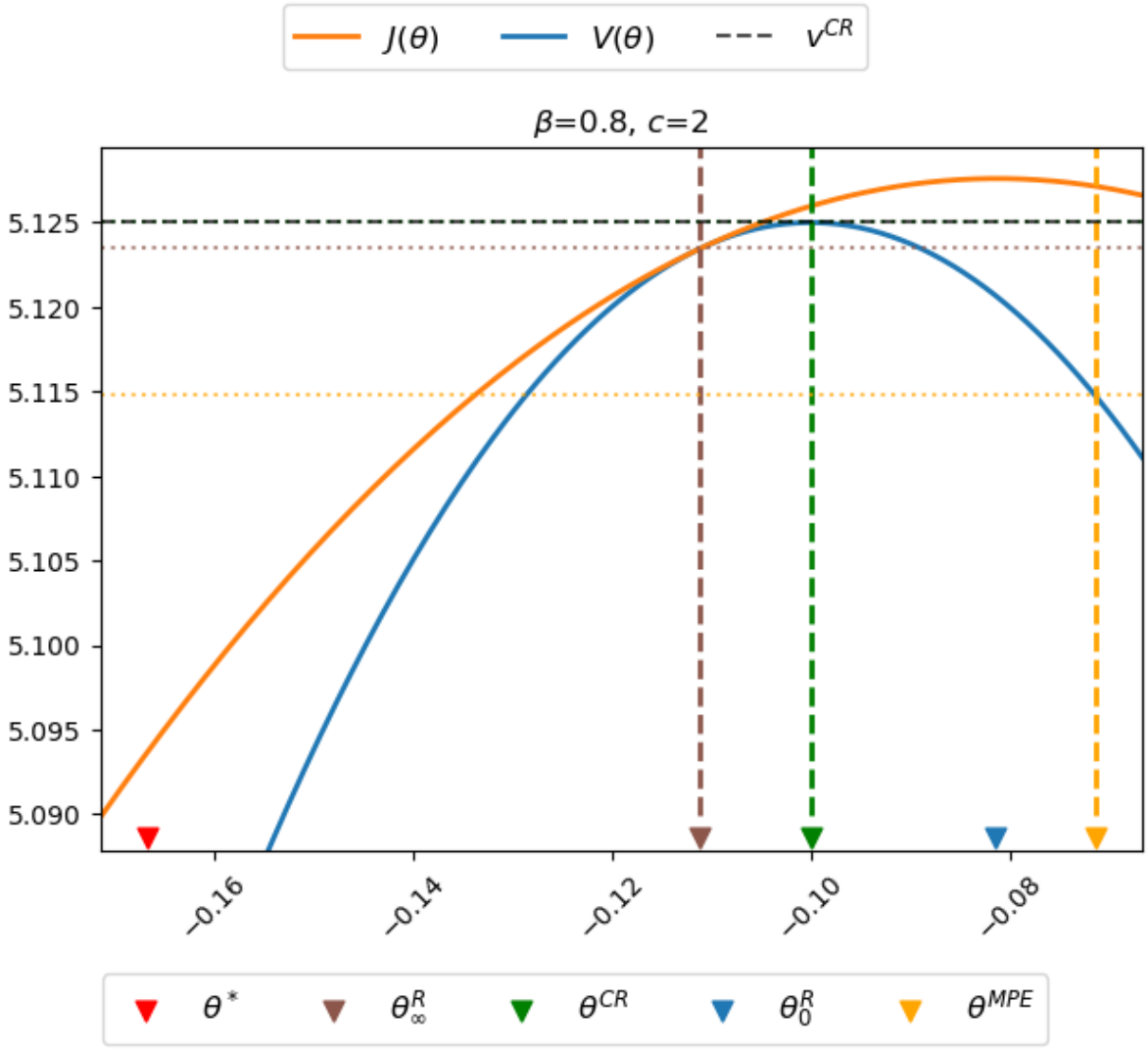
with strict inequality when $c > 0$.

Thus, the value of the plan that sets the money growth rate $\mu_t = \theta_\infty^R$ for all $t \geq 0$ is worse than the value attained by a Ramsey planner who is constrained to set a constant μ_t.

Now let's write some code to plot outcomes under our three timing protocols.

For some default parameter values, the next figure plots the Ramsey planner's continuation value function $J(\theta)$ (orange curve) and the restricted-to-constant-μ Ramsey planner's value function $V(\theta)$ (blue curve).

The figure uses colored arrows to indicate locations of $\theta^*, \theta_\infty^R, \theta^{CR}, \theta_0^R$, and θ^{MPE}, ordered as they are from left to right, on the θ axis.

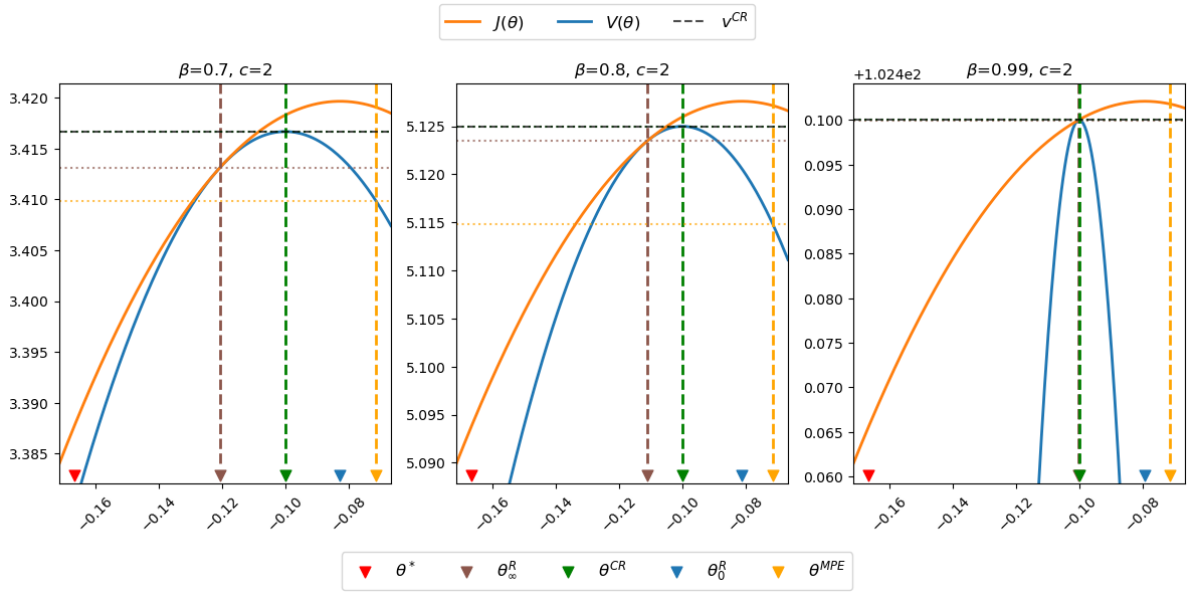


In the above figure, notice that

- the orange J value function lies above the blue V value function except at $\theta = \theta^R_\infty$
- the maximizer θ^R_0 of $J(\theta)$ occurs at the top of the orange curve
- the maximizer θ^{CR} of $V(\theta)$ occurs at the top of the blue curve
- the "timeless perspective" inflation and money creation rate θ^R_∞ occurs where $J(\theta)$ is tangent to $V(\theta)$
- the Markov perfect inflation and money creation rate θ^{MPE} exceeds θ^R_0.
- the value $V(\theta^{MPE})$ of the Markov perfect rate of money creation rate θ^{MPE} is less than the value $V(\theta^R_\infty)$ of the worst continuation Ramsey plan
- the continuation value $J(\theta^{MPE})$ of the Markov perfect rate of money creation rate θ^{MPE} is greater than the value $V(\theta^R_\infty)$ and of the continuation value $J(\theta^R_\infty)$ of the worst continuation Ramsey plan

43.16 Perturbing Model Parameters

Now let's present some graphs that teach us how outcomes change when we assume different values of β



The horizontal dotted lines indicate values $V(\mu^R_\infty), V(\mu^{CR}), V(\mu^{MPE})$ of time-invariant money growth rates μ^R_∞, μ^{CR} and μ^{MPE}, respectfully.

Notice how $J(\theta)$ and $V(\theta)$ are tangent and increasing at $\theta = \theta^R_\infty$, which implies that $\theta^{CR} > \theta^R_\infty$ and $J(\theta^{CR}) > J(\theta^R_\infty)$.

Notice how changes in β alter θ^R_∞ and θ^R_0 but neither θ^*, θ^{CR}, nor θ^{MPE}, in accord with formulas (43.8), (43.26), and (43.31), which imply that

$$\theta^{CR} = -\frac{\alpha u_1}{\alpha^2 u_2 + c}$$
$$\theta^{MPE} = -\frac{\alpha u_1}{\alpha^2 u_2 + (1+\alpha)c}$$
$$\theta^* = -\frac{u_1}{u_2 \alpha}$$

The following table summarizes some outcomes.

	$\beta = 0.7, c = 2$	$\beta = 0.8, c = 2$	$\beta = 0.99, c = 2$
θ^*	-0.167	-0.167	-0.167
θ^R_∞	-0.121	-0.111	-0.100
θ^{CR}	-0.100	-0.100	-0.100
θ^R_0	-0.083	-0.081	-0.079
θ^{MPE}	-0.071	-0.071	-0.071

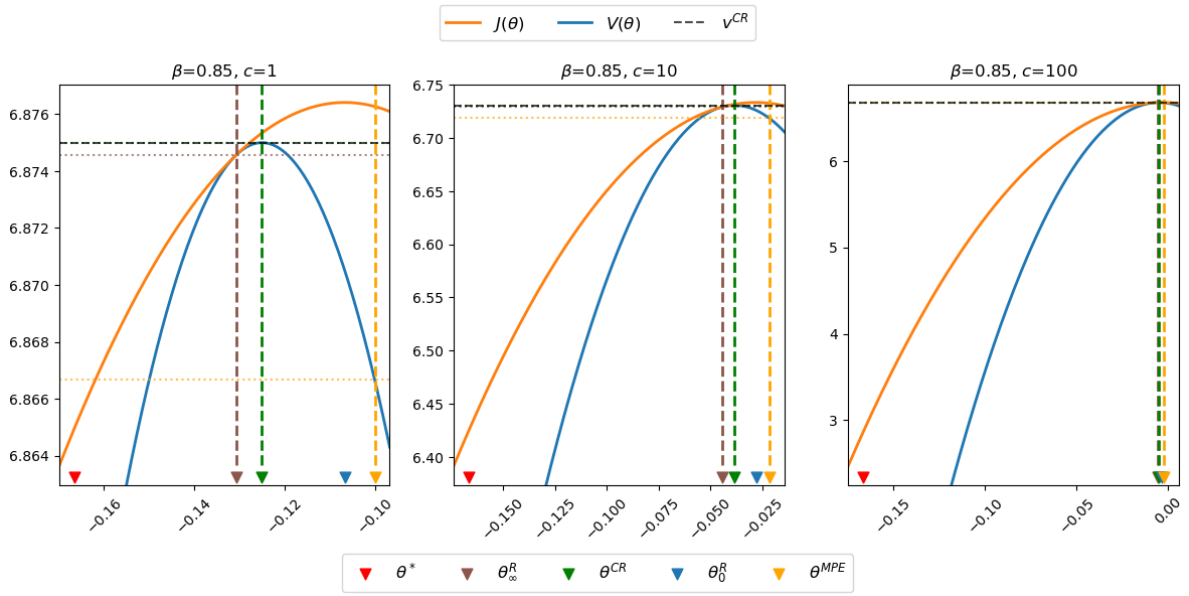
But let's see what happens when we change c.

```
# Increase c to 100
fig, axes = plt.subplots(1, 3, figsize=(12, 5))
c_values = [1, 10, 100]
```

(continues on next page)

(continued from previous page)

```
clqs = [ChangLQ(β=0.85, c=c) for c in c_values]
plt_clqs(clqs, axes)
```




```
generate_table(clqs, dig=4)
```

	$\beta = 0.85, c = 1$	$\beta = 0.85, c = 10$	$\beta = 0.85, c = 100$
θ^*	-0.167	-0.167	-0.167
θ^R_∞	-0.131	-0.044	-0.006
θ^{CR}	-0.125	-0.038	-0.005
θ^R_0	-0.107	-0.028	-0.003
θ^{MPE}	-0.100	-0.022	-0.003

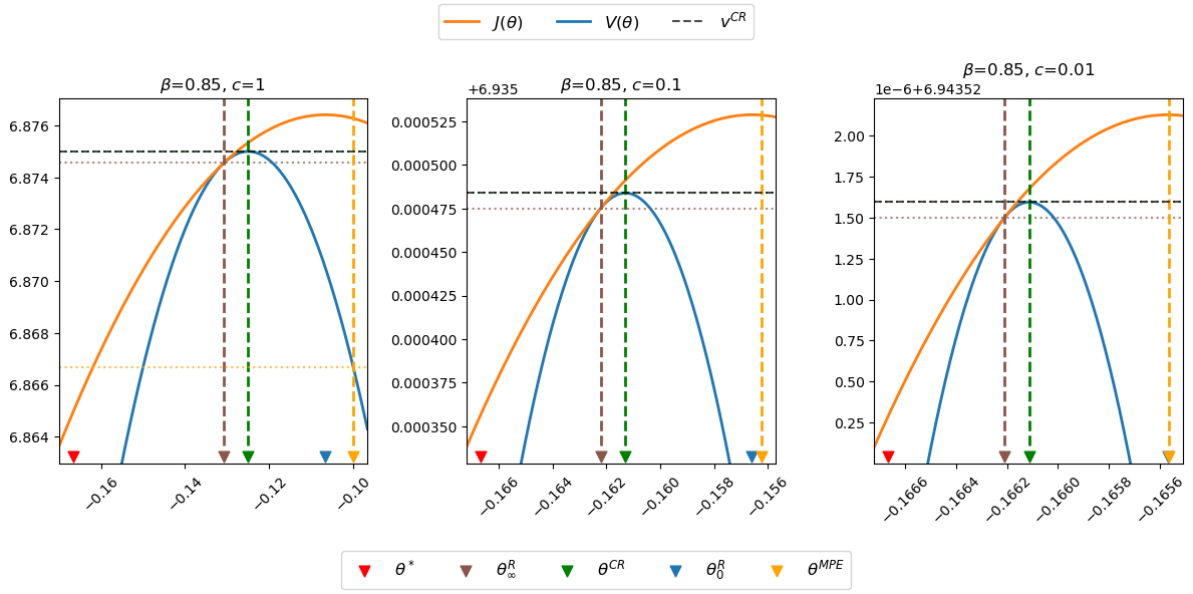
The above table and figures show how changes in c alter θ^R_∞ and θ^R_0 as well as θ^{CR} and θ^{MPE}, but not θ^*, again in accord with formulas (43.8), (43.26), and (43.31).

Notice that as c gets larger and larger, $\theta^R_\infty, \theta^R_0$ and θ^{CR} all converge to θ^{MPE}.

Now let's watch what happens when we drive c toward zero.

```
# Decrease c towards 0
fig, axes = plt.subplots(1, 3, figsize=(12, 5))
c_limits = [1, 0.1, 0.01]

clqs = [ChangLQ(β=0.85, c=c) for c in c_limits]
plt_clqs(clqs, axes)
```



The above graphs indicate that as c approaches zero, $\theta^R_\infty, \theta^R_0, \theta^{CR}$, and θ^{MPE} all approach θ^*.

This makes sense, because it was by adding costs of distorting taxes that Calvo [Calvo, 1978] drove a wedge between Friedman's optimal deflation rate and the inflation rates chosen by a Ramsey planner.

The following code plots sequences $\vec{\mu}$ and $\vec{\theta}$ prescribed by a Ramsey plan as well as the constant levels μ^{CR} and μ^{MPE}.

The following graphs report values for the value function parameters g_0, g_1, g_2, and the Ramsey policy function parameters b_0, b_1, d_0, d_1 associated with the indicated parameter pair β, c.

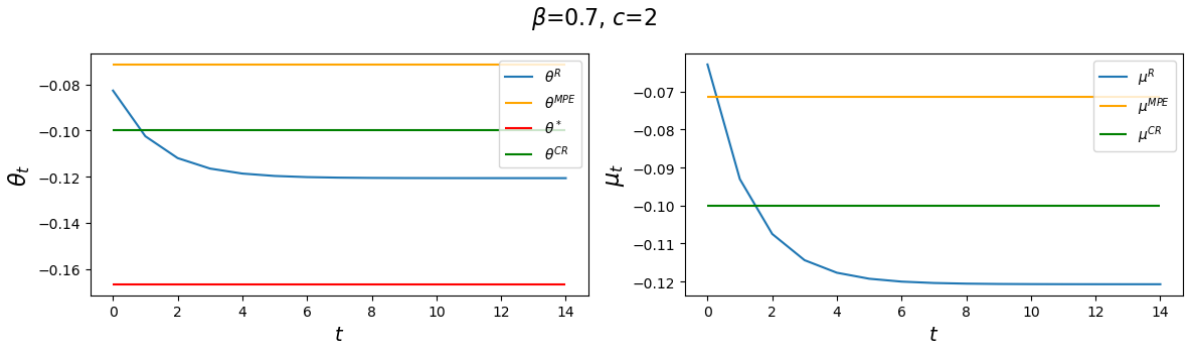
We'll vary β while keeping a small c.

After that we'll study consequences of raising c.

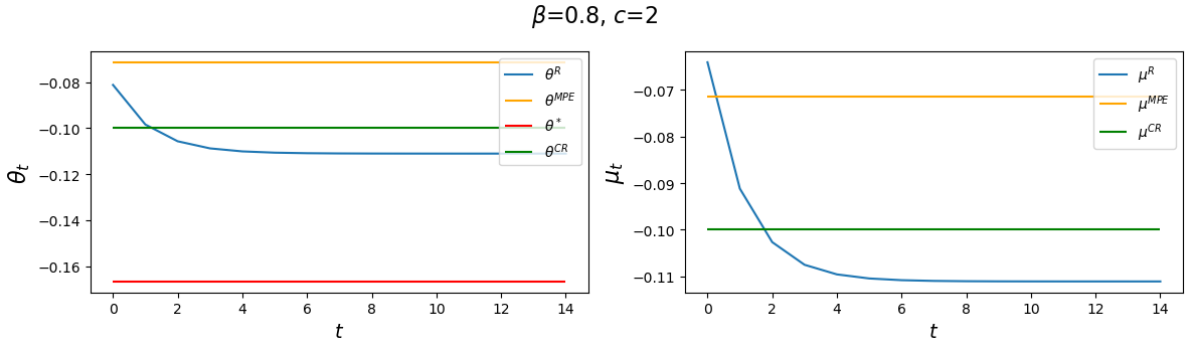
We'll watch how the decay rate d_1 governing the dynamics of θ^R_t is affected by alterations in the parameters β, c.

```
for β in β_values:
    clq = ChangLQ(β=β, c=2)
    generate_param_table(clq)
    plot_ramsey_MPE(clq)
```

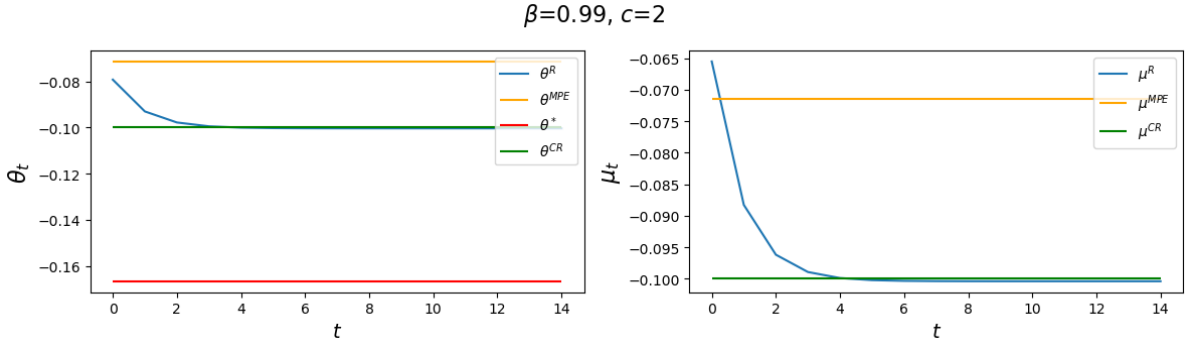
	g_0	g_1	g_2	b_0	b_1	d_0	d_1
$\beta = 0.7, c = 2$	3.39	−0.75	−4.54	−0.06	−1.52	−0.06	0.48



	g_0	g_1	g_2	b_0	b_1	d_0	d_1
$\beta = 0.8, c = 2$	5.1	−0.76	−4.65	−0.06	−1.58	−0.06	0.42



	g_0	g_1	g_2	b_0	b_1	d_0	d_1
$\beta = 0.99, c = 2$	102.47	−0.76	−4.81	−0.07	−1.65	−0.07	0.35

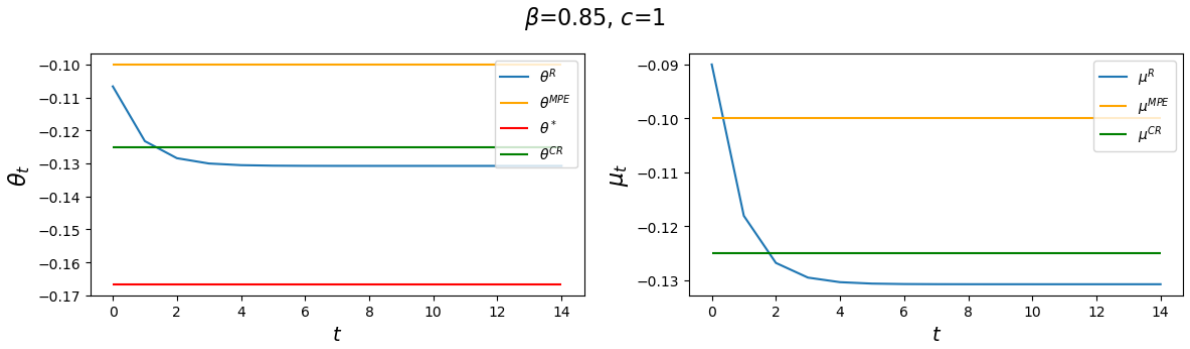


Notice how d_1 changes as we raise the discount factor parameter β.

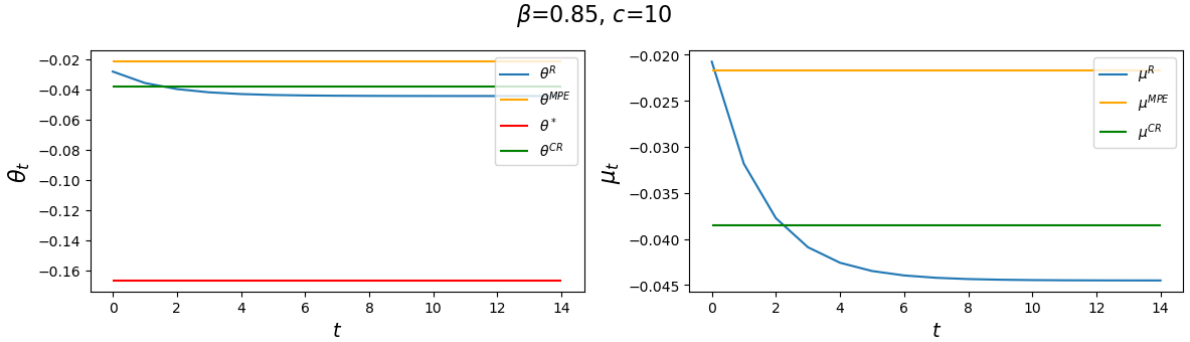
Now let's study how increasing c affects $\vec{\theta}, \vec{\mu}$ outcomes.

```
# Increase c to 100
for c in c_values:
    clq = ChangLQ(β=0.85, c=c)
    generate_param_table(clq)
    plot_ramsey_MPE(clq)
```

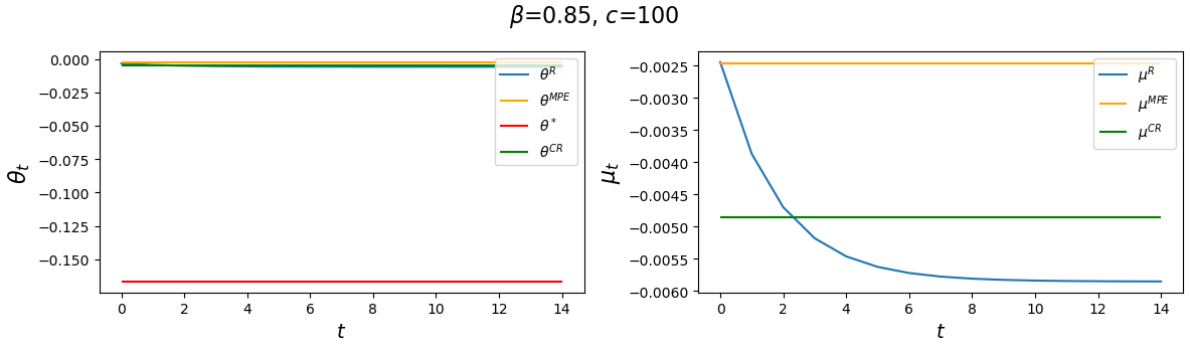
	g_0	g_1	g_2	b_0	b_1	d_0	d_1
$\beta = 0.85, c = 1$	6.84	−0.68	−3.19	−0.09	−1.69	−0.09	0.31



	g_0	g_1	g_2	b_0	b_1	d_0	d_1
$\beta = 0.85, c = 10$	6.72	−0.92	−16.16	−0.02	−1.47	−0.02	0.53



	g_0	g_1	g_2	b_0	b_1	d_0	d_1
$\beta = 0.85, c = 100$	6.67	−0.99	−143.29	−0.0	−1.42	−0.0	0.58

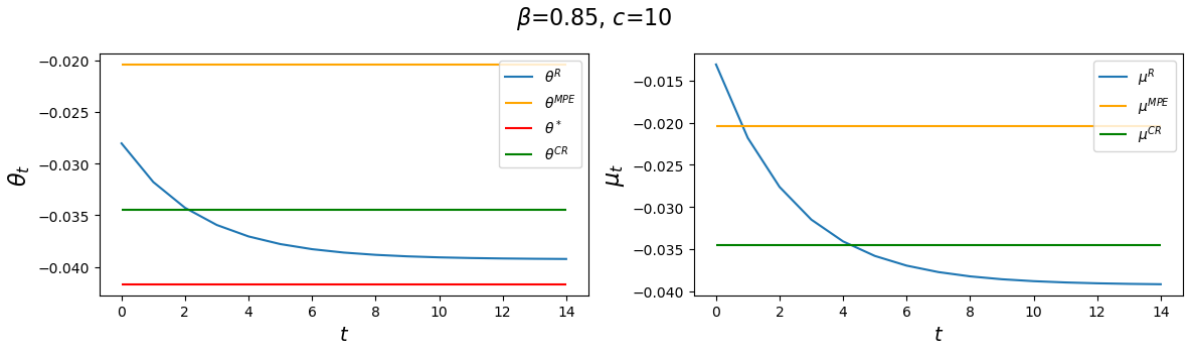


Evidently, increasing c causes the decay factor d_1 to increase.

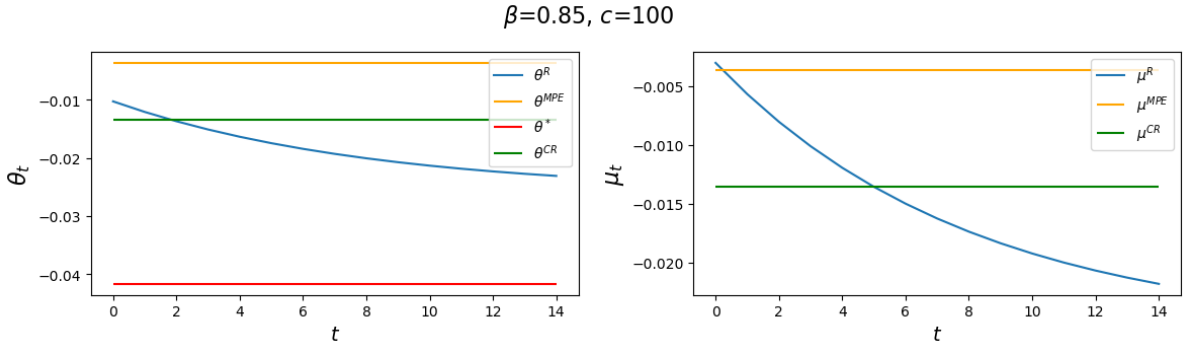
Next, let's look at consequences of increasing the demand for real balances parameter α from its default value $\alpha = 1$ to $\alpha = 4$.

```
# Increase c to 100
for c in [10, 100]:
    clq = ChangLQ(α=4, β=0.85, c=c)
    generate_param_table(clq)
    plot_ramsey_MPE(clq)
```

	g_0	g_1	g_2	b_0	b_1	d_0	d_1
$\beta = 0.85, c = 10$	6.84	−4.62	−82.32	−0.05	−2.33	−0.01	0.67



	g_0	g_1	g_2	b_0	b_1	d_0	d_1
$\beta = 0.85, c = 100$	6.74	−8.03	−390.65	−0.01	−1.47	−0.0	0.88



The above panels for an $\alpha = 4$ setting indicate that α and c affect outcomes in interesting ways.

We leave it to the reader to explore consequences of other constellations of parameter values.

43.16.1 Implausibility of Ramsey Plan

Many economists regard a time inconsistent plan as implausible because they question the plausibility of timing protocol in which a plan for setting a sequence of policy variables is chosen once-and-for-all at time 0.

For that reason, the Markov perfect equilibrium concept attracts many economists.

- A Markov perfect equilibrium plan is constructed to insure that a sequence of government policymakers who choose sequentially do not want to deviate from it.

43.16.2 Ramsey Plan Strikes Back

Research by Abreu [Abreu, 1988], Chari and Kehoe [Chari and Kehoe, 1990] [Stokey, 1989], and Stokey [Stokey, 1991] described conditions under which a Ramsey plan can be rescued from the complaint that it is not credible.

They accomplished this by expanding the description of a plan to include expectations about *adverse consequences* of deviating from it that can serve to deter deviations.

We turn to such theories in this quantecon lecture *Sustainable Plans for a Calvo Model*.

CHAPTER

FORTYFOUR

SUSTAINABLE PLANS FOR A CALVO MODEL

44.1 Overview

This is a sequel to this quantecon lecture *Time Inconsistency of Ramsey Plans*.

That lecture studied a linear-quadratic version of a model that Guillermo Calvo [Calvo, 1978] used to study the **time inconsistency** of the optimal government plan that emerges when a **Stackelberg** government (a.k.a. a **Ramsey planner**) at time 0 once and for all chooses a sequence $\vec{\mu} = \{\mu_t\}_{t=0}^{\infty}$ of gross rates of growth in the supply of money.

A consequence of that choice is a (rational expectations equilibrium) sequence $\vec{\theta} = \{\theta_t\}_{t=0}^{\infty}$ of gross rates of increase in the price level that we call inflation rates.

[Calvo, 1978] showed that a Ramsey plan would not emerge from alternative timing protocols and associated supplementary assumptions about what government authorities who set μ_t at time t believe how future government authorities who set μ_{t+j} for $j > 0$ will respond to their decisions.

In this lecture, we explore another set of assumptions about what government authorities who set μ_t at time t believe how future government authorities who set μ_{t+j} for $j > 0$ will respond to their decisions.

We shall assume that there is sequence of separate policymakers; a time t policymaker chooses only μ_t, but now believes that its choice of μ_t shapes the representative agent's beliefs about future rates of money creation and inflation, and through them, future government actions.

This timing protocol and belief structure leads to a model of a **credible government policy**, also known as a **sustainable plan**.

In quantecon lecture *Time Inconsistency of Ramsey Plans* we used ideas from papers by Cagan [Cagan, 1956], Calvo [Calvo, 1978], and Chang [Chang, 1998].

In addition to those ideas, we'll also use ideas from Abreu [Abreu, 1988], Stokey [Stokey, 1989], [Stokey, 1991], and Chari and Kehoe [Chari and Kehoe, 1990] to study outcomes under our timing protocol.

44.2 Model Components

We'll start with a brief review of the setup.

There is no uncertainty.

Let

- p_t be the log of the price level
- m_t be the log of nominal money balances
- $\theta_t = p_{t+1} - p_t$ be the net rate of inflation between t and $t+1$

- $\mu_t = m_{t+1} - m_t$ be the net rate of growth of nominal balances

The demand for real balances is governed by a perfect foresight version of a Cagan [Cagan, 1956] demand function for real balances:

$$m_t - p_t = -\alpha(p_{t+1} - p_t)\ ,\ \alpha > 0 \tag{44.1}$$

for $t \geq 0$.

Equation (44.1) asserts that the demand for real balances is inversely related to the public's expected rate of inflation, which equals the actual rate of inflation because there is no uncertainty here.

(When there is no uncertainty, an assumption of **rational expectations** that becomes equivalent to **perfect foresight**).

Subtracting the demand function (44.1) at time t from the demand function at $t+1$ gives:

$$\mu_t - \theta_t = -\alpha\theta_{t+1} + \alpha\theta_t$$

or

$$\theta_t = \frac{\alpha}{1+\alpha}\theta_{t+1} + \frac{1}{1+\alpha}\mu_t \tag{44.2}$$

Because $\alpha > 0$, $0 < \frac{\alpha}{1+\alpha} < 1$.

Definition: For scalar b_t, let L^2 be the space of sequences $\{b_t\}_{t=0}^{\infty}$ satisfying

$$\sum_{t=0}^{\infty} b_t^2 < +\infty$$

We say that a sequence that belongs to L^2 is **square summable**.

When we assume that the sequence $\vec{\mu} = \{\mu_t\}_{t=0}^{\infty}$ is square summable and we require that the sequence $\vec{\theta} = \{\theta_t\}_{t=0}^{\infty}$ is square summable, the linear difference equation (44.2) can be solved forward to get:

$$\theta_t = \frac{1}{1+\alpha}\sum_{j=0}^{\infty}\left(\frac{\alpha}{1+\alpha}\right)^j \mu_{t+j} \tag{44.3}$$

Insight: In the spirit of Chang [Chang, 1998], equations (44.1) and (44.3) show that θ_t intermediates how choices of $\mu_{t+j},\ j = 0, 1, \ldots$ impinge on time t real balances $m_t - p_t = -\alpha\theta_t$.

An equivalence class of continuation money growth sequences $\{\mu_{t+j}\}_{j=0}^{\infty}$ deliver the same θ_t.

That future rates of money creation influence earlier rates of inflation makes timing protocols matter for modeling optimal government policies.

Quantecon lecture *Time Inconsistency of Ramsey Plans* used this insight to simplify analysis of alternative government policy problems.

44.3 Another Timing Protocol

The Quantecon lecture *Time Inconsistency of Ramsey Plans* considered three models of government policy making that differ in

- *what* a policymaker chooses, either a sequence $\vec{\mu}$ or just μ_t in a single period t.
- *when* a policymaker chooses, either once and for all at time 0, or at some time or times $t \geq 0$.
- what a policymaker *assumes* about how its choice of μ_t affects the representative agent's expectations about inflation rates.

In this lecture, there is a sequence of policymakers, each of whom sets μ_t at t only.

To set the stage, recall that in a **Markov perfect equilibrium**

- a time t policymaker cares only about v_t and ignores effects that its choice of μ_t has on v_s at dates $s = 0, 1, \ldots, t-1$.

In particular, in a Markov perfect equilibrium, there is a sequence indexed by $t = 0, 1, 2, \ldots$ of separate policymakers; a time t policymaker chooses μ_t only and forecasts that future government decisions are unaffected by its choice.

By way of contrast, in this lecture there is sequence of distinct policymakers; a time t policymaker chooses only μ_t, but now believes that its choice of μ_t shapes the representative agent's beliefs about future rates of money creation and inflation, and through them, future government actions.

This timing protocol and belief structure leads to a model of a **credible government policy** also known as a **sustainable plan**

The relationship between outcomes under a (Ramsey) timing protocol and the timing protocol and belief structure in this lecture is the subject of a literature on **sustainable** or **credible** public policies created by Abreu [Abreu, 1988], [Chari and Kehoe, 1990] [Stokey, 1989], and Stokey [Stokey, 1991].

They discovered conditions under which a Ramsey plan can be rescued from the complaint that it is not credible.

They accomplished this by expanding the description of a plan to include expectations about *adverse consequences* of deviating from it that can serve to deter deviations.

In this version of our model

- the government does not set $\{\mu_t\}_{t=0}^{\infty}$ once and for all at $t = 0$
- instead it sets μ_t at time t
- the representative agent's forecasts of $\{\mu_{t+j+1}, \theta_{t+j+1}\}_{j=0}^{\infty}$ respond to whether the government at t *confirms* or *disappoints* its forecasts of μ_t brought into period t from period $t-1$.
- the government at each time t understands how the representative agent's forecasts will respond to its choice of μ_t.
- at each t, the government chooses μ_t to maximize a continuation discounted utility.

44.3.1 Government Decisions

$\vec{\mu}$ is chosen by a sequence of government decision makers, one for each $t \geq 0$.

The time t decision maker chooses μ_t.

We assume the following within-period and between-period timing protocol for each $t \geq 0$:

- at time $t-1$, private agents expect that the government will set $\mu_t = \tilde{\mu}_t$, and more generally that it will set $\mu_{t+j} = \tilde{\mu}_{t+j}$ for all $j \geq 0$.
- The forecasts $\{\tilde{\mu}_{t+j}\}_{j \geq 0}$ determine a $\theta_t = \tilde{\theta}_t$ and an associated log of real balances $m_t - p_t = -\alpha\tilde{\theta}_t$ at t.
- Given those expectations and an associated $\theta_t = \tilde{\theta}_t$, at t a government is free to set $\mu_t \in \mathbf{R}$.
- If the government at t *confirms* the representative agent's expectations by setting $\mu_t = \tilde{\mu}_t$ at time t, private agents expect the continuation government policy $\{\tilde{\mu}_{t+j+1}\}_{j=0}^{\infty}$ and therefore bring expectation $\tilde{\theta}_{t+1}$ into period $t+1$.
- If the government at t *disappoints* private agents by setting $\mu_t \neq \tilde{\mu}_t$, private agents expect $\{\mu_j^A\}_{j=0}^{\infty}$ as the continuation policy for $t+1$, i.e., $\{\mu_{t+j+1}\} = \{\mu_j^A\}_{j=0}^{\infty}$ and therefore expect an associated θ_0^A for $t+1$. Here $\vec{\mu}^A = \{\mu_j^A\}_{j=0}^{\infty}$ is an alternative government plan to be described below.

44.3.2 Temptation to Deviate from Plan

The government's one-period return function $s(\theta, \mu)$ described in equation (43.10) in quantecon lecture [Calvo, 1978] has the property that for all θ

$$s(\theta, 0) \geq s(\theta, \mu)$$

This inequality implies that whenever the policy calls for the government to set $\mu \neq 0$, the government could raise its one-period payoff by setting $\mu = 0$.

Disappointing private sector expectations in that way would increase the government's *current* payoff but would have adverse consequences for *subsequent* government payoffs because the private sector would alter its expectations about future settings of μ.

The *temporary* gain constitutes the government's temptation to deviate from a plan.

If the government at t is to resist the temptation to raise its current payoff, it is only because it forecasts adverse consequences that its setting of μ_t would bring for continuation government payoffs via alterations in the private sector's expectations.

44.4 Sustainable or Credible Plan

We call a plan $\vec{\mu}$ **sustainable** or **credible** if at each $t \geq 0$ the government chooses to confirm private agents' prior expectation of its setting for μ_t.

The government will choose to confirm prior expectations only if the long-term *loss* from disappointing private sector expectations – coming from the government's understanding of the way the private sector adjusts its expectations in response to having its prior expectations at t disappointed – outweigh the short-term *gain* from disappointing those expectations.

The theory of sustainable or credible plans assumes throughout that private sector expectations about what future governments will do are based on the assumption that governments at times $t \geq 0$ always act to maximize the continuation discounted utilities that describe those governments' purposes.

This aspect of the theory means that credible plans always come in *pairs*:

- a credible (continuation) plan to be followed if the government at t *confirms* private sector expectations
- a credible plan to be followed if the government at t *disappoints* private sector expectations

That credible plans come in pairs threaten to bring an explosion of plans to keep track of

- each credible plan itself consists of two credible plans
- therefore, the number of plans underlying one plan is unbounded

But Dilip Abreu showed how to render manageable the number of plans that must be kept track of.

The key is an object called a **self-enforcing** plan.

We'll proceed to compute one.

In addition to what's in Anaconda, this lecture will use the following libraries:

```
!pip install --upgrade quantecon
```

We'll start with some imports:

```
import numpy as np
from quantecon import LQ
import matplotlib.pyplot as plt
import pandas as pd
```

44.4.1 Abreu's Self-Enforcing Plan

A plan $\vec{\mu}^A$ (here the superscipt A is for Abreu) is said to be **self-enforcing** if

- the consequence of disappointing the representative agent's expectations at time j is to *restart* plan $\vec{\mu}^A$ at time $j+1$
- the consequence of restarting the plan is sufficiently adverse that it forever deters all deviations from the plan

More precisely, a government plan $\vec{\mu}^A$ with equilibrium inflation sequence $\vec{\theta}^A$ is **self-enforcing** if

$$\begin{aligned} v_j^A &= s(\theta_j^A, \mu_j^A) + \beta v_{j+1}^A \\ &\geq s(\theta_j^A, 0) + \beta v_0^A \equiv v_j^{A,D}, \quad j \geq 0 \end{aligned} \tag{44.4}$$

(Here it is useful to recall that setting $\mu = 0$ is the maximizing choice for the government's one-period return function)

The first line tells the consequences of confirming the representative agent's expectations by following the plan, while the second line tells the consequences of disappointing the representative agent's expectations by deviating from the plan.

A consequence of the inequality stated in the definition is that a self-enforcing plan is credible.

Self-enforcing plans can be used to construct other credible plans, including ones with better values.

Thus, where $\vec{v}^A$ is the value associated with a self-enforcing plan $\vec{\mu}^A$, a sufficient condition for another plan $\vec{\mu}$ associated with inflation $\vec{\theta}$ and value $\vec{v}$ to be **credible** is that

$$\begin{aligned} v_j &= s(\theta_j, \mu_j) + \beta v_{j+1} \\ &\geq s(\theta_j, 0) + \beta v_0^A \quad \forall j \geq 0 \end{aligned} \tag{44.5}$$

For this condition to be satisfied it is necessary and sufficient that

$$s(\theta_j, 0) - s(\theta_j, \mu_j) < \beta(v_{j+1} - v_0^A)$$

The left side of the above inequality is the government's *gain* from deviating from the plan, while the right side is the government's *loss* from deviating from the plan.

A government never wants to deviate from a credible plan.

Abreu taught us that key step in constructing a credible plan is first constructing a self-enforcing plan that has a low time 0 value.

The idea is to use the self-enforcing plan as a continuation plan whenever the government's choice at time t fails to confirm private agents' expectation.

We shall use a construction featured in Abreu ([Abreu, 1988]) to construct a self-enforcing plan with low time 0 value.

44.4.2 Abreu's Carrot-Stick Plan

[Abreu, 1988] invented a way to create a self-enforcing plan with a low initial value.

Imitating his idea, we can construct a self-enforcing plan $\vec{\mu}$ with a low time 0 value to the government by insisting that future government decision makers set μ_t to a value yielding low one-period utilities to the household for a long time, after which government decisions thereafter yield high one-period utilities.

- Low one-period utilities early are a *stick*
- High one-period utilities later are a *carrot*

Consider a candidate plan $\vec{\mu}^A$ that sets $\mu_t^A = \bar{\mu}$ (a high positive number) for T_A periods, and then reverts to the Ramsey plan.

Denote this sequence by $\{\mu_t^A\}_{t=0}^{\infty}$.

The sequence of inflation rates implied by this plan, $\{\theta_t^A\}_{t=0}^{\infty}$, can be calculated using:

$$\theta_t^A = \frac{1}{1+\alpha} \sum_{j=0}^{\infty} \left(\frac{\alpha}{1+\alpha} \right)^j \mu_{t+j}^A$$

The value of $\{\theta_t^A, \mu_t^A\}_{t=0}^{\infty}$ at time 0 is

$$v_0^A = \sum_{t=0}^{T_A - 1} \beta^t s(\theta_t^A, \mu_t^A) + \beta^{T_A} J(\theta_0^R)$$

For an appropriate T_A, this plan can be verified to be self-enforcing and therefore credible.

From quantecon lecture *Time Inconsistency of Ramsey Plans*, we'll again bring in the Python class ChangLQ that constructs equilibria under timing protocols studied in that lecture.

```
class ChangLQ:
    """
    Class to solve LQ Chang model
    """
    def __init__(self, β, c, α=1, u0=1, u1=0.5, u2=3, T=1000, θ_n=200):
        # Record parameters
        self.α, self.u0, self.u1, self.u2 = α, u0, u1, u2
        self.β, self.c, self.T, self.θ_n = β, c, T, θ_n

        self.setup_LQ_matrices()
        self.solve_LQ_problem()
        self.compute_policy_functions()
        self.simulate_ramsey_plan()
        self.compute_θ_range()
        self.compute_value_and_policy()

    def setup_LQ_matrices(self):
        # LQ Matrices
        self.R = -np.array([[self.u0, -self.u1 * self.α / 2],
                            [-self.u1 * self.α / 2,
                             -self.u2 * self.α**2 / 2]])
        self.Q = -np.array([[-self.c / 2]])
        self.A = np.array([[1, 0], [0, (1 + self.α) / self.α]])
        self.B = np.array([[0], [-1 / self.α]])

    def solve_LQ_problem(self):
```

(continues on next page)

(continued from previous page)

```
        # Solve LQ Problem (Subproblem 1)
        lq = LQ(self.Q, self.R, self.A, self.B, beta=self.β)
        self.P, self.F, self.d = lq.stationary_values()

        # Compute g0, g1, and g2 (41.16)
        self.g0, self.g1, self.g2 = [-self.P[0, 0],
                                     -2 * self.P[1, 0], -self.P[1, 1]]

        # Compute b0 and b1 (41.17)
        [[self.b0, self.b1]] = self.F

        # Compute d0 and d1 (41.18)
        self.cl_mat = (self.A - self.B @ self.F)  # Closed loop matrix
        [[self.d0, self.d1]] = self.cl_mat[1:]

        # Solve Subproblem 2
        self.θ_R = -self.P[0, 1] / self.P[1, 1]

        # Find the bliss level of θ
        self.θ_B = -self.u1 / (self.u2 * self.α)

    def compute_policy_functions(self):
        # Solve the Markov Perfect Equilibrium
        self.μ_MPE = -self.u1 / ((1 + self.α) / self.α * self.c
                                 + self.α / (1 + self.α)
                                 * self.u2 + self.α**2
                                 / (1 + self.α) * self.u2)
        self.θ_MPE = self.μ_MPE
        self.μ_CR = -self.α * self.u1 / (self.u2 * self.α**2 + self.c)
        self.θ_CR = self.μ_CR

        # Calculate value under MPE and CR economy
        self.J_θ = lambda θ_array: - np.array([1, θ_array]) \
                                   @ self.P @ np.array([1, θ_array]).T
        self.V_θ = lambda θ: (self.u0 + self.u1 * (-self.α * θ)
                              - self.u2 / 2 * (-self.α * θ)**2
                              - self.c / 2 * θ**2) / (1 - self.β)

        self.J_MPE = self.V_θ(self.μ_MPE)
        self.J_CR = self.V_θ(self.μ_CR)

    def simulate_ramsey_plan(self):
        # Simulate Ramsey plan for large number of periods
        θ_series = np.vstack((np.ones((1, self.T)), np.zeros((1, self.T))))
        μ_series = np.zeros(self.T)
        J_series = np.zeros(self.T)
        θ_series[1, 0] = self.θ_R
        [μ_series[0]] = -self.F.dot(θ_series[:, 0])
        J_series[0] = self.J_θ(θ_series[1, 0])

        for i in range(1, self.T):
            θ_series[:, i] = self.cl_mat @ θ_series[:, i-1]
            [μ_series[i]] = -self.F @ θ_series[:, i]
            J_series[i] = self.J_θ(θ_series[1, i])

        self.J_series = J_series
```

(continues on next page)

(continued from previous page)

```
        self.μ_series = μ_series
        self.θ_series = θ_series

    def compute_θ_range(self):
        # Find the range of θ in Ramsey plan
        θ_LB = min(min(self.θ_series[1, :]), self.θ_B)
        θ_UB = max(max(self.θ_series[1, :]), self.θ_MPE)
        θ_range = θ_UB - θ_LB
        self.θ_LB = θ_LB - 0.05 * θ_range
        self.θ_UB = θ_UB + 0.05 * θ_range
        self.θ_range = θ_range

    def compute_value_and_policy(self):
        # Create the θ_space
        self.θ_space = np.linspace(self.θ_LB, self.θ_UB, 200)

        # Find value function and policy functions over range of θ
        self.J_space = np.array([self.J_θ(θ) for θ in self.θ_space])
        self.μ_space = -self.F @ np.vstack((np.ones(200), self.θ_space))
        x_prime = self.cl_mat @ np.vstack((np.ones(200), self.θ_space))
        self.θ_prime = x_prime[1, :]
        self.CR_space = np.array([self.V_θ(θ) for θ in self.θ_space])

        self.μ_space = self.μ_space[0, :]

        # Calculate J_range, J_LB, and J_UB
        self.J_range = np.ptp(self.J_space)
        self.J_LB = np.min(self.J_space) - 0.05 * self.J_range
        self.J_UB = np.max(self.J_space) + 0.05 * self.J_range
```

Let's create an instance of ChangLQ with the following parameters:

```
clq = ChangLQ(β=0.85, c=2)
```

44.4.3 Example of Self-Enforcing Plan

The following example implements an Abreu stick-and-carrot plan.

The government sets $\mu_t^A = 0.1$ for $t = 0, 1, \ldots, 9$ and then starts the **Ramsey plan**.

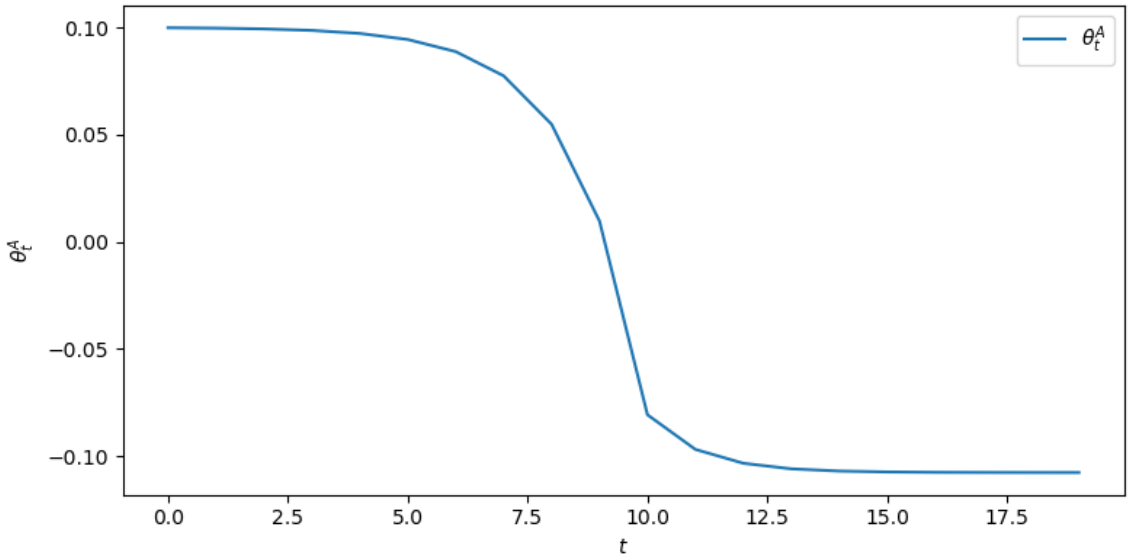
We have computed outcomes for this plan.

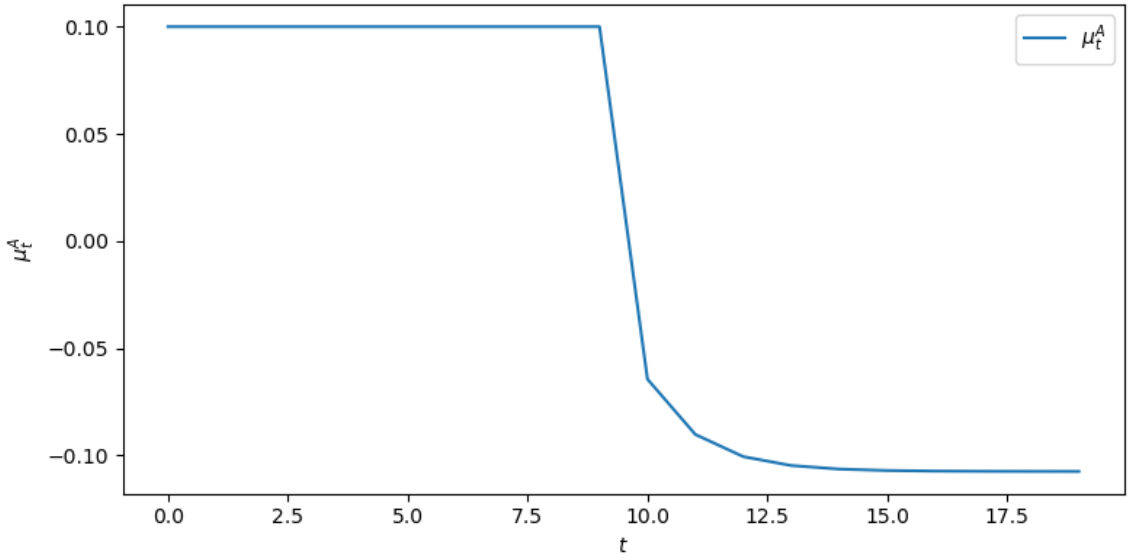
For this plan, we plot the θ^A, μ^A sequences as well as the implied v^A sequence.

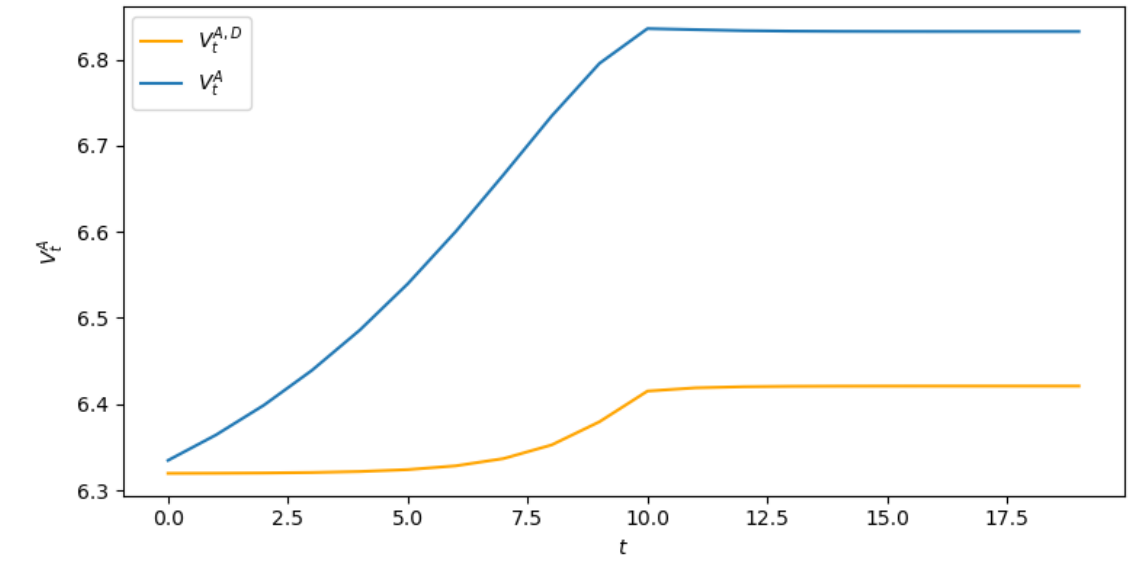
Notice that because the government sets money supply growth high for 10 periods, inflation starts high.

Inflation gradually slowly declines because people expect the government to lower the money growth rate after period 10.

From the 10th period onwards, the inflation rate θ_t^A associated with this **Abreu plan** starts the Ramsey plan from its beginning, i.e., $\theta_{t+10}^A = \theta_t^R \ \ \forall t \geq 0$.


0.10
0.05
0.00
−0.05
−0.10
θ_t^A
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
t
θ_t^A


0.10
0.05
0.00
−0.05
−0.10
μ_t^A
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
t
μ_t^A


6.8
6.7
6.6
6.5
6.4
6.3
V_t^A
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
t
$V_t^{A,D}$
V_t^A

To confirm that the plan $\vec{\mu}^A$ is **self-enforcing**, we plot an object that we call $V_t^{A,D}$, defined in the key inequality in the second line of equation (44.4) above.

$V_t^{A,D}$ is the value at t of deviating from the self-enforcing plan $\vec{\mu}^A$ by setting $\mu_t = 0$ and then restarting the plan at v_0^A at $t+1$:

$$v_t^{A,D} = s(\theta_j, 0) + \beta v_0^A$$

In the above graph $v_t^A > v_t^{A,D}$, which confirms that $\vec{\mu}^A$ is a self-enforcing plan.

We can also verify the inequalities required for $\vec{\mu}^A$ to be self-confirming numerically as follows

```
np.all(clq.V_A[0:20] > clq.V_dev[0:20])
```

```
True
```

Given that plan $\vec{\mu}^A$ is self-enforcing, we can check that the Ramsey plan $\vec{\mu}^R$ is credible by verifying that:

$$v_t^R \geq s(\theta_t^R, 0) + \beta v_0^A, \quad \forall t \geq 0$$

```
def check_ramsey(clq, T=1000):
    # Make sure Ramsey plan is sustainable
    R_dev = np.zeros(T)
    for t in range(T):
        R_dev[t] = (clq.u0 + clq.u1 * (-clq.θ_series[1, t])
                    - clq.u2 / 2 * (-clq.θ_series[1, t])**2) \
                    + clq.β * clq.V_A[0]

    return np.all(clq.J_series > R_dev)

check_ramsey(clq)
```

```
True
```

44.4.4 Recursive Representation of a Sustainable Plan

We can represent a sustainable plan recursively by taking the continuation value v_t as a state variable.

We form the following 3-tuple of functions:

$$\begin{aligned} \hat{\mu}_t &= \nu_\mu(v_t) \\ \theta_t &= \nu_\theta(v_t) \\ v_{t+1} &= \nu_v(v_t, \mu_t) \end{aligned} \tag{44.6}$$

In addition to these equations, we need an initial value v_0 to characterize a sustainable plan.

The first equation of (44.6) tells the recommended value of $\hat{\mu}_t$ as a function of the promised value v_t.

The second equation of (44.6) tells the inflation rate as a function of v_t.

The third equation of (44.6) updates the continuation value in a way that depends on whether the government at t confirms the representative agent's expectations by setting μ_t equal to the recommended value $\hat{\mu}_t$, or whether it disappoints those expectations.

44.5 Whose Plan is It?

A credible government plan $\vec{\mu}$ plays multiple roles.

- It is a sequence of actions chosen by the government.
- It is a sequence of the representative agent's forecasts of government actions.

Thus, $\vec{\mu}$ is both a government policy and a collection of the representative agent's forecasts of government policy.

Does the government *choose* policy actions or does it simply *confirm* prior private sector forecasts of those actions?

An argument in favor of the *government chooses* interpretation comes from noting that the theory of credible plans builds in a theory that the government each period chooses the action that it wants.

An argument in favor of the *simply confirm* interpretation is gathered from staring at the key inequality (44.5) that defines a credible policy.

We have also computed **credible plans** for a government or sequence of governments that choose sequentially.

These include

- a **self-enforcing** plan that gives a low initial value v_0.
- a better plan – possibly one that attains values associated with Ramsey plan – that is not self-enforcing.

CHAPTER

FORTYFIVE

OPTIMAL TAXATION WITH STATE-CONTINGENT DEBT

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install --upgrade quantecon
```

45.1 Overview

This lecture describes a celebrated model of optimal fiscal policy by Robert E. Lucas, Jr., and Nancy Stokey [Lucas and Stokey, 1983].

The model revisits classic issues about how to pay for a war.

Here a *war* means a more or less temporary surge in an exogenous government expenditure process.

The model features

- a government that must finance an exogenous stream of government expenditures with either
  - a flat rate tax on labor, or
  - purchases and sales from a full array of Arrow state-contingent securities
- a representative household that values consumption and leisure
- a linear production function mapping labor into a single good
- a Ramsey planner who at time $t = 0$ chooses a plan for taxes and trades of Arrow securities for all $t \geq 0$

After first presenting the model in a space of sequences, we shall represent it recursively in terms of two Bellman equations formulated along lines that we encountered in *Dynamic Stackelberg models*.

As in *Dynamic Stackelberg models*, to apply dynamic programming we shall define the state vector artfully.

In particular, we shall include forward-looking variables that summarize optimal responses of private agents to a Ramsey plan.

See *Optimal taxation* for analysis within a linear-quadratic setting.

Let's start with some standard imports:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import root
from quantecon import MarkovChain
from quantecon.optimize.nelder_mead import nelder_mead
from numba import njit, prange, float64
from numba.experimental import jitclass
```

45.2 A Competitive Equilibrium with Distorting Taxes

At time $t \geq 0$ a random variable s_t belongs to a time-invariant set $S = [1, 2, \ldots, S]$.

For $t \geq 0$, a history $s^t = [s_t, s_{t-1}, \ldots, s_0]$ of an exogenous state s_t has joint probability density $\pi_t(s^t)$.

We begin by assuming that government purchases $g_t(s^t)$ at time $t \geq 0$ depend on s^t.

Let $c_t(s^t)$, $\ell_t(s^t)$, and $n_t(s^t)$ denote consumption, leisure, and labor supply, respectively, at history s^t and date t.

A representative household is endowed with one unit of time that can be divided between leisure ℓ_t and labor n_t:

$$n_t(s^t) + \ell_t(s^t) = 1 \tag{45.1}$$

Output equals $n_t(s^t)$ and can be divided between $c_t(s^t)$ and $g_t(s^t)$

$$c_t(s^t) + g_t(s^t) = n_t(s^t) \tag{45.2}$$

A representative household's preferences over $\{c_t(s^t), \ell_t(s^t)\}_{t=0}^{\infty}$ are ordered by

$$\sum_{t=0}^{\infty} \sum_{s^t} \beta^t \pi_t(s^t) u[c_t(s^t), \ell_t(s^t)] \tag{45.3}$$

where the utility function u is increasing, strictly concave, and three times continuously differentiable in both arguments.

The technology pins down a pre-tax wage rate to unity for all t, s^t.

The government imposes a flat-rate tax $\tau_t(s^t)$ on labor income at time t, history s^t.

There are complete markets in one-period Arrow securities.

One unit of an Arrow security issued at time t at history s^t and promising to pay one unit of time $t+1$ consumption in state s_{t+1} costs $p_{t+1}(s_{t+1}|s^t)$.

The government issues one-period Arrow securities each period.

The government has a sequence of budget constraints whose time $t \geq 0$ component is

$$g_t(s^t) = \tau_t(s^t) n_t(s^t) + \sum_{s_{t+1}} p_{t+1}(s_{t+1}|s^t) b_{t+1}(s_{t+1}|s^t) - b_t(s_t|s^{t-1}) \tag{45.4}$$

where

- $p_{t+1}(s_{t+1}|s^t)$ is a competitive equilibrium price of one unit of consumption at date $t+1$ in state s_{t+1} at date t and history s^t.
- $b_t(s_t|s^{t-1})$ is government debt falling due at time t, history s^t.

Government debt $b_0(s_0)$ is an exogenous initial condition.

The representative household has a sequence of budget constraints whose time $t \geq 0$ component is

$$c_t(s^t) + \sum_{s_{t+1}} p_t(s_{t+1}|s^t) b_{t+1}(s_{t+1}|s^t) = [1 - \tau_t(s^t)]\, n_t(s^t) + b_t(s_t|s^{t-1}) \quad \forall t \geq 0 \tag{45.5}$$

A **government policy** is an exogenous sequence $\{g(s_t)\}_{t=0}^{\infty}$, a tax rate sequence $\{\tau_t(s^t)\}_{t=0}^{\infty}$, and a government debt sequence $\{b_{t+1}(s^{t+1})\}_{t=0}^{\infty}$.

A **feasible allocation** is a consumption-labor supply plan $\{c_t(s^t), n_t(s^t)\}_{t=0}^{\infty}$ that satisfies (45.2) at all t, s^t.

A **price system** is a sequence of Arrow security prices $\{p_{t+1}(s_{t+1}|s^t)\}_{t=0}^{\infty}$.

The household faces the price system as a price-taker and takes the government policy as given.

The household chooses $\{c_t(s^t), \ell_t(s^t)\}_{t=0}^{\infty}$ to maximize (45.3) subject to (45.5) and (45.1) for all t, s^t.

A **competitive equilibrium with distorting taxes** is a feasible allocation, a price system, and a government policy such that

- Given the price system and the government policy, the allocation solves the household's optimization problem.
- Given the allocation, government policy, and price system, the government's budget constraint is satisfied for all t, s^t.

Note: There are many competitive equilibria with distorting taxes.

They are indexed by different government policies.

The **Ramsey problem** or **optimal taxation problem** is to choose a competitive equilibrium with distorting taxes that maximizes (45.3).

45.2.1 Arrow-Debreu Version of Price System

We find it convenient sometimes to work with the Arrow-Debreu price system that is implied by a sequence of Arrow securities prices.

Let $q_t^0(s^t)$ be the price at time 0, measured in time 0 consumption goods, of one unit of consumption at time t, history s^t.

The following recursion relates Arrow-Debreu prices $\{q_t^0(s^t)\}_{t=0}^{\infty}$ to Arrow securities prices $\{p_{t+1}(s_{t+1}|s^t)\}_{t=0}^{\infty}$

$$q_{t+1}^0(s^{t+1}) = p_{t+1}(s_{t+1}|s^t) q_t^0(s^t) \quad s.t. \quad q_0^0(s^0) = 1 \tag{45.6}$$

Arrow-Debreu prices are useful when we want to compress a sequence of budget constraints into a single intertemporal budget constraint, as we shall find it convenient to do below.

45.2.2 Primal Approach

We apply a popular approach to solving a Ramsey problem, called the *primal approach*.

The idea is to use first-order conditions for household optimization to eliminate taxes and prices in favor of quantities, then pose an optimization problem cast entirely in terms of quantities.

After Ramsey quantities have been found, taxes and prices can then be unwound from the allocation.

The primal approach uses four steps:

1. Obtain first-order conditions of the household's problem and solve them for $\{q_t^0(s^t), \tau_t(s^t)\}_{t=0}^{\infty}$ as functions of the allocation $\{c_t(s^t), n_t(s^t)\}_{t=0}^{\infty}$.
2. Substitute these expressions for taxes and prices in terms of the allocation into the household's present-value budget constraint.
    - This intertemporal constraint involves only the allocation and is regarded as an *implementability constraint*.
3. Find the allocation that maximizes the utility of the representative household (45.3) subject to the feasibility constraints (45.1) and (45.2) and the implementability condition derived in step 2.
    - This optimal allocation is called the **Ramsey allocation**.
4. Use the Ramsey allocation together with the formulas from step 1 to find taxes and prices.

45.2.3 The Implementability Constraint

By sequential substitution of one one-period budget constraint (45.5) into another, we can obtain the household's present-value budget constraint:

$$\sum_{t=0}^{\infty}\sum_{s^t} q_t^0(s^t)c_t(s^t) = \sum_{t=0}^{\infty}\sum_{s^t} q_t^0(s^t)[1-\tau_t(s^t)]n_t(s^t) + b_0 \tag{45.7}$$

$\{q_t^0(s^t)\}_{t=1}^{\infty}$ can be interpreted as a time 0 Arrow-Debreu price system.

To approach the Ramsey problem, we study the household's optimization problem.

First-order conditions for the household's problem for $\ell_t(s^t)$ and $b_t(s_{t+1}|s^t)$, respectively, imply

$$(1-\tau_t(s^t)) = \frac{u_l(s^t)}{u_c(s^t)} \tag{45.8}$$

and

$$p_{t+1}(s_{t+1}|s^t) = \beta\pi(s_{t+1}|s^t)\left(\frac{u_c(s^{t+1})}{u_c(s^t)}\right) \tag{45.9}$$

where $\pi(s_{t+1}|s^t)$ is the probability distribution of s_{t+1} conditional on history s^t.

Equation (45.9) implies that the Arrow-Debreu price system satisfies

$$q_t^0(s^t) = \beta^t\pi_t(s^t)\frac{u_c(s^t)}{u_c(s^0)} \tag{45.10}$$

(The stochastic process $\{q_t^0(s^t)\}$ is an instance of what finance economists call a *stochastic discount factor* process.)

Using the first-order conditions (45.8) and (45.9) to eliminate taxes and prices from (45.7), we derive the *implementability condition*

$$\sum_{t=0}^{\infty}\sum_{s^t}\beta^t\pi_t(s^t)[u_c(s^t)c_t(s^t) - u_\ell(s^t)n_t(s^t)] - u_c(s^0)b_0 = 0 \tag{45.11}$$

The **Ramsey problem** is to choose a feasible allocation that maximizes

$$\sum_{t=0}^{\infty}\sum_{s^t}\beta^t\pi_t(s^t)u[c_t(s^t), 1-n_t(s^t)] \tag{45.12}$$

subject to (45.11).

45.2.4 Solution Details

First, define a "pseudo utility function"

$$V\left[c_t(s^t), n_t(s^t), \Phi\right] = u[c_t(s^t), 1-n_t(s^t)] + \Phi\left[u_c(s^t)c_t(s^t) - u_\ell(s^t)n_t(s^t)\right] \tag{45.13}$$

where Φ is a Lagrange multiplier on the implementability condition (45.7).

Next form the Lagrangian

$$J = \sum_{t=0}^{\infty}\sum_{s^t}\beta^t\pi_t(s^t)\Big\{V[c_t(s^t), n_t(s^t), \Phi] + \theta_t(s^t)\Big[n_t(s^t) - c_t(s^t) - g_t(s_t)\Big]\Big\} - \Phi u_c(0)b_0 \tag{45.14}$$

where $\{\theta_t(s^t); \forall s^t\}_{t\geq 0}$ is a sequence of Lagrange multipliers on the feasible conditions (45.2).

Given an initial government debt b_0, we want to maximize J with respect to $\{c_t(s^t), n_t(s^t); \forall s^t\}_{t\geq 0}$ and to minimize with respect to Φ and with respect to $\{\theta(s^t); \forall s^t\}_{t\geq 0}$.

The first-order conditions for the Ramsey problem for periods $t \geq 1$ and $t = 0$, respectively, are

$$\begin{aligned} c_t(s^t)\colon\ & (1+\Phi)u_c(s^t) + \Phi\left[u_{cc}(s^t)c_t(s^t) - u_{\ell c}(s^t)n_t(s^t)\right] - \theta_t(s^t) = 0, \quad t \geq 1 \\ n_t(s^t)\colon\ & -(1+\Phi)u_\ell(s^t) - \Phi\left[u_{c\ell}(s^t)c_t(s^t) - u_{\ell\ell}(s^t)n_t(s^t)\right] + \theta_t(s^t) = 0, \quad t \geq 1 \end{aligned} \tag{45.15}$$

and

$$\begin{aligned} c_0(s^0,b_0)\colon\ & (1+\Phi)u_c(s^0,b_0) + \Phi\left[u_{cc}(s^0,b_0)c_0(s^0,b_0) - u_{\ell c}(s^0,b_0)n_0(s^0,b_0)\right] - \theta_0(s^0,b_0) \\ & - \Phi u_{cc}(s^0,b_0)b_0 = 0 \\ n_0(s^0,b_0)\colon\ & -(1+\Phi)u_\ell(s^0,b_0) - \Phi\left[u_{c\ell}(s^0,b_0)c_0(s^0,b_0) - u_{\ell\ell}(s^0,b_0)n_0(s^0,b_0)\right] + \theta_0(s^0,b_0) \\ & + \Phi u_{c\ell}(s^0,b_0)b_0 = 0 \end{aligned} \tag{45.16}$$

Please note how these first-order conditions differ between $t = 0$ and $t \geq 1$.

It is instructive to use first-order conditions (45.15) for $t \geq 1$ to eliminate the multipliers $\theta_t(s^t)$.

For convenience, we suppress the time subscript and the index s^t and obtain

$$\begin{aligned} &(1+\Phi)u_c(c,1-c-g) + \Phi\big[cu_{cc}(c,1-c-g) - (c+g)u_{\ell c}(c,1-c-g)\big] \\ &\quad = (1+\Phi)u_\ell(c,1-c-g) + \Phi\big[cu_{c\ell}(c,1-c-g) - (c+g)u_{\ell\ell}(c,1-c-g)\big] \end{aligned} \tag{45.17}$$

where we have imposed conditions (45.1) and (45.2).

Equation (45.17) is one equation that can be solved to express the unknown c as a function of the exogenous variable g and the Lagrange multiplier Φ.

We also know that time $t = 0$ quantities c_0 and n_0 satisfy

$$\begin{aligned} &(1+\Phi)u_c(c,1-c-g) + \Phi\big[cu_{cc}(c,1-c-g) - (c+g)u_{\ell c}(c,1-c-g)\big] \\ &\quad = (1+\Phi)u_\ell(c,1-c-g) + \Phi\big[cu_{c\ell}(c,1-c-g) - (c+g)u_{\ell\ell}(c,1-c-g)\big] + \Phi(u_{cc} - u_{c,\ell})b_0 \end{aligned} \tag{45.18}$$

Notice that a counterpart to b_0 does *not* appear in (45.17), so c does not *directly* depend on it for $t \geq 1$.

But things are different for time $t = 0$.

An analogous argument for the $t = 0$ equations (45.16) leads to one equation that can be solved for c_0 as a function of the pair $(g(s_0), b_0)$ and the Lagrange multiplier Φ.

These outcomes mean that the following statement would be true even when government purchases are history-dependent functions $g_t(s^t)$ of the history of s^t.

Proposition: If government purchases are equal after two histories s^t and $\tilde{s}^\tau$ for $t, \tau \geq 0$, i.e., if

$$g_t(s^t) = g^\tau(\tilde{s}^\tau) = g$$

then it follows from (45.17) that the Ramsey choices of consumption and leisure, $(c_t(s^t), \ell_t(s^t))$ and $(c_j(\tilde{s}^\tau), \ell_j(\tilde{s}^\tau))$, are identical.

The proposition asserts that the optimal allocation is a function of the currently realized quantity of government purchases g only and does *not* depend on the specific history that preceded that realization of g.

45.2.5 The Ramsey Allocation for a Given Multiplier

Temporarily take Φ as given.

We shall compute $c_0(s^0, b_0)$ and $n_0(s^0, b_0)$ from the first-order conditions (45.16).

Evidently, for $t \geq 1$, c and n depend on the time t realization of g only.

But for $t = 0$, c and n depend on both g_0 and the government's initial debt b_0.

Thus, while b_0 influences c_0 and n_0, there appears no analogous variable b_t that influences c_t and n_t for $t \geq 1$.

The absence of b_t as a direct determinant of the Ramsey allocation for $t \geq 1$ and its presence for $t = 0$ is a symptom of the *time-inconsistency* of a Ramsey plan.

Of course, b_0 affects the Ramsey allocation for $t \geq 1$ *indirectly* through its effect on Φ.

Φ has to take a value that assures that the household and the government's budget constraints are both satisfied at a candidate Ramsey allocation and price system associated with that Φ.

45.2.6 Further Specialization

At this point, it is useful to specialize the model in the following ways.

We assume that s is governed by a finite state Markov chain with states $s \in [1, \ldots, S]$ and transition matrix Π, where

$$\Pi(s'|s) = \text{Prob}(s_{t+1} = s'|s_t = s)$$

Also, assume that government purchases g are an exact time-invariant function $g(s)$ of s.

We maintain these assumptions throughout the remainder of this lecture.

45.2.7 Determining the Lagrange Multiplier

We complete the Ramsey plan by computing the Lagrange multiplier Φ on the implementability constraint (45.11).

Government budget balance restricts Φ via the following line of reasoning.

The household's first-order conditions imply

$$(1 - \tau_t(s^t)) = \frac{u_l(s^t)}{u_c(s^t)} \tag{45.19}$$

and the implied one-period Arrow securities prices

$$p_{t+1}(s_{t+1}|s^t) = \beta \Pi(s_{t+1}|s_t) \frac{u_c(s^{t+1})}{u_c(s^t)} \tag{45.20}$$

Substituting from (45.19), (45.20), and the feasibility condition (45.2) into the recursive version (45.5) of the household budget constraint gives

$$\begin{aligned} u_c(s^t)[n_t(s^t) - g_t(s^t)] + \beta \sum_{s_{t+1}} \Pi(s_{t+1}|s_t) u_c(s^{t+1}) b_{t+1}(s_{t+1}|s^t) \\ = u_l(s^t) n_t(s^t) + u_c(s^t) b_t(s_t|s^{t-1}) \end{aligned} \tag{45.21}$$

Define $x_t(s^t) = u_c(s^t) b_t(s_t|s^{t-1})$.

Notice that $x_t(s^t)$ appears on the right side of (45.21) while β times the conditional expectation of $x_{t+1}(s^{t+1})$ appears on the left side.

Hence the equation shares much of the structure of a simple asset pricing equation with x_t being analogous to the price of the asset at time t.

We learned earlier that for a Ramsey allocation $c_t(s^t), n_t(s^t)$, and $b_t(s_t|s^{t-1})$, and therefore also $x_t(s^t)$, are each functions of s_t only, being independent of the history s^{t-1} for $t \geq 1$.

That means that we can express equation (45.21) as

$$u_c(s)[n(s) - g(s)] + \beta \sum_{s'} \Pi(s'|s)x'(s') = u_l(s)n(s) + x(s) \tag{45.22}$$

where s' denotes a next period value of s and $x'(s')$ denotes a next period value of x.

Given $n(s)$ for $s = 1, \ldots, S$, equation (45.22) is easy to solve for $x(s)$ for $s = 1, \ldots, S$.

If we let $\vec{n}, \vec{g}, \vec{x}$ denote $S \times 1$ vectors whose ith elements are the respective n, g, and x values when $s = i$, and let Π be the transition matrix for the Markov state s, then we can express (45.22) as the matrix equation

$$\vec{u}_c(\vec{n} - \vec{g}) + \beta\Pi\vec{x} = \vec{u}_l\vec{n} + \vec{x} \tag{45.23}$$

This is a system of S linear equations in the $S \times 1$ vector x, whose solution is

$$\vec{x} = (I - \beta\Pi)^{-1}[\vec{u}_c(\vec{n} - \vec{g}) - \vec{u}_l\vec{n}] \tag{45.24}$$

In these equations, by $\vec{u}_c\vec{n}$, for example, we mean element-by-element multiplication of the two vectors.

After solving for $\vec{x}$, we can find $b(s_t|s^{t-1})$ in Markov state $s_t = s$ from $b(s) = \frac{x(s)}{u_c(s)}$ or the matrix equation

$$\vec{b} = \frac{\vec{x}}{\vec{u}_c} \tag{45.25}$$

where division here means an element-by-element division of the respective components of the $S \times 1$ vectors $\vec{x}$ and $\vec{u}_c$.

Here is a computational algorithm:

1. Start with a guess for the value for Φ, then use the first-order conditions and the feasibility conditions to compute $c(s_t), n(s_t)$ for $s \in [1, \ldots, S]$ and $c_0(s_0, b_0)$ and $n_0(s_0, b_0)$, given Φ.
    - these are $2(S+1)$ equations in $2(S+1)$ unknowns.
2. Solve the S equations (45.24) for the S elements of $\vec{x}$.
    - these depend on Φ.
3. Find a Φ that satisfies

$$u_{c,0}b_0 = u_{c,0}(n_0 - g_0) - u_{l,0}n_0 + \beta \sum_{s=1}^{S} \Pi(s|s_0)x(s) \tag{45.26}$$

   by gradually raising Φ if the left side of (45.26) exceeds the right side and lowering Φ if the left side is less than the right side.
4. After computing a Ramsey allocation, recover the flat tax rate on labor from (45.8) and the implied one-period Arrow securities prices from (45.9).

In summary, when g_t is a time-invariant function of a Markov state s_t, a Ramsey plan can be constructed by solving $3S + 3$ equations for S components each of $\vec{c}$, $\vec{n}$, and $\vec{x}$ together with n_0, c_0, and Φ.

45.2.8 Time Inconsistency

Let $\{\tau_t(s^t)\}_{t=0}^\infty, \{b_{t+1}(s_{t+1}|s^t)\}_{t=0}^\infty$ be a time 0, state s_0 Ramsey plan.

Then $\{\tau_j(s^j)\}_{j=t}^\infty, \{b_{j+1}(s_{j+1}|s^j)\}_{j=t}^\infty$ is a time t, history s^t continuation of a time 0, state s_0 Ramsey plan.

A time t, history s^t Ramsey plan is a Ramsey plan that starts from initial conditions $s^t, b_t(s_t|s^{t-1})$.

A time t, history s^t continuation of a time 0, state 0 Ramsey plan is *not* a time t, history s^t Ramsey plan.

The means that a Ramsey plan is *not time consistent*.

Another way to say the same thing is that a Ramsey plan is *time inconsistent*.

The reason is that a continuation Ramsey plan takes $u_{ct}b_t(s_t|s^{t-1})$ as given, not $b_t(s_t|s^{t-1})$.

We shall discuss this more below.

45.2.9 Specification with CRRA Utility

In our calculations below and in a *subsequent lecture* based on an *extension* of the Lucas-Stokey model by Aiyagari, Marcet, Sargent, and Seppälä (2002) [Aiyagari *et al.*, 2002], we shall modify the one-period utility function assumed above.

(We adopted the preceding utility specification because it was the one used in the original Lucas-Stokey paper [Lucas and Stokey, 1983]. We shall soon revert to that specification in a subsequent section.)

We will modify their specification by instead assuming that the representative agent has utility function

$$u(c,n) = \frac{c^{1-\sigma}}{1-\sigma} - \frac{n^{1+\gamma}}{1+\gamma}$$

where $\sigma > 0, \gamma > 0$.

We continue to assume that

$$c_t + g_t = n_t$$

We eliminate leisure from the model.

We also eliminate Lucas and Stokey's restriction that $\ell_t + n_t \leq 1$.

We replace these two things with the assumption that labor $n_t \in [0, +\infty]$.

With these adjustments, the analysis of Lucas and Stokey prevails once we make the following replacements

$$\begin{aligned}
u_\ell(c,\ell) &\sim -u_n(c,n) \\
u_c(c,\ell) &\sim u_c(c,n) \\
u_{\ell,\ell}(c,\ell) &\sim u_{nn}(c,n) \\
u_{c,c}(c,\ell) &\sim u_{c,c}(c,n) \\
u_{c,\ell}(c,\ell) &\sim 0
\end{aligned}$$

With these understandings, equations (45.17) and (45.18) simplify in the case of the CRRA utility function.

They become

$$(1+\Phi)[u_c(c) + u_n(c+g)] + \Phi[cu_{cc}(c) + (c+g)u_{nn}(c+g)] = 0 \tag{45.27}$$

and

$$(1+\Phi)[u_c(c_0) + u_n(c_0+g_0)] + \Phi[c_0u_{cc}(c_0) + (c_0+g_0)u_{nn}(c_0+g_0)] - \Phi u_{cc}(c_0)b_0 = 0 \tag{45.28}$$

In equation (45.27), it is understood that c and g are each functions of the Markov state s.

In addition, the time $t = 0$ budget constraint is satisfied at c_0 and initial government debt b_0:

$$b_0 + g_0 = \tau_0(c_0 + g_0) + \beta \sum_{s=1}^{S} \Pi(s|s_0) \frac{u_c(s)}{u_{c,0}} b_1(s) \tag{45.29}$$

where τ_0 is the time $t = 0$ tax rate.

In equation (45.29), it is understood that

$$\tau_0 = 1 - \frac{u_{l,0}}{u_{c,0}}$$

45.2.10 Sequence Implementation

The above steps are implemented in a class called SequentialLS

```
class SequentialLS:

    '''
    Class that takes a preference object, state transition matrix,
    and state contingent government expenditure plan as inputs, and
    solves the sequential allocation problem described above.
    It returns optimal allocations about consumption and labor supply,
    as well as the multiplier on the implementability constraint Φ.
    '''

    def __init__(self,
                 pref,
                 π=np.full((2, 2), 0.5),
                 g=np.array([0.1, 0.2])):

        # Initialize from pref object attributes
        self.β, self.π, self.g = pref.β, π, g
        self.mc = MarkovChain(self.π)
        self.S = len(π)  # Number of states
        self.pref = pref

        # Find the first best allocation
        self.find_first_best()

    def FOC_first_best(self, c, g):
        '''
        First order conditions that characterize
        the first best allocation.
        '''

        pref = self.pref
        Uc, Ul = pref.Uc, pref.Ul

        n = c + g
        l = 1 - n

        return Uc(c, l) - Ul(c, l)

    def find_first_best(self):
```

(continues on next page)

(continued from previous page)

```
        '''
        Find the first best allocation
        '''
        S, g = self.S, self.g

        res = root(self.FOC_first_best, np.full(S, 0.5), args=(g,))

        if (res.fun > 1e-10).any():
            raise Exception('Could not find first best')

        self.cFB = res.x
        self.nFB = self.cFB + g

    def FOC_time1(self, c, Φ, g):
        '''
        First order conditions that characterize
        optimal time 1 allocation problems.
        '''

        pref = self.pref
        Uc, Ucc, Ul, Ull, Ulc = pref.Uc, pref.Ucc, pref.Ul, pref.Ull, pref.Ulc

        n = c + g
        l = 1 - n

        LHS = (1 + Φ) * Uc(c, l) + Φ * (c * Ucc(c, l) - n * Ulc(c, l))
        RHS = (1 + Φ) * Ul(c, l) + Φ * (c * Ulc(c, l) - n * Ull(c, l))

        diff = LHS - RHS

        return diff

    def time1_allocation(self, Φ):
        '''
        Computes optimal allocation for time t >= 1 for a given Φ
        '''
        pref = self.pref
        S, g = self.S, self.g

        # use the first best allocation as intial guess
        res = root(self.FOC_time1, self.cFB, args=(Φ, g))

        if (res.fun > 1e-10).any():
            raise Exception('Could not find LS allocation.')

        c = res.x
        n = c + g
        l = 1 - n

        # Compute x
        I = pref.Uc(c, n) * c - pref.Ul(c, l) * n
        x = np.linalg.solve(np.eye(S) - self.β * self.π, I)

        return c, n, x

    def FOC_time0(self, c0, Φ, g0, b0):
```

(continues on next page)

(continued from previous page)

```
        '''
        First order conditions that characterize
        time 0 allocation problem.
        '''

        pref = self.pref
        Ucc, Ulc = pref.Ucc, pref.Ulc

        n0 = c0 + g0
        l0 = 1 - n0

        diff = self.FOC_time1(c0, Φ, g0)
        diff -= Φ * (Ucc(c0, l0) - Ulc(c0, l0)) * b0

        return diff

    def implementability(self, Φ, b0, s0, cn0_arr):
        '''
        Compute the differences between the RHS and LHS
        of the implementability constraint given Φ,
        initial debt, and initial state.
        '''

        pref, π, g, β = self.pref, self.π, self.g, self.β
        Uc, Ul = pref.Uc, pref.Ul
        g0 = self.g[s0]

        c, n, x = self.time1_allocation(Φ)

        res = root(self.FOC_time0, cn0_arr[0], args=(Φ, g0, b0))
        c0 = res.x
        n0 = c0 + g0
        l0 = 1 - n0

        cn0_arr[:] = c0.item(), n0.item()

        LHS = Uc(c0, l0) * b0
        RHS = Uc(c0, l0) * c0 - Ul(c0, l0) * n0 + β * π[s0] @ x

        return RHS - LHS

    def time0_allocation(self, b0, s0):
        '''
        Finds the optimal time 0 allocation given
        initial government debt b0 and state s0
        '''

        # use the first best allocation as initial guess
        cn0_arr = np.array([self.cFB[s0], self.nFB[s0]])

        res = root(self.implementability, 0., args=(b0, s0, cn0_arr))

        if (res.fun > 1e-10).any():
            raise Exception('Could not find time 0 LS allocation.')

        Φ = res.x[0]
```

(continues on next page)

(continued from previous page)

```
        c0, n0 = cn0_arr

        return Φ, c0, n0

    def τ(self, c, n):
        '''
        Computes τ given c, n
        '''
        pref = self.pref
        Uc, Ul = pref.Uc, pref.Ul

        return 1 - Ul(c, 1-n) / Uc(c, 1-n)

    def simulate(self, b0, s0, T, sHist=None):
        '''
        Simulates planners policies for T periods
        '''
        pref, π, β = self.pref, self.π, self.β
        Uc = pref.Uc

        if sHist is None:
            sHist = self.mc.simulate(T, s0)

        cHist, nHist, Bhist, τHist, ΦHist = np.empty((5, T))
        RHist = np.empty(T-1)

        # Time 0
        Φ, cHist[0], nHist[0] = self.time0_allocation(b0, s0)
        τHist[0] = self.τ(cHist[0], nHist[0])
        Bhist[0] = b0
        ΦHist[0] = Φ

        # Time 1 onward
        for t in range(1, T):
            c, n, x = self.time1_allocation(Φ)
            τ = self.τ(c, n)
            u_c = Uc(c, 1-n)
            s = sHist[t]
            Eu_c = π[sHist[t-1]] @ u_c
            cHist[t], nHist[t], Bhist[t], τHist[t] = c[s], n[s], x[s] / u_c[s], τ[s]
            RHist[t-1] = Uc(cHist[t-1], 1-nHist[t-1]) / (β * Eu_c)
            ΦHist[t] = Φ

        gHist = self.g[sHist]
        yHist = nHist

        return [cHist, nHist, Bhist, τHist, gHist, yHist, sHist, ΦHist, RHist]
```

45.3 Recursive Formulation of the Ramsey Problem

We now temporarily revert to Lucas and Stokey's specification.

We start by noting that $x_t(s^t) = u_c(s^t)b_t(s_t|s^{t-1})$ in equation (45.21) appears to be a purely "forward-looking" variable.

But $x_t(s^t)$ is a natural candidate for a state variable in a recursive formulation of the Ramsey problem, one that records history-dependence and so is `backward-looking`.

45.3.1 Intertemporal Delegation

To express a Ramsey plan recursively, we imagine that a time 0 Ramsey planner is followed by a sequence of continuation Ramsey planners at times $t = 1, 2, ...$

A "continuation Ramsey planner" at time $t \geq 1$ has a different objective function and faces different constraints and state variables than does the Ramsey planner at time $t = 0$.

A key step in representing a Ramsey plan recursively is to regard the marginal utility scaled government debts $x_t(s^t) = u_c(s^t)b_t(s_t|s^{t-1})$ as predetermined quantities that continuation Ramsey planners at times $t \geq 1$ are obligated to attain.

Continuation Ramsey planners do this by choosing continuation policies that induce the representative household to make choices that imply that $u_c(s^t)b_t(s_t|s^{t-1}) = x_t(s^t)$.

A time $t \geq 1$ continuation Ramsey planner faces x_t, s_t as state variables.

A time $t \geq 1$ continuation Ramsey planner delivers x_t by choosing a suitable n_t, c_t pair and a list of s_{t+1}-contingent continuation quantities x_{t+1} to bequeath to a time $t+1$ continuation Ramsey planner.

While a time $t \geq 1$ continuation Ramsey planner faces x_t, s_t as state variables, the time 0 Ramsey planner faces b_0, not x_0, as a state variable.

Furthermore, the Ramsey planner cares about $(c_0(s_0), \ell_0(s_0))$, while continuation Ramsey planners do not.

The time 0 Ramsey planner hands a state-contingent function that make x_1 a function of s_1 to a time 1, state s_1 continuation Ramsey planner.

These lines of delegated authorities and responsibilities across time express the continuation Ramsey planners' obligations to implement their parts of an original Ramsey plan that had been designed once-and-for-all at time 0.

45.3.2 Two Bellman Equations

After s_t has been realized at time $t \geq 1$, the state variables confronting the time t **continuation Ramsey planner** are (x_t, s_t).

- Let $V(x, s)$ be the value of a **continuation Ramsey plan** at $x_t = x, s_t = s$ for $t \geq 1$.
- Let $W(b, s)$ be the value of a **Ramsey plan** at time 0 at $b_0 = b$ and $s_0 = s$.

We work backward by preparing a Bellman equation for $V(x, s)$ first, then a Bellman equation for $W(b, s)$.

45.3.3 The Continuation Ramsey Problem

The Bellman equation for a time $t \geq 1$ continuation Ramsey planner is

$$V(x,s) = \max_{n,\{x'(s')\}} u(n-g(s),1-n) + \beta \sum_{s' \in S} \Pi(s'|s) V(x',s') \tag{45.30}$$

where maximization over n and the S elements of $x'(s')$ is subject to the single implementability constraint for $t \geq 1$:

$$x = u_c(n-g(s)) - u_l n + \beta \sum_{s' \in S} \Pi(s'|s) x'(s') \tag{45.31}$$

Here u_c and u_l are today's values of the marginal utilities.

For each given value of x, s, the continuation Ramsey planner chooses n and $x'(s')$ for each $s' \in S$.

Associated with a value function $V(x,s)$ that solves Bellman equation (45.30) are $S+1$ time-invariant policy functions

$$\begin{aligned} n_t &= f(x_t, s_t), \quad t \geq 1 \\ x_{t+1}(s_{t+1}) &= h(s_{t+1}; x_t, s_t),\ s_{t+1} \in S,\ t \geq 1 \end{aligned} \tag{45.32}$$

45.3.4 The Ramsey Problem

The Bellman equation of the time 0 Ramsey planner is

$$W(b_0, s_0) = \max_{n_0,\{x'(s_1)\}} u(n_0 - g_0, 1-n_0) + \beta \sum_{s_1 \in S} \Pi(s_1|s_0) V(x'(s_1), s_1) \tag{45.33}$$

where maximization over n_0 and the S elements of $x'(s_1)$ is subject to the time 0 implementability constraint

$$u_{c,0} b_0 = u_{c,0}(n_0 - g_0) - u_{l,0} n_0 + \beta \sum_{s_1 \in S} \Pi(s_1|s_0) x'(s_1) \tag{45.34}$$

coming from restriction (45.26).

Associated with a value function $W(b_0, n_0)$ that solves Bellman equation (45.33) are $S+1$ time 0 policy functions

$$\begin{aligned} n_0 &= f_0(b_0, s_0) \\ x_1(s_1) &= h_0(s_1; b_0, s_0) \end{aligned} \tag{45.35}$$

Notice the appearance of state variables (b_0, s_0) in the time 0 policy functions for the Ramsey planner as compared to (x_t, s_t) in the policy functions (45.32) for the time $t \geq 1$ continuation Ramsey planners.

The value function $V(x_t, s_t)$ of the time t continuation Ramsey planner equals $E_t \sum_{\tau=t}^{\infty} \beta^{\tau-t} u(c_\tau, l_\tau)$, where consumption and leisure processes are evaluated along the original time 0 Ramsey plan.

45.3.5 First-Order Conditions

Attach a Lagrange multiplier $\Phi_1(x,s)$ to constraint (45.31) and a Lagrange multiplier Φ_0 to constraint (45.26).

Time $t \geq 1$: First-order conditions for the time $t \geq 1$ constrained maximization problem on the right side of the continuation Ramsey planner's Bellman equation (45.30) are

$$\beta \Pi(s'|s) V_x(x',s') - \beta \Pi(s'|s) \Phi_1 = 0 \tag{45.36}$$

for $x'(s')$ and

$$(1+\Phi_1)(u_c - u_l) + \Phi_1 \left[n(u_{ll} - u_{lc}) + (n - g(s))(u_{cc} - u_{lc}) \right] = 0 \tag{45.37}$$

for n.

Given Φ_1, equation (45.37) is one equation to be solved for n as a function of s (or of $g(s)$).

Equation (45.36) implies $V_x(x', s') = \Phi_1$, while an envelope condition is $V_x(x, s) = \Phi_1$, so it follows that

$$V_x(x', s') = V_x(x, s) = \Phi_1(x, s) \tag{45.38}$$

Time $t = 0$: For the time 0 problem on the right side of the Ramsey planner's Bellman equation (45.33), first-order conditions are

$$V_x(x(s_1), s_1) = \Phi_0 \tag{45.39}$$

for $x(s_1), s_1 \in S$, and

$$\begin{aligned}(1 + \Phi_0)(u_{c,0} - u_{n,0}) + \Phi_0\left[n_0(u_{ll,0} - u_{lc,0}) + (n_0 - g(s_0))(u_{cc,0} - u_{cl,0})\right] \\ - \Phi_0(u_{cc,0} - u_{cl,0})b_0 = 0\end{aligned} \tag{45.40}$$

Notice similarities and differences between the first-order conditions for $t \geq 1$ and for $t = 0$.

An additional term is present in (45.40) except in three special cases

- $b_0 = 0$, or
- u_c is constant (i.e., preferences are quasi-linear in consumption), or
- initial government assets are sufficiently large to finance all government purchases with interest earnings from those assets so that $\Phi_0 = 0$

Except in these special cases, the allocation and the labor tax rate as functions of s_t differ between dates $t = 0$ and subsequent dates $t \geq 1$.

Naturally, the first-order conditions in this recursive formulation of the Ramsey problem agree with the first-order conditions derived when we first formulated the Ramsey plan in the space of sequences.

45.3.6 State Variable Degeneracy

Equations (45.38) and (45.39) imply that $\Phi_0 = \Phi_1$ and that

$$V_x(x_t, s_t) = \Phi_0 \tag{45.41}$$

for all $t \geq 1$.

When V is concave in x, this implies *state-variable degeneracy* along a Ramsey plan in the sense that for $t \geq 1$, x_t will be a time-invariant function of s_t.

Given Φ_0, this function mapping s_t into x_t can be expressed as a vector $\vec{x}$ that solves equation (45.34) for n and c as functions of g that are associated with $\Phi = \Phi_0$.

45.3.7 Manifestations of Time Inconsistency

While the marginal utility adjusted level of government debt x_t is a key state variable for the continuation Ramsey planners at $t \geq 1$, it is not a state variable at time 0.

The time 0 Ramsey planner faces b_0, not $x_0 = u_{c,0}b_0$, as a state variable.

The discrepancy in state variables faced by the time 0 Ramsey planner and the time $t \geq 1$ continuation Ramsey planners captures the differing obligations and incentives faced by the time 0 Ramsey planner and the time $t \geq 1$ continuation Ramsey planners.

- The time 0 Ramsey planner is obligated to honor government debt b_0 measured in time 0 consumption goods.
- The time 0 Ramsey planner can manipulate the *value* of government debt as measured by $u_{c,0}b_0$.
- In contrast, time $t \geq 1$ continuation Ramsey planners are obligated *not* to alter values of debt, as measured by $u_{c,t}b_t$, that they inherit from a preceding Ramsey planner or continuation Ramsey planner.

When government expenditures g_t are a time-invariant function of a Markov state s_t, a Ramsey plan and associated Ramsey allocation feature marginal utilities of consumption $u_c(s_t)$ that, given Φ, for $t \geq 1$ depend only on s_t, but that for $t = 0$ depend on b_0 as well.

This means that $u_c(s_t)$ will be a time-invariant function of s_t for $t \geq 1$, but except when $b_0 = 0$, a different function for $t = 0$.

This in turn means that prices of one-period Arrow securities $p_{t+1}(s_{t+1}|s_t) = p(s_{t+1}|s_t)$ will be the *same* time-invariant functions of (s_{t+1}, s_t) for $t \geq 1$, but a different function $p_0(s_1|s_0)$ for $t = 0$, except when $b_0 = 0$.

The differences between these time 0 and time $t \geq 1$ objects reflect the Ramsey planner's incentive to manipulate Arrow security prices and, through them, the value of initial government debt b_0.

45.3.8 Recursive Implementation

The above steps are implemented in a class called `RecursiveLS`.

```
class RecursiveLS:

    '''
    Compute the planner's allocation by solving Bellman
    equation.
    '''

    def __init__(self,
                 pref,
                 x_grid,
                 Π=np.full((2, 2), 0.5),
                 g=np.array([0.1, 0.2])):

        self.Π, self.g, self.S = Π, g, len(Π)
        self.pref, self.x_grid = pref, x_grid

        bounds = np.empty((self.S, 2))

        # bound for n
        bounds[0] = 0, 1

        # bound for xprime
        for s in range(self.S-1):
            bounds[s+1] = x_grid.min(), x_grid.max()

        self.bounds = bounds

        # initialization of time 1 value function
        self.V = None

    def time1_allocation(self, V=None, tol=1e-7):
        '''
        Solve the optimal time 1 allocation problem
        by iterating Bellman value function.
```

(continues on next page)

(continued from previous page)

```
        '''

        π, g, S = self.π, self.g, self.S
        pref, x_grid, bounds = self.pref, self.x_grid, self.bounds

        # initial guess of value function
        if V is None:
            V = np.zeros((len(x_grid), S))

        # initial guess of policy
        z = np.empty((len(x_grid), S, S+2))

        # guess of n
        z[:, :, 1] = 0.5

        # guess of xprime
        for s in range(S):
            for i in range(S-1):
                z[:, s, i+2] = x_grid

        while True:
            # value function iteration
            V_new, z_new = T(V, z, pref, π, g, x_grid, bounds)

            if np.max(np.abs(V - V_new)) < tol:
                break

            V = V_new
            z = z_new

        self.V = V_new
        self.z1 = z_new
        self.c1 = z_new[:, :, 0]
        self.n1 = z_new[:, :, 1]
        self.xprime1 = z_new[:, :, 2:]

        return V_new, z_new

    def time0_allocation(self, b0, s0):
        '''
        Find the optimal time 0 allocation by maximization.
        '''

        if self.V is None:
            self.time1_allocation()

        π, g, S = self.π, self.g, self.S
        pref, x_grid, bounds = self.pref, self.x_grid, self.bounds
        V, z1 = self.V, self.z1

        x = 1. # x is arbitrary
        res = nelder_mead(obj_V,
                          z1[0, s0, 1:-1],
                          args=(x, s0, V, pref, π, g, x_grid, b0),
                          bounds=bounds,
                          tol_f=1e-10)
```

(continues on next page)

(continued from previous page)

```
        n0, xprime0 = IC(res.x, x, s0, b0, pref, π, g)
        c0 = n0 - g[s0]
        z0 = np.array([c0, n0, *xprime0])

        self.z0 = z0
        self.n0 = n0
        self.c0 = n0 - g[s0]
        self.xprime0 = xprime0

        return z0

    def τ(self, c, n):
        '''
        Computes τ given c, n
        '''
        pref = self.pref
        uc, ul = pref.Uc(c, 1-n), pref.Ul(c, 1-n)

        return 1 - ul / uc

    def simulate(self, b0, s0, T, sHist=None):
        '''
        Simulates Ramsey plan for T periods
        '''
        pref, π = self.pref, self.π
        Uc = pref.Uc

        if sHist is None:
            sHist = self.mc.simulate(T, s0)

        cHist, nHist, Bhist, τHist, xHist = np.empty((5, T))
        RHist = np.zeros(T-1)

        # Time 0
        self.time0_allocation(b0, s0)
        cHist[0], nHist[0], xHist[0] = self.c0, self.n0, self.xprime0[s0]
        τHist[0] = self.τ(cHist[0], nHist[0])
        Bhist[0] = b0

        # Time 1 onward
        for t in range(1, T):
            s, x = sHist[t], xHist[t-1]
            cHist[t] = np.interp(x, self.x_grid, self.c1[:, s])
            nHist[t] = np.interp(x, self.x_grid, self.n1[:, s])

            τHist[t] = self.τ(cHist[t], nHist[t])

            Bhist[t] = x / Uc(cHist[t], 1-nHist[t])

            c, n = np.empty((2, self.S))
            for sprime in range(self.S):
                c[sprime] = np.interp(x, x_grid, self.c1[:, sprime])
                n[sprime] = np.interp(x, x_grid, self.n1[:, sprime])
            Euc = π[sHist[t-1]] @ Uc(c, 1-n)
            RHist[t-1] = Uc(cHist[t-1], 1-nHist[t-1]) / (self.pref.β * Euc)
```

(continues on next page)

(continued from previous page)

```
            gHist = self.g[sHist]
            yHist = nHist

            if t < T-1:
                sprime = sHist[t+1]
                xHist[t] = np.interp(x, self.x_grid, self.xprime1[:, s, sprime])

        return [cHist, nHist, Bhist, τHist, gHist, yHist, xHist, RHist]

# Helper functions

@njit(parallel=True)
def T(V, z, pref, π, g, x_grid, bounds):
    '''
    One step iteration of Bellman value function.
    '''

    S = len(π)

    V_new = np.empty_like(V)
    z_new = np.empty_like(z)

    for i in prange(len(x_grid)):
        x = x_grid[i]
        for s in prange(S):
            res = nelder_mead(obj_V,
                              z[i, s, 1:-1],
                              args=(x, s, V, pref, π, g, x_grid),
                              bounds=bounds,
                              tol_f=1e-10)

            # optimal policy
            n, xprime = IC(res.x, x, s, None, pref, π, g)
            z_new[i, s, 0] = n - g[s]        # c
            z_new[i, s, 1] = n               # n
            z_new[i, s, 2:] = xprime         # xprime

            V_new[i, s] = res.fun

    return V_new, z_new

@njit
def obj_V(z_sub, x, s, V, pref, π, g, x_grid, b0=None):
    '''
    The objective on the right hand side of the Bellman equation.
    z_sub contains guesses of n and xprime[:-1].
    '''

    S = len(π)
    β, U = pref.β, pref.U

    # find (n, xprime) that satisfies implementability constraint
    n, xprime = IC(z_sub, x, s, b0, pref, π, g)
    c, l = n-g[s], 1-n
```

(continues on next page)

(continued from previous page)

```
    # if xprime[-1] violates bound, return large penalty
    if (xprime[-1] < x_grid.min()):
        return -1e9 * (1 + np.abs(xprime[-1] - x_grid.min()))
    elif (xprime[-1] > x_grid.max()):
        return -1e9 * (1 + np.abs(xprime[-1] - x_grid.max()))

    # prepare Vprime vector
    Vprime = np.empty(S)
    for sprime in range(S):
        Vprime[sprime] = np.interp(xprime[sprime], x_grid, V[:, sprime])

    # compute the objective value
    obj = U(c, l) + β * π[s] @ Vprime

    return obj

@njit
def IC(z_sub, x, s, b0, pref, π, g):
    '''
    Find xprime[-1] that satisfies the implementability condition
    given the guesses of n and xprime[:-1].
    '''

    β, Uc, Ul = pref.β, pref.Uc, pref.Ul

    n = z_sub[0]
    xprime = np.empty(len(π))
    xprime[:-1] = z_sub[1:]

    c, l = n-g[s], 1-n
    uc = Uc(c, l)
    ul = Ul(c, l)

    if b0 is None:
        diff = x
    else:
        diff = uc * b0

    diff -= uc * (n - g[s]) - ul * n + β * π[s][:-1] @ xprime[:-1]
    xprime[-1] = diff / (β * π[s][-1])

    return n, xprime
```

45.4 Examples

We return to the setup with CRRA preferences described above.

45.4.1 Anticipated One-Period War

This example illustrates in a simple setting how a Ramsey planner manages risk.

Government expenditures are known for sure in all periods except one

- For $t < 3$ and $t > 3$ we assume that $g_t = g_l = 0.1$.
- At $t = 3$ a war occurs with probability 0.5.
  - If there is war, $g_3 = g_h = 0.2$
  - If there is no war $g_3 = g_l = 0.1$

We define the components of the state vector as the following six (t, g) pairs: $(0, g_l), (1, g_l), (2, g_l), (3, g_l), (3, g_h), (t \geq 4, g_l)$.

We think of these 6 states as corresponding to $s = 1, 2, 3, 4, 5, 6$.

The transition matrix is

$$\Pi = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.5 & 0.5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Government expenditures at each state are

$$g = \begin{pmatrix} 0.1 \\ 0.1 \\ 0.1 \\ 0.1 \\ 0.2 \\ 0.1 \end{pmatrix}$$

We assume that the representative agent has utility function

$$u(c, n) = \frac{c^{1-\sigma}}{1-\sigma} - \frac{n^{1+\gamma}}{1+\gamma}$$

and set $\sigma = 2$, $\gamma = 2$, and the discount factor $\beta = 0.9$.

Note: For convenience in terms of matching our code, we have expressed utility as a function of n rather than leisure l.

This utility function is implemented in the class `CRRAutility`.

```
crra_util_data = [
    ('β', float64),
    ('σ', float64),
    ('γ', float64)
]

@jitclass(crra_util_data)
class CRRAutility:

    def __init__(self,
                 β=0.9,
```

(continues on next page)

(continued from previous page)

```
                 σ=2,
                 γ=2):

        self.β, self.σ, self.γ = β, σ, γ

    # Utility function
    def U(self, c, l):
        # Note: `l` should not be interpreted as labor, it is an auxiliary
        # variable used to conveniently match the code and the equations
        # in the lecture
        σ = self.σ
        if σ == 1.:
            U = np.log(c)
        else:
            U = (c**(1 - σ) - 1) / (1 - σ)
        return U - (1-l) ** (1 + self.γ) / (1 + self.γ)

    # Derivatives of utility function
    def Uc(self, c, l):
        return c ** (-self.σ)

    def Ucc(self, c, l):
        return -self.σ * c ** (-self.σ - 1)

    def Ul(self, c, l):
        return (1-l) ** self.γ

    def Ull(self, c, l):
        return -self.γ * (1-l) ** (self.γ - 1)

    def Ucl(self, c, l):
        return 0

    def Ulc(self, c, l):
        return 0
```

We set initial government debt $b_0 = 1$.

We can now plot the Ramsey tax under both realizations of time $t = 3$ government expenditures

- black when $g_3 = .1$, and
- red when $g_3 = .2$

```
π = np.array([[0, 1, 0,   0,   0,  0],
              [0, 0, 1,   0,   0,  0],
              [0, 0, 0, 0.5, 0.5,  0],
              [0, 0, 0,   0,   0,  1],
              [0, 0, 0,   0,   0,  1],
              [0, 0, 0,   0,   0,  1]])

g = np.array([0.1, 0.1, 0.1, 0.2, 0.1, 0.1])
crra_pref = CRRAutility()

# Solve sequential problem
seq = SequentialLS(crra_pref, π=π, g=g)
sHist_h = np.array([0, 1, 2, 3, 5, 5, 5])
```

(continues on next page)

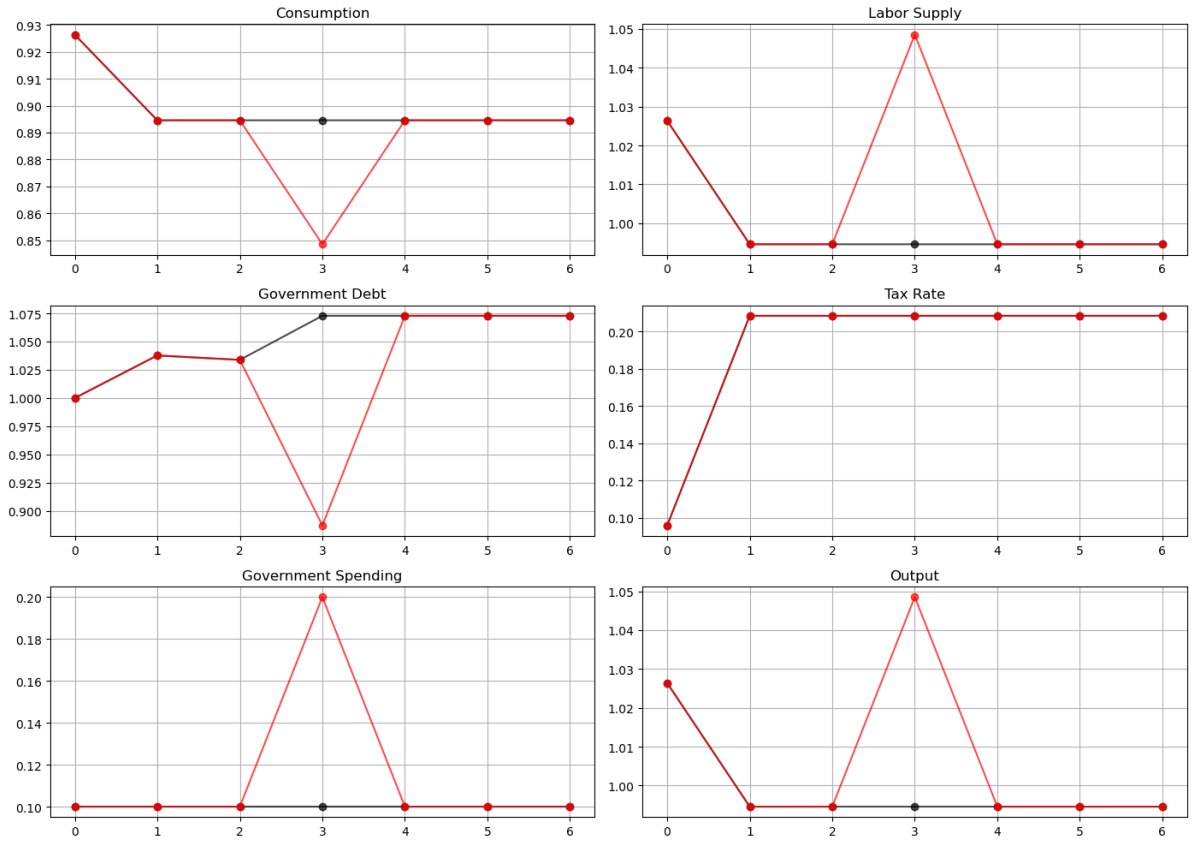
(continued from previous page)

```
sHist_l = np.array([0, 1, 2, 4, 5, 5, 5])
sim_seq_h = seq.simulate(1, 0, 7, sHist_h)
sim_seq_l = seq.simulate(1, 0, 7, sHist_l)

fig, axes = plt.subplots(3, 2, figsize=(14, 10))
titles = ['Consumption', 'Labor Supply', 'Government Debt',
          'Tax Rate', 'Government Spending', 'Output']

for ax, title, sim_l, sim_h in zip(axes.flatten(),
                                   titles,
                                   sim_seq_l[:6],
                                   sim_seq_h[:6]):
    ax.set(title=title)
    ax.plot(sim_l, '-ok', sim_h, '-or', alpha=0.7)
    ax.grid()

plt.tight_layout()
plt.show()
```



Tax smoothing

- the tax rate is constant for all $t \geq 1$
  - For $t \geq 1, t \neq 3$, this is a consequence of g_t being the same at all those dates.
  - For $t = 3$, it is a consequence of the special one-period utility function that we have assumed.
  - Under other one-period utility functions, the time $t = 3$ tax rate could be either higher or lower than for dates $t \geq 1, t \neq 3$.

- the tax rate is the same at $t = 3$ for both the high g_t outcome and the low g_t outcome

We have assumed that at $t = 0$, the government owes positive debt b_0.

It sets the time $t = 0$ tax rate partly with an eye to reducing the value $u_{c,0}b_0$ of b_0.

It does this by increasing consumption at time $t = 0$ relative to consumption in later periods.

This has the consequence of *lowering* the time $t = 0$ value of the gross interest rate for risk-free loans between periods t and $t + 1$, which equals

$$R_t = \frac{u_{c,t}}{\beta \mathbb{E}_t[u_{c,t+1}]}$$

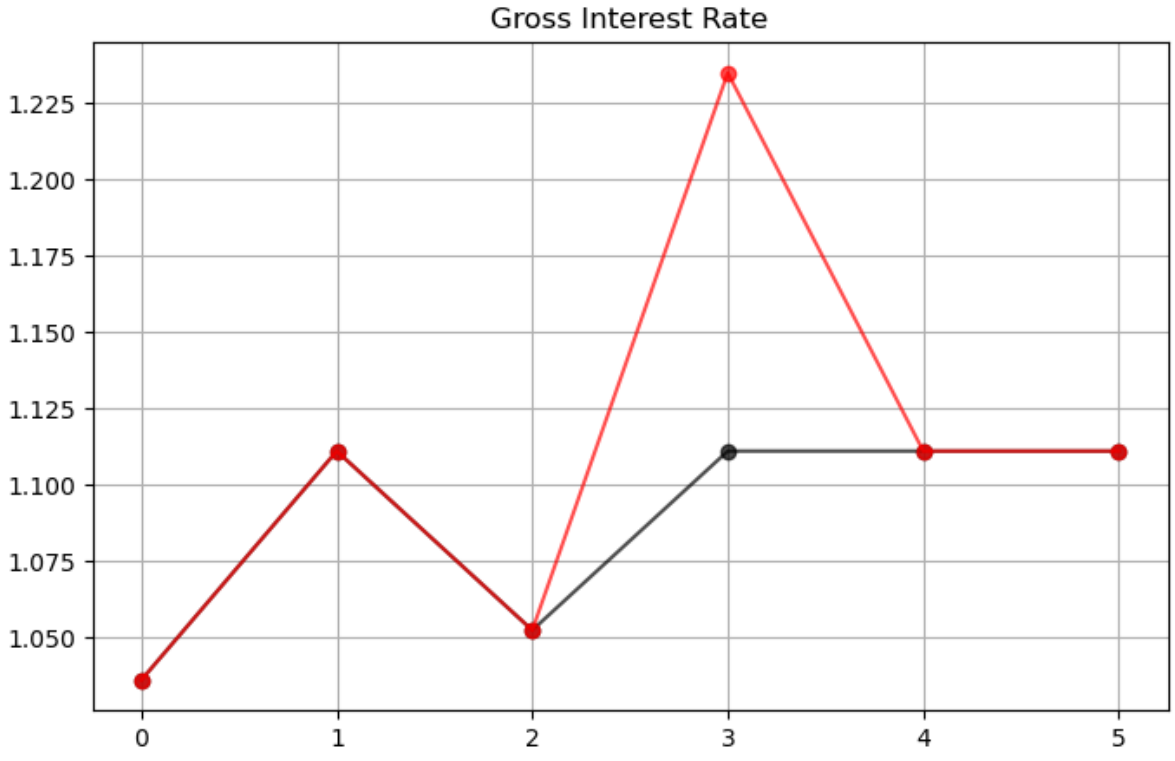
A tax policy that makes time $t = 0$ consumption be higher than time $t = 1$ consumption evidently decreases the risk-free rate one-period interest rate, R_t, at $t = 0$.

Lowering the time $t = 0$ risk-free interest rate makes time $t = 0$ consumption goods cheaper relative to consumption goods at later dates, thereby lowering the value $u_{c,0}b_0$ of initial government debt b_0.

We see this in a figure below that plots the time path for the risk-free interest rate under both realizations of the time $t = 3$ government expenditure shock.

The following plot illustrates how the government lowers the interest rate at time 0 by raising consumption

```
fix, ax = plt.subplots(figsize=(8, 5))
ax.set_title('Gross Interest Rate')
ax.plot(sim_seq_l[-1], '-ok', sim_seq_h[-1], '-or', alpha=0.7)
ax.grid()
plt.show()
```



45.4.2 Government Saving

At time $t = 0$ the government evidently *dissaves* since $b_1 > b_0$.

- This is a consequence of it setting a *lower* tax rate at $t = 0$, implying more consumption at $t = 0$.

At time $t = 1$, the government evidently *saves* since it has set the tax rate sufficiently high to allow it to set $b_2 < b_1$.

- Its motive for doing this is that it anticipates a likely war at $t = 3$.

At time $t = 2$ the government trades state-contingent Arrow securities to hedge against war at $t = 3$.

- It purchases a security that pays off when $g_3 = g_h$.
- It sells a security that pays off when $g_3 = g_l$.
- These purchases are designed in such a way that regardless of whether or not there is a war at $t = 3$, the government will begin period $t = 4$ with the *same* government debt.
- The time $t = 4$ debt level can be serviced with revenues from the constant tax rate set at times $t \geq 1$.

At times $t \geq 4$ the government rolls over its debt, knowing that the tax rate is set at a level that raises enough revenue to pay for government purchases and interest payments on its debt.

45.4.3 Time 0 Manipulation of Interest Rate

We have seen that when $b_0 > 0$, the Ramsey plan sets the time $t = 0$ tax rate partly with an eye toward lowering a risk-free interest rate for one-period loans between times $t = 0$ and $t = 1$.

By lowering this interest rate, the plan makes time $t = 0$ goods cheap relative to consumption goods at later times.

By doing this, it lowers the value of time $t = 0$ debt that it has inherited and must finance.

45.4.4 Time 0 and Time-Inconsistency

In the preceding example, the Ramsey tax rate at time 0 differs from its value at time 1.

To explore what is going on here, let's simplify things by removing the possibility of war at time $t = 3$.

The Ramsey problem then includes no randomness because $g_t = g_l$ for all t.

The figure below plots the Ramsey tax rates and gross interest rates at time $t = 0$ and time $t \geq 1$ as functions of the initial government debt (using the sequential allocation solution and a CRRA utility function defined above)

```
tax_seq = SequentialLS(CRRAutility(), g=np.array([0.15]), π=np.ones((1, 1)))

n = 100
tax_policy = np.empty((n, 2))
interest_rate = np.empty((n, 2))
gov_debt = np.linspace(-1.5, 1, n)

for i in range(n):
    tax_policy[i] = tax_seq.simulate(gov_debt[i], 0, 2)[3]
    interest_rate[i] = tax_seq.simulate(gov_debt[i], 0, 3)[-1]

fig, axes = plt.subplots(2, 1, figsize=(10,8), sharex=True)
titles = ['Tax Rate', 'Gross Interest Rate']

for ax, title, plot in zip(axes, titles, [tax_policy, interest_rate]):
```

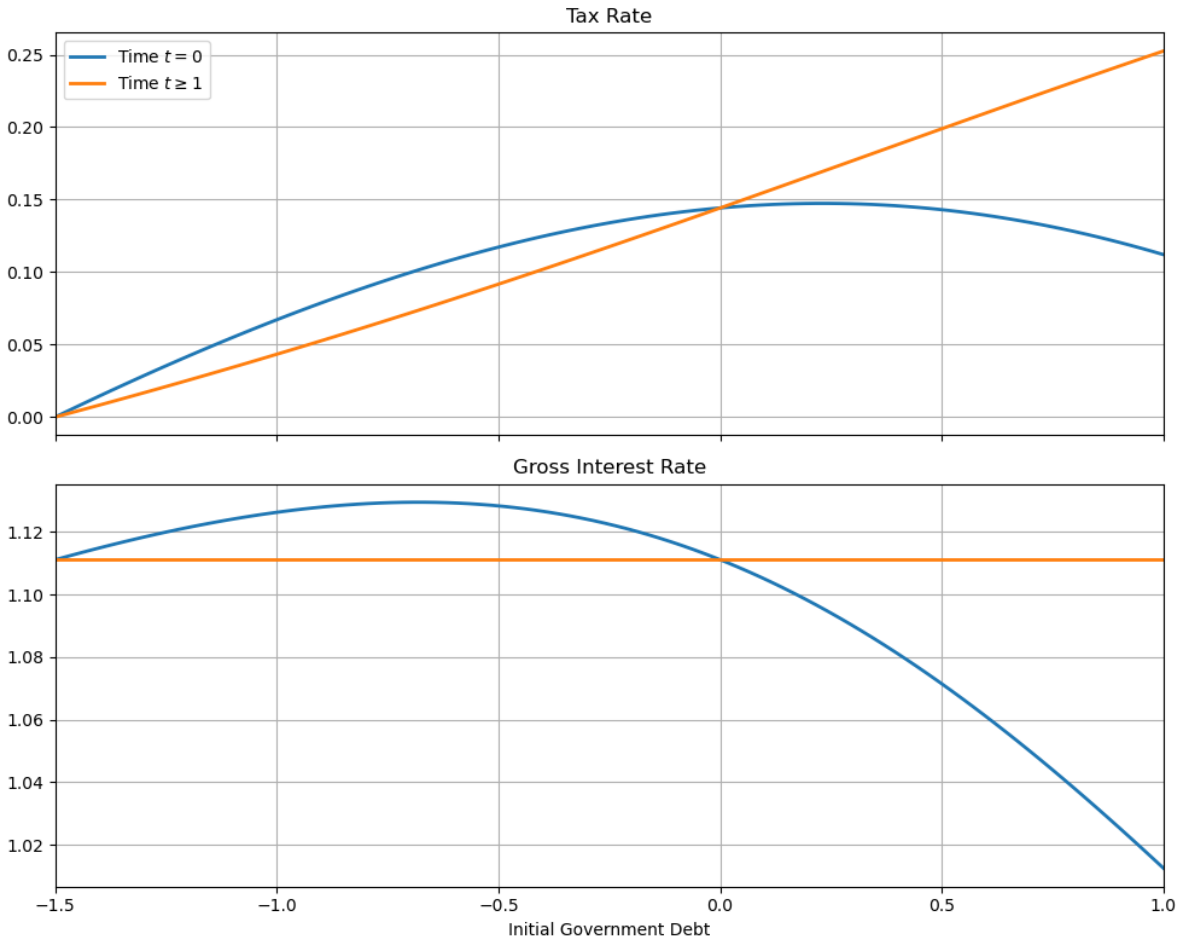
(continues on next page)

(continued from previous page)

```
    ax.plot(gov_debt, plot[:, 0], gov_debt, plot[:, 1], lw=2)
    ax.set(title=title, xlim=(min(gov_debt), max(gov_debt)))
    ax.grid()

axes[0].legend(('Time $t=0$', r'Time $t \geq 1$'))
axes[1].set_xlabel('Initial Government Debt')

fig.tight_layout()
plt.show()
```



The figure indicates that if the government enters with positive debt, it sets a tax rate at $t = 0$ that is less than all later tax rates.

By setting a lower tax rate at $t = 0$, the government raises consumption, which reduces the *value* $u_{c,0} b_0$ of its initial debt.

It does this by increasing c_0 and thereby lowering $u_{c,0}$.

Conversely, if $b_0 < 0$, the Ramsey planner sets the tax rate at $t = 0$ higher than in subsequent periods.

A side effect of lowering time $t = 0$ consumption is that it lowers the one-period interest rate at time $t = 0$ below that of subsequent periods.

There are only two values of initial government debt at which the tax rate is constant for all $t \geq 0$.

The first is $b_0 = 0$

- Here the government can't use the $t = 0$ tax rate to alter the value of the initial debt.

The second occurs when the government enters with sufficiently large assets that the Ramsey planner can achieve first best and sets $\tau_t = 0$ for all t.

It is only for these two values of initial government debt that the Ramsey plan is time-consistent.

Another way of saying this is that, except for these two values of initial government debt, a continuation of a Ramsey plan is not a Ramsey plan.

To illustrate this, consider a Ramsey planner who starts with an initial government debt b_1 associated with one of the Ramsey plans computed above.

Call τ_1^R the time $t = 0$ tax rate chosen by the Ramsey planner confronting this value for initial government debt government.

The figure below shows both the tax rate at time 1 chosen by our original Ramsey planner and what a new Ramsey planner would choose for its time $t = 0$ tax rate

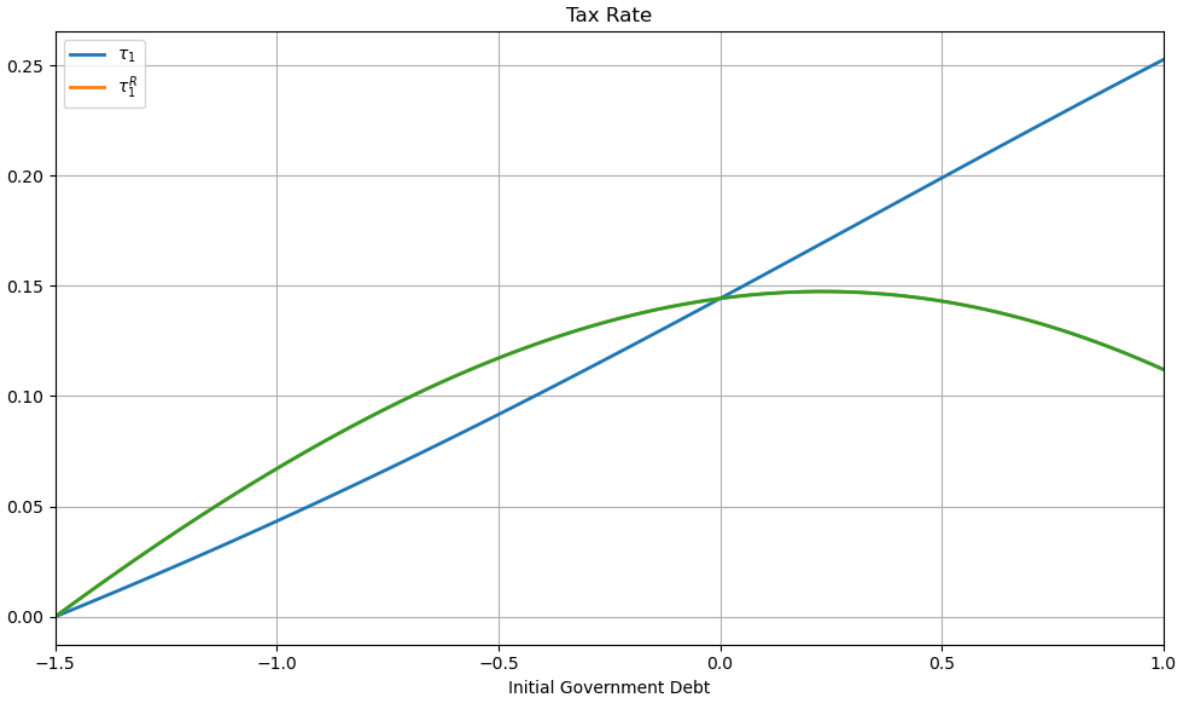
```
tax_seq = SequentialLS(CRRAutility(), g=np.array([0.15]), π=np.ones((1, 1)))

n = 100
tax_policy = np.empty((n, 2))
τ_reset = np.empty((n, 2))
gov_debt = np.linspace(-1.5, 1, n)

for i in range(n):
    tax_policy[i] = tax_seq.simulate(gov_debt[i], 0, 2)[3]
    τ_reset[i] = tax_seq.simulate(gov_debt[i], 0, 1)[3]

fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(gov_debt, tax_policy[:, 1], gov_debt, τ_reset, lw=2)
ax.set(xlabel='Initial Government Debt', title='Tax Rate',
       xlim=(min(gov_debt), max(gov_debt)))
ax.legend((r'$\tau_1$', r'$\tau_1^R$'))
ax.grid()

fig.tight_layout()
plt.show()
```



The tax rates in the figure are equal for only two values of initial government debt.

45.4.5 Tax Smoothing and non-CRRA Preferences

The complete tax smoothing for $t \geq 1$ in the preceding example is a consequence of our having assumed CRRA preferences.

To see what is driving this outcome, we begin by noting that the Ramsey tax rate for $t \geq 1$ is a time-invariant function $\tau(\Phi, g)$ of the Lagrange multiplier on the implementability constraint and government expenditures.

For CRRA preferences, we can exploit the relations $U_{cc}c = -\sigma U_c$ and $U_{nn}n = \gamma U_n$ to derive

$$\frac{(1 + (1-\sigma)\Phi)U_c}{(1 + (1-\gamma)\Phi)U_n} = 1$$

from the first-order conditions.

This equation immediately implies that the tax rate is constant.

For other preferences, the tax rate may not be constant.

For example, let the period utility function be

$$u(c, n) = \log(c) + 0.69 \log(1 - n)$$

We will create a new class LogUtility to represent this utility function

```
log_util_data = [
    ('β', float64),
    ('ψ', float64)
]

@jitclass(log_util_data)
```

(continues on next page)

(continued from previous page)

```
class LogUtility:

    def __init__(self,
                 β=0.9,
                 ψ=0.69):

        self.β, self.ψ = β, ψ

    # Utility function
    def U(self, c, l):
        return np.log(c) + self.ψ * np.log(l)

    # Derivatives of utility function
    def Uc(self, c, l):
        return 1 / c

    def Ucc(self, c, l):
        return -c**(-2)

    def Ul(self, c, l):
        return self.ψ / l

    def Ull(self, c, l):
        return -self.ψ / l**2

    def Ucl(self, c, l):
        return 0

    def Ulc(self, c, l):
        return 0
```

Also, suppose that g_t follows a two-state IID process with equal probabilities attached to g_l and g_h.

To compute the tax rate, we will use both the sequential and recursive approaches described above.

The figure below plots a sample path of the Ramsey tax rate

```
log_example = LogUtility()
# Solve sequential problem
seq_log = SequentialLS(log_example)

# Initialize grid for value function iteration and solve
x_grid = np.linspace(-3., 3., 200)

# Solve recursive problem
rec_log = RecursiveLS(log_example, x_grid)

T_length = 20
sHist = np.array([0, 0, 0, 0, 0,
                  0, 0, 0, 1, 1,
                  0, 0, 0, 1, 1,
                  1, 1, 1, 1, 0])

# Simulate
sim_seq = seq_log.simulate(0.5, 0, T_length, sHist)
sim_rec = rec_log.simulate(0.5, 0, T_length, sHist)
```

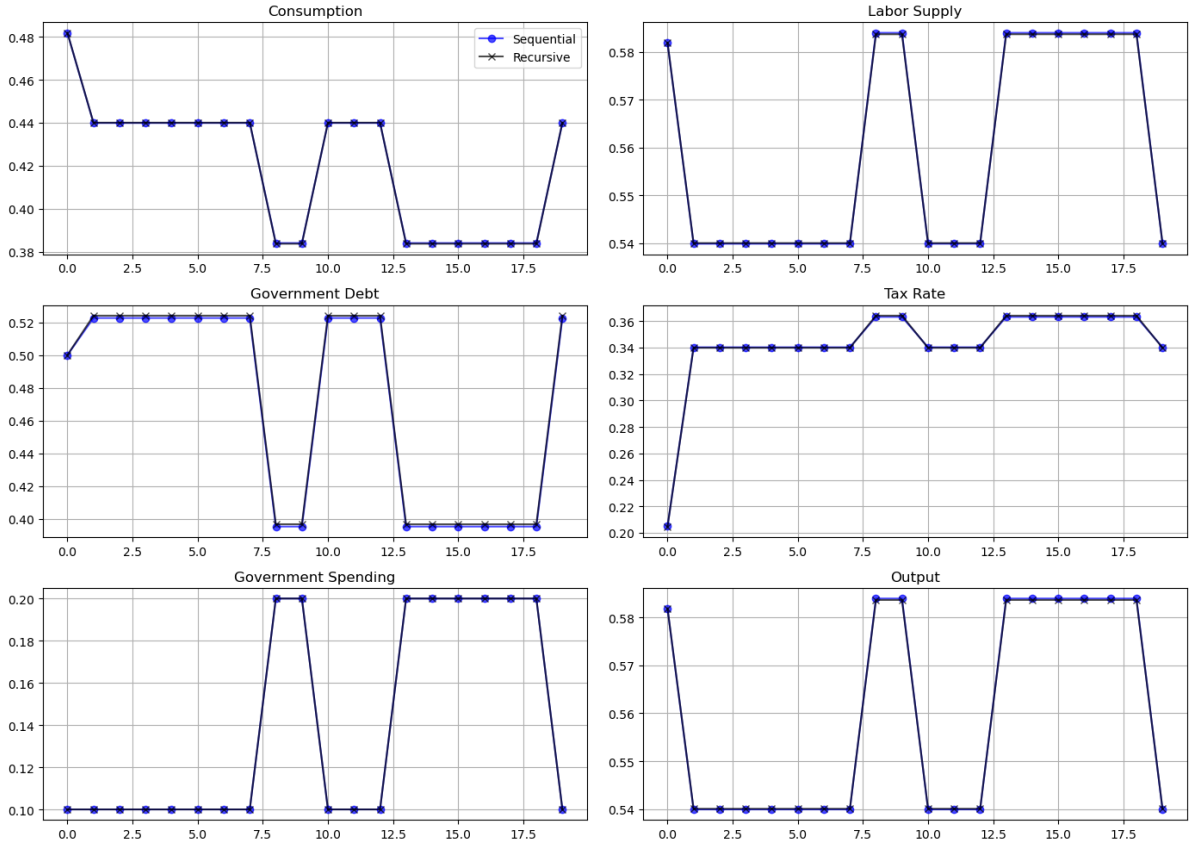
(continues on next page)

(continued from previous page)

```
fig, axes = plt.subplots(3, 2, figsize=(14, 10))
titles = ['Consumption', 'Labor Supply', 'Government Debt',
          'Tax Rate', 'Government Spending', 'Output']

for ax, title, sim_s, sim_b in zip(axes.flatten(), titles, sim_seq[:6], sim_rec[:6]):
                                    ax.plot(sim_s, '-ob', sim_b, '-xk', alpha=0.7)
                                    ax.set(title=title)
                                    ax.grid()

axes.flatten()[0].legend(('Sequential', 'Recursive'))
fig.tight_layout()
plt.show()
```



As should be expected, the recursive and sequential solutions produce almost identical allocations.

Unlike outcomes with CRRA preferences, the tax rate is not perfectly smoothed.

Instead, the government raises the tax rate when g_t is high.

45.4.6 Further Comments

A *related lecture* describes an extension of the Lucas-Stokey model by Aiyagari, Marcet, Sargent, and Seppälä (2002) [Aiyagari *et al.*, 2002].

In the AMSS economy, only a risk-free bond is traded.

That lecture compares the recursive representation of the Lucas-Stokey model presented in this lecture with one for an AMSS economy.

By comparing these recursive formulations, we shall glean a sense in which the dimension of the state is lower in the Lucas Stokey model.

Accompanying that difference in dimension will be different dynamics of government debt.

CHAPTER

FORTYSIX

OPTIMAL TAXATION WITHOUT STATE-CONTINGENT DEBT

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install --upgrade quantecon
!pip install interpolation
```

46.1 Overview

Let's start with following imports:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import root
from interpolation.splines import eval_linear, UCGrid, nodes
from quantecon import optimize, MarkovChain
from numba import njit, prange, float64
from numba.experimental import jitclass
```

In *an earlier lecture*, we described a model of optimal taxation with state-contingent debt due to Robert E. Lucas, Jr., and Nancy Stokey [Lucas and Stokey, 1983].

Aiyagari, Marcet, Sargent, and Seppälä [Aiyagari *et al.*, 2002] (hereafter, AMSS) studied optimal taxation in a model without state-contingent debt.

In this lecture, we

- describe assumptions and equilibrium concepts
- solve the model
- implement the model numerically
- conduct some policy experiments
- compare outcomes with those in a corresponding complete-markets model

We begin with an introduction to the model.

46.2 Competitive Equilibrium with Distorting Taxes

Many but not all features of the economy are identical to those of *the Lucas-Stokey economy*.

Let's start with things that are identical.

For $t \geq 0$, a history of the state is represented by $s^t = [s_t, s_{t-1}, \ldots, s_0]$.

Government purchases $g(s)$ are an exact time-invariant function of s.

Let $c_t(s^t)$, $\ell_t(s^t)$, and $n_t(s^t)$ denote consumption, leisure, and labor supply, respectively, at history s^t at time t.

Each period a representative household is endowed with one unit of time that can be divided between leisure ℓ_t and labor n_t:

$$n_t(s^t) + \ell_t(s^t) = 1 \tag{46.1}$$

Output equals $n_t(s^t)$ and can be divided between consumption $c_t(s^t)$ and $g(s_t)$

$$c_t(s^t) + g(s_t) = n_t(s^t) \tag{46.2}$$

Output is not storable.

The technology pins down a pre-tax wage rate to unity for all t, s^t.

A representative household's preferences over $\{c_t(s^t), \ell_t(s^t)\}_{t=0}^{\infty}$ are ordered by

$$\sum_{t=0}^{\infty} \sum_{s^t} \beta^t \pi_t(s^t) u[c_t(s^t), \ell_t(s^t)] \tag{46.3}$$

where

- $\pi_t(s^t)$ is a joint probability distribution over the sequence s^t, and
- the utility function u is increasing, strictly concave, and three times continuously differentiable in both arguments.

The government imposes a flat rate tax $\tau_t(s^t)$ on labor income at time t, history s^t.

Lucas and Stokey assumed that there are complete markets in one-period Arrow securities; also see *smoothing models*.

It is at this point that AMSS [Aiyagari *et al.*, 2002] modify the Lucas and Stokey economy.

AMSS allow the government to issue only one-period risk-free debt each period.

Ruling out complete markets in this way is a step in the direction of making total tax collections behave more like that prescribed in Robert Barro (1979) [Barro, 1979] than they do in Lucas and Stokey (1983) [Lucas and Stokey, 1983].

46.2.1 Risk-free One-Period Debt Only

In period t and history s^t, let

- $b_{t+1}(s^t)$ be the amount of the time $t+1$ consumption good that at time t, history s^t the government promised to pay
- $R_t(s^t)$ be the gross interest rate on risk-free one-period debt between periods t and $t+1$
- $T_t(s^t)$ be a non-negative lump-sum *transfer* to the representative household[1]

[1] In an allocation that solves the Ramsey problem and that levies distorting taxes on labor, why would the government ever want to hand revenues back to the private sector? It would not in an economy with state-contingent debt, since any such allocation could be improved by lowering distortionary taxes rather than handing out lump-sum transfers. But, without state-contingent debt there can be circumstances when a government would like to make lump-sum transfers to the private sector.

That $b_{t+1}(s^t)$ is the same for all realizations of s_{t+1} captures its *risk-free* character.

The market value at time t of government debt maturing at time $t+1$ equals $b_{t+1}(s^t)$ divided by $R_t(s^t)$.

The government's budget constraint in period t at history s^t is

$$\begin{aligned} b_t(s^{t-1}) &= \tau_t^n(s^t) n_t(s^t) - g(s_t) - T_t(s^t) + \frac{b_{t+1}(s^t)}{R_t(s^t)} \\ &\equiv z_t(s^t) + \frac{b_{t+1}(s^t)}{R_t(s^t)}, \end{aligned} \tag{46.4}$$

where $z_t(s^t)$ is the net-of-interest government surplus.

To rule out Ponzi schemes, we assume that the government is subject to a **natural debt limit** (to be discussed in a forthcoming lecture).

The consumption Euler equation for a representative household able to trade only one-period risk-free debt with one-period gross interest rate $R_t(s^t)$ is

$$\frac{1}{R_t(s^t)} = \sum_{s^{t+1}|s^t} \beta \pi_{t+1}(s^{t+1}|s^t) \frac{u_c(s^{t+1})}{u_c(s^t)}$$

Substituting this expression into the government's budget constraint (46.4) yields:

$$b_t(s^{t-1}) = z_t(s^t) + \beta \sum_{s^{t+1}|s^t} \pi_{t+1}(s^{t+1}|s^t) \frac{u_c(s^{t+1})}{u_c(s^t)} \, b_{t+1}(s^t) \tag{46.5}$$

Components of $z_t(s^t)$ on the right side depend on s^t, but the left side is required to depend only on s^{t-1} .

This is what it means for one-period government debt to be risk-free.

Therefore, the right side of equation (46.5) also has to depend only on s^{t-1}.

This requirement will give rise to **measurability constraints** on the Ramsey allocation to be discussed soon.

If we replace $b_{t+1}(s^t)$ on the right side of equation (46.5) by the right side of next period's budget constraint (associated with a particular realization s_t) we get

$$b_t(s^{t-1}) = z_t(s^t) + \sum_{s^{t+1}|s^t} \beta \pi_{t+1}(s^{t+1}|s^t) \frac{u_c(s^{t+1})}{u_c(s^t)} \left[z_{t+1}(s^{t+1}) + \frac{b_{t+2}(s^{t+1})}{R_{t+1}(s^{t+1})} \right]$$

After making similar repeated substitutions for all future occurrences of government indebtedness, and by invoking a natural debt limit, we arrive at:

$$b_t(s^{t-1}) = \sum_{j=0}^{\infty} \sum_{s^{t+j}|s^t} \beta^j \pi_{t+j}(s^{t+j}|s^t) \frac{u_c(s^{t+j})}{u_c(s^t)} \, z_{t+j}(s^{t+j}) \tag{46.6}$$

Notice how the conditioning sets in equation (46.6) differ: they are s^{t-1} on the left side and s^t on the right side.

Now let's

- substitute the resource constraint into the net-of-interest government surplus, and
- use the household's first-order condition $1 - \tau_t^n(s^t) = u_\ell(s^t)/u_c(s^t)$ to eliminate the labor tax rate

so that we can express the net-of-interest government surplus $z_t(s^t)$ as

$$z_t(s^t) = \left[1 - \frac{u_\ell(s^t)}{u_c(s^t)} \right] [c_t(s^t) + g(s_t)] - g(s_t) - T_t(s^t) \, . \tag{46.7}$$

If we substitute appropriate versions of the right side of (46.7) for $z_{t+j}(s^{t+j})$ into equation (46.6), we obtain a sequence of *implementability constraints* on a Ramsey allocation in an AMSS economy.

Expression (46.6) at time $t = 0$ and initial state s^0 was also an *implementability constraint* on a Ramsey allocation in a Lucas-Stokey economy:

$$b_0(s^{-1}) = \mathbb{E}_0 \sum_{j=0}^{\infty} \beta^j \frac{u_c(s^j)}{u_c(s^0)} \, z_j(s^j) \tag{46.8}$$

Indeed, it was the *only* implementability constraint there.

But now we also have a large number of additional implementability constraints

$$b_t(s^{t-1}) = \mathbb{E}_t \sum_{j=0}^{\infty} \beta^j \frac{u_c(s^{t+j})}{u_c(s^t)} \, z_{t+j}(s^{t+j}) \tag{46.9}$$

Equation (46.9) must hold for each s^t for each $t \geq 1$.

46.2.2 Comparison with Lucas-Stokey Economy

The expression on the right side of (46.9) in the Lucas-Stokey (1983) economy would equal the present value of a continuation stream of government net-of-interest surpluses evaluated at what would be competitive equilibrium Arrow-Debreu prices at date t.

In the Lucas-Stokey economy, that present value is measurable with respect to s^t.

In the AMSS economy, the restriction that government debt be risk-free imposes that that same present value must be measurable with respect to s^{t-1}.

In a language used in the literature on incomplete markets models, it can be said that the AMSS model requires that at each (t, s^t) what would be the present value of continuation government net-of-interest surpluses in the Lucas-Stokey model must belong to the **marketable subspace** of the AMSS model.

46.2.3 Ramsey Problem Without State-contingent Debt

After we have substituted the resource constraint into the utility function, we can express the Ramsey problem as being to choose an allocation that solves

$$\max_{\{c_t(s^t), b_{t+1}(s^t)\}} \mathbb{E}_0 \sum_{t=0}^{\infty} \beta^t u\left(c_t(s^t), 1 - c_t(s^t) - g(s_t)\right)$$

where the maximization is subject to

$$\mathbb{E}_0 \sum_{j=0}^{\infty} \beta^j \frac{u_c(s^j)}{u_c(s^0)} \, z_j(s^j) \geq b_0(s^{-1}) \tag{46.10}$$

and

$$\mathbb{E}_t \sum_{j=0}^{\infty} \beta^j \frac{u_c(s^{t+j})}{u_c(s^t)} \, z_{t+j}(s^{t+j}) = b_t(s^{t-1}) \quad \forall \, t, s^t \tag{46.11}$$

given $b_0(s^{-1})$.

Lagrangian Formulation

Let $\gamma_0(s^0)$ be a non-negative Lagrange multiplier on constraint (46.10).

As in the Lucas-Stokey economy, this multiplier is strictly positive when the government must resort to distortionary taxation; otherwise it equals zero.

A consequence of the assumption that there are no markets in state-contingent securities and that a market exists only in a risk-free security is that we have to attach a stochastic process $\{\gamma_t(s^t)\}_{t=1}^{\infty}$ of Lagrange multipliers to the implementability constraints (46.11).

Depending on how the constraints bind, these multipliers can be positive or negative:

$$\begin{aligned}\gamma_t(s^t) \;\geq\; (\leq)\; 0 \quad &\text{if the constraint binds in the following direction} \\ \mathbb{E}_t \sum_{j=0}^{\infty} \beta^j \frac{u_c(s^{t+j})}{u_c(s^t)}\, z_{t+j}(s^{t+j}) \;&\geq\; (\leq)\; b_t(s^{t-1})\end{aligned}$$

A negative multiplier $\gamma_t(s^t) < 0$ means that if we could relax constraint (46.11), we would like to *increase* the beginning-of-period indebtedness for that particular realization of history s^t.

That would let us reduce the beginning-of-period indebtedness for some other history[2].

These features flow from the fact that the government cannot use state-contingent debt and therefore cannot allocate its indebtedness efficiently across future states.

46.2.4 Some Calculations

It is helpful to apply two transformations to the Lagrangian.

Multiply constraint (46.10) by $u_c(s^0)$ and the constraints (46.11) by $\beta^t u_c(s^t)$.

Then a Lagrangian for the Ramsey problem can be represented as

$$\begin{aligned} J &= \mathbb{E}_0 \sum_{t=0}^{\infty} \beta^t \Big\{ u\left(c_t(s^t), 1 - c_t(s^t) - g(s_t)\right) \\ &\qquad + \gamma_t(s^t) \Big[\mathbb{E}_t \sum_{j=0}^{\infty} \beta^j u_c(s^{t+j})\, z_{t+j}(s^{t+j}) - u_c(s^t)\, b_t(s^{t-1}) \Big] \Big\} \\ &= \mathbb{E}_0 \sum_{t=0}^{\infty} \beta^t \Big\{ u\left(c_t(s^t), 1 - c_t(s^t) - g(s_t)\right) \\ &\qquad + \Psi_t(s^t)\, u_c(s^t)\, z_t(s^t) - \gamma_t(s^t)\, u_c(s^t)\, b_t(s^{t-1}) \Big\} \end{aligned} \tag{46.12}$$

where

$$\Psi_t(s^t) = \Psi_{t-1}(s^{t-1}) + \gamma_t(s^t) \quad \text{and} \quad \Psi_{-1}(s^{-1}) = 0 \tag{46.13}$$

In (46.12), the second equality uses the law of iterated expectations and Abel's summation formula (also called *summation by parts*, see this page).

First-order conditions with respect to $c_t(s^t)$ can be expressed as

$$\begin{aligned} u_c(s^t) - u_\ell(s^t) + \Psi_t(s^t) \left\{ \left[u_{cc}(s^t) - u_{c\ell}(s^t) \right] z_t(s^t) + u_c(s^t)\, z_c(s^t) \right\} \\ - \gamma_t(s^t) \left[u_{cc}(s^t) - u_{c\ell}(s^t) \right] b_t(s^{t-1}) = 0 \end{aligned} \tag{46.14}$$

[2] From the first-order conditions for the Ramsey problem, there exists another realization $\tilde{s}^t$ with the same history up until the previous period, i.e., $\tilde{s}^{t-1} = s^{t-1}$, but where the multiplier on constraint (46.11) takes a positive value, so $\gamma_t(\tilde{s}^t) > 0$.

and with respect to $b_t(s^t)$ as

$$\mathbb{E}_t \left[\gamma_{t+1}(s^{t+1})\, u_c(s^{t+1})\right] = 0 \tag{46.15}$$

If we substitute $z_t(s^t)$ from (46.7) and its derivative $z_c(s^t)$ into the first-order condition (46.14), we find two differences from the corresponding condition for the optimal allocation in a Lucas-Stokey economy with state-contingent government debt.

1. The term involving $b_t(s^{t-1})$ in the first-order condition (46.14) does not appear in the corresponding expression for the Lucas-Stokey economy.
    - This term reflects the constraint that beginning-of-period government indebtedness must be the same across all realizations of next period's state, a constraint that would not be present if government debt could be state-contingent.
2. The Lagrange multiplier $\Psi_t(s^t)$ in the first-order condition (46.14) may change over time in response to realizations of the state, while the multiplier Φ in the Lucas-Stokey economy is time-invariant.

We need some code from *an earlier lecture* on optimal taxation with state-contingent debt sequential allocation implementation:

```
class SequentialLS:

    '''
    Class that takes a preference object, state transition matrix,
    and state contingent government expenditure plan as inputs, and
    solves the sequential allocation problem described above.
    It returns optimal allocations about consumption and labor supply,
    as well as the multiplier on the implementability constraint Φ.
    '''

    def __init__(self,
                 pref,
                 π=np.full((2, 2), 0.5),
                 g=np.array([0.1, 0.2])):

        # Initialize from pref object attributes
        self.β, self.π, self.g = pref.β, π, g
        self.mc = MarkovChain(self.π)
        self.S = len(π)  # Number of states
        self.pref = pref

        # Find the first best allocation
        self.find_first_best()

    def FOC_first_best(self, c, g):
        '''
        First order conditions that characterize
        the first best allocation.
        '''

        pref = self.pref
        Uc, Ul = pref.Uc, pref.Ul

        n = c + g
        l = 1 - n

        return Uc(c, l) - Ul(c, l)
```

(continues on next page)

(continued from previous page)

```
    def find_first_best(self):
        '''
        Find the first best allocation
        '''
        S, g = self.S, self.g

        res = root(self.FOC_first_best, np.full(S, 0.5), args=(g,))

        if (res.fun > 1e-10).any():
            raise Exception('Could not find first best')

        self.cFB = res.x
        self.nFB = self.cFB + g

    def FOC_time1(self, c, Φ, g):
        '''
        First order conditions that characterize
        optimal time 1 allocation problems.
        '''

        pref = self.pref
        Uc, Ucc, Ul, Ull, Ulc = pref.Uc, pref.Ucc, pref.Ul, pref.Ull, pref.Ulc

        n = c + g
        l = 1 - n

        LHS = (1 + Φ) * Uc(c, l) + Φ * (c * Ucc(c, l) - n * Ulc(c, l))
        RHS = (1 + Φ) * Ul(c, l) + Φ * (c * Ulc(c, l) - n * Ull(c, l))

        diff = LHS - RHS

        return diff

    def time1_allocation(self, Φ):
        '''
        Computes optimal allocation for time t >= 1 for a given Φ
        '''
        pref = self.pref
        S, g = self.S, self.g

        # use the first best allocation as intial guess
        res = root(self.FOC_time1, self.cFB, args=(Φ, g))

        if (res.fun > 1e-10).any():
            raise Exception('Could not find LS allocation.')

        c = res.x
        n = c + g
        l = 1 - n

        # Compute x
        I = pref.Uc(c, n) * c - pref.Ul(c, l) * n
        x = np.linalg.solve(np.eye(S) - self.β * self.π, I)

        return c, n, x
```

(continues on next page)

(continued from previous page)

```
    def FOC_time0(self, c0, Φ, g0, b0):
        '''
        First order conditions that characterize
        time 0 allocation problem.
        '''

        pref = self.pref
        Ucc, Ulc = pref.Ucc, pref.Ulc

        n0 = c0 + g0
        l0 = 1 - n0

        diff = self.FOC_time1(c0, Φ, g0)
        diff -= Φ * (Ucc(c0, l0) - Ulc(c0, l0)) * b0

        return diff

    def implementability(self, Φ, b0, s0, cn0_arr):
        '''
        Compute the differences between the RHS and LHS
        of the implementability constraint given Φ,
        initial debt, and initial state.
        '''

        pref, π, g, β = self.pref, self.π, self.g, self.β
        Uc, Ul = pref.Uc, pref.Ul
        g0 = self.g[s0]

        c, n, x = self.time1_allocation(Φ)

        res = root(self.FOC_time0, cn0_arr[0], args=(Φ, g0, b0))
        c0 = res.x
        n0 = c0 + g0
        l0 = 1 - n0

        cn0_arr[:] = c0.item(), n0.item()

        LHS = Uc(c0, l0) * b0
        RHS = Uc(c0, l0) * c0 - Ul(c0, l0) * n0 + β * π[s0] @ x

        return RHS - LHS

    def time0_allocation(self, b0, s0):
        '''
        Finds the optimal time 0 allocation given
        initial government debt b0 and state s0
        '''

        # use the first best allocation as initial guess
        cn0_arr = np.array([self.cFB[s0], self.nFB[s0]])

        res = root(self.implementability, 0., args=(b0, s0, cn0_arr))

        if (res.fun > 1e-10).any():
            raise Exception('Could not find time 0 LS allocation.')
```

(continues on next page)

(continued from previous page)

```
        Φ = res.x[0]
        c0, n0 = cn0_arr

        return Φ, c0, n0

    def τ(self, c, n):
        '''
        Computes τ given c, n
        '''
        pref = self.pref
        Uc, Ul = pref.Uc, pref.Ul

        return 1 - Ul(c, 1-n) / Uc(c, 1-n)

    def simulate(self, b0, s0, T, sHist=None):
        '''
        Simulates planners policies for T periods
        '''
        pref, π, β = self.pref, self.π, self.β
        Uc = pref.Uc

        if sHist is None:
            sHist = self.mc.simulate(T, s0)

        cHist, nHist, Bhist, τHist, ΦHist = np.empty((5, T))
        RHist = np.empty(T-1)

        # Time 0
        Φ, cHist[0], nHist[0] = self.time0_allocation(b0, s0)
        τHist[0] = self.τ(cHist[0], nHist[0])
        Bhist[0] = b0
        ΦHist[0] = Φ

        # Time 1 onward
        for t in range(1, T):
            c, n, x = self.time1_allocation(Φ)
            τ = self.τ(c, n)
            u_c = Uc(c, 1-n)
            s = sHist[t]
            Eu_c = π[sHist[t-1]] @ u_c
            cHist[t], nHist[t], Bhist[t], τHist[t] = c[s], n[s], x[s] / u_c[s], τ[s]
            RHist[t-1] = Uc(cHist[t-1], 1-nHist[t-1]) / (β * Eu_c)
            ΦHist[t] = Φ

        gHist = self.g[sHist]
        yHist = nHist

        return [cHist, nHist, Bhist, τHist, gHist, yHist, sHist, ΦHist, RHist]
```

To analyze the AMSS model, we find it useful to adopt a recursive formulation using techniques like those in our lectures on *dynamic Stackelberg models* and *optimal taxation with state-contingent debt*.

46.3 Recursive Version of AMSS Model

We now describe a recursive formulation of the AMSS economy.

We have noted that from the point of view of the Ramsey planner, the restriction to one-period risk-free securities

- leaves intact the single implementability constraint on allocations (46.8) from the Lucas-Stokey economy, but
- adds measurability constraints (46.6) on functions of tails of allocations at each time and history

We now explore how these constraints alter Bellman equations for a time 0 Ramsey planner and for time $t \geq 1$, history s^t continuation Ramsey planners.

46.3.1 Recasting State Variables

In the AMSS setting, the government faces a sequence of budget constraints

$$\tau_t(s^t)n_t(s^t) + T_t(s^t) + b_{t+1}(s^t)/R_t(s^t) = g_t + b_t(s^{t-1})$$

where $R_t(s^t)$ is the gross risk-free rate of interest between t and $t+1$ at history s^t and $T_t(s^t)$ are non-negative transfers.

Throughout this lecture, we shall set transfers to zero (for some issues about the limiting behavior of debt, this is possibly an important difference from AMSS [Aiyagari *et al.*, 2002], who restricted transfers to be non-negative).

In this case, the household faces a sequence of budget constraints

$$b_t(s^{t-1}) + (1-\tau_t(s^t))n_t(s^t) = c_t(s^t) + b_{t+1}(s^t)/R_t(s^t) \tag{46.16}$$

The household's first-order conditions are $u_{c,t} = \beta R_t \mathbb{E}_t u_{c,t+1}$ and $(1-\tau_t)u_{c,t} = u_{l,t}$.

Using these to eliminate R_t and τ_t from budget constraint (46.16) gives

$$b_t(s^{t-1}) + \frac{u_{l,t}(s^t)}{u_{c,t}(s^t)}n_t(s^t) = c_t(s^t) + \frac{\beta(\mathbb{E}_t u_{c,t+1})b_{t+1}(s^t)}{u_{c,t}(s^t)} \tag{46.17}$$

or

$$u_{c,t}(s^t)b_t(s^{t-1}) + u_{l,t}(s^t)n_t(s^t) = u_{c,t}(s^t)c_t(s^t) + \beta(\mathbb{E}_t u_{c,t+1})b_{t+1}(s^t) \tag{46.18}$$

Now define

$$x_t \equiv \beta b_{t+1}(s^t)\mathbb{E}_t u_{c,t+1} = u_{c,t}(s^t)\frac{b_{t+1}(s^t)}{R_t(s^t)} \tag{46.19}$$

and represent the household's budget constraint at time t, history s^t as

$$\frac{u_{c,t}x_{t-1}}{\beta \mathbb{E}_{t-1}u_{c,t}} = u_{c,t}c_t - u_{l,t}n_t + x_t \tag{46.20}$$

for $t \geq 1$.

46.3.2 Measurability Constraints

Write equation (46.18) as

$$b_t(s^{t-1}) = c_t(s^t) - \frac{u_{l,t}(s^t)}{u_{c,t}(s^t)}n_t(s^t) + \frac{\beta(\mathbb{E}_t u_{c,t+1})b_{t+1}(s^t)}{u_{c,t}} \tag{46.21}$$

The right side of equation (46.21) expresses the time t value of government debt in terms of a linear combination of terms whose individual components are measurable with respect to s^t.

The sum of terms on the right side of equation (46.21) must equal $b_t(s^{t-1})$.

That implies that it has to be *measurable* with respect to s^{t-1}.

Equations (46.21) are the *measurability constraints* that the AMSS model adds to the single time 0 implementation constraint imposed in the Lucas and Stokey model.

46.3.3 Two Bellman Equations

Let $\Pi(s|s_-)$ be a Markov transition matrix whose entries tell probabilities of moving from state s_- to state s in one period.

Let

- $V(x_-, s_-)$ be the continuation value of a continuation Ramsey plan at $x_{t-1} = x_-, s_{t-1} = s_-$ for $t \geq 1$
- $W(b, s)$ be the value of the Ramsey plan at time 0 at $b_0 = b$ and $s_0 = s$

We distinguish between two types of planners:

For $t \geq 1$, the value function for a **continuation Ramsey planner** satisfies the Bellman equation

$$V(x_-, s_-) = \max_{\{n(s), x(s)\}} \sum_s \Pi(s|s_-) \left[u(n(s) - g(s), 1 - n(s)) + \beta V(x(s), s)\right] \tag{46.22}$$

subject to the following collection of implementability constraints, one for each $s \in S$:

$$\frac{u_c(s) x_-}{\beta \sum_{\tilde{s}} \Pi(\tilde{s}|s_-) u_c(\tilde{s})} = u_c(s)(n(s) - g(s)) - u_l(s) n(s) + x(s) \tag{46.23}$$

A continuation Ramsey planner at $t \geq 1$ takes $(x_{t-1}, s_{t-1}) = (x_-, s_-)$ as given and before s is realized chooses $(n_t(s_t), x_t(s_t)) = (n(s), x(s))$ for $s \in S$.

The **Ramsey planner** takes (b_0, s_0) as given and chooses (n_0, x_0).

The value function $W(b_0, s_0)$ for the time $t = 0$ Ramsey planner satisfies the Bellman equation

$$W(b_0, s_0) = \max_{n_0, x_0} u(n_0 - g_0, 1 - n_0) + \beta V(x_0, s_0) \tag{46.24}$$

where maximization is subject to

$$u_{c,0} b_0 = u_{c,0}(n_0 - g_0) - u_{l,0} n_0 + x_0 \tag{46.25}$$

46.3.4 Martingale Supercedes State-Variable Degeneracy

Let $\mu(s|s_-)\Pi(s|s_-)$ be a Lagrange multiplier on the constraint (46.23) for state s.

After forming an appropriate Lagrangian, we find that the continuation Ramsey planner's first-order condition with respect to $x(s)$ is

$$\beta V_x(x(s), s) = \mu(s|s_-) \tag{46.26}$$

Applying an envelope theorem to Bellman equation (46.22) gives

$$V_x(x_-, s_-) = \sum_s \Pi(s|s_-) \mu(s|s_-) \frac{u_c(s)}{\beta \sum_{\tilde{s}} \Pi(\tilde{s}|s_-) u_c(\tilde{s})} \tag{46.27}$$

Equations (46.26) and (46.27) imply that

$$V_x(x_-, s_-) = \sum_s \left(\Pi(s|s_-) \frac{u_c(s)}{\sum_{\tilde{s}} \Pi(\tilde{s}|s_-) u_c(\tilde{s})} \right) V_x(x, s) \tag{46.28}$$

Equation (46.28) states that $V_x(x, s)$ is a *risk-adjusted martingale*.

Saying that $V_x(x, s)$ is a risk-adjusted martingale means that $V_x(x, s)$ is a martingale with respect to the probability distribution over s^t sequences that are generated by the *twisted* transition probability matrix:

$$\breve{\Pi}(s|s_-) \equiv \Pi(s|s_-) \frac{u_c(s)}{\sum_{\tilde{s}} \Pi(\tilde{s}|s_-) u_c(\tilde{s})}$$

Exercise 46.3.1

Please verify that $\breve{\Pi}(s|s_-)$ is a valid Markov transition density, i.e., that its elements are all non-negative and that for each s_-, the sum over s equals unity.

46.3.5 Absence of State Variable Degeneracy

Along a Ramsey plan, the state variable $x_t = x_t(s^t, b_0)$ becomes a function of the history s^t and initial government debt b_0.

In *Lucas-Stokey model*, we found that

- a counterpart to $V_x(x, s)$ is time-invariant and equal to the Lagrange multiplier on the Lucas-Stokey implementability constraint
- time invariance of $V_x(x, s)$ is the source of a key feature of the Lucas-Stokey model, namely, **state variable degeneracy** in which x_t is an exact time-invariant function of s_t.

That $V_x(x, s)$ varies over time according to a twisted martingale means that there is no state-variable degeneracy in the AMSS model.

In the AMSS model, both x and s are needed to describe the state.

This property of the AMSS model transmits a twisted martingale component to consumption, employment, and the tax rate.

46.3.6 Digression on Non-negative Transfers

Throughout this lecture, we have imposed that transfers $T_t = 0$.

AMSS [Aiyagari *et al.*, 2002] instead imposed a nonnegativity constraint $T_t \geq 0$ on transfers.

They also considered a special case of quasi-linear preferences, $u(c, l) = c + H(l)$.

In this case, $V_x(x, s) \leq 0$ is a non-positive martingale.

By the *martingale convergence theorem* $V_x(x, s)$ converges almost surely.

Furthermore, when the Markov chain $\Pi(s|s_-)$ and the government expenditure function $g(s)$ are such that g_t is perpetually random, $V_x(x, s)$ almost surely converges to zero.

For quasi-linear preferences, the first-order condition for maximizing (46.22) subject to (46.23) with respect to $n(s)$ becomes

$$(1 - \mu(s|s_-))(1 - u_l(s)) + \mu(s|s_-)n(s)u_{ll}(s) = 0$$

When $\mu(s|s_-) = \beta V_x(x(s), x)$ converges to zero, in the limit $u_l(s) = 1 = u_c(s)$, so that $\tau(x(s), s) = 0$.

Thus, in the limit, if g_t is perpetually random, the government accumulates sufficient assets to finance all expenditures from earnings on those assets, returning any excess revenues to the household as non-negative lump-sum transfers.

46.3.7 Code

The recursive formulation is implemented as follows

```
class AMSS:
    # WARNING: THE CODE IS EXTREMELY SENSITIVE TO CHOCIES OF PARAMETERS.
    # DO NOT CHANGE THE PARAMETERS AND EXPECT IT TO WORK

    def __init__(self, pref, β, Π, g, x_grid, bounds_v):
        self.β, self.Π, self.g = β, Π, g
        self.x_grid = x_grid
        self.n = x_grid[0][2]
        self.S = len(Π)
        self.bounds = bounds_v
        self.pref = pref

        self.T_v, self.T_w = bellman_operator_factory(Π, β, x_grid, g,
                                                      bounds_v)

        self.V_solved = False
        self.W_solved = False

    def compute_V(self, V, σ_v_star, tol_vfi, maxitr, print_itr):

        T_v = self.T_v

        self.success = False

        V_new = np.zeros_like(V)

        Δ = 1.0
        for itr in range(maxitr):
            T_v(V, V_new, σ_v_star, self.pref)

            Δ = np.max(np.abs(V_new - V))

            if Δ < tol_vfi:
                self.V_solved = True
                print('Successfully completed VFI after %i iterations'
                      % (itr+1))
                break

            if (itr + 1) % print_itr == 0:
                print('Error at iteration %i : ' % (itr + 1), Δ)

            V[:] = V_new[:]
```

(continues on next page)

(continued from previous page)

```
        self.V = V
        self.σ_v_star = σ_v_star

        return V, σ_v_star

    def compute_W(self, b_0, W, σ_w_star):
        T_w = self.T_w
        V = self.V

        T_w(W, σ_w_star, V, b_0, self.pref)

        self.W = W
        self.σ_w_star = σ_w_star
        self.W_solved = True
        print('Succesfully solved the time 0 problem.')

        return W, σ_w_star

    def solve(self, V, σ_v_star,  b_0, W, σ_w_star, tol_vfi=1e-7,
              maxitr=1000, print_itr=10):
        print("===============")
        print("Solve time 1 problem")
        print("===============")
        self.compute_V(V, σ_v_star, tol_vfi, maxitr, print_itr)
        print("===============")
        print("Solve time 0 problem")
        print("===============")
        self.compute_W(b_0, W, σ_w_star)

    def simulate(self, s_hist, b_0):
        if not (self.V_solved and self.W_solved):
            msg = "V and W need to be successfully computed before simulation."
            raise ValueError(msg)

        pref = self.pref
        x_grid, g, β, S = self.x_grid, self.g, self.β, self.S
        σ_v_star, σ_w_star = self.σ_v_star, self.σ_w_star

        T = len(s_hist)
        s_0 = s_hist[0]

        # Pre-allocate
        n_hist = np.zeros(T)
        x_hist = np.zeros(T)
        c_hist = np.zeros(T)
        τ_hist = np.zeros(T)
        b_hist = np.zeros(T)
        g_hist = np.zeros(T)

        # Compute t = 0
        l_0, T_0 = σ_w_star[s_0]
        c_0 = (1 - l_0) - g[s_0]
        x_0 = (-pref.Uc(c_0, l_0) * (c_0 - T_0 - b_0) +
               pref.Ul(c_0, l_0) * (1 - l_0))
```

(continues on next page)

(continued from previous page)

```
        n_hist[0] = (1 - l_0)
        x_hist[0] = x_0
        c_hist[0] = c_0
        τ_hist[0] = 1 - pref.Ul(c_0, l_0) / pref.Uc(c_0, l_0)
        b_hist[0] = b_0
        g_hist[0] = g[s_0]

        # Compute t > 0
        for t in range(T - 1):
            x_ = x_hist[t]
            s_ = s_hist[t]
            l = np.zeros(S)
            T = np.zeros(S)
            for s in range(S):
                x_arr = np.array([x_])
                l[s] = eval_linear(x_grid, σ_v_star[s_, :, s], x_arr)
                T[s] = eval_linear(x_grid, σ_v_star[s_, :, S+s], x_arr)

            c = (1 - l) - g
            u_c = pref.Uc(c, l)
            Eu_c = Π[s_] @ u_c

            x = u_c * x_ / (β * Eu_c) - u_c * (c - T) + pref.Ul(c, l) * (1 - l)

            c_next = c[s_hist[t+1]]
            l_next = l[s_hist[t+1]]

            x_hist[t+1] = x[s_hist[t+1]]
            n_hist[t+1] = 1 - l_next
            c_hist[t+1] = c_next
            τ_hist[t+1] = 1 - pref.Ul(c_next, l_next) / pref.Uc(c_next, l_next)
            b_hist[t+1] = x_ / (β * Eu_c)
            g_hist[t+1] = g[s_hist[t+1]]

        return c_hist, n_hist, b_hist, τ_hist, g_hist, n_hist


def obj_factory(Π, β, x_grid, g):
    S = len(Π)

    @njit
    def obj_V(σ, state, V, pref):
        # Unpack state
        s_, x_ = state

        l = σ[:S]
        T = σ[S:]

        c = (1 - l) - g
        u_c = pref.Uc(c, l)
        Eu_c = Π[s_] @ u_c
        x = u_c * x_ / (β * Eu_c) - u_c * (c - T) + pref.Ul(c, l) * (1 - l)

        V_next = np.zeros(S)

        for s in range(S):
```

(continues on next page)

(continued from previous page)

```
            V_next[s] = eval_linear(x_grid, V[s], np.array([x[s]]))

        out = Π[s_] @ (pref.U(c, l) + β * V_next)

        return out

    @njit
    def obj_W(σ, state, V, pref):
        # Unpack state
        s_, b_0 = state
        l, T = σ

        c = (1 - l) - g[s_]
        x = -pref.Uc(c, l) * (c - T - b_0) + pref.Ul(c, l) * (1 - l)

        V_next = eval_linear(x_grid, V[s_], np.array([x]))

        out = pref.U(c, l) + β * V_next

        return out

    return obj_V, obj_W


def bellman_operator_factory(Π, β, x_grid, g, bounds_v):
    obj_V, obj_W = obj_factory(Π, β, x_grid, g)
    n = x_grid[0][2]
    S = len(Π)
    x_nodes = nodes(x_grid)

    @njit(parallel=True)
    def T_v(V, V_new, σ_star, pref):
        for s_ in prange(S):
            for x_i in prange(n):
                state = (s_, x_nodes[x_i])
                x0 = σ_star[s_, x_i]
                res = optimize.nelder_mead(obj_V, x0, bounds=bounds_v,
                                           args=(state, V, pref))

                if res.success:
                    V_new[s_, x_i] = res.fun
                    σ_star[s_, x_i] = res.x
                else:
                    print("Optimization routine failed.")

    bounds_w = np.array([[-9.0, 1.0], [0., 10.]])

    def T_w(W, σ_star, V, b_0, pref):
        for s_ in prange(S):
            state = (s_, b_0)
            x0 = σ_star[s_]
            res = optimize.nelder_mead(obj_W, x0, bounds=bounds_w,
                                       args=(state, V, pref))

            W[s_] = res.fun
            σ_star[s_] = res.x
```

(continues on next page)

(continued from previous page)

```
    return T_v, T_w
```

46.4 Examples

We now turn to some examples.

46.4.1 Anticipated One-Period War

In our lecture on *optimal taxation with state-contingent debt* we studied how the government manages uncertainty in a simple setting.

As in that lecture, we assume the one-period utility function

$$u(c, n) = \frac{c^{1-\sigma}}{1-\sigma} - \frac{n^{1+\gamma}}{1+\gamma}$$

Note: For convenience in matching our computer code, we have expressed utility as a function of n rather than leisure l.

We first consider a government expenditure process that we studied earlier in a lecture on *optimal taxation with state-contingent debt*.

Government expenditures are known for sure in all periods except one.

- For $t < 3$ or $t > 3$ we assume that $g_t = g_l = 0.1$.
- At $t = 3$ a war occurs with probability 0.5.
  - If there is war, $g_3 = g_h = 0.2$.
  - If there is no war $g_3 = g_l = 0.1$.

A useful trick is to define components of the state vector as the following six (t, g) pairs:

$$(0, g_l), (1, g_l), (2, g_l), (3, g_l), (3, g_h), (t \geq 4, g_l)$$

We think of these 6 states as corresponding to $s = 1, 2, 3, 4, 5, 6$.

The transition matrix is

$$P = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.5 & 0.5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

The government expenditure at each state is

$$g = \begin{pmatrix} 0.1 \\ 0.1 \\ 0.1 \\ 0.1 \\ 0.2 \\ 0.1 \end{pmatrix}$$

We assume the same utility parameters as in the *Lucas-Stokey economy*.

This utility function is implemented in the following class.

```
crra_util_data = [
    ('β', float64),
    ('σ', float64),
    ('γ', float64)
]

@jitclass(crra_util_data)
class CRRAutility:

    def __init__(self,
                 β=0.9,
                 σ=2,
                 γ=2):

        self.β, self.σ, self.γ = β, σ, γ

    # Utility function
    def U(self, c, l):
        # Note: `l` should not be interpreted as labor, it is an auxiliary
        # variable used to conveniently match the code and the equations
        # in the lecture
        σ = self.σ
        if σ == 1.:
            U = np.log(c)
        else:
            U = (c**(1 - σ) - 1) / (1 - σ)
        return U - (1-l) ** (1 + self.γ) / (1 + self.γ)

    # Derivatives of utility function
    def Uc(self, c, l):
        return c ** (-self.σ)

    def Ucc(self, c, l):
        return -self.σ * c ** (-self.σ - 1)

    def Ul(self, c, l):
        return (1-l) ** self.γ

    def Ull(self, c, l):
        return -self.γ * (1-l) ** (self.γ - 1)

    def Ucl(self, c, l):
        return 0

    def Ulc(self, c, l):
        return 0
```

The following figure plots Ramsey plans under complete and incomplete markets for both possible realizations of the state at time $t = 3$.

Ramsey outcomes and policies when the government has access to state-contingent debt are represented by black lines and by red lines when there is only a risk-free bond.

Paths with circles are histories in which there is peace, while those with triangle denote war.

```
# WARNING: DO NOT EXPECT THE CODE TO WORK IF YOU CHANGE PARAMETERS
σ = 2
γ = 2
β = 0.9
Π = np.array([[0, 1, 0,   0,   0,  0],
              [0, 0, 1,   0,   0,  0],
              [0, 0, 0, 0.5, 0.5,  0],
              [0, 0, 0,   0,   0,  1],
              [0, 0, 0,   0,   0,  1],
              [0, 0, 0,   0,   0,  1]])
g = np.array([0.1, 0.1, 0.1, 0.2, 0.1, 0.1])

x_min = -1.5555
x_max = 17.339
x_num = 300

x_grid = UCGrid((x_min, x_max, x_num))

crra_pref = CRRAutility(β=β, σ=σ, γ=γ)

S = len(Π)
bounds_v = np.vstack([np.hstack([np.full(S, -10.), np.zeros(S)]),
                      np.hstack([np.ones(S) - g, np.full(S, 10.)])]).T

amss_model = AMSS(crra_pref, β, Π, g, x_grid, bounds_v)
```

```
# WARNING: DO NOT EXPECT THE CODE TO WORK IF YOU CHANGE PARAMETERS
V = np.zeros((len(Π), x_num))
V[:] = -nodes(x_grid).T ** 2

σ_v_star = np.ones((S, x_num, S * 2))
σ_v_star[:, :, :S] = 0.0

W = np.empty(len(Π))
b_0 = 1.0
σ_w_star = np.ones((S, 2))
σ_w_star[:, 0] = -0.05
```

```
%%time

amss_model.solve(V, σ_v_star, b_0, W, σ_w_star)
```

```
===============
Solve time 1 problem
===============
```

```
Error at iteration 10 :  1.110064840137854
```

```
Error at iteration 20 :  0.30784885876438395
```

```
Error at iteration 30 :  0.03221851531398379
```

```
Error at iteration 40 :  0.014347598008733087
```

```
Error at iteration 50 :  0.0031219444631354065
```

```
Error at iteration 60 :  0.0010783647355108172
```

```
Error at iteration 70 :  0.0003761255356202753
```

```
Error at iteration 80 :  0.0001318127597098595
```

```
Error at iteration 90 :  4.650031579878089e-05
```

```
Error at iteration 100 :  1.801377708510188e-05
```

```
Error at iteration 110 :  6.175872600877597e-06
```

```
Error at iteration 120 :  2.4450291853383987e-06
```

```
Error at iteration 130 :  1.0836745989450947e-06
```

```
Error at iteration 140 :  5.682877084467464e-07
```

```
Error at iteration 150 :  3.567560966644123e-07
```

```
Error at iteration 160 :  2.5837734796141376e-07
```

```
Error at iteration 170 :  2.047536575844333e-07
```

```
Error at iteration 180 :  1.7066849622437985e-07
```

```
Error at iteration 190 :  1.4622035848788073e-07
```

```
Error at iteration 200 :  1.27387780324284e-07
```

```
Error at iteration 210 :  1.1226231499961159e-07
```

```
Successfully completed VFI after 220 iterations
===============
Solve time 0 problem
===============
```

```
Succesfully solved the time 0 problem.
CPU times: user 2min 10s, sys: 1.54 s, total: 2min 12s
Wall time: 1min 29s
```

```
# Solve the LS model
ls_model = SequentialLS(crra_pref, g=g, π=Π)
```

```
# WARNING: DO NOT EXPECT THE CODE TO WORK IF YOU CHANGE PARAMETERS
s_hist_h = np.array([0, 1, 2, 3, 5, 5, 5])
s_hist_l = np.array([0, 1, 2, 4, 5, 5, 5])

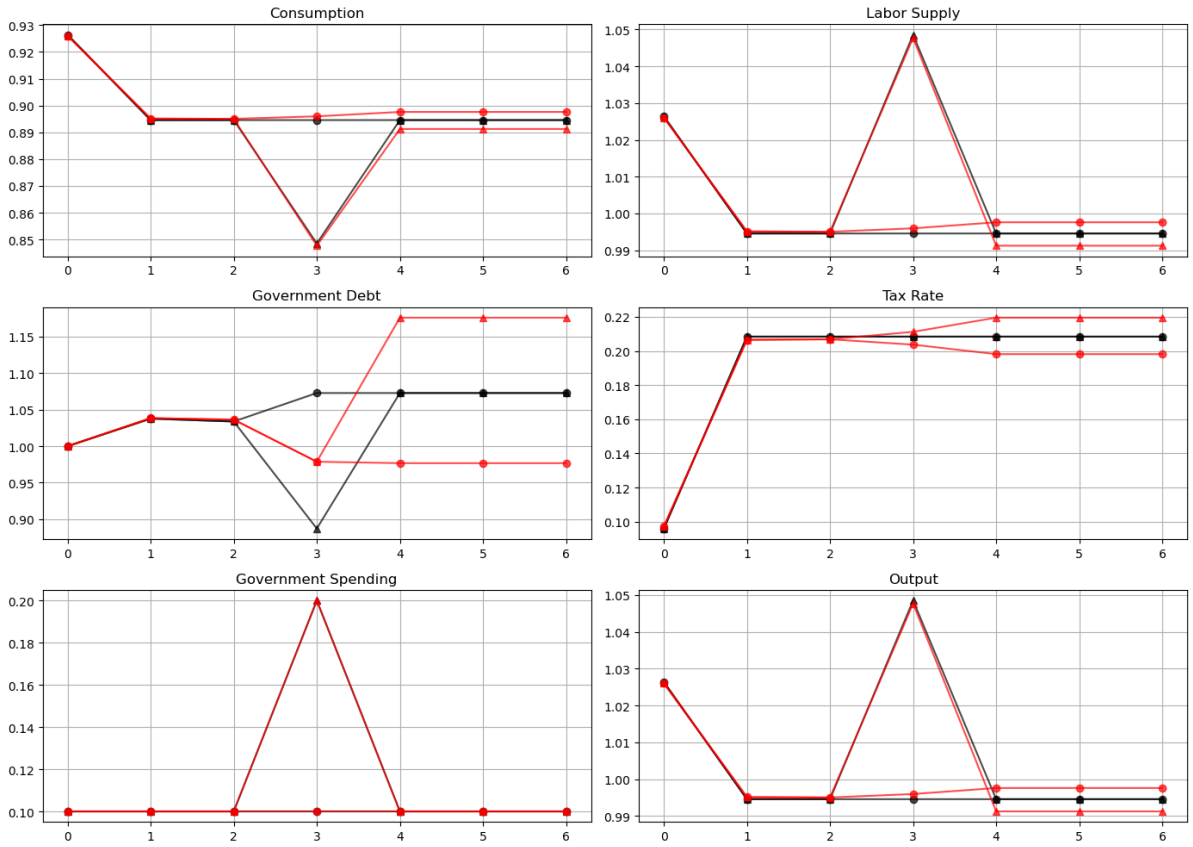
sim_h_amss = amss_model.simulate(s_hist_h, b_0)
sim_l_amss = amss_model.simulate(s_hist_l, b_0)

sim_h_ls = ls_model.simulate(b_0, 0, 7, s_hist_h)
sim_l_ls = ls_model.simulate(b_0, 0, 7, s_hist_l)

fig, axes = plt.subplots(3, 2, figsize=(14, 10))
titles = ['Consumption', 'Labor Supply', 'Government Debt',
          'Tax Rate', 'Government Spending', 'Output']

for ax, title, ls_l, ls_h, amss_l, amss_h in zip(axes.flatten(), titles,
                                                 sim_l_ls, sim_h_ls,
                                                 sim_l_amss, sim_h_amss):
    ax.plot(ls_l, '-ok', ls_h, '-^k', amss_l, '-or', amss_h, '-^r',
            alpha=0.7)
    ax.set(title=title)
    ax.grid()

plt.tight_layout()
plt.show()
```



How a Ramsey planner responds to war depends on the structure of the asset market.

If it is able to trade state-contingent debt, then at time $t = 2$

- the government **purchases** an Arrow security that pays off when $g_3 = g_h$
- the government **sells** an Arrow security that pays off when $g_3 = g_l$
- the Ramsey planner designs these purchases and sales designed so that, regardless of whether or not there is a war at $t = 3$, the government begins period $t = 4$ with the *same* government debt

This pattern facilities smoothing tax rates across states.

The government without state-contingent debt cannot do this.

Instead, it must enter time $t = 3$ with the same level of debt falling due whether there is peace or war at $t = 3$.

The risk-free rate between time 2 and time 3 is unusually **low** because at time 2 consumption at time 3 is expected to be unusually **low**.

A **low** risk-free rate of return on government debt between time 2 and time 3 allows the government to enter period 3 with **lower** government debt than it entered period 2.

To finance a war at time 3 it raises taxes and issues more debt to carry into perpetual peace that begins in period 4.

To service the additional debt burden, it raises taxes in all future periods.

The absence of state-contingent debt leads to an important difference in the optimal tax policy.

When the Ramsey planner has access to state-contingent debt, the optimal tax policy is history independent

- the tax rate is a function of the current level of government spending only, given the Lagrange multiplier on the implementability constraint

Without state-contingent debt, the optimal tax rate is history dependent.

- A war at time $t = 3$ causes a permanent **increase** in the tax rate.
- Peace at time $t = 3$ causes a permanent **reduction** in the tax rate.

Perpetual War Alert

History dependence occurs more dramatically in a case in which the government perpetually faces the prospect of war.

This case was studied in the final example of the lecture on *optimal taxation with state-contingent debt*.

There, each period the government faces a constant probability, 0.5, of war.

In addition, this example features the following preferences

$$u(c, n) = \log(c) + 0.69 \log(1 - n)$$

In accordance, we will re-define our utility function.

```
log_util_data = [
    ('β', float64),
    ('ψ', float64)
]

@jitclass(log_util_data)
class LogUtility:

    def __init__(self,
                 β=0.9,
                 ψ=0.69):

        self.β, self.ψ = β, ψ

    # Utility function
    def U(self, c, l):
        return np.log(c) + self.ψ * np.log(l)

    # Derivatives of utility function
    def Uc(self, c, l):
        return 1 / c

    def Ucc(self, c, l):
        return -c**(-2)

    def Ul(self, c, l):
        return self.ψ / l

    def Ull(self, c, l):
        return -self.ψ / l**2

    def Ucl(self, c, l):
        return 0

    def Ulc(self, c, l):
        return 0
```

With these preferences, Ramsey tax rates will vary even in the Lucas-Stokey model with state-contingent debt.

The figure below plots optimal tax policies for both the economy with state-contingent debt (circles) and the economy with only a risk-free bond (triangles).

```
# WARNING: DO NOT EXPECT THE CODE TO WORK IF YOU CHANGE PARAMETERS
ψ = 0.69
Π = np.full((2, 2), 0.5)
β = 0.9
g = np.array([0.1, 0.2])

x_min = -3.4107
x_max = 3.709
x_num = 300

x_grid = UCGrid((x_min, x_max, x_num))
log_pref = LogUtility(β=β, ψ=ψ)

S = len(Π)
bounds_v = np.vstack([np.zeros(2 * S), np.hstack([1 - g, np.ones(S)]) ]).T

V = np.zeros((len(Π), x_num))
V[:] = -(nodes(x_grid).T + x_max) ** 2 / 14

σ_v_star = 1 - np.full((S, x_num, S * 2), 0.55)

W = np.empty(len(Π))
b_0 = 0.5
σ_w_star = 1 - np.full((S, 2), 0.55)

amss_model = AMSS(log_pref, β, Π, g, x_grid, bounds_v)
```

```
%%time

amss_model.solve(V, σ_v_star, b_0, W, σ_w_star, tol_vfi=3e-5, maxitr=3000,
                 print_itr=100)
```

```
===============
Solve time 1 problem
===============
```

```
Error at iteration 100 :  0.0011569123052908026
```

```
Error at iteration 200 :  0.0005024948171925558
```

```
Error at iteration 300 :  0.0002995649778405607
```

```
Error at iteration 400 :  0.00020753209923363158
```

```
Error at iteration 500 :  0.00015556566848218267
```

```
Error at iteration 600 :  0.0001228034492957164
```

```
Error at iteration 700 :  0.00010068689697462219
```

```
Error at iteration 800 :  8.474340939912395e-05
```

```
Error at iteration 900 :  7.290920770763876e-05
```

```
Error at iteration 1000 :  6.375694017535238e-05
```

```
Error at iteration 1100 :  5.642689428775327e-05
```

```
Error at iteration 1200 :  5.045426282634935e-05
```

```
Error at iteration 1300 :  4.561168914030134e-05
```

```
Error at iteration 1400 :  4.150059282892471e-05
```

```
Error at iteration 1500 :  3.799110186264443e-05
```

```
Error at iteration 1600 :  3.5163266918658564e-05
```

```
Error at iteration 1700 :  3.263979350620616e-05
```

```
Error at iteration 1800 :  3.0359381506528393e-05
```

```
Successfully completed VFI after 1818 iterations
===============
Solve time 0 problem
===============
```

```
Succesfully solved the time 0 problem.
CPU times: user 2min, sys: 1.29 s, total: 2min 1s
Wall time: 1min 30s
```

```
ls_model = SequentialLS(log_pref, g=g, π=Π)  # Solve sequential problem
```

```
# WARNING: DO NOT EXPECT THE CODE TO WORK IF YOU CHANGE PARAMETERS
s_hist = np.array([0, 0, 0, 0, 0, 0, 0, 0, 1, 1,
                   0, 0, 0, 1, 1, 1, 1, 1, 1, 0])

T = len(s_hist)

sim_amss = amss_model.simulate(s_hist, b_0)
sim_ls = ls_model.simulate(0.5, 0, T, s_hist)

titles = ['Consumption', 'Labor Supply', 'Government Debt',
```

(continues on next page)

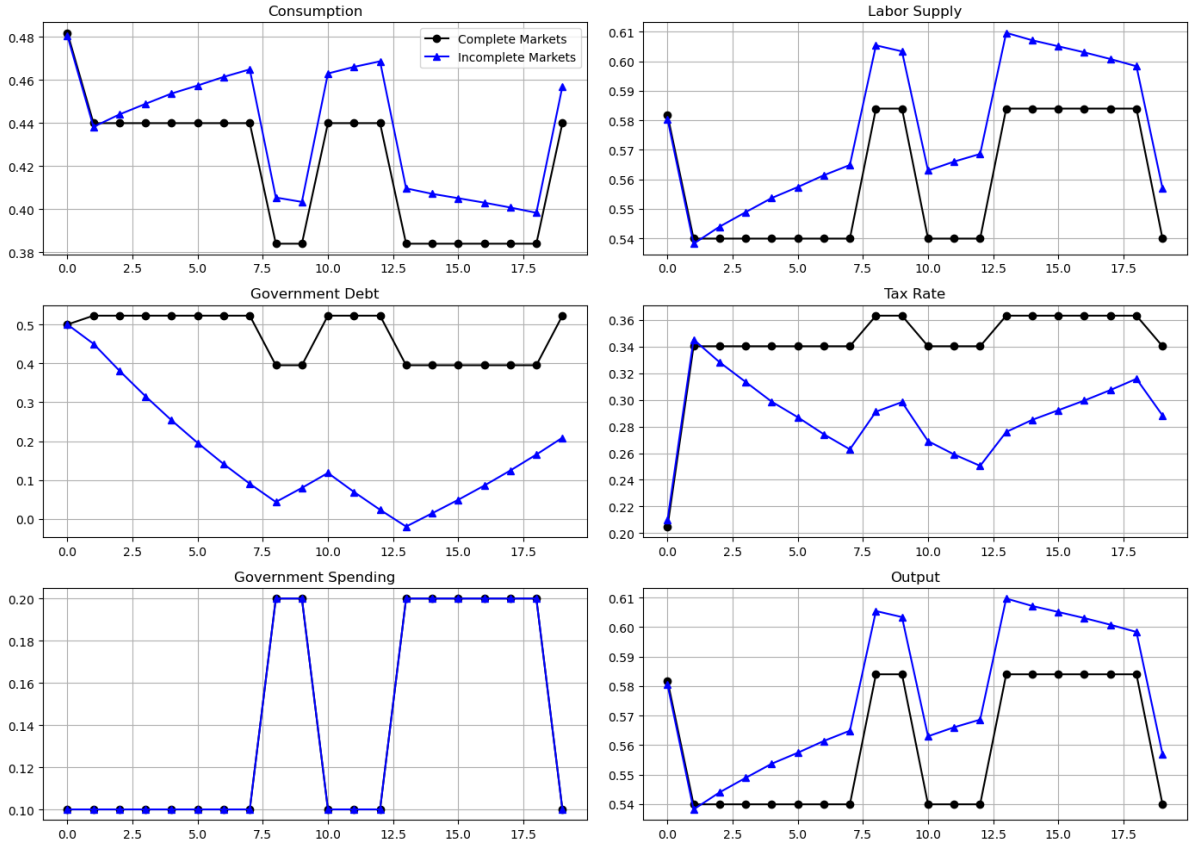
(continued from previous page)

```
          'Tax Rate', 'Government Spending', 'Output']

fig, axes = plt.subplots(3, 2, figsize=(14, 10))

for ax, title, ls, amss in zip(axes.flatten(), titles, sim_ls, sim_amss):
    ax.plot(ls, '-ok', amss, '-^b')
    ax.set(title=title)
    ax.grid()

axes[0, 0].legend(('Complete Markets', 'Incomplete Markets'))
plt.tight_layout()
plt.show()
```



When the government experiences a prolonged period of peace, it is able to reduce government debt and set persistently lower tax rates.

However, the government finances a long war by borrowing and raising taxes.

This results in a drift away from policies with state-contingent debt that depends on the history of shocks.

This is even more evident in the following figure that plots the evolution of the two policies over 200 periods.

This outcome reflects the presence of a force for **precautionary saving** that the incomplete markets structure imparts to the Ramsey plan.

In *this subsequent lecture* and *this subsequent lecture*, some ultimate consequences of that force are explored.

```
T = 200
s_0 = 0
mc = MarkovChain(Π)

s_hist_long = mc.simulate(T, init=s_0, random_state=5)
```

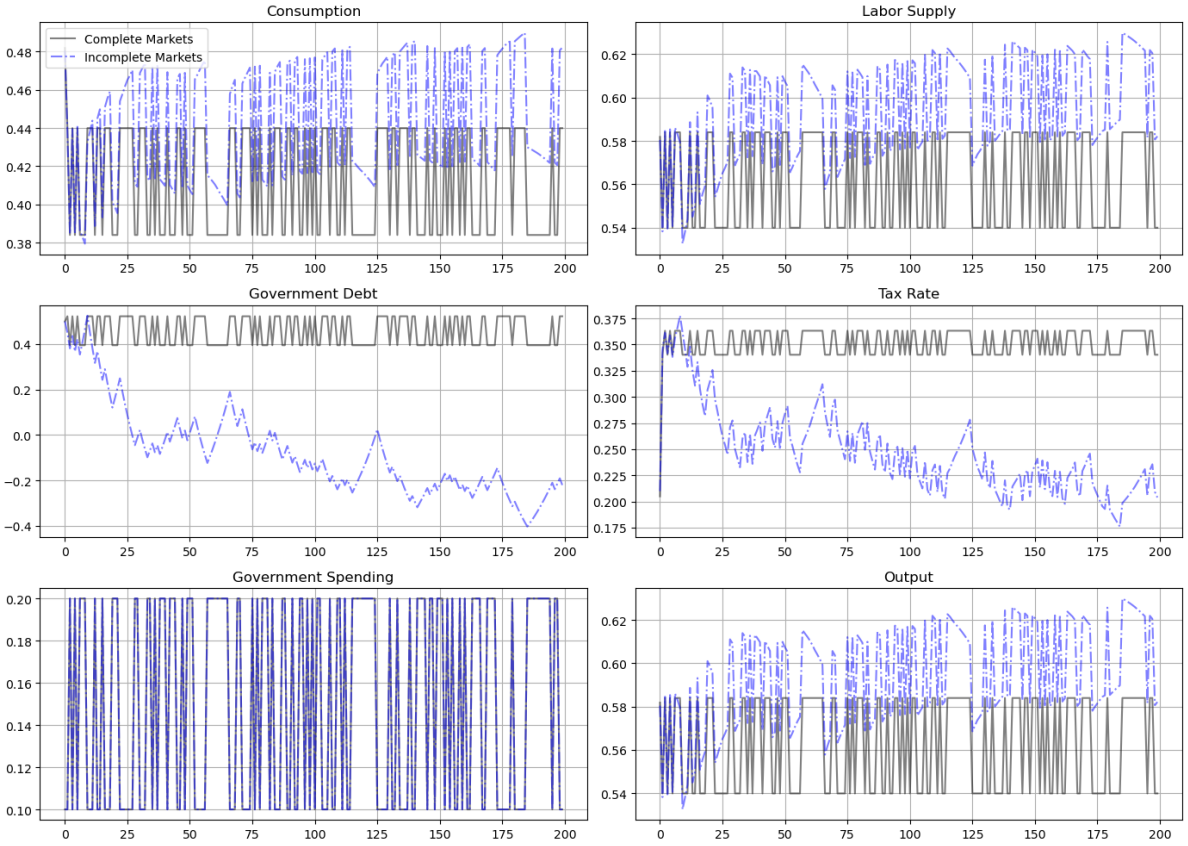
```
sim_amss = amss_model.simulate(s_hist_long, b_0)
sim_ls = ls_model.simulate(0.5, 0, T, s_hist_long)

titles = ['Consumption', 'Labor Supply', 'Government Debt',
          'Tax Rate', 'Government Spending', 'Output']


fig, axes = plt.subplots(3, 2, figsize=(14, 10))

for ax, title, ls, amss in zip(axes.flatten(), titles, sim_ls, \
        sim_amss):
    ax.plot(ls, '-k', amss, '-.b', alpha=0.5)
    ax.set(title=title)
    ax.grid()

axes[0, 0].legend(('Complete Markets','Incomplete Markets'))
plt.tight_layout()
plt.show()
```



CHAPTER

FORTYSEVEN

FLUCTUATING INTEREST RATES DELIVER FISCAL INSURANCE

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install --upgrade quantecon
```

47.1 Overview

This lecture extends our investigations of how optimal policies for levying a flat-rate tax on labor income and issuing government debt depend on whether there are complete markets for debt.

A Ramsey allocation and Ramsey policy in the AMSS [Aiyagari *et al.*, 2002] model described in *optimal taxation without state-contingent debt* generally differs from a Ramsey allocation and Ramsey policy in the Lucas-Stokey [Lucas and Stokey, 1983] model described in *optimal taxation with state-contingent debt*.

This is because the implementability restriction that a competitive equilibrium with a distorting tax imposes on allocations in the Lucas-Stokey model is just one among a set of implementability conditions imposed in the AMSS model.

These additional constraints require that time t components of a Ramsey allocation for the AMSS model be **measurable** with respect to time $t-1$ information.

The measurability constraints imposed by the AMSS model are inherited from the restriction that only one-period risk-free bonds can be traded.

Differences between the Ramsey allocations in the two models indicate that at least some of the **implementability constraints** of the AMSS model of *optimal taxation without state-contingent debt* are violated at the Ramsey allocation of a corresponding [Lucas and Stokey, 1983] model with state-contingent debt.

Another way to say this is that differences between the Ramsey allocations of the two models indicate that some of the **measurability constraints** imposed by the AMSS model are violated at the Ramsey allocation of the Lucas-Stokey model.

Nonzero Lagrange multipliers on those constraints make the Ramsey allocation for the AMSS model differ from the Ramsey allocation for the Lucas-Stokey model.

This lecture studies a special AMSS model in which

- The exogenous state variable s_t is governed by a finite-state Markov chain.
- With an arbitrary budget-feasible initial level of government debt, the measurability constraints
  - bind for many periods, but ....
  - eventually, they stop binding evermore, so that ...
  - in the tail of the Ramsey plan, the Lagrange multipliers $\gamma_t(s^t)$ on the AMSS implementability constraints (46.8) are zero.

- After the implementability constraints (46.8) no longer bind in the tail of the AMSS Ramsey plan
  - history dependence of the AMSS state variable x_t vanishes and x_t becomes a time-invariant function of the Markov state s_t.
  - the par value of government debt becomes **constant over time** so that $b_{t+1}(s^t) = \bar{b}$ for $t \geq T$ for a sufficiently large T.
  - $\bar{b} < 0$, so that the tail of the Ramsey plan instructs the government always to make a constant par value of risk-free one-period loans **to** the private sector.
  - the one-period gross interest rate $R_t(s^t)$ on risk-free debt converges to a time-invariant function of the Markov state s_t.
- For a **particular** $b_0 < 0$ (i.e., a positive level of initial government **loans** to the private sector), the measurability constraints **never** bind.
- In this special case
  - the **par value** $b_{t+1}(s_t) = \bar{b}$ of government debt at time t and Markov state s_t is constant across time and states, but ....
  - the **market value** $\frac{\bar{b}}{R_t(s_t)}$ of government debt at time t varies as a time-invariant function of the Markov state s_t.
  - fluctuations in the interest rate make gross earnings on government debt $\frac{\bar{b}}{R_t(s_t)}$ fully insure the gross-of-gross-interest-payments government budget against fluctuations in government expenditures.
  - the state variable x in a recursive representation of a Ramsey plan is a time-invariant function of the Markov state for $t \geq 0$.
- In this special case, the Ramsey allocation in the AMSS model agrees with that in a Lucas-Stokey [Lucas and Stokey, 1983] complete markets model in which the same amount of state-contingent debt falls due in all states tomorrow
  - it is a situation in which the Ramsey planner loses nothing from not being able to trade state-contingent debt and being restricted to exchange only risk-free debt debt.
- This outcome emerges only when we initialize government debt at a particular $b_0 < 0$.

In a nutshell, the reason for this striking outcome is that at a particular level of risk-free government **assets**, fluctuations in the one-period risk-free interest rate provide the government with complete insurance against stochastically varying government expenditures.

Let's start with some imports:

```
import matplotlib.pyplot as plt
from scipy.optimize import fsolve, fmin
```

47.2 Forces at Work

The forces driving asymptotic outcomes here are examples of dynamics present in a more general class of incomplete markets models analyzed in [Bhandari *et al.*, 2017] (BEGS).

BEGS provide conditions under which government debt under a Ramsey plan converges to an invariant distribution.

BEGS construct approximations to that asymptotically invariant distribution of government debt under a Ramsey plan.

BEGS also compute an approximation to a Ramsey plan's rate of convergence to that limiting invariant distribution.

We shall use the BEGS approximating limiting distribution and their approximating rate of convergence to help interpret outcomes here.

For a long time, the Ramsey plan puts a nontrivial martingale-like component into the par value of government debt as part of the way that the Ramsey plan imperfectly smooths distortions from the labor tax rate across time and Markov states.

But BEGS show that binding implementability constraints slowly push government debt in a direction designed to let the government use fluctuations in equilibrium interest rates rather than fluctuations in par values of debt to insure against shocks to government expenditures.

- This is a **weak** (but unrelenting) force that, starting from a positive initial debt level, for a long time is dominated by the stochastic martingale-like component of debt dynamics that the Ramsey planner uses to facilitate imperfect tax-smoothing across time and states.
- This weak force slowly drives the par value of government **assets** to a **constant** level at which the government can completely insure against government expenditure shocks while shutting down the stochastic component of debt dynamics.
- At that point, the tail of the par value of government debt becomes a trivial martingale: it is constant over time.

47.3 Logical Flow of Lecture

We present ideas in the following order

- We describe a two-state AMSS economy and generate a long simulation starting from a positive initial government debt.
- We observe that in a long simulation starting from positive government debt, the par value of government debt eventually converges to a constant $\bar{b}$.
- In fact, the par value of government debt converges to the same constant level $\bar{b}$ for alternative realizations of the Markov government expenditure process and for alternative settings of initial government debt b_0.
- We reverse engineer a particular value of initial government debt b_0 (it turns out to be negative) for which the continuation debt moves to $\bar{b}$ immediately.
- We note that for this particular initial debt b_0, the Ramsey allocations for the AMSS economy and the Lucas-Stokey model are identical
  - we verify that the LS Ramsey planner chooses to purchase **identical** claims to time $t+1$ consumption for all Markov states tomorrow for each Markov state today.
- We compute the BEGS approximations to check how accurately they describe the dynamics of the long-simulation.

47.3.1 Equations from Lucas-Stokey (1983) Model

Although we are studying an AMSS [Aiyagari *et al.*, 2002] economy, a Lucas-Stokey [Lucas and Stokey, 1983] economy plays an important role in the reverse-engineering calculation to be described below.

For that reason, it is helpful to have key equations underlying a Ramsey plan for the Lucas-Stokey economy readily available.

Recall first-order conditions for a Ramsey allocation for the Lucas-Stokey economy.

For $t \geq 1$, these take the form

$$\begin{aligned}(1+\Phi)u_c(c,1-c-g)+\Phi\left[cu_{cc}(c,1-c-g)-(c+g)u_{\ell c}(c,1-c-g)\right]\\ =(1+\Phi)u_\ell(c,1-c-g)+\Phi\left[cu_{c\ell}(c,1-c-g)-(c+g)u_{\ell\ell}(c,1-c-g)\right]\end{aligned} \tag{47.1}$$

There is one such equation for each value of the Markov state s_t.

Given an initial Markov state, the time $t = 0$ quantities c_0 and b_0 satisfy

$$\begin{aligned}(1+\Phi)u_c(c,1-c-g)+\Phi\left[cu_{cc}(c,1-c-g)-(c+g)u_{\ell c}(c,1-c-g)\right]\\ =(1+\Phi)u_\ell(c,1-c-g)+\Phi\left[cu_{c\ell}(c,1-c-g)-(c+g)u_{\ell\ell}(c,1-c-g)\right]+\Phi(u_{cc}-u_{c,\ell})b_0\end{aligned} \tag{47.2}$$

In addition, the time $t = 0$ budget constraint is satisfied at c_0 and initial government debt b_0

$$b_0 + g_0 = \tau_0(c_0 + g_0) + \frac{\bar{b}}{R_0} \tag{47.3}$$

where R_0 is the gross interest rate for the Markov state s_0 that is assumed to prevail at time $t = 0$ and τ_0 is the time $t = 0$ tax rate.

In equation (47.3), it is understood that

$$\tau_0 = 1 - \frac{u_{l,0}}{u_{c,0}}$$

$$R_0^{-1} = \beta \sum_{s=1}^{S} \Pi(s|s_0)\frac{u_c(s)}{u_{c,0}}$$

It is useful to transform some of the above equations to forms that are more natural for analyzing the case of a CRRA utility specification that we shall use in our example economies.

47.3.2 Specification with CRRA Utility

As in lectures *optimal taxation without state-contingent debt* and *optimal taxation with state-contingent debt*, we assume that the representative agent has utility function

$$u(c,n) = \frac{c^{1-\sigma}}{1-\sigma} - \frac{n^{1+\gamma}}{1+\gamma}$$

and set $\sigma = 2$, $\gamma = 2$, and the discount factor $\beta = 0.9$.

We eliminate leisure from the model and continue to assume that

$$c_t + g_t = n_t$$

The analysis of Lucas and Stokey prevails once we make the following replacements

$$\begin{aligned}u_\ell(c,\ell) &\sim -u_n(c,n)\\ u_c(c,\ell) &\sim u_c(c,n)\\ u_{\ell,\ell}(c,\ell) &\sim u_{nn}(c,n)\\ u_{c,c}(c,\ell) &\sim u_{c,c}(c,n)\\ u_{c,\ell}(c,\ell) &\sim 0\end{aligned}$$

With these understandings, equations (47.1) and (47.2) simplify in the case of the CRRA utility function.

They become

$$(1+\Phi)[u_c(c) + u_n(c+g)] + \Phi[cu_{cc}(c) + (c+g)u_{nn}(c+g)] = 0 \tag{47.4}$$

and

$$(1+\Phi)[u_c(c_0) + u_n(c_0+g_0)] + \Phi[c_0u_{cc}(c_0) + (c_0+g_0)u_{nn}(c_0+g_0)] - \Phi u_{cc}(c_0)b_0 = 0 \tag{47.5}$$

In equation (47.4), it is understood that c and g are each functions of the Markov state s.

The CRRA utility function is represented in the following class.

```
import numpy as np


class CRRAutility:

    def __init__(self,
                 β=0.9,
                 σ=2,
                 γ=2,
                 π=np.full((2, 2), 0.5),
                 G=np.array([0.1, 0.2]),
                 Θ=np.ones(2),
                 transfers=False):

        self.β, self.σ, self.γ = β, σ, γ
        self.π, self.G, self.Θ, self.transfers = π, G, Θ, transfers

    # Utility function
    def U(self, c, n):
        σ = self.σ
        if σ == 1.:
            U = np.log(c)
        else:
            U = (c**(1 - σ) - 1) / (1 - σ)
        return U - n**(1 + self.γ) / (1 + self.γ)

    # Derivatives of utility function
    def Uc(self, c, n):
        return c**(-self.σ)

    def Ucc(self, c, n):
        return -self.σ * c**(-self.σ - 1)

    def Un(self, c, n):
        return -n**self.γ

    def Unn(self, c, n):
        return -self.γ * n**(self.γ - 1)
```

47.4 Example Economy

We set the following parameter values.

The Markov state s_t takes two values, namely, $0, 1$.

The initial Markov state is 0.

The Markov transition matrix is $.5I$ where I is a 2×2 identity matrix, so the s_t process is IID.

Government expenditures $g(s)$ equal $.1$ in Markov state 0 and $.2$ in Markov state 1.

We set preference parameters as follows:

$$\beta = .9$$
$$\sigma = 2$$
$$\gamma = 2$$

Here are several classes that do most of the work for us.

The code is mostly taken or adapted from the earlier lectures *optimal taxation without state-contingent debt* and *optimal taxation with state-contingent debt*.

```
import numpy as np
from scipy.optimize import root
from quantecon import MarkovChain


class SequentialAllocation:

    '''
    Class that takes CESutility or BGPutility object as input returns
    planner's allocation as a function of the multiplier on the
    implementability constraint μ.
    '''

    def __init__(self, model):

        # Initialize from model object attributes
        self.β, self.π, self.G = model.β, model.π, model.G
        self.mc, self.Θ = MarkovChain(self.π), model.Θ
        self.S = len(model.π)  # Number of states
        self.model = model

        # Find the first best allocation
        self.find_first_best()

    def find_first_best(self):
        '''
        Find the first best allocation
        '''
        model = self.model
        S, Θ, G = self.S, self.Θ, self.G
        Uc, Un = model.Uc, model.Un

        def res(z):
            c = z[:S]
            n = z[S:]
            return np.hstack([Θ * Uc(c, n) + Un(c, n), Θ * n - c - G])

        res = root(res, np.full(2 * S, 0.5))

        if not res.success:
            raise Exception('Could not find first best')

        self.cFB = res.x[:S]
        self.nFB = res.x[S:]

        # Multiplier on the resource constraint
        self.ΞFB = Uc(self.cFB, self.nFB)
        self.zFB = np.hstack([self.cFB, self.nFB, self.ΞFB])

    def time1_allocation(self, μ):
        '''
        Computes optimal allocation for time t >= 1 for a given μ
        '''
```

(continues on next page)

(continued from previous page)

```
        model = self.model
        S, Θ, G = self.S, self.Θ, self.G
        Uc, Ucc, Un, Unn = model.Uc, model.Ucc, model.Un, model.Unn

        def FOC(z):
            c = z[:S]
            n = z[S:2 * S]
            Ξ = z[2 * S:]
            # FOC of c
            return np.hstack([Uc(c, n) - μ * (Ucc(c, n) * c + Uc(c, n)) - Ξ,
                              Un(c, n) - μ * (Unn(c, n) * n + Un(c, n)) \
                              + Θ * Ξ,  # FOC of n
                              Θ * n - c - G])

        # Find the root of the first-order condition
        res = root(FOC, self.zFB)
        if not res.success:
            raise Exception('Could not find LS allocation.')
        z = res.x
        c, n, Ξ = z[:S], z[S:2 * S], z[2 * S:]

        # Compute x
        I = Uc(c, n) * c + Un(c, n) * n
        x = np.linalg.solve(np.eye(S) - self.β * self.π, I)

        return c, n, x, Ξ

    def time0_allocation(self, B_, s_0):
        '''
        Finds the optimal allocation given initial government debt B_ and
        state s_0
        '''
        model, π, Θ, G, β = self.model, self.π, self.Θ, self.G, self.β
        Uc, Ucc, Un, Unn = model.Uc, model.Ucc, model.Un, model.Unn

        # First order conditions of planner's problem
        def FOC(z):
            μ, c, n, Ξ = z
            xprime = self.time1_allocation(μ)[2]
            return np.hstack([Uc(c, n) * (c - B_) + Un(c, n) * n + β * π[s_0]
                                        @ xprime,
                              Uc(c, n) - μ * (Ucc(c, n)
                                        * (c - B_) + Uc(c, n)) - Ξ,
                              Un(c, n) - μ * (Unn(c, n) * n
                                        + Un(c, n)) + Θ[s_0] * Ξ,
                              (Θ * n - c - G)[s_0]])

        # Find root
        res = root(FOC, np.array(
            [0, self.cFB[s_0], self.nFB[s_0], self.ΞFB[s_0]]))
        if not res.success:
            raise Exception('Could not find time 0 LS allocation.')

        return res.x

    def time1_value(self, μ):
```

(continues on next page)

(continued from previous page)

```
        '''
        Find the value associated with multiplier μ
        '''
        c, n, x, Ξ = self.time1_allocation(μ)
        U = self.model.U(c, n)
        V = np.linalg.solve(np.eye(self.S) - self.β * self.π, U)
        return c, n, x, V

    def T(self, c, n):
        '''
        Computes T given c, n
        '''
        model = self.model
        Uc, Un = model.Uc(c, n), model.Un(c,  n)

        return 1 + Un / (self.Θ * Uc)

    def simulate(self, B_, s_0, T, sHist=None):
        '''
        Simulates planners policies for T periods
        '''
        model, π, β = self.model, self.π, self.β
        Uc = model.Uc

        if sHist is None:
            sHist = self.mc.simulate(T, s_0)

        cHist, nHist, Bhist, THist, μHist = np.zeros((5, T))
        RHist = np.zeros(T - 1)

        # Time 0
        μ, cHist[0], nHist[0], _ = self.time0_allocation(B_, s_0)
        THist[0] = self.T(cHist[0], nHist[0])[s_0]
        Bhist[0] = B_
        μHist[0] = μ

        # Time 1 onward
        for t in range(1, T):
            c, n, x, Ξ = self.time1_allocation(μ)
            T = self.T(c, n)
            u_c = Uc(c, n)
            s = sHist[t]
            Eu_c = π[sHist[t - 1]] @ u_c
            cHist[t], nHist[t], Bhist[t], THist[t] = c[s], n[s], x[s] / u_c[s], \
                                                     T[s]
            RHist[t - 1] = Uc(cHist[t - 1], nHist[t - 1]) / (β * Eu_c)
            μHist[t] = μ

        return [cHist, nHist, Bhist, THist, sHist, μHist, RHist]
```

```
import numpy as np
from scipy.optimize import fmin_slsqp
from scipy.optimize import root
from quantecon import MarkovChain
```

(continues on next page)

(continued from previous page)

```
class RecursiveAllocationAMSS:

    def __init__(self, model, μgrid, tol_diff=1e-7, tol=1e-7):

        self.β, self.π, self.G = model.β, model.π, model.G
        self.mc, self.S = MarkovChain(self.π), len(model.π)  # Number of states
        self.Θ, self.model, self.μgrid = model.Θ, model, μgrid
        self.tol_diff, self.tol = tol_diff, tol

        # Find the first best allocation
        self.solve_time1_bellman()
        self.T.time_0 = True  # Bellman equation now solves time 0 problem

    def solve_time1_bellman(self):
        '''
        Solve the time  1 Bellman equation for calibration model and
        initial grid μgrid0
        '''
        model, μgrid0 = self.model, self.μgrid
        π = model.π
        S = len(model.π)

        # First get initial fit from Lucas Stokey solution.
        # Need to change things to be ex ante
        pp = SequentialAllocation(model)
        interp = interpolator_factory(2, None)

        def incomplete_allocation(μ_, s_):
            c, n, x, V = pp.time1_value(μ_)
            return c, n, π[s_] @ x, π[s_] @ V
        cf, nf, xgrid, Vf, xprimef = [], [], [], [], []
        for s_ in range(S):
            c, n, x, V = zip(*map(lambda μ: incomplete_allocation(μ, s_), μgrid0))
            c, n = np.vstack(c).T, np.vstack(n).T
            x, V = np.hstack(x), np.hstack(V)
            xprimes = np.vstack([x] * S)
            cf.append(interp(x, c))
            nf.append(interp(x, n))
            Vf.append(interp(x, V))
            xgrid.append(x)
            xprimef.append(interp(x, xprimes))
        cf, nf, xprimef = fun_vstack(cf), fun_vstack(nf), fun_vstack(xprimef)
        Vf = fun_hstack(Vf)
        policies = [cf, nf, xprimef]

        # Create xgrid
        x = np.vstack(xgrid).T
        xbar = [x.min(0).max(), x.max(0).min()]
        xgrid = np.linspace(xbar[0], xbar[1], len(μgrid0))
        self.xgrid = xgrid

        # Now iterate on Bellman equation
        T = BellmanEquation(model, xgrid, policies, tol=self.tol)
        diff = 1
        while diff > self.tol_diff:
```

(continues on next page)

(continued from previous page)

```
            PF = T(Vf)

            Vfnew, policies = self.fit_policy_function(PF)
            diff = np.abs((Vf(xgrid) - Vfnew(xgrid)) / Vf(xgrid)).max()

            print(diff)
            Vf = Vfnew

        # Store value function policies and Bellman Equations
        self.Vf = Vf
        self.policies = policies
        self.T = T

    def fit_policy_function(self, PF):
        '''
        Fits the policy functions
        '''
        S, xgrid = len(self.π), self.xgrid
        interp = interpolator_factory(3, 0)
        cf, nf, xprimef, Tf, Vf = [], [], [], [], []
        for s_ in range(S):
            PFvec = np.vstack([PF(x, s_) for x in self.xgrid]).T
            Vf.append(interp(xgrid, PFvec[0, :]))
            cf.append(interp(xgrid, PFvec[1:1 + S]))
            nf.append(interp(xgrid, PFvec[1 + S:1 + 2 * S]))
            xprimef.append(interp(xgrid, PFvec[1 + 2 * S:1 + 3 * S]))
            Tf.append(interp(xgrid, PFvec[1 + 3 * S:]))
        policies = fun_vstack(cf), fun_vstack(
            nf), fun_vstack(xprimef), fun_vstack(Tf)
        Vf = fun_hstack(Vf)
        return Vf, policies

    def T(self, c, n):
        '''
        Computes T given c and n
        '''
        model = self.model
        Uc, Un = model.Uc(c, n), model.Un(c, n)

        return 1 + Un / (self.Θ * Uc)

    def time0_allocation(self, B_, s0):
        '''
        Finds the optimal allocation given initial government debt B_ and
        state s_0
        '''
        PF = self.T(self.Vf)
        z0 = PF(B_, s0)
        c0, n0, xprime0, T0 = z0[1:]
        return c0, n0, xprime0, T0

    def simulate(self, B_, s_0, T, sHist=None):
        '''
        Simulates planners policies for T periods
        '''
        model, π = self.model, self.π
```

(continues on next page)

(continued from previous page)

```
        Uc = model.Uc
        cf, nf, xprimef, Tf = self.policies

        if sHist is None:
            sHist = simulate_markov(π, s_0, T)

        cHist, nHist, Bhist, xHist, THist, THist, μHist = np.zeros((7, T))
        # Time 0
        cHist[0], nHist[0], xHist[0], THist[0] = self.time0_allocation(B_, s_0)
        THist[0] = self.T(cHist[0], nHist[0])[s_0]
        Bhist[0] = B_
        μHist[0] = self.Vf[s_0](xHist[0])

        # Time 1 onward
        for t in range(1, T):
            s_, x, s = sHist[t - 1], xHist[t - 1], sHist[t]
            c, n, xprime, T = cf[s_, :](x), nf[s_, :](
                x), xprimef[s_, :](x), Tf[s_, :](x)

            T = self.T(c, n)[s]
            u_c = Uc(c, n)
            Eu_c = π[s_, :] @ u_c

            μHist[t] = self.Vf[s](xprime[s])

            cHist[t], nHist[t], Bhist[t], THist[t] = c[s], n[s], x / Eu_c, T
            xHist[t], THist[t] = xprime[s], T[s]
        return [cHist, nHist, Bhist, THist, THist, μHist, sHist, xHist]


class BellmanEquation:
    '''
    Bellman equation for the continuation of the Lucas-Stokey Problem
    '''

    def __init__(self, model, xgrid, policies0, tol, maxiter=1000):

        self.β, self.π, self.G = model.β, model.π, model.G
        self.S = len(model.π)  # Number of states
        self.Θ, self.model, self.tol = model.Θ, model, tol
        self.maxiter = maxiter

        self.xbar = [min(xgrid), max(xgrid)]
        self.time_0 = False

        self.z0 = {}
        cf, nf, xprimef = policies0

        for s_ in range(self.S):
            for x in xgrid:
                self.z0[x, s_] = np.hstack([cf[s_, :](x),
                                            nf[s_, :](x),
                                            xprimef[s_, :](x),
                                            np.zeros(self.S)])

        self.find_first_best()
```

(continues on next page)

(continued from previous page)

```
    def find_first_best(self):
        '''
        Find the first best allocation
        '''
        model = self.model
        S, Θ, Uc, Un, G = self.S, self.Θ, model.Uc, model.Un, self.G

        def res(z):
            c = z[:S]
            n = z[S:]
            return np.hstack([Θ * Uc(c, n) + Un(c, n), Θ * n - c - G])

        res = root(res, np.full(2 * S, 0.5))
        if not res.success:
            raise Exception('Could not find first best')

        self.cFB = res.x[:S]
        self.nFB = res.x[S:]
        IFB = Uc(self.cFB, self.nFB) * self.cFB + \
            Un(self.cFB, self.nFB) * self.nFB

        self.xFB = np.linalg.solve(np.eye(S) - self.β * self.π, IFB)

        self.zFB = {}
        for s in range(S):
            self.zFB[s] = np.hstack(
                [self.cFB[s], self.nFB[s], self.π[s] @ self.xFB, 0.])

    def __call__(self, Vf):
        '''
        Given continuation value function next period return value function this
        period return T(V) and optimal policies
        '''
        if not self.time_0:
            def PF(x, s): return self.get_policies_time1(x, s, Vf)
        else:
            def PF(B_, s0): return self.get_policies_time0(B_, s0, Vf)
        return PF

    def get_policies_time1(self, x, s_, Vf):
        '''
        Finds the optimal policies
        '''
        model, β, Θ, G, S, π = self.model, self.β, self.Θ, self.G, self.S, self.π
        U, Uc, Un = model.U, model.Uc, model.Un

        def objf(z):
            c, n, xprime = z[:S], z[S:2 * S], z[2 * S:3 * S]

            Vprime = np.empty(S)
            for s in range(S):
                Vprime[s] = Vf[s](xprime[s])

            return -π[s_] @ (U(c, n) + β * Vprime)
```

(continues on next page)

(continued from previous page)

```
        def objf_prime(x):

            epsilon = 1e-7
            x0 = np.asfarray(x)
            f0 = np.atleast_1d(objf(x0))
            jac = np.zeros([len(x0), len(f0)])
            dx = np.zeros(len(x0))
            for i in range(len(x0)):
                dx[i] = epsilon
                jac[i] = (objf(x0+dx) - f0)/epsilon
                dx[i] = 0.0

            return jac.transpose()

        def cons(z):
            c, n, xprime, T = z[:S], z[S:2 * S], z[2 * S:3 * S], z[3 * S:]
            u_c = Uc(c, n)
            Eu_c = π[s_] @ u_c
            return np.hstack([
                x * u_c / Eu_c - u_c * (c - T) - Un(c, n) * n - β * xprime,
                Θ * n - c - G])

        if model.transfers:
            bounds = [(0., 100)] * S + [(0., 100)] * S + \
                [self.xbar] * S + [(0., 100.)] * S
        else:
            bounds = [(0., 100)] * S + [(0., 100)] * S + \
                [self.xbar] * S + [(0., 0.)] * S
        out, fx, _, imode, smode = fmin_slsqp(objf, self.z0[x, s_],
                                              f_eqcons=cons, bounds=bounds,
                                              fprime=objf_prime, full_output=True,
                                              iprint=0, acc=self.tol, iter=self.
↪maxiter)

        if imode > 0:
            raise Exception(smode)

        self.z0[x, s_] = out
        return np.hstack([-fx, out])

    def get_policies_time0(self, B_, s0, Vf):
        '''
        Finds the optimal policies
        '''
        model, β, Θ, G = self.model, self.β, self.Θ, self.G
        U, Uc, Un = model.U, model.Uc, model.Un

        def objf(z):
            c, n, xprime = z[:-1]

            return -(U(c, n) + β * Vf[s0](xprime))

        def cons(z):
            c, n, xprime, T = z
            return np.hstack([
                -Uc(c, n) * (c - B_ - T) - Un(c, n) * n - β * xprime,
```

(continues on next page)

(continued from previous page)

```
                (Θ * n - c - G)[s0]])

        if model.transfers:
            bounds = [(0., 100), (0., 100), self.xbar, (0., 100.)]
        else:
            bounds = [(0., 100), (0., 100), self.xbar, (0., 0.)]
        out, fx, _, imode, smode = fmin_slsqp(objf, self.zFB[s0], f_eqcons=cons,
                                              bounds=bounds, full_output=True,
                                              iprint=0)

        if imode > 0:
            raise Exception(smode)

        return np.hstack([-fx, out])
```

```
import numpy as np
from scipy.interpolate import UnivariateSpline


class interpolate_wrapper:

    def __init__(self, F):
        self.F = F

    def __getitem__(self, index):
        return interpolate_wrapper(np.asarray(self.F[index]))

    def reshape(self, *args):
        self.F = self.F.reshape(*args)
        return self

    def transpose(self):
        self.F = self.F.transpose()

    def __len__(self):
        return len(self.F)

    def __call__(self, xvec):
        x = np.atleast_1d(xvec)
        shape = self.F.shape
        if len(x) == 1:
            fhat = np.hstack([f(x) for f in self.F.flatten()])
            return fhat.reshape(shape)
        else:
            fhat = np.vstack([f(x) for f in self.F.flatten()])
            return fhat.reshape(np.hstack((shape, len(x))))


class interpolator_factory:

    def __init__(self, k, s):
        self.k, self.s = k, s

    def __call__(self, xgrid, Fs):
        shape, m = Fs.shape[:-1], Fs.shape[-1]
```

(continues on next page)

(continued from previous page)

```
        Fs = Fs.reshape((-1, m))
        F = []
        xgrid = np.sort(xgrid)  # Sort xgrid
        for Fhat in Fs:
            F.append(UnivariateSpline(xgrid, Fhat, k=self.k, s=self.s))
        return interpolate_wrapper(np.array(F).reshape(shape))


def fun_vstack(fun_list):

    Fs = [IW.F for IW in fun_list]
    return interpolate_wrapper(np.vstack(Fs))


def fun_hstack(fun_list):

    Fs = [IW.F for IW in fun_list]
    return interpolate_wrapper(np.hstack(Fs))


def simulate_markov(π, s_0, T):

    sHist = np.empty(T, dtype=int)
    sHist[0] = s_0
    S = len(π)
    for t in range(1, T):
        sHist[t] = np.random.choice(np.arange(S), p=π[sHist[t - 1]])

    return sHist
```

47.5 Reverse Engineering Strategy

We can reverse engineer a value b_0 of initial debt due that renders the AMSS measurability constraints not binding from time $t = 0$ onward.

We accomplish this by recognizing that if the AMSS measurability constraints never bind, then the AMSS allocation and Ramsey plan is equivalent with that for a Lucas-Stokey economy in which for each period $t \geq 0$, the government promises to pay the **same** state-contingent amount $\bar{b}$ in each state tomorrow.

This insight tells us to find a b_0 and other fundamentals for the Lucas-Stokey [Lucas and Stokey, 1983] model that make the Ramsey planner want to borrow the same value $\bar{b}$ next period for all states and all dates.

We accomplish this by using various equations for the Lucas-Stokey [Lucas and Stokey, 1983] model presented in *optimal taxation with state-contingent debt*.

We use the following steps.

Step 1: Pick an initial Φ.

Step 2: Given that Φ, jointly solve two versions of equation (47.4) for $c(s), s = 1, 2$ associated with the two values for $g(s), s = 1, 2$.

Step 3: Solve the following equation for $\vec{x}$

$$\vec{x} = (I - \beta\Pi)^{-1}[\vec{u}_c(\vec{n} - \vec{g}) - \vec{u}_l\vec{n}] \tag{47.6}$$

Step 4: After solving for $\vec{x}$, we can find $b(s_t|s^{t-1})$ in Markov state $s_t = s$ from $b(s) = \frac{x(s)}{u_c(s)}$ or the matrix equation

$$\vec{b} = \frac{\vec{x}}{\vec{u}_c} \tag{47.7}$$

Step 5: Compute $J(\Phi) = (b(1) - b(2))^2$.

Step 6: Put steps 2 through 6 in a function minimizer and find a Φ that minimizes $J(\Phi)$.

Step 7: At the value of Φ and the value of $\bar{b}$ that emerged from step 6, solve equations (47.5) and (47.3) jointly for c_0, b_0.

47.6 Code for Reverse Engineering

Here is code to do the calculations for us.

```
u = CRRAutility()

def min_Φ(Φ):

    g1, g2 = u.G  # Government spending in s=0 and s=1

    # Solve Φ(c)
    def equations(unknowns, Φ):
        c1, c2 = unknowns
        # First argument of .Uc and second argument of .Un are redundant

        # Set up simultaneous equations
        eq = lambda c, g: (1 + Φ) * (u.Uc(c, 1) - -u.Un(1, c + g)) + \
                          Φ * ((c + g) * u.Unn(1, c + g) + c * u.Ucc(c, 1))

        # Return equation evaluated at s=1 and s=2
        return np.array([eq(c1, g1), eq(c2, g2)]).flatten()

    global c1                 # Update c1 globally
    global c2                 # Update c2 globally

    c1, c2 = fsolve(equations, np.ones(2), args=(Φ))

    uc = u.Uc(np.array([c1, c2]), 1)                              # uc(n - g)
    # ul(n) = -un(c + g)
    ul = -u.Un(1, np.array([c1 + g1, c2 + g2])) * [c1 + g1, c2 + g2]
    # Solve for x
    x = np.linalg.solve(np.eye((2)) - u.β * u.π, uc * [c1, c2] - ul)

    global b                  # Update b globally
    b = x / uc
    loss = (b[0] - b[1])**2

    return loss

Φ_star = fmin(min_Φ, .1, ftol=1e-14)
```

```
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 24
         Function evaluations: 48
```

To recover and print out $\bar{b}$

```
b_bar = b[0]
b_bar
```

```
-1.0757576567504166
```

To complete the reverse engineering exercise by jointly determining c_0, b_0, we set up a function that returns two simultaneous equations.

```
def solve_cb(unknowns, Φ, b_bar, s=1):

    c0, b0 = unknowns

    g0 = u.G[s-1]

    R_0 = u.β * u.π[s] @ [u.Uc(c1, 1) / u.Uc(c0, 1), u.Uc(c2, 1) / u.Uc(c0, 1)]
    R_0 = 1 / R_0

    τ_0 = 1 + u.Un(1, c0 + g0) / u.Uc(c0, 1)

    eq1 = τ_0 * (c0 + g0) + b_bar / R_0 - b0 - g0
    eq2 = (1 + Φ) * (u.Uc(c0, 1)  + u.Un(1, c0 + g0)) \
           + Φ * (c0 * u.Ucc(c0, 1) + (c0 + g0) * u.Unn(1, c0 + g0)) \
           - Φ * u.Ucc(c0, 1) * b0

    return np.array([eq1, eq2.item()], dtype='float64')
```

To solve the equations for c_0, b_0, we use SciPy's fsolve function

```
c0, b0 = fsolve(solve_cb, np.array([1., -1.], dtype='float64'),
                args=(Φ_star, b[0], 1), xtol=1.0e-12)
c0, b0
```

```
(0.9344994030900681, -1.0386984075517638)
```

Thus, we have reverse engineered an initial $b0 = -1.038698407551764$ that ought to render the AMSS measurability constraints slack.

47.7 Short Simulation for Reverse-engineered: Initial Debt

The following graph shows simulations of outcomes for both a Lucas-Stokey economy and for an AMSS economy starting from initial government debt equal to $b_0 = -1.038698407551764$.

These graphs report outcomes for both the Lucas-Stokey economy with complete markets and the AMSS economy with one-period risk-free debt only.

```
μ_grid = np.linspace(-0.09, 0.1, 100)

log_example = CRRAutility()

log_example.transfers = True                    # Government can use transfers
log_sequential = SequentialAllocation(log_example)  # Solve sequential problem
```

(continues on next page)

(continued from previous page)

```
log_bellman = RecursiveAllocationAMSS(log_example, μ_grid,
                                      tol_diff=1e-10, tol=1e-10)

T = 20
sHist = np.array([0, 0, 0, 0, 0, 0, 0, 0, 1, 1,
                  0, 0, 0, 1, 1, 1, 1, 1, 1, 0])


sim_seq = log_sequential.simulate(-1.03869841, 0, T, sHist)
sim_bel = log_bellman.simulate(-1.03869841, 0, T, sHist)

titles = ['Consumption', 'Labor Supply', 'Government Debt',
          'Tax Rate', 'Government Spending', 'Output']

# Government spending paths
sim_seq[4] = log_example.G[sHist]
sim_bel[4] = log_example.G[sHist]

# Output paths
sim_seq[5] = log_example.Θ[sHist] * sim_seq[1]
sim_bel[5] = log_example.Θ[sHist] * sim_bel[1]

fig, axes = plt.subplots(3, 2, figsize=(14, 10))

for ax, title, seq, bel in zip(axes.flatten(), titles, sim_seq, sim_bel):
    ax.plot(seq, '-ok', bel, '-^b')
    ax.set(title=title)
    ax.grid()

axes[0, 0].legend(('Complete Markets', 'Incomplete Markets'))
plt.tight_layout()
plt.show()
```

```
/home/runner/miniconda3/envs/quantecon/lib/python3.12/site-packages/scipy/optimize/
 ↪_slsqp_py.py:437: RuntimeWarning: Values in x were outside bounds during a␣
 ↪minimize step, clipping to bounds
  fx = wrapped_fun(x)
/home/runner/miniconda3/envs/quantecon/lib/python3.12/site-packages/scipy/optimize/
 ↪_slsqp_py.py:441: RuntimeWarning: Values in x were outside bounds during a␣
 ↪minimize step, clipping to bounds
  g = append(wrapped_grad(x), 0.0)
/home/runner/miniconda3/envs/quantecon/lib/python3.12/site-packages/scipy/optimize/
 ↪_slsqp_py.py:495: RuntimeWarning: Values in x were outside bounds during a␣
 ↪minimize step, clipping to bounds
  a_eq = vstack([con['jac'](x, *con['args'])
```

```
/tmp/ipykernel_6863/108196118.py:24: RuntimeWarning: divide by zero encountered in␣
 ↪reciprocal
  U = (c**(1 - σ) - 1) / (1 - σ)
/tmp/ipykernel_6863/108196118.py:29: RuntimeWarning: divide by zero encountered in␣
 ↪power
  return c**(-self.σ)
/tmp/ipykernel_6863/1277371586.py:249: RuntimeWarning: invalid value encountered␣
 ↪in divide
  x * u_c / Eu_c - u_c * (c - T) - Un(c, n) * n - β * xprime,
```

(continues on next page)

(continued from previous page)

```
/tmp/ipykernel_6863/1277371586.py:249: RuntimeWarning: invalid value encountered␣
↪in multiply
  x * u_c / Eu_c - u_c * (c - T) - Un(c, n) * n - β * xprime,
```

```
0.04094445433232542
```

```
0.001673211146137493
```

```
0.001484674847917127
```

```
0.001313772136887205
```

```
0.0011814037130420663
```

```
0.001055965336102068
```

```
0.0009446661649946108
```

```
0.0008463807319492324
```

```
0.0007560453788611131
```

```
0.0006756001033938903
```

```
0.000604152845540819
```

```
0.0005396004518747859
```

```
0.00048207169166290613
```

```
0.00043082732064067867
```

```
0.00038481851351225495
```

```
0.000343835217593145
```

```
0.0003072436935049677
```

```
0.0002745009146233244
```

```
0.00024531773293589513
```

```
0.00021923324298642947
```

```
0.00019593539310787213
```

```
0.00017514303481690137
```

```
0.0001565593985003591
```

```
0.00013996737081815812
```

```
0.00012514457789841946
```

```
0.00011190070823325749
```

```
0.0001000702000922041
```

```
8.949728534363834e-05
```

```
8.00497532414663e-05
```

```
7.160585250570457e-05
```

```
6.405840591557493e-05
```

```
5.731160522780524e-05
```

```
5.1279701373366633e-05
```

```
4.588651722582404e-05
```

```
4.106390497232627e-05
```

```
3.6750969979187823e-05
```

```
3.289357328148953e-05
```

```
2.9443322731171715e-05
```

```
2.6356778254647064e-05
```

```
2.3595477005441402e-05
```

```
2.1124867549068547e-05
```

```
1.8914292342161616e-05
```

```
1.6935989661294087e-05
```

```
1.5165570482803087e-05
```

```
1.3581075188566359e-05
```

```
1.2162766163347089e-05
```

```
1.0893227516817513e-05
```

```
9.756678182519297e-06
```

```
8.739234428152772e-06
```

```
7.828320614508025e-06
```

```
7.012602839408298e-06
```

```
6.282198813865695e-06
```

```
5.628118884533389e-06
```

```
5.0424276120745635e-06
```

```
4.517800318375349e-06
```

```
4.048011435284343e-06
```

```
3.6271819852132397e-06
```

```
3.250228025571809e-06
```

```
2.91255521672949e-06
```

```
2.6100632205124585e-06
```

```
2.339096372677708e-06
```

```
2.096300057053759e-06
```

```
1.8787856014677842e-06
```

```
1.6838896002658147e-06
```

```
1.5092763000475938e-06
```

```
1.352790440377663e-06
```

```
1.2125870135921682e-06
```

```
1.0869367592654264e-06
```

```
9.74329344948381e-07
```

```
8.734258726613521e-07
```

```
7.82979401245993e-07
```

```
7.019280421759928e-07
```

```
6.292786681149374e-07
```

```
5.641636376342722e-07
```

```
5.058008139530142e-07
```

```
4.5348427330256424e-07
```

```
4.0659062310367744e-07
```

```
3.6455314441729855e-07
```

```
3.2687002299145745e-07
```

```
2.930882045255147e-07
```

```
2.6280345786809706e-07
```

```
2.356529429295176e-07
```

```
2.1131168850248635e-07
1.8948851788438695e-07
```

```
1.6992245629426705e-07
```

```
1.5237965358488245e-07
```

```
1.3665054480740185e-07
```

```
1.2254729288266142e-07
```

```
1.0990157880047098e-07
```

```
9.85625196806722e-08
```

```
8.839490296454315e-08
```

```
7.927751099544721e-08
7.110169892267009e-08
```

```
6.377012234144897e-08
```

```
5.719543299951795e-08
```

```
5.129944108294742e-08
```

```
4.6011930465755267e-08
```

```
4.127024907212617e-08
```

```
3.7017901411273995e-08
```

```
3.320421136675924e-08
2.9783836454122435e-08
```

```
2.6716185879207155e-08
2.3964828404060055e-08
```

```
2.1497111441656643e-08
```

```
1.928376711102591e-08
1.7298534286134342e-08
```

```
1.5517887041510468e-08
1.3920711115842077e-08
```

```
1.2488086772484325e-08
```

```
1.120303914946054e-08
```

```
1.0050349805051883e-08
```

```
9.016372957223345e-09
8.088867717275256e-09
```

```
7.256860052028448e-09
6.5105080491085e-09
```

```
5.8409842196277625e-09
```

```
5.240371187393206e-09
4.701571286205833e-09
```

```
4.2182149401635156e-09
```

```
3.784594252430241e-09
3.3955835551064364e-09
```

```
3.0465910785331343e-09
```

```
2.7334965385949916e-09
```

```
2.4526029798499404e-09
2.2005967896788517e-09
```

```
1.9745023230252437e-09
```

```
1.7716540861495694e-09
1.5896779606666392e-09
```

```
1.4263644656786832e-09
```

```
1.279915801041798e-09
```

```
1.1484611488603225e-09
1.0305702313922867e-09
```

```
9.247647878021015e-10
```

```
8.298468061604299e-10
7.446744286173443e-10
```

```
6.682506157688693e-10
```

```
5.996765544062293e-10
```

```
5.381420956749845e-10
4.829271458904042e-10
```

```
4.3337871811544764e-10
3.8891892933983235e-10
```

```
3.4902066124392655e-10
3.1321799130111273e-10
```

```
2.8109002457092086e-10
```

```
2.5225950288597284e-10
2.263868938948011e-10
```

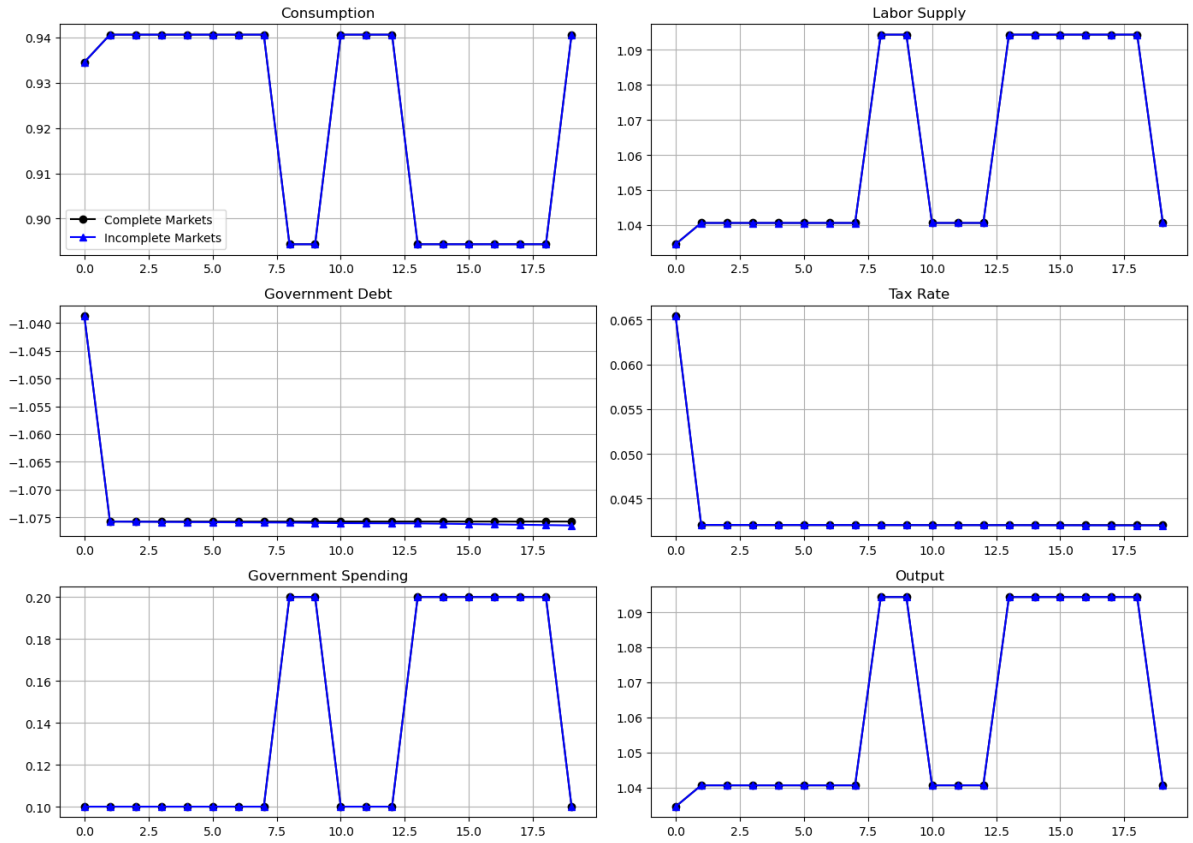
```
2.0316830484184638e-10
1.8233409175417047e-10
```

```
1.6363582056463494e-10
1.4685617665861112e-10
```

```
1.3179940303096093e-10
```

```
1.1828486777347211e-10
1.0615888599012755e-10
```

```
9.527490070407684e-11
```



The Ramsey allocations and Ramsey outcomes are **identical** for the Lucas-Stokey and AMSS economies.

This outcome confirms the success of our reverse-engineering exercises.

Notice how for $t \geq 1$, the tax rate is a constant - so is the par value of government debt.

However, output and labor supply are both nontrivial time-invariant functions of the Markov state.

47.8 Long Simulation

The following graph shows the par value of government debt and the flat-rate tax on labor income for a long simulation for our sample economy.

For the **same** realization of a government expenditure path, the graph reports outcomes for two economies

- the gray lines are for the Lucas-Stokey economy with complete markets
- the blue lines are for the AMSS economy with risk-free one-period debt only

For both economies, initial government debt due at time 0 is $b_0 = .5$.

For the Lucas-Stokey complete markets economy, the government debt plotted is $b_{t+1}(s_{t+1})$.

- Notice that this is a time-invariant function of the Markov state from the beginning.

For the AMSS incomplete markets economy, the government debt plotted is $b_{t+1}(s^t)$.

- Notice that this is a martingale-like random process that eventually seems to converge to a constant $\bar{b} \approx -1.07$.
- Notice that the limiting value $\bar{b} < 0$ so that asymptotically the government makes a constant level of risk-free loans to the public.

- In the simulation displayed as well as other simulations we have run, the par value of government debt converges to about 1.07 after between 1400 to 2000 periods.

For the AMSS incomplete markets economy, the marginal tax rate on labor income τ_t converges to a constant

- labor supply and output each converge to time-invariant functions of the Markov state

```
T = 2000  # Set T to 200 periods

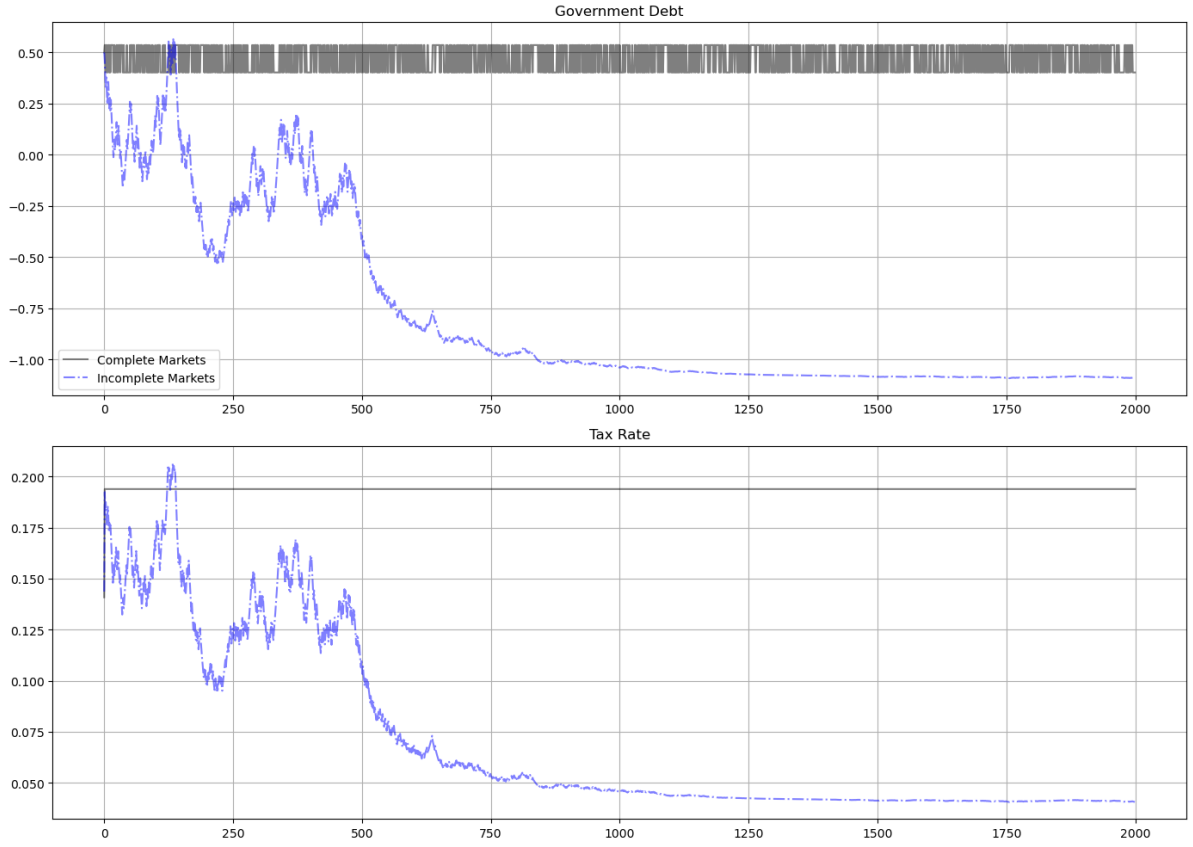
sim_seq_long = log_sequential.simulate(0.5, 0, T)
sHist_long = sim_seq_long[-3]
sim_bel_long = log_bellman.simulate(0.5, 0, T, sHist_long)

titles = ['Government Debt', 'Tax Rate']

fig, axes = plt.subplots(2, 1, figsize=(14, 10))

for ax, title, id in zip(axes.flatten(), titles, [2, 3]):
    ax.plot(sim_seq_long[id], '-k', sim_bel_long[id], '-.b', alpha=0.5)
    ax.set(title=title)
    ax.grid()

axes[0].legend(('Complete Markets', 'Incomplete Markets'))
plt.tight_layout()
plt.show()
```



47.8.1 Remarks about Long Simulation

As remarked above, after $b_{t+1}(s^t)$ has converged to a constant, the measurability constraints in the AMSS model cease to bind

- the associated Lagrange multipliers on those implementability constraints converge to zero

This leads us to seek an initial value of government debt b_0 that renders the measurability constraints slack from time $t = 0$ onward

- a tell-tale sign of this situation is that the Ramsey planner in a corresponding Lucas-Stokey economy would instruct the government to issue a constant level of government debt $b_{t+1}(s_{t+1})$ across the two Markov states

We now describe how to find such an initial level of government debt.

47.9 BEGS Approximations of Limiting Debt and Convergence Rate

It is useful to link the outcome of our reverse engineering exercise to limiting approximations constructed by BEGS [Bhandari *et al.*, 2017].

BEGS [Bhandari *et al.*, 2017] used a slightly different notation to represent a generalization of the AMSS model.

We'll introduce a version of their notation so that readers can quickly relate notation that appears in their key formulas to the notation that we have used.

BEGS work with objects $B_t, \mathcal{B}_t, \mathcal{R}_t, \mathcal{X}_t$ that are related to our notation by

$$
\begin{aligned}
\mathcal{R}_t &= \frac{u_{c,t}}{u_{c,t-1}} R_{t-1} = \frac{u_{c,t}}{\beta E_{t-1} u_{c,t}} \\
B_t &= \frac{b_{t+1}(s^t)}{R_t(s^t)} \\
b_t(s^{t-1}) &= \mathcal{R}_{t-1} B_{t-1} \\
\mathcal{B}_t &= u_{c,t} B_t = (\beta E_t u_{c,t+1}) b_{t+1}(s^t) \\
\mathcal{X}_t &= u_{c,t}[g_t - \tau_t n_t]
\end{aligned}
$$

In terms of their notation, equation (44) of [Bhandari *et al.*, 2017] expresses the time t state s government budget constraint as

$$
\mathcal{B}(s) = \mathcal{R}_\tau(s, s_-)\mathcal{B}_- + \mathcal{X}_{\tau(s)}(s) \tag{47.8}
$$

where the dependence on τ is to remind us that these objects depend on the tax rate and s_- is last period's Markov state.

BEGS interpret random variations in the right side of (47.8) as a measure of **fiscal risk** composed of

- interest-rate-driven fluctuations in time t effective payments due on the government portfolio, namely, $\mathcal{R}_\tau(s, s_-)\mathcal{B}_-$, and
- fluctuations in the effective government deficit $\mathcal{X}_t$

47.9.1 Asymptotic Mean

BEGS give conditions under which the ergodic mean of $\mathcal{B}_t$ is

$$\mathcal{B}^* = -\frac{\text{cov}^\infty(\mathcal{R}, \mathcal{X})}{\text{var}^\infty(\mathcal{R})} \tag{47.9}$$

where the superscript ∞ denotes a moment taken with respect to an ergodic distribution.

Formula (47.9) presents $\mathcal{B}^*$ as a regression coefficient of $\mathcal{X}_t$ on $\mathcal{R}_t$ in the ergodic distribution.

This regression coefficient emerges as the minimizer for a variance-minimization problem:

$$\mathcal{B}^* = \text{argmin}_{\mathcal{B}} \text{var}(\mathcal{R}\mathcal{B} + \mathcal{X}) \tag{47.10}$$

The minimand in criterion (47.10) is the measure of fiscal risk associated with a given tax-debt policy that appears on the right side of equation (47.8).

Expressing formula (47.9) in terms of our notation tells us that $\bar{b}$ should approximately equal

$$\hat{b} = \frac{\mathcal{B}^*}{\beta E_t u_{c,t+1}} \tag{47.11}$$

47.9.2 Rate of Convergence

BEGS also derive the following approximation to the rate of convergence to $\mathcal{B}^*$ from an arbitrary initial condition.

$$\frac{E_t(\mathcal{B}_{t+1} - \mathcal{B}^*)}{(\mathcal{B}_t - \mathcal{B}^*)} \approx \frac{1}{1 + \beta^2 \text{var}(\mathcal{R})} \tag{47.12}$$

(See the equation above equation (47) in [Bhandari *et al.*, 2017])

47.9.3 Formulas and Code Details

For our example, we describe some code that we use to compute the steady state mean and the rate of convergence to it.

The values of $\pi(s)$ are 0.5, 0.5.

We can then construct $\mathcal{X}(s), \mathcal{R}(s), u_c(s)$ for our two states using the definitions above.

We can then construct $\beta E_{t-1} u_c = \beta \sum_s u_c(s)\pi(s)$, $\text{cov}(\mathcal{R}(s), \mathcal{X}(s))$ and $\text{var}(\mathcal{R}(s))$ to be plugged into formula (47.11).

We also want to compute $\text{var}(\mathcal{X})$.

To compute the variances and covariance, we use the following standard formulas.

Temporarily let $x(s), s = 1, 2$ be an arbitrary random variables.

Then we define

$$\begin{aligned}
\mu_x &= \sum_s x(s)\pi(s) \\
\text{var}(x) &= \left(\sum_s \sum_s x(s)^2 \pi(s) \right) - \mu_x^2 \\
\text{cov}(x, y) &= \left(\sum_s x(s) y(s) \pi(s) \right) - \mu_x \mu_y
\end{aligned}$$

After we compute these moments, we compute the BEGS approximation to the asymptotic mean $\hat{b}$ in formula (47.11).

After that, we move on to compute $\mathcal{B}^*$ in formula (47.9).

We'll also evaluate the BEGS criterion (47.8) at the limiting value $\mathcal{B}^*$

$$J(\mathcal{B}^*) = \text{var}(\mathcal{R})\left(\mathcal{B}^*\right)^2 + 2\mathcal{B}^*\text{cov}(\mathcal{R},\mathcal{X}) + \text{var}(\mathcal{X}) \tag{47.13}$$

Here are some functions that we'll use to compute key objects that we want

```
def mean(x):
    '''Returns mean for x given initial state'''
    x = np.array(x)
    return x @ u.π[s]

def variance(x):
    x = np.array(x)
    return x**2 @ u.π[s] - mean(x)**2

def covariance(x, y):
    x, y = np.array(x), np.array(y)
    return x * y @ u.π[s] - mean(x) * mean(y)
```

Now let's form the two random variables $\mathcal{R}, \mathcal{X}$ appearing in the BEGS approximating formulas

```
u = CRRAutility()

s = 0
c = [0.940580824225584, 0.8943592757759343]  # Vector for c
g = u.G       # Vector for g
n = c + g     # Total population
τ = lambda s: 1 + u.Un(1, n[s]) / u.Uc(c[s], 1)

R_s = lambda s: u.Uc(c[s], n[s]) / (u.β * (u.Uc(c[0], n[0]) * u.π[0, 0] \
                + u.Uc(c[1], n[1]) * u.π[1, 0]))
X_s = lambda s: u.Uc(c[s], n[s]) * (g[s] - τ(s) * n[s])

R = [R_s(0), R_s(1)]
X = [X_s(0), X_s(1)]

print(f"R, X = {R}, {X}")
```

```
R, X = [1.055169547122964, 1.1670526750992583], [0.06357685646224803, 0.
↪19251010100512958]
```

Now let's compute the ingredient of the approximating limit and the approximating rate of convergence

```
bstar = -covariance(R, X) / variance(R)
div = u.β * (u.Uc(c[0], n[0]) * u.π[s, 0] + u.Uc(c[1], n[1]) * u.π[s, 1])
bhat = bstar / div
bhat
```

```
-1.0757585378303758
```

Print out $\hat{b}$ and $\bar{b}$

```
bhat, b_bar
```

```
(-1.0757585378303758, -1.0757576567504166)
```

So we have

```
bhat - b_bar
```

```
-8.810799592140484e-07
```

These outcomes show that $\hat{b}$ does a remarkably good job of approximating $\bar{b}$.

Next, let's compute the BEGS fiscal criterion that $\hat{b}$ is minimizing

```
Jmin = variance(R) * bstar**2 + 2 * bstar * covariance(R, X) + variance(X)
Jmin
```

```
-9.020562075079397e-17
```

This is *machine zero*, a verification that $\hat{b}$ succeeds in minimizing the nonnegative fiscal cost criterion $J(\mathcal{B}^*)$ defined in BEGS and in equation (47.13) above.

Let's push our luck and compute the mean reversion speed in the formula above equation (47) in [Bhandari *et al.*, 2017].

```
den2 = 1 + (u.β**2) * variance(R)
speedrever = 1/den2
print(f'Mean reversion speed = {speedrever}')
```

```
Mean reversion speed = 0.9974715478249827
```

Now let's compute the implied meantime to get to within 0.01 of the limit

```
ttime = np.log(.01) / np.log(speedrever)
print(f"Time to get within .01 of limit = {ttime}")
```

```
Time to get within .01 of limit = 1819.0360880098472
```

The slow rate of convergence and the implied time of getting within one percent of the limiting value do a good job of approximating our long simulation above.

In *a subsequent lecture* we shall study an extension of the model in which the force highlighted in this lecture causes government debt to converge to a nontrivial distribution instead of the single debt level discovered here.

CHAPTER

FORTYEIGHT

FISCAL RISK AND GOVERNMENT DEBT

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install --upgrade quantecon
```

48.1 Overview

This lecture studies government debt in an AMSS economy [Aiyagari *et al.*, 2002] of the type described in *Optimal Taxation without State-Contingent Debt*.

We study the behavior of government debt as time $t \to +\infty$.

We use these techniques

- simulations
- a regression coefficient from the tail of a long simulation that allows us to verify that the asymptotic mean of government debt solves a fiscal-risk minimization problem
- an approximation to the **mean** of an ergodic distribution of government debt
- an approximation to the **rate of convergence** to an ergodic distribution of government debt

We apply tools that are applicable to more general incomplete markets economies that are presented on pages 648 - 650 in section III.D of [Bhandari *et al.*, 2017] (BEGS).

We study an AMSS economy [Aiyagari *et al.*, 2002] with three Markov states driving government expenditures.

- In a *previous lecture*, we showed that with only two Markov states, it is possible that endogenous interest rate fluctuations eventually can support complete markets allocations and Ramsey outcomes.
- The presence of three states prevents the full spanning that eventually prevails in the two-state example featured in *Fiscal Insurance via Fluctuating Interest Rates*.

The lack of full spanning means that the ergodic distribution of the par value of government debt is nontrivial, in contrast to the situation in *Fiscal Insurance via Fluctuating Interest Rates* in which the ergodic distribution of the par value of government debt is concentrated on one point.

Nevertheless, [Bhandari *et al.*, 2017] (BEGS) establish that, for general settings that include ours, the Ramsey planner steers government assets to a level that comes **as close as possible** to providing full spanning in a precise a sense defined by BEGS that we describe below.

We use code constructed in *Fluctuating Interest Rates Deliver Fiscal Insurance*.

Warning: Key equations in [Bhandari *et al.*, 2017] section III.D carry typos that we correct below.

Let's start with some imports:

```
import matplotlib.pyplot as plt
from scipy.optimize import minimize
```

48.2 The Economy

As in *Optimal Taxation without State-Contingent Debt* and *Optimal Taxation with State-Contingent Debt*, we assume that the representative agent has utility function

$$u(c, n) = \frac{c^{1-\sigma}}{1-\sigma} - \frac{n^{1+\gamma}}{1+\gamma}$$

We work directly with labor supply instead of leisure.

We assume that

$$c_t + g_t = n_t$$

The Markov state s_t takes **three** values, namely, $0, 1, 2$.

The initial Markov state is 0.

The Markov transition matrix is $(1/3)I$ where I is a 3×3 identity matrix, so the s_t process is IID.

Government expenditures $g(s)$ equal .1 in Markov state 0, .2 in Markov state 1, and .3 in Markov state 2.

We set preference parameters

$$\beta = .9$$
$$\sigma = 2$$
$$\gamma = 2$$

The following Python code sets up the economy

```
import numpy as np


class CRRAutility:

    def __init__(self,
                 β=0.9,
                 σ=2,
                 γ=2,
                 π=np.full((2, 2), 0.5),
                 G=np.array([0.1, 0.2]),
                 Θ=np.ones(2),
                 transfers=False):

        self.β, self.σ, self.γ = β, σ, γ
        self.π, self.G, self.Θ, self.transfers = π, G, Θ, transfers

    # Utility function
    def U(self, c, n):
        σ = self.σ
        if σ == 1.:
            U = np.log(c)
```

(continues on next page)

(continued from previous page)

```
        else:
            U = (c**(1 - σ) - 1) / (1 - σ)
        return U - n**(1 + self.γ) / (1 + self.γ)

    # Derivatives of utility function
    def Uc(self, c, n):
        return c**(-self.σ)

    def Ucc(self, c, n):
        return -self.σ * c**(-self.σ - 1)

    def Un(self, c, n):
        return -n**self.γ

    def Unn(self, c, n):
        return -self.γ * n**(self.γ - 1)
```

48.2.1 First and Second Moments

We'll want first and second moments of some key random variables below.

The following code computes these moments; the code is recycled from *Fluctuating Interest Rates Deliver Fiscal Insurance*.

```
def mean(x, s):
    '''Returns mean for x given initial state'''
    x = np.array(x)
    return x @ u.π[s]

def variance(x, s):
    x = np.array(x)
    return x**2 @ u.π[s] - mean(x, s)**2

def covariance(x, y, s):
    x, y = np.array(x), np.array(y)
    return x * y @ u.π[s] - mean(x, s) * mean(y, s)
```

48.3 Long Simulation

To generate a long simulation we use the following code.

We begin by showing the code that we used in earlier lectures on the AMSS model.

Here it is

```
import numpy as np
from scipy.optimize import root
from quantecon import MarkovChain


class SequentialAllocation:

    '''
```

(continues on next page)

(continued from previous page)

```
    Class that takes CESutility or BGPutility object as input returns
    planner's allocation as a function of the multiplier on the
    implementability constraint μ.
    '''

    def __init__(self, model):

        # Initialize from model object attributes
        self.β, self.π, self.G = model.β, model.π, model.G
        self.mc, self.Θ = MarkovChain(self.π), model.Θ
        self.S = len(model.π)  # Number of states
        self.model = model

        # Find the first best allocation
        self.find_first_best()

    def find_first_best(self):
        '''
        Find the first best allocation
        '''
        model = self.model
        S, Θ, G = self.S, self.Θ, self.G
        Uc, Un = model.Uc, model.Un

        def res(z):
            c = z[:S]
            n = z[S:]
            return np.hstack([Θ * Uc(c, n) + Un(c, n), Θ * n - c - G])

        res = root(res, np.full(2 * S, 0.5))

        if not res.success:
            raise Exception('Could not find first best')

        self.cFB = res.x[:S]
        self.nFB = res.x[S:]

        # Multiplier on the resource constraint
        self.ΞFB = Uc(self.cFB, self.nFB)
        self.zFB = np.hstack([self.cFB, self.nFB, self.ΞFB])

    def time1_allocation(self, μ):
        '''
        Computes optimal allocation for time t >= 1 for a given μ
        '''
        model = self.model
        S, Θ, G = self.S, self.Θ, self.G
        Uc, Ucc, Un, Unn = model.Uc, model.Ucc, model.Un, model.Unn

        def FOC(z):
            c = z[:S]
            n = z[S:2 * S]
            Ξ = z[2 * S:]
            # FOC of c
            return np.hstack([Uc(c, n) - μ * (Ucc(c, n) * c + Uc(c, n)) - Ξ,
                              Un(c, n) - μ * (Unn(c, n) * n + Un(c, n)) \
```

(continues on next page)

(continued from previous page)

```
                                + Θ * Ξ,  # FOC of n
                                Θ * n - c - G])

        # Find the root of the first-order condition
        res = root(FOC, self.zFB)
        if not res.success:
            raise Exception('Could not find LS allocation.')
        z = res.x
        c, n, Ξ = z[:S], z[S:2 * S], z[2 * S:]

        # Compute x
        I = Uc(c, n) * c + Un(c, n) * n
        x = np.linalg.solve(np.eye(S) - self.β * self.π, I)

        return c, n, x, Ξ

    def time0_allocation(self, B_, s_0):
        '''
        Finds the optimal allocation given initial government debt B_ and
        state s_0
        '''
        model, π, Θ, G, β = self.model, self.π, self.Θ, self.G, self.β
        Uc, Ucc, Un, Unn = model.Uc, model.Ucc, model.Un, model.Unn

        # First order conditions of planner's problem
        def FOC(z):
            μ, c, n, Ξ = z
            xprime = self.time1_allocation(μ)[2]
            return np.hstack([Uc(c, n) * (c - B_) + Un(c, n) * n + β * π[s_0]
                                            @ xprime,
                              Uc(c, n) - μ * (Ucc(c, n)
                                              * (c - B_) + Uc(c, n)) - Ξ,
                              Un(c, n) - μ * (Unn(c, n) * n
                                              + Un(c, n)) + Θ[s_0] * Ξ,
                              (Θ * n - c - G)[s_0]])

        # Find root
        res = root(FOC, np.array(
            [0, self.cFB[s_0], self.nFB[s_0], self.ΞFB[s_0]]))
        if not res.success:
            raise Exception('Could not find time 0 LS allocation.')

        return res.x

    def time1_value(self, μ):
        '''
        Find the value associated with multiplier μ
        '''
        c, n, x, Ξ = self.time1_allocation(μ)
        U = self.model.U(c, n)
        V = np.linalg.solve(np.eye(self.S) - self.β * self.π, U)
        return c, n, x, V

    def T(self, c, n):
        '''
        Computes T given c, n
```

(continues on next page)

(continued from previous page)

```
        '''
        model = self.model
        Uc, Un = model.Uc(c, n), model.Un(c,  n)

        return 1 + Un / (self.Θ * Uc)

    def simulate(self, B_, s_0, T, sHist=None):
        '''
        Simulates planners policies for T periods
        '''
        model, π, β = self.model, self.π, self.β
        Uc = model.Uc

        if sHist is None:
            sHist = self.mc.simulate(T, s_0)

        cHist, nHist, Bhist, THist, μHist = np.zeros((5, T))
        RHist = np.zeros(T - 1)

        # Time 0
        μ, cHist[0], nHist[0], _ = self.time0_allocation(B_, s_0)
        THist[0] = self.T(cHist[0], nHist[0])[s_0]
        Bhist[0] = B_
        μHist[0] = μ

        # Time 1 onward
        for t in range(1, T):
            c, n, x, Ξ = self.time1_allocation(μ)
            T = self.T(c, n)
            u_c = Uc(c, n)
            s = sHist[t]
            Eu_c = π[sHist[t - 1]] @ u_c
            cHist[t], nHist[t], Bhist[t], THist[t] = c[s], n[s], x[s] / u_c[s], \
                                                     T[s]
            RHist[t - 1] = Uc(cHist[t - 1], nHist[t - 1]) / (β * Eu_c)
            μHist[t] = μ

        return [cHist, nHist, Bhist, THist, sHist, μHist, RHist]
```

```
import numpy as np
from scipy.optimize import fmin_slsqp
from scipy.optimize import root
from quantecon import MarkovChain


class RecursiveAllocationAMSS:

    def __init__(self, model, μgrid, tol_diff=1e-7, tol=1e-7):

        self.β, self.π, self.G = model.β, model.π, model.G
        self.mc, self.S = MarkovChain(self.π), len(model.π)  # Number of states
        self.Θ, self.model, self.μgrid = model.Θ, model, μgrid
        self.tol_diff, self.tol = tol_diff, tol

        # Find the first best allocation
```

(continues on next page)

(continued from previous page)

```
        self.solve_time1_bellman()
        self.T.time_0 = True  # Bellman equation now solves time 0 problem

    def solve_time1_bellman(self):
        '''
        Solve the time  1 Bellman equation for calibration model and
        initial grid μgrid0
        '''
        model, μgrid0 = self.model, self.μgrid
        π = model.π
        S = len(model.π)

        # First get initial fit from Lucas Stokey solution.
        # Need to change things to be ex ante
        pp = SequentialAllocation(model)
        interp = interpolator_factory(2, None)

        def incomplete_allocation(μ_, s_):
            c, n, x, V = pp.time1_value(μ_)
            return c, n, π[s_] @ x, π[s_] @ V
        cf, nf, xgrid, Vf, xprimef = [], [], [], [], []
        for s_ in range(S):
            c, n, x, V = zip(*map(lambda μ: incomplete_allocation(μ, s_), μgrid0))
            c, n = np.vstack(c).T, np.vstack(n).T
            x, V = np.hstack(x), np.hstack(V)
            xprimes = np.vstack([x] * S)
            cf.append(interp(x, c))
            nf.append(interp(x, n))
            Vf.append(interp(x, V))
            xgrid.append(x)
            xprimef.append(interp(x, xprimes))
        cf, nf, xprimef = fun_vstack(cf), fun_vstack(nf), fun_vstack(xprimef)
        Vf = fun_hstack(Vf)
        policies = [cf, nf, xprimef]

        # Create xgrid
        x = np.vstack(xgrid).T
        xbar = [x.min(0).max(), x.max(0).min()]
        xgrid = np.linspace(xbar[0], xbar[1], len(μgrid0))
        self.xgrid = xgrid

        # Now iterate on Bellman equation
        T = BellmanEquation(model, xgrid, policies, tol=self.tol)
        diff = 1
        while diff > self.tol_diff:
            PF = T(Vf)

            Vfnew, policies = self.fit_policy_function(PF)
            diff = np.abs((Vf(xgrid) - Vfnew(xgrid)) / Vf(xgrid)).max()

            print(diff)
            Vf = Vfnew

        # Store value function policies and Bellman Equations
        self.Vf = Vf
        self.policies = policies
```

(continues on next page)

(continued from previous page)

```
        self.T = T

    def fit_policy_function(self, PF):
        '''
        Fits the policy functions
        '''
        S, xgrid = len(self.π), self.xgrid
        interp = interpolator_factory(3, 0)
        cf, nf, xprimef, Tf, Vf = [], [], [], [], []
        for s_ in range(S):
            PFvec = np.vstack([PF(x, s_) for x in self.xgrid]).T
            Vf.append(interp(xgrid, PFvec[0, :]))
            cf.append(interp(xgrid, PFvec[1:1 + S]))
            nf.append(interp(xgrid, PFvec[1 + S:1 + 2 * S]))
            xprimef.append(interp(xgrid, PFvec[1 + 2 * S:1 + 3 * S]))
            Tf.append(interp(xgrid, PFvec[1 + 3 * S:]))
        policies = fun_vstack(cf), fun_vstack(
            nf), fun_vstack(xprimef), fun_vstack(Tf)
        Vf = fun_hstack(Vf)
        return Vf, policies

    def T(self, c, n):
        '''
        Computes T given c and n
        '''
        model = self.model
        Uc, Un = model.Uc(c, n), model.Un(c, n)

        return 1 + Un / (self.Θ * Uc)

    def time0_allocation(self, B_, s0):
        '''
        Finds the optimal allocation given initial government debt B_ and
        state s_0
        '''
        PF = self.T(self.Vf)
        z0 = PF(B_, s0)
        c0, n0, xprime0, T0 = z0[1:]
        return c0, n0, xprime0, T0

    def simulate(self, B_, s_0, T, sHist=None):
        '''
        Simulates planners policies for T periods
        '''
        model, π = self.model, self.π
        Uc = model.Uc
        cf, nf, xprimef, Tf = self.policies

        if sHist is None:
            sHist = simulate_markov(π, s_0, T)

        cHist, nHist, Bhist, xHist, THist, THist, μHist = np.zeros((7, T))
        # Time 0
        cHist[0], nHist[0], xHist[0], THist[0] = self.time0_allocation(B_, s_0)
        THist[0] = self.T(cHist[0], nHist[0])[s_0]
        Bhist[0] = B_
```

(continues on next page)

(continued from previous page)

```
        μHist[0] = self.Vf[s_0](xHist[0])

        # Time 1 onward
        for t in range(1, T):
            s_, x, s = sHist[t - 1], xHist[t - 1], sHist[t]
            c, n, xprime, T = cf[s_, :](x), nf[s_, :](
                x), xprimef[s_, :](x), Tf[s_, :](x)

            T = self.T(c, n)[s]
            u_c = Uc(c, n)
            Eu_c = π[s_, :] @ u_c

            μHist[t] = self.Vf[s](xprime[s])

            cHist[t], nHist[t], Bhist[t], THist[t] = c[s], n[s], x / Eu_c, T
            xHist[t], THist[t] = xprime[s], T[s]
        return [cHist, nHist, Bhist, THist, THist, μHist, sHist, xHist]


class BellmanEquation:
    '''
    Bellman equation for the continuation of the Lucas-Stokey Problem
    '''

    def __init__(self, model, xgrid, policies0, tol, maxiter=1000):

        self.β, self.π, self.G = model.β, model.π, model.G
        self.S = len(model.π)  # Number of states
        self.Θ, self.model, self.tol = model.Θ, model, tol
        self.maxiter = maxiter

        self.xbar = [min(xgrid), max(xgrid)]
        self.time_0 = False

        self.z0 = {}
        cf, nf, xprimef = policies0

        for s_ in range(self.S):
            for x in xgrid:
                self.z0[x, s_] = np.hstack([cf[s_, :](x),
                                            nf[s_, :](x),
                                            xprimef[s_, :](x),
                                            np.zeros(self.S)])

        self.find_first_best()

    def find_first_best(self):
        '''
        Find the first best allocation
        '''
        model = self.model
        S, Θ, Uc, Un, G = self.S, self.Θ, model.Uc, model.Un, self.G

        def res(z):
            c = z[:S]
            n = z[S:]
```

(continues on next page)

(continued from previous page)

```
            return np.hstack([Θ * Uc(c, n) + Un(c, n), Θ * n - c - G])

        res = root(res, np.full(2 * S, 0.5))
        if not res.success:
            raise Exception('Could not find first best')

        self.cFB = res.x[:S]
        self.nFB = res.x[S:]
        IFB = Uc(self.cFB, self.nFB) * self.cFB + \
            Un(self.cFB, self.nFB) * self.nFB

        self.xFB = np.linalg.solve(np.eye(S) - self.β * self.π, IFB)

        self.zFB = {}
        for s in range(S):
            self.zFB[s] = np.hstack(
                [self.cFB[s], self.nFB[s], self.π[s] @ self.xFB, 0.])

    def __call__(self, Vf):
        '''
        Given continuation value function next period return value function this
        period return T(V) and optimal policies
        '''
        if not self.time_0:
            def PF(x, s): return self.get_policies_time1(x, s, Vf)
        else:
            def PF(B_, s0): return self.get_policies_time0(B_, s0, Vf)
        return PF

    def get_policies_time1(self, x, s_, Vf):
        '''
        Finds the optimal policies
        '''
        model, β, Θ, G, S, π = self.model, self.β, self.Θ, self.G, self.S, self.π
        U, Uc, Un = model.U, model.Uc, model.Un

        def objf(z):
            c, n, xprime = z[:S], z[S:2 * S], z[2 * S:3 * S]

            Vprime = np.empty(S)
            for s in range(S):
                Vprime[s] = Vf[s](xprime[s])

            return -π[s_] @ (U(c, n) + β * Vprime)

        def objf_prime(x):

            epsilon = 1e-7
            x0 = np.asfarray(x)
            f0 = np.atleast_1d(objf(x0))
            jac = np.zeros([len(x0), len(f0)])
            dx = np.zeros(len(x0))
            for i in range(len(x0)):
                dx[i] = epsilon
                jac[i] = (objf(x0+dx) - f0)/epsilon
                dx[i] = 0.0
```

(continues on next page)

(continued from previous page)

```
            return jac.transpose()

        def cons(z):
            c, n, xprime, T = z[:S], z[S:2 * S], z[2 * S:3 * S], z[3 * S:]
            u_c = Uc(c, n)
            Eu_c = π[s_] @ u_c
            return np.hstack([
                x * u_c / Eu_c - u_c * (c - T) - Un(c, n) * n - β * xprime,
                Θ * n - c - G])

        if model.transfers:
            bounds = [(0., 100)] * S + [(0., 100)] * S + \
                [self.xbar] * S + [(0., 100.)] * S
        else:
            bounds = [(0., 100)] * S + [(0., 100)] * S + \
                [self.xbar] * S + [(0., 0.)] * S
        out, fx, _, imode, smode = fmin_slsqp(objf, self.z0[x, s_],
                                              f_eqcons=cons, bounds=bounds,
                                              fprime=objf_prime, full_output=True,
                                              iprint=0, acc=self.tol, iter=self.
↪maxiter)

        if imode > 0:
            raise Exception(smode)

        self.z0[x, s_] = out
        return np.hstack([-fx, out])

    def get_policies_time0(self, B_, s0, Vf):
        '''
        Finds the optimal policies
        '''
        model, β, Θ, G = self.model, self.β, self.Θ, self.G
        U, Uc, Un = model.U, model.Uc, model.Un

        def objf(z):
            c, n, xprime = z[:-1]

            return -(U(c, n) + β * Vf[s0](xprime))

        def cons(z):
            c, n, xprime, T = z
            return np.hstack([
                -Uc(c, n) * (c - B_ - T) - Un(c, n) * n - β * xprime,
                (Θ * n - c - G)[s0]])

        if model.transfers:
            bounds = [(0., 100), (0., 100), self.xbar, (0., 100.)]
        else:
            bounds = [(0., 100), (0., 100), self.xbar, (0., 0.)]
        out, fx, _, imode, smode = fmin_slsqp(objf, self.zFB[s0], f_eqcons=cons,
                                              bounds=bounds, full_output=True,
                                              iprint=0)

        if imode > 0:
```

(continues on next page)

(continued from previous page)

```
            raise Exception(smode)

        return np.hstack([-fx, out])
```

```
import numpy as np
from scipy.interpolate import UnivariateSpline


class interpolate_wrapper:

    def __init__(self, F):
        self.F = F

    def __getitem__(self, index):
        return interpolate_wrapper(np.asarray(self.F[index]))

    def reshape(self, *args):
        self.F = self.F.reshape(*args)
        return self

    def transpose(self):
        self.F = self.F.transpose()

    def __len__(self):
        return len(self.F)

    def __call__(self, xvec):
        x = np.atleast_1d(xvec)
        shape = self.F.shape
        if len(x) == 1:
            fhat = np.hstack([f(x) for f in self.F.flatten()])
            return fhat.reshape(shape)
        else:
            fhat = np.vstack([f(x) for f in self.F.flatten()])
            return fhat.reshape(np.hstack((shape, len(x))))


class interpolator_factory:

    def __init__(self, k, s):
        self.k, self.s = k, s

    def __call__(self, xgrid, Fs):
        shape, m = Fs.shape[:-1], Fs.shape[-1]
        Fs = Fs.reshape((-1, m))
        F = []
        xgrid = np.sort(xgrid)  # Sort xgrid
        for Fhat in Fs:
            F.append(UnivariateSpline(xgrid, Fhat, k=self.k, s=self.s))
        return interpolate_wrapper(np.array(F).reshape(shape))


def fun_vstack(fun_list):

    Fs = [IW.F for IW in fun_list]
```

(continues on next page)

(continued from previous page)

```
    return interpolate_wrapper(np.vstack(Fs))


def fun_hstack(fun_list):

    Fs = [IW.F for IW in fun_list]
    return interpolate_wrapper(np.hstack(Fs))


def simulate_markov(π, s_0, T):

    sHist = np.empty(T, dtype=int)
    sHist[0] = s_0
    S = len(π)
    for t in range(1, T):
        sHist[t] = np.random.choice(np.arange(S), p=π[sHist[t - 1]])

    return sHist
```

Next, we show the code that we use to generate a very long simulation starting from initial government debt equal to $-.5$.

Here is a graph of a long simulation of 102000 periods.

```
μ_grid = np.linspace(-0.09, 0.1, 100)

log_example = CRRAutility(π=np.full((3, 3), 1 / 3),
                          G=np.array([0.1, 0.2, .3]),
                          Θ=np.ones(3))

log_example.transfers = True        # Government can use transfers
log_sequential = SequentialAllocation(log_example)  # Solve sequential problem
log_bellman = RecursiveAllocationAMSS(log_example, μ_grid,
                                      tol=1e-12, tol_diff=1e-10)



T = 102000  # Set T to 102000 periods

sim_seq_long = log_sequential.simulate(0.5, 0, T)
sHist_long = sim_seq_long[-3]
sim_bel_long = log_bellman.simulate(0.5, 0, T, sHist_long)

titles = ['Government Debt', 'Tax Rate']

fig, axes = plt.subplots(2, 1, figsize=(10, 8))

for ax, title, id in zip(axes.flatten(), titles, [2, 3]):
    ax.plot(sim_seq_long[id], '-k', sim_bel_long[id], '-.b', alpha=0.5)
    ax.set(title=title)
    ax.grid()

axes[0].legend(('Complete Markets', 'Incomplete Markets'))
plt.tight_layout()
plt.show()
```

```
/home/runner/miniconda3/envs/quantecon/lib/python3.12/site-packages/scipy/optimize/
↪_slsqp_py.py:437: RuntimeWarning: Values in x were outside bounds during a␣
↪minimize step, clipping to bounds
  fx = wrapped_fun(x)
/home/runner/miniconda3/envs/quantecon/lib/python3.12/site-packages/scipy/optimize/
↪_slsqp_py.py:441: RuntimeWarning: Values in x were outside bounds during a␣
↪minimize step, clipping to bounds
  g = append(wrapped_grad(x), 0.0)
/home/runner/miniconda3/envs/quantecon/lib/python3.12/site-packages/scipy/optimize/
↪_slsqp_py.py:495: RuntimeWarning: Values in x were outside bounds during a␣
↪minimize step, clipping to bounds
  a_eq = vstack([con['jac'](x, *con['args'])
```

```
/tmp/ipykernel_6915/108196118.py:24: RuntimeWarning: divide by zero encountered in␣
↪reciprocal
  U = (c**(1 - σ) - 1) / (1 - σ)
/tmp/ipykernel_6915/108196118.py:29: RuntimeWarning: divide by zero encountered in␣
↪power
  return c**(-self.σ)
/tmp/ipykernel_6915/1277371586.py:249: RuntimeWarning: invalid value encountered␣
↪in divide
  x * u_c / Eu_c - u_c * (c - T) - Un(c, n) * n - β * xprime,
```

```
/tmp/ipykernel_6915/1277371586.py:249: RuntimeWarning: invalid value encountered␣
↪in multiply
  x * u_c / Eu_c - u_c * (c - T) - Un(c, n) * n - β * xprime,
```

```
0.038266353387659546
```

```
0.0015144378246632448
```

```
0.0013387575049931865
```

```
0.0011833202400662248
```

```
0.0010600307116134505
```

```
0.0009506620324908642
```

```
0.0008518776517238551
```

```
0.0007625857031042564
```

```
0.0006819563061669217
```

```
0.0006094002927240671
```

```
0.0005443007356805235
```

```
0.00048599500343956094
```

```
0.0004338395935928358
```

```
0.00038722730865154364
```

```
0.00034559541217657187
```

```
0.00030842870645340995
```

```
0.00027525901875688697
```

```
0.00024566631291987257
```

```
0.00021925988533911457
```

```
0.00019570695819278 78
```

```
0.00017469751641633328
```

```
0.00015595697131045533
```

```
0.00013923987965580473
```

```
0.00012432704762446 32
```

```
0.00011102285954170156
```

```
9.915283206080047e-05
```

```
8.856139177373994e-05
```

```
7.910986485356134e-05
```

```
7.067466534026614e-05
```

```
6.314566737649043e-05
```

```
5.6424746008715835e-05
```

```
5.04244714230645e-05
```

```
4.5066942129829506e-05
```

```
4.028274354582181e-05
```

```
3.601001917066026e-05
```

```
3.219364287744318e-05
```

```
2.878448158073308e-05
```

```
2.5738738366349524e-05
```

```
2.3017369974638877e-05
```

```
2.0585562530972924e-05
```

```
1.8412273759209572e-05
```

```
1.6470096733078585e-05
```

```
1.4734148603737835e-05
```

```
1.3182214255360329e-05
```

```
1.1794654716176686e-05
```

```
1.0553942898779478e-05
```

```
9.444436197515114e-06
```

```
8.452171093491432e-06
```

```
7.564681603048501e-06
```

```
6.770836606096674e-06
```

```
6.060699172057158e-06
```

```
5.4253876343226e-06
```

```
4.856977544060761e-06
```

```
4.348382732427091e-06
```

```
3.893276456302588e-06
```

```
3.4860028420224977e-06
```

```
3.1215110784890745e-06
```

```
2.7952840260155024e-06
```

```
2.503284254157189e-06
```

```
2.241904747465382e-06
```

```
2.0079209145832687e-06
```

```
1.7984472260187192e-06
```

```
1.610904141295967e-06
```

```
1.4429883256895489e-06
```

```
1.2926354365994746e-06
```

```
1.1580011940576491e-06
```

```
1.0374362190402233e-06
```

```
9.294651286343194e-07
```

```
8.327660623755013e-07
```

```
7.461585686381671e-07
```

```
6.68586648784756e-07
```

```
5.991017296865946e-07
```

```
5.368606502407216e-07
```

```
4.811037017633464e-07
```

```
4.3115434615062044e-07
```

```
3.8640500348483447e-07
```

```
3.4631274740294855e-07
```

```
3.1039146715661056e-07
```

```
2.782060642970499e-07
```

```
2.493665449692665e-07
```

```
2.235241683944158e-07
```

```
2.0036660045892633e-07
```

```
1.796140357496926e-07
```

```
1.610161234596195e-07
```

```
1.4434845857135709e-07
```

```
1.29410194199688e-07
```

```
1.1602140686642469e-07
```

```
1.04020962175412e-07
```

```
9.326451087350253e-08
```

```
8.362279520562034e-08
```

```
7.49799979528415e-08
```

```
6.723237810210067e-08
```

```
6.028699653820159e-08
```

```
5.4060588066801066e-08
```

```
4.847855517381241e-08
```

```
4.347405660607874e-08
```

```
3.898720608840536e-08
```

```
3.496434157686767e-08
```

```
3.135737680533792e-08
```

```
2.8123222131646282e-08
```

```
2.5223262308472423e-08
```

```
2.2622892571432625e-08
```

```
2.0291098813063476e-08
```

```
1.820008555543109e-08
```

```
1.6324938418135388e-08
```

```
1.4643330672610771e-08
```

```
1.3135245110419445e-08
```

```
1.178274355586975e-08
```

```
1.0569743803546048e-08
```

```
9.48183058751907e-09
```

```
8.506079544395937e-09
```

```
7.630907318911004e-09
```

```
6.845926774203295e-09
```

```
6.141826797773109e-09
```

```
5.510259068441386e-09
```

```
4.943738281315066e-09
```

```
4.435554859709816e-09
```

```
3.979736766026741e-09
```

```
3.5708317622814044e-09
```

```
3.2040044801866767e-09
```

```
2.874916539533131e-09
```

```
2.579680212253616e-09
```

```
2.3148068175021918e-09
```

```
2.077170148801081e-09
```

```
1.8639635474165993e-09
```

```
1.6726726276855955e-09
```

```
1.5010414936033808e-09
```

```
1.3470449992327086e-09
```

```
1.2088698423920761e-09
```

```
1.0848882197883804e-09
```

```
9.736395405805598e-10
```

```
8.738135346705384e-10
```

```
7.842367703299733e-10
```

```
7.03855297579472e-10
```

```
6.317225605423774e-10
```

```
5.669925787732949e-10
```

```
5.089032105148693e-10
```

```
4.5677367318159076e-10
```

```
4.0999013116379334e-10
```

```
3.680044560697966e-10
```

```
3.3032415368561477e-10
```

```
2.96506010211222e-10
```

```
2.6615516244191936e-10
```

```
2.389139399385772e-10
```

```
2.144649644252697e-10
```

```
1.9252092177853976e-10
```

```
1.7282471699749249e-10
```

```
1.551454449875162e-10
```

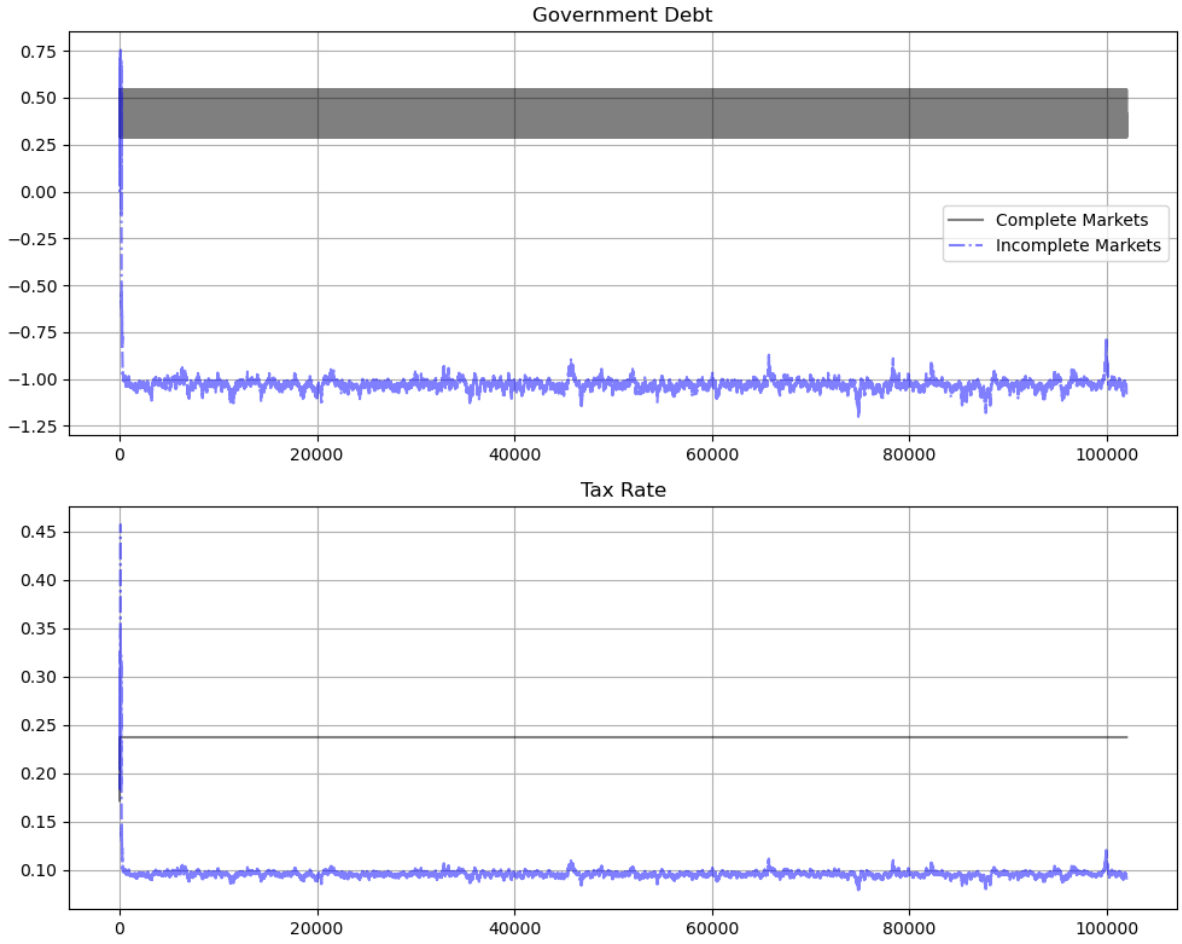
```
1.3927730577138407e-10
```

```
1.2503449048385917e-10
```

```
1.1224916676355658e-10
```

```
1.0077318342152794e-10
```

```
9.047094182757221e-11
```



The long simulation apparently indicates eventual convergence to an ergodic distribution.

It takes about 1000 periods to reach the ergodic distribution – an outcome that is forecast by approximations to rates of convergence that appear in BEGS [Bhandari *et al.*, 2017] and that we discuss in *Fluctuating Interest Rates Deliver Fiscal Insurance*.

Let's discard the first 2000 observations of the simulation and construct the histogram of the par value of government debt.

We obtain the following graph for the histogram of the last 100,000 observations on the par value of government debt.

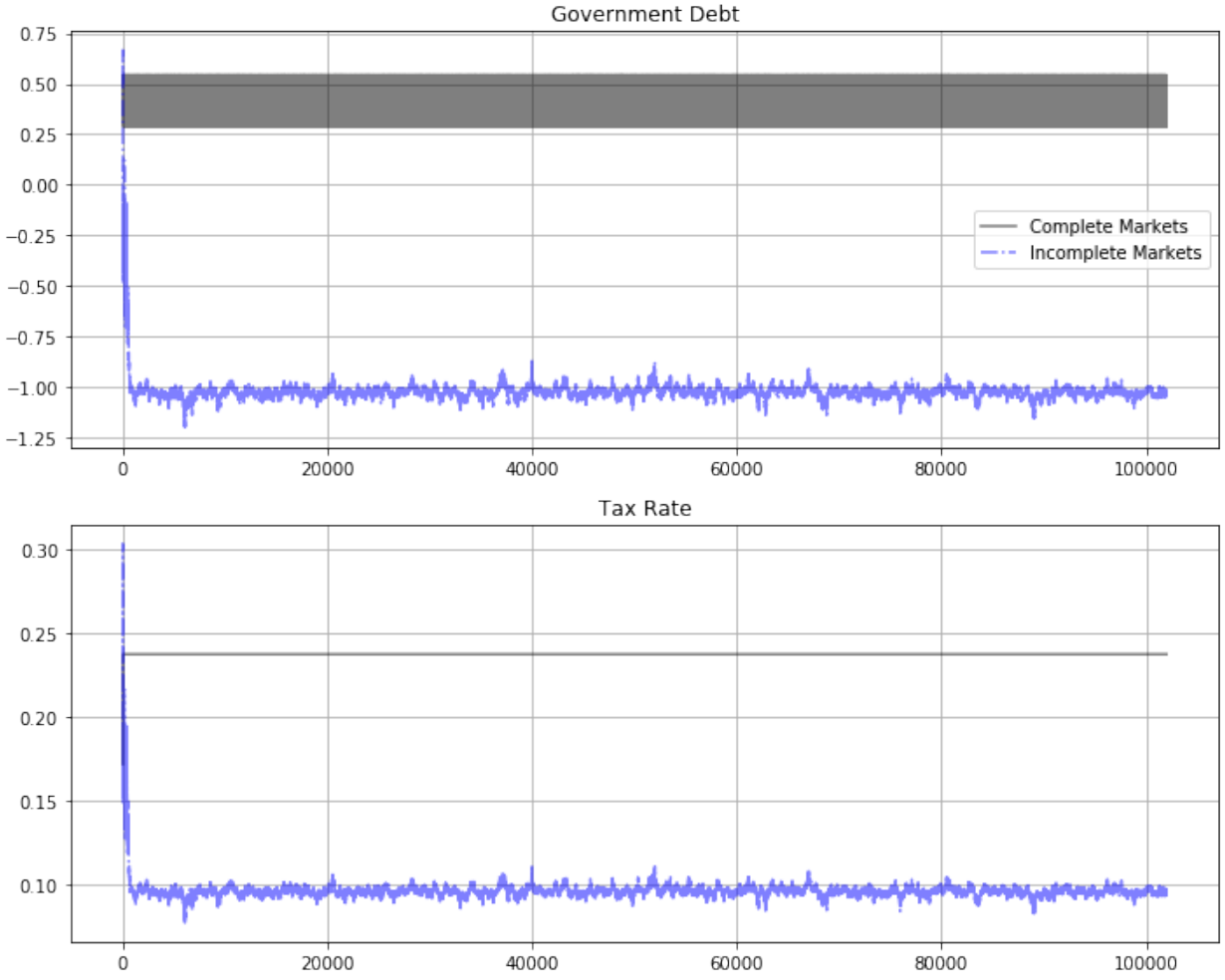
The black vertical line denotes the sample mean for the last 100,000 observations included in the histogram; the green vertical line denotes the value of $\frac{\mathcal{B}^*}{Eu_c}$, associated with a sample from our approximation to the ergodic distribution where $\mathcal{B}^*$ is a regression coefficient to be described below; the red vertical line denotes an approximation by [Bhandari *et al.*, 2017] to the mean of the ergodic distribution that can be computed **before** the ergodic distribution has been approximated, as described below.

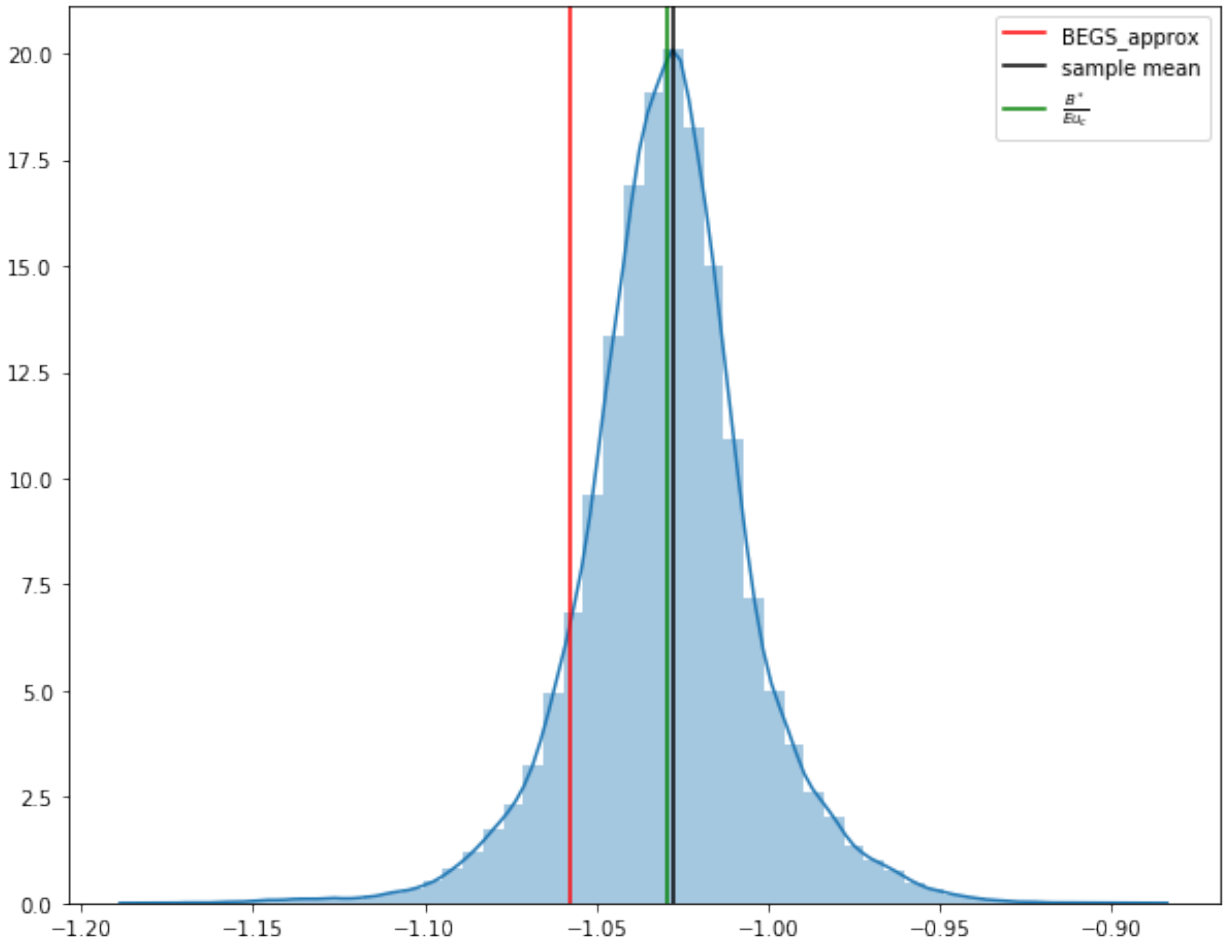
Before moving on to discuss the histogram and the vertical lines approximating the ergodic mean of government debt in more detail, the following graphs show government debt and taxes early in the simulation, for periods 1-100 and 101 to 200 respectively.

```
titles = ['Government Debt', 'Tax Rate']

fig, axes = plt.subplots(4, 1, figsize=(10, 15))
```

(continues on next page)

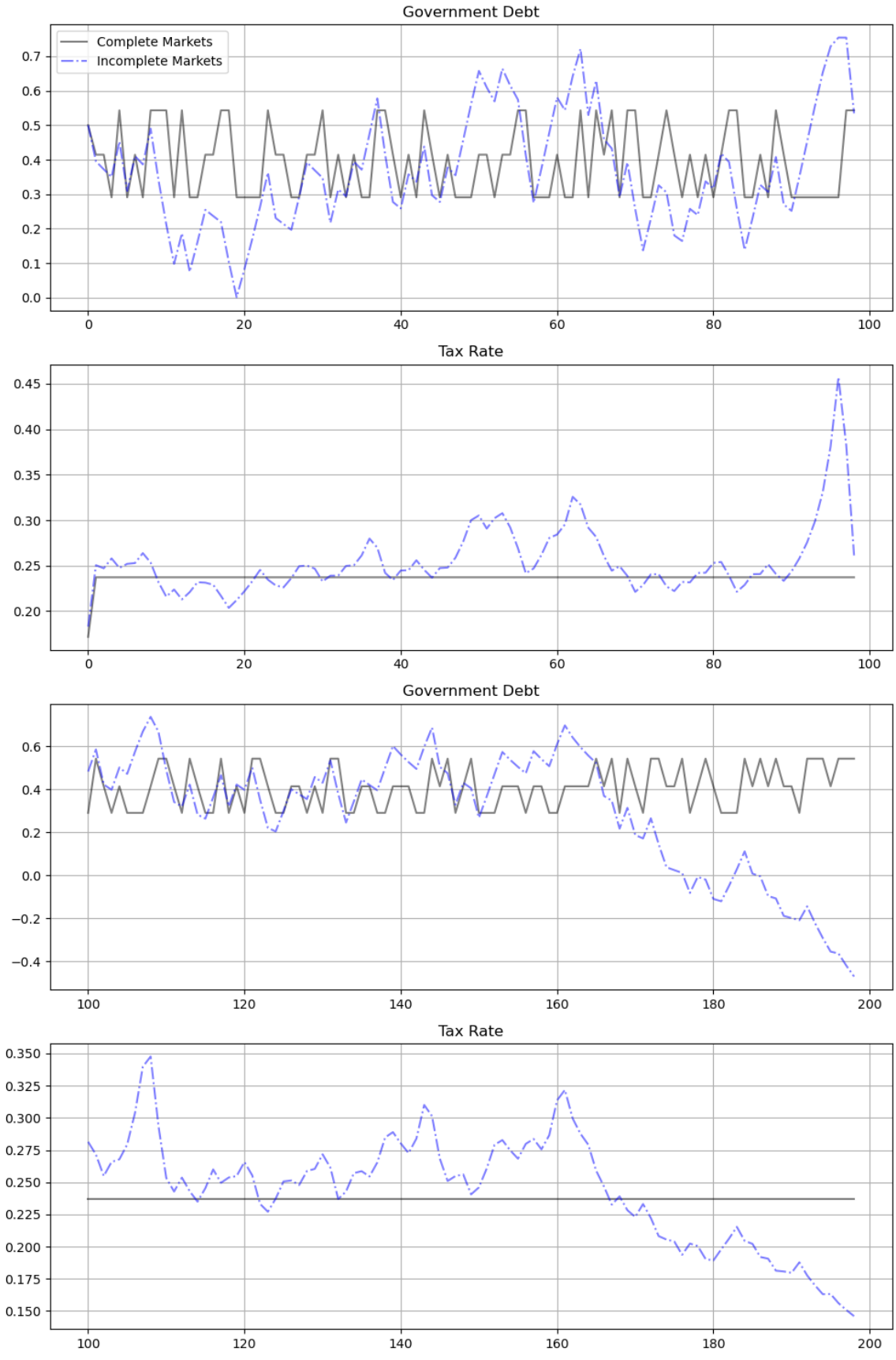

Government Debt
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.25
0
20000
40000
60000
80000
100000
Complete Markets
Incomplete Markets
Tax Rate
0.30
0.25
0.20
0.15
0.10
0
20000
40000
60000
80000
100000

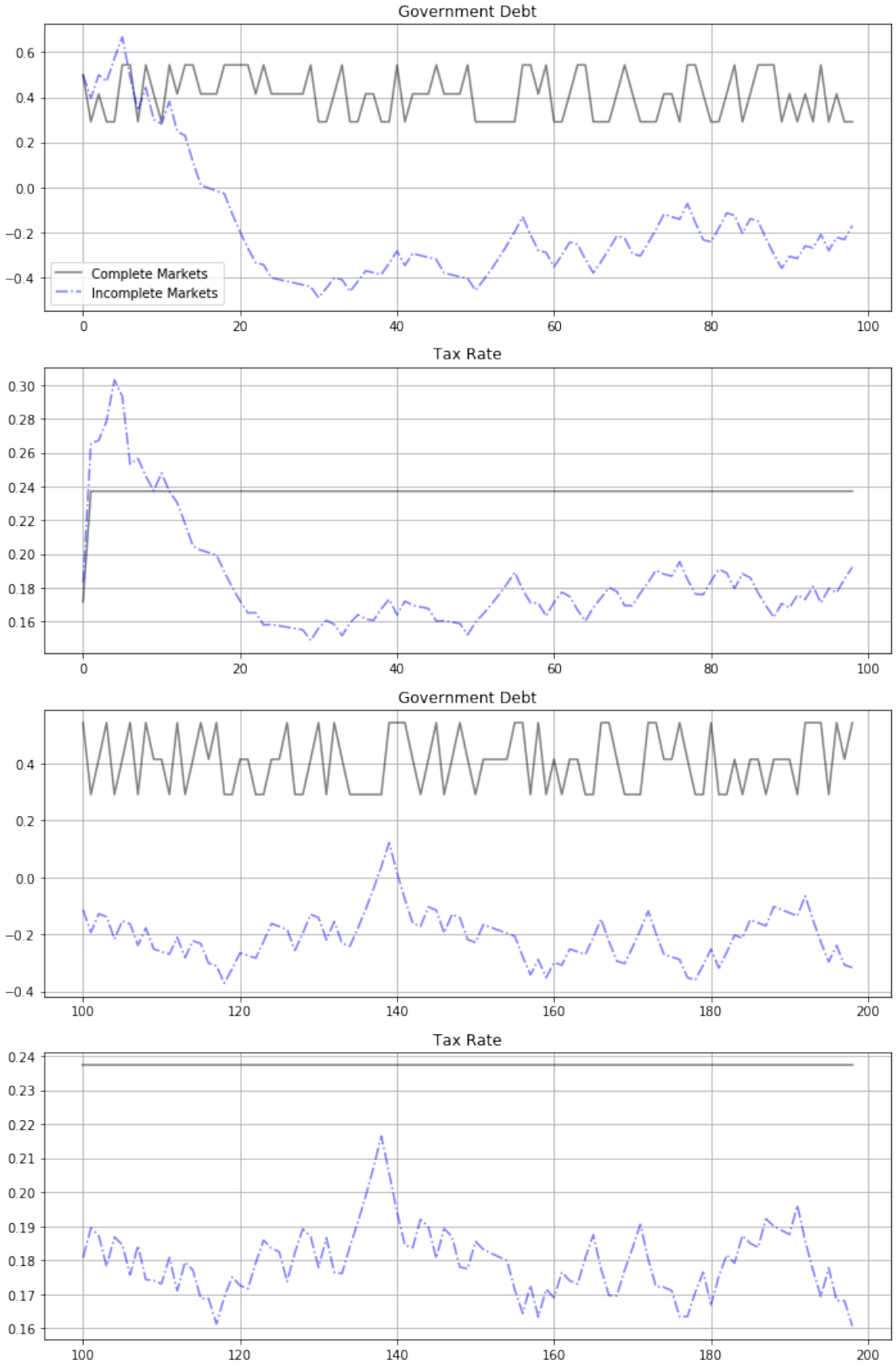

BEGS_approx
sample mean
20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
−1.20
−1.15
−1.10
−1.05
−1.00
−0.95
−0.90

(continued from previous page)

```
for i, id in enumerate([2, 3]):
    axes[i].plot(sim_seq_long[id][:99], '-k', sim_bel_long[id][:99],
                 '-.b', alpha=0.5)
    axes[i+2].plot(range(100, 199), sim_seq_long[id][100:199], '-k',
                   range(100, 199), sim_bel_long[id][100:199], '-.b',
                   alpha=0.5)
    axes[i].set(title=titles[i])
    axes[i+2].set(title=titles[i])
    axes[i].grid()
    axes[i+2].grid()

axes[0].legend(('Complete Markets', 'Incomplete Markets'))
plt.tight_layout()
plt.show()
```


Government Debt
Complete Markets
Incomplete Markets
Tax Rate
Government Debt
Tax Rate


Government Debt
Complete Markets
Incomplete Markets
Tax Rate
Government Debt
Tax Rate

For the short samples early in our simulated sample of 102,000 observations, fluctuations in government debt and the tax rate conceal the weak but inexorable force that the Ramsey planner puts into both series driving them toward ergodic marginal distributions that are far from these early observations

- early observations are more influenced by the initial value of the par value of government debt than by the ergodic mean of the par value of government debt
- much later observations are more influenced by the ergodic mean and are independent of the par value of initial government debt

48.4 Asymptotic Mean and Rate of Convergence

We apply the results of BEGS [Bhandari *et al.*, 2017] to interpret

- the mean of the ergodic distribution of government debt
- the rate of convergence to the ergodic distribution from an arbitrary initial government debt

We begin by computing objects required by the theory of section III.i of BEGS [Bhandari *et al.*, 2017].

As in *Fiscal Insurance via Fluctuating Interest Rates*, we recall that BEGS [Bhandari *et al.*, 2017] used a particular notation to represent what we can regard as their generalization of an AMSS model.

We introduce some of the [Bhandari *et al.*, 2017] notation so that readers can quickly relate notation that appears in key BEGS formulas to the notation that we have used in previous lectures *here* and *here*.

BEGS work with objects $B_t, \mathcal{B}_t, \mathcal{R}_t, \mathcal{X}_t$ that are related to notation that we used in earlier lectures by

$$
\begin{aligned}
\mathcal{R}_t &= \frac{u_{c,t}}{u_{c,t-1}} R_{t-1} = \frac{u_{c,t}}{\beta E_{t-1} u_{c,t}} \\
B_t &= \frac{b_{t+1}(s^t)}{R_t(s^t)} \\
b_t(s^{t-1}) &= \mathcal{R}_{t-1} B_{t-1} \\
\mathcal{B}_t &= u_{c,t} B_t = (\beta E_t u_{c,t+1}) b_{t+1}(s^t) \\
\mathcal{X}_t &= u_{c,t}[g_t - \tau_t n_t]
\end{aligned}
$$

BEGS [Bhandari *et al.*, 2017] call $\mathcal{X}_t$ the **effective** government deficit and $\mathcal{B}_t$ the **effective** government debt.

Equation (44) of [Bhandari *et al.*, 2017] expresses the time t state s government budget constraint as

$$
\mathcal{B}(s) = \mathcal{R}_\tau(s, s_-)\mathcal{B}_- + \mathcal{X}_\tau(s) \tag{48.1}
$$

where the dependence on τ is meant to remind us that these objects depend on the tax rate; s_- is last period's Markov state.

BEGS interpret random variations in the right side of (48.1) as **fiscal risks** generated by

- interest-rate-driven fluctuations in time t effective payments due on the government portfolio, namely, $\mathcal{R}_\tau(s, s_-)\mathcal{B}_-$, and
- fluctuations in the effective government deficit $\mathcal{X}_t$

48.4.1 Asymptotic Mean

BEGS give conditions under which the ergodic mean of $\mathcal{B}_t$ is approximated by

$$\mathcal{B}^* = -\frac{\text{cov}^\infty(\mathcal{R}_t, \mathcal{X}_t)}{\text{var}^\infty(\mathcal{R}_t)} \tag{48.2}$$

where the superscript ∞ denotes a moment taken with respect to an ergodic distribution.

Formula (48.2) represents $\mathcal{B}^*$ as a regression coefficient of $\mathcal{X}_t$ on $\mathcal{R}_t$ in the ergodic distribution.

Regression coefficient $\mathcal{B}^*$ solves a variance-minimization problem:

$$\mathcal{B}^* = \text{argmin}_{\mathcal{B}} \text{var}^\infty(\mathcal{R}\mathcal{B} + \mathcal{X}) \tag{48.3}$$

The minimand in criterion (48.3) measures **fiscal risk** associated with a given tax-debt policy that appears on the right side of equation (48.1).

Expressing formula (48.2) in terms of our notation tells us that the ergodic mean of the par value b of government debt in the AMSS model should be approximately

$$\hat{b} = \frac{\mathcal{B}^*}{\beta E(E_t u_{c,t+1})} = \frac{\mathcal{B}^*}{\beta E(u_{c,t+1})} \tag{48.4}$$

where mathematical expectations are taken with respect to the ergodic distribution.

48.4.2 Rate of Convergence

BEGS also derive the following approximation to the rate of convergence to $\mathcal{B}^*$ from an arbitrary initial condition.

$$\frac{E_t(\mathcal{B}_{t+1} - \mathcal{B}^*)}{(\mathcal{B}_t - \mathcal{B}^*)} \approx \frac{1}{1 + \beta^2 \text{var}^\infty(\mathcal{R})} \tag{48.5}$$

(See the equation above equation (47) in BEGS [Bhandari *et al.*, 2017])

48.4.3 More Advanced Topic

The remainder of this lecture is about technical material based on formulas from BEGS [Bhandari *et al.*, 2017].

The topic involves interpreting and extending formula (48.3) for the ergodic mean $\mathcal{B}^*$.

48.4.4 Chicken and Egg

Notice how attributes of the ergodic distribution for $\mathcal{B}_t$ appear on the right side of formula (48.3) for approximating the ergodic mean via $\mathcal{B}^*$.

Therefor, formula (48.3) is not useful for estimating the mean of the ergodic **in advance** of actually approximating the ergodic distribution.

- we need to know the ergodic distribution to compute the right side of formula (48.3)

So the primary use of equation (48.3) is how it **confirms** that the ergodic distribution solves a **fiscal-risk minimization problem**.

As an example, notice how we used the formula for the mean of $\mathcal{B}$ in the ergodic distribution of the special AMSS economy in *Fiscal Insurance via Fluctuating Interest Rates*

- **first** we computed the ergodic distribution using a reverse-engineering construction
- **then** we verified that $\mathcal{B}^*$ agrees with the mean of that distribution

48.4.5 Approximating the Ergodic Mean

BEGS also [Bhandari *et al.*, 2017] propose an approximation to $\mathcal{B}^*$ that can be computed **without** first approximating the ergodic distribution.

To construct the BEGS approximation to $\mathcal{B}^*$, we just follow steps set forth on pages 648 - 650 of section III.D of [Bhandari *et al.*, 2017]

- notation in BEGS might be confusing at first sight, so it is important to stare and digest before computing
- there are also some sign errors in the [Bhandari *et al.*, 2017] text that we'll want to correct here

Here is a step-by-step description of the BEGS [Bhandari *et al.*, 2017] approximation procedure.

48.4.6 Step by Step

Step 1: For a given τ we compute a vector of values $c_\tau(s), s = 1, 2, \ldots, S$ that satisfy

$$(1-\tau)c_\tau(s)^{-\sigma} - (c_\tau(s) + g(s))^{\gamma} = 0$$

This is a nonlinear equation to be solved for $c_\tau(s), s = 1, \ldots, S$.

$S = 3$ in our case, but we'll write code for a general integer S.

Typo alert: Please note that there is a sign error in equation (42) of BEGS [Bhandari *et al.*, 2017] – it should be a **minus** rather than a **plus** in the middle.

- We have made the appropriate correction in the above equation.

Step 2: Knowing $c_\tau(s), s = 1, \ldots, S$ for a given τ, we want to compute the random variables

$$\mathcal{R}_\tau(s) = \frac{c_\tau(s)^{-\sigma}}{\beta \sum_{s'=1}^{S} c_\tau(s')^{-\sigma}\pi(s')}$$

and

$$\mathcal{X}_\tau(s) = (c_\tau(s) + g(s))^{1+\gamma} - c_\tau(s)^{1-\sigma}$$

each for $s = 1, \ldots, S$.

BEGS call $\mathcal{R}_\tau(s)$ the **effective return** on risk-free debt and they call $\mathcal{X}_\tau(s)$ the **effective government deficit**.

Step 3: With the preceding objects in hand, for a given $\mathcal{B}$, we seek a τ that satisfies

$$\mathcal{B} = -\frac{\beta}{1-\beta} E\mathcal{X}_\tau \equiv -\frac{\beta}{1-\beta} \sum_s \mathcal{X}_\tau(s)\pi(s)$$

This equation says that at a constant discount factor β, equivalent government debt $\mathcal{B}$ equals the present value of the mean effective government **surplus**.

Another typo alert: there is a sign error in equation (46) of BEGS [Bhandari *et al.*, 2017] –the left side should be multiplied by -1.

- We have made this correction in the above equation.

For a given $\mathcal{B}$, let a τ that solves the above equation be called $\tau(\mathcal{B})$.

We'll use a Python root solver to find a τ that solves this equation for a given $\mathcal{B}$.

We'll use this function to induce a function $\tau(\mathcal{B})$.

Step 4: With a Python program that computes $\tau(\mathcal{B})$ in hand, next we write a Python function to compute the random variable.

$$J(\mathcal{B})(s) = \mathcal{R}_{\tau(\mathcal{B})}(s)\mathcal{B} + \mathcal{X}_{\tau(\mathcal{B})}(s), \quad s = 1, \ldots, S$$

Step 5: Now that we have a way to compute the random variable $J(\mathcal{B})(s), s = 1, \ldots, S$, via a composition of Python functions, we can use the population variance function that we defined in the code above to construct a function var$(J(\mathcal{B}))$.

We put var$(J(\mathcal{B}))$ into a Python function minimizer and compute

$$\mathcal{B}^* = \text{argmin}_{\mathcal{B}}\text{var}(J(\mathcal{B}))$$

Step 6: Next we take the minimizer $\mathcal{B}^*$ and the Python functions for computing means and variances and compute

$$\text{rate} = \frac{1}{1 + \beta^2 \text{var}(\mathcal{R}_{\tau(\mathcal{B}^*)})}$$

Ultimate outputs of this string of calculations are two scalars

$$(\mathcal{B}^*, \text{rate})$$

Step 7: Compute the divisor

$$div = \beta E u_{c,t+1}$$

and then compute the mean of the par value of government debt in the AMSS model

$$\hat{b} = \frac{\mathcal{B}^*}{div}$$

In the two-Markov-state AMSS economy in *Fiscal Insurance via Fluctuating Interest Rates*, $E_t u_{c,t+1} = E u_{c,t+1}$ in the ergodic distribution.

We have confirmed that this formula very accurately describes a **constant** par value of government debt that

- supports full fiscal insurance via fluctuating interest parameters, and
- is the limit of government debt as $t \to +\infty$

In the three-Markov-state economy of this lecture, the par value of government debt fluctuates in a history-dependent way even asymptotically.

In this economy, $\hat{b}$ given by the above formula approximates the mean of the ergodic distribution of the par value of government debt

so while the approximation circumvents the chicken and egg problem that surrounds
the much better approximation associated with the green vertical line, it does so by enlarging the approximation error

- $\hat{b}$ is represented by the red vertical line plotted in the histogram of the last 100,000 observations of our simulation of the par value of government debt plotted above
- the approximation is fairly accurate but not perfect

48.4.7 Execution

Now let's move on to compute things step by step.

Step 1

```
u = CRRAutility(π=np.full((3, 3), 1 / 3),
                G=np.array([0.1, 0.2, .3]),
                Θ=np.ones(3))

τ = 0.05            # Initial guess of τ (to displays calcs along the way)
S = len(u.G)        # Number of states

def solve_c(c, τ, u):
    return (1 - τ) * c**(-u.σ) - (c + u.G)**u.γ

# .x returns the result from root
c = root(solve_c, np.ones(S), args=(τ, u)).x
c
```

```
array([0.93852387, 0.89231015, 0.84858872])
```

```
root(solve_c, np.ones(S), args=(τ, u))
```

```
 message: The solution converged.
 success: True
  status: 1
     fun: [ 5.618e-10 -4.769e-10  1.175e-11]
       x: [ 9.385e-01  8.923e-01  8.486e-01]
  method: hybr
    nfev: 11
    fjac: [[-9.999e-01 -4.954e-03 -1.261e-02]
           [-5.156e-03  9.999e-01  1.610e-02]
           [-1.253e-02 -1.616e-02  9.998e-01]]
       r: [ 4.269e+00  8.685e-02 -6.301e-02 -4.713e+00 -7.433e-02
            -5.508e+00]
     qtf: [ 1.556e-08  1.283e-08  7.899e-11]
```

Step 2

```
n = c + u.G    # Compute labor supply
```

48.4.8 Note about Code

Remember that in our code π is a 3×3 transition matrix.

But because we are studying an IID case, π has identical rows and we need only to compute objects for one row of π.

This explains why at some places below we set $s = 0$ just to pick off the first row of π.

48.4.9 Running the code

Let's take the code out for a spin.

First, let's compute $\mathcal{R}$ and $\mathcal{X}$ according to our formulas

```
def compute_R_X(τ, u, s):
    c = root(solve_c, np.ones(S), args=(τ, u)).x  # Solve for vector of c's
    div = u.β * (u.Uc(c[0], n[0]) * u.π[s, 0]  \
                 +  u.Uc(c[1], n[1]) * u.π[s, 1] \
                 +  u.Uc(c[2], n[2]) * u.π[s, 2])
    R = c**(-u.σ) / (div)
    X = (c + u.G)**(1 + u.γ) - c**(1 - u.σ)
    return R, X
```

```
c**(-u.σ) @ u.π
```

```
array([1.25997521, 1.25997521, 1.25997521])
```

```
u.π
```

```
array([[0.33333333, 0.33333333, 0.33333333],
       [0.33333333, 0.33333333, 0.33333333],
       [0.33333333, 0.33333333, 0.33333333]])
```

We only want unconditional expectations because we are in an IID case.

So we'll set $s = 0$ and just pick off expectations associated with the first row of π

```
s = 0

R, X = compute_R_X(τ, u, s)
```

Let's look at the random variables $\mathcal{R}, \mathcal{X}$

```
R
```

```
array([1.00116313, 1.10755123, 1.22461897])
```

```
mean(R, s)
```

```
1.1111111111111112
```

```
X
```

```
array([0.05457803, 0.18259396, 0.33685546])
```

```
mean(X, s)
```

```
0.19134248445303795
```

```
X @ u.π
```

```
array([0.19134248, 0.19134248, 0.19134248])
```

Step 3

```
def solve_τ(τ, B, u, s):
    R, X = compute_R_X(τ, u, s)
    return ((u.β - 1) / u.β) * B - X @ u.π[s]
```

Note that B is a scalar.

Let's try out our method computing τ

```
s = 0
B = 1.0

τ = root(solve_τ, .1, args=(B, u, s)).x[0]  # Very sensitive to initial value
τ
```

```
0.2740159773695818
```

In the above cell, B is fixed at 1 and τ is to be computed as a function of B.

Note that 0.2 is the initial value for τ in the root-finding algorithm.

Step 4

```
def min_J(B, u, s):
    # Very sensitive to initial value of τ
    τ = root(solve_τ, .5, args=(B, u, s)).x[0]
    R, X = compute_R_X(τ, u, s)
    return variance(R * B + X, s)
```

```
min_J(B, u, s)
```

```
0.035564405653720765
```

Step 6

```
B_star = minimize(min_J, .5, args=(u, s)).x[0]
B_star
```

```
-1.199483167941158
```

```
n = c + u.G  # Compute labor supply
```

```
div = u.β * (u.Uc(c[0], n[0]) * u.π[s, 0]  \
             +  u.Uc(c[1], n[1]) * u.π[s, 1] \
             +  u.Uc(c[2], n[2]) * u.π[s, 2])
```

```
B_hat = B_star/div
B_hat
```

```
-1.0577661126390971
```

```
τ_star = root(solve_τ, 0.05, args=(B_star, u, s)).x[0]
τ_star
```

```
0.09572916798461703
```

```
R_star, X_star = compute_R_X(τ_star, u, s)
R_star, X_star
```

```
(array([0.9998398 , 1.10746593, 1.2260276 ]),
 array([0.0020272 , 0.12464752, 0.27315299]))
```

```
rate = 1 / (1 + u.β**2 * variance(R_star, s))
rate
```

```
0.9931353432732218
```

```
root(solve_c, np.ones(S), args=(τ_star, u)).x
```

```
array([0.9264382 , 0.88027117, 0.83662635])
```

CHAPTER

FORTYNINE

COMPETITIVE EQUILIBRIA OF A MODEL OF CHANG

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install polytope cvxopt
```

49.1 Overview

This lecture describes how Chang [Chang, 1998] analyzed **competitive equilibria** and a best competitive equilibrium called a **Ramsey plan**.

He did this by

- characterizing a competitive equilibrium recursively in a way also employed in the *dynamic Stackelberg problems* and *Calvo model* lectures to pose Stackelberg problems in linear economies, and then
- appropriately adapting an argument of Abreu, Pearce, and Stachetti [Abreu *et al.*, 1990] to describe key features of the set of competitive equilibria

Roberto Chang [Chang, 1998] chose a model of Calvo [Calvo, 1978] as a simple structure that conveys ideas that apply more broadly.

A textbook version of Chang's model appears in chapter 25 of [Ljungqvist and Sargent, 2018].

This lecture and *Credible Government Policies in Chang Model* can be viewed as more sophisticated and complete treatments of the topics discussed in *Ramsey plans, time inconsistency, sustainable plans*.

Both this lecture and *Credible Government Policies in Chang Model* make extensive use of an idea to which we apply the nickname **dynamic programming squared**.

In dynamic programming squared problems there are typically two interrelated Bellman equations

- A Bellman equation for a set of agents or followers with value or value function v_a.
- A Bellman equation for a principal or Ramsey planner or Stackelberg leader with value or value function v_p in which v_a appears as an argument.

We encountered problems with this structure in *dynamic Stackelberg problems*, *optimal taxation with state-contingent debt*, and other lectures.

We'll start with some standard imports:

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

49.1.1 The Setting

First, we introduce some notation.

For a sequence of scalars $\vec{z} \equiv \{z_t\}_{t=0}^{\infty}$, let $\vec{z}^t = (z_0, \dots, z_t)$, $\vec{z}_t = (z_t, z_{t+1}, \dots)$.

An infinitely lived representative agent and an infinitely lived government exist at dates $t = 0, 1, \dots$.

The objects in play are

- an initial quantity M_{-1} of nominal money holdings
- a sequence of inverse money growth rates $\vec{h}$ and an associated sequence of nominal money holdings $\vec{M}$
- a sequence of values of money $\vec{q}$
- a sequence of real money holdings $\vec{m}$
- a sequence of total tax collections $\vec{x}$
- a sequence of per capita rates of consumption $\vec{c}$
- a sequence of per capita incomes $\vec{y}$

A benevolent government chooses sequences $(\vec{M}, \vec{h}, \vec{x})$ subject to a sequence of budget constraints and other constraints imposed by competitive equilibrium.

Given tax collection and price of money sequences, a representative household chooses sequences $(\vec{c}, \vec{m})$ of consumption and real balances.

In competitive equilibrium, the price of money sequence $\vec{q}$ clears markets, thereby reconciling decisions of the government and the representative household.

Chang adopts a version of a model that [Calvo, 1978] designed to exhibit time-inconsistency of a Ramsey policy in a simple and transparent setting.

By influencing the representative household's expectations, government actions at time t affect components of household utilities for periods s before t.

When setting a path for monetary expansion rates, the government takes into account how the household's anticipations of the government's future actions affect the household's current decisions.

The ultimate source of time inconsistency is that a time 0 Ramsey planner takes these effects into account in designing a plan of government actions for $t \geq 0$.

49.2 Decisions

49.2.1 The Household's Problem

A representative household faces a nonnegative value of money sequence $\vec{q}$ and sequences $\vec{y}, \vec{x}$ of income and total tax collections, respectively.

Facing vector $\vec{q}$ as a price taker, the representative household chooses nonnegative sequences $\vec{c}, \vec{M}$ of consumption and nominal balances, respectively, to maximize

$$\sum_{t=0}^{\infty} \beta^t \left[u(c_t) + v(q_t M_t)\right] \tag{49.1}$$

subject to

$$q_t M_t \leq y_t + q_t M_{t-1} - c_t - x_t \tag{49.2}$$

and

$$q_t M_t \leq \bar{m} \tag{49.3}$$

Here q_t is the reciprocal of the price level at t, which we can also call the *value of money*.

Chang [Chang, 1998] assumes that

- $u : \mathbb{R}_+ \to \mathbb{R}$ is twice continuously differentiable, strictly concave, and strictly increasing;
- $v : \mathbb{R}_+ \to \mathbb{R}$ is twice continuously differentiable and strictly concave;
- $u'(c)_{c \to 0} = \lim_{m \to 0} v'(m) = +\infty$;
- there is a finite level $m = m^f$ such that $v'(m^f) = 0$

The household carries real balances out of a period equal to $m_t = q_t M_t$.

Inequality (49.2) is the household's time t budget constraint.

It tells how real balances $q_t M_t$ carried out of period t depend on real balances $q_t M_{t-1}$ carried into period t, income, consumption, taxes.

Equation (49.3) imposes an exogenous upper bound $\bar{m}$ on the household's choice of real balances, where $\bar{m} \geq m^f$.

49.2.2 Government

The government chooses a sequence of inverse money growth rates with time t component $h_t \equiv \frac{M_{t-1}}{M_t} \in \Pi \equiv [\underline{\pi}, \overline{\pi}]$, where $0 < \underline{\pi} < 1 < \frac{1}{\beta} \leq \overline{\pi}$.

The government purchases no goods.

It taxes only to acquire paper currency that it will withdraw from circulation (e.g., by burning it).

Let p_t be the price level at time t, measured as time t dollars per unit of the consumption good.

Evidently, the value of paper currency meassured in units of the consumption good at time t is

$$q_t = \frac{1}{p_t}.$$

The government faces a sequence of budget constraints with time t component

$$x_t + \frac{M_t - M_{t-1}}{p_t} = 0,$$

where x_t is the real value of revenue that the government raises from taxes and $\frac{M_t - M_{t-1}}{p_t}$ is the real value of revenue that the government raises by printing new paper currency.

Evidently, this budget constraint can be rewritten as

$$-x_t = q_t(M_t - M_{t-1})$$

which by using the definitions of m_t and h_t can also be expressed as

$$-x_t = m_t(1 - h_t) \tag{49.4}$$

The restrictions $m_t \in [0, \bar{m}]$ and $h_t \in \Pi = [\underline{\pi}, \overline{\pi}]$ evidently imply that $x_t \in X \equiv [(\underline{\pi} - 1)\bar{m}, (\overline{\pi} - 1)\bar{m}]$.

We define the set $E \equiv [0, \bar{m}] \times \Pi \times X$, so that we require that $(m, h, x) \in E$.

To represent the idea that taxes are distorting, Chang makes the following assumption about outcomes for per capita output:

$$y_t = f(x_t), \tag{49.5}$$

where $f : \mathbb{R} \to \mathbb{R}$ satisfies $f(x) > 0$, $f(x)$ is twice continuously differentiable, $f''(x) < 0$, $f'(0) = 0$, and $f(x) = f(-x)$ for all $x \in \mathbb{R}$, so that subsidies and taxes are equally distorting.

Example parameterizations

In some of our Python code deployed later in this lecture, we'll assume the following functional forms:

$$u(c) = \log(c)$$

$$v(m) = \frac{1}{500}(m\bar{m} - 0.5m^2)^{0.5}$$

$$f(x) = 180 - (0.4x)^2$$

The tax distortion function

Calvo's and Chang's purpose is not to model the causes of tax distortions in any detail but simply to summarize the *outcome* of those distortions via the function $f(x)$.

A key part of the specification is that tax distortions are increasing in the absolute value of tax revenues.

Ramsey plan: A Ramsey plan is a competitive equilibrium that maximizes (49.1).

Within-period timing of decisions is as follows:

- first, the government chooses h_t and x_t;
- then given $\vec{q}$ and its expectations about future values of x and y's, the household chooses M_t and therefore m_t because $m_t = q_t M_t$;
- then output $y_t = f(x_t)$ is realized;
- finally $c_t = y_t$

This within-period timing confronts the government with choices framed by how the private sector wants to respond when the government takes time t actions that differ from what the private sector had expected.

This consideration will be important in lecture *credible government policies* when we study *credible government policies*.

The model is designed to focus on the intertemporal trade-offs between the welfare benefits of deflation and the welfare costs associated with the high tax collections required to retire money at a rate that delivers deflation.

A benevolent time 0 government can promote utility generating increases in real balances only by imposing sufficiently large distorting tax collections.

To promote the welfare increasing effects of high real balances, the government wants to induce *gradual deflation*.

49.2.3 Household's Problem

Given M_{-1} and $\{q_t\}_{t=0}^{\infty}$, the household's problem is

$$\begin{aligned}\mathcal{L} = \max_{\vec{c}, \vec{M}} \min_{\vec{\lambda}, \vec{\mu}} \sum_{t=0}^{\infty} \beta^t \{u(c_t) + v(M_t q_t) + \lambda_t [y_t - c_t - x_t + q_t M_{t-1} - q_t M_t] \\ + \mu_t [\bar{m} - q_t M_t]\}\end{aligned}$$

First-order conditions with respect to c_t and M_t, respectively, are

$$u'(c_t) = \lambda_t$$
$$q_t[u'(c_t) - v'(M_t q_t)] \leq \beta u'(c_{t+1}) q_{t+1}, \quad = \text{ if } M_t q_t < \bar{m}$$

The last equation expresses Karush-Kuhn-Tucker complementary slackness conditions (see here).

These insist that the inequality is an equality at an interior solution for M_t.

Using $h_t = \frac{M_{t-1}}{M_t}$ and $q_t = \frac{m_t}{M_t}$ in these first-order conditions and rearranging implies

$$m_t[u'(c_t) - v'(m_t)] \leq \beta u'(f(x_{t+1})) m_{t+1} h_{t+1}, \quad = \text{ if } m_t < \bar{m} \tag{49.6}$$

Define the following key variable

$$\theta_{t+1} \equiv u'(f(x_{t+1})) m_{t+1} h_{t+1} \tag{49.7}$$

This is real money balances at time $t+1$ measured in units of marginal utility, which Chang refers to as 'the marginal utility of real balances'.

From the standpoint of the household at time t, equation (49.7) shows that θ_{t+1} intermediates the influences of $(\vec{x}_{t+1}, \vec{m}_{t+1})$ on the household's choice of real balances m_t.

By "intermediates" we mean that the future paths $(\vec{x}_{t+1}, \vec{m}_{t+1})$ influence m_t entirely through their effects on the scalar θ_{t+1}.

The observation that the one dimensional promised marginal utility of real balances θ_{t+1} functions in this way is an important step in constructing a class of competitive equilibria that have a recursive representation.

A closely related observation pervaded the analysis of Stackelberg plans in lecture *dynamic Stackelberg problems*.

49.3 Competitive Equilibrium

Definition:

- A *government policy* is a pair of sequences $(\vec{h}, \vec{x})$ where $h_t \in \Pi \; \forall t \geq 0$.
- A *price system* is a nonnegative value of money sequence $\vec{q}$.
- An *allocation* is a triple of nonnegative sequences $(\vec{c}, \vec{m}, \vec{y})$.

It is required that time t components $(m_t, x_t, h_t) \in E$.

Definition:

Given M_{-1}, a government policy $(\vec{h}, \vec{x})$, price system $\vec{q}$, and allocation $(\vec{c}, \vec{m}, \vec{y})$ are said to be a *competitive equilibrium* if

- $m_t = q_t M_t$ and $y_t = f(x_t)$.
- The government budget constraint is satisfied.
- Given $\vec{q}, \vec{x}, \vec{y}$, $(\vec{c}, \vec{m})$ solves the household's problem.

49.4 Inventory of Objects in Play

Chang constructs the following objects

1. A set Ω of initial marginal utilities of money θ_0
   - Let Ω denote the set of initial promised marginal utilities of money θ_0 associated with competitive equilibria.
   - Chang exploits the fact that a competitive equilibrium consists of a first period outcome (h_0, m_0, x_0) and a continuation competitive equilibrium with marginal utility of money $\theta_1 \in \Omega$.
2. Competitive equilibria that have a recursive representation
   - A competitive equilibrium with a recursive representation consists of an initial θ_0 and a four-tuple of functions (h, m, x, Ψ) mapping θ into this period's (h, m, x) and next period's θ, respectively.
   - A competitive equilibrium can be represented recursively by iterating on

$$\begin{aligned} h_t &= h(\theta_t) \\ m_t &= m(\theta_t) \\ x_t &= x(\theta_t) \\ \theta_{t+1} &= \Psi(\theta_t) \end{aligned} \tag{49.8}$$

     starting from θ_0

     The range and domain of $\Psi(\cdot)$ are both Ω
3. A recursive representation of a Ramsey plan
   - A recursive representation of a Ramsey plan is a recursive competitive equilibrium $\theta_0, (h, m, x, \Psi)$ that, among all recursive competitive equilibria, maximizes $\sum_{t=0}^{\infty} \beta^t \left[u(c_t) + v(q_t M_t)\right]$.
   - The Ramsey planner chooses $\theta_0, (h, m, x, \Psi)$ from among the set of recursive competitive equilibria at time 0.
   - Iterations on the function Ψ determine subsequent θ_t's that summarize the aspects of the continuation competitive equilibria that influence the household's decisions.
   - At time 0, the Ramsey planner commits to this implied sequence $\{\theta_t\}_{t=0}^{\infty}$ and therefore to an associated sequence of continuation competitive equilibria.
4. A characterization of time-inconsistency of a Ramsey plan
   - Imagine that after a 'revolution' at time $t \geq 1$, a new Ramsey planner is given the opportunity to ignore history and solve a brand new Ramsey plan.
   - This new planner would want to reset the θ_t associated with the original Ramsey plan to θ_0.
   - The incentive to reinitialize θ_t associated with this revolution experiment indicates the time-inconsistency of the Ramsey plan.
   - By resetting θ to θ_0, the new planner avoids the costs at time t that the original Ramsey planner must pay to reap the beneficial effects that the original Ramsey plan for $s \geq t$ had achieved via its influence on the household's decisions for $s = 0, \ldots, t-1$.

49.5 Analysis

A competitive equilibrium is a triple of sequences $(\vec{m}, \vec{x}, \vec{h}) \in E^\infty$ that satisfies (49.2), (49.3), and (49.6).

Chang works with a set of competitive equilibria defined as follows.

Definition: $CE = \{(\vec{m}, \vec{x}, \vec{h}) \in E^\infty$ such that (49.2), (49.3), and (49.6) are satisfied $\}$.

CE is not empty because there exists a competitive equilibrium with $h_t = 1$ for all $t \geq 1$, namely, an equilibrium with a constant money supply and constant price level.

Chang establishes that CE is also compact.

Chang makes the following key observation that combines ideas of Abreu, Pearce, and Stacchetti [Abreu *et al.*, 1990] with insights of Kydland and Prescott [Kydland and Prescott, 1980].

Proposition: The continuation of a competitive equilibrium is a competitive equilibrium.

That is, $(\vec{m}, \vec{x}, \vec{h}) \in CE$ implies that $(\vec{m}_t, \vec{x}_t, \vec{h}_t) \in CE \; \forall \; t \geq 1$.

(Lecture *dynamic Stackelberg problems* also used a version of this insight)

We can now state that a **Ramsey problem** is to

$$\max_{(\vec{m}, \vec{x}, \vec{h}) \in E^\infty} \sum_{t=0}^{\infty} \beta^t \left[u(c_t) + v(m_t) \right]$$

subject to restrictions (49.2), (49.3), and (49.6).

Evidently, associated with any competitive equilibrium (m_0, x_0) is an implied value of $\theta_0 = u'(f(x_0))(m_0 + x_0)$.

To bring out a recursive structure inherent in the Ramsey problem, Chang defines the set

$$\Omega = \left\{ \theta \in \mathbb{R} \text{ such that } \theta = u'(f(x_0))(m_0 + x_0) \text{ for some } (\vec{m}, \vec{x}, \vec{h}) \in CE \right\}$$

Equation (49.6) inherits from the household's Euler equation for money holdings the property that the value of m_0 consistent with the representative household's choices depends on $(\vec{h}_1, \vec{m}_1)$.

This dependence is captured in the definition above by making Ω be the set of first period values of θ_0 satisfying $\theta_0 = u'(f(x_0))(m_0 + x_0)$ for first period component (m_0, h_0) of competitive equilibrium sequences $(\vec{m}, \vec{x}, \vec{h})$.

Chang establishes that Ω is a nonempty and compact subset of $\mathbb{R}_+$.

Next Chang advances:

Definition: $\Gamma(\theta) = \{(\vec{m}, \vec{x}, \vec{h}) \in CE | \theta = u'(f(x_0))(m_0 + x_0)\}$.

Thus, $\Gamma(\theta)$ is the set of competitive equilibrium sequences $(\vec{m}, \vec{x}, \vec{h})$ whose first period components (m_0, h_0) deliver the prescribed value θ for first period marginal utility.

If we knew the sets $\Omega, \Gamma(\theta)$, we could use the following two-step procedure to find at least the *value* of the Ramsey outcome to the representative household

1. Find the indirect value function $w(\theta)$ defined as

$$w(\theta) = \max_{(\vec{m}, \vec{x}, \vec{h}) \in \Gamma(\theta)} \sum_{t=0}^{\infty} \beta^t \left[u(f(x_t)) + v(m_t) \right]$$

2. Compute the value of the Ramsey outcome by solving $\max_{\theta \in \Omega} w(\theta)$.

Thus, Chang states the following

Proposition:

$w(\theta)$ satisfies the Bellman equation

$$w(\theta) = \max_{x,m,h,\theta'} \{u(f(x)) + v(m) + \beta w(\theta')\} \tag{49.9}$$

where maximization is subject to

$$(m, x, h) \in E \text{ and } \theta' \in \Omega \tag{49.10}$$

and

$$\theta = u'(f(x))(m + x) \tag{49.11}$$

and

$$-x = m(1 - h) \tag{49.12}$$

and

$$m \cdot [u'(f(x)) - v'(m)] \leq \beta\theta', \quad = \text{ if } m < \bar{m} \tag{49.13}$$

Before we use this proposition to recover a recursive representation of the Ramsey plan, note that the proposition relies on knowing the set Ω.

To find Ω, Chang uses the insights of Kydland and Prescott [Kydland and Prescott, 1980] together with a method based on the Abreu, Pearce, and Stacchetti [Abreu *et al.*, 1990] iteration to convergence on an operator B that maps continuation values into values.

We want an operator that maps a continuation θ into a current θ.

Chang lets Q be a nonempty, bounded subset of $\mathbb{R}$.

Elements of the set Q are taken to be candidate values for continuation marginal utilities.

Chang defines an operator

$$B(Q) = \theta \in \mathbb{R} \text{ such that there is } (m, x, h, \theta') \in E \times Q$$

such that (49.11), (49.12), and (49.13) hold.

Thus, $B(Q)$ is the set of first period θ's attainable with $(m, x, h) \in E$ and some $\theta' \in Q$.

Proposition:

1. $Q \subset B(Q)$ implies $B(Q) \subset \Omega$ ('self-generation').
2. $\Omega = B(\Omega)$ ('factorization').

The proposition characterizes Ω as the largest fixed point of B.

It is easy to establish that $B(Q)$ is a monotone operator.

This property allows Chang to compute Ω as the limit of iterations on B provided that iterations begin from a sufficiently large initial set.

49.5.1 Some Useful Notation

Let $\vec{h}^t = (h_0, h_1, \ldots, h_t)$ denote a history of inverse money creation rates with time t component $h_t \in \Pi$.

A *government strategy* $\sigma = \{\sigma_t\}_{t=0}^{\infty}$ is a $\sigma_0 \in \Pi$ and for $t \geq 1$ a sequence of functions $\sigma_t : \Pi^{t-1} \to \Pi$.

Chang restricts the government's choice of strategies to the following space:

$$CE_\pi = \{\vec{h} \in \Pi^\infty : \text{ there is some } (\vec{m}, \vec{x}) \text{ such that } (\vec{m}, \vec{x}, \vec{h}) \in CE\}$$

In words, CE_π is the set of money growth sequences consistent with the existence of competitive equilibria.

Chang observes that CE_π is nonempty and compact.

Definition: σ is said to be *admissible* if for all $t \geq 1$ and after any history $\vec{h}^{t-1}$, the continuation $\vec{h}_t$ implied by σ belongs to CE_π.

Admissibility of σ means that anticipated policy choices associated with σ are consistent with the existence of competitive equilibria after each possible subsequent history.

After any history $\vec{h}^{t-1}$, admissibility restricts the government's choice in period t to the set

$$CE_\pi^0 = \{h \in \Pi : \text{there is } \vec{h} \in CE_\pi \text{ with } h = h_0\}$$

In words, CE_π^0 is the set of all first period money growth rates $h = h_0$, each of which is consistent with the existence of a sequence of money growth rates $\vec{h}$ starting from h_0 in the initial period and for which a competitive equilibrium exists.

Remark:

$CE_\pi^0 = \{h \in \Pi : \text{ there is } (m, \theta') \in [0, \bar{m}] \times \Omega \text{ such that } u'[f((h-1)m) - v'(m)] \leq \beta\theta' \text{ with equality if } m < \bar{m}\}$.

Definition: An *allocation rule* is a sequence of functions $\vec{\alpha} = \{\alpha_t\}_{t=0}^{\infty}$ such that $\alpha_t : \Pi^t \to [0, \bar{m}] \times X$.

Thus, the time t component of $\alpha_t(h^t)$ is a pair of functions $(m_t(h^t), x_t(h^t))$.

Definition: Given an admissible government strategy σ, an allocation rule α is called *competitive* if given any history $\vec{h}^{t-1}$ and $h_t \in CE_\pi^0$, the continuations of σ and α after $(\vec{h}^{t-1}, h_t)$ induce a competitive equilibrium sequence.

49.5.2 Another Operator

At this point it is convenient to introduce another operator that can be used to compute a Ramsey plan.

For computing a Ramsey plan, this operator is wasteful because it works with a state vector that is bigger than necessary.

We introduce this operator because it helps to prepare the way for Chang's operator called $\tilde{D}(Z)$ that we shall describe in lecture *credible government policies*.

It is also useful because a fixed point of the operator to be defined here provides a good guess for an initial set from which to initiate iterations on Chang's set-to-set operator $\tilde{D}(Z)$ to be described in lecture *credible government policies*.

Let S be the set of all pairs (w, θ) of competitive equilibrium values and associated initial marginal utilities.

Let W be a bounded set of *values* in $\mathbb{R}$.

Let Z be a nonempty subset of $W \times \Omega$.

Think of using pairs (w', θ') drawn from Z as candidate continuation value, θ pairs.

Define the operator

$$D(Z) = \Big\{(w, \theta) : \text{there is } h \in CE_\pi^0$$

$$\text{and a four-tuple } (m(h), x(h), w'(h), \theta'(h)) \in [0, \bar{m}] \times X \times Z$$

such that

$$w = u(f(x(h))) + v(m(h)) + \beta w'(h) \tag{49.14}$$

$$\theta = u'(f(x(h)))(m(h) + x(h)) \tag{49.15}$$

$$x(h) = m(h)(h-1) \tag{49.16}$$

$$m(h)(u'(f(x(h))) - v'(m(h))) \leq \beta\theta'(h) \tag{49.17}$$

$$\text{with equality if } m(h) < \bar{m}\Big\}$$

It is possible to establish.

Proposition:

1. If $Z \subset D(Z)$, then $D(Z) \subset S$ ('self-generation').
2. $S = D(S)$ ('factorization').

Proposition:

1. Monotonicity of D: $Z \subset Z'$ implies $D(Z) \subset D(Z')$.
2. Z compact implies that $D(Z)$ is compact.

It can be shown that S is compact and that therefore there exists a (w, θ) pair within this set that attains the highest possible value w.

This (w, θ) pair i associated with a Ramsey plan.

Further, we can compute S by iterating to convergence on D provided that one begins with a sufficiently large initial set S_0.

As a very useful by-product, the algorithm that finds the largest fixed point $S = D(S)$ also produces the Ramsey plan, its value w, and the associated competitive equilibrium.

49.6 Calculating all Promise-Value Pairs in CE

Above we have defined the $D(Z)$ operator as:

$$D(Z) = \{(w, \theta) : \exists h \in CE^0_\pi \text{ and } (m(h), x(h), w'(h), \theta'(h)) \in [0, \bar{m}] \times X \times Z$$

such that

$$w = u(f(x(h))) + v(m(h)) + \beta w'(h)$$

$$\theta = u'(f(x(h)))(m(h) + x(h))$$

$$x(h) = m(h)(h-1)$$

$$m(h)(u'(f(x(h))) - v'(m(h))) \leq \beta\theta'(h) \text{ (with equality if } m(h) < \bar{m})\}$$

We noted that the set S can be found by iterating to convergence on D, provided that we start with a sufficiently large initial set S_0.

Our implementation builds on ideas in this notebook.

To find S we use a numerical algorithm called the *outer hyperplane approximation algorithm*.

It was invented by Judd, Yeltekin, Conklin [Judd *et al.*, 2003].

This algorithm constructs the smallest convex set that contains the fixed point of the $D(S)$ operator.

Given that we are finding the smallest convex set that contains S, we can represent it on a computer as the intersection of a finite number of half-spaces.

Let H be a set of subgradients, and C be a set of hyperplane levels.

We approximate S by:

$$\tilde{S} = \{(w, \theta) | H \cdot (w, \theta) \leq C\}$$

A key feature of this algorithm is that we discretize the action space, i.e., we create a grid of possible values for m and h (note that x is implied by m and h). This discretization simplifies computation of $\tilde{S}$ by allowing us to find it by solving a sequence of linear programs.

The *outer hyperplane approximation algorithm* proceeds as follows:

1. Initialize subgradients, H, and hyperplane levels, C_0.
2. Given a set of subgradients, H, and hyperplane levels, C_t, for each subgradient $h_i \in H$:
    - Solve a linear program (described below) for each action in the action space.
    - Find the maximum and update the corresponding hyperplane level, $C_{i,t+1}$.
3. If $|C_{t+1} - C_t| > \epsilon$, return to 2.

Step 1 simply creates a large initial set S_0.

Given some set S_t, **Step 2** then constructs the set $S_{t+1} = D(S_t)$. The linear program in Step 2 is designed to construct a set S_{t+1} that is as large as possible while satisfying the constraints of the $D(S)$ operator.

To do this, for each subgradient h_i, and for each point in the action space (m_j, h_j), we solve the following problem:

$$\max_{[w', \theta']} h_i \cdot (w, \theta)$$

subject to

$$H \cdot (w', \theta') \leq C_t$$

$$w = u(f(x_j)) + v(m_j) + \beta w'$$

$$\theta = u'(f(x_j))(m_j + x_j)$$

$$x_j = m_j(h_j - 1)$$

$$m_j(u'(f(x_j)) - v'(m_j)) \leq \beta\theta' \quad (= \text{if } m_j < \bar{m})$$

This problem maximizes the hyperplane level for a given set of actions.

The second part of Step 2 then finds the maximum possible hyperplane level across the action space.

The algorithm constructs a sequence of progressively smaller sets $S_{t+1} \subset S_t \subset S_{t-1} \cdots \subset S_0$.

Step 3 ends the algorithm when the difference between these sets is small enough.

We have created a Python class that solves the model assuming the following functional forms:

$$u(c) = \log(c)$$

$$v(m) = \frac{1}{500}(m\bar{m} - 0.5m^2)^{0.5}$$

$$f(x) = 180 - (0.4x)^2$$

The remaining parameters $\{\beta, \bar{m}, \underline{h}, \bar{h}\}$ are then variables to be specified for an instance of the Chang class.

Below we use the class to solve the model and plot the resulting equilibrium set, once with $\beta = 0.3$ and once with $\beta = 0.8$.

(Here we have set the number of subgradients to 10 in order to speed up the code for now - we can increase accuracy by increasing the number of subgradients)

```
"""
Provides a class called ChangModel to solve different
parameterizations of the Chang (1998) model.
"""

import numpy as np
import time

from scipy.optimize import linprog, minimize
import numpy.polynomial.chebyshev as cheb


class ChangModel:
    """
    Class to solve for the competitive and sustainable sets in the Chang (1998)
    model, for different parameterizations.
    """

    def __init__(self, β, mbar, h_min, h_max, n_h, n_m, N_g):
        # Record parameters
        self.β, self.mbar, self.h_min, self.h_max = β, mbar, h_min, h_max
        self.n_h, self.n_m, self.N_g = n_h, n_m, N_g

        # Create other parameters
        self.m_min = 1e-9
        self.m_max = self.mbar
        self.N_a = self.n_h*self.n_m

        # Utility and production functions
        uc = lambda c: np.log(np.maximum(c, 1e-10))  # Clip to 1e-10 to avoid log(0)␣
↪or log(-ve)
        uc_p = lambda c: 1/c
        v = lambda m: 1/500 * (mbar * m - 0.5 * m**2)**0.5
        v_p = lambda m: 0.5/500 * (mbar * m - 0.5 * m**2)**(-0.5) * (mbar - m)
        u = lambda h, m: uc(f(h, m)) + v(m)

        def f(h, m):
            x = m * (h - 1)
            f = 180 - (0.4 * x)**2
            return f

        def θ(h, m):
            x = m * (h - 1)
            θ = uc_p(f(h, m)) * (m + x)
            return θ

        # Create set of possible action combinations, A
        A1 = np.linspace(h_min, h_max, n_h).reshape(n_h, 1)
        A2 = np.linspace(self.m_min, self.m_max, n_m).reshape(n_m, 1)
        self.A = np.concatenate((np.kron(np.ones((n_m, 1)), A1),
```

(continues on next page)

(continued from previous page)

```
                            np.kron(A2, np.ones((n_h, 1)))), axis=1)

        # Pre-compute utility and output vectors
        self.euler_vec = -np.multiply(self.A[:, 1], \
            uc_p(f(self.A[:, 0], self.A[:, 1])) - v_p(self.A[:, 1]))
        self.u_vec = u(self.A[:, 0], self.A[:, 1])
        self.Θ_vec = θ(self.A[:, 0], self.A[:, 1])
        self.f_vec = f(self.A[:, 0], self.A[:, 1])
        self.bell_vec = np.multiply(uc_p(f(self.A[:, 0],
                                    self.A[:, 1])),
                                    np.multiply(self.A[:, 1],
                                    (self.A[:, 0] - 1))) \
                        + np.multiply(self.A[:, 1],
                                      v_p(self.A[:, 1]))

        # Find extrema of (w, θ) space for initial guess of equilibrium sets
        p_vec = np.zeros(self.N_a)
        w_vec = np.zeros(self.N_a)
        for i in range(self.N_a):
            p_vec[i] = self.Θ_vec[i]
            w_vec[i] = self.u_vec[i]/(1 - β)

        w_space = np.array([min(w_vec[~np.isinf(w_vec)]),
                            max(w_vec[~np.isinf(w_vec)])])
        p_space = np.array([0, max(p_vec[~np.isinf(w_vec)])])
        self.p_space = p_space

        # Set up hyperplane levels and gradients for iterations
        def SG_H_V(N, w_space, p_space):
            """
            This function  initializes the subgradients, hyperplane levels,
            and extreme points of the value set by choosing an appropriate
            origin and radius. It is based on a similar function in QuantEcon's
            Games.jl
            """

            # First, create a unit circle. Want points placed on [0, 2π]
            inc = 2 * np.pi / N
            degrees = np.arange(0, 2 * np.pi, inc)

            # Points on circle
            H = np.zeros((N, 2))
            for i in range(N):
                x = degrees[i]
                H[i, 0] = np.cos(x)
                H[i, 1] = np.sin(x)

            # Then calculate origin and radius
            o = np.array([np.mean(w_space), np.mean(p_space)])
            r1 = max((max(w_space) - o[0])**2, (o[0] - min(w_space))**2)
            r2 = max((max(p_space) - o[1])**2, (o[1] - min(p_space))**2)
            r = np.sqrt(r1 + r2)

            # Now calculate vertices
            Z = np.zeros((2, N))
            for i in range(N):
```

(continues on next page)

(continued from previous page)

```
                Z[0, i] = o[0] + r*H.T[0, i]
                Z[1, i] = o[1] + r*H.T[1, i]

            # Corresponding hyperplane levels
            C = np.zeros(N)
            for i in range(N):
                C[i] = np.dot(Z[:, i], H[i, :])

            return C, H, Z

        C, self.H, Z = SG_H_V(N_g, w_space, p_space)
        C = C.reshape(N_g, 1)
        self.c0_c, self.c0_s, self.c1_c, self.c1_s = np.copy(C), np.copy(C), \
            np.copy(C), np.copy(C)
        self.z0_s, self.z0_c, self.z1_s, self.z1_c = np.copy(Z), np.copy(Z), \
            np.copy(Z), np.copy(Z)

        self.w_bnds_s, self.w_bnds_c = (w_space[0], w_space[1]), \
            (w_space[0], w_space[1])
        self.p_bnds_s, self.p_bnds_c = (p_space[0], p_space[1]), \
            (p_space[0], p_space[1])

        # Create dictionaries to save equilibrium set for each iteration
        self.c_dic_s, self.c_dic_c = {}, {}
        self.c_dic_s[0], self.c_dic_c[0] = self.c0_s, self.c0_c

    def solve_worst_spe(self):
        """
        Method to solve for BR(Z). See p.449 of Chang (1998)
        """

        p_vec = np.full(self.N_a, np.nan)
        c = [1, 0]

        # Pre-compute constraints
        aineq_mbar = np.vstack((self.H, np.array([0, -self.β])))
        bineq_mbar = np.vstack((self.c0_s, 0))

        aineq = self.H
        bineq = self.c0_s
        aeq = [[0, -self.β]]

        for j in range(self.N_a):
            # Only try if consumption is possible
            if self.f_vec[j] > 0:
                # If m = mbar, use inequality constraint
                if self.A[j, 1] == self.mbar:
                    bineq_mbar[-1] = self.euler_vec[j]
                    res = linprog(c, A_ub=aineq_mbar, b_ub=bineq_mbar,
                                  bounds=(self.w_bnds_s, self.p_bnds_s))
                else:
                    beq = self.euler_vec[j]
                    res = linprog(c, A_ub=aineq, b_ub=bineq, A_eq=aeq, b_eq=beq,
                                  bounds=(self.w_bnds_s, self.p_bnds_s))
                if res.status == 0:
                    p_vec[j] = self.u_vec[j] + self.β * res.x[0]
```

(continues on next page)

(continued from previous page)

```
        # Max over h and min over other variables (see Chang (1998) p.449)
        self.br_z = np.nanmax(np.nanmin(p_vec.reshape(self.n_m, self.n_h), 0))

    def solve_subgradient(self):
        """
        Method to solve for E(Z). See p.449 of Chang (1998)
        """

        # Pre-compute constraints
        aineq_C_mbar = np.vstack((self.H, np.array([0, -self.β])))
        bineq_C_mbar = np.vstack((self.c0_c, 0))

        aineq_C = self.H
        bineq_C = self.c0_c
        aeq_C = [[0, -self.β]]

        aineq_S_mbar = np.vstack((np.vstack((self.H, np.array([0, -self.β]))),
                                  np.array([-self.β, 0])))
        bineq_S_mbar = np.vstack((self.c0_s, np.zeros((2, 1))))

        aineq_S = np.vstack((self.H, np.array([-self.β, 0])))
        bineq_S = np.vstack((self.c0_s, 0))
        aeq_S = [[0, -self.β]]

        # Update maximal hyperplane level
        for i in range(self.N_g):
            c_a1a2_c, t_a1a2_c = np.full(self.N_a, -np.inf), \
                np.zeros((self.N_a, 2))
            c_a1a2_s, t_a1a2_s = np.full(self.N_a, -np.inf), \
                np.zeros((self.N_a, 2))

            c = [-self.H[i, 0], -self.H[i, 1]]

            for j in range(self.N_a):
                # Only try if consumption is possible
                if self.f_vec[j] > 0:

                    # COMPETITIVE EQUILIBRIA
                    # If m = mbar, use inequality constraint
                    if self.A[j, 1] == self.mbar:
                        bineq_C_mbar[-1] = self.euler_vec[j]
                        res = linprog(c, A_ub=aineq_C_mbar, b_ub=bineq_C_mbar,
                                      bounds=(self.w_bnds_c, self.p_bnds_c))
                    # If m < mbar, use equality constraint
                    else:
                        beq_C = self.euler_vec[j]
                        res = linprog(c, A_ub=aineq_C, b_ub=bineq_C, A_eq = aeq_C,
                                      b_eq = beq_C, bounds=(self.w_bnds_c, \
                                          self.p_bnds_c))
                    if res.status == 0:
                        c_a1a2_c[j] = self.H[i, 0] * (self.u_vec[j] \
                            + self.β * res.x[0]) + self.H[i, 1] * self.Θ_vec[j]
                        t_a1a2_c[j] = res.x

                    # SUSTAINABLE EQUILIBRIA
```

(continues on next page)

(continued from previous page)

```
                    # If m = mbar, use inequality constraint
                    if self.A[j, 1] == self.mbar:
                        bineq_S_mbar[-2] = self.euler_vec[j]
                        bineq_S_mbar[-1] = self.u_vec[j] - self.br_z
                        res = linprog(c, A_ub=aineq_S_mbar, b_ub=bineq_S_mbar,
                                      bounds=(self.w_bnds_s, self.p_bnds_s))
                    # If m < mbar, use equality constraint
                    else:
                        bineq_S[-1] = self.u_vec[j] - self.br_z
                        beq_S = self.euler_vec[j]
                        res = linprog(c, A_ub=aineq_S, b_ub=bineq_S, A_eq = aeq_S,
                                      b_eq = beq_S, bounds=(self.w_bnds_s, \
                                          self.p_bnds_s))
                    if res.status == 0:
                        c_a1a2_s[j] = self.H[i, 0] * (self.u_vec[j] \
                            + self.β*res.x[0]) + self.H[i, 1] * self.Θ_vec[j]
                        t_a1a2_s[j] = res.x

            idx_c = np.where(c_a1a2_c == max(c_a1a2_c))[0][0]
            self.z1_c[:, i] = np.array([self.u_vec[idx_c]
                                        + self.β * t_a1a2_c[idx_c, 0],
                                        self.Θ_vec[idx_c]])

            idx_s = np.where(c_a1a2_s == max(c_a1a2_s))[0][0]
            self.z1_s[:, i] = np.array([self.u_vec[idx_s]
                                        + self.β * t_a1a2_s[idx_s, 0],
                                        self.Θ_vec[idx_s]])

        for i in range(self.N_g):
            self.c1_c[i] = np.dot(self.z1_c[:, i], self.H[i, :])
            self.c1_s[i] = np.dot(self.z1_s[:, i], self.H[i, :])

    def solve_sustainable(self, tol=1e-5, max_iter=250):
        """
        Method to solve for the competitive and sustainable equilibrium sets.
        """

        t = time.time()
        diff = tol + 1
        iters = 0

        print('### --------------- ###')
        print('Solving Chang Model Using Outer Hyperplane Approximation')
        print('### --------------- ### \n')

        print('Maximum difference when updating hyperplane levels:')

        while diff > tol and iters < max_iter:
            iters = iters + 1
            self.solve_worst_spe()
            self.solve_subgradient()
            diff = max(np.maximum(abs(self.c0_c - self.c1_c),
                       abs(self.c0_s - self.c1_s)))
            print(diff)

            # Update hyperplane levels
```

(continues on next page)

(continued from previous page)

```
            self.c0_c, self.c0_s = np.copy(self.c1_c), np.copy(self.c1_s)

            # Update bounds for w and θ
            wmin_c, wmax_c = np.min(self.z1_c, axis=1)[0], \
                np.max(self.z1_c, axis=1)[0]
            pmin_c, pmax_c = np.min(self.z1_c, axis=1)[1], \
                np.max(self.z1_c, axis=1)[1]

            wmin_s, wmax_s = np.min(self.z1_s, axis=1)[0], \
                np.max(self.z1_s, axis=1)[0]
            pmin_S, pmax_S = np.min(self.z1_s, axis=1)[1], \
                np.max(self.z1_s, axis=1)[1]

            self.w_bnds_s, self.w_bnds_c = (wmin_s, wmax_s), (wmin_c, wmax_c)
            self.p_bnds_s, self.p_bnds_c = (pmin_S, pmax_S), (pmin_c, pmax_c)

            # Save iteration
            self.c_dic_c[iters], self.c_dic_s[iters] = np.copy(self.c1_c), \
                np.copy(self.c1_s)
            self.iters = iters

        elapsed = time.time() - t
        print('Convergence achieved after {} iterations and {} \
            seconds'.format(iters, round(elapsed, 2)))

    def solve_bellman(self, θ_min, θ_max, order, disp=False, tol=1e-7, maxiters=100):
        """
        Continuous Method to solve the Bellman equation in section 25.3
        """
        mbar = self.mbar

        # Utility and production functions
        uc = lambda c: np.log(np.maximum(c, 1e-10))  # Clip to 1e-10 to avoid log(0)␣
↪or log(-ve)
        uc_p = lambda c: 1 / c
        v = lambda m: 1 / 500 * (mbar * m - 0.5 * m**2)**0.5
        v_p = lambda m: 0.5/500 * (mbar*m - 0.5 * m**2)**(-0.5) * (mbar - m)
        u = lambda h, m: uc(f(h, m)) + v(m)

        def f(h, m):
            x = m * (h - 1)
            f = 180 - (0.4 * x)**2
            return f

        def θ(h, m):
            x = m * (h - 1)
            θ = uc_p(f(h, m)) * (m + x)
            return θ

        # Bounds for Maximization
        lb1 = np.array([self.h_min, 0, θ_min])
        ub1 = np.array([self.h_max, self.mbar - 1e-5, θ_max])
        lb2 = np.array([self.h_min, θ_min])
        ub2 = np.array([self.h_max, θ_max])

        # Initialize Value Function coefficients
```

(continues on next page)

(continued from previous page)

```
        # Calculate roots of Chebyshev polynomial
        k = np.linspace(order, 1, order)
        roots = np.cos((2 * k - 1) * np.pi / (2 * order))
        # Scale to approximation space
        s = θ_min + (roots - -1) / 2 * (θ_max - θ_min)
        # Create a basis matrix
        Φ = cheb.chebvander(roots, order - 1)
        c = np.zeros(Φ.shape[0])

        # Function to minimize and constraints
        def p_fun(x):
            scale = -1 + 2 * (x[2] - θ_min)/(θ_max - θ_min)
            p_fun = - (u(x[0], x[1]) \
                + self.β * np.dot(cheb.chebvander(scale, order - 1), c))
            return p_fun.item()

        def p_fun2(x):
            scale = -1 + 2*(x[1] - θ_min)/(θ_max - θ_min)
            p_fun = - (u(x[0],mbar) \
                + self.β * np.dot(cheb.chebvander(scale, order - 1), c))
            return p_fun.item()

        cons1 = ({'type': 'eq',   'fun': lambda x: uc_p(f(x[0], x[1])) * x[1]
                    * (x[0] - 1) + v_p(x[1]) * x[1] + self.β * x[2] - θ},
                 {'type': 'eq',   'fun': lambda x: uc_p(f(x[0], x[1]))
                    * x[0] * x[1] - θ})
        cons2 = ({'type': 'ineq', 'fun': lambda x: uc_p(f(x[0], mbar)) * mbar
                    * (x[0] - 1) + v_p(mbar) * mbar + self.β * x[1] - θ},
                 {'type': 'eq',   'fun': lambda x: uc_p(f(x[0], mbar))
                    * x[0] * mbar - θ})

        bnds1 = np.concatenate([lb1.reshape(3, 1), ub1.reshape(3, 1)], axis=1)
        bnds2 = np.concatenate([lb2.reshape(2, 1), ub2.reshape(2, 1)], axis=1)

        # Bellman Iterations
        diff = 1
        iters = 1

        while diff > tol:
        # 1. Maximization, given value function guess
            p_iter1 = np.zeros(order)
            for i in range(order):
                θ = s[i]
                x0 = np.clip(lb1 + (ub1-lb1)/2, lb1, ub1)
                res = minimize(p_fun,
                               x0,
                               method='SLSQP',
                               bounds=bnds1,
                               constraints=cons1,
                               tol=1e-10)
                if res.success == True:
                    p_iter1[i] = -p_fun(res.x)
                x0 = np.clip(lb2 + (ub2-lb2)/2, lb2, ub2)
                res = minimize(p_fun2,
                               x0,
                               method='SLSQP',
```

(continues on next page)

(continued from previous page)

```
                            bounds=bnds2,
                            constraints=cons2,
                            tol=1e-10)
                if -p_fun2(res.x) > p_iter1[i] and res.success == True:
                    p_iter1[i] = -p_fun2(res.x)

            # 2. Bellman updating of Value Function coefficients
            c1 = np.linalg.solve(Φ, p_iter1)
            # 3. Compute distance and update
            diff = np.linalg.norm(c - c1)
            if bool(disp == True):
                print(diff)
            c = np.copy(c1)
            iters = iters + 1
            if iters > maxiters:
                print('Convergence failed after {} iterations'.format(maxiters))
                break

        self.θ_grid = s
        self.p_iter = p_iter1
        self.Φ = Φ
        self.c = c
        print('Convergence achieved after {} iterations'.format(iters))

        # Check residuals
        θ_grid_fine = np.linspace(θ_min, θ_max, 100)
        resid_grid = np.zeros(100)
        p_grid = np.zeros(100)
        θ_prime_grid = np.zeros(100)
        m_grid = np.zeros(100)
        h_grid = np.zeros(100)
        for i in range(100):
            θ = θ_grid_fine[i]
            x0 = np.clip(lb1 + (ub1-lb1)/2, lb1, ub1)
            res = minimize(p_fun,
                           x0,
                           method='SLSQP',
                           bounds=bnds1,
                           constraints=cons1,
                           tol=1e-10)
            if res.success == True:
                p = -p_fun(res.x)
                p_grid[i] = p
                θ_prime_grid[i] = res.x[2]
                h_grid[i] = res.x[0]
                m_grid[i] = res.x[1]
            x0 = np.clip(lb2 + (ub2-lb2)/2, lb2, ub2)
            res = minimize(p_fun2,
                           x0,
                           method='SLSQP',
                           bounds=bnds2,
                           constraints=cons2,
                           tol=1e-10)
            if -p_fun2(res.x) > p and res.success == True:
                p = -p_fun2(res.x)
                p_grid[i] = p
```

(continues on next page)

(continued from previous page)

```
                θ_prime_grid[i] = res.x[1]
                h_grid[i] = res.x[0]
                m_grid[i] = self.mbar
            scale = -1 + 2 * (θ - θ_min)/(θ_max - θ_min)
            resid_grid_val = np.dot(cheb.chebvander(scale, order-1), c) - p
            resid_grid[i] = resid_grid_val.item()

        self.resid_grid = resid_grid
        self.θ_grid_fine = θ_grid_fine
        self.θ_prime_grid = θ_prime_grid
        self.m_grid = m_grid
        self.h_grid = h_grid
        self.p_grid = p_grid
        self.x_grid = m_grid * (h_grid - 1)

        # Simulate
        θ_series = np.zeros(31)
        m_series = np.zeros(30)
        h_series = np.zeros(30)

        # Find initial θ
        def ValFun(x):
            scale = -1 + 2*(x - θ_min)/(θ_max - θ_min)
            p_fun = np.dot(cheb.chebvander(scale, order - 1), c)
            return -p_fun

        res = minimize(ValFun,
                       (θ_min + θ_max)/2,
                       bounds=[(θ_min, θ_max)])
        θ_series[0] = res.x.item()

        # Simulate
        for i in range(30):
            θ = θ_series[i]
            x0 = np.clip(lb1 + (ub1-lb1)/2, lb1, ub1)
            res = minimize(p_fun,
                           x0,
                           method='SLSQP',
                           bounds=bnds1,
                           constraints=cons1,
                           tol=1e-10)
            if res.success == True:
                p = -p_fun(res.x)
                h_series[i] = res.x[0]
                m_series[i] = res.x[1]
                θ_series[i+1] = res.x[2]
            x0 = np.clip(lb2 + (ub2-lb2)/2, lb2, ub2)
            res2 = minimize(p_fun2,
                            x0,
                            method='SLSQP',
                            bounds=bnds2,
                            constraints=cons2,
                            tol=1e-10)
            if -p_fun2(res2.x) > p and res2.success == True:
                h_series[i] = res2.x[0]
                m_series[i] = self.mbar
```

(continues on next page)

(continued from previous page)

```
                θ_series[i+1] = res2.x[1]

        self.θ_series = θ_series
        self.m_series = m_series
        self.h_series = h_series
        self.x_series = m_series * (h_series - 1)
```

```
ch1 = ChangModel(β=0.3, mbar=30, h_min=0.9, h_max=2, n_h=8, n_m=35, N_g=10)
ch1.solve_sustainable()
```

```
### --------------- ###
Solving Chang Model Using Outer Hyperplane Approximation
### --------------- ###

Maximum difference when updating hyperplane levels:
```

```
[1.9168]
```

```
[0.66782]
```

```
[0.49235]
```

```
[0.32412]
```

```
[0.19022]
```

```
[0.10863]
```

```
[0.05817]
```

```
[0.0262]
```

```
[0.01836]
```

```
[0.01415]
```

```
[0.00297]
```

```
[0.00089]
```

```
[0.00027]
```

```
[0.00008]
```

```
[0.00002]
```

```
[0.00001]
Convergence achieved after 16 iterations and 39.06              seconds
```

```
def plot_competitive(ChangModel):
    """
    Method that only plots competitive equilibrium set
    """
    poly_C = polytope.Polytope(ChangModel.H, ChangModel.c1_c)
    ext_C = polytope.extreme(poly_C)

    fig, ax = plt.subplots(figsize=(7, 5))

    ax.set_xlabel('w', fontsize=16)
    ax.set_ylabel(r"$\theta$", fontsize=18)

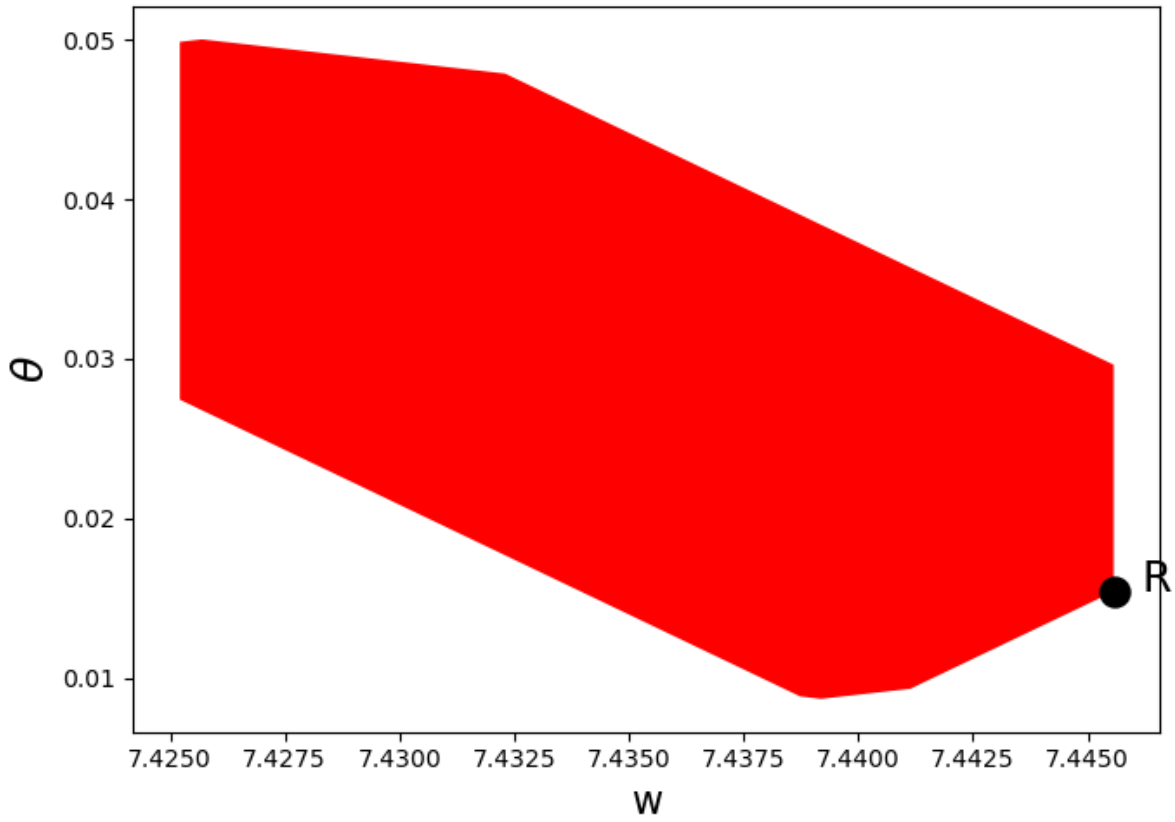
    ax.fill(ext_C[:,0], ext_C[:,1], 'r', zorder=0)
    ChangModel.min_theta = min(ext_C[:, 1])
    ChangModel.max_theta = max(ext_C[:, 1])

    # Add point showing Ramsey Plan
    idx_Ramsey = np.where(ext_C[:, 0] == max(ext_C[:, 0]))[0][0]
    R = ext_C[idx_Ramsey, :]
    ax.scatter(R[0], R[1], 150, 'black', marker='o', zorder=1)
    w_min = min(ext_C[:, 0])

    # Label Ramsey Plan slightly to the right of the point
    ax.annotate("R", xy=(R[0], R[1]), xytext=(R[0] + 0.03 * (R[0] - w_min),
                R[1]), fontsize=18)

    plt.tight_layout()
    plt.show()

plot_competitive(ch1)
```




```
ch2 = ChangModel(β=0.8, mbar=30, h_min=0.9, h_max=1/0.8,
                 n_h=8, n_m=35, N_g=10)
ch2.solve_sustainable()
```

```
### --------------- ###
Solving Chang Model Using Outer Hyperplane Approximation
### --------------- ###

Maximum difference when updating hyperplane levels:
```

```
[0.06369]
```

```
[0.02476]
```

```
[0.02153]
```

```
[0.01915]
```

```
[0.01795]
```

```
[0.01642]
```

```
[0.01507]
```

```
[0.01284]
```

```
[0.01106]
```

```
[0.00694]
```

```
[0.0085]
```

```
[0.00781]
```

```
[0.00433]
```

```
[0.00492]
```

```
[0.00303]
```

```
[0.00182]
```

```
[0.00638]
```

```
[0.00116]
```

```
[0.00093]
```

```
[0.00075]
```

```
[0.0006]
```

```
[0.00494]
```

```
[0.00038]
```

```
[0.00121]
```

```
[0.00024]
```

```
[0.0002]
```

```
[0.00016]
```

```
[0.00013]
```

```
[0.0001]
```

```
[0.00008]
```

```
[0.00006]
```

```
[0.00005]
```

```
[0.00004]
```

```
[0.00003]
```

```
[0.00003]
```

```
[0.00002]
```

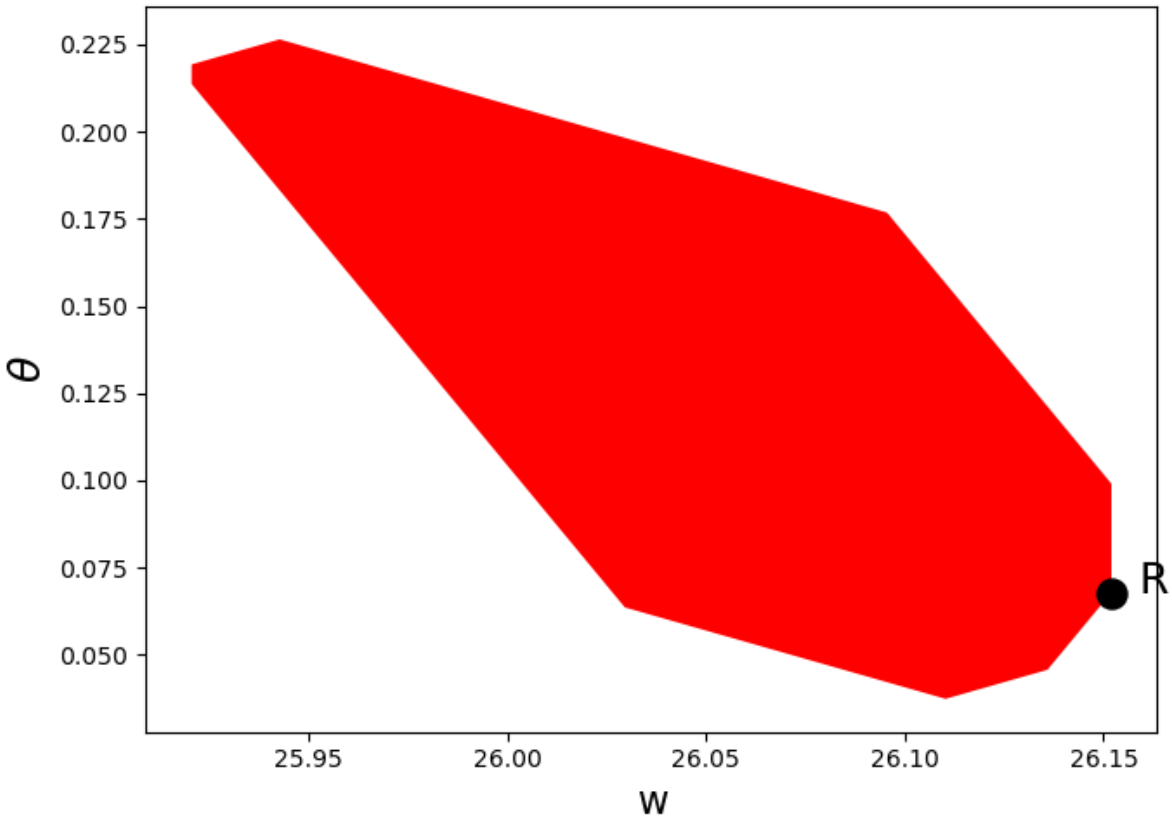
```
[0.00002]
```

```
[0.00001]
```

```
[0.00001]
```

```
[0.00001]
Convergence achieved after 40 iterations and 113.42             seconds
```

```
plot_competitive(ch2)
```



49.7 Solving a Continuation Ramsey Planner's Bellman Equation

In this section we solve the Bellman equation confronting a **continuation Ramsey planner**.

The construction of a Ramsey plan is decomposed into a two subproblems in *Ramsey plans, time inconsistency, sustainable plans* and *dynamic Stackelberg problems*.

- Subproblem 1 is faced by a sequence of continuation Ramsey planners at $t \geq 1$.
- Subproblem 2 is faced by a Ramsey planner at $t = 0$.

The problem is:

$$J(\theta) = \max_{m,x,h,\theta'} u(f(x)) + v(m) + \beta J(\theta')$$

subject to:

$$\theta \leq u'(f(x))x + v'(m)m + \beta\theta'$$

$$\theta = u'(f(x))(m + x)$$

$$x = m(h - 1)$$

$$(m, x, h) \in E$$

$$\theta' \in \Omega$$

To solve this Bellman equation, we must know the set Ω.

We have solved the Bellman equation for the two sets of parameter values for which we computed the equilibrium value sets above.

Hence for these parameter configurations, we know the bounds of Ω.

The two sets of parameters differ only in the level of β.

From the figures earlier in this lecture, we know that when $\beta = 0.3$, $\Omega = [0.0088, 0.0499]$, and when $\beta = 0.8$, $\Omega = [0.0395, 0.2193]$

```
ch1 = ChangModel(β=0.3, mbar=30, h_min=0.99, h_max=1/0.3,
                 n_h=8, n_m=35, N_g=50)
ch2 = ChangModel(β=0.8, mbar=30, h_min=0.1, h_max=1/0.8,
                 n_h=20, n_m=50, N_g=50)
```

```
ch1.solve_bellman(θ_min=0.01, θ_max=0.0499, order=30, tol=1e-6)
ch2.solve_bellman(θ_min=0.045, θ_max=0.15, order=30, tol=1e-6)
```

```
Convergence achieved after 15 iterations
```

```
/home/runner/miniconda3/envs/quantecon/lib/python3.12/site-packages/scipy/optimize/
 ↪_slsqp_py.py:437: RuntimeWarning: Values in x were outside bounds during a␣
 ↪minimize step, clipping to bounds
  fx = wrapped_fun(x)
```

```
/home/runner/miniconda3/envs/quantecon/lib/python3.12/site-packages/scipy/optimize/
 ↪_slsqp_py.py:441: RuntimeWarning: Values in x were outside bounds during a␣
 ↪minimize step, clipping to bounds
  g = append(wrapped_grad(x), 0.0)
/home/runner/miniconda3/envs/quantecon/lib/python3.12/site-packages/scipy/optimize/
 ↪_slsqp_py.py:495: RuntimeWarning: Values in x were outside bounds during a␣
 ↪minimize step, clipping to bounds
  a_eq = vstack([con['jac'](x, *con['args'])
/home/runner/miniconda3/envs/quantecon/lib/python3.12/site-packages/scipy/optimize/
 ↪_slsqp_py.py:501: RuntimeWarning: Values in x were outside bounds during a␣
 ↪minimize step, clipping to bounds
  a_ieq = vstack([con['jac'](x, *con['args'])
```

```
Convergence achieved after 72 iterations
```

First, a quick check that our approximations of the value functions are good.

We do this by calculating the residuals between iterates on the value function on a fine grid:

```
max(abs(ch1.resid_grid)), max(abs(ch2.resid_grid))
```

```
(6.46313155971967e-06, 6.875358415925348e-07)
```

The value functions plotted below trace out the right edges of the sets of equilibrium values plotted above

```
fig, axes = plt.subplots(1, 2, figsize=(12, 4))

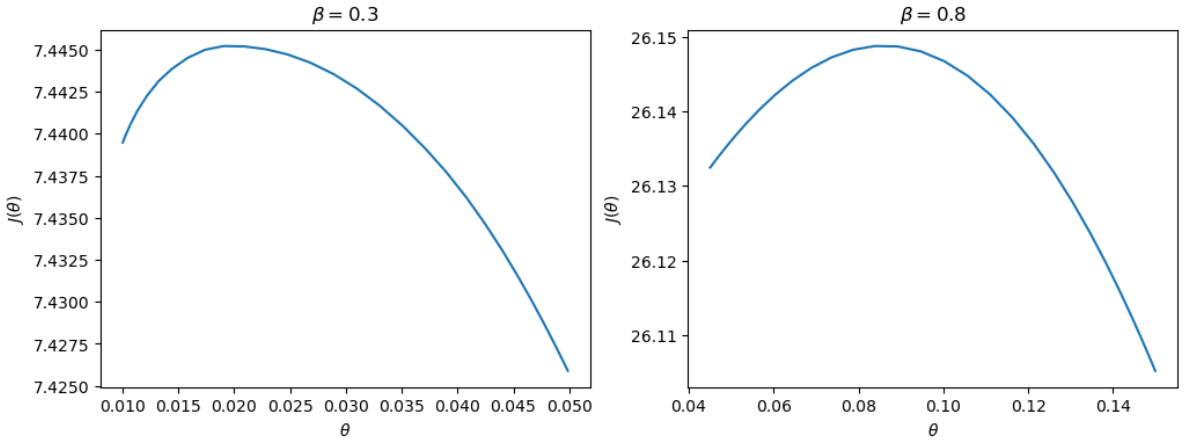
for ax, model in zip(axes, (ch1, ch2)):
```

(continues on next page)

(continued from previous page)

```
    ax.plot(model.θ_grid, model.p_iter)
    ax.set(xlabel=r"$\theta$",
           ylabel=r"$J(\theta)$",
           title=rf"$\beta = {model.β}$")

plt.show()
```



The next figure plots the optimal policy functions; values of θ', m, x, h for each value of the state θ:

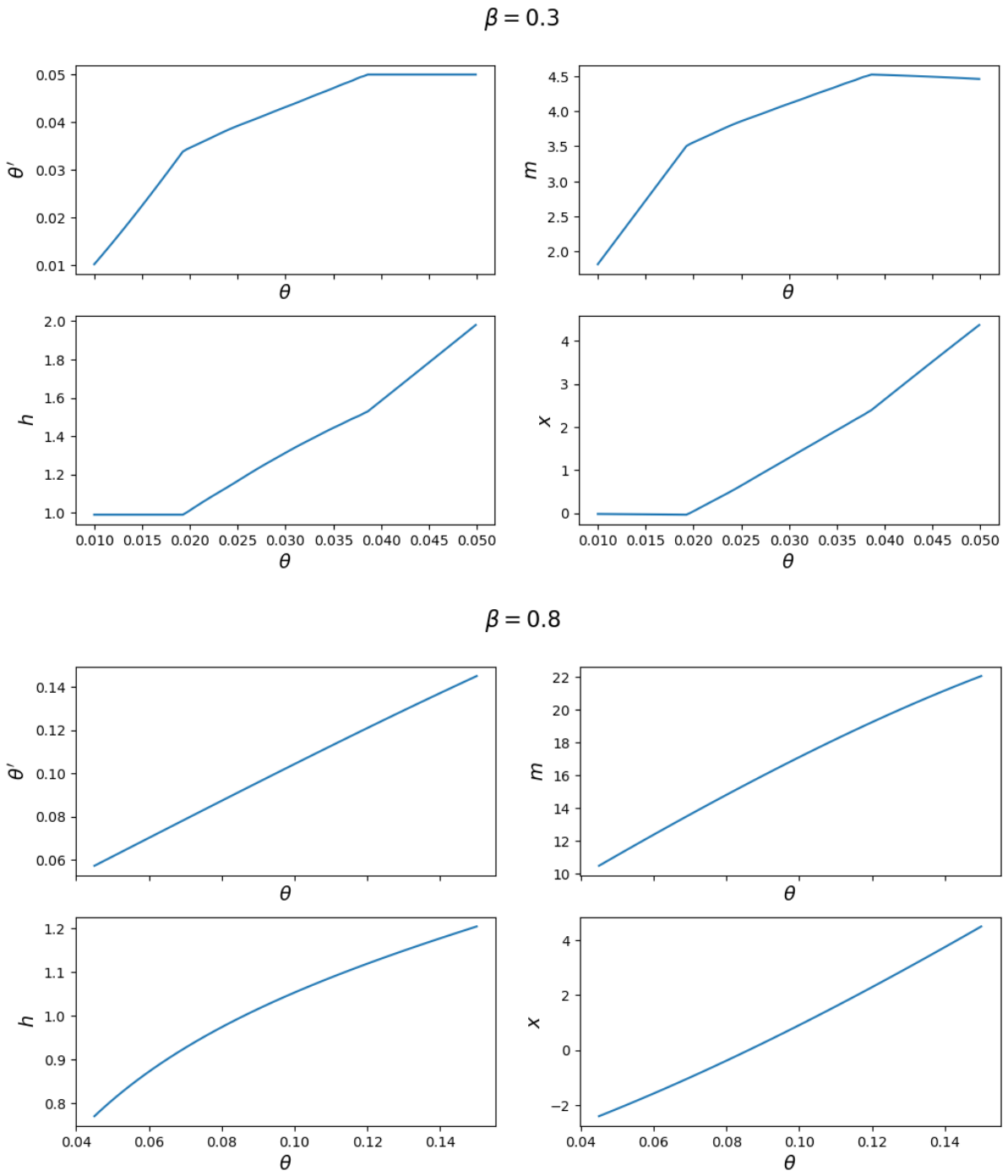
```
for model in (ch1, ch2):

    fig, axes = plt.subplots(2, 2, figsize=(12, 6), sharex=True)
    fig.suptitle(rf"$\beta = {model.β}$", fontsize=16)

    plots = [model.θ_prime_grid, model.m_grid,
             model.h_grid, model.x_grid]
    labels = [r"$\theta'$", "$m$", "$h$", "$x$"]

    for ax, plot, label in zip(axes.flatten(), plots, labels):
        ax.plot(model.θ_grid_fine, plot)
        ax.set_xlabel(r"$\theta$", fontsize=14)
        ax.set_ylabel(label, fontsize=14)

    plt.show()
```



With the first set of parameter values, the value of θ' chosen by the Ramsey planner quickly hits the upper limit of Ω.

But with the second set of parameters it converges to a value in the interior of the set.

Consequently, the choice of $\bar{\theta}$ is clearly important with the first set of parameter values.

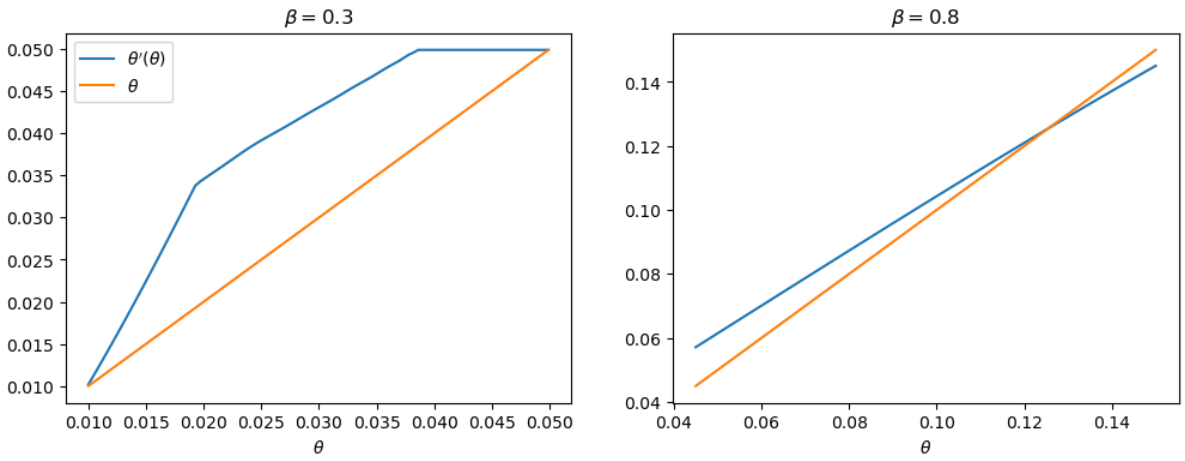
One way of seeing this is plotting $\theta'(\theta)$ for each set of parameters.

With the first set of parameter values, this function does not intersect the 45-degree line until $\bar{\theta}$, whereas in the second set of parameter values, it intersects in the interior.

```
fig, axes = plt.subplots(1, 2, figsize=(12, 4))

for ax, model in zip(axes, (ch1, ch2)):
    ax.plot(model.θ_grid_fine, model.θ_prime_grid, label=r"$\theta'(\theta)$")
    ax.plot(model.θ_grid_fine, model.θ_grid_fine, label=r"$\theta$")
    ax.set(xlabel=r"$\theta$", title=rf"$\beta = {model.β}$")

axes[0].legend()
plt.show()
```



Subproblem 2 is equivalent to the planner choosing the initial value of θ (i.e. the value which maximizes the value function).

From this starting point, we can then trace out the paths for $\{\theta_t, m_t, h_t, x_t\}_{t=0}^{\infty}$ that support this equilibrium.

These are shown below for both sets of parameters

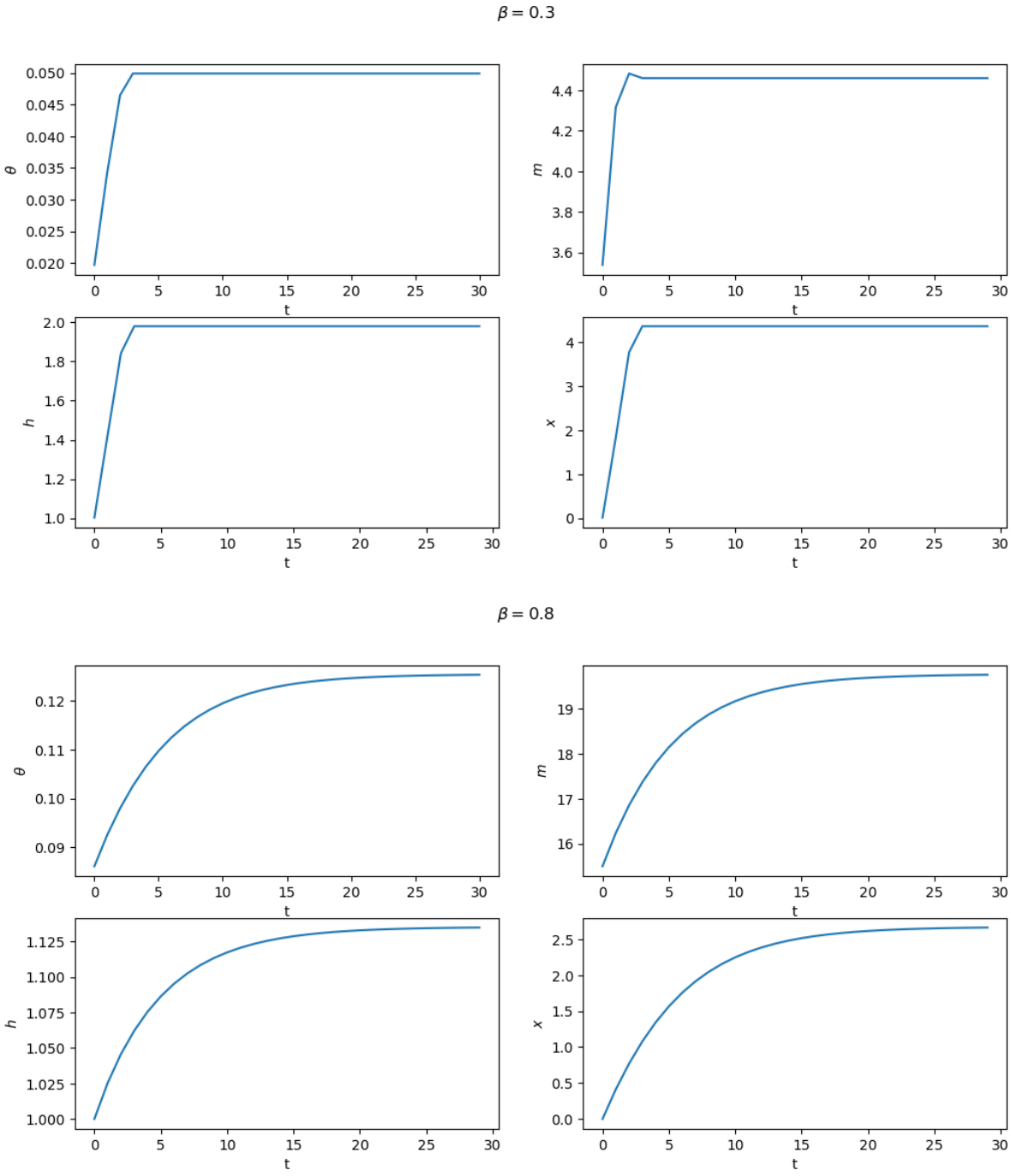
```
for model in (ch1, ch2):

    fig, axes = plt.subplots(2, 2, figsize=(12, 6))
    fig.suptitle(rf"$\beta = {model.β}$")

    plots = [model.θ_series, model.m_series, model.h_series, model.x_series]
    labels = [r"$\theta$", "$m$", "$h$", "$x$"]

    for ax, plot, label in zip(axes.flatten(), plots, labels):
        ax.plot(plot)
        ax.set(xlabel='t', ylabel=label)

    plt.show()
```


$\beta = 0.3$
θ
m
h
x
t
$\beta = 0.8$
θ
m
h
x
t

49.7.1 Next Steps

In *Credible Government Policies in Chang Model* we shall find a subset of competitive equilibria that are **sustainable** in the sense that a sequence of government administrations that chooses sequentially, rather than once and for all at time 0 will choose to implement them.

In the process of constructing them, we shall construct another, smaller set of competitive equilibria.

CHAPTER

FIFTY

CREDIBLE GOVERNMENT POLICIES IN A MODEL OF CHANG

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install polytope
```

50.1 Overview

Some of the material in this lecture and *competitive equilibria in the Chang model* can be viewed as more sophisticated and complete treatments of the topics discussed in *Ramsey plans, time inconsistency, sustainable plans*.

This lecture assumes almost the same economic environment analyzed in *competitive equilibria in the Chang model*.

The only change – and it is a substantial one – is the timing protocol for making government decisions.

In *competitive equilibria in the Chang model*, a *Ramsey planner* chose a comprehensive government policy once-and-for-all at time 0.

Now in this lecture, there is no time 0 Ramsey planner.

Instead there is a sequence of government decision-makers, one for each t.

The time t government decision-maker choose time t government actions after forecasting what future governments will do.

We use the notion of a *sustainable plan* proposed in [Chari and Kehoe, 1990], also referred to as a *credible public policy* in [Stokey, 1989].

Technically, this lecture starts where lecture *competitive equilibria in the Chang model* on Ramsey plans within the Chang [Chang, 1998] model stopped.

That lecture presents recursive representations of *competitive equilibria* and a *Ramsey plan* for a version of a model of Calvo [Calvo, 1978] that Chang used to analyze and illustrate these concepts.

We used two operators to characterize competitive equilibria and a Ramsey plan, respectively.

In this lecture, we define a *credible public policy* or *sustainable plan*.

Starting from a large enough initial set Z_0, we use iterations on Chang's set-to-set operator $\tilde{D}(Z)$ to compute a set of values associated with sustainable plans.

Chang's operator $\tilde{D}(Z)$ is closely connected with the operator $D(Z)$ introduced in lecture *competitive equilibria in the Chang model*.

- $\tilde{D}(Z)$ incorporates all of the restrictions imposed in constructing the operator $D(Z)$, but ....
- It adds some additional restrictions
  - these additional restrictions incorporate the idea that a plan must be *sustainable*.

- *sustainable* means that the government wants to implement it at all times after all histories.

Let's start with some standard imports:

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

```
`polytope` failed to import `cvxopt.glpk`.
```

```
will use `scipy.optimize.linprog`
```

50.2 The Setting

We begin by reviewing the set up deployed in *competitive equilibria in the Chang model*.

Chang's model, adopted from Calvo, is designed to focus on the intertemporal trade-offs between the welfare benefits of deflation and the welfare costs associated with the high tax collections required to retire money at a rate that delivers deflation.

A benevolent time 0 government can promote utility generating increases in real balances only by imposing an infinite sequence of sufficiently large distorting tax collections.

To promote the welfare increasing effects of high real balances, the government wants to induce *gradual deflation*.

We start by reviewing notation.

For a sequence of scalars $\vec{z} \equiv \{z_t\}_{t=0}^{\infty}$, let $\vec{z}^t = (z_0, \ldots, z_t)$, $\vec{z}_t = (z_t, z_{t+1}, \ldots)$.

An infinitely lived representative agent and an infinitely lived government exist at dates $t = 0, 1, \ldots$.

The objects in play are

- an initial quantity M_{-1} of nominal money holdings
- a sequence of inverse money growth rates $\vec{h}$ and an associated sequence of nominal money holdings $\vec{M}$
- a sequence of values of money $\vec{q}$
- a sequence of real money holdings $\vec{m}$
- a sequence of total tax collections $\vec{x}$
- a sequence of per capita rates of consumption $\vec{c}$
- a sequence of per capita incomes $\vec{y}$

A benevolent government chooses sequences $(\vec{M}, \vec{h}, \vec{x})$ subject to a sequence of budget constraints and other constraints imposed by competitive equilibrium.

Given tax collection and price of money sequences, a representative household chooses sequences $(\vec{c}, \vec{m})$ of consumption and real balances.

In competitive equilibrium, the price of money sequence $\vec{q}$ clears markets, thereby reconciling decisions of the government and the representative household.

50.2.1 The Household's Problem

A representative household faces a nonnegative value of money sequence $\vec{q}$ and sequences $\vec{y}, \vec{x}$ of income and total tax collections, respectively.

The household chooses nonnegative sequences $\vec{c}, \vec{M}$ of consumption and nominal balances, respectively, to maximize

$$\sum_{t=0}^{\infty} \beta^t \left[u(c_t) + v(q_t M_t) \right] \tag{50.1}$$

subject to

$$q_t M_t \leq y_t + q_t M_{t-1} - c_t - x_t \tag{50.2}$$

and

$$q_t M_t \leq \bar{m} \tag{50.3}$$

Here q_t is the reciprocal of the price level at t, also known as the *value of money*.

Chang [Chang, 1998] assumes that

- $u : \mathbb{R}_+ \to \mathbb{R}$ is twice continuously differentiable, strictly concave, and strictly increasing;
- $v : \mathbb{R}_+ \to \mathbb{R}$ is twice continuously differentiable and strictly concave;
- $u'(c)_{c \to 0} = \lim_{m \to 0} v'(m) = +\infty$;
- there is a finite level $m = m^f$ such that $v'(m^f) = 0$

Real balances carried out of a period equal $m_t = q_t M_t$.

Inequality (50.2) is the household's time t budget constraint.

It tells how real balances $q_t M_t$ carried out of period t depend on income, consumption, taxes, and real balances $q_t M_{t-1}$ carried into the period.

Equation (50.3) imposes an exogenous upper bound $\bar{m}$ on the choice of real balances, where $\bar{m} \geq m^f$.

50.2.2 Government

The government chooses a sequence of inverse money growth rates with time t component $h_t \equiv \frac{M_{t-1}}{M_t} \in \Pi \equiv [\underline{\pi}, \overline{\pi}]$, where $0 < \underline{\pi} < 1 < \frac{1}{\beta} \leq \overline{\pi}$.

The government faces a sequence of budget constraints with time t component

$$-x_t = q_t (M_t - M_{t-1})$$

which, by using the definitions of m_t and h_t, can also be expressed as

$$-x_t = m_t (1 - h_t) \tag{50.4}$$

The restrictions $m_t \in [0, \bar{m}]$ and $h_t \in \Pi$ evidently imply that $x_t \in X \equiv [(\underline{\pi} - 1)\bar{m}, (\overline{\pi} - 1)\bar{m}]$.

We define the set $E \equiv [0, \bar{m}] \times \Pi \times X$, so that we require that $(m, h, x) \in E$.

To represent the idea that taxes are distorting, Chang makes the following assumption about outcomes for per capita output:

$$y_t = f(x_t) \tag{50.5}$$

where $f : \mathbb{R} \to \mathbb{R}$ satisfies $f(x) > 0$, $f(x)$ is twice continuously differentiable, $f''(x) < 0$, $f'(0) = 0$, and $f(x) = f(-x)$ for all $x \in \mathbb{R}$, so that subsidies and taxes are equally distorting.

The purpose is not to model the causes of tax distortions in any detail but simply to summarize the *outcome* of those distortions via the function $f(x)$.

A key part of the specification is that tax distortions are increasing in the absolute value of tax revenues.

The government chooses a competitive equilibrium that maximizes (50.1).

50.2.3 Within-period Timing Protocol

For the results in this lecture, the *timing* of actions within a period is important because of the incentives that it activates.

Chang assumed the following within-period timing of decisions:

- first, the government chooses h_t and x_t;
- then given $\vec{q}$ and its expectations about future values of x and y's, the household chooses M_t and therefore m_t because $m_t = q_t M_t$;
- then output $y_t = f(x_t)$ is realized;
- finally $c_t = y_t$

This within-period timing confronts the government with choices framed by how the private sector wants to respond when the government takes time t actions that differ from what the private sector had expected.

This timing will shape the incentives confronting the government at each history that are to be incorporated in the construction of the $\tilde{D}$ operator below.

50.2.4 Household's Problem

Given M_{-1} and $\{q_t\}_{t=0}^{\infty}$, the household's problem is

$$\begin{aligned}\mathcal{L} = \max_{\vec{c},\vec{M}} \min_{\vec{\lambda},\vec{\mu}} \sum_{t=0}^{\infty} \beta^t \big\{ & u(c_t) + v(M_t q_t) + \lambda_t [y_t - c_t - x_t + q_t M_{t-1} - q_t M_t] \\ & + \mu_t [\bar{m} - q_t M_t] \big\}\end{aligned}$$

First-order conditions with respect to c_t and M_t, respectively, are

$$\begin{aligned} u'(c_t) &= \lambda_t \\ q_t [u'(c_t) - v'(M_t q_t)] &\leq \beta u'(c_{t+1}) q_{t+1}, \quad = \text{ if } M_t q_t < \bar{m} \end{aligned}$$

Using $h_t = \frac{M_{t-1}}{M_t}$ and $q_t = \frac{m_t}{M_t}$ in these first-order conditions and rearranging implies

$$m_t [u'(c_t) - v'(m_t)] \leq \beta u'(f(x_{t+1})) m_{t+1} h_{t+1}, \quad = \text{ if } m_t < \bar{m} \tag{50.6}$$

Define the following key variable

$$\theta_{t+1} \equiv u'(f(x_{t+1})) m_{t+1} h_{t+1} \tag{50.7}$$

This is real money balances at time $t+1$ measured in units of marginal utility, which Chang refers to as 'the marginal utility of real balances'.

From the standpoint of the household at time t, equation (50.7) shows that θ_{t+1} intermediates the influences of $(\vec{x}_{t+1}, \vec{m}_{t+1})$ on the household's choice of real balances m_t.

By "intermediates" we mean that the future paths $(\vec{x}_{t+1}, \vec{m}_{t+1})$ influence m_t entirely through their effects on the scalar θ_{t+1}.

The observation that the one dimensional promised marginal utility of real balances θ_{t+1} functions in this way is an important step in constructing a class of competitive equilibria that have a recursive representation.

A closely related observation pervaded the analysis of Stackelberg plans in *dynamic Stackelberg problems* and *the Calvo model*.

50.2.5 Competitive Equilibrium

Definition:

- A *government policy* is a pair of sequences $(\vec{h}, \vec{x})$ where $h_t \in \Pi \ \forall t \geq 0$.
- A *price system* is a non-negative value of money sequence $\vec{q}$.
- An *allocation* is a triple of non-negative sequences $(\vec{c}, \vec{m}, \vec{y})$.

It is required that time t components $(m_t, x_t, h_t) \in E$.

Definition:

Given M_{-1}, a government policy $(\vec{h}, \vec{x})$, price system $\vec{q}$, and allocation $(\vec{c}, \vec{m}, \vec{y})$ are said to be a *competitive equilibrium* if

- $m_t = q_t M_t$ and $y_t = f(x_t)$.
- The government budget constraint is satisfied.
- Given $\vec{q}, \vec{x}, \vec{y}$, $(\vec{c}, \vec{m})$ solves the household's problem.

50.2.6 A Credible Government Policy

Chang works with

A credible government policy with a recursive representation

- Here there is no time 0 Ramsey planner.
- Instead there is a sequence of governments, one for each t, that choose time t government actions after forecasting what future governments will do.
- Let $w = \sum_{t=0}^{\infty} \beta^t \left[u(c_t) + v(q_t M_t)\right]$ be a value associated with a particular competitive equilibrium.
- A recursive representation of a credible government policy is a pair of initial conditions (w_0, θ_0) and a five-tuple of functions

$$h(w_t, \theta_t), m(h_t, w_t, \theta_t), x(h_t, w_t, \theta_t), \chi(h_t, w_t, \theta_t), \Psi(h_t, w_t, \theta_t)$$

  mapping w_t, θ_t and in some cases h_t into $\hat{h}_t, m_t, x_t, w_{t+1}$, and θ_{t+1}, respectively.
- Starting from an initial condition (w_0, θ_0), a credible government policy can be constructed by iterating on these functions in the following order that respects the within-period timing:

$$\begin{aligned} \hat{h}_t &= h(w_t, \theta_t) \\ m_t &= m(h_t, w_t, \theta_t) \\ x_t &= x(h_t, w_t, \theta_t) \\ w_{t+1} &= \chi(h_t, w_t, \theta_t) \\ \theta_{t+1} &= \Psi(h_t, w_t, \theta_t) \end{aligned} \tag{50.8}$$

- Here it is to be understood that $\hat{h}_t$ is the action that the government policy instructs the government to take, while h_t possibly not equal to $\hat{h}_t$ is some other action that the government is free to take at time t.

The plan is *credible* if it is in the time t government's interest to execute it.

Credibility requires that the plan be such that for all possible choices of h_t that are consistent with competitive equilibria,

$$\begin{aligned} & u(f(x(\hat{h}_t, w_t, \theta_t))) + v(m(\hat{h}_t, w_t, \theta_t)) + \beta\chi(\hat{h}_t, w_t, \theta_t) \\ & \geq u(f(x(h_t, w_t, \theta_t))) + v(m(h_t, w_t, \theta_t)) + \beta\chi(h_t, w_t, \theta_t) \end{aligned}$$

so that at each instance and circumstance of choice, a government attains a weakly higher lifetime utility with continuation value $w_{t+1} = \Psi(h_t, w_t, \theta_t)$ by adhering to the plan and confirming the associated time t action $\hat{h}_t$ that the public had expected earlier.

Please note the subtle change in arguments of the functions used to represent a competitive equilibrium and a Ramsey plan, on the one hand, and a credible government plan, on the other hand.

The extra arguments appearing in the functions used to represent a credible plan come from allowing the government to contemplate disappointing the private sector's expectation about its time t choice $\hat{h}_t$.

A credible plan induces the government to confirm the private sector's expectation.

The recursive representation of the plan uses the evolution of continuation values to deter the government from wanting to disappoint the private sector's expectations.

Technically, a Ramsey plan and a credible plan both incorporate history dependence.

For a Ramsey plan, this is encoded in the dynamics of the state variable θ_t, a promised marginal utility that the Ramsey plan delivers to the private sector.

For a credible government plan, we the two-dimensional state vector (w_t, θ_t) encodes history dependence.

50.2.7 Sustainable Plans

A government strategy σ and an allocation rule α are said to constitute a *sustainable plan* (SP) if.

1. σ is admissible.
2. Given σ, α is competitive.
3. After any history $\vec{h}^{t-1}$, the continuation of σ is optimal for the government; i.e., the sequence $\vec{h}_t$ induced by σ after $\vec{h}^{t-1}$ maximizes over CE_π given α.

Given any history $\vec{h}^{t-1}$, the continuation of a sustainable plan is a sustainable plan.

Let $\Theta = \{(\vec{m}, \vec{x}, \vec{h}) \in CE : \text{there is an SP whose outcome is} (\vec{m}, \vec{x}, \vec{h})\}$.

Sustainable outcomes are elements of Θ.

Now consider the space

$$S = \Big\{(w, \theta) : \text{there is a sustainable outcome } (\vec{m}, \vec{x}, \vec{h}) \in \Theta$$

with value

$$w = \sum_{t=0}^{\infty} \beta^t [u(f(x_t)) + v(m_t)] \text{ and such that } u'(f(x_0))(m_0 + x_0) = \theta\Big\}$$

The space S is a compact subset of $W \times \Omega$ where $W = [\underline{w}, \overline{w}]$ is the space of values associated with sustainable plans. Here $\underline{w}$ and $\overline{w}$ are finite bounds on the set of values.

Because there is at least one sustainable plan, S is nonempty.

Now recall the within-period timing protocol, which we can depict $(h, x) \to m = qM \to y = c$.

With this timing protocol in mind, the time 0 component of an SP has the following components:

1. A period 0 action $\hat{h} \in \Pi$ that the public expects the government to take, together with subsequent within-period consequences $m(\hat{h}), x(\hat{h})$ when the government acts as expected.
2. For any first-period action $h \neq \hat{h}$ with $h \in CE^0_\pi$, a pair of within-period consequences $m(h), x(h)$ when the government does not act as the public had expected.
3. For every $h \in \Pi$, a pair $(w'(h), \theta'(h)) \in S$ to carry into next period.

These components must be such that it is optimal for the government to choose $\hat{h}$ as expected; and for every possible $h \in \Pi$, the government budget constraint and the household's Euler equation must hold with continuation θ being $\theta'(h)$.

Given the timing protocol within the model, the representative household's response to a government deviation to $h \neq \hat{h}$ from a prescribed $\hat{h}$ consists of a first-period action $m(h)$ and associated subsequent actions, together with future equilibrium prices, captured by $(w'(h), \theta'(h))$.

At this point, Chang introduces an idea in the spirit of Abreu, Pearce, and Stacchetti [Abreu *et al.*, 1990].

Let Z be a nonempty subset of $W \times \Omega$.

Think of using pairs (w', θ') drawn from Z as candidate continuation value, promised marginal utility pairs.

Define the following operator:

$$\tilde{D}(Z) = \Big\{(w, \theta) : \text{there is } \hat{h} \in CE^0_\pi \text{ and for each } h \in CE^0_\pi \\ \text{a four-tuple } (m(h), x(h), w'(h), \theta'(h)) \in [0, \bar{m}] \times X \times Z \tag{50.9}$$

such that

$$w = u(f(x(\hat{h}))) + v(m(\hat{h})) + \beta w'(\hat{h}) \tag{50.10}$$

$$\theta = u'(f(x(\hat{h})))(m(\hat{h}) + x(\hat{h})) \tag{50.11}$$

and for all $h \in CE^0_\pi$

$$w \geq u(f(x(h))) + v(m(h)) + \beta w'(h) \tag{50.12}$$

$$x(h) = m(h)(h - 1) \tag{50.13}$$

and

$$m(h)(u'(f(x(h))) - v'(m(h))) \leq \beta \theta'(h) \tag{50.14}$$

$$\text{with equality if } m(h) < \bar{m} \Big\}$$

This operator adds the key incentive constraint to the conditions that had defined the earlier $D(Z)$ operator defined in *competitive equilibria in the Chang model*.

Condition (50.12) requires that the plan deter the government from wanting to take one-shot deviations when candidate continuation values are drawn from Z.

Proposition:

1. If $Z \subset \tilde{D}(Z)$, then $\tilde{D}(Z) \subset S$ ('self-generation').
2. $S = \tilde{D}(S)$ ('factorization').

Proposition:.

1. Monotonicity of $\tilde{D}$: $Z \subset Z'$ implies $\tilde{D}(Z) \subset \tilde{D}(Z')$.

2. Z compact implies that $\tilde{D}(Z)$ is compact.

Chang establishes that S is compact and that therefore there exists a highest value SP and a lowest value SP.

Further, the preceding structure allows Chang to compute S by iterating to convergence on $\tilde{D}$ provided that one begins with a sufficiently large initial set Z_0.

This structure delivers the following recursive representation of a sustainable outcome:

1. choose an initial $(w_0, \theta_0) \in S$;
2. generate a sustainable outcome recursively by iterating on (50.8), which we repeat here for convenience:

$$\begin{aligned} \hat{h}_t &= h(w_t, \theta_t) \\ m_t &= m(h_t, w_t, \theta_t) \\ x_t &= x(h_t, w_t, \theta_t) \\ w_{t+1} &= \chi(h_t, w_t, \theta_t) \\ \theta_{t+1} &= \Psi(h_t, w_t, \theta_t) \end{aligned}$$

50.3 Calculating the Set of Sustainable Promise-Value Pairs

Above we defined the $\tilde{D}(Z)$ operator as (50.9).

Chang (1998) provides a method for dealing with the final three constraints.

These incentive constraints ensure that the government wants to choose $\hat{h}$ as the private sector had expected it to.

Chang's simplification starts from the idea that, when considering whether or not to confirm the private sector's expectation, the government only needs to consider the payoff of the *best* possible deviation.

Equally, to provide incentives to the government, we only need to consider the harshest possible punishment.

Let h denote some possible deviation. Chang defines:

$$P(h; Z) = \min u(f(x)) + v(m) + \beta w'$$

where the minimization is subject to

$$x = m(h - 1)$$

$$m(h)(u'(f(x(h))) + v'(m(h))) \leq \beta\theta'(h) \text{ (with equality if } m(h) < \bar{m})\}$$

$$(m, x, w', \theta') \in [0, \bar{m}] \times X \times Z$$

For a given deviation h, this problem finds the worst possible sustainable value.

We then define:

$$BR(Z) = \max P(h; Z) \text{ subject to } h \in CE^0_\pi$$

$BR(Z)$ is the value of the government's most tempting deviation.

With this in hand, we can define a new operator $E(Z)$ that is equivalent to the $\tilde{D}(Z)$ operator but simpler to implement:

$$E(Z) = \Big\{(w, \theta) : \exists h \in CE^0_\pi \text{ and } (m(h), x(h), w'(h), \theta'(h)) \in [0, \bar{m}] \times X \times Z$$

such that

$$w = u(f(x(h))) + v(m(h)) + \beta w'(h)$$

$$\theta = u'(f(x(h)))(m(h) + x(h))$$

$$x(h) = m(h)(h-1)$$

$$m(h)(u'(f(x(h))) - v'(m(h))) \leq \beta\theta'(h) \text{ (with equality if } m(h) < \bar{m})$$

and

$$w \geq BR(Z)\Big\}$$

Aside from the final incentive constraint, this is the same as the operator in *competitive equilibria in the Chang model*.

Consequently, to implement this operator we just need to add one step to our *outer hyperplane approximation algorithm* :

1. Initialize subgradients, H, and hyperplane levels, C_0.
2. Given a set of subgradients, H, and hyperplane levels, C_t, calculate $BR(S_t)$.
3. Given H, C_t, and $BR(S_t)$, for each subgradient $h_i \in H$:
    - Solve a linear program (described below) for each action in the action space.
    - Find the maximum and update the corresponding hyperplane level, $C_{i,t+1}$.
4. If $|C_{t+1} - C_t| > \epsilon$, return to 2.

Step 1 simply creates a large initial set S_0.

Given some set S_t, **Step 2** then constructs the value $BR(S_t)$.

To do this, we solve the following problem for each point in the action space (m_j, h_j):

$$\min_{[w',\theta']} u(f(x_j)) + v(m_j) + \beta w'$$

subject to

$$H \cdot (w', \theta') \leq C_t$$

$$x_j = m_j(h_j - 1)$$

$$m_j(u'(f(x_j)) - v'(m_j)) \leq \beta\theta' \quad (= \text{if } m_j < \bar{m})$$

This gives us a matrix of possible values, corresponding to each point in the action space.

To find $BR(Z)$, we minimize over the m dimension and maximize over the h dimension.

Step 3 then constructs the set $S_{t+1} = E(S_t)$. The linear program in Step 3 is designed to construct a set S_{t+1} that is as large as possible while satisfying the constraints of the $E(S)$ operator.

To do this, for each subgradient h_i, and for each point in the action space (m_j, h_j), we solve the following problem:

$$\max_{[w',\theta']} h_i \cdot (w, \theta)$$

subject to

$$H \cdot (w', \theta') \leq C_t$$

$$w = u(f(x_j)) + v(m_j) + \beta w'$$

$$\theta = u'(f(x_j))(m_j + x_j)$$

$$x_j = m_j(h_j - 1)$$

$$m_j(u'(f(x_j)) - v'(m_j)) \leq \beta\theta' \quad (= \text{if } m_j < \bar{m})$$

$$w \geq BR(Z)$$

This problem maximizes the hyperplane level for a given set of actions.

The second part of Step 3 then finds the maximum possible hyperplane level across the action space.

The algorithm constructs a sequence of progressively smaller sets $S_{t+1} \subset S_t \subset S_{t-1} \cdots \subset S_0$.

Step 4 ends the algorithm when the difference between these sets is small enough.

We have created a Python class that solves the model assuming the following functional forms:

$$u(c) = \log(c)$$

$$v(m) = \frac{1}{500}(m\bar{m} - 0.5m^2)^{0.5}$$

$$f(x) = 180 - (0.4x)^2$$

The remaining parameters $\{\beta, \bar{m}, \underline{h}, \bar{h}\}$ are then variables to be specified for an instance of the Chang class.

Below we use the class to solve the model and plot the resulting equilibrium set, once with $\beta = 0.3$ and once with $\beta = 0.8$. We also plot the (larger) competitive equilibrium sets, which we described in *competitive equilibria in the Chang model*.

(We have set the number of subgradients to 10 in order to speed up the code for now. We can increase accuracy by increasing the number of subgradients)

The following code computes sustainable plans

```
"""
Provides a class called ChangModel to solve different
parameterizations of the Chang (1998) model.
"""

import numpy as np
import time

from scipy.optimize import linprog, minimize
import numpy.polynomial.chebyshev as cheb


class ChangModel:
    """
    Class to solve for the competitive and sustainable sets in the Chang (1998)
    model, for different parameterizations.
    """

    def __init__(self, β, mbar, h_min, h_max, n_h, n_m, N_g):
        # Record parameters
        self.β, self.mbar, self.h_min, self.h_max = β, mbar, h_min, h_max
        self.n_h, self.n_m, self.N_g = n_h, n_m, N_g

        # Create other parameters
        self.m_min = 1e-9
        self.m_max = self.mbar
        self.N_a = self.n_h*self.n_m

        # Utility and production functions
        uc = lambda c: np.log(np.maximum(c, 1e-10))  # Clip to 1e-10 to avoid log(0)␣
```

(continues on next page)

(continued from previous page)

```
↪or log(-ve)
        uc_p = lambda c: 1/c
        v = lambda m: 1/500 * (mbar * m - 0.5 * m**2)**0.5
        v_p = lambda m: 0.5/500 * (mbar * m - 0.5 * m**2)**(-0.5) * (mbar - m)
        u = lambda h, m: uc(f(h, m)) + v(m)

        def f(h, m):
            x = m * (h - 1)
            f = 180 - (0.4 * x)**2
            return f

        def θ(h, m):
            x = m * (h - 1)
            θ = uc_p(f(h, m)) * (m + x)
            return θ

        # Create set of possible action combinations, A
        A1 = np.linspace(h_min, h_max, n_h).reshape(n_h, 1)
        A2 = np.linspace(self.m_min, self.m_max, n_m).reshape(n_m, 1)
        self.A = np.concatenate((np.kron(np.ones((n_m, 1)), A1),
                                 np.kron(A2, np.ones((n_h, 1)))), axis=1)

        # Pre-compute utility and output vectors
        self.euler_vec = -np.multiply(self.A[:, 1], \
            uc_p(f(self.A[:, 0], self.A[:, 1])) - v_p(self.A[:, 1]))
        self.u_vec = u(self.A[:, 0], self.A[:, 1])
        self.Θ_vec = θ(self.A[:, 0], self.A[:, 1])
        self.f_vec = f(self.A[:, 0], self.A[:, 1])
        self.bell_vec = np.multiply(uc_p(f(self.A[:, 0],
                                    self.A[:, 1])),
                                    np.multiply(self.A[:, 1],
                                    (self.A[:, 0] - 1))) \
                        + np.multiply(self.A[:, 1],
                                      v_p(self.A[:, 1]))

        # Find extrema of (w, θ) space for initial guess of equilibrium sets
        p_vec = np.zeros(self.N_a)
        w_vec = np.zeros(self.N_a)
        for i in range(self.N_a):
            p_vec[i] = self.Θ_vec[i]
            w_vec[i] = self.u_vec[i]/(1 - β)

        w_space = np.array([min(w_vec[~np.isinf(w_vec)]),
                            max(w_vec[~np.isinf(w_vec)])])
        p_space = np.array([0, max(p_vec[~np.isinf(w_vec)])])
        self.p_space = p_space

        # Set up hyperplane levels and gradients for iterations
        def SG_H_V(N, w_space, p_space):
            """
            This function  initializes the subgradients, hyperplane levels,
            and extreme points of the value set by choosing an appropriate
            origin and radius. It is based on a similar function in QuantEcon's
            Games.jl
            """
```

(continues on next page)

(continued from previous page)

```
            # First, create a unit circle. Want points placed on [0, 2π]
            inc = 2 * np.pi / N
            degrees = np.arange(0, 2 * np.pi, inc)

            # Points on circle
            H = np.zeros((N, 2))
            for i in range(N):
                x = degrees[i]
                H[i, 0] = np.cos(x)
                H[i, 1] = np.sin(x)

            # Then calculate origin and radius
            o = np.array([np.mean(w_space), np.mean(p_space)])
            r1 = max((max(w_space) - o[0])**2, (o[0] - min(w_space))**2)
            r2 = max((max(p_space) - o[1])**2, (o[1] - min(p_space))**2)
            r = np.sqrt(r1 + r2)

            # Now calculate vertices
            Z = np.zeros((2, N))
            for i in range(N):
                Z[0, i] = o[0] + r*H.T[0, i]
                Z[1, i] = o[1] + r*H.T[1, i]

            # Corresponding hyperplane levels
            C = np.zeros(N)
            for i in range(N):
                C[i] = np.dot(Z[:, i], H[i, :])

            return C, H, Z

        C, self.H, Z = SG_H_V(N_g, w_space, p_space)
        C = C.reshape(N_g, 1)
        self.c0_c, self.c0_s, self.c1_c, self.c1_s = np.copy(C), np.copy(C), \
            np.copy(C), np.copy(C)
        self.z0_s, self.z0_c, self.z1_s, self.z1_c = np.copy(Z), np.copy(Z), \
            np.copy(Z), np.copy(Z)

        self.w_bnds_s, self.w_bnds_c = (w_space[0], w_space[1]), \
            (w_space[0], w_space[1])
        self.p_bnds_s, self.p_bnds_c = (p_space[0], p_space[1]), \
            (p_space[0], p_space[1])

        # Create dictionaries to save equilibrium set for each iteration
        self.c_dic_s, self.c_dic_c = {}, {}
        self.c_dic_s[0], self.c_dic_c[0] = self.c0_s, self.c0_c

    def solve_worst_spe(self):
        """
        Method to solve for BR(Z). See p.449 of Chang (1998)
        """

        p_vec = np.full(self.N_a, np.nan)
        c = [1, 0]

        # Pre-compute constraints
        aineq_mbar = np.vstack((self.H, np.array([0, -self.β])))
```

(continues on next page)

(continued from previous page)

```
        bineq_mbar = np.vstack((self.c0_s, 0))

        aineq = self.H
        bineq = self.c0_s
        aeq = [[0, -self.β]]

        for j in range(self.N_a):
            # Only try if consumption is possible
            if self.f_vec[j] > 0:
                # If m = mbar, use inequality constraint
                if self.A[j, 1] == self.mbar:
                    bineq_mbar[-1] = self.euler_vec[j]
                    res = linprog(c, A_ub=aineq_mbar, b_ub=bineq_mbar,
                                  bounds=(self.w_bnds_s, self.p_bnds_s))
                else:
                    beq = self.euler_vec[j]
                    res = linprog(c, A_ub=aineq, b_ub=bineq, A_eq=aeq, b_eq=beq,
                                  bounds=(self.w_bnds_s, self.p_bnds_s))
                if res.status == 0:
                    p_vec[j] = self.u_vec[j] + self.β * res.x[0]

        # Max over h and min over other variables (see Chang (1998) p.449)
        self.br_z = np.nanmax(np.nanmin(p_vec.reshape(self.n_m, self.n_h), 0))

    def solve_subgradient(self):
        """
        Method to solve for E(Z). See p.449 of Chang (1998)
        """

        # Pre-compute constraints
        aineq_C_mbar = np.vstack((self.H, np.array([0, -self.β])))
        bineq_C_mbar = np.vstack((self.c0_c, 0))

        aineq_C = self.H
        bineq_C = self.c0_c
        aeq_C = [[0, -self.β]]

        aineq_S_mbar = np.vstack((np.vstack((self.H, np.array([0, -self.β]))),
                                  np.array([-self.β, 0])))
        bineq_S_mbar = np.vstack((self.c0_s, np.zeros((2, 1))))

        aineq_S = np.vstack((self.H, np.array([-self.β, 0])))
        bineq_S = np.vstack((self.c0_s, 0))
        aeq_S = [[0, -self.β]]

        # Update maximal hyperplane level
        for i in range(self.N_g):
            c_a1a2_c, t_a1a2_c = np.full(self.N_a, -np.inf), \
                np.zeros((self.N_a, 2))
            c_a1a2_s, t_a1a2_s = np.full(self.N_a, -np.inf), \
                np.zeros((self.N_a, 2))

            c = [-self.H[i, 0], -self.H[i, 1]]

            for j in range(self.N_a):
                # Only try if consumption is possible
```

(continues on next page)

(continued from previous page)

```
                if self.f_vec[j] > 0:

                    # COMPETITIVE EQUILIBRIA
                    # If m = mbar, use inequality constraint
                    if self.A[j, 1] == self.mbar:
                        bineq_C_mbar[-1] = self.euler_vec[j]
                        res = linprog(c, A_ub=aineq_C_mbar, b_ub=bineq_C_mbar,
                                      bounds=(self.w_bnds_c, self.p_bnds_c))
                    # If m < mbar, use equality constraint
                    else:
                        beq_C = self.euler_vec[j]
                        res = linprog(c, A_ub=aineq_C, b_ub=bineq_C, A_eq = aeq_C,
                                      b_eq = beq_C, bounds=(self.w_bnds_c, \
                                          self.p_bnds_c))
                    if res.status == 0:
                        c_a1a2_c[j] = self.H[i, 0] * (self.u_vec[j] \
                            + self.β * res.x[0]) + self.H[i, 1] * self.Θ_vec[j]
                        t_a1a2_c[j] = res.x

                    # SUSTAINABLE EQUILIBRIA
                    # If m = mbar, use inequality constraint
                    if self.A[j, 1] == self.mbar:
                        bineq_S_mbar[-2] = self.euler_vec[j]
                        bineq_S_mbar[-1] = self.u_vec[j] - self.br_z
                        res = linprog(c, A_ub=aineq_S_mbar, b_ub=bineq_S_mbar,
                                      bounds=(self.w_bnds_s, self.p_bnds_s))
                    # If m < mbar, use equality constraint
                    else:
                        bineq_S[-1] = self.u_vec[j] - self.br_z
                        beq_S = self.euler_vec[j]
                        res = linprog(c, A_ub=aineq_S, b_ub=bineq_S, A_eq = aeq_S,
                                      b_eq = beq_S, bounds=(self.w_bnds_s, \
                                          self.p_bnds_s))
                    if res.status == 0:
                        c_a1a2_s[j] = self.H[i, 0] * (self.u_vec[j] \
                            + self.β*res.x[0]) + self.H[i, 1] * self.Θ_vec[j]
                        t_a1a2_s[j] = res.x

            idx_c = np.where(c_a1a2_c == max(c_a1a2_c))[0][0]
            self.z1_c[:, i] = np.array([self.u_vec[idx_c]
                                        + self.β * t_a1a2_c[idx_c, 0],
                                        self.Θ_vec[idx_c]])

            idx_s = np.where(c_a1a2_s == max(c_a1a2_s))[0][0]
            self.z1_s[:, i] = np.array([self.u_vec[idx_s]
                                        + self.β * t_a1a2_s[idx_s, 0],
                                        self.Θ_vec[idx_s]])

        for i in range(self.N_g):
            self.c1_c[i] = np.dot(self.z1_c[:, i], self.H[i, :])
            self.c1_s[i] = np.dot(self.z1_s[:, i], self.H[i, :])

    def solve_sustainable(self, tol=1e-5, max_iter=250):
        """
        Method to solve for the competitive and sustainable equilibrium sets.
        """
```

(continues on next page)

(continued from previous page)

```
        t = time.time()
        diff = tol + 1
        iters = 0

        print('### --------------- ###')
        print('Solving Chang Model Using Outer Hyperplane Approximation')
        print('### --------------- ### \n')

        print('Maximum difference when updating hyperplane levels:')

        while diff > tol and iters < max_iter:
            iters = iters + 1
            self.solve_worst_spe()
            self.solve_subgradient()
            diff = max(np.maximum(abs(self.c0_c - self.c1_c),
                       abs(self.c0_s - self.c1_s)))
            print(diff)

            # Update hyperplane levels
            self.c0_c, self.c0_s = np.copy(self.c1_c), np.copy(self.c1_s)

            # Update bounds for w and θ
            wmin_c, wmax_c = np.min(self.z1_c, axis=1)[0], \
                np.max(self.z1_c, axis=1)[0]
            pmin_c, pmax_c = np.min(self.z1_c, axis=1)[1], \
                np.max(self.z1_c, axis=1)[1]

            wmin_s, wmax_s = np.min(self.z1_s, axis=1)[0], \
                np.max(self.z1_s, axis=1)[0]
            pmin_S, pmax_S = np.min(self.z1_s, axis=1)[1], \
                np.max(self.z1_s, axis=1)[1]

            self.w_bnds_s, self.w_bnds_c = (wmin_s, wmax_s), (wmin_c, wmax_c)
            self.p_bnds_s, self.p_bnds_c = (pmin_S, pmax_S), (pmin_c, pmax_c)

            # Save iteration
            self.c_dic_c[iters], self.c_dic_s[iters] = np.copy(self.c1_c), \
                np.copy(self.c1_s)
            self.iters = iters

        elapsed = time.time() - t
        print('Convergence achieved after {} iterations and {} \
            seconds'.format(iters, round(elapsed, 2)))

    def solve_bellman(self, θ_min, θ_max, order, disp=False, tol=1e-7, maxiters=100):
        """
        Continuous Method to solve the Bellman equation in section 25.3
        """
        mbar = self.mbar

        # Utility and production functions
        uc = lambda c: np.log(np.maximum(c, 1e-10))  # Clip to 1e-10 to avoid log(0)␣
↪or log(-ve)
        uc_p = lambda c: 1 / c
        v = lambda m: 1 / 500 * (mbar * m - 0.5 * m**2)**0.5
```

(continues on next page)

(continued from previous page)

```
        v_p = lambda m: 0.5/500 * (mbar*m - 0.5 * m**2)**(-0.5) * (mbar - m)
        u = lambda h, m: uc(f(h, m)) + v(m)

        def f(h, m):
            x = m * (h - 1)
            f = 180 - (0.4 * x)**2
            return f

        def θ(h, m):
            x = m * (h - 1)
            θ = uc_p(f(h, m)) * (m + x)
            return θ

        # Bounds for Maximization
        lb1 = np.array([self.h_min, 0, θ_min])
        ub1 = np.array([self.h_max, self.mbar - 1e-5, θ_max])
        lb2 = np.array([self.h_min, θ_min])
        ub2 = np.array([self.h_max, θ_max])

        # Initialize Value Function coefficients
        # Calculate roots of Chebyshev polynomial
        k = np.linspace(order, 1, order)
        roots = np.cos((2 * k - 1) * np.pi / (2 * order))
        # Scale to approximation space
        s = θ_min + (roots - -1) / 2 * (θ_max - θ_min)
        # Create a basis matrix
        Φ = cheb.chebvander(roots, order - 1)
        c = np.zeros(Φ.shape[0])

        # Function to minimize and constraints
        def p_fun(x):
            scale = -1 + 2 * (x[2] - θ_min)/(θ_max - θ_min)
            p_fun = - (u(x[0], x[1]) \
                + self.β * np.dot(cheb.chebvander(scale, order - 1), c))
            return p_fun.item()

        def p_fun2(x):
            scale = -1 + 2*(x[1] - θ_min)/(θ_max - θ_min)
            p_fun = - (u(x[0],mbar) \
                + self.β * np.dot(cheb.chebvander(scale, order - 1), c))
            return p_fun.item()

        cons1 = ({'type': 'eq',   'fun': lambda x: uc_p(f(x[0], x[1])) * x[1]
                    * (x[0] - 1) + v_p(x[1]) * x[1] + self.β * x[2] - θ},
                 {'type': 'eq',   'fun': lambda x: uc_p(f(x[0], x[1]))
                    * x[0] * x[1] - θ})
        cons2 = ({'type': 'ineq', 'fun': lambda x: uc_p(f(x[0], mbar)) * mbar
                    * (x[0] - 1) + v_p(mbar) * mbar + self.β * x[1] - θ},
                 {'type': 'eq',   'fun': lambda x: uc_p(f(x[0], mbar))
                    * x[0] * mbar - θ})

        bnds1 = np.concatenate([lb1.reshape(3, 1), ub1.reshape(3, 1)], axis=1)
        bnds2 = np.concatenate([lb2.reshape(2, 1), ub2.reshape(2, 1)], axis=1)

        # Bellman Iterations
        diff = 1
```

(continues on next page)

(continued from previous page)

```
        iters = 1

        while diff > tol:
        # 1. Maximization, given value function guess
            p_iter1 = np.zeros(order)
            for i in range(order):
                θ = s[i]
                x0 = np.clip(lb1 + (ub1-lb1)/2, lb1, ub1)
                res = minimize(p_fun,
                               x0,
                               method='SLSQP',
                               bounds=bnds1,
                               constraints=cons1,
                               tol=1e-10)
                if res.success == True:
                    p_iter1[i] = -p_fun(res.x)
                x0 = np.clip(lb2 + (ub2-lb2)/2, lb2, ub2)
                res = minimize(p_fun2,
                               x0,
                               method='SLSQP',
                               bounds=bnds2,
                               constraints=cons2,
                               tol=1e-10)
                if -p_fun2(res.x) > p_iter1[i] and res.success == True:
                    p_iter1[i] = -p_fun2(res.x)

            # 2. Bellman updating of Value Function coefficients
            c1 = np.linalg.solve(Φ, p_iter1)
            # 3. Compute distance and update
            diff = np.linalg.norm(c - c1)
            if bool(disp == True):
                print(diff)
            c = np.copy(c1)
            iters = iters + 1
            if iters > maxiters:
                print('Convergence failed after {} iterations'.format(maxiters))
                break

        self.θ_grid = s
        self.p_iter = p_iter1
        self.Φ = Φ
        self.c = c
        print('Convergence achieved after {} iterations'.format(iters))

        # Check residuals
        θ_grid_fine = np.linspace(θ_min, θ_max, 100)
        resid_grid = np.zeros(100)
        p_grid = np.zeros(100)
        θ_prime_grid = np.zeros(100)
        m_grid = np.zeros(100)
        h_grid = np.zeros(100)
        for i in range(100):
            θ = θ_grid_fine[i]
            x0 = np.clip(lb1 + (ub1-lb1)/2, lb1, ub1)
            res = minimize(p_fun,
                           x0,
```

(continues on next page)

(continued from previous page)

```
                        method='SLSQP',
                        bounds=bnds1,
                        constraints=cons1,
                        tol=1e-10)
            if res.success == True:
                p = -p_fun(res.x)
                p_grid[i] = p
                θ_prime_grid[i] = res.x[2]
                h_grid[i] = res.x[0]
                m_grid[i] = res.x[1]
            x0 = np.clip(lb2 + (ub2-lb2)/2, lb2, ub2)
            res = minimize(p_fun2,
                        x0,
                        method='SLSQP',
                        bounds=bnds2,
                        constraints=cons2,
                        tol=1e-10)
            if -p_fun2(res.x) > p and res.success == True:
                p = -p_fun2(res.x)
                p_grid[i] = p
                θ_prime_grid[i] = res.x[1]
                h_grid[i] = res.x[0]
                m_grid[i] = self.mbar
            scale = -1 + 2 * (θ - θ_min)/(θ_max - θ_min)
            resid_grid_val = np.dot(cheb.chebvander(scale, order-1), c) - p
            resid_grid[i] = resid_grid_val.item()

        self.resid_grid = resid_grid
        self.θ_grid_fine = θ_grid_fine
        self.θ_prime_grid = θ_prime_grid
        self.m_grid = m_grid
        self.h_grid = h_grid
        self.p_grid = p_grid
        self.x_grid = m_grid * (h_grid - 1)

        # Simulate
        θ_series = np.zeros(31)
        m_series = np.zeros(30)
        h_series = np.zeros(30)

        # Find initial θ
        def ValFun(x):
            scale = -1 + 2*(x - θ_min)/(θ_max - θ_min)
            p_fun = np.dot(cheb.chebvander(scale, order - 1), c)
            return -p_fun

        res = minimize(ValFun,
                       (θ_min + θ_max)/2,
                       bounds=[(θ_min, θ_max)])
        θ_series[0] = res.x.item()

        # Simulate
        for i in range(30):
            θ = θ_series[i]
            x0 = np.clip(lb1 + (ub1-lb1)/2, lb1, ub1)
            res = minimize(p_fun,
```

(continues on next page)

(continued from previous page)

```
                        x0,
                        method='SLSQP',
                        bounds=bnds1,
                        constraints=cons1,
                        tol=1e-10)
            if res.success == True:
                p = -p_fun(res.x)
                h_series[i] = res.x[0]
                m_series[i] = res.x[1]
                θ_series[i+1] = res.x[2]
            x0 = np.clip(lb2 + (ub2-lb2)/2, lb2, ub2)
            res2 = minimize(p_fun2,
                            x0,
                            method='SLSQP',
                            bounds=bnds2,
                            constraints=cons2,
                            tol=1e-10)
            if -p_fun2(res2.x) > p and res2.success == True:
                h_series[i] = res2.x[0]
                m_series[i] = self.mbar
                θ_series[i+1] = res2.x[1]

        self.θ_series = θ_series
        self.m_series = m_series
        self.h_series = h_series
        self.x_series = m_series * (h_series - 1)
```

50.3.1 Comparison of Sets

The set of (w, θ) associated with sustainable plans is smaller than the set of (w, θ) pairs associated with competitive equilibria, since the additional constraints associated with sustainability must also be satisfied.

Let's compute two examples, one with a low β, another with a higher β

```
ch1 = ChangModel(β=0.3, mbar=30, h_min=0.9, h_max=2, n_h=8, n_m=35, N_g=10)
```

```
ch1.solve_sustainable()
```

```
### ---------------- ###
Solving Chang Model Using Outer Hyperplane Approximation
### ---------------- ###

Maximum difference when updating hyperplane levels:
```

```
[1.9168]
```

```
[0.66782]
```

```
[0.49235]
```

```
[0.32412]
```

```
[0.19022]
```

```
[0.10863]
```

```
[0.05817]
```

```
[0.0262]
```

```
[0.01836]
```

```
[0.01415]
```

```
[0.00297]
```

```
[0.00089]
```

```
[0.00027]
```

```
[0.00008]
```

```
[0.00002]
```

```
[0.00001]
Convergence achieved after 16 iterations and 39.36            seconds
```

The following plot shows both the set of w, θ pairs associated with competitive equilibria (in red) and the smaller set of w, θ pairs associated with sustainable plans (in blue).

```
def plot_equilibria(ChangModel):
    """
    Method to plot both equilibrium sets
    """
    fig, ax = plt.subplots(figsize=(7, 5))

    ax.set_xlabel('w', fontsize=16)
    ax.set_ylabel(r"$\theta$", fontsize=18)

    poly_S = polytope.Polytope(ChangModel.H, ChangModel.c1_s)
    poly_C = polytope.Polytope(ChangModel.H, ChangModel.c1_c)
    ext_C = polytope.extreme(poly_C)
    ext_S = polytope.extreme(poly_S)

    ax.fill(ext_C[:, 0], ext_C[:, 1], 'r', zorder=-1)
    ax.fill(ext_S[:, 0], ext_S[:, 1], 'b', zorder=0)
```

(continues on next page)

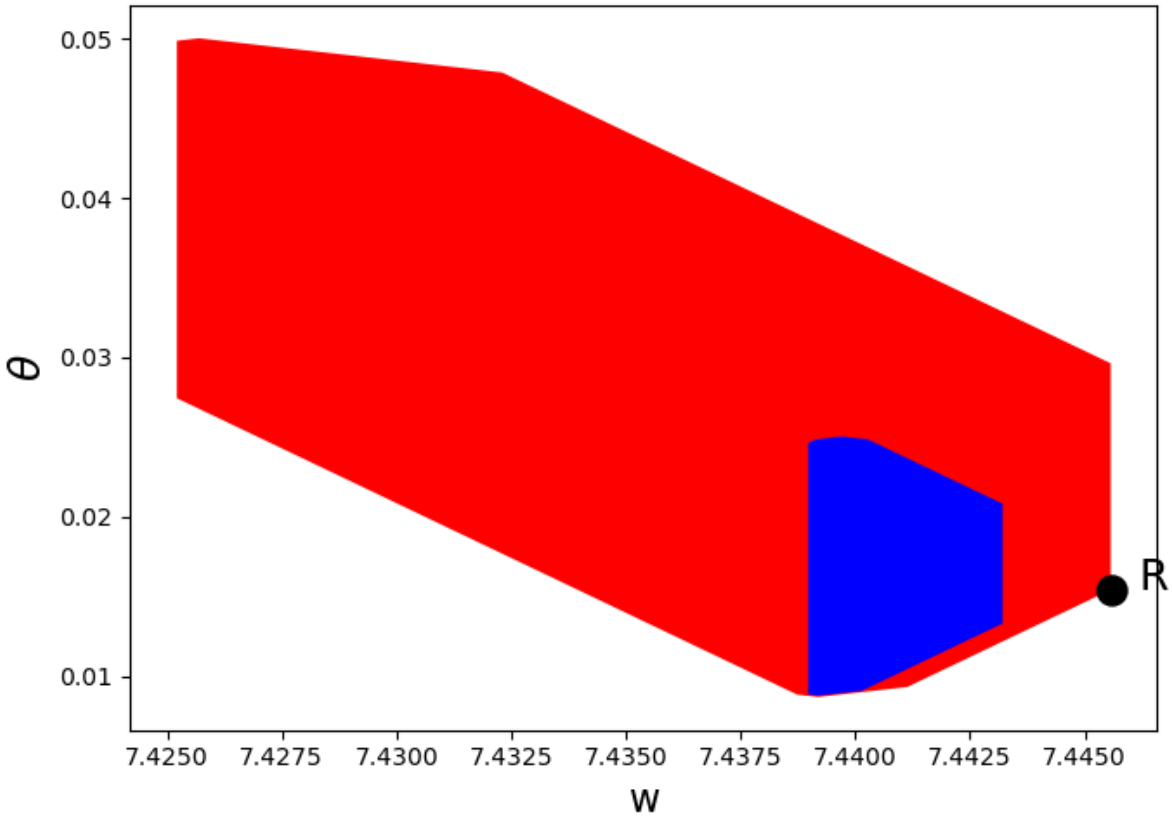
(continued from previous page)

```
    # Add point showing Ramsey Plan
    idx_Ramsey = np.where(ext_C[:, 0] == max(ext_C[:, 0]))[0][0]
    R = ext_C[idx_Ramsey, :]
    ax.scatter(R[0], R[1], 150, 'black', 'o', zorder=1)
    w_min = min(ext_C[:, 0])

    # Label Ramsey Plan slightly to the right of the point
    ax.annotate("R", xy=(R[0], R[1]),
                xytext=(R[0] + 0.03 * (R[0] - w_min),
                R[1]), fontsize=18)

    plt.tight_layout()
    plt.show()

plot_equilibria(ch1)
```



Evidently, the Ramsey plan, denoted by the R, is not sustainable.

Let's raise the discount factor and recompute the sets

```
ch2 = ChangModel(β=0.8, mbar=30, h_min=0.9, h_max=1/0.8,
    n_h=8, n_m=35, N_g=10)
```

```
ch2.solve_sustainable()
```

```
### ---------------- ###
Solving Chang Model Using Outer Hyperplane Approximation
### ---------------- ###

Maximum difference when updating hyperplane levels:
```

```
[0.06369]
```

```
[0.02476]
```

```
[0.02153]
```

```
[0.01915]
```

```
[0.01795]
```

```
[0.01642]
```

```
[0.01507]
```

```
[0.01284]
```

```
[0.01106]
```

```
[0.00694]
```

```
[0.0085]
```

```
[0.00781]
```

```
[0.00433]
```

```
[0.00492]
```

```
[0.00303]
```

```
[0.00182]
```

```
[0.00638]
```

```
[0.00116]
```

```
[0.00093]
```

```
[0.00075]
```

```
[0.0006]
```

```
[0.00494]
```

```
[0.00038]
```

```
[0.00121]
```

```
[0.00024]
```

```
[0.0002]
```

```
[0.00016]
```

```
[0.00013]
```

```
[0.0001]
```

```
[0.00008]
```

```
[0.00006]
```

```
[0.00005]
```

```
[0.00004]
```

```
[0.00003]
```

```
[0.00003]
```

```
[0.00002]
```

```
[0.00002]
```

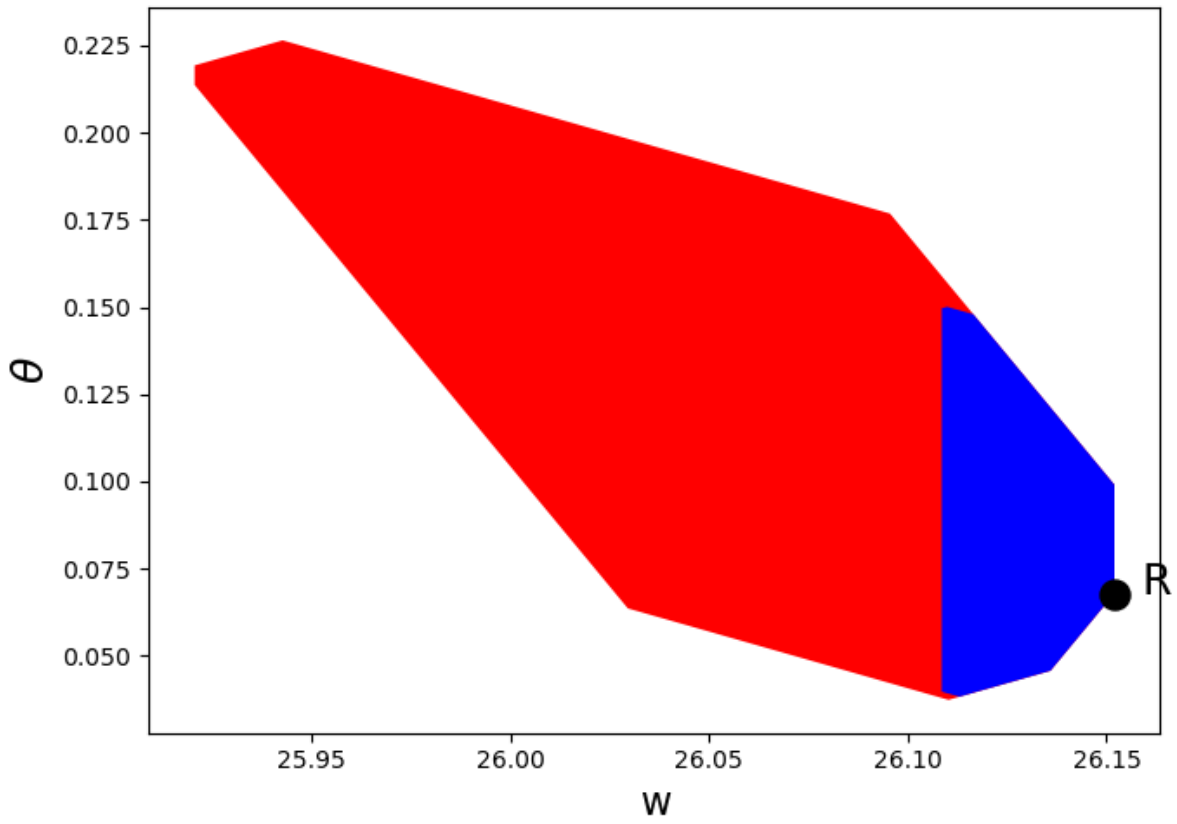
```
[0.00001]
```

```
[0.00001]
```

```
[0.00001]
Convergence achieved after 40 iterations and 115.85              seconds
```

Let's plot both sets

```
plot_equilibria(ch2)
```



Evidently, the Ramsey plan is now sustainable.

Part IX

Other

CHAPTER

FIFTYONE

TROUBLESHOOTING

This page is for readers experiencing errors when running the code from the lectures.

51.1 Fixing Your Local Environment

The basic assumption of the lectures is that code in a lecture should execute whenever

1. it is executed in a Jupyter notebook and
2. the notebook is running on a machine with the latest version of Anaconda Python.

You have installed Anaconda, haven't you, following the instructions in this lecture?

Assuming that you have, the most common source of problems for our readers is that their Anaconda distribution is not up to date.

Here's a useful article on how to update Anaconda.

Another option is to simply remove Anaconda and reinstall.

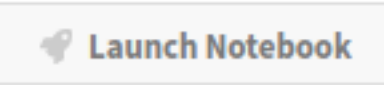
You also need to keep the external code libraries, such as QuantEcon.py up to date.

For this task you can either

- use pip install –upgrade quantecon on the command line, or
- execute !pip install –upgrade quantecon within a Jupyter notebook.

If your local environment is still not working you can do two things.

First, you can use a remote machine instead, by clicking on the Launch Notebook icon available for each lecture



Second, you can report an issue, so we can try to fix your local set up.

We like getting feedback on the lectures so please don't hesitate to get in touch.

51.2 Reporting an Issue

One way to give feedback is to raise an issue through our issue tracker.

Please be as specific as possible. Tell us where the problem is and as much detail about your local set up as you can provide.

Finally, you can provide direct feedback to contact@quantecon.org

CHAPTER

FIFTYTWO

REFERENCES

CHAPTER

FIFTYTHREE

EXECUTION STATISTICS

This table contains the latest execution statistics.

Document	Modified	Method	Run Time (s)	Status
BCG_complete_mkts	2025-08-04 03:46	cache	59.32	✓
BCG_incomplete_mkts	2025-08-04 03:47	cache	93.57	✓
additive_functionals	2025-08-04 03:48	cache	14.03	✓
amss	2025-08-04 03:51	cache	190.26	✓
amss2	2025-08-04 03:52	cache	60.96	✓
amss3	2025-08-04 03:56	cache	272.51	✓
arellano	2025-08-04 03:58	cache	83.49	✓
arma	2025-08-04 03:58	cache	6.79	✓
asset_pricing_lph	2025-08-04 03:58	cache	2.45	✓
black_litterman	2025-08-04 03:59	cache	31.67	✓
calvo	2025-08-04 03:59	cache	9.13	✓
calvo_abreu	2025-08-04 03:59	cache	3.99	✓
calvo_machine_learn	2025-08-04 03:59	cache	17.7	✓
cattle_cycles	2025-08-04 03:59	cache	3.79	✓
chang_credible	2025-08-04 04:02	cache	159.18	✓
chang_ramsey	2025-08-04 04:08	cache	350.78	✓
classical_filtering	2025-08-04 04:08	cache	1.51	✓
coase	2025-08-04 04:08	cache	3.72	✓
cons_news	2025-08-04 04:08	cache	4.06	✓
discrete_dp	2025-08-04 04:08	cache	30.08	✓
dyn_stack	2025-08-04 04:08	cache	5.99	✓
entropy	2025-08-04 04:08	cache	1.13	✓
estspec	2025-08-04 04:09	cache	4.66	✓
five_preferences	2025-08-04 04:09	cache	51.4	✓
growth_in_dles	2025-08-04 04:09	cache	3.97	✓
hs_invertibility_example	2025-08-04 04:10	cache	4.08	✓
hs_recursive_models	2025-08-04 04:10	cache	1.03	✓
intro	2025-08-04 04:10	cache	1.03	✓
irfs_in_hall_model	2025-08-04 04:10	cache	3.99	✓
knowing_forecasts_of_others	2025-08-04 04:10	cache	28.64	✓
lqramsey	2025-08-04 04:10	cache	5.52	✓
lu_tricks	2025-08-04 04:10	cache	2.32	✓
lucas_asset_pricing_dles	2025-08-04 04:10	cache	3.66	✓
lucas_model	2025-08-04 04:11	cache	12.84	✓
markov_jump_lq	2025-08-04 04:12	cache	76.34	✓

continues on next page

Table 53.1 – continued from previous page

Document	Modified	Method	Run Time (s)	Status
match_transport	2025-08-04 04:12	cache	22.6	✅
matsuyama	2025-08-04 04:12	cache	6.75	✅
muth_kalman	2025-08-04 04:12	cache	4.25	✅
opt_tax_recur	2025-08-04 04:14	cache	70.66	✅
orth_proj	2025-08-04 09:06	cache	1.06	✅
permanent_income_dles	2025-08-04 04:14	cache	4.01	✅
rob_markov_perf	2025-08-04 04:14	cache	3.92	✅
robustness	2025-08-04 04:14	cache	4.42	✅
rosen_schooling_model	2025-08-04 04:14	cache	3.73	✅
smoothing	2025-08-04 04:14	cache	4.06	✅
smoothing_tax	2025-08-04 04:14	cache	6.5	✅
stationary_densities	2025-08-04 04:14	cache	7.98	✅
status	2025-08-04 04:14	cache	4.19	✅
tax_smoothing_1	2025-08-04 04:14	cache	10.65	✅
tax_smoothing_2	2025-08-04 04:14	cache	4.38	✅
tax_smoothing_3	2025-08-04 04:15	cache	4.56	✅
troubleshooting	2025-08-04 04:10	cache	1.03	✅
un_insure	2025-08-04 04:15	cache	12.36	✅
zreferences	2025-08-04 04:10	cache	1.03	✅

These lectures are built on `linux` instances through `github actions`.

These lectures are using the following python version

```
!python --version
```

```
Python 3.12.7
```

and the following package versions

```
!conda list
```

BIBLIOGRAPHY

[Abr88] Dilip Abreu. On the theory of infinitely repeated games with discounting. *Econometrica*, 56:383–396, 1988.

[APS90] Dilip Abreu, David Pearce, and Ennio Stacchetti. Toward a theory of discounted repeated games with imperfect monitoring. *Econometrica*, 58(5):1041–1063, September 1990.

[AMSSeppala02] S Rao Aiyagari, Albert Marcet, Thomas J Sargent, and Juha Seppälä. Optimal taxation without state-contingent debt. *Journal of Political Economy*, 110(6):1220–1254, 2002.

[AMS02] Franklin Allen, Stephen Morris, and Hyun Song Shin. Beauty contests, bubbles, and iterated expectations in asset markets. *mimeo*, 2002.

[AHMS96] Evan Anderson, Lars Peter Hansen, Ellen R. McGrattan, and Thomas J. Sargent. Mechanics of forming and estimating dynamic linear economies. In Hans M. Amman, David A. Kendrick, and John Rust, editors, *Handbook of computational economics*, 171–252. Elsevier Science, North-Holland, 1996.

[AHS03] Evan W. Anderson, Lars Peter Hansen, and Thomas J. Sargent. A Quartet of Semigroups for Model Specification, Robustness, Prices of Risk, and Model Detection. *Journal of the European Economic Association*, 1(1):68–123, March 2003. URL: https://ideas.repec.org/a/tpr/jeurec/v1y2003i1p68-123.html, doi:.

[Are08] Cristina Arellano. Default risk and income fluctuations in emerging economies. *The American Economic Review*, pages 690–712, 2008.

[AP91] Papoulis Athanasios and S Unnikrishna Pillai. *Probability, random variables, and stochastic processes*. McGraw Hill, 1991.

[AP11] Orazio P Attanasio and Nicola Pavoni. Risk sharing in private information models with asset accumulation: explaining the excess smoothness of consumption. *Econometrica*, 79(4):1027–1068, 2011.

[BCZ14] David Backus, Mikhail Chernov, and Stanley Zin. Sources of Entropy in Representative Agent Models. *Journal of Finance*, 69(1):51–99, February 2014. URL: https://ideas.repec.org/a/bla/jfinan/v69y2014i1p51-99.html, doi:.

[BHS09] Francisco Barillas, Lars Peter Hansen, and Thomas J. Sargent. Doubts or variability? *Journal of Economic Theory*, 144(6):2388–2418, November 2009. URL: https://ideas.repec.org/a/eee/jetheo/v144y2009i6p2388-2418.html, doi:.

[Bar79] Robert J Barro. On the Determination of the Public Debt. *Journal of Political Economy*, 87(5):940–971, 1979.

[Bar99] Robert J Barro. Determinants of democracy. *Journal of Political economy*, 107(S6):S158–S183, 1999.

[BM03] Robert J Barro and Rachel McCleary. Religion and economic growth. Technical Report, National Bureau of Economic Research, 2003.

[BEGS17] Anmol Bhandari, David Evans, Mikhail Golosov, and Thomas J. Sargent. Fiscal Policy and Debt Management with Incomplete Markets. *The Quarterly Journal of Economics*, 132(2):617–663, 2017.

[BCG18] Alberto Bisin, Gian Luca Clementi, and Piero Gottardi. Capital and hedging demand with incomplete markets. Technical Report, NYU and EUI, 2018.

[BL92] Fischer Black and Robert Litterman. Global portfolio optimization. *Financial analysts journal*, 48(5):28–43, 1992.

[BTWZ24] Job Boerma, Aleh Tsyvinski, Ruodu Wang, and Zhenyuan Zhang. Composite sorting. Technical Report, University of Wisconsin, 2024.

[Buc04] James A. Bucklew. *An Introduction to Rare Event Simulation*. Springer Verlag, New York, 2004.

[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.

[Cal78] Guillermo A. Calvo. On the time consistency of optimal policy in a monetary economy. *Econometrica*, 46(6):1411–1428, 1978.

[CR83] Gary Chamberlain and Michael Rothschild. Arbitrage, Factor Structure, and Mean-Variance Analysis on Large Asset Markets. *Econometrica*, 51(5):1281–1304, September 1983. URL: https://ideas.repec.org/a/ecm/emetrp/v51y1983i5p1281-304.html, doi:.

[Cha98] Roberto Chang. Credible monetary policy in an infinite horizon model: recursive approaches. *Journal of Economic Theory*, 81(2):431–461, 1998.

[CK90] Varadarajan V Chari and Patrick J Kehoe. Sustainable plans. *Journal of Political Economy*, pages 783–802, 1990.

[Coa37] Ronald Harry Coase. The nature of the firm. *economica*, 4(16):386–405, 1937.

[Coc05] John H. Cochrane. *Asset Pricing: revised edition*. Princeton University Press, Princeton, New Jersey, 2005.

[CC08] J. D. Cryer and K-S. Chan. *Time Series Analysis*. Springer, 2nd edition edition, 2008.

[DSS11] Julie Delon, Julien Salomon, and Andrei Sobolevski. Minimum-weight perfect matching for non-intrinsic distances on the line. *arXiv preprint arXiv:1102.1558*, 2011.

[DJ92] Raymond J Deneckere and Kenneth L Judd. Cyclical and chaotic behavior in a dynamic equilibrium model, with implications for fiscal policy. *Cycles and chaos in economic equilibrium*, pages 308–329, 1992.

[Dic75] J Dickey. Bayesian alternatives to the f-test and least-squares estimate in the normal linear model. In S.E. Fienberg and A. Zellner, editors, *Studies in Bayesian econometrics and statistics*, pages 515–554. North-Holland, Amsterdam, 1975.

[DVGC99] JBR Do Val, JC Geromel, and OLV Costa. Solutions for the linear-quadratic control problem of markov jump linear systems. *Journal of Optimization Theory and Applications*, 103(2):283–311, 1999.

[Fri56] M. Friedman. *A Theory of the Consumption Function*. Princeton University Press, 1956.

[Gal37] Albert Gallatin. Report on the finances**, november, 1807. In *Reports of the Secretary of the Treasury of the United States, Vol 1*. Government printing office, Washington, DC, 1837.

[GS89] Itzhak Gilboa and David Schmeidler. Maxmin Expected Utility with Non-Unique Prior. *Journal of Mathematical Economics*, 18(2):141–153, apr 1989.

[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.

[HS08a] L P Hansen and T J Sargent. *Robustness*. Princeton University Press, 2008.

[Han12] Lars Peter Hansen. Dynamic Valuation Decomposition Within Stochastic Economies. *Econometrica*, 80(3):911–967, May 2012. URL: https://ideas.repec.org/a/ecm/emetrp/v80y2012i3p911-967.html, doi:10.3982/ECTA8070.

[HJ91] Lars Peter Hansen and Ravi Jagannathan. Implications of Security Market Data for Models of Dynamic Economies. *Journal of Political Economy*, 99(2):225–262, April 1991. URL: https://ideas.repec.org/a/ucp/jpolec/v99y1991i2p225-62.html, doi:10.1086/261749.

[HR87] Lars Peter Hansen and Scott F Richard. The Role of Conditioning Information in Deducing Testable. *Econometrica*, 55(3):587–613, May 1987.

[HS80] Lars Peter Hansen and Thomas J Sargent. Formulating and estimating dynamic linear rational expectations models. *Journal of Economic Dynamics and control*, 2:7–46, 1980.

[HS00] Lars Peter Hansen and Thomas J Sargent. Wanting robustness in macroeconomics. *Manuscript, Department of Economics, Stanford University.*, 2000.

[HS08b] Lars Peter Hansen and Thomas J Sargent. *Robustness*. Princeton University Press, 2008.

[HS01] Lars Peter Hansen and Thomas J. Sargent. Robust control and model uncertainty. *American Economic Review*, 91(2):60–66, 2001.

[HS13] Lars Peter Hansen and Thomas J. Sargent. *Recursive Linear Models of Dynamic Economics*. Princeton University Press, Princeton, New Jersey, 2013.

[HS24] Lars Peter Hansen and Thomas J. Sargent. Risk, uncertainty, and value. University of Chicago and NYU manuscript, 2024.

[HST99] Lars Peter Hansen, Thomas J. Sargent, and Thomas D. Tallarini. Robust Permanent Income and Pricing. *Review of Economic Studies*, 66(4):873–907, 1999. URL: https://ideas.repec.org/a/oup/restud/v66y1999i4p873-907..html, doi:.

[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. URL: https://ideas.repec.org/a/eee/jetheo/v20y1979i3p381-408.html, doi:.

[HK85] Elhanan Helpman and Paul Krugman. *Market structure and international trade*. MIT Press Cambridge, 1985.

[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.

[HN97] Hugo A Hopenhayn and Juan Pablo Nicolini. Optimal Unemployment Insurance. *Journal of Political Economy*, 105(2):412–438, April 1997. URL: https://ideas.repec.org/a/ucp/jpolec/v105y1997i2p412-38.html, doi:10.1086/262078.

[HR93] Hugo A Hopenhayn and Richard Rogerson. Job Turnover and Policy Evaluation: A General Equilibrium Analysis. *Journal of Political Economy*, 101(5):915–938, 1993.

[Jac73] D. H. Jacobson. Optimal stochastic linear systems with exponential performance criteria and their relation to differential games. *IEEE Transactions on Automatic Control*, 18(2):124–131, 1973.

[Jud98] K L Judd. *Numerical Methods in Economics*. Scientific and Engineering. MIT Press, 1998.

[Jud85] Kenneth L Judd. On the performance of patents. *Econometrica*, pages 567–585, 1985.

[JYC03] Kenneth L. Judd, Sevin Yeltekin, and James Conklin. Computing Supergame Equilibria. *Econometrica*, 71(4):1239–1254, 07 2003. URL: https://ideas.repec.org/a/ecm/emetrp/v71y2003i4p1239-1254.html, doi:.

[Kas00] Kenneth Kasa. Forecasting the forecasts of others in the frequency domain. *Review of Economic Dynamics*, 3:726–756, 2000.

[KNS18] Tomoo Kikuchi, Kazuo Nishimura, and John Stachurski. Span of control, transaction costs, and the structure of production chains. *Theoretical Economics*, 13(2):729–760, 2018.

[Kni21] Frank H. Knight. *Risk, Uncertainty, and Profit*. Houghton Mifflin, 1921.

[Kre81] David M. Kreps. Arbitrage and equilibrium in economies with infinitely many commodities. *Journal of Mathematical Economics*, 8(1):15–35, March 1981. URL: https://ideas.repec.org/a/eee/mateco/v8y1981i1p15-35.html, doi:.

[KP80] Finn E Kydland and Edward C Prescott. Dynamic optimal taxation, rational expectations and optimal control. *Journal of Economic Dynamics and Control*, 2:79–91, 1980.

[LM94] A Lasota and M C MacKey. *Chaos, Fractals, and Noise: Stochastic Aspects of Dynamics*. Applied Mathematical Sciences. Springer-Verlag, 1994.

[Lea78] Edward E Leamer. *Specification searches: Ad hoc inference with nonexperimental data*. Volume 53. John Wiley & Sons Incorporated, 1978.

[LWY13] Eric M. Leeper, Todd B. Walker, and Shu-Chun Susan Yang. Fiscal foresight and information flows. *Econometrica*, 81(3):1115–1145, May 2013.

[LS18] L Ljungqvist and T J Sargent. *Recursive Macroeconomic Theory*. MIT Press, 4 edition, 2018.

[Luc87] Robert E Lucas. *Models of business cycles*. Volume 26. Oxford Blackwell, 1987.

[Luc78] Robert E Lucas, Jr. Asset prices in an exchange economy. *Econometrica: Journal of the Econometric Society*, 46(6):1429–1445, 1978.

[LS83] Robert E Lucas, Jr. and Nancy L Stokey. Optimal Fiscal and Monetary Policy in an Economy without Capital. *Journal of monetary Economics*, 12(3):55–93, 1983.

[MMR06] Fabio Maccheroni, Massimo Marinacci, and Aldo Rustichini. Ambiguity Aversion, Robustness, and the Variational Representation of Preferences. *Econometrica*, 74(6):1147–1498, 2006.

[MT09] S P Meyn and R L Tweedie. *Markov Chains and Stochastic Stability*. Cambridge University Press, 2009.

[MF02] Mario J Miranda and P L Fackler. *Applied Computational Economics and Finance*. Cambridge: MIT Press, 2002.

[MM58] Franco Modigliani and Merton H. Miller. Corporation finance and the theory of investment. *American Economic Review*, XLVIII(3):261–297, 1958.

[Mut60] John F Muth. Optimal properties of exponentially weighted forecasts. *Journal of the american statistical association*, 55(290):299–306, 1960.

[Orf88] Sophocles J Orfanidis. *Optimum Signal Processing: An Introduction*. McGraw Hill Publishing, New York, New York, 1988.

[PCL86] Joseph Pearlman, David Currie, and Paul Levine. Rational Expectations Models with Private Information. *Economic Modelling*, 3(2):90–105, 1986.

[PS05] Joseph G. Pearlman and Thomas J. Sargent. Knowing the Forecasts of Others. *Review of Economic Dynamics*, 8(2):480–497, April 2005. URL: https://ideas.repec.org/a/red/issued/v8y2005i2p480-497.html, doi:10.1016/j.red.2004.10.011.

[Put05] Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2005.

[Ram27] F. P. Ramsey. A Contribution to the theory of taxation. *Economic Journal*, 37(145):47–61, 1927.

[REL75] Jr. Robert E. Lucas. An equilibrium model of the business cycle. *Journal of Political Economy*, 83:1113–1144, 1975.

[Rom05] Steven Roman. *Advanced linear algebra*. Volume 3. Springer, 2005.

[RMS94] Sherwin Rosen, Kevin M Murphy, and Jose A Scheinkman. Cattle cycles. *Journal of Political Economy*, 102(3):468–492, 1994.

[Ros78] Stephen A Ross. A Simple Approach to the Valuation of Risky Streams. *The Journal of Business*, 51(3):453–475, July 1978. URL: https://ideas.repec.org/a/ucp/jnlbus/v51y1978i3p453-75.html.

[Ros76] Stephen A. Ross. The arbitrage theory of capital asset pricing. *Journal of Economic Theory*, 13(3):341–360, December 1976. URL: https://ideas.repec.org/a/eee/jetheo/v13y1976i3p341-360.html, doi:.

[Roz67] Y. A. Rozanov. *Stationary Random Processes*. Holden-Day, San Francisco, 1967.

[Rus96] John Rust. Numerical dynamic programming in economics. *Handbook of computational economics*, 1:619–729, 1996.

[RR04] Jaewoo Ryoo and Sherwin Rosen. The engineering labor market. *Journal of political economy*, 112(S1):S110–S140, 2004.

[SHR91] Thomas Sargent, Lars Peter Hansen, and Will Roberts. Observable implications of present value budget balance. In *Rational Expectations Econometrics*. Westview Press, 1991.

[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.

[SW73] Thomas J Sargent and Neil Wallace. The stability of models of money and growth with perfect foresight. *Econometrica: Journal of the Econometric Society*, pages 1043–1048, 1973.

[Sar91] Thomas J. Sargent. Equilibrium with signal extraction from endogenous variables. *Journal of Economic Dynamics and Control*, 15:245–273, 1991.

[SW49] Claude E. Shannon and Warren Weaver. *The Mathematical Theory of Communication*. University of Illinois Press, Urbana, 1949.

[SW79] Steven Shavell and Laurence Weiss. The optimal payment of unemployment insurance benefits over time. *Journal of political Economy*, 87(6):1347–1362, 1979.

[Shi95] A N Shiriaev. *Probability*. Graduate texts in mathematics. Springer. Springer, 2nd edition, 1995.

[Sin87] Kenneth J. Singleton. Asset prices in a time-series model with disparately informed competitive traders. In William A. Barnett and Kenneth J. Singleton, editors, *New Apprroaches to Monetary Economics*. Cambridge University Press, 1987.

[SLP89] N L Stokey, R E Lucas, and E C Prescott. *Recursive Methods in Economic Dynamics*. Harvard University Press, 1989.

[Sto89] Nancy L Stokey. Reputation and time consistency. *The American Economic Review*, pages 134–139, 1989.

[Sto91] Nancy L. Stokey. Credible public policy. *Journal of Economic Dynamics and Control*, 15(4):627–656, October 1991.

[SW09] Lars E.O. Svensson and Noah Williams. Optimal Monetary Policy under Uncertainty in DSGE Models: A Markov Jump-Linear-Quadratic Approach. In Klaus Schmidt-Hebbel, Carl E. Walsh, Norman Loayza (Series Editor), and Klaus Schmidt-Hebbel (Series, editors, *Monetary Policy under Uncertainty and Learning*, volume 13 of Central Banking, Analysis, and Economic Policies Book Series, chapter 3, pages 077–114. Central Bank of Chile, edition, March 2009.

[SW+08] Lars EO Svensson, Noah Williams, and others. Optimal monetary policy under uncertainty: a markov jump-linear-quadratic approach. *Federal Reserve Bank of St. Louis Review*, 90(4):275–293, 2008.

[Tal00] Thomas D Tallarini. Risk-sensitive real business cycles. *Journal of Monetary Economics*, 45(3):507–532, June 2000.

[Tow83] Robert M. Townsend. Forecasting the forecasts of others. *Journal of Political Economy*, 91:546–588, 1983.

[Whi63] Peter Whittle. *Prediction and regulation by linear least-square methods*. English Univ. Press, 1963.

[Whi81] Peter Whittle. Risk-sensitive linear/quadratic/gaussian control. *Advances in Applied Probability*, 13(4):764–777, 1981.

[Whi83] Peter Whittle. *Prediction and Regulation by Linear Least Squares Methods*. University of Minnesota Press, Minneapolis, Minnesota, 2nd edition, 1983.

[Whi90] Peter Whittle. *Risk-Sensitive Optimal Control*. Wiley, New York, 1990.

PROOF INDEX

opt

opt (*orth_proj*), 9

opt_another

opt_another (*orth_proj*), 11

proj_matrix

proj_matrix (*orth_proj*), 14

square-summable

square-summable (*calvo_machine_learn*), 797

theorem-2

theorem-2 (*orth_proj*), 13

theorem-4

theorem-4 (*orth_proj*), 15

theorem-5

theorem-5 (*orth_proj*), 17

theorem-6

theorem-6 (*orth_proj*), 18

INDEX

A

AR, 528
ARMA, 525, 528
ARMA Processes, 521

B

Bellman Equation, 479

C

Coase's Theory of the Firm, 289
Complex Numbers, 526
Consumption
    Tax, 91
Covariance Stationary, 522
Covariance Stationary Processes, 521
    AR, 524
    MA, 524

D

Discrete State Dynamic Programming, 53

E

Elementary Asset Pricing, 659

F

Fixed Point Theory, 653

G

General Linear Processes, 523

L

Linear Markov Perfect Equilibria, 500
Lucas Model, 649
    Assets, 650
    Computation, 654
    Consumers, 650
    Dynamic Program, 651
    Equilibrium Constraints, 652
    Equilibrium Price Function, 652
    Pricing, 650
    Solving, 652

M

MA, 528
Markov Chains
    Continuous State, 23
Markov Perfect Equilibrium
    Applications, 503
    Overview, 499
Models
    Additive functionals, 559, 823, 847
    Lucas Asset Pricing, 649

N

Nonparametric Estimation, 546

O

Orthogonal Projection, 5

P

Periodograms, 543
    Computation, 545
    Interpretation, 544
python, 43, 142, 195, 205, 219, 359, 388, 399, 407, 417, 422, 428, 435, 672

R

Ramsey Problem
    Optimal Taxation, 227
Robustness, 479

S

Smoothing, 546
    Tax, 105
Spectra
    Estimation, 543
Spectra, Estimation
    AR(1) Setting, 551
    Fast Fourier Transform, 543
    Pre-Filtering, 551
    Smoothing, 546, 549, 551
Spectral Analysis, 521, 526
Spectral Densities, 527

Spectral Density, 528
    interpretation, 528
    Inverting the Transformation, 533
    Mathematical Theory, 533

W

White Noise, 523, 527
Wold Representation, 523