# Advanced Quantitative Economics with Python

**Thomas J. Sargent & John Stachurski**

**Nov 24, 2021**

# 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 . . . 10
1.4 Orthonormal Basis . . . 13
1.5 Projection Via Matrix Algebra . . . 15
1.6 Least Squares Regression . . . 17
1.7 Orthogonalization and Decomposition . . . 19
1.8 Exercises . . . 20
1.9 Solutions . . . 20

**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 Solutions . . . 38
2.7 Appendix . . . 43

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

**4 Discrete State Dynamic Programming** **53**
4.1 Overview . . . 53
4.2 Discrete DPs . . . 55
4.3 Solving Discrete DPs . . . 58
4.4 Example: A Growth Model . . . 59
4.5 Exercises . . . 64
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 the **Same** Nonfinancial Income Process . . . 78
5.3 State Space Representations . . . 82

**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 . . . 93
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 107**
7.1 Overview . . . 107
7.2 Tax Smoothing with Complete Markets . . . 112
7.3 Returns on State-Contingent Debt . . . 113
7.4 More Finite Markov Chain Tax-Smoothing Examples . . . 117

**8 Markov Jump Linear Quadratic Dynamic Programming 141**
8.1 Overview . . . 141
8.2 Review of useful LQ dynamic programming formulas . . . 142
8.3 Linked Riccati equations for Markov LQ dynamic programming . . . 143
8.4 Applications . . . 144
8.5 Example 1 . . . 144
8.6 Example 2 . . . 160
8.7 More examples . . . 179

**9 How to Pay for a War: Part 1 181**
9.1 Reader's Guide . . . 181
9.2 Public Finance Questions . . . 183
9.3 Barro (1979) Model . . . 183
9.4 Python Class to Solve Markov Jump Linear Quadratic Control Problems . . . 187
9.5 Barro Model with a Time-varying Interest Rate . . . 187

**10 How to Pay for a War: Part 2 191**
10.1 An Application of Markov Jump Linear Quadratic Dynamic Programming . . . 191
10.2 Two example specifications . . . 192
10.3 One- and Two-period Bonds but No Restructuring . . . 192
10.4 Mapping into an LQ Markov Jump Problem . . . 193
10.5 Penalty on Different Issuance Across Maturities . . . 196
10.6 A Model with Restructuring . . . 198
10.7 Restructuring as a Markov Jump Linear Quadratic Control Problem . . . 201

**11 How to Pay for a War: Part 3 205**
11.1 Another Application of Markov Jump Linear Quadratic Dynamic Programming . . . 205
11.2 Roll-Over Risk . . . 206
11.3 A Dead End . . . 206
11.4 Better Representation of Roll-Over Risk . . . 207

**12 Optimal Taxation in an LQ Economy 211**
12.1 Overview . . . 211
12.2 The Ramsey Problem . . . 212
12.3 Implementation . . . 218

12.4 Examples . . . 224
12.5 Exercises . . . 228
12.6 Solutions . . . 229

## III Multiple Agent Models 233

**13 Default Risk and Income Fluctuations 235**
13.1 Overview . . . 235
13.2 Structure . . . 236
13.3 Equilibrium . . . 238
13.4 Computation . . . 239
13.5 Results . . . 245
13.6 Exercises . . . 247
13.7 Solutions . . . 249

**14 Globalization and Cycles 255**
14.1 Overview . . . 255
14.2 Key Ideas . . . 256
14.3 Model . . . 257
14.4 Simulation . . . 260
14.5 Exercises . . . 270
14.6 Solutions . . . 270

**15 Coase's Theory of the Firm 273**
15.1 Overview . . . 273
15.2 The Model . . . 275
15.3 Equilibrium . . . 277
15.4 Existence, Uniqueness and Computation of Equilibria . . . 278
15.5 Implementation . . . 280
15.6 Exercises . . . 284
15.7 Solutions . . . 284

## IV Dynamic Linear Economies 287

**16 Recursive Models of Dynamic Linear Economies 289**
16.1 A Suite of Models . . . 289
16.2 Econometrics . . . 306
16.3 Dynamic Demand Curves and Canonical Household Technologies . . . 309
16.4 Gorman Aggregation and Engel Curves . . . 310
16.5 Partial Equilibrium . . . 311
16.6 Equilibrium Investment Under Uncertainty . . . 312
16.7 A Rosen-Topel Housing Model . . . 312
16.8 Cattle Cycles . . . 312
16.9 Models of Occupational Choice and Pay . . . 313
16.10 Permanent Income Models . . . 315
16.11 Gorman Heterogeneous Households . . . 316
16.12 Non-Gorman Heterogeneous Households . . . 317

**17 Growth in Dynamic Linear Economies 319**
17.1 Common Structure . . . 319
17.2 A Planning Problem . . . 320
17.3 Example Economies . . . 321

**18 Lucas Asset Pricing Using DLE** **329**
18.1 Asset Pricing Equations . . . 330
18.2 Asset Pricing Simulations . . . 330

**19 IRFs in Hall Models** **337**
19.1 Example 1: Hall (1978) . . . 337
19.2 Example 2: Higher Adjustment Costs . . . 340
19.3 Example 3: Durable Consumption Goods . . . 342

**20 Permanent Income Model using the DLE Class** **345**
20.1 The Permanent Income Model . . . 346

**21 Rosen Schooling Model** **351**
21.1 A One-Occupation Model . . . 351
21.2 Mapping into HS2013 Framework . . . 352

**22 Cattle Cycles** **357**
22.1 The Model . . . 357
22.2 Mapping into HS2013 Framework . . . 358

**23 Shock Non Invertibility** **365**
23.1 Overview . . . 365
23.2 Model . . . 366
23.3 Code . . . 368

**V Risk, Model Uncertainty, and Robustness** **371**

**24 Risk and Model Uncertainty** **373**
24.1 Overview . . . 373
24.2 Basic objects . . . 374
24.3 Five preference specifications . . . 377
24.4 Expected utility . . . 378
24.5 Constraint preferences . . . 378
24.6 Multiplier preferences . . . 379
24.7 Risk-sensitive preferences . . . 380
24.8 Ex post Bayesian preferences . . . 383
24.9 Comparing preferences . . . 383
24.10 Risk aversion and misspecification aversion . . . 385
24.11 Indifference curves . . . 386
24.12 State price deflators . . . 389
24.13 Iso-utility and iso-entropy curves and expansion paths . . . 392
24.14 Bounds on expected utility . . . 393
24.15 Why entropy? . . . 395

**25 Robustness** **397**
25.1 Overview . . . 397
25.2 The Model . . . 401
25.3 Constructing More Robust Policies . . . 402
25.4 Robustness as Outcome of a Two-Person Zero-Sum Game . . . 403
25.5 The Stochastic Case . . . 407
25.6 Implementation . . . 409
25.7 Application . . . 409
25.8 Appendix . . . 415

**26 Robust Markov Perfect Equilibrium** **417**
26.1 Overview . . . 417
26.2 Linear Markov Perfect Equilibria with Robust Agents . . . 418
26.3 Application . . . 421

## VI Classic Linear Models 437

**27 Von Neumann Growth Model (and a Generalization)** **439**
27.1 Notation . . . 444
27.2 Model Ingredients and Assumptions . . . 445
27.3 Dynamic Interpretation . . . 446
27.4 Duality . . . 447
27.5 Interpretation as Two-player Zero-sum Game . . . 449

## VII Time Series Models 455

**28 Covariance Stationary Processes** **457**
28.1 Overview . . . 457
28.2 Introduction . . . 458
28.3 Spectral Analysis . . . 462
28.4 Implementation . . . 470

**29 Estimation of Spectra** **479**
29.1 Overview . . . 479
29.2 Periodograms . . . 480
29.3 Smoothing . . . 484
29.4 Exercises . . . 488
29.5 Solutions . . . 490

**30 Additive and Multiplicative Functionals** **495**
30.1 Overview . . . 495
30.2 A Particular Additive Functional . . . 496
30.3 Dynamics . . . 497
30.4 Code . . . 509
30.5 More About the Multiplicative Martingale . . . 512

**31 Classical Control with Linear Algebra** **519**
31.1 Overview . . . 519
31.2 A Control Problem . . . 520
31.3 Finite Horizon Theory . . . 521
31.4 The Infinite Horizon Limit . . . 525
31.5 Undiscounted Problems . . . 528
31.6 Implementation . . . 529
31.7 Exercises . . . 537

**32 Classical Prediction and Filtering With Linear Algebra** **539**
32.1 Overview . . . 539
32.2 Finite Dimensional Prediction . . . 540
32.3 Combined Finite Dimensional Control and Prediction . . . 551
32.4 Infinite Horizon Prediction and Filtering Problems . . . 552
32.5 Exercises . . . 556

**33 Knowing the Forecasts of Others** **559**

33.1 Introduction . . . 559
33.2 The Setting . . . 561
33.3 Tactics . . . 561
33.4 Equilibrium conditions . . . 563
33.5 Equilibrium with $\theta_t$ stochastic but observed at $t$ . . . 564
33.6 Guess-and-verify tactic . . . 567
33.7 Equilibrium with one signal on $\theta_t$ . . . 567
33.8 Equilibrium with two noisy signals on $\theta_t$ . . . 573
33.9 Key step . . . 577
33.10 Comparison of the two signal structures . . . 577
33.11 Notes on History of the Problem . . . 579

**VIII Asset Pricing and Finance 581**

**34 Asset Pricing II: The Lucas Asset Pricing Model 583**
34.1 Overview . . . 583
34.2 The Lucas Model . . . 584
34.3 Exercises . . . 592
34.4 Solutions . . . 592

**35 Elementary Asset Pricing Theory 593**
35.1 Overview . . . 593
35.2 Key Equation . . . 594
35.3 Implications of Key Equation . . . 594
35.4 Expected Return - Beta Representation . . . 595
35.5 Mean-Variance Frontier . . . 596
35.6 Empirical Implementations . . . 597
35.7 Exercises . . . 598
35.8 Solutions . . . 601

**36 Two Modifications of Mean-Variance Portfolio Theory 605**
36.1 Overview . . . 605
36.2 Appendix . . . 620

**37 Irrelevance of Capital Structure with Complete Markets 627**
37.1 Introduction . . . 627
37.2 Competitive equilibrium . . . 631
37.3 Code . . . 637

**38 Equilibrium Capital Structures with Incomplete Markets 647**
38.1 Introduction . . . 647
38.2 Asset Markets . . . 651
38.3 Equilibrium verification . . . 654
38.4 Pseudo Code . . . 654
38.5 Code . . . 656
38.6 Examples . . . 666
38.7 A picture worth a thousand words . . . 692

**IX Dynamic Programming Squared 693**

**39 Stackelberg Plans 695**
39.1 Overview . . . 695
39.2 Duopoly . . . 696

39.3 The Stackelberg Problem . . . 699
39.4 Stackelberg Plan . . . 702
39.5 Recursive Representation of Stackelberg Plan . . . 703
39.6 Computing the Stackelberg Plan . . . 706
39.7 Exhibiting Time Inconsistency of Stackelberg Plan . . . 710
39.8 Recursive Formulation of the Follower's Problem . . . 711
39.9 Markov Perfect Equilibrium . . . 715
39.10 MPE vs. Stackelberg . . . 718

**40 Ramsey Plans, Time Inconsistency, Sustainable Plans** **721**
40.1 Overview . . . 722
40.2 The Model . . . 722
40.3 Structure . . . 724
40.4 Intertemporal Influences . . . 725
40.5 Four Models of Government Policy . . . 725
40.6 A Ramsey Planner . . . 726
40.7 A Constrained-to-a-Constant-Growth-Rate Ramsey Government . . . 728
40.8 Markov Perfect Governments . . . 728
40.9 Equilibrium Outcomes for Three Models of Government Policy Making . . . 729
40.10 A Fourth Model of Government Decision Making . . . 736
40.11 Sustainable or Credible Plan . . . 737
40.12 Whose Credible Plan is it? . . . 743
40.13 Comparison of Equilibrium Values . . . 743
40.14 Note on Dynamic Programming Squared . . . 744

**41 Optimal Taxation with State-Contingent Debt** **745**
41.1 Overview . . . 745
41.2 A Competitive Equilibrium with Distorting Taxes . . . 746
41.3 Recursive Formulation of the Ramsey Problem . . . 757
41.4 Examples . . . 765

**42 Optimal Taxation without State-Contingent Debt** **777**
42.1 Overview . . . 777
42.2 Competitive Equilibrium with Distorting Taxes . . . 778
42.3 Recursive Version of AMSS Model . . . 786
42.4 Examples . . . 793

**43 Fluctuating Interest Rates Deliver Fiscal Insurance** **805**
43.1 Overview . . . 805
43.2 Forces at Work . . . 807
43.3 Logical Flow of Lecture . . . 807
43.4 Example Economy . . . 810
43.5 Reverse Engineering Strategy . . . 820
43.6 Code for Reverse Engineering . . . 820
43.7 Short Simulation for Reverse-engineered: Initial Debt . . . 822
43.8 Long Simulation . . . 831
43.9 BEGS Approximations of Limiting Debt and Convergence Rate . . . 833

**44 Fiscal Risk and Government Debt** **839**
44.1 Overview . . . 839
44.2 The Economy . . . 840
44.3 Long Simulation . . . 842
44.4 Asymptotic Mean and Rate of Convergence . . . 866

**45 Competitive Equilibria of a Model of Chang** **875**

45.1 Overview . . . 875
45.2 Setting . . . 877
45.3 Competitive Equilibrium . . . 879
45.4 Inventory of Objects in Play . . . 879
45.5 Analysis . . . 880
45.6 Calculating all Promise-Value Pairs in CE . . . 884
45.7 Solving a Continuation Ramsey Planner's Bellman Equation . . . 899

**46 Credible Government Policies in a Model of Chang** **907**
46.1 Overview . . . 907
46.2 The Setting . . . 908
46.3 Calculating the Set of Sustainable Promise-Value Pairs . . . 914

**X Other** **931**

**47 Troubleshooting** **933**
47.1 Fixing Your Local Environment . . . 933
47.2 Reporting an Issue . . . 934

**48 References** **935**

**49 Execution Statistics** **937**

**Bibliography** **939**

**Index** **945**

This website presents a set of advanced lectures on quantitative economic modeling, designed and written by Thomas J. Sargent and John Stachurski.

For an overview of the series, see this page


- 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*
- 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*
  - *Robustness*
  - *Robust Markov Perfect Equilibrium*
- Classic Linear Models
  - *Von Neumann Growth Model (and a Generalization)*

- 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 Structure with Complete Markets*
  - *Equilibrium Capital Structures with Incomplete Markets*
- Dynamic Programming Squared
  - *Stackelberg Plans*
  - *Ramsey Plans, Time Inconsistency, Sustainable Plans*
  - *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*


**Previous website**

While this new site will receive all future updates, you may still view the old site here for the next month.

# Part I

# Tools and Techniques

CHAPTER

ONE

# ORTHOGONAL PROJECTIONS AND THEIR APPLICATIONS

**Contents**


- *Orthogonal Projections and Their Applications*
  - *Overview*
  - *Key Definitions*
  - *The Orthogonal Projection Theorem*
  - *Orthonormal Basis*
  - *Projection Via Matrix Algebra*
  - *Least Squares Regression*
  - *Orthogonalization and Decomposition*
  - *Exercises*
  - *Solutions*


## 1.1 Overview

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

These include, but are not limited to,

- 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, [Rom05].

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 $\theta$ 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

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

If $\{x_1, \dots, x_k\}$ 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

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

$x$
$z$

$S$
$x$

$S^\perp$
$S$

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

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

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.

### 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 OPT, 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 $S$.

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

$y$
$S$
$Py$
$y'$
$Py'$

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

It is immediate from the OPT 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 OPT:

**Theorem**. 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 $\{e_1, \dots, e_n\}$ that forms an orthonormal basis of $\mathbb{R}^n$, where $e_i$ is the $i$ th unit vector.

If $\{u_1, \dots, u_k\}$ 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 $x \in \text{span}\{u_1, \dots, u_k\}$, we can find scalars $\alpha_1, \dots, \alpha_k$ that verify

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

$y$
$S^\perp$
$S$
$My$
$Py$

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

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

Proof: Fix $y \in \mathbb{R}^n$ and let $Py$ be defined as in (1.2).

Clearly, $Py \in S$.

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

It sufficies to show that $y - Py \perp$ 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.** 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

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

Then the columns of $X$ form a basis of $S$.

From the preceding theorem, $P = X(X'X)^{-1}X'y$ 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 $u_i := \operatorname{col} U_i$ 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 $b$ 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 OPT.

**Theorem** 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 $\text{span}(X)$ 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 OPT.

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

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

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

One description of this procedure is as follows:

- For $i = 1, \dots, k$, form $S_i := \text{span}\{x_1, \dots, x_i\}$ 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 $u_1, \dots, u_k$ 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** 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 sketch: Let

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

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

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

Some rearranging gives $X = QR$.

### 1.7.3 Linear Regression via QR Decomposition

For matrices $X$ and $y$ that overdetermine $\beta$ 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

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

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

## 1.8 Exercises

### 1.8.1 Exercise 1

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

### 1.8.2 Exercise 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?

### 1.8.3 Exercise 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

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

## 1.9 Solutions

### 1.9.1 Exercise 1

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

### 1.9.2 Exercise 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.).

### 1.9.3 Exercise 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

    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 col of U is just the normalized first col 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[:, 0:i]      # First i-1 columns of X

        # Project onto the orthogonal complement of the col 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]]
```

(continues on next page)

(continued from previous page)

```
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],
       [ 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

**Contents**


- *Continuous State Markov Chains*
  - *Overview*
  - *The Density Case*
  - *Beyond Densities*
  - *Stability*
  - *Exercises*
  - *Solutions*
  - *Appendix*


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., [MT09]) 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
%matplotlib inline
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 $\phi$ is a given density on $\mathbb{R}$
- $\mu$ and $\sigma$ 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

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

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 $t$
- $\delta$ 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 [LS18], 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 $\sigma(x) = s f(x)$.

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

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

where $\phi$ is the density of $A_{t+1}$.

(Regarding the state space $S$ for this model, a natural choice is $(0, \infty)$ — in which case $\sigma(x) = s f(x)$ 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 $\psi_t$

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

Letting $\psi_{t+1}$ 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 $\psi$ and sends it into new density $\psi P$, 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., $\psi P$ 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 $\psi_0$, 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 $\psi P$ from $\psi$ 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 $\psi_0$.

To approximate $\psi_t$ by simulation, recall that, by definition, $\psi_t$ 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 $\psi_0$
2. draw the shocks $A_1, \dots, A_t$ from their specified density $\phi$
3. compute $k_t$ iteratively via (2.12)

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

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

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 $\psi_t$, we actually generate $n$ observations of $k_{t-1}$, rather than $k_t$.

Now we take these $n$ observations $k_{t-1}^1, \dots, k_{t-1}^n$ 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 $\psi_{t-1}$, 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)]
```

(continues on next page)

(continued from previous page)

```
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 [HR93] 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 $\psi^*$ 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 $\psi^*$ is stationary for $P$, and the distribution of $X_0$ is $\psi^*$, then, in view of (2.11), $X_t$ will have this same distribution for all $t$.

Hence $\psi^*$ 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 $\psi^*$ 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 $h\colon S \to \mathbb{R}$ 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 $\psi^*$ (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

- [SLP89] contains a classic (but slightly outdated) treatment of these topics.
- From the mathematical literature, [LM94] and [MT09] 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 $\psi^*$.

Even from this figure, we can get a fair idea what $\psi^*$ 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 $\psi^*$, we can simply generate a long time-series $X_0, X_1, \dots, X_n$ and estimate $\psi^*$ 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

### 2.5.1 Exercise 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 $\psi^*$ given by

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

Here $\phi$ is the standard normal density and $\Phi$ is the standard normal cdf.

As an exercise, compute the look-ahead estimate of $\psi^*$, as defined in (2.18), and compare it with $\psi^*$ 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.

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

### 2.5.2 Exercise 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)
```

### 2.5.3 Exercise 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 $\theta = 0.9, n = 20, k = 5000, J = 8$

## 2.6 Solutions

### 2.6.1 Exercise 1

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

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

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()
```

### 2.6.2 Exercise 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)

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)
```

(continues on next page)

(continued from previous page)

```
    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()
```

### 2.6.3 Exercise 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 = 6

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 = 6.4
time series from t = 4.8
time series from t = 3.1999999999999993
time series from t = 1.5999999999999996
time series from t = 0.0

## 2.7 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

**Contents**


- *Reverse Engineering a la Muth*
  - *Friedman (1956) and Muth (1960)*
  - *A Process for Which Adaptive Expectations are Optimal*
  - *Some Useful State-Space Math*
  - *Estimates of Unobservables*
  - *Relation between Unobservable and Observable*
  - *MA and AR Representations*


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
%matplotlib inline
import numpy as np
import scipy.linalg as la

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

This lecture uses the Kalman filter to reformulate John F. Muth's first paper [Mut60] about rational expectations.

Muth used *classical* prediction methods to reverse engineer a stochastic process that renders optimal Milton Friedman's [Fri56] "adaptive expectations" scheme.

## 3.1 Friedman (1956) and Muth (1960)

Milton Friedman [Fri56] (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 [Mut60] 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 $\hat{x}_t = E[x_t|y_{t-1}, y_{t-2}, ...]$ and $a_t = y_t - E[y_t|y_{t-1}, y_{t-2}, ...]$.

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

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)

# 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()

# 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 $\hat{x}_t$.

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

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

so that

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

The stacked system

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

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

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 $\hat{x}_{t+1}, x_t$, 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}")
```

```
/tmp/ipykernel_13238/1459986415.py:3: VisibleDeprecationWarning: Creating an␣
↪ndarray from ragged nested sequences (which is a list-or-tuple of lists-or-
↪tuples-or ndarrays with different lengths or shapes) is deprecated. If you meant␣
↪to do this, you must specify 'dtype=object' when creating the ndarray.
  Af = np.array([[ 1,      0,        0],
/tmp/ipykernel_13238/1459986415.py:6: VisibleDeprecationWarning: Creating an␣
↪ndarray from ragged nested sequences (which is a list-or-tuple of lists-or-
↪tuples-or ndarrays with different lengths or shapes) is deprecated. If you meant␣
↪to do this, you must specify 'dtype=object' when creating the ndarray.
  Cf = np.array([[σ_x,        0],
```

```
Kalman gain = [[0.181]]
Conditional variance = [[5.5249]]
```

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 Relation between Unobservable and Observable

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 $y$")
```

(continues on next page)

(continued from previous page)

```
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 $a_t$s 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")
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 = [[0.819]]
```

CHAPTER

FOUR

# DISCRETE STATE DYNAMIC PROGRAMMING

**Contents**


- *Discrete State Dynamic Programming*
  - *Overview*
  - *Discrete DPs*
  - *Solving Discrete DPs*
  - *Example: A Growth Model*
  - *Exercises*
  - *Solutions*
  - *Appendix: Algorithms*


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
%matplotlib inline
import quantecon as qe
import scipy.sparse as sparse
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,

- [LS18]
- [HLL96], section 3.5
- [Put05]
- [SLP89]
- [Rus96]

- [MF02]
- 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
- $\beta$ 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 $\sigma$ 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 [Put05] 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* $S = \{0, \dots, n-1\}$.
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 $A := \bigcup_{s \in S} A(s) = \{0, \dots, m-1\}$ 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 $\Sigma$.

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

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

Notice that $Q_\sigma$ is a stochastic matrix on $S$.

It gives transition probabilities of the *controlled chain* when we follow policy $\sigma$.

If we think of $r_\sigma$ 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 $Q_\sigma$.
- 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 $\sigma$ 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 $\sigma$.

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 $T_\sigma$

They are also contraction mappings with modulus $\beta$

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

For any policy $\sigma$, its value $v_\sigma$ is the unique fixed point of $T_\sigma$.

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

- $\sigma^*$ is an optimal policy function if and only if it is $v^*$-greedy

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

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

## 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 $v^0 \colon S \to \mathbb{R}$ 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 $v_\sigma$ of a policy $\sigma$ by solving the linear equation $v = T_\sigma v$.
2. A policy improvement step that computes a $v_\sigma$-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 $\sigma$ by iterating $T_\sigma$ 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 $\beta$.

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))
        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 $Q_{\sigma^*}$, where $\sigma^*$ 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

### 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
```

(continues on next page)

(continued from previous page)

```
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)
```

(continues on next page)

(continued from previous page)

```
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]
```

(continues on next page)

(continued from previous page)

```
# 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()
```

```
121.49819147053373
```

```
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')
```

```
179 ms ± 114 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
16.9 ms ± 8.66 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
19.4 ms ± 34.6 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'
```

(continues on next page)

(continued from previous page)

```
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')

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,
```

(continues on next page)

(continued from previous page)

```
                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      1.238e-03
2            4.070e+00      2.125e-03
Iteration    Distance       Elapsed (seconds)
--------------------------------------------
1            5.518e+00      7.658e-04
2            4.070e+00      1.572e-03
3            3.866e+00      2.390e-03
4            3.673e+00      3.157e-03
Iteration    Distance       Elapsed (seconds)
--------------------------------------------
1            5.518e+00      7.429e-04
2            4.070e+00      1.454e-03
3            3.866e+00      2.239e-03
4            3.673e+00      2.991e-03
5            3.489e+00      3.732e-03
6            3.315e+00      4.478e-03
```

```
/usr/share/miniconda3/envs/quantecon/lib/python3.8/site-packages/quantecon/compute_
 ↪fp.py:151: RuntimeWarning: max_iter attained before convergence in compute_fixed_
 ↪point
  warnings.warn(_non_convergence_msg, RuntimeWarning)
/usr/share/miniconda3/envs/quantecon/lib/python3.8/site-packages/quantecon/compute_
 ↪fp.py:151: RuntimeWarning: max_iter attained before convergence in compute_fixed_
 ↪point
  warnings.warn(_non_convergence_msg, RuntimeWarning)
/usr/share/miniconda3/envs/quantecon/lib/python3.8/site-packages/quantecon/compute_
 ↪fp.py:151: 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=f'$\\beta = {beta}$')

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 $\varepsilon$-approximation of $v^*$ if $\|v - v^*\| < \varepsilon$.
- A policy $\sigma \in \Sigma$ is called $\varepsilon$-optimal if $v_\sigma$ is an $\varepsilon$-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 $\sigma$, and return $v^{i+1}$ and $\sigma$.

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 $\varepsilon$-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 $\sigma^0$; set $i = 0$.
2. Compute the value $v_{\sigma^i}$ by solving the equation $v = T_{\sigma^i} v$.
3. Compute a $v_{\sigma^i}$-greedy policy $\sigma^{i+1}$; let $\sigma^{i+1} = \sigma^i$ if possible.
4. If $\sigma^{i+1} = \sigma^i$, then return $v_{\sigma^i}$ and $\sigma^{i+1}$; 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 $\sigma^{i+1}$; 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 $\sigma_{i+1}$.

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 $\varepsilon$-optimal policy function (unless `iter_max` is reached).

See also the documentation for `DiscreteDP`.

# Part II

# LQ Control

CHAPTER

FIVE

# INFORMATION AND CONSUMPTION SMOOTHING

**Contents**


- *Information and Consumption Smoothing*
  - *Overview*
  - *Two Representations of the **Same** Nonfinancial Income Process*
  - *State Space Representations*


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

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

## 5.1 Overview

This lecture studies 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 in the quantecon lecture.

The two consumers have different information about future nonfinancial incomes.

One 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.

The other, less well informed, consumer each period receives a shock that equals the part of today's nonfinancial income that could not be forecast from all 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 below.

This difference in their information sets manifests itself in their responding differently to what they regard as time $t$ information shocks.

Thus, while they receive exactly the same histories of nonfinancial income, our two consumers receive different **shocks** or **news** about their **future** nonfinancial incomes.

We compare behaviors of our two consumers as a way to learn about

- operating characteristics of the linear-quadratic permanent income model

- how the Kalman filter introduced in this lecture and/or the theory of optimal forecasting introduced in *this lecture* embody lessons that can be applied to the **news** and **noise** literature
- various ways of representing and computing optimal decision rules in the linear-quadratic permanent income model
- a **Ricardian equivalence** outcome describing 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 some of the **invertibility** issues that take center stage in the analysis of **fiscal foresight** by Eric Leeper, Todd Walker, and Susan Yang [LWY13].

## 5.2 Two Representations of the Same Nonfinancial Income Process

Where $\beta \in (0,1)$, we study consequences of endowing a consumer with one of the two alternative representations for the change in the consumer's nonfinancial income $y_{t+1} - y_t$.

The first representation, which we shall refer to as the **original 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 $\sigma_\epsilon^2$.

This representation of the process is used by a consumer who at time $t$ knows both $y_t$ and the original shock $\epsilon_t$ and can use both of them to forecast future $y_{t+j}$'s.

Furthermore, as we'll see below, representation (5.1) has the peculiar property that a positive shock $\epsilon_{t+1}$ 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 two 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 $\beta^{-2}$.

The second representation is the **innovations representation** from 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 $j$th 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 the coefficient $g_0 = \beta^{-2} \sigma_\epsilon^2$ and that $g_j = 0$ for $j \neq 0$.

Alternatively, if you are uncomfortable with covariance generating functions, note that we can directly calculate $\sigma_a^2$ from formula (5.3) according to

$$\sigma_a^2 = \sigma_\epsilon^2 + [1 + (\beta - \beta^{-1})^2 \sum_{j=0}^{\infty} \beta^{2j}] = \beta^{-1} \sigma_\epsilon^2.$$

### 5.2.1 Application of Kalman filter

We can also obtain representation (5.2) from representation (5.1) by using the **Kalman filter**.

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

Compute the steady-state Kalman filter for this system and let $K$ be the steady-state gain and $a_{t+1}$ the one-step ahead innovation.

The innovations representation is

$$\begin{aligned} \hat{\epsilon}_{t+1} &= 0 \hat{\epsilon}_t + K a_{t+1} \\ z_{t+1} &= -\beta a_t + a_{t+1} \end{aligned}$$

By applying formulas for the steady-state Kalman filter, by hand we computed that $K = \beta^2, \sigma_a^2 = \beta^{-2} \sigma_\epsilon^2 = \beta^{-2}$, and $\Sigma = (1 - \beta^2) \sigma_\epsilon^2$.

We can also obtain these formulas via the classical filtering theory described in *this lecture*.

### 5.2.2 News Shocks and Less Informative Shocks

Representation (5.1) is cast in terms of a **news shock** $\epsilon_{t+1}$ that represents a shock to nonfinancial income coming from taxes, transfers, and other random sources of income changes known to a well-informed person having 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 $\epsilon_t$.

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 contained solely 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 $y_t, y_{t-1}, \ldots$.

The shock $\epsilon_t$ is **not fundamental** and 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** $\epsilon_t$ contains more information about future $y$'s than is contained in the semi-infinite history $y^t = [y_t, y_{t-1}, \ldots]$ of current and past $y$'s.

Staring at representation (5.3) for $a_{t+1}$ shows that it consists both of **new news** $\epsilon_{t+1}$ as well as a long moving average $(\beta - \beta^{-1}) \sum_{j=0}^{\infty} \beta^j \epsilon_{t-j}$ of **old news**.

The **better informed** representation (5.1) asserts that a shock $\epsilon_t$ results in an impulse response to nonfinancial income of $\epsilon_t$ times the sequence

$$1, 1 - \beta^{-1}, 1 - \beta^{-1}, \ldots$$

so that a shock that **increases** nonfinancial income $y_t$ by $\epsilon_t$ at time $t$ is followed by an **increase** in future $y$ of $\epsilon_t$ times $1 - \beta^{-1} < 0$ in **all** subsequent periods.

Because $1 - \beta^{-1} < 0$, this means that a positive shock of $\epsilon_t$ today raises income at time $t$ by $\epsilon_t$ and then **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 $\epsilon_t$ 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 $\beta^{-1}$.
- In each future period, the government **rolls over** the one-period bond and so continues to borrow $\epsilon_t$ 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 successors created by the roll-over operation.
- In all future periods $t+1, t+2, \ldots$, 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 innovation representation, asserts that a shock $a_t$ results in an impulse response to nonfinancial income of $a_t$ times

$$1, 1 - \beta, 1 - \beta, \ldots$$

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 $j = 1, 2, \ldots$.

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.

### 5.2.3 Representation of $\epsilon_t$ in Terms of Future $y$'s

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 eventually obtain

$$\epsilon_t = \beta(y_t - (1-\beta)\sum_{j=0}^{\infty}\beta^j y_{t+j+1})$$

This equation shows that $\epsilon_t$ equals $\beta$ times the one-step-backwards error in optimally **backcasting** $y_t$ based on the **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, $\epsilon_t$ contains **exact** information about an important linear combination of **future** nonfinancial income.

### 5.2.4 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

$$a_{t+1} = y_{t+1} - (1-\beta)\sum_{j=0}^{\infty}\beta^j y_{t-j}$$

and where the information set is $y^t = [y_t, y_{t-1}, ...]$, the one step-ahead optimal prediction is

$$E[y_{t+1}|y^t] = (1-\beta)\sum_{j=0}^{\infty}\beta^j y_{t-j}$$

### 5.2.5 Permanent Income Consumption-Smoothing Model

When we computed optimal consumption-saving policies for the two representations using formulas obtained with the difference equation approach described in the quantecon lecture, we obtain:

**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 the Python programs to be reported below using state-space representations and dynamic programming.

Evidently the two consumers behave differently though they receive exactly the same histories of nonfinancial income.

The consumer with information associated with representation (5.1) responds to each shock $\epsilon_{t+1}$ by leaving his consumption unaltered and **saving** all of $\epsilon_{t+1}$ 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 $\epsilon_{t+1}$ to his time $t+1$ nonfinancial income.

The consumer with information 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.

We can regard the first consumer as someone whose behavior sharply illustrates the behavior assumed in a classic Ricardian equivalence experiment.

## 5.3 State Space Representations

We can cast our two representations in terms of the following two state space systems

$$\begin{bmatrix} y_{t+1} \\ \epsilon_{t+1} \end{bmatrix} = \begin{bmatrix} 1 & -\beta^{-1} \\ 0 & 0 \end{bmatrix} \begin{bmatrix} y_t \\ \epsilon_t \end{bmatrix} + \begin{bmatrix} \sigma_\epsilon \\ \sigma_\epsilon \end{bmatrix} v_{t+1}$$
$$y_t = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} y_t \\ \epsilon_t \end{bmatrix}$$

and

$$\begin{bmatrix} y_{t+1} \\ a_{t+1} \end{bmatrix} = \begin{bmatrix} 1 & -\beta \\ 0 & 0 \end{bmatrix} \begin{bmatrix} y_t \\ a_t \end{bmatrix} + \begin{bmatrix} \sigma_a \\ \sigma_a \end{bmatrix} u_{t+1}$$
$$y_t = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} y_t \\ a_t \end{bmatrix}$$

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 the quantecon lecture.

All the code that we shall use below is presented in that lecture.

### 5.3.1 Computations

We shall use Python to form **both** of the above two state-space representations, using the following parameter values $\sigma_\epsilon = 1, \sigma_a = \beta^{-1}\sigma_\epsilon = \beta^{-1}$ where $\beta$ 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 the 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 following initial conditions $x_0 =$

$$\begin{bmatrix} 10 \\ 0 \end{bmatrix}$$

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 $\epsilon_t$ and $a_t$ for the two representations.
- run simulations of $\{y_t, c_t, b_t\}$ of length $T$ under both of the representations (later I'll give some more details about how we'll run some special versions of these)

We want to solve the LQ problem:

$$\min \sum_{t=0}^{\infty} \beta^t \left(c_t - \gamma\right)^2$$

subject to the 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 $\gamma$ 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
%matplotlib inline
```

```
# 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.])
```

```
# Original representation state transition matrices
ALQ1 = np.array([[1, -R, 0],
                 [0, 0, 0],
```

(continues on next page)

(continued from previous page)

```
                  [-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 $u_t = -Fx_t$ for the control variable.

Take the original representation case 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 of the innovations representation case, 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)
```

In the following we compute the 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([1.99997796e-11, 1.89474007e-11, 1.78947650e-11, 1.68421292e-11,
        1.57894935e-11]),
 array([ 0.        , -1.05263158, -1.05263158, -1.05263158, -1.05263158]))
```

```
plt.title("original 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 0x7f79cc9096a0>
```

The above two impulse response functions show that when the consumer has the information assumed in the original representation, his response to receiving a positive shock of $\epsilon_t$ 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 $\beta^{-1}$

```
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 0x7f798f6002e0>
```

The above impulse responses show that when the consumer has only the information that is assumed to be available under the innovations representation 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 in order to lower $b_{t+1}$ 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, constructed as described below.

```
# 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("original representation")
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7f798f18d9a0>
```

```
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 0x7f798f91fd90>
```

### 5.3.2 Simulating the Income Process and Two Associated Shock Processes

We now describe how we form a **single** $\{y_t\}_{t=0}^T$ realization that we will use to simulate the two different decision rules 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 **same** $\{y_t\}$ 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 the 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.

### 5.3.3 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

$$\begin{aligned}
a_{t+1} &= \beta a_t + \epsilon_{t+1} - \beta^{-1}\epsilon_t \\
&= \beta\left(\beta a_{t-1} + \epsilon_t - \beta^{-1}\epsilon_{t-1}\right) + \epsilon_{t+1} - \beta^{-1}\epsilon_t \\
&= \beta^2 a_{t-1} + \beta\left(\epsilon_t - \beta^{-1}\epsilon_{t-1}\right) + \epsilon_{t+1} - \beta^{-1}\epsilon_t \\
&= \cdots \\
&= \beta^{t+1} a_0 + \sum_{j=0}^{t} \beta^j \left(\epsilon_{t+1-j} - \beta^{-1}\epsilon_{t-j}\right) \\
&= \beta^{t+1} a_0 + \epsilon_{t+1} + \left(\beta - \beta^{-1}\right) \sum_{j=0}^{t-1} \beta^j \epsilon_{t-j} - \beta^{t-1}\epsilon_0.
\end{aligned}$$

We can verify that we recover the same $\{a_t\}$ sequence computed earlier.

### 5.3.4 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 [LWY13] in their analysis of **fiscal foresight**.

CHAPTER

SIX

# CONSUMPTION SMOOTHING WITH COMPLETE AND INCOMPLETE MARKETS

**Contents**


- *Consumption Smoothing with Complete and Incomplete Markets*
  - *Overview*
  - *Background*
  - *Linear State Space Version of Complete Markets Model*
  - *Model 1 (Complete Markets)*
  - *Model 2 (One-Period Risk-Free Debt Only)*


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 [Hal78]

*Complete markets* allow a consumer to buy or sell claims contingent on all possible states of the world.

*Incomplete markets* allow a consumer to buy or sell only a limited set of securities, often only a single risk-free security.

Hall [Hal78] 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 $\beta^{-1}$, where $\beta \in (0, 1)$ is also a consumer's intertemporal discount factor.

This is equivalent to saying that it costs $\beta$ of time $t$ consumption to buy one unit of consumption at time $t + 1$ for sure.

So $\beta$ is the price of a one-period risk-free claim to consumption next period.

We maintain 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 its 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 but distinct 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
%matplotlib inline
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 purchasing or 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 are 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 $\beta^{-1}$.

## 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 $\beta^{-1}$.
- 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 $\beta^{-1}$.

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 $\mu$ and covariance matrix $\Sigma$.

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}
    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
```

(continues on next page)

(continued from previous page)

```
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

$$P_{ij} = \mathbb{P}\{s_{t+1} = j \mid s_t = i\}$$

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 $\beta^{-1}$ 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

$$\sum_j q(j \mid i) = \sum_j \beta P_{ij} = \beta$$

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

$$\beta \frac{u'(c_{t+1})}{u'(c_t)} \mathbb{P}\{s_{t+1} \mid s_t\} = q(s_{t+1} \mid s_t)$$

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

$$b_{t+1}(s_{t+1} = j \mid s_t = i) = b(j), \quad i = 1, 2; \;\; j = 1, 2 \tag{6.8}$$

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

$$\begin{aligned} \bar{c} + b(1) &= y(1) + q(1 \mid 1) b(1) + q(2 \mid 1) b(2) \\ \bar{c} + b(2) &= y(2) + q(1 \mid 2) b(1) + q(2 \mid 2) b(2) \end{aligned} \tag{6.9}$$

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 the 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

$$\bar{c} + b_0 = y(1) + q(1 \mid 1)b(1) + q(2 \mid 1)b(2) \tag{6.10}$$

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

$$y(1) - y(2) + [q(1 \mid 1) - q(1 \mid 2) - 1]b_0 + [q(2 \mid 1) + 1 - q(2 \mid 2)]b(2) = 0, \tag{6.12}$$

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 $j$ satisfied $b(j) = b_0$

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)
        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
```

(continues on next page)

(continued from previous page)

```
    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, :]
    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)
```

(continues on next page)

(continued from previous page)

```
    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 [Hal78]

## 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 $\beta^{-1}$.

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 $\beta$ 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),
                    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 [Bar79] as well as in the complete markets tradition of Lucas and Stokey [LS83].

CHAPTER

SEVEN

# TAX SMOOTHING WITH COMPLETE AND INCOMPLETE MARKETS

**Contents**


- *Tax Smoothing with Complete and Incomplete Markets*
  - *Overview*
  - *Tax Smoothing with Complete Markets*
  - *Returns on State-Contingent Debt*
  - *More Finite Markov Chain Tax-Smoothing Examples*


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 [LS83].
- the other is in the **incomplete markets** tradition of Barro [Bar79].

*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 [Bar79] 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 [Hal78] had assumed an exogenous stochastic process of nonfinancial income and an exogenous gross interest rate on one period risk-free debt that equals $\beta^{-1}$, where $\beta \in (0, 1)$ is also a consumer's intertemporal discount factor.

Barro [Bar79] 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 [LS83].

### 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) [Gal37] seems to have prescribed policies that come from Barro's model [Bar79]

Let's start with some standard imports:

```
import numpy as np
import quantecon as qe
import matplotlib.pyplot as plt
%matplotlib inline
import scipy.linalg as la
```

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,
```

(continues on next page)

(continued from previous page)

```
                 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)
        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
```

(continues on next page)

(continued from previous page)

```
    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, :]
    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 [LS83], 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 $j = 1, \dots, N$ 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

$$R^T(s_{t+T}, s_{t+T-1}, \dots, s_t) \equiv R(s_{t+1}|s_t) R(s_{t+2}|s_{t+1}) \cdots R(s_{t+T}|s_{t+T-1})$$

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:* 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)
        # if the path of states is specified
        else:
            self.cp = ConsumptionProblem(β, g, b0, P, init=s_path[0])
            self.s_path = s_path
```

(continues on next page)

(continued from previous page)

```
        # 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)[0]
        c1 = line1.get_color()
        ax.set_xlabel('Periods')
        ax.set_ylabel('Cumulative return', color=c1)

        ax_ = ax.twinx()
        ax_._get_lines.prop_cycler = ax._get_lines.prop_cycler
        line2 = ax_.plot(np.arange(N), self.g_path, ls='--')[0]
        c2 = line2.get_color()
        ax_.set_ylabel('Government expenditures', color=c2)

        plt.show()

        # plot detailed information
        Q = self.cp.β * self.cp.P
```

(continues on next page)

(continued from previous page)

```
        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
```

(continues on next page)

(continued from previous page)

```
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
incomplete market
complete market
govt expenditures
value of debts today
0
10
20
30
40
50
60
70
80
Periods

```
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 = 1.8845674740484437

Total government spending in peace, war, postwar
  peace = 1.754809688581315
  war = 4.825743944636679
  postwar = 1.8845674740484437

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
  π(war|war) = 1.1574074074074074
  π(postwar|war) = 10.416666666666666
  π(postwar|postwar) = 1.0416666666666667
```

(continues on next page)

(continued from previous page)

```
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
0.17908622141460384
```

```
# 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()
```

```
P
 [[0.9 0.1 0. ]
 [0.  0.9 0.1]
 [0.  0.  1. ]]
Q
```

(continues on next page)

(continued from previous page)

```
 [[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 = 1.8845674740484437

Total government spending in peace, war, postwar
  peace = 1.754809688581315
  war = 4.825743944636679
  postwar = 1.8845674740484437

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
  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
0.9045311615620274
```

### 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]])
```

(continues on next page)

(continued from previous page)

```
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()
```

```
P
 [[1.  0.  0. ]
 [0.  0.9 0.1]
 [0.1 0.  0.9]]
Q
```

(continues on next page)

(continued from previous page)

```
 [[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 = -2.027889273356399
  T+b in temporary peace = 1.6053287197231834
  T+b in war = 3.1191695501730106

Total government spending in peace, temporary peace, war
  peace = -2.027889273356399
  temporary peace = 1.6053287197231834
  war = 3.119169550173011

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
  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
-9.368991732594212
```

### 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],
```

(continues on next page)

(continued from previous page)

```
                    [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()
```

```
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]]
```

(continues on next page)

(continued from previous page)

```
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 = 5.1281293302540405

Total government spending in peace1, peace2, war1, war2
  peace1 = 1.6927944572748268
  peace2 = 4.117736720554273
  war1 = 7.553071593533487
  war2 = 5.1281293302540405

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(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
0.02371440178864223
```

### 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()
```

```
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]
```

(continues on next page)

(continued from previous page)

```
 [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 = 2.407109110283644
  T+b in permanent peace = -2.739949713245767

Total government spending in peace1, peace2, war1, war2, permanent peace
  peace1 = 1.6349979047185736
  peace2 = 3.457892745537051
  war1 = 6.040327103363228
  war2 = 2.407109110283644
  permanent peace = -2.739949713245767

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
-11.132109773063592
```

### 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],
                  [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()
```

Tax collection paths
1.2
1.1
1.0
0.9
0.8
0.7
0.6
0.5
0
1
2
3
4
5
6
Periods
incomplete market
complete market
govt expenditures

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.   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 = 0.5571895472128002
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 = 1.6584286588928006
  T+b in war1 = 1.7638860668928005
  T+b in war2 = 1.1445708668928007
  T+b in war3 = 0.4994508668928011
  T+b in permanent peace = -0.1725491331071991
```

(continues on next page)

(continued from previous page)

```
Total government spending in peace1, peace2, war1, war2, war3, permanent peace
  peace1 = 1.5571895472128003
  peace2 = 1.6584286588928003
  war1 = 1.7638860668928005
  war2 = 1.1445708668928007
  war3 = 0.4994508668928006
  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
1.2775343959060064
```

### 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 [Bar79].

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

**Contents**


- *Markov Jump Linear Quadratic Dynamic Programming*
  - *Overview*
  - *Review of useful LQ dynamic programming formulas*
  - *Linked Riccati equations for Markov LQ dynamic programming*
  - *Applications*
  - *Example 1*
  - *Example 2*
  - *More examples*


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 [DVGC99] and the references cited there.

The method has been applied to problems in macroeconomics and monetary economics by [SW+08] and [SW09].

The periodic models of seasonality described in chapter 14 of [HS13] 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

$$-\left(x_t' P x_t + \rho\right)$$

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 $\rho$ 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 $s_t \in [1, 2, \dots, N]$ be a time $t$ realization of an $N$-state Markov chain with transition matrix $\Pi$ having typical element $\Pi_{ij}$.

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 $i = 1, \dots, n$ satisfy the $N$ interrelated Bellman equations

$$-x' P_i x - \rho_i = \max_u - \left[ x' R_i x + u' Q_i u + 2 u' W_i x + \beta \sum_j \Pi_{ij} E((A_i x + B_i u + C_i w)' P_j (A_i x + B_i u + C_i w) x + \rho_j) \right]$$

The matrices $P_{s_t} = P_i$ and the scalars $\rho_{s_t} = \rho_i, i = 1, \dots,$ 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 $F_i$ 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 some Python code and a few examples that put the code to work.

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

%matplotlib inline
```

```
# 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 $\Pi$:

$$\Pr\left(s_{t+1} = j \mid s_t = i\right) = \Pi_{ij}$$

Let $x_t = \begin{bmatrix} k_t \\ 1 \end{bmatrix}$

We can represent the one-period payoff function $r\left(s_t, k_t\right)$ and the state-transition law as

$$\begin{aligned} r\left(s_t, k_t\right) &= f_{1,s_t} k_t - f_{2,s_t} k_t^2 - d_{s_t} {u_t}^2 \\ &= -x_t' \underbrace{\begin{bmatrix} f_{2,s_t} & -\frac{f_{1,s_t}}{2} \\ -\frac{f_{1,s_t}}{2} & 0 \end{bmatrix}}_{\equiv R(s_t)} x_t + \underbrace{d_{s_t}}_{\equiv Q(s_t)} {u_t}^2 \end{aligned}$$

$$x_{t+1} = \begin{bmatrix} k_{t+1} \\ 1 \end{bmatrix} = \underbrace{I_2}_{\equiv A(s_t)} x_t + \underbrace{\begin{bmatrix} 1 \\ 0 \end{bmatrix}}_{\equiv B(s_t)} u_t$$

```
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="$\overline{s}_1$ (high)")
ax.plot(k_grid, k_grid + u2_star, label="$\overline{s}_2$ (low)")
```

(continues on next page)

(continued from previous page)

```
# 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], '--')

# 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]

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

(continues on next page)

(continued from previous page)

```
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]])
```

(continues on next page)

(continued from previous page)

```
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 $\lambda$ 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="$\overline{s}_1$", color="b")
    ax.plot(λ_vals, F2[:, i], label="$\overline{s}_2$", color="r")

    ax.set_xlabel("$\lambda$")
    ax.set_ylabel("$F_{s_t}$")
    ax.set_title(f"Coefficient on {state_var}")
    ax.legend()
    plt.show()
```

Notice how the decision rules' constants and slopes behave as functions of $\lambda$.

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 $\Pi$ 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 $\lambda$ and $\delta$ values.

```
λ_vals = np.linspace(0., 1., 10)
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_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("$\lambda$")
    ax.set_ylabel("$\delta$")
    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
    """
```

(continues on next page)

(continued from previous page)

```
    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="$\overline{s}_1$", color="b")
        ax.plot(λ_vals, F2[:, i], label="$\overline{s}_2$", color="r")

        ax.set_xlabel("$\lambda$")
        ax.set_ylabel("$F(\overline{s}_t)$")
        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="$k^*(\overline{s}_"+str(i+1)+")$")
            ax.plot(λ_vals, - F[:, 1] / F[:, 0], color=c,
                    label="$k^{target}(\overline{s}_"+str(i+1)+")$")

        # Plot a vertical line at λ=0.5
        ax.plot([0.5, 0.5], [min(k_star), max(k_star)], "-.")

        ax.set_xlabel("$\lambda$")
        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, "$\lambda=0.5$")
        ax.legend(bbox_to_anchor=(1., 1.))
        plt.show()

    # Asymmetric Π
    print("asymmetric Π case:\n")
```

(continues on next page)

(continued from previous page)

```
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("$\lambda$")
    ax.set_ylabel("$\delta$")
    ax.set_zlabel("$F(\overline{s}_t)$")
    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 $\lambda$.

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 $\lambda$ changes.

```
run(construct_arrays1, {"f1_vals":[0.5, 1.]}, state_vec1)
```

```
symmetric Π case:
```

```
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:
```

```
asymmetric Π case:
```

coefficient on k
0.60
0.58
0.56
0.54
0.52
0.50
$F(\bar{s}_t)$
0.0
0.2
0.4
0.6
0.8
1.0
$\lambda$
0.0
0.2
0.4
0.6
0.8
1.0
$\sigma$

coefficient on constant term
−0.35
−0.40
−0.45
$F(\bar{s}_t)$
0.0
0.2
0.4
0.6
0.8
1.0
$\lambda$
0.0
0.2
0.4
0.6
0.8
1.0
$\sigma$

## 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 $r (s_t, k_t, w_t)$ and the state-transition law 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

$$x_{t+1} = \begin{bmatrix} k_{t+1} \\ 1 \\ w_{t+1} \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & \alpha_0 (s_t) & \rho (s_t) \end{bmatrix}}_{\equiv A(s_t)} x_t + \underbrace{\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}}_{\equiv B(s_t)} u_t + \underbrace{\begin{bmatrix} 0 \\ 0 \\ \sigma (s_t) \end{bmatrix}}_{\equiv C(s_t)} \epsilon_{t+1}$$

```
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:
```

```
asymmetric Π case:
```

coefficient on k
0.75
0.70
0.65
0.60
$F(\bar{s}_t)$
0.0
0.2
0.4
0.6
0.8
1.0
$\lambda$
0.0
0.2
0.4
0.6
0.8
1.0
$\sigma$

coefficient on constant term
0.18
0.16
0.14
0.12
0.10
$F(\bar{s}_t)$
0.0
0.2
0.4
0.6
0.8
1.0
$\lambda$
0.0
0.2
0.4
0.6
0.8
1.0
$\sigma$

Only $f_{1,s_t}$ depends on $s_t$.

```
run(construct_arrays2, {"f1_vals":[0.5, 1.]}, state_vec2)
```

```
symmetric Π case:
```

```
asymmetric Π case:
```

coefficient on k
0.63
0.62
0.61
0.60
0.59
0.58
$F(\tilde{S}_t)$
0.0
0.2
0.4
0.6
0.8
1.0
$\lambda$
0.0
0.2
0.4
0.6
0.8
1.0
$\theta$

coefficient on constant term
0.300
0.275
0.250
0.225
0.200
0.175
$F(\tilde{S}_t)$
0.0
0.2
0.4
0.6
0.8
1.0
$\lambda$
0.0
0.2
0.4
0.6
0.8
1.0
$\theta$

Only $f_{2,s_t}$ depends on $s_t$.

```
run(construct_arrays2, {"f2_vals":[0.5, 1.]}, state_vec2)
```

```
symmetric Π case:
```

```
asymmetric Π case:
```

coefficient on k
0.60
0.58
0.56
0.54
0.52
0.50
$F(\bar{S}_t)$
0.0
0.2
0.4
0.6
0.8
1.0
$\lambda$
0.0
0.2
0.4
0.6
0.8
1.0
$\sigma$

coefficient on constant term
0.35
0.30
0.25
0.20
$F(\bar{S}_t)$
0.0
0.2
0.4
0.6
0.8
1.0
$\lambda$
0.0
0.2
0.4
0.6
0.8
1.0
$\sigma$

Only $\alpha_0(s_t)$ depends on $s_t$.

```
run(construct_arrays2, {"α0_vals":[0.5, 1.]}, state_vec2)
```

```
symmetric Π case:
```

```
asymmetric Π case:
```

coefficient on k
0.63
0.62
0.61
0.60
0.59
0.58
$F(\bar{S}_t)$
0.0
0.2
0.4
0.6
0.8
1.0
$\lambda$
0.0
0.2
0.4
0.6
0.8
1.0
$\theta$

coefficient on constant term
0.15
0.10
0.05
0.00
−0.05
$F(\bar{S}_t)$
0.0
0.2
0.4
0.6
0.8
1.0
$\lambda$
0.0
0.2
0.4
0.6
0.8
1.0
$\theta$

Only $\rho_{s_t}$ depends on $s_t$.

```
run(construct_arrays2, {"ρ_vals":[0.5, 0.9]}, state_vec2)
```

```
symmetric Π case:
```

```
asymmetric Π case:
```

coefficient on k
0.63
0.62
0.61
0.60
0.59
0.58
$F(\bar{S}_t)$
0.0
0.2
0.4
0.6
0.8
1.0
$\lambda$
0.0
0.2
0.4
0.6
0.8
1.0
$\sigma$

coefficient on constant term
0.14
0.12
0.10
0.08
$F(\bar{S}_t)$
0.0
0.2
0.4
0.6
0.8
1.0
$\lambda$
0.0
0.2
0.4
0.6
0.8
1.0
$\sigma$

Only $\sigma_{s_t}$ depends on $s_t$.

```
run(construct_arrays2, {"σ_vals":[0.5, 1.]}, state_vec2)
```

```
symmetric Π case:
```

```
asymmetric Π case:
```

coefficient on k
0.63
0.62
0.61
0.60
0.59
0.58
$F(\bar{s}_t)$
0.0
0.2
0.4
0.6
0.8
1.0
$\lambda$
0.0
0.2
0.4
0.6
0.8
1.0
$\sigma$

coefficient on constant term
5.5
5.0
4.5
4.0
3.5
3.0
$F(\bar{s}_t)$
1e−13+1.54682591685e−1
0.0
0.2
0.4
0.6
0.8
1.0
$\lambda$
0.0
0.2
0.4
0.6
0.8
1.0
$\sigma$

## 8.7 More examples

The following lectures describe how Markov jump linear quadratic dynamic programming can be used to extend the [Bar79] 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

**Contents**


- *How to Pay for a War: Part 1*
  - *Reader's Guide*
  - *Public Finance Questions*
  - *Barro (1979) Model*
  - *Python Class to Solve Markov Jump Linear Quadratic Control Problems*
  - *Barro Model with a Time-varying Interest Rate*


In addition to what's in Anaconda, this lecture will deploy quantecon:

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

## 9.1 Reader's Guide

Let's start with some standard imports:

```
import quantecon as qe
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

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 [Bar79] 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 later suggested by Barro (1999 [Bar99], 2003 [BM03]).

Barro's original 1979 [Bar79] model is about a government that borrows and lends in order to minimize an intertemporal measure of distortions caused by taxes.

Technical tractability induced Barro [Bar79] 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.

It is by doing these two things that we extend Barro's 1979 [Bar79] model along lines he suggested in Barro (1999 [Bar99], 2003 [BM03]).

Barro (1979) [Bar79] 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 the following measure of tax distortions: $E_0 \sum_{t=0}^{\infty} \beta^t T_t^2$, 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 $\beta^{-1}$, the reciprocal of the factor $\beta$ 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) [Bar99] and Barro (2003) [BM03], our generalizations of Barro's (1979) [Bar79] model 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's 1979 [Bar79] model 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's 1999 [Bar99] and 2003 [BM03] models 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.

## 9.3 Barro (1979) Model

We begin by solving a version of the Barro (1979) [Bar79] model 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, $\beta$ 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.

Evidently, $p_{t,t+1}$ is inversely related to appropriate corresponding gross interest rates on government debt.

In the spirit of Barro (1979) [Bar79], 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 $\beta$)

- 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:

$$T_t = S x_t + M_t u_t$$

It follows that the $(R, Q, W)$ matrices are implicitly defined by:

$$T_t^2 = x_t' S' S x_t + u_t' M_t' M_t u_t + 2 u_t' M_t' S x_t$$

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
```

(continues on next page)

(continued from previous page)

```
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, computing by simulation the Barro model a large number of times:

```
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" for simulating the 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. The simplest 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 also need to specify a transition matrix for the Markov state.

We use:

$$\Pi = \begin{bmatrix} 0.8 & 0.2 \\ 0.2 & 0.8 \end{bmatrix}$$

(so each Markov state is persistent, and there is an equal chance 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, :])
```

(continues on next page)

(continued from previous page)

```
plt.xlabel('Time')
plt.ylabel('Govt Debt')
plt.show()
```

CHAPTER

TEN

# HOW TO PAY FOR A WAR: PART 2

**Contents**


- *How to Pay for a War: Part 2*
  - *An Application of Markov Jump Linear Quadratic Dynamic Programming*
  - *Two example specifications*
  - *One- and Two-period Bonds but No Restructuring*
  - *Mapping into an LQ Markov Jump Problem*
  - *Penalty on Different Issuance Across Maturities*
  - *A Model with Restructuring*
  - *Restructuring as a Markov Jump Linear Quadratic Control Problem*


In addition to what's in Anaconda, this lecture deploys the quantecon library:

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

## 10.1 An Application of Markov Jump Linear Quadratic Dynamic Programming

This is a *sequel to an earlier lecture*.

We use a method introduced in lecture *Markov Jump LQ dynamic programming* to implement suggestions by Barro (1999 [Bar99], 2003 [BM03]) for extending his classic 1979 model of tax smoothing.

Barro's 1979 [Bar79] 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's 1979 [Bar79] model looks a lot like a consumption-smoothing model.

Our generalizations of his 1979 [Bar79] model will also look like souped-up consumption-smoothing models.

Wanting tractability induced Barro in 1979 [Bar79] 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.

Let's start with some standard imports:

```
import quantecon as qe
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

## 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, $\beta$ a discount factor, $b_{t,t+1}$ time $t+1$ goods that the government promises to pay at $t$, $b_{t,t+2}$ time $t+2$ goods that the government promises to pay at time $t$, $G_t$ government purchases, $p_{t,t+1}$ the number of time $t$ goods received per time $t+1$ goods promised, and $p_{t,t+2}$ 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) [Bar79], 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 $\Pi_{ij}$ is the probability that the Markov state moves from state $i$ to state $j$ in one period.

The variables $T_t, b_{t,t+1}, b_{t,t+2}$ are *control* variables chosen at $t$, while the variables $b_{t-1,t}, b_{t-2,t}$ are endogenous state variables inherited from the past at time $t$ and $p_{t,t+1}, p_{t,t+2}$ 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 this 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

$$x_t = \begin{bmatrix} \bar{b}_t \\ z_t \end{bmatrix}$$

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

$$M_t = [-p_{t,t+1} \quad -p_{t,t+2}]$$

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 overall $Q$ matrix for 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*, the `LQMarkov` class can solve Markov jump LQ problems when provided with the $A, B, C, R, Q, W$ matrices for each Markov state.

The function below maps the primitive matrices and parameters from the above two-period model into the matrices that the `LQMarkov` class requires:

```python
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)
    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 Issuance 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:

$$A_{22} = \begin{bmatrix} 1 & 0 \\ \bar{G} & \rho \end{bmatrix} \ , \ C_2 = \begin{bmatrix} 0 \\ \sigma \end{bmatrix} \ , \ U_g = \begin{bmatrix} 0 & 1 \end{bmatrix}$$

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 also need to specify a transition matrix for the Markov state, we use:

$$\Pi = \begin{bmatrix} 0.9 & 0.1 \\ 0.1 & 0.9 \end{bmatrix}$$

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],
```

(continues on next page)

(continued from previous page)

```
                [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, :])
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 an outcome, we now set $c_1 = 0.01$.

This penalty is enough to ensure that the government issues 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]
```

(continues on next page)

(continued from previous page)

```
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
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

This model alters 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. Rearranging the government budget constraint gives:

$$T_t = b_t^{t-1} + \sum_{j=1}^{H-1} p_{t+j}^t b_{t+j}^{t-1} + G_t - \sum_{j=1}^{H} p_{t+j}^t b_{t+j}^t$$

or

$$T_t = \tilde{S}_x \bar{b}_t + (S_s p_t) \cdot (S_x \bar{b}_t) + U_g z_t - p_t \cdot u_t$$

If we want to write this in terms of the full state, we have:

$$T_t = \begin{bmatrix} (\tilde{S}_x + p_t' S_s' S_x) & Ug \end{bmatrix} x_t - p_t' u_t$$

To simplify the notation, let $S_t = \begin{bmatrix} (\tilde{S}_x + p_t' S_s' S_x) & Ug \end{bmatrix}$.

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

$$R_t = S_t' S_t, \qquad Q_t = p_t p_t', \qquad W_t = -p_t S_t$$

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 continue to use this convention also 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

$$A_t = \begin{bmatrix} 0 & 0 \\ 0 & A_{22,t} \end{bmatrix}, \qquad B = \begin{bmatrix} I \\ 0 \end{bmatrix}, \qquad C = \begin{bmatrix} 0 \\ C_{2,t} \end{bmatrix}$$

This completes the mapping into a Markov jump LQ problem.

## 10.7 Restructuring as a Markov Jump Linear Quadratic Control Problem

As with the previous model, we can use a function to map the primitives of the model with restructuring into the matrices that the `LQMarkov` class requires:

```
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
    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 of the model with restructuring, consider this model where $H = 3$.

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:

$$p^1_{t,t+1} = 0.9695 \ \ , \ \ p^1_{t,t+2} = 0.902 \ \ , \ \ p^1_{t,t+3} = 0.8369$$

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 will assume the same transition matrix and $G_t$ process as above

```
# 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')
```

(continues on next page)

(continued from previous page)

```
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()
```

One-period debt issuance share
0.5
0.4
0.3
0.2
0.1
0.0
0
50
100
150
200
250
300
Time

CHAPTER

ELEVEN

# HOW TO PAY FOR A WAR: PART 3

**Contents**


- *How to Pay for a War: Part 3*
  - *Another Application of Markov Jump Linear Quadratic Dynamic Programming*
  - *Roll-Over Risk*
  - *A Dead End*
  - *Better Representation of Roll-Over Risk*


In addition to what's in Anaconda, this lecture deploys the quantecon library:

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

## 11.1 Another Application of Markov Jump Linear Quadratic Dynamic Programming

This is another *sequel to an earlier lecture*.

We again use a method introduced in lecture *Markov Jump LQ dynamic programming* to implement some ideas Barro (1999 [Bar99], 2003 [BM03]) that extend his classic 1979 [Bar79] model of tax smoothing.

Barro's 1979 [Bar79] 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's 1979 [Bar79] model looks a lot like a consumption-smoothing model.

Our generalizations of his 1979 model will also look like souped-up consumption-smoothing models.

In this lecture, we describe a tax-smoothing problem of a government that faces **roll-over risk**.

Let's start with some standard imports:

```
import quantecon as qe
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

## 11.2 Roll-Over Risk

Let $T_t$ denote tax collections, $\beta$ a discount factor, $b_{t,t+1}$ time $t+1$ goods that the government promises to pay at $t$, $G_t$ government purchases, $p_{t+1}^t$ 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

$$T_t + p_{t+1}^t b_{t,t+1} = G_t + b_{t-1,t}$$

$$G_t = U_{g,t} z_t$$

$$z_{t+1} = A_{22,t} z_t + C_{2,t} w_{t+1}$$

where $w_{t+1} \sim N(0, I)$. 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 $p_{t+1}^t$ is an exogenous state variable 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 $p_{t+1}^t$ to vary over time with:

$$p_{t+1}^t = \beta$$

in Markov state 1 and

$$p_{t+1}^t = 0$$

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 (Our Riccati equations will discover this for us!).

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))
```

(continues on next page)

(continued from previous page)

```
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]])
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();
```

This model is simulated below, using the same process for $G_t$ as in *this lecture*.

When $p_{t+1}^t = \beta$ government debt fluctuates around zero.

The spikes in the series for taxation show periods when the government is unable to access financial markets: positive spikes occur when debt is positive, and the government must raise taxes in the current period.

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')
```

(continues on next page)

(continued from previous page)

```
ax2.plot(tax)
ax2.set_title('Taxation')
ax2.set_xlabel('Time')
plt.show()
```

We can adjust the model so that, rather than having debt fluctuate around zero, the government is a debtor in every period we allow it to borrow.

To accomplish this, we simply raise $p_{t+1}^t$ 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 taxes spike up.

Consequently, the government is wary of letting debt get too high, due to the high costs of a "sudden stop".

CHAPTER

TWELVE

# OPTIMAL TAXATION IN AN LQ ECONOMY

**Contents**


- *Optimal Taxation in an LQ Economy*
  - *Overview*
  - *The Ramsey Problem*
  - *Implementation*
  - *Examples*
  - *Exercises*
  - *Solutions*


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 [LS83] 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* [Ram27] 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 [Bar79].

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
%matplotlib inline
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

- $\beta$ 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.
- $\tau_t$ 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 $\beta^t$ 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 $\mu$ 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 $\ell_t$ 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 $\nu$, all of these quantities are expressed in terms of exogenous variables.

To solve for $\nu$, 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 $\nu$.
- With $\nu$ 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 $\nu$ 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 $\nu$.

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
- $v = \text{trace}\,(C'QC)\beta/(1-\beta)$

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 $(h(x^1), \dots, h(x^N))$.
- $(P^t h)[j]$ 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 $\pi_t$

$$\Pi_t := \sum_{s=0}^{t} \pi_t$$

The term $\pi_{t+1}$ 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, $\pi_{t+1}$ 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 [Bar79], 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
                 '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
```

(continues on next page)

(continued from previous page)

```
        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

    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), float(b0)
    else:
        H = Sm.T @ Sm
```

(continues on next page)

(continued from previous page)

```
        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 ν 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:
        ν = 0.5 * (a0 - sqrt(disc)) / a0
    else:
        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)
```

(continues on next page)

(continued from previous page)

```
    # 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.
    """

    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()
```

(continues on next page)

(continued from previous page)

```
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)
    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
# Each column is a state of the world. The rows are [g d b s 1]
```

(continues on next page)

(continued from previous page)

```
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

### 12.5.1 Exercise 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.

## 12.6 Solutions

### 12.6.1 Exercise 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)
```

$\xi_t$
1.015
1.010
1.005
1.000
0.995
0.990
0
10
20
30
40
50
Time

$\Pi_t$
0.2
0.1
0.0
−0.1
−0.2
0
10
20
30
40
50
Time

# Part III

# Multiple Agent Models

CHAPTER

THIRTEEN

# DEFAULT RISK AND INCOME FLUCTUATIONS

**Contents**


- *Default Risk and Income Fluctuations*
  - *Overview*
  - *Structure*
  - *Equilibrium*
  - *Computation*
  - *Results*
  - *Exercises*
  - *Solutions*


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 [Are08] 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
import random

from numba import jit, int64, float64
from numba.experimental import jitclass
%matplotlib inline
```

## 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 $\delta$, 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 $\delta$.

### 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 $\theta$.

## 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 [Are08] 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 value function $v(B, y)$ and price function $q(B', y)$.
2. At each pair $(B, y)$,
   - update the value of defaulting $v_d(y)$.
   - update the value of continuing $v_c(B, y)$.
3. Update the value function $v(B, y)$, the default rule, the implied ex ante default probability, and the price function.
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 Tauchen's quadrature method.

As we have in other places, we will accelerate our code using Numba.

We start by defining the data structure that will help us compile the class (for more information on why we do this, see the lecture on numba.)

```
# Define the data information for the jitclass
arellano_data = [
    ('B', float64[:]), ('P', float64[:, :]), ('y', float64[:]),
    ('β', float64), ('γ', float64), ('r', float64),
    ('ρ', float64), ('η', float64), ('θ', float64),
    ('def_y', float64[:])
]

# Define utility function
@jit(nopython=True)
def u(c, γ):
    return c**(1-γ)/(1-γ)
```

We then define our `jitclass` that will store various parameters and contain the code that can apply the Bellman operators and determine the savings policy given prices and value functions

```
@jitclass(arellano_data)
class Arellano_Economy:
    """
    Arellano 2008 deals with a small open economy whose government
    invests in foreign assets in order to smooth the consumption of
    domestic households. Domestic households receive a stochastic
    path of income.

    Parameters
    ----------
    B : vector(float64)
        A grid for bond holdings
    P : matrix(float64)
        The transition matrix for a country's output
    y : vector(float64)
        The possible output states
```

(continues on next page)

(continued from previous page)

```
    β : float
        Time discounting parameter
    γ : float
        Risk-aversion parameter
    r : float
        int lending rate
    ρ : float
        Persistence in the income process
    η : float
        Standard deviation of the income process
    θ : float
        Probability of re-entering financial markets in each period
    """

    def __init__(
            self, B, P, y,
            β=0.953, γ=2.0, r=0.017,
            ρ=0.945, η=0.025, θ=0.282
        ):

        # Save parameters
        self.B, self.P, self.y = B, P, y
        self.β, self.γ, self.r, = β, γ, r
        self.ρ, self.η, self.θ = ρ, η, θ

        # Compute the mean output
        self.def_y = np.minimum(0.969 * np.mean(y), y)

    def bellman_default(self, iy, EVd, EV):
        """
        The RHS of the Bellman equation when the country is in a
        defaulted state on their debt
        """
        # Unpack certain parameters for simplification
        β, γ, θ = self.β, self.γ, self.θ

        # Compute continuation value
        zero_ind = len(self.B) // 2
        cont_value = θ * EV[iy, zero_ind] + (1 - θ) * EVd[iy]

        return u(self.def_y[iy], γ) + β*cont_value

    def bellman_nondefault(self, iy, iB, q, EV, iB_tp1_star=-1):
        """
        The RHS of the Bellman equation when the country is not in a
        defaulted state on their debt
        """
        # Unpack certain parameters for simplification
        β, γ, θ = self.β, self.γ, self.θ
        B, y = self.B, self.y

        # Compute the RHS of Bellman equation
        if iB_tp1_star < 0:
            iB_tp1_star = self.compute_savings_policy(iy, iB, q, EV)
        c = max(y[iy] - q[iy, iB_tp1_star]*B[iB_tp1_star] + B[iB], 1e-14)
```

(continues on next page)

(continued from previous page)

```
        return u(c, γ) + β*EV[iy, iB_tp1_star]

    def compute_savings_policy(self, iy, iB, q, EV):
        """
        Finds the debt/savings that maximizes the value function
        for a particular state given prices and a value function
        """
        # Unpack certain parameters for simplification
        β, γ, θ = self.β, self.γ, self.θ
        B, y = self.B, self.y

        # Compute the RHS of Bellman equation
        current_max = -1e14
        iB_tp1_star = 0
        for iB_tp1, B_tp1 in enumerate(B):
            c = max(y[iy] - q[iy, iB_tp1]*B[iB_tp1] + B[iB], 1e-14)
            m = u(c, γ) + β*EV[iy, iB_tp1]

            if m > current_max:
                iB_tp1_star = iB_tp1
                current_max = m

        return iB_tp1_star
```

We can now write a function that will use this class to compute the solution to our model

```
@jit(nopython=True)
def solve(model, tol=1e-8, maxiter=10_000):
    """
    Given an Arellano_Economy type, this function computes the optimal
    policy and value functions
    """
    # Unpack certain parameters for simplification
    β, γ, r, θ = model.β, model.γ, model.r, model.θ
    B = np.ascontiguousarray(model.B)
    P, y = np.ascontiguousarray(model.P), np.ascontiguousarray(model.y)
    nB, ny = B.size, y.size

    # Allocate space
    iBstar = np.zeros((ny, nB), int64)
    default_prob = np.zeros((ny, nB))
    default_states = np.zeros((ny, nB))
    q = np.full((ny, nB), 0.95)
    Vd = np.zeros(ny)
    Vc, V, Vupd = np.zeros((ny, nB)), np.zeros((ny, nB)), np.zeros((ny, nB))

    it = 0
    dist = 10.0
    while (it < maxiter) and (dist > tol):

        # Compute expectations used for this iteration
        EV = P@V
        EVd = P@Vd

        for iy in range(ny):
            # Update value function for default state
```

(continues on next page)

(continued from previous page)

```
            Vd[iy] = model.bellman_default(iy, EVd, EV)

            for iB in range(nB):
                # Update value function for non-default state
                iBstar[iy, iB] = model.compute_savings_policy(iy, iB, q, EV)
                Vc[iy, iB] = model.bellman_nondefault(iy, iB, q, EV, iBstar[iy, iB])

        # Once value functions are updated, can combine them to get
        # the full value function
        Vd_compat = np.reshape(np.repeat(Vd, nB), (ny, nB))
        Vupd[:, :] = np.maximum(Vc, Vd_compat)

        # Can also compute default states and update prices
        default_states[:, :] = 1.0 * (Vd_compat > Vc)
        default_prob[:, :] = P @ default_states
        q[:, :] = (1 - default_prob) / (1 + r)

        # Check tolerance etc...
        dist = np.max(np.abs(Vupd - V))
        V[:, :] = Vupd[:, :]
        it += 1

    return V, Vc, Vd, iBstar, default_prob, default_states, q
```

and, finally, we write a function that will allow us to simulate the economy once we have the policy functions

```
def simulate(model, T, default_states, iBstar, q, y_init=None, B_init=None):
    """
    Simulates the Arellano 2008 model of sovereign debt

    Parameters
    ----------
    model: Arellano_Economy
        An instance of the Arellano model with the corresponding parameters
    T: integer
        The number of periods that the model should be simulated
    default_states: array(float64, 2)
        A matrix of 0s and 1s that denotes whether the country was in
        default on their debt in that period (default = 1)
    iBstar: array(float64, 2)
        A matrix which specifies the debt/savings level that a country holds
        during a given state
    q: array(float64, 2)
        A matrix that specifies the price at which a country can borrow/save
        for a given state
    y_init: integer
        Specifies which state the income process should start in
    B_init: integer
        Specifies which state the debt/savings state should start

    Returns
    -------
    y_sim: array(float64, 1)
        A simulation of the country's income
    B_sim: array(float64, 1)
        A simulation of the country's debt/savings
```

(continues on next page)

(continued from previous page)

```
    q_sim: array(float64, 1)
        A simulation of the price required to have an extra unit of
        consumption in the following period
    default_sim: array(bool, 1)
        A simulation of whether the country was in default or not
    """
    # Find index i such that Bgrid[i] is approximately 0
    zero_B_index = np.searchsorted(model.B, 0.0)

    # Set initial conditions
    in_default = False
    max_y_default = 0.969 * np.mean(model.y)
    if y_init == None:
        y_init = np.searchsorted(model.y, model.y.mean())
    if B_init == None:
        B_init = zero_B_index

    # Create Markov chain and simulate income process
    mc = qe.MarkovChain(model.P, model.y)
    y_sim_indices = mc.simulate_indices(T+1, init=y_init)

    # Allocate memory for remaining outputs
    Bi = B_init
    B_sim = np.empty(T)
    y_sim = np.empty(T)
    q_sim = np.empty(T)
    default_sim = np.empty(T, dtype=bool)

    # Perform simulation
    for t in range(T):
        yi = y_sim_indices[t]

        # Fill y/B for today
        if not in_default:
            y_sim[t] = model.y[yi]
        else:
            y_sim[t] = np.minimum(model.y[yi], max_y_default)
        B_sim[t] = model.B[Bi]
        default_sim[t] = in_default

        # Check whether in default and branch depending on that state
        if not in_default:
            if default_states[yi, Bi] > 1e-4:
                in_default=True
                Bi_next = zero_B_index
            else:
                Bi_next = iBstar[yi, Bi]
        else:
            Bi_next = zero_B_index
            if np.random.rand() < model.θ:
                in_default=False

        # Fill in states
        q_sim[t] = q[yi, Bi_next]
        Bi = Bi_next

    return y_sim, B_sim, q_sim, default_sim
```

## 13.5 Results

Let's start by trying to replicate the results obtained in [Are08].

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 coarse (`ny, nB = 21, 251`) in order to match Arrelano's findings.

Here's the same relationships computed on a finer grid (`ny, nB = 51, 551`)

In either case, 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.

Bond price schedule $q(y, B')$
$y_H$
$y_L$
1.0
0.8
0.6
0.4
0.2
0.0
−0.35
−0.30
−0.25
−0.20
−0.15
−0.10
−0.05
0.00
$B'$

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
$B$

The next figure plots value functions and replicates the right hand panel of Figure 4 of [Are08].

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.

## 13.6 Exercises

### 13.6.1 Exercise 1

To the extent that you can, replicate the figures shown above

- Use the parameter values listed as defaults in the `__init__` method of the `Arellano_Economy`.
- The time series will of course vary depending on the shock draws.

output
1.20
1.15
1.10
1.05
1.00
0.95
0.90
0
50
100
150
200
250
time
foreign assets
0.00
−0.02
−0.04
−0.06
−0.08
−0.10
−0.12
−0.14
0
50
100
150
200
250
time
bond price
0.98
0.96
0.94
0.92
0.90
0
50
100
150
200
250
time

## 13.7 Solutions

Compute the value function, policy and equilibrium prices

```
β, γ, r = 0.953, 2.0, 0.017
ρ, η, θ = 0.945, 0.025, 0.282
ny = 21
nB = 251
Bgrid = np.linspace(-0.45, 0.45, nB)
mc = qe.markov.tauchen(ρ, η, 0, 3, ny)
ygrid, P = np.exp(mc.state_values), mc.P

ae = Arellano_Economy(
    Bgrid, P, ygrid, β=β, γ=γ, r=r, ρ=ρ, η=η, θ=θ
)
```

```
V, Vc, Vd, iBstar, default_prob, default_states, q = solve(ae)
```

Compute the bond price schedule as seen in figure 3 of Arellano (2008)

```
# Create "Y High" and "Y Low" values as 5% devs from mean
high, low = np.mean(ae.y) * 1.05, np.mean(ae.y) * .95
iy_high, iy_low = (np.searchsorted(ae.y, 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 in range(nB):
    b = ae.B[i]
    if -0.35 <= b <= 0:  # To match fig 3 of Arellano
        x.append(b)
        q_low.append(q[iy_low, i])
        q_high.append(q[iy_high, i])
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

```
# Create "Y High" and "Y Low" values as 5% devs from mean
high, low = np.mean(ae.y) * 1.05, np.mean(ae.y) * .95
iy_high, iy_low = (np.searchsorted(ae.y, x) for x in (high, low))

fig, ax = plt.subplots(figsize=(10, 6.5))
ax.set_title("Value Functions")
ax.plot(ae.B, V[iy_high], label="$y_H$", lw=2, alpha=0.7)
ax.plot(ae.B, 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(ae.B.min(), ae.B.max())
plt.show()
```

Draw a heat map for default probability

```
xx, yy = ae.B, ae.y
zz = default_prob

# Create figure
fig, ax = plt.subplots(figsize=(10, 6.5))
hm = ax.pcolormesh(xx, yy, zz)
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()
```

```
/tmp/ipykernel_11288/4099469044.py:6: MatplotlibDeprecationWarning: shading='flat'␣
↪when X and Y have the same dimensions as C is deprecated since 3.3.  Either␣
↪specify the corners of the quadrilaterals with X and Y, or pass shading='auto',
↪'nearest' or 'gouraud', or set rcParams['pcolor.shading'].  This will become an␣
↪error two minor releases later.
  hm = ax.pcolormesh(xx, yy, zz)
```

Plot a time series of major variables simulated from the model

```
T = 250

np.random.seed(42)
y_vec, B_vec, q_vec, default_vec = simulate(ae, T, default_states, iBstar, q)

# Pick up default start and end dates
start_end_pairs = []
i = 0
while i < len(default_vec):
    if default_vec[i] == 0:
        i += 1
    else:
        # If we get to here we're in default
        start_default = i
        while i < len(default_vec) and default_vec[i] == 1:
            i += 1
        end_default = i - 1
        start_end_pairs.append((start_default, end_default))

plot_series = (y_vec, B_vec, q_vec)
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
```

(continues on next page)

(continued from previous page)

```
    y_max = s_max + s_range * 0.1
    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
0.98
0.96
0.94
0.92
0.90
0
50
100
150
200
250
time

CHAPTER

FOURTEEN

# GLOBALIZATION AND CYCLES

**Contents**


- *Globalization and Cycles*
  - *Overview*
  - *Key Ideas*
  - *Model*
  - *Simulation*
  - *Exercises*
  - *Solutions*


## 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
%matplotlib inline
import seaborn as sns
```

(continues on next page)

(continued from previous page)

```
from numba import jit, vectorize
from ipywidgets import interact
```

### 14.1.1 Background

The model builds on work by Judd [Jud85], Deneckner and Judd [DJ92] and Helpman and Krugman [HK85] 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 $\Omega_t$ 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 $\nu$ 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 $\nu$ 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 $\psi$ units of labor, so the marginal cost is equal to $\psi$ 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 $\Omega^c$ and $\Omega^m$ 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

$$N_{j,t}^m \geq 0, \quad \pi_{j,t}^m := (p_{j,t}^m - \psi) y_{j,t}^m - f \leq 0 \quad \text{and} \quad \pi_{j,t}^m N_{j,t}^m = 0$$

With further manipulations, this becomes

$$N_{j,t}^m = \theta(M_{j,t} - N_{j,t}^c) \geq 0, \quad \frac{1}{\sigma}\left[\frac{\alpha L_j}{\theta(M_{j,t} + \rho M_{k,t})} + \frac{\alpha L_k}{\theta(M_{j,t} + M_{k,t}/\rho)}\right] \leq f$$

### 14.3.3 Law of Motion

With $\delta$ as the exogenous probability of a variety becoming obsolete, the dynamic equation for the measure of firms becomes

$$N_{j,t+1}^c = \delta(N_{j,t}^c + N_{j,t}^m) = \delta(N_{j,t}^c + \theta(M_{j,t} - N_{j,t}^c))$$

We will work with a normalized measure of varieties

$$n_{j,t} := \frac{\theta \sigma f N_{j,t}^c}{\alpha(L_1 + L_2)}, \quad i_{j,t} := \frac{\theta \sigma f N_{j,t}^m}{\alpha(L_1 + L_2)}, \quad m_{j,t} := \frac{\theta \sigma f M_{j,t}}{\alpha(L_1 + L_2)} = n_{j,t} + \frac{i_{j,t}}{\theta}$$

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
        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))
```

(continues on next page)

(continued from previous page)

```
@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)
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
```

(continues on next page)

(continued from previous page)

```
    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
        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))
```

(continues on next page)

(continued from previous page)

```
    # 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)
        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)
```

(continues on next page)

(continued from previous page)

```
        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)

        # 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
```

(continues on next page)

(continued from previous page)

```
        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]
        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
    """
```

(continues on next page)

(continued from previous page)

```
    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 $\rho$ 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
250
225
200
175
150
125
100
75
50
25
0

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

### 14.5.1 Exercise 1

Replicate the figure *shown above* by coloring initial conditions according to whether or not synchronization occurs from those conditions.

## 14.6 Solutions

```
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",
```

(continues on next page)

(continued from previous page)

```
                x=0.475, y=0.915, size=26)
plt.show()
```

```
/tmp/ipykernel_10795/1064408373.py:9: MatplotlibDeprecationWarning: shading='flat'␣
↪when X and Y have the same dimensions as C is deprecated since 3.3.  Either␣
↪specify the corners of the quadrilaterals with X and Y, or pass shading='auto',
↪'nearest' or 'gouraud', or set rcParams['pcolor.shading'].  This will become an␣
↪error two minor releases later.
  cf = ax.pcolormesh(unitrange, unitrange, ab, cmap="viridis")
```

### 14.6.1 Interactive Version

Additionally, instead of just seeing 4 plots at once, we might want to manually be able to change $\rho$ 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.

```
def interact_attraction_basis(ρ=0.2, maxiter=250, npts=250):
    # Create the figure and axis that we will plot on
```

(continues on next page)

(continued from previous page)

```
    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))
```

```
interactive(children=(FloatSlider(value=0.2, description='ρ', max=1.0, step=0.05),␣
↪IntSlider(value=250, descri…
```

CHAPTER

FIFTEEN

# COASE'S THEORY OF THE FIRM

**Contents**


- *Coase's Theory of the Firm*
  - *Overview*
  - *The Model*
  - *Equilibrium*
  - *Existence, Uniqueness and Computation of Equilibria*
  - *Implementation*
  - *Exercises*
  - *Solutions*


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

```
!pip install interpolation
```

## 15.1 Overview

In 1937, Ronald Coase wrote a brilliant essay on the nature of the firm [Coa37].

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
%matplotlib inline
from scipy.optimize import fminbound
from interpolation import interp
```

### 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…" ([Coa37], 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 [KNS18].

## 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 $\ell_i$.

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 $\ell$ 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 $\delta v$, 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,

- $\ell_i$ 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 $\delta$ 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 [KNS18] 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 [KNS18] 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

$$p^*(s) = \min_{t \leq s} \{c(s-t) + \delta p^*(t)\} \quad \text{for all} \quad s \in [0,1] \tag{15.7}$$

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 [KNS18] it is shown that $t^*$ and $\ell^*$ 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 $p^*(s) = \min_{t \leq s} \{c(s-t) + \delta p^*(t)\}$ and the envelope theorem for derivatives.

A related equation is the first order condition for $p^*(s) = \min_{t \leq s} \{c(s-t) + \delta p^*(t)\}$, 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 $\delta$ 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,
```

(continues on next page)

(continued from previous page)

```
            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 * interp(grid, p, t) + 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}")

    p_func = lambda x: interp(grid, p, x)
    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
```

(continues on next page)

(continued from previous page)

```
    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 $\ell^*$, 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="$\ell^*$")
ax.legend(fontsize=14)
plt.show()
```

Note that downstream firms choose to be larger, a point we return to below.

## 15.6 Exercises

### 15.6.1 Exercise 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 $\delta$ increases? Why?

Check your intuition by computing the number of firms at delta in (1.01, 1.05, 1.1).

### 15.6.2 Exercise 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.

## 15.7 Solutions

### 15.7.1 Exercise 1

```
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
```

### 15.7.2 Exercise 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
```

# Part IV

# Dynamic Linear Economies

CHAPTER

SIXTEEN

# RECURSIVE MODELS OF DYNAMIC LINEAR ECONOMIES

**Contents**


- *Recursive Models of Dynamic Linear Economies*
  - *A Suite of Models*
  - *Econometrics*
  - *Dynamic Demand Curves and Canonical Household Technologies*
  - *Gorman Aggregation and Engel Curves*
  - *Partial Equilibrium*
  - *Equilibrium Investment Under Uncertainty*
  - *A Rosen-Topel Housing Model*
  - *Cattle Cycles*
  - *Models of Occupational Choice and Pay*
  - *Permanent Income Models*
  - *Gorman Heterogeneous Households*
  - *Non-Gorman Heterogeneous Households*


"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

## 16.1 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 [HS13].

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*

### 16.1.1 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 [HS13] 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 [HS13] 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 [HS13] 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.

### 16.1.2 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.

### 16.1.3 Theory and Econometrics

For an application of the [HS13] 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.

### 16.1.4 More Details

A [HS13] 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.

[HS13] 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

### 16.1.5 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 $t = 0, 1, \dots$ .

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 $t = 0, 1, \dots$ .

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 $\mathcal{N}(Ax_t, CC')$.

### 16.1.6 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 $\{x_0, w_1, \dots, w_t\}$.

### 16.1.7 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

$$\begin{aligned} x_t &= Ax_{t-1} + Cw_t \\ &= A^2x_{t-2} + ACw_{t-1} + Cw_t \\ &= \left[\sum_{\tau=0}^{t-1} A^\tau Cw_{t-\tau}\right] + A^t x_0 \end{aligned}$$

Shift forward in time:

$$x_{t+j} = \sum_{s=0}^{j-1} A^s Cw_{t+j-s} + A^j x_t$$

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,

$$E_t(x_{t+j} - E_t x_{t+j})(x_{t+j} - E_t x_{t+j})' = \sum_{k=0}^{j-1} A^k C C' A^{k'} \equiv v_j$$

$v_j$ can be calculated recursively via

$$\begin{aligned} v_1 &= CC' \\ v_j &= CC' + A v_{j-1} A', \quad j \geq 2 \end{aligned}$$

### 16.1.8 Orthogonal Decomposition

To decompose these covariances into parts attributable to the individual components of $w_t$, we let $i_\tau$ be an $N$-dimensional column vector of zeroes except in position $\tau$, where there is a one. Define a matrix $v_{j,\tau}$

$$v_{j,\tau} = \sum_{k=0}^{j-1} A^k C i_\tau i_\tau' C' A'^k.$$

Note that $\sum_{\tau=1}^{N} i_\tau i_\tau' = I$, so that we have

$$\sum_{\tau=1}^{N} v_{j,\tau} = v_j$$

Evidently, the matrices $\{v_{j,\tau}, \tau = 1, \dots, N\}$ give an orthogonal decomposition of the covariance matrix of $j$-step-ahead prediction errors into the parts attributable to each of the components $\tau = 1, \dots, N$.

### 16.1.9 Taste and Technology Shocks

$E(w_t \mid J_{t-1}) = 0$ and $E(w_t w_t' \mid J_{t-1}) = I$ for $t = 1, 2, \dots$

$$b_t = U_b z_t \text{ and } d_t = U_d z_t,$$

$U_b$ and $U_d$ are matrices that select entries of $z_t$. The law of motion for $\{z_t : t = 0, 1, \dots\}$ is

$$z_{t+1} = A_{22} z_t + C_2 w_{t+1} \text{ for } t = 0, 1, \dots$$

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

### 16.1.10 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 $\ell_t$ 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.

### 16.1.11 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

$$\begin{aligned}s_t &= \Lambda h_{t-1} + \Pi c_t \\ h_t &= \Delta_h h_{t-1} + \Theta_h c_t\end{aligned}$$

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 $\Delta_h$ are less than or equal to one.

Below, we'll outline further assumptions that we shall occasionally impose.

### 16.1.12 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

$$\left(\frac{1}{2}\right) E \sum_{t=0}^{\infty} \beta^t [(s_t - b_t) \cdot (s_t - b_t) + \ell_t^2] | J_0, \ 0 < \beta < 1$$

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

$$\Phi_c c_t + \Phi_g g_t + \Phi_i i_t = \Gamma k_{t-1} + d_t$$

$$| \ g_t \ | \leq \ell_t$$

so we'll be looking for specifications of the matrices $\Phi_c, \Phi_g, \Phi_i, \Gamma, \Delta_k, \Theta_k$ that define them.

### 16.1.13 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 $\phi_1$ 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.

### 16.1.14 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 $\Delta_h$ 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**

$$-\Big(\frac{1}{2}\Big)\,E\sum_{t=0}^{\infty}\beta^t\Big[\big(c_t-\lambda(1-\delta_h)\sum_{j=0}^{\infty}\delta_h^j\,c_{t-j-1}-b_t\big)^2+\ell_t^2\Big]\big|J_0$$

$$0<\beta<1 \; , \; 0<\delta_h<1 \; , \; \lambda>0$$

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 $\mu_t$ 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 $\pi_2$ and $\pi_3$ 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

### 16.1.15 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

### 16.1.16 Lagrangian Formulation

Form the Lagrangian

$$\begin{aligned}
\mathcal{L} = -E \sum_{t=0}^{\infty} \beta^t \Bigg[ & \left(\frac{1}{2}\right)[(s_t - b_t) \cdot (s_t - b_t) + g_t \cdot g_t] \\
& + M_t^{d\prime} \cdot (\Phi_c c_t + \Phi_g g_t + \Phi_i i_t - \Gamma k_{t-1} - d_t) \\
& + M_t^{k\prime} \cdot (k_t - \Delta_k k_{t-1} - \Theta_k i_t) \\
& + M_t^{h\prime} \cdot (h_t - \Delta_h h_{t-1} - \Theta_h c_t) \\
& + M_t^{s\prime} \cdot (s_t - \Lambda h_{t-1} - \Pi c_t) \Bigg] \Big| J_0
\end{aligned}$$

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

$$
\begin{aligned}
M_t^s &= b_t - s_t \\
M_t^h &= E\left[\sum_{\tau=1}^{\infty} \beta^\tau (\Delta_h')^{\tau-1} \Lambda' M_{t+\tau}^s \mid J_t\right] \\
M_t^d &= \begin{bmatrix} \Phi_c' \\ \Phi_g' \end{bmatrix}^{-1} \begin{bmatrix} \Theta_h' M_t^h + \Pi' M_t^s \\ -g_t \end{bmatrix} \\
M_t^k &= E\left[\sum_{\tau=1}^{\infty} \beta^\tau (\Delta_k')^{\tau-1} \Gamma' M_{t+\tau}^d \mid J_t\right] \\
M_t^i &= \Theta_k' M_t^k
\end{aligned}
$$

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.

### 16.1.17 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

$$V(x) = x'Px + \rho$$

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

$$-E\sum_{t=0}^{\infty}\beta^t[x_t'Rx_t + u_t'Qu_t + 2u_t'W'x_t],\ 0<\beta<1$$

subject to

$$x_{t+1} = Ax_t + Bu_t + Cw_{t+1},\ t\geq 0$$

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 $Ew_tw_t' = I$, and that $C$ is a matrix conformable to $x$ and $w$.

The optimal value function $V(x)$ satisfies the Bellman equation

$$V(x_t) = \max_{u_t}\Big\{-(x_t'Rx_t + u_t'Qu_t + 2u_t'Wx_t) + \beta E_tV(x_{t+1})\Big\}$$

where maximization is subject to

$$x_{t+1} = Ax_t + Bu_t + Cw_{t+1},\ t\geq 0$$

$$V(x_t) = -x_t'Px_t - \rho$$

$P$ 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

$$V_{j+1}(x_t) = \max_{u_t}\Big\{-(x_t'Rx_t + u_t'Qu_t + 2u_t'Wx_t) + \beta E_tV_j(x_{t+1})\Big\}$$

$$V_j(x_t) = -x_t'P_jx_t - \rho_j$$

where $P_j$ and $\rho_j$ satisfy the equations

$$\begin{aligned}P_{j+1} &= R + \beta A'P_jA - (\beta A'P_jB + W)(Q + \beta B'P_jB)^{-1}(\beta B'P_jA + W')\\ \rho_{j+1} &= \beta\rho_j + \beta \text{ trace } P_jCC'\end{aligned}$$

We can now state the planning problem as a dynamic programming problem

$$\max_{\{u_t, x_{t+1}\}} -E\sum_{t=0}^{\infty}\beta^t[x_t'Rx_t + u_t'Qu_t + 2u_t'W'x_t],\quad 0<\beta<1$$

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.

### 16.1.18 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**.

### 16.1.19 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.

### 16.1.20 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

$$\Phi_c\,c_t+\Phi_g\,g_t+\Phi_i\,i_t=\Gamma k_{t-1}+d_t$$

$$-\ell_t^2+g_t\cdot g_t=0$$

### 16.1.21 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

$$k_t=\Delta_k k_{t-1}+\Theta_k i_t$$

### 16.1.22 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**

$$p_t^0 = [\Pi' M_t^s + \Theta_h' M_t^h]/\mu_0^w = M_t^c/\mu_0^w$$

$$w_t^0 =\mid S_g x_t \mid /\mu_0^w$$

$$r_t^0 = \Gamma' M_t^d/\mu_0^w$$

$$q_t^0 = \Theta_k' M_t^k/\mu_0^w = M_t^i/\mu_0^w$$

$$\alpha_t^0 = M_t^d/\mu_0^w$$

$$v_0 = \Gamma' M_0^d/\mu_0^w + \Delta_k' M_0^k/\mu_0^w$$

**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.

### 16.1.23 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:

$$a_0 = x_0' \, \mu_a \, x_0 + \sigma_a$$

$$\mu_a = \sum_{\tau=0}^{\infty} \beta^\tau \, (A^{o\prime})^\tau \, Z_a \, A^{o\tau}$$

$$\sigma_a = \frac{\beta}{1-\beta} \text{ trace} \left( Z_a \sum_{\tau=0}^{\infty} \beta^\tau \, (A^o)^\tau \, CC' (A^{o\prime})^\tau \right)$$

### 16.1.24 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

$$\begin{aligned} L_t^2 = [\{y_s\}_{s=t}^{\infty} : \; & y_s \text{ is a random variable in } J_s \text{ for } s \geq t \\ \text{and } E \sum_{s=t}^{\infty} \beta^{s-t} \, y_s^2 \mid J_t < +\infty]. \end{aligned}$$

$$\begin{aligned} p_s^t &= M_c x_s / [\bar{e}_j M_c x_t], \qquad s \geq t \\ w_s^t &= \mid S_g x_s \mid / [\bar{e}_j M_c x_t], \quad s \geq t \\ r_s^t &= \Gamma' M_d x_s / [\bar{e}_j M_c x_t], \quad s \geq t \\ q_s^t &= M_i x_s / [\bar{e}_j \, M_c x_t], \qquad s \geq t \\ \alpha_s^t &= M_d x_s / [\bar{e}_j \, M_c x_t], \quad s \geq t \\ v_t &= [\Gamma' M_d + \Delta_k' M_k] x_t / \, [\bar{e}_j \, M_c x_t] \end{aligned}$$

## 16.2 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 [HS13] 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 $E v_t v_t' = R, E w_{t+1} v_s' = 0$ 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:

$$\begin{aligned}\Sigma_{t+1} &= A^o\Sigma_t A^{o\prime} + CC' \\ &\quad - A^o\Sigma_t G'(G\Sigma_t G' + R)^{-1}G\Sigma_t A^{o\prime}\end{aligned}$$

**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**

$$\begin{aligned}\Sigma &= A^o\Sigma A^{o\prime} + CC' \\ &\quad - A^o\Sigma G'(G\Sigma G' + R)^{-1}G\Sigma A^{o\prime} \\ K &= A^o\Sigma_t G'(G\Sigma G' + R)^{-1}\end{aligned}$$

$$\begin{aligned}\hat{x}_{t+1} &= A^o\hat{x}_t + Ka_t \\ y_t &= G\hat{x}_t + a_t\end{aligned}$$

where $Ea_t a_t' \equiv \Omega = G\Sigma G' + R$.

### 16.2.1 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:

$$-.5\sum_{t=0}^T\left\{n_y\ln(2\pi) + \ln|\Omega_t| + a_t'\Omega_t^{-1}a_t\right\}$$

### 16.2.2 Covariance Generating Functions

Autocovariance: $C_x(\tau) = Ex_t x'_{t-\tau}$.

Generating Function: $S_x(z) = \sum_{\tau=-\infty}^{\infty} C_x(\tau) z^\tau, z \in C$.

### 16.2.3 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

### 16.2.4 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**

$$y_t = \sum_{j=1}^{\infty} G(A^o - KG)^{j-1} K y_{t-j} + a_t$$

## 16.3 Dynamic Demand Curves and Canonical Household Technologies

### 16.3.1 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

- $\Pi$ 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 $L_0^2$.

### 16.3.2 Dynamic Demand Functions

$$
\rho_t^0 \equiv \Pi^{-1\prime} \Big[ p_t^0 - \Theta_h' E_t \sum_{\tau=1}^{\infty} \beta^\tau (\Delta_h' - \Lambda' \Pi^{-1\prime} \Theta_h')^{\tau-1} \Lambda' \Pi^{-1\prime} p_{t+\tau}^0 \Big]
$$

$$
\begin{aligned}
s_{i,t} &= \Lambda h_{i,t-1} \\
h_{i,t} &= \Delta_h h_{i,t-1}
\end{aligned}
$$

where $h_{i,-1} = h_{-1}$.

$$
W_0 = E_0 \sum_{t=0}^{\infty} \beta^t (w_t^0 \ell_t + \alpha_t^0 \cdot d_t) + v_0 \cdot k_{-1}
$$

$$
\mu_0^w = \frac{E_0 \sum_{t=0}^{\infty} \beta^t \rho_t^0 \cdot (b_t - s_{i,t}) - W_0}{E_0 \sum_{t=0}^{\infty} \beta^t \rho_t^0 \cdot \rho_t^0}
$$

$$
\begin{aligned}
c_t &= -\Pi^{-1} \Lambda h_{t-1} + \Pi^{-1} b_t - \Pi^{-1} \mu_0^w E_t \{ \Pi'^{-1} - \Pi'^{-1} \Theta_h' \\
&\qquad [I - (\Delta_h' - \Lambda' \Pi'^{-1} \Theta_h') \beta L^{-1}]^{-1} \Lambda' \Pi'^{-1} \beta L^{-1} \} p_t^0 \\
h_t &= \Delta_h h_{t-1} + \Theta_h c_t
\end{aligned}
$$

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 $\mu_0^w$ and wealth $W_0$; and (iii) past values of consumption, as mediated through the state variable $h_{t-1}$.

## 16.4 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 $\mu_0^2$ of individual wealth with a coefficient on $\mu_0^w$ that depends only on prices.

The multiplier $\mu_0^w$ 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.

### 16.4.1 Re-Opened Markets

$$\rho_t^t \equiv \Pi^{-1\prime}\Big[p_t^t - \Theta_h' E_t \sum_{\tau=1}^{\infty} \beta^\tau (\Delta_h' - \Lambda'\Pi^{-1\prime}\Theta_h')^{\tau-1}\Lambda'\Pi^{-1\prime} p_{t+\tau}^t\Big]$$

$$\begin{aligned} s_{i,t} &= \Lambda h_{i,t-1} \\ h_{i,t} &= \Delta_h h_{i,t-1}, \end{aligned}$$

where now $h_{i,t-1} = h_{t-1}$. Define time $t$ wealth $W_t$

$$W_t = E_t \sum_{j=0}^{\infty} \beta^j (w_{t+j}^t \ell_{t+j} + \alpha_{t+j}^t \cdot d_{t+j}) + v_t \cdot k_{t-1}$$

$$\mu_t^w = \frac{E_t \sum_{j=0}^{\infty} \beta^j \rho_{t+j}^t \cdot (b_{t+j} - s_{i,t+j}) - W_t}{E_t \sum_{t=0}^{\infty} \beta^j \rho_{t+j}^t \cdot \rho_{t+j}^t}$$

$$\begin{aligned} c_t = &-\Pi^{-1}\Lambda h_{t-1} + \Pi^{-1} b_t - \Pi^{-1}\mu_t^w E_t\{\Pi'^{-1} - \Pi'^{-1}\Theta_h' \\ &[I - (\Delta_h' - \Lambda'\Pi'^{-1}\Theta_h')\beta L^{-1}]^{-1}\Lambda'\Pi'^{-1}\beta L^{-1}\} p_t^t \\ h_t = &\ \Delta_h h_{t-1} + \Theta_h c_t \end{aligned}$$

### 16.4.2 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 $\mu_t^w$, 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 $\{p_t^t\}$.

### 16.4.3 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.

## 16.5 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:

$$\begin{aligned}c_t &= -\Pi^{-1} \Lambda h_{t-1} + \Pi^{-1} b_t - \Pi^{-1} \mu_0^w E_t \{ \Pi'^{-1} - \Pi'^{-1} \Theta_h' \\ &\qquad [I - (\Delta_h' - \Lambda' \Pi'^{-1} \Theta_h') \beta L^{-1}]^{-1} \Lambda' \Pi'^{-1} \beta L^{-1} \} p_t \\ h_t &= \Delta_h h_{t-1} + \Theta_h c_t\end{aligned}$$

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

$$\begin{aligned}\Phi_c c_t + \Phi_i i_t + \Phi_g g_t &= \Gamma k_{t-1} + d_t \\ k_t &= \Delta_k k_{t-1} + \Theta_k i_t .\end{aligned}$$

## 16.6 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 $\alpha_0, \alpha_1$ are positive parameters.

Assume that $u_t, d_t$ are uncorrelated first-order autoregressive processes.

## 16.7 A Rosen-Topel Housing Model

$$\begin{aligned} R_t &= b_t + \alpha h_t \\ p_t &= E_t \sum_{\tau=0}^{\infty} (\beta \delta_h)^\tau R_{t+\tau} \end{aligned}$$

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 $\delta_h$ 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 $\rho_t^0$ by $\rho_t^0 = \pi^{-1}[p_t - \beta \delta_h E_t p_{t+1}]$.

## 16.8 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*

## 16.9 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

### 16.9.1 Market for Engineers

Ryoo and Rosen's (2004) [RR04] 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 $\lambda_1$ embodies Rosen's demand parameter $\alpha_d$.

- 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*.

### 16.9.2 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 $\alpha_d$ is a $(2 \times 2)$ matrix of demand parameters and $\epsilon_{1t}$ 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:

$$v_{st} = E_t \beta^k \sum_{j=0}^{\infty} (\beta \delta_N)^j w_{st+k+j}$$

$$v_{ut} = E_t \sum_{j=0}^{\infty} (\beta \delta_N)^j w_{ut+j}$$

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

$$v_{ut} = v_{st}$$

## 16.10 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

$$\begin{aligned} \phi_c \cdot c_t + i_t &= \gamma k_{t-1} + e_t \\ k_t &= k_{t-1} + i_t \end{aligned}$$

$$\phi_i i_t - g_t = 0$$

**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

$$\begin{aligned} \mathcal{M}_t^k &= E(\mathcal{M}_{t+1}^k | J_t) \\ \mathcal{M}_t^e &= E(\mathcal{M}_{t+1}^e | J_t) \end{aligned}$$

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) [SHR91] and by Attanasio and Pavoni (2011) [AP11].

## 16.11 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 $j^{\text{th}}$ 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 $\Lambda$, $\Pi$, $\Delta_h$, $\Theta_h$ do not depend on $j$.

This makes everybody's demand system have the form described earlier, with different $\mu_{j0}^w$'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 $\mu_{0j}^w$:

$$\mu_{0j}^w E_0 \sum_{t=0}^{\infty} \beta^t \{\rho_t^0 \cdot \rho_t^0 + (w_t^0 / \mu_{0a}^w) \ell_{at}\} = E_0 \sum_{t=0}^{\infty} \beta^t \{\rho_t^0 \cdot (b_{jt} - s_{jt}^i) - \alpha_t^0 \cdot d_{jt}\} - v_0 k_{j,-1}$$

$$s_{jt} - b_{jt} = \mu_{0j}^w \rho_t^0$$

$$c_{jt} = -\Pi^{-1} \Lambda h_{j,t-1} + \Pi^{-1} s_{jt}$$

$$h_{jt} = (\Delta_h - \Theta_h \Pi^{-1} \Lambda) h_{j,t-1} + \Pi^{-1} \Theta_h s_{jt}$$

Here $h_{j,-1}$ given.

## 16.12 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:

$$-\frac{1}{2}E\sum_{t=0}^{\infty}\beta^t\left[(s_{it}-b_{it})\cdot(s_{it}-b_{it})+\ell_{it}^2\right]\mid J_0$$

$$\begin{aligned}s_{it}&=\Lambda_i h_{it-1}+\Pi_i\, c_{it}\\ h_{it}&=\Delta_{h_i}\, h_{it-1}+\Theta_{h_i}c_{it}\ ,\ i=1,2.\end{aligned}$$

$$b_{it}=U_{bi}z_t$$

$$z_{t+1}=A_{22}z_t+C_2w_{t+1}$$

**Production Technology**

$$\Phi_c(c_{1t}+c_{2t})+\Phi_g g_t+\Phi_i i_t=\Gamma k_{t-1}+d_{1t}+d_{2t}$$

$$k_t=\Delta_k k_{t-1}+\Theta_k i_t$$

$$g_t\cdot g_t=\ell_t^2,\qquad \ell_t=\ell_{1t}+\ell_{2t}$$

$$d_{it}=U_{d_i}z_t,\qquad i=1,2$$

**Pareto Problem:**

$$\begin{aligned}&-\frac{1}{2}\lambda E_0\sum_{t=0}^{\infty}\beta^t[(s_{1t}-b_{1t})\cdot(s_{1t}-b_{1t})+\ell_{1t}^2]\\ &-\frac{1}{2}(1-\lambda)E_0\sum_{t=0}^{\infty}\beta^t[(s_{2t}-b_{2t})\cdot(s_{2t}-b_{2t})+\ell_{2t}^2]\end{aligned}$$

**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

$$\begin{aligned} p_t &= (\mu_{01} \Pi_1^{-1} \Pi_1^{-1\prime} + \mu_{02} \Pi_2^{-1} \Pi_2^{-1\prime})^{-1} (\Pi_1^{-1} b_{1t} + \Pi_2^{-1} b_{2t}) \\ &\quad - (\mu_{01} \Pi_1^{-1} \Pi_1^{-1\prime} + \mu_{02} \Pi_2^{-1} \Pi_2^{-1\prime})^{-1} c_t \end{aligned}$$

**Punchline:** choose $\Pi$ 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 [HS13] where they deploy a

**Secret Weapon:** Another application of the spectral factorization identity.

**Concluding remark:** The [HS13] 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

SEVENTEEN

# GROWTH IN DYNAMIC LINEAR ECONOMIES

**Contents**


- *Growth in Dynamic Linear Economies*
  - *Common Structure*
  - *A Planning Problem*
  - *Example Economies*


This is another member of a suite of lectures that use the quantecon DLE class to instantiate models within the [HS13] 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
%matplotlib inline
from quantecon import LQ, DLE
```

## 17.1 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 $\beta$.

## 17.2 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) [HS13] 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

$$x_{t+1} = A^o x_t + Cw_{t+1}$$

## 17.3 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.

### 17.3.1 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 $\delta_h$ and $\theta_h$ 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 $\phi_1$.

```
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]] [[-0.021]]
```

However, the near-unity endogenous eigenvalue means that these steady state values are of little relevance.

### 17.3.2 Example 2: Altered Growth Condition

We generate our next economy by making two alterations to the parameters of Example 1.

- First, we raise $\phi_1$ 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 $\gamma_1$ 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]))
```

### 17.3.3 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 $\gamma_1$ 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 $\delta_h$ and $\theta_h$ were defined earlier, the only change we need to make from the parameters of Example 1 is to define the new value of $\lambda$.

```
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.

### 17.3.4 Example 3.1: Varying Sensitivity

Next we raise $\lambda$ 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 $\lambda$ 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]))
```

### 17.3.5 Example 3.2: More Impatience

Next let's lower $\beta$ 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 $\beta$, since we now have $\beta(\gamma_1 + \delta_k) < 1$.

Consumption and investment explode downwards, as a lower value of $\beta$ 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

EIGHTEEN

# LUCAS ASSET PRICING USING DLE

**Contents**


- *Lucas Asset Pricing Using DLE*
  - *Asset Pricing Equations*
  - *Asset Pricing Simulations*


This is one of a suite of lectures that use the quantecon DLE class to instantiate models within the [HS13] 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 LQ
from quantecon import DLE
%matplotlib inline
```

We use a linear-quadratic version of an economy that Lucas (1978) [Luc78] used to develop an equilibrium theory of asset prices:

**Preferences**

$$-\frac{1}{2}\mathbb{E}\sum_{t=0}^{\infty}\beta^t[(c_t-b_t)^2+l_t^2]|J_0$$

$$s_t=c_t$$

$$b_t=U_bz_t$$

**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**

$$z_{t+1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0.8 & 0 \\ 0 & 0 & 0.5 \end{bmatrix} z_t + \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} w_{t+1}$$

$$U_b = \begin{bmatrix} 30 & 0 & 0 \end{bmatrix}$$

$$U_d = \begin{bmatrix} 5 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

$$x_0 = \begin{bmatrix} 5 & 150 & 1 & 0 & 0 \end{bmatrix}'$$

## 18.1 Asset Pricing Equations

[HS13] 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.

## 18.2 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],
```

(continues on next page)

(continued from previous page)

```
                [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.        , -0.48584045],
       [-0.48584045,  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.        , -0.66217487],
       [-0.66217487,  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

NINETEEN

# IRFS IN HALL MODELS

**Contents**


- *IRFs in Hall Models*
  - *Example 1: Hall (1978)*
  - *Example 2: Higher Adjustment Costs*
  - *Example 3: Durable Consumption Goods*


This is another member of a suite of lectures that use the quantecon DLE class to instantiate models within the [HS13] 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
%matplotlib inline
from quantecon import LQ
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) [Hal78].

Knowledge of the basic economic environment is assumed.

See the lecture "Growth in Dynamic Linear Economies" for more details.

## 19.1 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 $\delta_h$ and $\theta_h$ 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]])
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.

## 19.2 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,$\phi_1$, 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.]] [[2.12173041e-12]] [[4.2434517e-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.

## 19.3 Example 3: Durable Consumption Goods

We generate our third economy by raising $\phi_1$ 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]])
```

(continues on next page)

(continued from previous page)

```
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

TWENTY

# PERMANENT INCOME MODEL USING THE DLE CLASS

**Contents**


- *Permanent Income Model using the DLE Class*
  - *The Permanent Income Model*


This lecture is part of a suite of lectures that use the quantecon DLE class to instantiate models within the [HS13] 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" [HS13].

We'll also require the following imports

```
import quantecon as qe
import numpy as np
import scipy.linalg as la
import matplotlib.pyplot as plt
%matplotlib inline
from quantecon import DLE

np.set_printoptions(suppress=True, precision=4)
```

## 20.1 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

$$E_0 \sum_{t=0}^{\infty} \beta^t u(c_t) \tag{20.1}$$

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 (20.1) by choosing a consumption, borrowing plan $\{c_t, b_{t+1}\}_{t=0}^{\infty}$ subject to the sequence of budget constraints

$$c_t + b_t = R^{-1} b_{t+1} + y_t, t \geq 0 \tag{20.2}$$

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 (20.2) 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

$$\begin{aligned} z_{t+1} &= A_{22} z_t + C_2 w_{t+1} \\ y_t &= U_y z_t \end{aligned} \tag{20.3}$$

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:

$$E_0 \sum_{t=0}^{\infty} \beta^t b_t^2 < +\infty \tag{20.4}$$

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:

$$y_{t+1} = \alpha + \rho_1 y_t + \rho_2 y_{t-1} + \sigma w_{t+1}$$

### 20.1.1 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 [HS13] 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 $\gamma$ 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]])
```

(continues on next page)

(continued from previous page)

```
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()
```

Consumption
Income

Debt

CHAPTER

TWENTYONE

# ROSEN SCHOOLING MODEL

**Contents**


- *Rosen Schooling Model*
  - *A One-Occupation Model*
  - *Mapping into HS2013 Framework*


This lecture is yet another part of a suite of lectures that use the quantecon DLE class to instantiate models within the [HS13] 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 quantecon import LQ
from collections import namedtuple
from quantecon import DLE
from math import sqrt
%matplotlib inline
```

## 21.1 A One-Occupation Model

Ryoo and Rosen's (2004) [RR04] 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 $v_t$:

$$n_t = \alpha_s v_t + \epsilon_{st}$$

## 21.2 Mapping into HS2013 Framework

We represent this model in the [HS13] 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

### 21.2.1 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 $\Lambda$ is a k+1 x 1 matrix, $\Delta_h$ is a k_1 x k+1 matrix, and $\Theta_h$ 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.

### 21.2.2 Technology

To capture Ryoo and Rosen's [RR04] supply curve, we use the physical technology:

$$c_t = i_t + d_{1t}$$

$$\psi_1 i_t = g_t$$

where $\psi_1$ is inversely proportional to $\alpha_s$.

### 21.2.3 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 $\rho_s$ and $\rho_d$ 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'])
```

### 21.2.4 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]])
```

(continues on next page)

(continued from previous page)

```
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 $\alpha_d$ 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 $\alpha_d$, 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 $\alpha_d$ 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='$\\alpha_d = 0.1$')
ax1.plot(econ2.c_irf,label='$\\alpha_d = 2$')
ax1.legend()
ax1.set_title('Response of $n_t$ to a demand shock')

ax2.plot(econ1.h_irf[:, 0], label='$\\alpha_d = 0.1$')
ax2.plot(econ2.h_irf[:, 0], label='$\\alpha_d = 24$')
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

TWENTYTWO

# CATTLE CYCLES

**Contents**


- *Cattle Cycles*
  - *The Model*
  - *Mapping into HS2013 Framework*


This is another member of a suite of lectures that use the quantecon DLE class to instantiate models within the [HS13] 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) [RMS94].

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 quantecon import LQ
from collections import namedtuple
from quantecon import DLE
from math import sqrt
%matplotlib inline
```

## 22.1 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:

$$\mathbb{E}_0 \sum_{t=0}^{\infty} \beta^t \{p_t c_t - h_t x_t - \gamma_0 h_t(gx_{t-1}) - \gamma_1 h_t(gx_{t-2}) - m_t c_t - \frac{\psi_1}{2} x_t^2 - \frac{\psi_2}{2} x_{t-1}^2 - \frac{\psi_3}{2} x_{t-3}^2 - \frac{\psi_4}{2} c_t^2\}$$

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 $\psi_j$ 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.

## 22.2 Mapping into HS2013 Framework

### 22.2.1 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.

### 22.2.2 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

$$\Phi_c = \begin{bmatrix} 1 \\ f_1 \\ 0 \\ 0 \\ -f_7 \end{bmatrix}, \ \Phi_g = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \ \Phi_i = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \ \Gamma = \begin{bmatrix} 0 & 0 & 0 \\ f_1(1-\delta) & 0 & gf_1 \\ f_3 & 0 & 0 \\ 0 & f_5 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

### 22.2.3 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 $f_1^2 = \frac{\Psi_1}{2}$, $f_2^2 = \frac{\Psi_2}{2}$, $f_3^2 = \frac{\Psi_3}{2}$, $2f_1f_2 = 1$, $2f_3f_4 = \gamma_0 g$, $2f_5f_6 = \gamma_1 g$.

```
# 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 [RMS94]

```
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]])
```

(continues on next page)

(continued from previous page)

```
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)

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]])

ub = np.array([[πh * a0, 0, 0, πh]])
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])
```

```
/tmp/ipykernel_10942/3161597780.py:59: VisibleDeprecationWarning: Creating an␣
↪ndarray from ragged nested sequences (which is a list-or-tuple of lists-or-
↪tuples-or ndarrays with different lengths or shapes) is deprecated. If you meant␣
↪to do this, you must specify 'dtype=object' when creating the ndarray.
  ub = np.array([[πh * a0, 0, 0, πh]])
```

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],
```

(continues on next page)

(continued from previous page)

```
                  [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)
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)
```

[RMS94] 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, [RMS94] 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 $\rho_3$.

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, [RMS94] 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')
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

TWENTYTHREE

# SHOCK NON INVERTIBILITY

**Contents**


- *Shock Non Invertibility*
  - *Overview*
  - *Model*
  - *Code*


## 23.1 Overview

This is another member of a suite of lectures that use the quantecon DLE class to instantiate models within the [HS13] 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 quantecon as qe
import matplotlib.pyplot as plt
from quantecon import LQ
from quantecon import DLE
from math import sqrt
%matplotlib inline
```

This lecture can be viewed as introducing 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.

This situation sets the stage for an econometrician who is unaware of the problem and consequently misinterprets 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 [LWY13] in their analysis of **fiscal foresight**.

A distinct shock-invertibility issue is present in the special LQ consumption smoothing model in *quantecon lecture*.

## 23.2 Model

We consider the following modification of Robert Hall's (1978) model [Hal78] 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.

```
γ_1 = 0.05
γ = np.array([[γ_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]])
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

$$\mathbb{E}\sum_{j=0}^{\infty}\beta^j(c_{t+j} - d_{t+j})|J_t = \beta^{-1}k_{t-1}\,\forall t$$

If we define the moving average representation of $(c_t, c_t - d_t)$ in terms of the $w_t$s to be:

$$\begin{bmatrix} c_t \\ c_t - d_t \end{bmatrix} = \begin{bmatrix} \sigma_1(L) \\ \sigma_2(L) \end{bmatrix} w_t$$

then Hall's model imposes the restriction $\sigma_2(\beta) = [0 \;\; 0]$.

The agent inside this model sees 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 $w_t$.

The econometrician obtains a Wold representation for the process $[c_t, c_t - d_t]$:

$$\begin{bmatrix} c_t \\ c_t - d_t \end{bmatrix} = \begin{bmatrix} \sigma_1^*(L) \\ \sigma_2^*(L) \end{bmatrix} u_t$$

The Appendix of chapter 8 of [HS13] 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 deficit to innovations to agents' information.

Technically, $\sigma_2(\beta) = [0 \;\; 0]$ implies that the history of $u_t$s spans a *smaller* linear space than does the history of $w_t$s.

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 $u_t$ potentially responds belatedly to agents' news $w_t$.

## 23.3 Code

We will construct Figures from Chapter 8 Appendix E of [HS13] 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 deficit to the endowment innovations.

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')
```

(continues on next page)

(continued from previous page)

```
# This is Fig 8.E.2 from p.189 of HS2013

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

TWENTYFOUR

# RISK AND MODEL UNCERTAINTY

## 24.1 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 [Kni21].

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 [HS01].

Constraint and multiplier preferences express aversion to not knowing a unique probabiity distribution that desribes 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
plt.rcParams["figure.figsize"] = (11, 5)
```

(continues on next page)

(continued from previous page)

```
from matplotlib import rc, cm
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
```

## 24.2 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** $I = \{i = 1, \dots, I\}$.
- A (consumption) **plan** is a function $c : I \to \mathbb{R}$.
- $u : \mathbb{R} \to \mathbb{R}$ is a **utility function**.
- $\pi$ 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 $\pi$ 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:

$$\text{ent}(\pi, \hat{\pi}) = \sum_{i=1}^I \hat{\pi}_i \log\left(\frac{\hat{\pi}_i}{\pi_i}\right) = \sum_{i=1}^I \pi_i \left(\frac{\hat{\pi}_i}{\pi_i}\right) \log\left(\frac{\hat{\pi}_i}{\pi_i}\right)$$

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 $\pi$ 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 $\pi$ 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. 24.1: Figure 1

The heat maps in the next two figures vary both $\hat{\pi}_1$ and $\pi_1$.

The following figure plots entropy.

The next figure plots the logarithm of entropy.

```
3.8205752275831846
```

```
/tmp/ipykernel_14914/3759713737.py:2: RuntimeWarning: divide by zero encountered␣
↪in log
  plt.pcolormesh(x, y, np.log(ent_vals_mat.T), shading='gouraud', cmap='seismic')
```

## 24.3 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.

## 24.4 Expected utility

A decision maker is said to have **expected utility preferences** when he ranks plans $c$ by their expected utilities

$$\sum_{i=1}^{I} u(c_i)\pi_i, \tag{24.1}$$

where $u$ is a unique utility function and $\pi$ is a unique probability measure over states.

- A known $\pi$ expresses risk.
- Curvature of $u$ expresses risk aversion.

## 24.5 Constraint preferences

A decision maker is said to have **constraint preferences** when he ranks plans $c$ according to

$$\min_{\{m_i \geq 0\}_{i=1}^{I}} \sum_{i=1}^{I} m_i \pi_i u(c_i) \tag{24.2}$$

subject to

$$\sum_{i=1}^{I} \pi_i m_i \log m_i \leq \eta \tag{24.3}$$

and

$$\sum_{i=1}^{I} \pi_i m_i = 1. \tag{24.4}$$

In (24.3), $\eta \geq 0$ defines an entropy ball of probability distributions $\hat{\pi} = m\pi$ that surround a baseline distribution $\pi$.

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 $\eta$ indicate more apprehension about the baseline probability distribution $\{\pi_i\}_{i=1}^{I}$.

Following [HS01] and [HS08a], we call minimization problem (24.2) subject to (24.3) and(24.4) a **constraint problem**.

To find minimizing probabilities, we form a Lagrangian

$$L = \sum_{i=1}^{I} m_i \pi_i u(c_i) + \tilde{\theta}[\sum_{i=1}^{I} \pi_i m_i \log m_i - \eta] \tag{24.5}$$

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 (24.5) with respect to $\{m_i\}_{i=1}^{I}$ and to maximize it with respect to $\tilde{\theta}$.

The minimizing probability distortions (likelihood ratios) are

$$\tilde{m}_i(c; \tilde{\theta}) = \frac{\exp(-u(c_i)/\tilde{\theta})}{\sum_j \pi_j \exp(-u(c_j)/\tilde{\theta})}. \tag{24.6}$$

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

$$\sum_i \pi_i \frac{\exp(-u(c_i)/\tilde{\theta})}{\sum_j \pi_j \exp(-u(c_j)/\tilde{\theta})} \log\left[\frac{\exp(-u(c_i)/\tilde{\theta})}{\sum_j \pi_j \exp(-u(c_j)/\tilde{\theta})}\right] = \eta \tag{24.7}$$

for $\tilde{\theta} = \tilde{\theta}(c;\eta)$.

For a fixed $\eta$, the $\tilde{\theta}$ that solves equation (24.7) 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 $\eta$.

The **indirect (expected) utility function** under constraint preferences is

$$\sum_{i=1}^I \pi_i \tilde{m}_i(c_i;\eta) u(c_i) = \sum_{i=1}^I \pi_i \left[\frac{\exp(-\tilde{\theta}^{-1}u(c_i))}{\sum_{j=1}^I \exp(-\tilde{\theta}^{-1}u(c_j))\pi_j}\right] u(c_i). \tag{24.8}$$

Entropy evaluated at the minimizing probability distortion (24.6) equals $E\tilde{m}\log\tilde{m}$ or

$$\begin{aligned} & \sum_{i=1}^I \left[\frac{\exp(-\tilde{\theta}^{-1}u(c_i))}{\sum_{j=1}^I \exp(-\tilde{\theta}^{-1}u(c_j))\pi_j}\right] \times \\ & \left\{-\tilde{\theta}^{-1}u(c_i) + \log\left(\sum_{j=1}^I \exp(-\tilde{\theta}^{-1}u(c_j))\pi_j\right)\right\}\pi_i \\ = \quad & -\tilde{\theta}^{-1}\sum_{i=1}^I \pi_i \left[\frac{\exp(-\tilde{\theta}^{-1}u(c_i))}{\sum_{j=1}^I \exp(-\tilde{\theta}^{-1}u(c_j))\pi_j}\right] u(c_i) \\ & + \log\left(\sum_{j=1}^I \exp(-\tilde{\theta}^{-1}u(c_j))\pi_j\right). \end{aligned} \tag{24.9}$$

Expression (24.9) implies that

$$\begin{aligned} -\tilde{\theta}\log\left(\sum_{j=1}^I \exp(-\tilde{\theta}^{-1}u(c_j))\pi_j\right) = & \sum_{i=1}^I \pi_i \left[\frac{\exp(-\tilde{\theta}^{-1}u(c_i))}{\sum_{j=1}^I \exp(-\tilde{\theta}^{-1}u(c_j))\pi_j}\right] u(c_i) \\ & + \tilde{\theta}(c;\eta)\sum_{i=1}^I \log\tilde{m}_i(c;\eta)\tilde{m}_i(c;\eta)\pi_i, \end{aligned} \tag{24.10}$$

where the last term is $\tilde{\theta}$ times the entropy of the worst-case probability distribution.

## 24.6 Multiplier preferences

A decision maker is said to have **multiplier preferences** when he ranks consumption plans $c$ according to

$$\mathsf{T}u(c) \doteq \min_{\{m_i \geq 0\}_{i=1}^I} \sum_{i=1}^I \pi_i m_i [u(c_i) + \theta \log m_i] \tag{24.11}$$

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 $\theta$ express more apprehension about the baseline probability distribution $\pi$.

Following [HS01] and [HS08a], we call the minimization problem on the right side of (24.11) a **multiplier problem**.

The minimizing probability distortion that solves the multiplier problem is

$$\hat{m}_i(c;\theta) = \frac{\exp(-u(c_i)/\theta)}{\sum_j \pi_j \exp(-u(c_j)/\theta)}. \tag{24.12}$$

We can solve

$$\sum_i \pi_i \frac{\exp(-u(c_i)/\theta)}{\sum_j \pi_j \exp(-u(c_j)/\theta)} \log\left[\frac{\exp(-u(c_i)/\theta)}{\sum_j \pi_j \exp(-u(c_j)/\theta)}\right] = \tilde{\eta} \tag{24.13}$$

to find an entropy level $\tilde{\eta}(c;\theta)$ associated with multiplier preferences with penalty parameter $\theta$ and allocation $c$.

For a fixed $\theta$, the $\tilde{\eta}$ that solves equation (24.13) is a function of the consumption plan $c$

The forms of expressions (24.6) and (24.12) are identical, but the Lagrange multiplier $\tilde{\theta}$ appears in (24.6), while the penalty parameter $\theta$ appears in (24.12).

Formulas (24.6) and (24.12) 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 $\theta$ 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

$$-\theta \log\left(\sum_{j=1}^I \exp(-\theta^{-1}u(c_j))\pi_j\right). \tag{24.14}$$

## 24.7 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

$$\mathsf{T}u(c) \doteq -\theta \log \sum_{i=1}^I \pi_i \exp(-u(c_i)/\theta). \tag{24.15}$$

Here $\mathsf{T}u$ in (24.15) is the **risk-sensitivity** operator of [Jac73], [Whi81], and [Whi90].

It defines a **risk-sensitive** preference ordering over plans $c$.

Because it is not linear in utilities $u(c_i)$ and probabilities $\pi_i$, 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 (24.15) as a function of $\pi_1$ for values of $\theta$ of 100 and .6.

For large values of $\theta$, $\mathsf{T}u(c)$ is approximately linear in the probability $\pi_1$, but for lower values of $\theta$, $\mathsf{T}u(c)$ has considerable curvature as a function of $\pi_1$.

Under expected utility, i.e., $\theta = +\infty$, $\mathsf{T}u(c)$ is linear in $\pi_1$, but it is convex as a function of $\pi_1$ 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 $\theta$.

In the left panel, the red line is our tool for computing the mathematical expectation for different values of $\pi$.

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.

### 24.7.1 Digression on moment generating functions

The risk-sensitivity operator $\mathsf{T}$ is intimately connected to a moment generating function.

In particular, a principal constinuent of the $\mathsf{T}$ 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 $\kappa_j$ is the $j$th 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

$$\mathsf{T}u(c) = \mu_u - \frac{1}{2\theta}\sigma_u^2, \tag{24.16}$$

which becomes expected utility $\mu_u$ when $\theta^{-1} = 0$.

The right side of equation (24.16) is a special case of **stochastic differential utility** preferences in which consumption plans are ranked not just by their expected utilities $\mu_u$ but also the variances $\sigma_u^2$ of their expected utilities.

## 24.8 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

$$\sum_i \hat{\pi}_i(c^*)u(c_i) \tag{24.17}$$

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 $\theta$ and $\eta$, respectively.

## 24.9 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 (24.12), lower values of $\theta$ 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 $\eta$ and a given consumption plan.

Thus, to draw the figure, we set the penalty parameter for multiplier preferences $\theta$ 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.

## 24.10 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 $\eta$ that circumscribes an entropy ball around an approximating probability distribution $\pi$, 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 $\theta$ represents an **increase** in ambiguity (or what [Kni21] called uncertainty) about the specification of the baseline approximating model $\{\pi_i\}_{i=1}^{I}$.

Formulas (24.6) assert that the decision maker acts as if he is pessimistic relative to an approximating model $\pi$.

It expresses what [Buc04] [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 (24.19) to be displayed below, the decision maker uses **pessimism** instrumentally to protect himself against model misspecification.

The penalty parameter $\theta$ for multipler preferences or the entropy level $\eta$ 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 $\pi$ for which it is optimal.

A decision rule is said to be **admissible** if it is undominated.

[HS08a] use ex post Bayesian preferences to show that robust decision rules are undominated and therefore admissible.

## 24.11 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 (24.6) 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 (24.12) 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 $\eta$, 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 $\eta$.

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 $\theta^*$ 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 (24.7) as a root finding problem works much better.

But notice that equation (24.7) 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 $\eta$ and a given allocatin $(c_1, c_2)$ off the 45 degree line, by solving equations (24.7) and (24.13), 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 $\eta$ and $\theta$ 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 $\eta$, 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 $\theta$, the implied entropy $\tilde{\eta}$ from equation (24.13) 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 $c_1 = 1 < c_2 = 3$.

## 24.12 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

$$q_i = \pi_i \hat{m}_i u'(\bar{c}_i) = \pi_i \left( \frac{\exp(-u(\bar{c}_i)/\theta)}{\sum_j \pi_j \exp(-u(\bar{c}_j)/\theta)} \right) u'(\bar{c}_i). \tag{24.18}$$

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 $\eta$ and $\theta$ are adjusted according to (24.7) or (24.13) 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 $\theta$ and $\eta$ are adjusted to verify (24.7) or (24.13) 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 [Luc87], [HST99], [Tal00], and [BHS09] used to measure the costs of business cycles must differ.

### 24.12.1 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.

## 24.13 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 $\theta^{-1}$ 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 $\theta$ 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 $\theta$, causing the associated entropies to differ.

**Color bars:**

- First color bar: variation in $\theta$
- Second color bar: variation in utility levels
- Third color bar: variation in entropy levels

```
/tmp/ipykernel_14914/2075626421.py:36: RuntimeWarning: invalid value encountered␣
↪in true_divide
  m =  m_unnormalized / (π * m_unnormalized).sum()
```

```
/tmp/ipykernel_14914/2075626421.py:35: RuntimeWarning: overflow encountered in exp
  m_unnormalized = np.exp(-u(c) / θ)
/tmp/ipykernel_14914/2075626421.py:36: RuntimeWarning: invalid value encountered␣
↪in true_divide
  m =  m_unnormalized / (π * m_unnormalized).sum()
```

## 24.14 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 $\eta$.

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:

$$\sum_{i=1}^{I} m_i \pi_i u(c_i) \geq \mathsf{T}_\theta u(c) - \theta \sum_{i=1}^{I} \pi_i m_i \log m_i. \tag{24.19}$$

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 $\theta$.

Lowering $\theta$ 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 $\eta$ 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 $\breve{\theta} > 0$ associated with each value of entropy less than or equal to $\eta$ recorded on the ordinate axis, i.e., it is $\sum_{i=1}^I \pi_i \breve{m}_i(\breve{\theta}(\eta))u(c_i)$ where $\breve{m}_i(\breve{\theta}(c,\eta)) \propto \exp\left(\frac{u(c_i)}{\breve{\theta}}\right)$.

(Here $\breve{\theta}$ 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 $\eta$ on the ordinate axis.

The straight line depicts the right side of inequality (24.19) for a particular value of the penalty parameter $\theta$.

As noted, when one lowers $\theta$, the intercept $\mathsf{T}_\theta u(c)$ and the absolute value of the slope both decrease.

Thus, as $\theta$ is lowered, $\mathsf{T}_\theta u(c)$ becomes a more conservative estimate of expected utility under the approximating model $\pi$.

However, as $\theta$ is lowered, the robustness bound (24.19) 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 $\theta$ when it is interpreted as a Lagrange multiplier on the entropy constraint in the constraint problem .

This is an application of the envelope theorem.

## 24.15 Why entropy?

Beyond the helpful mathematical fact that it leads directly to convenient exponential twisting formulas (24.6) and (24.12) 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.

### 24.15.1 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.

[AHS03] and [BHS09]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 $\eta$ in constraint preferences. [AHS03] and [BHS09] use detection error probabilities to calibrate reasonable values of $\eta$.

[AHS03] and [HS08a] also use detection error probabilities to calibrate reasonable values of the penalty parameter $\theta$ in multiplier preferences.

For a fixed sample size and a fixed $\theta$, they would calculate the worst-case $\widehat{m}_i(\theta)$, 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 $\theta$.

They then invert this function to calibrate $\theta$ 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 [BHS09] 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 $\mu$ and $\sigma_c$ 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.

### 24.15.2 Axiomatic justifications

Multiplier and constraint preferences are both special cases of what [MMR06] 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 [GS89].

CHAPTER

TWENTYFIVE

# ROBUSTNESS

**Contents**


- *Robustness*
  - *Overview*
  - *The Model*
  - *Constructing More Robust Policies*
  - *Robustness as Outcome of a Two-Person Zero-Sum Game*
  - *The Stochastic Case*
  - *Implementation*
  - *Application*
  - *Appendix*


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

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

## 25.1 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*.

He calls it *approximating* to acknowledge that he doesn't completely trust it.

He fears that outcomes will actually be 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
%matplotlib inline
import quantecon as qe
```

### 25.1.1 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 these *sets* 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.

1e5
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
Value
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
1e6
Entropy

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).

### 25.1.2 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.

### 25.1.3 Other References

Our discussion in this lecture is based on

- [HS00]
- [HS08a]

## 25.2 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

$$\sum_{t=0}^{\infty} \beta^t \{x_t' R x_t + u_t' Q u_t\} \tag{25.1}$$

subject to the linear law of motion

$$x_{t+1} = A x_t + B u_t + C w_{t+1}, \qquad t = 0, 1, 2, ... \tag{25.2}$$

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 "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 $\{w_t\}$.

Soon we'll quantify the quality of a model specification in terms of the maximal size of the expression $\sum_{t=0}^{\infty} \beta^{t+1} w_{t+1}' w_{t+1}$.

## 25.3 Constructing More Robust Policies

If our agent takes $\{w_t\}$ as a given deterministic sequence, then, drawing on intuition from earlier lectures on dynamic programming, we can anticipate Bellman equations such as

$$J_{t-1}(x) = \min_u \{x'Rx + u'Qu + \beta J_t(Ax + Bu + Cw_t)\}$$

(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 the Bellman equation

$$J(x) = \min_u \max_w \{x'Rx + u'Qu + \beta [J(Ax + Bu + Cw) - \theta w'w]\} \tag{25.3}$$

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 $\theta$ controls how much we penalize the maximizing agent for "harming" the minimizing agent.

By raising $\theta$ more and more, we more and more limit the ability of maximizing agent to distort outcomes relative to the approximating model.

So bigger $\theta$ is implicitly associated with smaller distortion sequences $\{w_t\}$.

### 25.3.1 Analyzing the Bellman Equation

So what does $J$ in (25.3) 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

$$\begin{aligned} \max_w \{(Ax + Bu + Cw)'P(Ax + Bu + Cw) - \theta w'w\} \\ = (Ax + Bu)'\mathcal{D}(P)(Ax + Bu) \end{aligned} \tag{25.4}$$

where

$$\mathcal{D}(P) := P + PC(\theta I - C'PC)^{-1}C'P \tag{25.5}$$

and $I$ is a $j \times j$ identity matrix. Substituting this expression for the maximum into (25.3) yields

$$x'Px = \min_u \{x'Rx + u'Qu + \beta (Ax + Bu)'\mathcal{D}(P)(Ax + Bu)\} \tag{25.6}$$

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 (25.6) 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 (25.5) 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 [HS08a]), 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 $\theta$, is $u = -\hat{F}x$ where

$$\hat{F} := (Q + \beta B'\mathcal{D}(\hat{P})B)^{-1}\beta B'\mathcal{D}(\hat{P})A \tag{25.7}$$

We also define

$$\hat{K} := (\theta I - C'\hat{P}C)^{-1}C'\hat{P}(A - B\hat{F}) \tag{25.8}$$

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 (25.4) evaluated at the fixed rule $u = -\hat{F}x$.

Note that $\hat{P}, \hat{F}, \hat{K}$ are all determined by the primitives and $\theta$.

Note also that if $\theta$ is very large, then $\mathcal{D}$ is approximately equal to the identity mapping.

Hence, when $\theta$ is large, $\hat{P}$ and $\hat{F}$ are approximately equal to their standard LQ values.

Furthermore, when $\theta$ is large, $\hat{K}$ is approximately equal to zero.

Conversely, smaller $\theta$ is associated with greater fear of model misspecification and greater concern for robustness.

## 25.4 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.

### 25.4.1 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

$$\sum_{t=1}^{\infty} \beta^t w_t' w_t \leq \eta \tag{25.9}$$

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 (25.1), this value can be written as

$$J_F(x_0, \mathbf{w}) := \sum_{t=0}^{\infty} \beta^t x_t'(R + F'QF)x_t \tag{25.10}$$

where

$$x_{t+1} = (A - BF)x_t + Cw_{t+1} \tag{25.11}$$

and the initial condition $x_0$ is as specified in the left side of (25.10).

Agent 2 chooses $\mathbf{w}$ to maximize agent 1's loss $J_F(x_0, \mathbf{w})$ subject to (25.9).

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 (25.11) and $\theta$ is a Lagrange multiplier on constraint (25.9).

For the moment, let's take $\theta$ 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,

$$\min_{\mathbf{w}} \sum_{t=0}^{\infty} \beta^t \left\{ -x_t'(R + F'QF)x_t + \beta\theta w_{t+1}' w_{t+1} \right\} \tag{25.12}$$

subject to (25.11).

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 $\theta$ to enforce the constraint (25.9), which can be done by choosing $\theta = \theta_\eta$ such that

$$\beta \sum_{t=0}^{\infty} \beta^t x_t' K(F, \theta_\eta)' K(F, \theta_\eta) x_t = \eta \tag{25.13}$$

Here $x_t$ is given by (25.11) — which in this case becomes $x_{t+1} = (A - BF + CK(F, \theta))x_t$.

### 25.4.2 Using Agent 2's Problem to Construct Bounds on the Value Sets

#### The Lower Bound

Define the minimized object on the right side of problem (25.12) 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

$$R_\theta(x_0, F) - \theta \text{ ent} \leq \sum_{t=0}^{\infty} \beta^t \left\{-x_t'(R + F'QF)x_t\right\} \tag{25.14}$$

where

$$\text{ent} := \beta \sum_{t=0}^{\infty} \beta^t w_{t+1}' w_{t+1}$$

The left side of inequality (25.14) 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 (25.14) is attained when

$$\text{ent} = \beta \sum_{t=0}^{\infty} \beta^t x_t' K(F, \theta)' K(F, \theta) x_t \tag{25.15}$$

To construct the *lower bound* on the set of values associated with all perturbations **w** satisfying the entropy constraint (25.9) at a given entropy level, we proceed as follows:

- For a given $\theta$, solve the minimization problem (25.12).
- Compute the minimizer $R_\theta(x_0, F)$ and the associated entropy using (25.15).
- 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 $\theta$ to trace out the lower bound.

---

**Note:** This procedure sweeps out a set of separating hyperplanes indexed by different values for the Lagrange multiplier $\theta$.

---

#### The Upper Bound

To construct an *upper bound* we use a very similar procedure.

We simply replace the *minimization* problem (25.12) with the *maximization* problem

$$V_{\tilde{\theta}}(x_0, F) = \max_{\mathbf{w}} \sum_{t=0}^{\infty} \beta^t \left\{-x_t'(R + F'QF)x_t - \beta\tilde{\theta} w_{t+1}' w_{t+1}\right\} \tag{25.16}$$

where now $\tilde{\theta} > 0$ penalizes the choice of **w** with larger entropy.

(Notice that $\tilde{\theta} = -\theta$ in problem (25.12))

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

$$V_{\tilde{\theta}}(x_0, F) + \tilde{\theta}\,\text{ent} \geq \sum_{t=0}^{\infty} \beta^t \left\{ -x_t'(R + F'QF)x_t \right\} \tag{25.17}$$

where

$$\text{ent} \equiv \beta \sum_{t=0}^{\infty} \beta^t w_{t+1}' w_{t+1}$$

The left side of inequality (25.17) is a straight line with slope $\tilde{\theta}$.

The upper bound on the left side of (25.17) is attained when

$$\text{ent} = \beta \sum_{t=0}^{\infty} \beta^t x_t' K(F, \tilde{\theta})' K(F, \tilde{\theta}) x_t \tag{25.18}$$

To construct the *upper bound* on the set of values associated all perturbations **w** with a given entropy we proceed much as we did for the lower bound

- For a given $\tilde{\theta}$, solve the maximization problem (25.16).
- Compute the maximizer $V_{\tilde{\theta}}(x_0, F)$ and the associated entropy using (25.18).
- 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.

### 25.4.3 Agent 1's Problem

Now we turn to agent 1, who solves

$$\min_{\{u_t\}} \sum_{t=0}^{\infty} \beta^t \left\{ x_t' R x_t + u_t' Q u_t - \beta\theta w_{t+1}' w_{t+1} \right\} \tag{25.19}$$

where $\{w_{t+1}\}$ satisfies $w_{t+1} = K x_t$.

In other words, agent 1 minimizes

$$\sum_{t=0}^{\infty} \beta^t \left\{ x_t'(R - \beta\theta K'K)x_t + u_t' Q u_t \right\} \tag{25.20}$$

subject to

$$x_{t+1} = (A + CK)x_t + Bu_t \tag{25.21}$$

Once again, the expression for the optimal policy can be found here — we denote it by $\tilde{F}$.

### 25.4.4 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 $\Phi$, 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 (25.7) is a fixed point of the mapping $\Phi$.

In particular, for any given $\theta$,

1. $K(\hat{F}, \theta) = \hat{K}$, where $\hat{K}$ is as given in (25.8)
2. $\Phi(\hat{K}) = \hat{F}$

A sketch of the proof is given in *the appendix*.

## 25.5 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 (25.3) with the stochastic analog

$$J(x) = \min_u \max_{\psi \in \mathcal{P}} \left\{ x' R x + u' Q u + \beta \left[ \int J(Ax + Bu + Cw) \, \psi(dw) - \theta D_{KL}(\psi, \phi) \right] \right\} \tag{25.22}$$

Here $\mathcal{P}$ represents the set of all densities on $\mathbb{R}^n$ and $\phi$ is the benchmark distribution $N(0, I)$.

The distribution $\phi$ 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 (25.3), since it discourages large deviations from the benchmark.

### 25.5.1 Solving the Model

The maximization problem in (25.22) 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 $J(x) = x'Px + d$ for some positive definite matrix $P$ and constant real number $d$.

Moreover, it turns out that if $(I - \theta^{-1}C'PC)^{-1}$ is nonsingular, then

$$\begin{aligned} \max_{\psi \in \mathcal{P}} \left\{ \int (Ax + Bu + Cw)'P(Ax + Bu + Cw)\, \psi(dw) - \theta D_{KL}(\psi, \phi) \right\} \\ = (Ax + Bu)'\mathcal{D}(P)(Ax + Bu) + \kappa(\theta, P) \end{aligned} \tag{25.23}$$

where

$$\kappa(\theta, P) := \theta \ln[\det(I - \theta^{-1}C'PC)^{-1}]$$

and the maximizer is the Gaussian distribution

$$\psi = N\left((\theta I - C'PC)^{-1}C'P(Ax + Bu), (I - \theta^{-1}C'PC)^{-1}\right) \tag{25.24}$$

Substituting the expression for the maximum into Bellman equation (25.22) and using $J(x) = x'Px + d$ gives

$$x'Px + d = \min_u \{x'Rx + u'Qu + \beta\,(Ax + Bu)'\mathcal{D}(P)(Ax + Bu) + \beta\,[d + \kappa(\theta, P)]\} \tag{25.25}$$

Since constant terms do not affect minimizers, the solution is the same as (25.6), leading to

$$x'Px + d = x'\mathcal{B}(\mathcal{D}(P))x + \beta\,[d + \kappa(\theta, P)]$$

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 $d = \beta\,[d + \kappa(\theta, P)]$, which is

$$\hat{d} := \frac{\beta}{1 - \beta}\kappa(\theta, P) \tag{25.26}$$

The robust policy in this stochastic case is the minimizer in (25.25), which is once again $u = -\hat{F}x$ for $\hat{F}$ given by (25.7).

Substituting the robust policy into (25.24) 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 (25.8).

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.

### 25.5.2 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 (25.22), 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 $J_F(x) = x' P_F x + d_F$ and applying the same argument used to derive (25.23) we get

$$x' P_F x + d_F = x'(R + F'QF)x + \beta \left[ x'(A - BF)' \mathcal{D}(P_F)(A - BF)x + d_F + \kappa(\theta, P_F) \right]$$

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

$$d_F := \frac{\beta}{1-\beta} \kappa(\theta, P_F) = \frac{\beta}{1-\beta} \theta \ln[\det(I - \theta^{-1} C' P_F C)^{-1}] \tag{25.27}$$

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.

## 25.6 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 (25.7) – (25.8) 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 (25.13)
- `evaluate_F()` computes the loss and entropy associated with a given policy — see *this discussion*

## 25.7 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 $\theta$.

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 = 0.002
ac = (a_0 - c) / 2.0

# Define LQ matrices

R = np.array([[0.,   ac,   0.],
              [ac, -a_1,  0.5],
```

(continues on next page)

(continued from previous page)

```
            [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.
    """

    rlq = qe.robustlq.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, entropy)))


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
```

(continues on next page)

(continued from previous page)

```
    =======
    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
# ------------------------------------------------------------------------ #


# Compute the optimal rule
optimal_lq = qe.lqcontrol.LQ(Q, R, A, B, C, beta=β)
Po, Fo, do = optimal_lq.stationary_values()

# Compute a robust rule given 9
baseline_robust = qe.robustlq.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 9 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()
```

(continues on next page)

(continued from previous page)

```
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)
        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 0x7f3ad0c01490>]
[<matplotlib.lines.Line2D object at 0x7f3ad0c018b0>]
[<matplotlib.lines.Line2D object at 0x7f3ad0c160a0>]
[<matplotlib.lines.Line2D object at 0x7f3ad0c16790>]
```

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?

## 25.8 Appendix

We sketch the proof only of the first claim in *this section*, which is that, for any given $\theta$, $K(\hat{F}, \theta) = \hat{K}$, where $\hat{K}$ is as given in (25.8).

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 (25.7), then

$$K(\hat{F}, \theta) = (\theta I - C'\hat{P}C)^{-1}C'\hat{P}(A - B\hat{F}) \tag{25.28}$$

*Proof:* As a first step, observe that when $F = \hat{F}$, the Bellman equation associated with the LQ problem (25.11) – (25.12) is

$$\tilde{P} = -R - \hat{F}'Q\hat{F} - \beta^2(A - B\hat{F})'\tilde{P}C(\beta\theta I + \beta C'\tilde{P}C)^{-1}C'\tilde{P}(A - B\hat{F}) + \beta(A - B\hat{F})'\tilde{P}(A - B\hat{F}) \tag{25.29}$$

(revisit this discussion if you don't know where (25.29) 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 (25.29).

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 (25.28).

Hence it remains only to show that $-\hat{P}$ solves (25.29), 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

TWENTYSIX

# ROBUST MARKOV PERFECT EQUILIBRIUM

**Contents**


- *Robust Markov Perfect Equilibrium*
  - *Overview*
  - *Linear Markov Perfect Equilibria with Robust Agents*
  - *Application*


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

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

## 26.1 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 [HS08a] 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
%matplotlib inline
```

### 26.1.1 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.

## 26.2 Linear Markov Perfect Equilibria with Robust Agents

As we saw in Markov perfect equilibria, 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.

### 26.2.1 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

$$\sum_{t=t_0}^{t_1-1} \beta^{t-t_0} \{x_t' R_i x_t + u_{it}' Q_i u_{it} + u_{-it}' S_i u_{-it} + 2x_t' W_i u_{it} + 2u_{-it}' M_i u_{it} - \theta_i v_{it}' v_{it}\} \tag{26.1}$$

while thinking that the state evolves according to

$$x_{t+1} = Ax_t + B_1 u_{1t} + B_2 u_{2t} + Cv_{it} \tag{26.2}$$

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.

### 26.2.2 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 (26.1) and (26.2), then player 1's problem becomes minimization-maximization of

$$\sum_{t=t_0}^{t_1-1} \beta^{t-t_0} \{x_t' \Pi_{1t} x_t + u_{1t}' Q_1 u_{1t} + 2u_{1t}' \Gamma_{1t} x_t - \theta_1 v_{1t}' v_{1t}\} \tag{26.3}$$

subject to

$$x_{t+1} = \Lambda_{1t} x_t + B_1 u_{1t} + C v_{1t} \tag{26.4}$$

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

$$\mathcal{D}_1(P) := P + PC(\theta_1 I - C'PC)^{-1} C'P \tag{26.5}$$

The matrix $F_{1t}$ in the policy rule $u_{1t} = -F_{1t} x_t$ that solves agent 1's problem satisfies

$$F_{1t} = (Q_1 + \beta B'_1 \mathcal{D}_1(P_{1t+1}) B_1)^{-1} (\beta B'_1 \mathcal{D}_1(P_{1t+1}) \Lambda_{1t} + \Gamma_{1t}) \tag{26.6}$$

where $P_{1t}$ solves the matrix Riccati difference equation

$$\begin{aligned} P_{1t} = \Pi_{1t} - (\beta B'_1 \mathcal{D}_1(P_{1t+1}) \Lambda_{1t} + \Gamma_{1t})' (Q_1 + \beta B'_1 \mathcal{D}_1(P_{1t+1}) B_1)^{-1} (\beta B'_1 \mathcal{D}_1(P_{1t+1}) \Lambda_{1t} + \Gamma_{1t}) + \\ \beta \Lambda'_{1t} \mathcal{D}_1(P_{1t+1}) \Lambda_{1t} \end{aligned} \tag{26.7}$$

Similarly, the policy that solves player 2's problem is

$$F_{2t} = (Q_2 + \beta B'_2 \mathcal{D}_2(P_{2t+1}) B_2)^{-1} (\beta B'_2 \mathcal{D}_2(P_{2t+1}) \Lambda_{2t} + \Gamma_{2t}) \tag{26.8}$$

where $P_{2t}$ solves

$$\begin{aligned} P_{2t} = \Pi_{2t} - (\beta B'_2 \mathcal{D}_2(P_{2t+1}) \Lambda_{2t} + \Gamma_{2t})' (Q_2 + \beta B'_2 \mathcal{D}_2(P_{2t+1}) B_2)^{-1} (\beta B'_2 \mathcal{D}_2(P_{2t+1}) \Lambda_{2t} + \Gamma_{2t}) + \\ \beta \Lambda'_{2t} \mathcal{D}_2(P_{2t+1}) \Lambda_{2t} \end{aligned} \tag{26.9}$$

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 (26.6), (26.7), (26.8), and (26.9), 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 (26.6) contain $F_{2t}$
- some terms on the right-hand side of (26.8) contain $F_{1t}$

we need to solve these $k_1 + k_2$ equations simultaneously.

### 26.2.3 Key Insight

As in Markov perfect equilibrium, a key insight here is that equations (26.6) and (26.8) 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 (26.7) and (26.9).

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.

### 26.2.4 Worst-case Shocks

For agent $i$ the maximizing or worst-case shock $v_{it}$ is

$$v_{it} = K_{it} x_t$$

where

$$K_{it} = \theta_i^{-1}(I - \theta_i^{-1} C' P_{i,t+1} C)^{-1} C' P_{i,t+1}(A - B_1 F_{it} - B_2 F_{2t})$$

### 26.2.5 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.

### 26.2.6 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.

## 26.3 Application

### 26.3.1 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

$$p = a_0 - a_1(q_1 + q_2) \tag{26.10}$$

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 (26.10) 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:

$$\pi_i = p q_i - \gamma(\hat{q}_i - q_i)^2, \quad \gamma > 0, \tag{26.11}$$

Substituting the inverse demand curve (26.10) into (26.11) lets us express the one-period payoff as

$$\pi_i(q_i, q_{-i}, \hat{q}_i) = a_0 q_i - a_1 q_i^2 - a_1 q_i q_{-i} - \gamma(\hat{q}_i - q_i)^2, \tag{26.12}$$

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 (26.11) 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:

$$x_{t+1} = (A - B_1 F_1 - B_1 F_2) x_t \tag{26.13}$$

### 26.3.2 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)

```
γ = 12.0

# In LQ form
A = np.eye(3)
B1 = np.array([[0.], [1.], [0.]])
B2 = np.array([[0.], [0.], [1.]])


R1 = [[      0.,     -a0 / 2,           0.],
      [-a0 / 2.,          a1,     a1 / 2.],
      [       0,     a1 / 2.,           0.]]

R2 = [[     0.,           0.,      -a0 / 2],
      [     0.,           0.,      a1 / 2.],
      [-a0 / 2,      a1 / 2.,           a1]]

Q1 = Q2 = γ
S1 = S2 = W1 = W2 = M1 = M2 = 0.0

# Solve using QE's nnash function
F1, F2, P1, P2 = qe.nnash(A, B1, B2, R1, R2, Q1,
                          Q2, S1, S2, W1, W2, M1,
                          M2, beta=β)

# Display policies
print("Computed policies for firm 1 and firm 2:\n")
print(f"F1 = {F1}")
print(f"F2 = {F2}")
print("\n")
```

```
Computed policies for firm 1 and firm 2:

F1 = [[-0.66846615  0.29512482  0.07584666]]
F2 = [[-0.66846615  0.07584666  0.29512482]]
```

### 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):

    """
    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
```

### 26.3.3 Some Details

Firm $i$ wants to minimize

$$\sum_{t=t_0}^{t_1-1} \beta^{t-t_0} \left\{ x_t' R_i x_t + u_{it}' Q_i u_{it} + u_{-it}' S_i u_{-it} + 2 x_t' W_i u_{it} + 2 u_{-it}' M_i u_{it} \right\}$$

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]
ax.plot(q2, 'g-', lw=2, alpha=0.75, label='firm 2 MPE output')
```

(continues on next page)

(continued from previous page)

```
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')
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)
```

(continues on next page)

(continued from previous page)

```
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.

([HS08a] discuss how this property of robust decision rules is connected to the concept of *admissibility* in Bayesian statistical decision theory)

# Part VI

# Classic Linear Models

CHAPTER

TWENTYSEVEN

# VON NEUMANN GROWTH MODEL (AND A GENERALIZATION)

**Contents**


- *Von Neumann Growth Model (and a Generalization)*
  - *Notation*
  - *Model Ingredients and Assumptions*
  - *Dynamic Interpretation*
  - *Duality*
  - *Interpretation as Two-player Zero-sum Game*


This lecture uses the class `Neumann` to calculate key objects of a linear growth model of John von Neumann [vN37] that was generalized by Kemeny, Morgenstern and Thompson [KMT56].

Objects of interest are the maximal expansion rate ($\alpha$), the interest factor ($\beta$), the optimal intensities ($x$), and prices ($p$).

In addition to watching how the towering mind of John von Neumann formulated an equilibrium model of price and quantity vectors in balanced growth, this lecture shows how fruitfully to employ the following important tools:

- a zero-sum two-player game
- linear programming
- the Perron-Frobenius theorem

We'll begin with some imports:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.linalg import solve
from scipy.optimize import fsolve, linprog
from textwrap import dedent
%matplotlib inline

np.set_printoptions(precision=2)
```

The code below provides the `Neumann` class

```
class Neumann(object):

    """
    This class describes the Generalized von Neumann growth model as it was
```

(continues on next page)

(continued from previous page)

```
    discussed in Kemeny et al. (1956, ECTA) and Gale (1960, Chapter 9.5):

    Let:
    n ... number of goods
    m ... number of activities
    A ... input matrix is m-by-n
        a_{i,j} - amount of good j consumed by activity i
    B ... output matrix is m-by-n
        b_{i,j} - amount of good j produced by activity i

    x ... intensity vector (m-vector) with non-negative entries
        x'B - the vector of goods produced
        x'A - the vector of goods consumed
    p ... price vector (n-vector) with non-negative entries
        Bp - the revenue vector for every activity
        Ap - the cost of each activity

    Both A and B have non-negative entries. Moreover, we assume that
    (1) Assumption I (every good which is consumed is also produced):
        for all j, b_{.,j} > 0, i.e. at least one entry is strictly positive
    (2) Assumption II (no free lunch):
        for all i, a_{i,.} > 0, i.e. at least one entry is strictly positive

    Parameters
    ----------
    A : array_like or scalar(float)
        Part of the state transition equation.  It should be `n x n`
    B : array_like or scalar(float)
        Part of the state transition equation.  It should be `n x k`
    """

    def __init__(self, A, B):

        self.A, self.B = list(map(self.convert, (A, B)))
        self.m, self.n = self.A.shape

        # Check if (A, B) satisfy the basic assumptions
        assert self.A.shape == self.B.shape, 'The input and output matrices \
              must have the same dimensions!'
        assert (self.A >= 0).all() and (self.B >= 0).all(), 'The input and \
              output matrices must have only non-negative entries!'

        # (1) Check whether Assumption I is satisfied:
        if (np.sum(B, 0) <= 0).any():
            self.AI = False
        else:
            self.AI = True

        # (2) Check whether Assumption II is satisfied:
        if (np.sum(A, 1) <= 0).any():
            self.AII = False
        else:
            self.AII = True

    def __repr__(self):
        return self.__str__()
```

(continues on next page)

(continued from previous page)

```
    def __str__(self):

        me = """
        Generalized von Neumann expanding model:
          - number of goods          : {n}
          - number of activities     : {m}

        Assumptions:
          - AI:  every column of B has a positive entry    : {AI}
          - AII: every row of A has a positive entry       : {AII}

        """
        # Irreducible                                        : {irr}
        return dedent(me.format(n=self.n, m=self.m,
                                AI=self.AI, AII=self.AII))

    def convert(self, x):
        """
        Convert array_like objects (lists of lists, floats, etc.) into
        well-formed 2D NumPy arrays
        """
        return np.atleast_2d(np.asarray(x))


    def bounds(self):
        """
        Calculate the trivial upper and lower bounds for alpha (expansion rate)
        and beta (interest factor). See the proof of Theorem 9.8 in Gale (1960)
        """

        n, m = self.n, self.m
        A, B = self.A, self.B

        f = lambda α: ((B - α * A) @ np.ones((n, 1))).max()
        g = lambda β: (np.ones((1, m)) @ (B - β * A)).min()

        UB = fsolve(f, 1).item()  # Upper bound for α, β
        LB = fsolve(g, 2).item()  # Lower bound for α, β

        return LB, UB


    def zerosum(self, γ, dual=False):
        """
        Given gamma, calculate the value and optimal strategies of a
        two-player zero-sum game given by the matrix

                M(gamma) = B - gamma * A

        Row player maximizing, column player minimizing

        Zero-sum game as an LP (primal --> α)

            max (0', 1) @ (x', v)
            subject to
```

(continues on next page)

(continued from previous page)

```
        [-M', ones(n, 1)] @ (x', v)' <= 0
        (x', v) @ (ones(m, 1), 0) = 1
        (x', v) >= (0', -inf)

    Zero-sum game as an LP (dual --> beta)

        min (0', 1) @ (p', u)
        subject to
        [M, -ones(m, 1)] @ (p', u)' <= 0
        (p', u) @ (ones(n, 1), 0) = 1
        (p', u) >= (0', -inf)

    Outputs:
    --------
    value: scalar
        value of the zero-sum game

    strategy: vector
        if dual = False, it is the intensity vector,
        if dual = True, it is the price vector
    """

    A, B, n, m = self.A, self.B, self.n, self.m
    M = B - γ * A

    if dual == False:
        # Solve the primal LP (for details see the description)
        # (1) Define the problem for v as a maximization (linprog minimizes)
        c = np.hstack([np.zeros(m), -1])

        # (2) Add constraints :
        # ... non-negativity constraints
        bounds = tuple(m * [(0, None)] + [(None, None)])
        # ... inequality constraints
        A_iq = np.hstack([-M.T, np.ones((n, 1))])
        b_iq = np.zeros((n, 1))
        # ... normalization
        A_eq = np.hstack([np.ones(m), 0]).reshape(1, m + 1)
        b_eq = 1

        res = linprog(c, A_ub=A_iq, b_ub=b_iq, A_eq=A_eq, b_eq=b_eq,
                      bounds=bounds)

    else:
        # Solve the dual LP (for details see the description)
        # (1) Define the problem for v as a maximization (linprog minimizes)
        c = np.hstack([np.zeros(n), 1])

        # (2) Add constraints :
        # ... non-negativity constraints
        bounds = tuple(n * [(0, None)] + [(None, None)])
        # ... inequality constraints
        A_iq = np.hstack([M, -np.ones((m, 1))])
        b_iq = np.zeros((m, 1))
        # ... normalization
        A_eq = np.hstack([np.ones(n), 0]).reshape(1, n + 1)
```

(continues on next page)

(continued from previous page)

```
            b_eq = 1

            res = linprog(c, A_ub=A_iq, b_ub=b_iq, A_eq=A_eq, b_eq=b_eq,
                          bounds=bounds)

        if res.status != 0:
            print(res.message)

        # Pull out the required quantities
        value = res.x[-1]
        strategy = res.x[:-1]

        return value, strategy


    def expansion(self, tol=1e-8, maxit=1000):
        """
        The algorithm used here is described in Hamburger-Thompson-Weil
        (1967, ECTA). It is based on a simple bisection argument and utilizes
        the idea that for a given γ (= α or β), the matrix "M = B - γ * A"
        defines a two-player zero-sum game, where the optimal strategies are
        the (normalized) intensity and price vector.

        Outputs:
        --------
        alpha: scalar
            optimal expansion rate
        """

        LB, UB = self.bounds()

        for iter in range(maxit):

            γ = (LB + UB) / 2
            ZS = self.zerosum(γ=γ)
            V = ZS[0]      # value of the game with γ

            if V >= 0:
                LB = γ
            else:
                UB = γ

            if abs(UB - LB) < tol:
                γ = (UB + LB) / 2
                x = self.zerosum(γ=γ)[1]
                p = self.zerosum(γ=γ, dual=True)[1]
                break

        return γ, x, p

    def interest(self, tol=1e-8, maxit=1000):
        """
        The algorithm used here is described in Hamburger-Thompson-Weil
        (1967, ECTA). It is based on a simple bisection argument and utilizes
        the idea that for a given gamma (= alpha or beta),
        the matrix "M = B - γ * A" defines a two-player zero-sum game,
```

(continues on next page)

(continued from previous page)

```
        where the optimal strategies are the (normalized) intensity and price
        vector

        Outputs:
        --------
        beta: scalar
            optimal interest rate
        """

        LB, UB = self.bounds()

        for iter in range(maxit):
            y = (LB + UB) / 2
            ZS = self.zerosum(y=y, dual=True)
            V = ZS[0]

            if V > 0:
                LB = y
            else:
                UB = y

            if abs(UB - LB) < tol:
                y = (UB + LB) / 2
                p = self.zerosum(y=y, dual=True)[1]
                x = self.zerosum(y=y)[1]
                break

        return y, x, p
```

## 27.1 Notation

We use the following notation.

$\mathbf{0}$ denotes a vector of zeros.

We call an $n$-vector positive and write $x \gg \mathbf{0}$ if $x_i > 0$ for all $i = 1, 2, \dots, n$.

We call a vector non-negative and write $x \geq \mathbf{0}$ if $x_i \geq 0$ for all $i = 1, 2, \dots, n$.

We call a vector semi-positive and written $x > \mathbf{0}$ if $x \geq \mathbf{0}$ and $x \neq \mathbf{0}$.

For two conformable vectors $x$ and $y$, $x \gg y$, $x \geq y$ and $x > y$ mean $x - y \gg \mathbf{0}$, $x - y \geq \mathbf{0}$, and $x - y > \mathbf{0}$, respectively.

We let all vectors in this lecture be column vectors; $x^T$ denotes the transpose of $x$ (i.e., a row vector).

Let $\iota_n$ denote a column vector composed of $n$ ones, i.e. $\iota_n = (1, 1, \dots, 1)^T$.

Let $e^i$ denote a vector (of arbitrary size) containing zeros except for the $i$ th position where it is one.

We denote matrices by capital letters. For an arbitrary matrix $A$, $a_{i,j}$ represents the entry in its $i$ th row and $j$ th column.

$a_{\cdot j}$ and $a_{i \cdot}$ denote the $j$ th column and $i$ th row of $A$, respectively.

## 27.2 Model Ingredients and Assumptions

A pair $(A, B)$ of $m \times n$ non-negative matrices defines an economy.

- $m$ is the number of *activities* (or sectors)
- $n$ is the number of *goods* (produced and/or consumed).
- $A$ is called the *input matrix*; $a_{i,j}$ denotes the amount of good $j$ consumed by activity $i$
- $B$ is called the *output matrix*; $b_{i,j}$ represents the amount of good $j$ produced by activity $i$

Two key assumptions restrict economy $(A, B)$:

- **Assumption I:** (every good that is consumed is also produced)

$$b_{.,j} > \mathbf{0} \quad \forall j = 1, 2, \ldots, n$$

- **Assumption II:** (no free lunch)

$$a_{i,.} > \mathbf{0} \quad \forall i = 1, 2, \ldots, m$$

A semi-positive *intensity* $m$-vector $x$ denotes levels at which activities are operated.

Therefore,

- vector $x^T A$ gives the total amount of *goods used in production*
- vector $x^T B$ gives *total outputs*

An economy $(A, B)$ is said to be *productive*, if there exists a non-negative intensity vector $x \geq 0$ such that $x^T B > x^T A$.

The semi-positive $n$-vector $p$ contains prices assigned to the $n$ goods.

The $p$ vector implies *cost* and *revenue* vectors

- the vector $Ap$ tells *costs* of the vector of activities
- the vector $Bp$ tells *revenues* from the vector of activities

Satisfaction or a property of an input-output pair $(A, B)$ called *irreducibility* (or indecomposability) determines whether an economy can be decomposed into multiple "sub-economies".

**Definition:** For an economy $(A, B)$, the set of goods $S \subset \{1, 2, \ldots, n\}$ is called an *independent subset* if it is possible to produce every good in $S$ without consuming goods from outside $S$. Formally, the set $S$ is independent if $\exists T \subset \{1, 2, \ldots, m\}$ (a subset of activities) such that $a_{i,j} = 0 \; \forall i \in T$ and $j \in S^c$ and for all $j \in S$, $\exists i \in T$ for which $b_{i,j} > 0$. The economy is **irreducible** if there are no proper independent subsets.

We study two examples, both in Chapter 9.6 of Gale [Gal89]

```
# (1) Irreducible (A, B) example: α_0 = β_0
A1 = np.array([[0, 1, 0, 0],
               [1, 0, 0, 1],
               [0, 0, 1, 0]])

B1 = np.array([[1, 0, 0, 0],
               [0, 0, 2, 0],
               [0, 1, 0, 1]])

# (2) Reducible (A, B) example: β_0 < α_0
A2 = np.array([[0, 1, 0, 0, 0, 0],
```

(continues on next page)

(continued from previous page)

```
                [1, 0, 1, 0, 0, 0],
                [0, 0, 0, 1, 0, 0],
                [0, 0, 1, 0, 0, 1],
                [0, 0, 0, 0, 1, 0]])

B2 = np.array([[1, 0, 0, 1, 0, 0],
                [0, 1, 0, 0, 0, 0],
                [0, 0, 1, 0, 0, 0],
                [0, 0, 0, 0, 2, 0],
                [0, 0, 0, 1, 0, 1]])
```

The following code sets up our first Neumann economy or `Neumann` instance

```
n1 = Neumann(A1, B1)
n1
```

```
Generalized von Neumann expanding model:
  - number of goods          : 4
  - number of activities     : 3

Assumptions:
  - AI:  every column of B has a positive entry    : True
  - AII: every row of A has a positive entry       : True
```

Here is a second instance of a Neumann economy

```
n2 = Neumann(A2, B2)
n2
```

```
Generalized von Neumann expanding model:
  - number of goods          : 6
  - number of activities     : 5

Assumptions:
  - AI:  every column of B has a positive entry    : True
  - AII: every row of A has a positive entry       : True
```

## 27.3 Dynamic Interpretation

Attach a time index $t$ to the preceding objects, regard an economy as a dynamic system, and study sequences

$$\{(A_t, B_t)\}_{t\geq 0}, \qquad \{x_t\}_{t\geq 0}, \qquad \{p_t\}_{t\geq 0}$$

An interesting special case holds the technology process constant and investigates the dynamics of quantities and prices only.

Accordingly, in the rest of this lecture, we assume that $(A_t, B_t) = (A, B)$ for all $t \geq 0$.

A crucial element of the dynamic interpretation involves the timing of production.

We assume that production (consumption of inputs) takes place in period $t$, while the consequent output materializes in period $t+1$, i.e., consumption of $x_t^T A$ in period $t$ results in $x_t^T B$ amounts of output in period $t+1$.

These timing conventions imply the following feasibility condition:

$$x_t^T B \geq x_{t+1}^T A \qquad \forall t \geq 1$$

which asserts that no more goods can be used today than were produced yesterday.

Accordingly, $Ap_t$ tells the costs of production in period $t$ and $Bp_t$ tells revenues in period $t+1$.

### 27.3.1 Balanced Growth

We follow John von Neumann in studying "balanced growth".

Let $./$ denote an elementwise division of one vector by another and let $\alpha > 0$ be a scalar.

Then *balanced growth* is a situation in which

$$x_{t+1}./x_t = \alpha, \quad \forall t \geq 0$$

With balanced growth, the law of motion of $x$ is evidently $x_{t+1} = \alpha x_t$ and so we can rewrite the feasibility constraint as

$$x_t^T B \geq \alpha x_t^T A \qquad \forall t$$

In the same spirit, define $\beta \in \mathbb{R}$ as the **interest factor** per unit of time.

We assume that it is always possible to earn a gross return equal to the constant interest factor $\beta$ by investing "outside the model".

Under this assumption about outside investment opportunities, a no-arbitrage condition gives rise to the following (no profit) restriction on the price sequence:

$$\beta A p_t \geq B p_t \qquad \forall t$$

This says that production cannot yield a return greater than that offered by the outside investment opportunity (here we compare values in period $t+1$).

The balanced growth assumption allows us to drop time subscripts and conduct an analysis purely in terms of a time-invariant growth rate $\alpha$ and interest factor $\beta$.

## 27.4 Duality

Two problems are connected by a remarkable dual relationship between technological and valuation characteristics of the economy:

**Definition:** The *technological expansion problem* (TEP) for the economy $(A, B)$ is to find a semi-positive $m$-vector $x > 0$ and a number $\alpha \in \mathbb{R}$ that satisfy

$$\begin{aligned} \max_{\alpha} \quad & \alpha \\ \text{s.t.} \quad & x^T B \geq \alpha x^T A \end{aligned}$$

Theorem 9.3 of David Gale's book [Gal89] asserts that if Assumptions I and II are both satisfied, then a maximum value of $\alpha$ exists and that it is positive.

The maximal value is called the *technological expansion rate* and is denoted by $\alpha_0$. The associated intensity vector $x_0$ is the *optimal intensity vector*.

**Definition:** The economic expansion problem (EEP) for $(A, B)$ is to find a semi-positive $n$-vector $p > 0$ and a number $\beta \in \mathbb{R}$ that satisfy

$$\begin{aligned} \min_{\beta} \quad & \beta \\ \text{s.t.} \quad & Bp \leq \beta Ap \end{aligned}$$

Assumptions I and II imply existence of a minimum value $\beta_0 > 0$ called the *economic expansion rate*.

The corresponding price vector $p_0$ is the *optimal price vector*.

Because the criterion functions in the *technological expansion* problem and the *economical expansion problem* are both linearly homogeneous, the optimality of $x_0$ and $p_0$ are defined only up to a positive scale factor.

For convenience (and to emphasize a close connection to zero-sum games), we normalize both vectors $x_0$ and $p_0$ to have unit length.

A standard duality argument (see Lemma 9.4. in (Gale, 1960) [Gal89]) implies that under Assumptions I and II, $\beta_0 \leq \alpha_0$.

But to deduce that $\beta_0 \geq \alpha_0$, Assumptions I and II are not sufficient.

Therefore, von Neumann [vN37] went on to prove the following remarkable "duality" result that connects TEP and EEP.

**Theorem 1 (von Neumann):** If the economy $(A, B)$ satisfies Assumptions I and II, then there exist $(\gamma^*, x_0, p_0)$, where $\gamma^* \in [\beta_0, \alpha_0] \subset \mathbb{R}$, $x_0 > 0$ is an $m$-vector, $p_0 > 0$ is an $n$-vector, and the following arbitrage true

$$\begin{aligned} x_0^T B &\geq \gamma^* x_0^T A \\ Bp_0 &\leq \gamma^* Ap_0 \\ x_0^T \left(B - \gamma^* A\right) p_0 &= 0 \end{aligned}$$

---

**Note:** *Proof (Sketch):* Assumption I and II imply that there exist $(\alpha_0, x_0)$ and $(\beta_0, p_0)$ that solve the TEP and EEP, respectively. If $\gamma^* > \alpha_0$, then by definition of $\alpha_0$, there cannot exist a semi-positive $x$ that satisfies $x^T B \geq \gamma^* x^T A$. Similarly, if $\gamma^* < \beta_0$, there is no semi-positive $p$ for which $Bp \leq \gamma^* Ap$. Let $\gamma^* \in [\beta_0, \alpha_0]$, then $x_0^T B \geq \alpha_0 x_0^T A \geq \gamma^* x_0^T A$. Moreover, $Bp_0 \leq \beta_0 Ap_0 \leq \gamma^* Ap_0$. These two inequalities imply $x_0 \left(B - \gamma^* A\right) p_0 = 0$.

---

Here the constant $\gamma^*$ is both an expansion factor and an interest factor (not necessarily optimal).

We have already encountered and discussed the first two inequalities that represent feasibility and no-profit conditions.

Moreover, the equality $x_0^T \left(B - \gamma^* A\right) p_0 = 0$ concisely expresses the requirements that if any good grows at a rate larger than $\gamma^*$ (i.e., if it is *oversupplied*), then its price must be zero; and that if any activity provides negative profit, it must be unused.

Therefore, the conditions stated in Theorem I ex encode all equilibrium conditions.

So Theorem I essentially states that under Assumptions I and II there always exists an equilibrium $(\gamma^*, x_0, p_0)$ with balanced growth.

Note that Theorem I is silent about uniqueness of the equilibrium. In fact, it does not rule out (trivial) cases with $x_0^T Bp_0 = 0$ so that nothing of value is produced.

To exclude such uninteresting cases, Kemeny, Morgenstern and Thomspson [KMT56] add an extra requirement

$$x_0^T Bp_0 > 0$$

and call the associated equilibria *economic solutions*.

They show that this extra condition does not affect the existence result, while it significantly reduces the number of (relevant) solutions.

## 27.5 Interpretation as Two-player Zero-sum Game

To compute the equilibrium $(\gamma^*, x_0, p_0)$, we follow the algorithm proposed by Hamburger, Thompson and Weil (1967), building on the key insight that an equilibrium (with balanced growth) can be solves a particular two-player zero-sum game. First, we introduce some notation.

Consider the $m \times n$ matrix $C$ as a payoff matrix, with the entries representing payoffs from the **minimizing** column player to the **maximizing** row player and assume that the players can use mixed strategies. Thus,

- the row player chooses the $m$-vector $x > \mathbf{0}$ subject to $\iota_m^T x = 1$
- the column player chooses the $n$-vector $p > \mathbf{0}$ subject to $\iota_n^T p = 1$.

**Definition:** The $m \times n$ matrix game $C$ has the *solution* $(x^*, p^*, V(C))$ in mixed strategies if

$$(x^*)^T C e^j \geq V(C) \quad \forall j \in \{1, \dots, n\} \qquad \text{and} \qquad (e^i)^T C p^* \leq V(C) \quad \forall i \in \{1, \dots, m\}$$

The number $V(C)$ is called the *value* of the game.

From the above definition, it is clear that the value $V(C)$ has two alternative interpretations:

- by playing the appropriate mixed stategy, the maximizing player can assure himself at least $V(C)$ (no matter what the column player chooses)
- by playing the appropriate mixed stategy, the minimizing player can make sure that the maximizing player will not get more than $V(C)$ (irrespective of what is the maximizing player's choice)

A famous theorem of Nash (1951) tells us that there always exists a mixed strategy Nash equilibrium for any *finite* two-player zero-sum game.

Moreover, von Neumann's Minmax Theorem [vN28] implies that

$$V(C) = \max_x \min_p \quad x^T C p = \min_p \max_x \quad x^T C p = (x^*)^T C p^*$$

### 27.5.1 Connection with Linear Programming (LP)

Nash equilibria of a finite two-player zero-sum game solve a linear programming problem.

To see this, we introduce the following notation

- For a fixed $x$, let $v$ be the value of the minimization problem: $v \equiv \min_p x^T C p = \min_j x^T C e^j$
- For a fixed $p$, let $u$ be the value of the maximization problem: $u \equiv \max_x x^T C p = \max_i (e^i)^T C p$

Then the *max-min problem* (the game from the maximizing player's point of view) can be written as the *primal* LP

$$\begin{aligned} V(C) = & \max \quad v \\ \text{s.t.} \quad & v \iota_n^T \leq x^T C \\ & x \geq \mathbf{0} \\ & \iota_n^T x = 1 \end{aligned}$$

while the *min-max problem* (the game from the minimizing player's point of view) is the *dual* LP

$$\begin{aligned} V(C) = & \min \quad u \\ \text{s.t.} \quad & u \iota_m \geq C p \\ & p \geq \mathbf{0} \\ & \iota_m^T p = 1 \end{aligned}$$

Hamburger, Thompson and Weil [HTW67] view the input-output pair of the economy as payoff matrices of two-player zero-sum games.

Using this interpretation, they restate Assumption I and II as follows

$$V(-A) < 0 \qquad \text{and} \qquad V(B) > 0$$

---

**Note:** *Proof (Sketch)*:

- $\Rightarrow V(B) > 0$ implies $x_0^T B \gg \mathbf{0}$, where $x_0$ is a maximizing vector. Since $B$ is non-negative, this requires that each column of $B$ has at least one positive entry, which is Assumption I.
- $\Leftarrow$ From Assumption I and the fact that $p > \mathbf{0}$, it follows that $Bp > \mathbf{0}$. This implies that the maximizing player can always choose $x$ so that $x^T Bp > 0$ so that it must be the case that $V(B) > 0$.

---

In order to (re)state Theorem I in terms of a particular two-player zero-sum game, we define a matrix for $\gamma \in \mathbb{R}$

$$M(\gamma) \equiv B - \gamma A$$

For fixed $\gamma$, treating $M(\gamma)$ as a matrix game, calculating the solution of the game implies

- If $\gamma > \alpha_0$, then for all $x > 0$, there $\exists j \in \{1, \dots, n\}$, s.t. $[x^T M(\gamma)]_j < 0$ implying that $V(M(\gamma)) < 0$.
- If $\gamma < \beta_0$, then for all $p > 0$, there $\exists i \in \{1, \dots, m\}$, s.t. $[M(\gamma)p]_i > 0$ implying that $V(M(\gamma)) > 0$.
- If $\gamma \in \{\beta_0, \alpha_0\}$, then (by Theorem I) the optimal intensity and price vectors $x_0$ and $p_0$ satisfy

$$x_0^T M(\gamma) \geq \mathbf{0}^T \qquad \text{and} \qquad M(\gamma)p_0 \leq \mathbf{0}$$

That is, $(x_0, p_0, 0)$ is a solution of the game $M(\gamma)$ so that $V\left(M(\beta_0)\right) = V\left(M(\alpha_0)\right) = 0$.

- If $\beta_0 < \alpha_0$ and $\gamma \in (\beta_0, \alpha_0)$, then $V(M(\gamma)) = 0$.

Moreover, if $x'$ is optimal for the maximizing player in $M(\gamma')$ for $\gamma' \in (\beta_0, \alpha_0)$ and $p''$ is optimal for the minimizing player in $M(\gamma'')$ where $\gamma'' \in (\beta_0, \gamma')$, then $(x', p'', 0)$ is a solution for $M(\gamma)$ $\forall \gamma \in (\gamma'', \gamma')$.

*Proof (Sketch):* If $x'$ is optimal for a maximizing player in game $M(\gamma')$, then $(x')^T M(\gamma') \geq \mathbf{0}^T$ and so for all $\gamma < \gamma'$.

$$(x')^T M(\gamma) = (x')^T M(\gamma') + (x')^T (\gamma' - \gamma) A \geq \mathbf{0}^T$$

hence $V(M(\gamma)) \geq 0$. If $p''$ is optimal for a minimizing player in game $M(\gamma'')$, then $M(\gamma)p \leq \mathbf{0}$ and so for all $\gamma'' < \gamma$

$$M(\gamma)p'' = M(\gamma'') + (\gamma'' - \gamma) A p'' \leq \mathbf{0}$$

hence $V(M(\gamma)) \leq 0$.

It is clear from the above argument that $\beta_0$, $\alpha_0$ are the minimal and maximal $\gamma$ for which $V(M(\gamma)) = 0$.

Furthermore, Hamburger et al. [HTW67] show that the function $\gamma \mapsto V(M(\gamma))$ is continuous and nonincreasing in $\gamma$.

This suggests an algorithm to compute $(\alpha_0, x_0)$ and $(\beta_0, p_0)$ for a given input-output pair $(A, B)$.

### 27.5.2 Algorithm

Hamburger, Thompson and Weil [HTW67] propose a simple bisection algorithm to find the minimal and maximal roots (i.e. $\beta_0$ and $\alpha_0$) of the function $\gamma \mapsto V(M(\gamma))$.

#### Step 1

First, notice that we can easily find trivial upper and lower bounds for $\alpha_0$ and $\beta_0$.

- TEP requires that $x^T(B - \alpha A) \geq \mathbf{0}^T$ and $x > \mathbf{0}$, so if $\alpha$ is so large that $\max_i\{[(B - \alpha A)\iota_n]_i\} < 0$, then TEP ceases to have a solution.

Accordingly, let **UB** be the $\alpha^*$ that solves $\max_i\{[(B - \alpha^* A)\iota_n]_i\} = 0$.

- Similar to the upper bound, if $\beta$ is so low that $\min_j\{[\iota_m^T(B - \beta A)]_j\} > 0$, then the EEP has no solution and so we can define **LB** as the $\beta^*$ that solves $\min_j\{[\iota_m^T(B - \beta^* A)]_j\} = 0$.

The *bounds* method calculates these trivial bounds for us

```
n1.bounds()
```

```
(1.0, 2.0)
```

#### Step 2

Compute $\alpha_0$ and $\beta_0$

- Finding $\alpha_0$
  1. Fix $\gamma = \frac{UB+LB}{2}$ and compute the solution of the two-player zero-sum game associated with $M(\gamma)$. We can use either the primal or the dual LP problem.
  2. If $V(M(\gamma)) \geq 0$, then set $LB = \gamma$, otherwise let $UB = \gamma$.
  3. Iterate on 1. and 2. until $|UB - LB| < \epsilon$.
- Finding $\beta_0$
  1. Fix $\gamma = \frac{UB+LB}{2}$ and compute the solution of the two-player zero-sum game associated. with $M(\gamma)$. We can use either the primal or the dual LP problem.
  2. If $V(M(\gamma)) > 0$, then set $LB = \gamma$, otherwise let $UB = \gamma$.
  3. Iterate on 1. and 2. until $|UB - LB| < \epsilon$.
- *Existence*: Since $V(M(LB)) > 0$ and $V(M(UB)) < 0$ and $V(M(\cdot))$ is a continuous, nonincreasing function, there is at least one $\gamma \in [LB, UB]$, s.t. $V(M(\gamma)) = 0$.

The *zerosum* method calculates the value and optimal strategies associated with a given $\gamma$.

```
γ = 2

print(f'Value of the game with γ = {γ}')
print(n1.zerosum(γ=γ)[0])
print('Intensity vector (from the primal)')
print(n1.zerosum(γ=γ)[1])
print('Price vector (from the dual)')
print(n1.zerosum(γ=γ, dual=True)[1])
```

```
Value of the game with γ = 2
-0.24000000097850305
Intensity vector (from the primal)
[0.32 0.28 0.4 ]
Price vector (from the dual)
[4.00e-01 3.20e-01 2.80e-01 2.54e-10]
```

```
numb_grid = 100
γ_grid = np.linspace(0.4, 2.1, numb_grid)

value_ex1_grid = np.asarray([n1.zerosum(γ=γ_grid[i])[0]
                             for i in range(numb_grid)])
value_ex2_grid = np.asarray([n2.zerosum(γ=γ_grid[i])[0]
                             for i in range(numb_grid)])

fig, axes = plt.subplots(1, 2, figsize=(14, 5), sharey=True)
fig.suptitle(r'The function $V(M(\gamma))$', fontsize=16)

for ax, grid, N, i in zip(axes, (value_ex1_grid, value_ex2_grid),
                          (n1, n2), (1, 2)):
    ax.plot(γ_grid, grid)
    ax.set(title=f'Example {i}', xlabel='$\gamma$')
    ax.axhline(0, c='k', lw=1)
    ax.axvline(N.bounds()[0], c='r', ls='--', label='lower bound')
    ax.axvline(N.bounds()[1], c='g', ls='--', label='upper bound')

plt.show()
```

The *expansion* method implements the bisection algorithm for $\alpha_0$ (and uses the primal LP problem for $x_0$)

```
α_0, x, p = n1.expansion()
print(f'α_0 = {α_0}')
print(f'x_0 = {x}')
print(f'The corresponding p from the dual = {p}')
```

```
α_0 = 1.2599210478365421
x_0 = [0.33 0.26 0.41]
```

(continues on next page)

(continued from previous page)

```
The corresponding p from the dual = [4.13e-01 3.27e-01 2.60e-01 1.82e-10]
```

The *interest* method implements the bisection algorithm for $\beta_0$ (and uses the dual LP problem for $p_0$)

```
β_0, x, p = n1.interest()
print(f'β_0 = {β_0}')
print(f'p_0 = {p}')
print(f'The corresponding x from the primal = {x}')
```

```
β_0 = 1.2599210478365421
p_0 = [4.13e-01 3.27e-01 2.60e-01 1.82e-10]
The corresponding x from the primal = [0.33 0.26 0.41]
```

Of course, when $\gamma^*$ is unique, it is irrelevant which one of the two methods we use – both work.

In particular, as will be shown below, in case of an irreducible $(A, B)$ (like in Example 1), the maximal and minimal roots of $V(M(\gamma))$ necessarily coincide implying a "full duality" result, i.e. $\alpha_0 = \beta_0 = \gamma^*$ so that the expansion (and interest) rate $\gamma^*$ is unique.

### 27.5.3 Uniqueness and Irreducibility

As an illustration, compute first the maximal and minimal roots of $V(M(\cdot))$ for our Example 2 that has a reducible input-output pair $(A, B)$

```
α_0, x, p = n2.expansion()
print(f'α_0 = {α_0}')
print(f'x_0 = {x}')
print(f'The corresponding p from the dual = {p}')
```

```
α_0 = 1.1343231229111552
x_0 = [1.67e-11 1.83e-11 3.24e-01 2.61e-01 4.15e-01]
The corresponding p from the dual = [5.04e-01 4.96e-01 2.96e-12 2.24e-12 3.08e-12␣
↪3.56e-12]
```

```
β_0, x, p = n2.interest()
print(f'β_0 = {β_0}')
print(f'p_0 = {p}')
print(f'The corresponding x from the primal = {x}')
```

```
β_0 = 1.2579826870933175
p_0 = [5.11e-01 4.89e-01 2.73e-08 2.17e-08 1.88e-08 2.66e-09]
The corresponding x from the primal = [1.61e-09 1.65e-09 3.27e-01 2.60e-01 4.12e-
↪01]
```

As we can see, with a reducible $(A, B)$, the roots found by the bisection algorithms might differ, so there might be multiple $\gamma^*$ that make the value of the game with $M(\gamma^*)$ zero. (see the figure above).

Indeed, although the von Neumann theorem assures existence of the equilibrium, Assumptions I and II are not sufficient for uniqueness. Nonetheless, Kemeny et al. (1967) show that there are at most finitely many economic solutions, meaning that there are only finitely many $\gamma^*$ that satisfy $V(M(\gamma^*)) = 0$ and $x_0^T B p_0 > 0$ and that for each such $\gamma_i^*$, there is a self-contained part of the economy (a sub-economy) that in equilibrium can expand independently with the expansion coefficient $\gamma_i^*$.

The following theorem (see Theorem 9.10. in Gale [Gal89]) asserts that imposing irreducibility is sufficient for uniqueness of $(\gamma^*, x_0, p_0)$.

**Theorem II:** Adopt the conditions of Theorem 1. If the economy $(A, B)$ is irreducible, then $\gamma^* = \alpha_0 = \beta_0$.

### 27.5.4 A Special Case

There is a special $(A, B)$ that allows us to simplify the solution method significantly by invoking the powerful Perron-Frobenius theorem for non-negative matrices.

**Definition:** We call an economy *simple* if it satisfies

- $n = m$
- Each activity produces exactly one good
- Each good is produced by one and only one activity.

These assumptions imply that $B = I_n$, i.e., that $B$ can be written as an identity matrix (possibly after reshuffling its rows and columns).

The simple model has the following special property (Theorem 9.11. in Gale [Gal89]): if $x_0$ and $\alpha_0 > 0$ solve the TEP with $(A, I_n)$, then

$$x_0^T = \alpha_0 x_0^T A \qquad \Leftrightarrow \qquad x_0^T A = \left(\frac{1}{\alpha_0}\right) x_0^T$$

The latter shows that $1/\alpha_0$ is a positive eigenvalue of $A$ and $x_0$ is the corresponding non-negative left eigenvector.

The classic result of **Perron and Frobenius** implies that a non-negative matrix has a non-negative eigenvalue-eigenvector pair.

Moreover, if $A$ is irreducible, then the optimal intensity vector $x_0$ is positive and *unique* up to multiplication by a positive scalar.

Suppose that $A$ is reducible with $k$ irreducible subsets $S_1, \dots, S_k$. Let $A_i$ be the submatrix corresponding to $S_i$ and let $\alpha_i$ and $\beta_i$ be the associated expansion and interest factors, respectively. Then we have

$$\alpha_0 = \max_i \{\alpha_i\} \qquad \text{and} \qquad \beta_0 = \min_i \{\beta_i\}$$

# Part VII

# Time Series Models

CHAPTER

TWENTYEIGHT

# COVARIANCE STATIONARY PROCESSES

**Contents**


- *Covariance Stationary Processes*
  - *Overview*
  - *Introduction*
  - *Spectral Analysis*
  - *Implementation*


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

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

## 28.1 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.

### 28.1.1 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.

### 28.1.2 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.

### 28.1.3 Other Reading

For supplementary reading, see

- [LS18], chapter 2
- [Sar87], chapter 11
- John Cochrane's notes on time series analysis, chapter 8
- [Shi95], chapter 6
- [CC08], all

Let's start with some imports:

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import quantecon as qe
```

## 28.2 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*).

### 28.2.1 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.

### 28.2.2 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.

### 28.2.3 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*

$$X_t = \sum_{j=0}^{\infty} \psi_j \epsilon_{t-j}, \qquad t \in \mathbb{Z} \tag{28.1}$$

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 (28.1) 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 (28.1) is

$$\gamma(k) = \sigma^2 \sum_{j=0}^{\infty} \psi_j \psi_{j+k} \tag{28.2}$$

By the Cauchy-Schwartz inequality, one can show that $\gamma(k)$ satisfies equation (28.2).

Evidently, $\gamma(k)$ does not depend on $t$.

### 28.2.4 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 $X_{t-1}, X_{t-2}, \ldots$
- $\eta_t$ 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 [Sar87], p. 286.

### 28.2.5 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

$$X_t = \phi X_{t-1} + \epsilon_t \quad \text{where} \quad |\phi| < 1 \quad \text{and } \{\epsilon_t\} \text{ is white noise} \tag{28.3}$$

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 (28.2) to the previous expression for $X_t$, we get the AR(1) autocovariance function

$$\gamma(k) = \phi^k \frac{\sigma^2}{1-\phi^2}, \qquad k = 0, 1, \ldots \tag{28.4}$$

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=f'autocovariance, $\phi = {ϕ:.2}$')
    ax.legend(loc='upper right')
```

(continues on next page)

(continued from previous page)

```
    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

### 28.2.6 ARMA Processes

A stochastic process $\{X_t\}$ is called an *autoregressive moving average process*, or ARMA$(p, q)$, if it can be written as

$$X_t = \phi_1 X_{t-1} + \cdots + \phi_p X_{t-p} + \epsilon_t + \theta_1 \epsilon_{t-1} + \cdots + \theta_q \epsilon_{t-q} \tag{28.5}$$

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 (28.5) as

$$L^0 X_t - \phi_1 L^1 X_t - \cdots - \phi_p L^p X_t = L^0 \epsilon_t + \theta_1 L^1 \epsilon_t + \cdots + \theta_q L^q \epsilon_t \tag{28.6}$$

If we let $\phi(z)$ and $\theta(z)$ be the polynomials

$$\phi(z) := 1 - \phi_1 z - \cdots - \phi_p z^p \quad \text{and} \quad \theta(z) := 1 + \theta_1 z + \cdots + \theta_q z^q \tag{28.7}$$

then (28.6) becomes

$$\phi(L) X_t = \theta(L) \epsilon_t \tag{28.8}$$

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 [CC08].

The function $t \mapsto \psi_t$ is often called the *impulse response function*.

## 28.3 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.

### 28.3.1 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 $\tan(\omega) = y/x$.

### 28.3.2 Spectral Densities

Let $\{X_t\}$ be a covariance stationary process with autocovariance function $\gamma$ 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 $\gamma$.

$$f(\omega) := \sum_{k \in \mathbb{Z}} \gamma(k) e^{-i\omega k}, \qquad \omega \in \mathbb{R}$$

(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 $\gamma$ is *even*, in the sense that $\gamma(t) = \gamma(-t)$ for all $t$, we can show that

$$f(\omega) = \gamma(0) + 2 \sum_{k \geq 1} \gamma(k) \cos(\omega k) \tag{28.9}$$

It is not difficult to confirm that $f$ is

- real-valued
- even ($f(\omega) = f(-\omega)$ ), and
- $2\pi$-periodic, in the sense that $f(2\pi + \omega) = f(\omega)$ for all $\omega$

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 $[0, \pi]$.

### 28.3.3 Example 1: White Noise

Consider a white noise process $\{\epsilon_t\}$ with standard deviation $\sigma$.

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")

### 28.3.4 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

$$f(\omega) = \sigma^2(1 + 2\theta\cos(\omega) + \theta^2) \tag{28.10}$$

With a bit more effort, it's possible to show (see, e.g., p. 261 of [Sar87]) that the spectral density of the AR(1) process $X_t = \phi X_{t-1} + \epsilon_t$ is

$$f(\omega) = \frac{\sigma^2}{1 - 2\phi\cos(\omega) + \phi^2} \tag{28.11}$$

More generally, it can be shown that the spectral density of the ARMA process (28.5) is

$$f(\omega) = \left|\frac{\theta(e^{i\omega})}{\phi(e^{i\omega})}\right|^2 \sigma^2 \tag{28.12}$$

where

- $\sigma$ is the standard deviation of the white noise process $\{\epsilon_t\}$.
- the polynomials $\phi(\cdot)$ and $\theta(\cdot)$ are as defined in (28.7).

The derivation of (28.12) uses the fact that convolutions become products under Fourier transformations.

The proof is elegant and can be found in many places — see, for example, [Sar87], chapter 11, section 4.

It's a nice exercise to verify that (28.10) and (28.11) are indeed special cases of (28.12).

### 28.3.5 Interpreting the Spectral Density

Plotting (28.11) reveals the shape of the spectral density for the AR(1) model when $\phi$ 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)):
```

(continues on next page)

(continued from previous page)

```
    ax = axes[i]
    sd = ar1_sd(ϕ, ωs)
    ax.plot(ωs, sd, 'b-', alpha=0.6, lw=2,
            label='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

$$f(\omega) = \gamma(0) + 2 \sum_{k \geq 1} \gamma(k) \cos(\omega k) = \gamma(0) + 2 \sum_{k \geq 1} (-0.8)^k \cos(\omega k) \tag{28.13}$$

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 (28.13) 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]
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='$\gamma(k)$')
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='$\cos(\pi k)$')
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='$\gamma(k) \cos(\pi k)$')
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='$\gamma(k)$')
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='$\cos(\pi k/3)$')
ax.legend(loc='upper right')
```

(continues on next page)

(continued from previous page)

```
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='$\gamma(k) \cos(\pi k/3)$')
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 $\omega$ where the autocovariance function exhibits damped cycles.

### 28.3.6 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

$$\gamma(k) = \frac{1}{2\pi} \int_{-\pi}^{\pi} f(\omega) e^{i\omega k} d\omega \tag{28.14}$$

This is convenient in situations where the spectral density is easier to calculate and manipulate than the autocovariance function.

(For example, the expression (28.12) for the ARMA spectral density is much easier to work with than the expression for the ARMA autocovariance)

### 28.3.7 Mathematical Theory

This section is loosely based on [Sar87], 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

$$f = \sum_k \alpha_k h_k \quad \text{where} \quad \alpha_k := \langle f, h_k \rangle \tag{28.15}$$

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, $\alpha$ is in $\ell_2$, 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 $\alpha$ and $\beta$ are distinct in $\ell_2$, then so are their expansions in $H$.
- $T$ is onto — if $f \in H$ then its preimage in $\ell_2$ is the sequence $\alpha$ 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 $\ell_2$.

In essence, this says that each separable Hilbert space we consider is just a different way of looking at the fundamental space $\ell_2$.

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

$$T\gamma = \sum_{k \in \mathbb{Z}} \gamma(k) \frac{e^{i\omega k}}{\sqrt{2\pi}} = \frac{1}{\sqrt{2\pi}} f(\omega) \tag{28.16}$$

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 $\gamma$.

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 $\gamma$, and, in particular, $\gamma(k) = \langle T\gamma, h_k \rangle$.

Transforming this inner product into its integral expression and using (28.16) gives (28.14), justifying our earlier expression for the inverse transform.

## 28.4 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

### 28.4.1 Application

Let's use this code to replicate the plots on pages 68–69 of [LS18].

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)
```

```
/usr/share/miniconda3/envs/quantecon/lib/python3.8/site-packages/numpy/core/_
↪asarray.py:102: ComplexWarning: Casting complex values to real discards the␣
↪imaginary part
  return array(a, dtype, copy=False, order=order)
```

(continues on next page)

(continued from previous page)

```
/tmp/ipykernel_10857/4271821819.py:15: UserWarning: Attempted to set non-positive␣
↪bottom ylim on a log-scaled axis.
Invalid limit will be ignored.
  ax.set(xlim=(0, np.pi), ylim=(0, np.max(spect)),
/usr/share/miniconda3/envs/quantecon/lib/python3.8/site-packages/matplotlib/
↪transforms.py:975: 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 [LS18] 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
```

(continues on next page)

(continued from previous page)

```
arma = qe.ARMA(ϕ, θ)
quad_plot(arma)
```

```
/usr/share/miniconda3/envs/quantecon/lib/python3.8/site-packages/numpy/core/_
↪asarray.py:102: ComplexWarning: Casting complex values to real discards the␣
↪imaginary part
  return array(a, dtype, copy=False, order=order)
/tmp/ipykernel_10857/4271821819.py:15: UserWarning: Attempted to set non-positive␣
↪bottom ylim on a log-scaled axis.
Invalid limit will be ignored.
  ax.set(xlim=(0, np.pi), ylim=(0, np.max(spect)),
/usr/share/miniconda3/envs/quantecon/lib/python3.8/site-packages/matplotlib/
↪transforms.py:975: ComplexWarning: Casting complex values to real discards the␣
↪imaginary part
  self._points[:, 1] = interval
```

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)
```

```
/usr/share/miniconda3/envs/quantecon/lib/python3.8/site-packages/numpy/core/_
↪asarray.py:102: ComplexWarning: Casting complex values to real discards the␣
↪imaginary part
```

(continues on next page)

(continued from previous page)

```
  return array(a, dtype, copy=False, order=order)
/tmp/ipykernel_10857/4271821819.py:15: UserWarning: Attempted to set non-positive␣
↪bottom ylim on a log-scaled axis.
Invalid limit will be ignored.
  ax.set(xlim=(0, np.pi), ylim=(0, np.max(spect)),
/usr/share/miniconda3/envs/quantecon/lib/python3.8/site-packages/matplotlib/
↪transforms.py:975: ComplexWarning: Casting complex values to real discards the␣
↪imaginary part
  self._points[:, 1] = interval
```

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)
```

```
/usr/share/miniconda3/envs/quantecon/lib/python3.8/site-packages/numpy/core/_
↪asarray.py:102: ComplexWarning: Casting complex values to real discards the␣
↪imaginary part
  return array(a, dtype, copy=False, order=order)
/tmp/ipykernel_10857/4271821819.py:15: UserWarning: Attempted to set non-positive␣
↪bottom ylim on a log-scaled axis.
Invalid limit will be ignored.
  ax.set(xlim=(0, np.pi), ylim=(0, np.max(spect)),
/usr/share/miniconda3/envs/quantecon/lib/python3.8/site-packages/matplotlib/
↪transforms.py:975: ComplexWarning: Casting complex values to real discards the␣
↪imaginary part
```

(continues on next page)

(continued from previous page)

```
  self._points[:, 1] = interval
```

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)
```

```
/usr/share/miniconda3/envs/quantecon/lib/python3.8/site-packages/numpy/core/_
↪asarray.py:102: ComplexWarning: Casting complex values to real discards the␣
↪imaginary part
  return array(a, dtype, copy=False, order=order)
/tmp/ipykernel_10857/4271821819.py:15: UserWarning: Attempted to set non-positive␣
↪bottom ylim on a log-scaled axis.
Invalid limit will be ignored.
  ax.set(xlim=(0, np.pi), ylim=(0, np.max(spect)),
/usr/share/miniconda3/envs/quantecon/lib/python3.8/site-packages/matplotlib/
↪transforms.py:975: ComplexWarning: Casting complex values to real discards the␣
↪imaginary part
  self._points[:, 1] = interval
```

### 28.4.2 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 (28.8).

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 (28.7) — 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.

### 28.4.3 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 (28.14), we have now shown that, for $n$ sufficiently large, $a_k \approx \gamma(k)$ — which is exactly what we want to compute.

CHAPTER

TWENTYNINE

# ESTIMATION OF SPECTRA

**Contents**


- *Estimation of Spectra*
  - *Overview*
  - *Periodograms*
  - *Smoothing*
  - *Exercises*
  - *Solutions*


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

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

## 29.1 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 [Sar87] or [CC08].

Let's start with some standard imports:

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from quantecon import ARMA, periodogram, ar_periodogram
```

## 29.2 Periodograms

*Recall that* the spectral density $f$ of a covariance stationary process with autocorrelation function $\gamma$ 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 $\gamma$ is unknown.

In particular, let $X_0, \dots, X_{n-1}$ 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 $X_0, \dots, X_{n-1}$, which is defined as

$$I(\omega) := \frac{1}{n} \left| \sum_{t=0}^{n-1} X_t e^{it\omega} \right|^2, \qquad \omega \in \mathbb{R} \tag{29.1}$$

(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 $2\pi$-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.

### 29.2.1 Interpretation

To interpret the periodogram, it is convenient to focus on its values at the *Fourier frequencies*

$$\omega_j := \frac{2\pi j}{n}, \quad j = 0, \dots, n-1$$

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 $j > 0$,

$$\sum_{t=0}^{n-1} e^{it\omega_j} = \sum_{t=0}^{n-1} \exp\left\{ i2\pi j \frac{t}{n} \right\} = 0$$

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

$$\hat{\gamma}(k) := \frac{1}{n} \sum_{t=k}^{n-1} (X_t - \bar{X})(X_{t-k} - \bar{X}), \qquad k = 0, 1, \dots, n-1$$

This is the sample autocovariance function, the natural "plug-in estimator" of the *autocovariance function* $\gamma$.

("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 $f(\omega_j)$.

### 29.2.2 Calculation

Let's now consider how to compute the periodogram as defined in (29.1).

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 $a_0, \dots, a_{n-1}$, the discrete Fourier transform computes the sequence

$$A_j := \sum_{t=0}^{n-1} a_t \exp\left\{ i 2\pi \frac{tj}{n} \right\}, \qquad j = 0, \dots, n-1$$

With `numpy.fft.fft` imported as `fft` and $a_0, \dots, a_{n-1}$ stored in NumPy array `a`, the function call `fft(a)` returns the values $A_0, \dots, A_{n-1}$ as a NumPy array.

It follows that when the data $X_0, \dots, X_{n-1}$ are stored in array `X`, the values $I(\omega_j)$ at the Fourier frequencies, which are given by

$$\frac{1}{n} \left| \sum_{t=0}^{n-1} X_t \exp\left\{ i 2\pi \frac{tj}{n} \right\} \right|^2, \qquad j = 0, \dots, n-1$$

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

$$X_t = 0.5 X_{t-1} + \epsilon_t - 0.8 \epsilon_{t-2} \tag{29.2}$$

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()
```

```
/usr/share/miniconda3/envs/quantecon/lib/python3.8/site-packages/numpy/core/_
↪asarray.py:102: ComplexWarning: Casting complex values to real discards the␣
↪imaginary part
  return array(a, dtype, copy=False, order=order)
```

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.

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

## 29.3 Smoothing

There are two related issues here.

One is that, given the way the fast Fourier transform is implemented, the number of points $\omega$ 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.

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

$$I(\omega_{j-p}), I(\omega_{j-p+1}), \dots, I(\omega_j), \dots, I(\omega_{j+p})$$

This weighted average can be written as

$$I_S(\omega_j) := \sum_{\ell=-p}^{p} w(\ell) I(\omega_{j+\ell}) \tag{29.3}$$

where the weights $w(-p), \dots, w(p)$ 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 (29.3).

```
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()
```

### 29.3.1 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 (29.2) — 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.

### 29.3.2 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 (29.3).

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 $\omega$ is close to $\omega'$.

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 $\omega'$ near to $\omega$ 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.

### 29.3.3 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

$$X_{t+1} = \mu + \phi X_t + \epsilon_{t+1} \tag{29.4}$$

where $\mu$ 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

$$f(\omega) = \left| \frac{1}{1 - \phi e^{i\omega}} \right|^2 g(\omega) \tag{29.5}$$

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 $\phi$.

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 (29.4) 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.

## 29.4 Exercises

### 29.4.1 Exercise 1

Replicate *this figure* (modulo randomness).

The model is as in equation (29.2) and there are 400 observations.

For the smoothed periodogram, the window type is "hamming".

### 29.4.2 Exercise 2

Replicate *this figure* (modulo randomness).

The model is as in equation (29.4), 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.

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

## 29.5 Solutions

### 29.5.1 Exercise 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')

    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()
```

```
/usr/share/miniconda3/envs/quantecon/lib/python3.8/site-packages/numpy/core/_
↪asarray.py:102: ComplexWarning: Casting complex values to real discards the␣
↪imaginary part
  return array(a, dtype, copy=False, order=order)
/usr/share/miniconda3/envs/quantecon/lib/python3.8/site-packages/numpy/core/_
↪asarray.py:102: ComplexWarning: Casting complex values to real discards the␣
↪imaginary part
  return array(a, dtype, copy=False, order=order)
/usr/share/miniconda3/envs/quantecon/lib/python3.8/site-packages/numpy/core/_
↪asarray.py:102: ComplexWarning: Casting complex values to real discards the␣
↪imaginary part
  return array(a, dtype, copy=False, order=order)
```

window length = 15
periodogram
spectral density
smoothed periodogram
window length = 55
periodogram
spectral density
smoothed periodogram
window length = 175
periodogram
spectral density
smoothed periodogram

### 29.5.2 Exercise 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()
```

```
/usr/share/miniconda3/envs/quantecon/lib/python3.8/site-packages/numpy/core/_
↪asarray.py:102: ComplexWarning: Casting complex values to real discards the␣
↪imaginary part
  return array(a, dtype, copy=False, order=order)
/usr/share/miniconda3/envs/quantecon/lib/python3.8/site-packages/numpy/core/_
↪asarray.py:102: ComplexWarning: Casting complex values to real discards the␣
↪imaginary part
  return array(a, dtype, copy=False, order=order)
/usr/share/miniconda3/envs/quantecon/lib/python3.8/site-packages/numpy/core/_
↪asarray.py:102: ComplexWarning: Casting complex values to real discards the␣
↪imaginary part
  return array(a, dtype, copy=False, order=order)
```

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

THIRTY

# ADDITIVE AND MULTIPLICATIVE FUNCTIONALS

**Contents**


- *Additive and Multiplicative Functionals*
  - *Overview*
  - *A Particular Additive Functional*
  - *Dynamics*
  - *Code*
  - *More About the Multiplicative Martingale*


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

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

## 30.1 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.

Are there 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?

The answer provided by Hansen and Scheinkman [HS09] is yes.

They 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.

Hansen and Sargent [HS08b] (chs. 5 and 8) describe discrete time versions of additive and multiplicative functionals.

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 and Sargent [HS08b].

Let's start with some imports:

```
import numpy as np
import scipy as sp
import scipy.linalg as la
import quantecon as qe
import matplotlib.pyplot as plt
%matplotlib inline
from scipy.stats import norm, lognorm
```

## 30.2 A Particular Additive Functional

Hansen and Sargent [HS08b] describe 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)

$$x_{t+1} = Ax_t + Bz_{t+1} \tag{30.1}$$

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 $\nu$,
- the vector $x_t$, and
- the same Gaussian vector $z_{t+1}$ that appears in the VAR (30.1)

In particular,

$$y_{t+1} - y_t = \nu + Dx_t + Fz_{t+1} \tag{30.2}$$

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*.

### 30.2.1 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.

## 30.3 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.

$$\tilde{x}_{t+1} = \phi_1 \tilde{x}_t + \phi_2 \tilde{x}_{t-1} + \phi_3 \tilde{x}_{t-2} + \phi_4 \tilde{x}_{t-3} + \sigma z_{t+1} \tag{30.3}$$

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 (30.3) is not a first order system like (30.1), 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 (30.1) – (30.2) 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.

### 30.3.1 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 and Scheinkman [HS09].

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, v=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 v
        if not np.any(v):
            self.v = np.zeros((self.nm, 1))
        elif type(v) == float:
            self.v = np.asarray([[v]])
        elif len(v.shape) == 1:
            self.v = np.expand_dims(v, 1)
        else:
            self.v = v
```

(continues on next page)

(continued from previous page)

```
        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):
        """
        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])
```

(continues on next page)

(continued from previous page)

```
        # 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.lss.LinearStateSpace(Abar, Bbar, Gbar, Hbar, mu_0=x0, Sigma_0=S0)

        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
    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))
    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 $y_t$.

### 30.3.2 Decomposition

Hansen and Sargent [HS08b] 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 (30.1) and (30.2), we first construct the matrices

$$H := F + D(I - A)^{-1}B$$
$$g := D(I - A)^{-1}$$

Then the Hansen-Scheinkman [HS09] decomposition is

$$y_t = \underbrace{t\nu}_{\text{trend component}} + \overbrace{\sum_{j=1}^{t} H z_j}^{\text{Martingale component}} - \underbrace{g x_t}_{\text{stationary component}} + \overbrace{g x_0 + y_0}^{\text{initial conditions}}$$

At this stage, you should pause and verify that $y_{t+1} - y_t$ satisfies (30.2).

It is convenient for us to introduce the following notation:

- $\tau_t = \nu t$ , a linear, deterministic trend
- $m_t = \sum_{j=1}^{t} H z_j$, a martingale with time $t+1$ increment $H z_{t+1}$
- $s_t = g x_t$, an (asymptotically) stationary component

We want to characterize and simulate components $\tau_t, m_t, s_t$ 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 (30.1) and (30.2) 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

$$\tilde{x}_{t+1} = \tilde{A}\tilde{x}_t + \tilde{B} z_{t+1}$$
$$\tilde{y}_t = \tilde{D}\tilde{x}_t$$

By picking out components of $\tilde{y}_t$, we can track all variables of interest.

## 30.4 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

### 30.4.1 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

$$\tilde{e}(x) = \exp[g(x)] = \exp\big[D(I-A)^{-1}x\big]$$

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 $150$.

### 30.4.2 Peculiar Large Sample Property

Hansen and Sargent [HS08b] (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 [HS08b].

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.

## 30.5 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.

### 30.5.1 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.

### 30.5.2 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.lss.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.v, H, g

    def multiplicative_decomp(self):
        """
        Return values for the multiplicative decomposition (Example 5.4.4.)
            - v_tilde  : eigenvalue
            - H        : vector for the Jensen term
        """
        v, H, g = self.additive_decomp()
        v_tilde = v + (.5) * H**2

        return v_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()

fig.suptitle(r"Densities of $\tilde{M}_t$", fontsize=18, y=1.02)
for (it, dens_t) in enumerate(dens_to_plot):
```

(continues on next page)

(continued from previous page)

```
    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 $0$.

### 30.5.3 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

THIRTYONE

# CLASSICAL CONTROL WITH LINEAR ALGEBRA

**Contents**


- *Classical Control with Linear Algebra*
  - *Overview*
  - *A Control Problem*
  - *Finite Horizon Theory*
  - *The Infinite Horizon Limit*
  - *Undiscounted Problems*
  - *Implementation*
  - *Exercises*


## 31.1 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
%matplotlib inline
```

### 31.1.1 References

Useful references include [Whi63], [HS80], [Orf88], [AP91], and [Mut60].

## 31.2 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 $d_0, d_1, \ldots, d_m$ is a given scalar sequence.

Consider the discrete-time control problem

$$\max_{\{y_t\}} \lim_{N\to\infty} \sum_{t=0}^{N} \beta^t \left\{ a_t y_t - \frac{1}{2} h y_t^2 - \frac{1}{2} \left[d(L) y_t\right]^2 \right\}, \tag{31.1}$$

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 (31.1) is subject to initial conditions for $y_{-1}, y_{-2} \cdots, y_{-m}$.

Maximization is over infinite sequences $\{y_t\}_{t\geq 0}$.

### 31.2.1 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 (31.1) 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 [Sar87].

## 31.3 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 (31.1) are not sufficient for maximizing (31.1))

We begin by

1. fixing $N > m$,
2. differentiating the finite version of (31.1) with respect to $y_0, y_1, \dots, y_N$, and
3. setting these derivatives to zero.

For $t = 0, \dots, N - m$ these first-order necessary conditions are the *Euler equations*.

For $t = N - m + 1, \dots, N$, 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 $t = 0,\ 1,\ \dots,\ N - m$ 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

$$\frac{\partial J}{\partial y_t} = 2\beta^t\, d(\beta L^{-1})\, d(L) y_t \tag{31.2}$$

Differentiating $J$ with respect to $y_t$ for $t = N - m + 1, \dots, N$ gives

$$\begin{aligned}
\frac{\partial J}{\partial y_N} &= 2\beta^N \, d_0 \, d(L) y_N \\
\frac{\partial J}{\partial y_{N-1}} &= 2\beta^{N-1} \left[d_0 + \beta \, d_1 \, L^{-1}\right] d(L) y_{N-1} \\
\vdots \quad & \quad \vdots \\
\frac{\partial J}{\partial y_{N-m+1}} &= 2\beta^{N-m+1} \left[d_0 + \beta L^{-1} \, d_1 + \cdots + \beta^{m-1} \, L^{-m+1} \, d_{m-1}\right] d(L) y_{N-m+1}
\end{aligned} \tag{31.3}$$

With these preliminaries under our belts, we are ready to differentiate (31.1).

Differentiating (31.1) with respect to $y_t$ for $t = 0, \dots, N - m$ gives the Euler equations

$$\left[h + d\left(\beta L^{-1}\right) d(L)\right] y_t = a_t, \quad t = 0, \, 1, \, \dots, N - m \tag{31.4}$$

The system of equations (31.4) forms a $2 \times m$ order linear *difference equation* that must hold for the values of $t$ indicated.

Differentiating (31.1) with respect to $y_t$ for $t = N - m + 1, \dots, N$ gives the terminal conditions

$$\begin{aligned}
\beta^N (a_N - h y_N - d_0 \, d(L) y_N) &= 0 \\
\beta^{N-1} \left(a_{N-1} - h y_{N-1} - \left(d_0 + \beta \, d_1 \, L^{-1}\right) d(L) \, y_{N-1}\right) &= 0 \\
&\vdots\vdots \\
\beta^{N-m+1} \left(a_{N-m+1} - h y_{N-m+1} - (d_0 + \beta L^{-1} d_1 + \cdots + \beta^{m-1} L^{-m+1} d_{m-1}) d(L) y_{N-m+1}\right) &= 0
\end{aligned} \tag{31.5}$$

In the finite $N$ problem, we want simultaneously to solve (31.4) subject to the $m$ initial conditions $y_{-1}, \dots, y_{-m}$ and the $m$ terminal conditions (31.5).

These conditions uniquely pin down the solution of the finite $N$ problem.

That is, for the finite $N$ problem, conditions (31.4) and (31.5) are necessary and sufficient for a maximum, by concavity of the objective function.

Next, we describe how to obtain the solution using matrix methods.

### 31.3.1 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

$$\begin{aligned}
\left[h + d\left(\beta L^{-1}\right) d\left(L\right)\right] y_t &= a_t, \quad t = 0, \, 1, \, \dots, \, N - 1 \\
\beta^N \left[a_N - h \, y_N - d_0 \, d\left(L\right) y_N\right] &= 0
\end{aligned} \tag{31.6}$$

where $d(L) = d_0 + d_1 L$.

These equations are to be solved for $y_0, y_1, \dots, y_N$ as functions of $a_0, a_1, \dots, a_N$ 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 (31.6) as the matrix equation

$$\begin{bmatrix} (\phi_0 - d_1^2) & \phi_1 & 0 & 0 & \dots & \dots & 0 \\ \beta\phi_1 & \phi_0 & \phi_1 & 0 & \dots & \dots & 0 \\ 0 & \beta\phi_1 & \phi_0 & \phi_1 & \dots & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & \dots & \dots & \dots & \beta\phi_1 & \phi_0 & \phi_1 \\ 0 & \dots & \dots & \dots & 0 & \beta\phi_1 & \phi_0 \end{bmatrix} \begin{bmatrix} y_N \\ y_{N-1} \\ y_{N-2} \\ \vdots \\ y_1 \\ y_0 \end{bmatrix} = \begin{bmatrix} a_N \\ a_{N-1} \\ a_{N-2} \\ \vdots \\ a_1 \\ a_0 - \phi_1 y_{-1} \end{bmatrix} \tag{31.7}$$

or

$$W\bar{y} = \bar{a} \tag{31.8}$$

Notice how we have chosen to arrange the $y_t$'s in reverse time order.

The matrix $W$ on the left side of (31.7) 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 $\beta$ time the super-diagonal elements.

The solution of (31.8) can be expressed in the form

$$\bar{y} = W^{-1}\bar{a} \tag{31.9}$$

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 (31.7) or (31.8) 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 (31.9), we obtain

$$U\bar{y} = L^{-1}\bar{a} \tag{31.10}$$

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 (31.7) 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, (31.10) has the form

$$\begin{bmatrix} 1 & U_{12} & 0 & 0 & \dots & 0 & 0 \\ 0 & 1 & U_{23} & 0 & \dots & 0 & 0 \\ 0 & 0 & 1 & U_{34} & \dots & 0 & 0 \\ 0 & 0 & 0 & 1 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & 1 & U_{N,N+1} \\ 0 & 0 & 0 & 0 & \dots & 0 & 1 \end{bmatrix} \begin{bmatrix} y_N \\ y_{N-1} \\ y_{N-2} \\ y_{N-3} \\ \vdots \\ y_1 \\ y_0 \end{bmatrix} =$$

$$\begin{bmatrix} L^{-1}_{11} & 0 & 0 & \dots & 0 \\ L^{-1}_{21} & L^{-1}_{22} & 0 & \dots & 0 \\ L^{-1}_{31} & L^{-1}_{32} & L^{-1}_{33} & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ L^{-1}_{N,1} & L^{-1}_{N,2} & L^{-1}_{N,3} & \dots & 0 \\ L^{-1}_{N+1,1} & L^{-1}_{N+1,2} & L^{-1}_{N+1,3} & \dots & L^{-1}_{N+1\,N+1} \end{bmatrix} \begin{bmatrix} a_N \\ a_{N-1} \\ a_{N-2} \\ \vdots \\ a_1 \\ a_0 - \phi_1 y_{-1} \end{bmatrix}$$

where $L^{-1}_{ij}$ 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 $a_t, a_{t+1}, \dots, a_N$.

### 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:

$$D_{jk} = d_0 d_{k-j} + d_1 d_{k-j+1} + \dots + d_{j-1} d_{k-1}, \qquad k \geq j$$

Let $I_{m+1}$ be the $(m+1) \times (m+1)$ identity matrix.

Let $\phi_j$ be the coefficients in the expansion $\phi(L) = h + d(L^{-1})d(L)$.

Then the first order conditions (31.4) and (31.5) 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

$$M_{ij} = \begin{cases} D_{i-j,\,m+1} \text{ for } i > j \\ 0 \text{ for } i \leq j \end{cases}$$

$$\begin{aligned} \phi_m y_{N-1} + \phi_{m-1} y_{N-2} + \dots + \phi_0 y_{N-m-1} + \phi_1 y_{N-m-2} + \\ \dots + \phi_m y_{N-2m-1} = a_{N-m-1} \\ \phi_m y_{N-2} + \phi_{m-1} y_{N-3} + \dots + \phi_0 y_{N-m-2} + \phi_1 y_{N-m-3} + \\ \dots + \phi_m y_{N-2m-2} = a_{N-m-2} \\ \vdots \\ \phi_m y_{m+1} + \phi_{m-1} y_m + + \dots + \phi_0 y_1 + \phi_1 y_0 + \phi_m y_{-m+1} = a_1 \\ \phi_m y_m + \phi_{m-1} y_{m-1} + \phi_{m-2} + \dots + \phi_0 y_0 + \phi_1 y_{-1} + \dots + \phi_m y_{-m} = a_0 \end{aligned}$$

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

$$U\bar{y} = L^{-1}\bar{a} \tag{31.11}$$

$$\sum_{j=0}^{t} U_{-t+N+1,\,-t+N+j+1}\, y_{t-j} = \sum_{j=0}^{N-t} L_{-t+N+1,\,-t+N+1-j}\, \bar{a}_{t+j}\ ,$$
$$t = 0, 1, \ldots, N$$

where $L^{-1}_{t,s}$ is the element in the $(t, s)$ position of $L$, and similarly for $U$.

The left side of equation (31.11) 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 $N \to \infty$, (31.11) 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.

## 31.4 The Infinite Horizon Limit

For the infinite horizon problem, we propose to discover first-order necessary conditions by taking the limits of (31.4) and (31.5) as $N \to \infty$.

This approach is valid, and the limits of (31.4) and (31.5) as $N$ approaches infinity are first-order necessary conditions for a maximum.

However, for the infinite horizon problem with $\beta < 1$, the limits of (31.4) and (31.5) are, in general, not sufficient for a maximum.

That is, the limits of (31.5) do not provide enough information uniquely to determine the solution of the Euler equation (31.4) that maximizes (31.1).

As we shall see below, a side condition on the path of $y_t$ that together with (31.4) is sufficient for an optimum is

$$\sum_{t=0}^{\infty} \beta^t\, h y_t^2 < \infty \tag{31.12}$$

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 (31.1) than does the optimal path selected by the solution procedure below.

Consider the *characteristic equation* associated with the Euler equation

$$h + d\left(\beta z^{-1}\right) d\left(z\right) = 0 \tag{31.13}$$

Notice that if $\tilde{z}$ is a root of equation (31.13), then so is $\beta\tilde{z}^{-1}$.

Thus, the roots of (31.13) come in "$\beta$-reciprocal" pairs.

Assume that the roots of (31.13) 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 (31.13) can be expressed as

$$\begin{aligned} h + d(\beta z^{-1})d(z) &= z^{-m} z_0 (z - z_1) \cdots (z - z_m)(z - z_{m+1}) \cdots (z - z_{2m}) \\ &= z^{-m} z_0 (z - z_1)(z - z_2) \cdots (z - z_m)(z - \beta z_m^{-1}) \cdots (z - \beta z_2^{-1})(z - \beta z_1^{-1}) \end{aligned} \tag{31.14}$$

where $z_0$ is a constant.

In (31.14), 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

$$c(z) = \left[(-1)^m z_0 z_1 \cdots z_m\right]^{1/2} (1 - \frac{z}{z_1})(1 - \frac{z}{z_2}) \cdots (1 - \frac{z}{z_m}) \tag{31.15}$$

Notice that (31.14) can be written

$$h + d(\beta z^{-1}) d(z) = c(\beta z^{-1}) c(z) \tag{31.16}$$

It is useful to write (31.15) as

$$c(z) = c_0 (1 - \lambda_1 z) \ldots (1 - \lambda_m z) \tag{31.17}$$

where

$$c_0 = [(-1)^m z_0 z_1 \cdots z_m]^{1/2}; \quad \lambda_j = \frac{1}{z_j}, \ j = 1, \ldots, m$$

Since $|z_j| > \sqrt{\beta}$ for $j = 1, \ldots, m$ it follows that $|\lambda_j| < 1/\sqrt{\beta}$ for $j = 1, \ldots, m$.

Using (31.17), we can express the factorization (31.16) 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 (31.16) 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 (31.16), 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 (31.12) is

$$c(L) y_t = c(\beta L^{-1})^{-1} a_t \tag{31.18}$$

This can be established by using an argument paralleling that in chapter IX of [Sar87].

To exhibit the solution in a form paralleling that of [Sar87], we use (31.17) to write (31.18) as

$$(1-\lambda_1 L)\cdots(1-\lambda_m L)y_t = \frac{c_0^{-2}a_t}{(1-\beta\lambda_1 L^{-1})\cdots(1-\beta\lambda_m L^{-1})} \tag{31.19}$$

Using partial fractions, we can write the characteristic polynomial on the right side of (31.19) 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 (31.19) 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

$$(1-\lambda_1 L)\cdots(1-\lambda_m L)y_t = \sum_{j=1}^{m}A_j\sum_{k=0}^{\infty}(\lambda_j\beta)^k\,a_{t+k} \tag{31.20}$$

Equation (31.20) 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, (31.20) is the unique solution of the Euler equation that satisfies the initial conditions and condition (31.12).

In effect, condition (31.12) compels us to solve the "unstable" roots of $h + d(\beta z^{-1})d(z)$ forward (see [Sar87]).

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 (31.20)

- 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 (31.20) converge.

We immediately see that those sums will converge under the weaker condition that $\{a_t\}$ is of exponential order less than $\phi^{-1}$ where $\phi = \max\{\beta\lambda_i, i = 1, \dots, m\}$.

Note that with $a_t$ identically zero, (31.20) 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 (31.12) is satisfied.

In fact, $\max_i |\lambda_i| < 1/\sqrt{\beta}$ is a necessary condition for (31.12) to hold.

Were (31.12) 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 (31.17).

We can simply let $y_t = 0$ for $t \geq 0$.

This policy involves at most $m$ nonzero values of $hy_t^2$ and $[d(L)y_t]^2$, and so yields a finite value of (31.1).

Therefore it is easy to dominate a path that violates (31.12).

## 31.5 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 (31.16) becomes

$$\left(h + d\left(z^{-1}\right)d(z)\right) = c\left(z^{-1}\right) c\left(z\right)$$

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

$$(1-\lambda_1 L)\cdots(1-\lambda_m L)y_t = \sum_{j=1}^{m} A_j \sum_{k=0}^{\infty} \lambda_j^k a_{t+k}$$

### 31.5.1 Transforming Discounted to Undiscounted Problem

Discounted problems can always be converted into undiscounted problems via a simple transformation.

Consider problem (31.1) with $0 < \beta < 1$.

Define the transformed variables

$$\tilde{a}_t = \beta^{t/2} a_t,\ \tilde{y}_t = \beta^{t/2} y_t \tag{31.21}$$

Then notice that $\beta^t \left[d\left(L\right)y_t\right]^2 = \left[\tilde{d}\left(L\right)\tilde{y}_t\right]^2$ 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 (31.1) is equivalent to

$$\lim_{N\to\infty} \sum_{t=0}^{N} \{\tilde{a}_t\, \tilde{y}_t - \frac{1}{2} h\, \tilde{y}_t^2 - \frac{1}{2}[\tilde{d}\left(L\right)\tilde{y}_t]^2\} \tag{31.22}$$

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 $\left[h + \tilde{d}\left(L^{-1}\right)\tilde{d}\left(L\right)\right]\tilde{y}_t = \tilde{a}_t$.

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

$$\tilde{y}_t = \tilde{f}_1 \, \tilde{y}_{t-1} + \cdots + \tilde{f}_m \, \tilde{y}_{t-m} + \sum_{j=1}^{m} \tilde{A}_j \sum_{k=0}^{\infty} \tilde{\lambda}_j^k \, \tilde{a}_{t+k}, \tag{31.23}$$

where $\tilde{c}\,(z^{-1})\tilde{c}\,(z) = h + \tilde{d}\,(z^{-1})\tilde{d}\,(z)$, and where

$$\left[(-1)^m \, \tilde{z}_0 \tilde{z}_1 \ldots \tilde{z}_m\right]^{1/2} (1 - \tilde{\lambda}_1 \, z) \ldots (1 - \tilde{\lambda}_m \, z) = \tilde{c}\,(z), \text{ where } |\tilde{\lambda}_j| < 1$$

We leave it to the reader to show that (31.23) 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

$$f_j = \tilde{f}_j \, \beta^{-j/2}, \; A_j = \tilde{A}_j, \; \lambda_j = \tilde{\lambda}_j \, \beta^{-1/2} \tag{31.24}$$

The transformations (31.21) and the inverse formulas (31.24) allow us to solve a discounted problem by first solving a related undiscounted problem.

## 31.6 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
```

(continues on next page)

(continued from previous page)

```
        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:
            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
        """
```

(continues on next page)

(continued from previous page)

```
        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

        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
```

(continues on next page)

(continued from previous page)

```
        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
            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
```

(continues on next page)

(continued from previous page)

```
        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]
        """

        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
```

(continues on next page)

(continued from previous page)

```
        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}

            # 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
```

### 31.6.1 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 $\gamma$, 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
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 $\gamma$ to 5.0

```
plot_simulation(γ=5)
```

And here's $\gamma = 10$

```
plot_simulation(γ=10)
```

## 31.7 Exercises

### 31.7.1 Exercise 1

Consider solving a discounted version $(\beta < 1)$ of problem (31.1), as follows.

Convert (31.1) to the undiscounted problem (31.22).

Let the solution of (31.22) 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

$$\tilde{y}_t = \tilde{f}_1 \tilde{y}_{t-1} + \cdots + \tilde{f}_m \tilde{y}_{t-m} + \sum_{j=1}^{m} \tilde{A}_j \sum_{k=0}^{\infty} \tilde{\lambda}_j^k \tilde{a}_{t+k} \tag{31.25}$$

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 (31.25) implies that the solution for $y_t$ in feedback form is

$$y_t = f_1 y_{t-1} + ... + f_m y_{t-m} + \sum_{j=1}^{m} A_j \sum_{k=0}^{\infty} \beta^k \lambda_j^k a_{t+k}$$

where $f_j = \tilde{f}_j \beta^{-j/2}$, $A_j = \tilde{A}_j$, and $\lambda_j = \tilde{\lambda}_j \beta^{-1/2}$.

### 31.7.2 Exercise 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" (31.20), giving numerical values for the coefficients.

Make sure that the boundary conditions (31.5) are satisfied.

(Note: this problem differs from the problem in the text in one important way: instead of $h > 0$ in (31.1), $h = 0$. This has an important influence on the solution.)

### 31.7.3 Exercise 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

$$y_t = 2y_{t-1} = 2^{t+1} y_{-1} \qquad t > 0$$

### 31.7.4 Exercise 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 (31.12), and so is not optimal.

Prove that the optimal solution is approximately $y_t = .5y_{t-1}$.

CHAPTER

THIRTYTWO

# CLASSICAL PREDICTION AND FILTERING WITH LINEAR ALGEBRA

**Contents**


- *Classical Prediction and Filtering With Linear Algebra*
  - *Overview*
  - *Finite Dimensional Prediction*
  - *Combined Finite Dimensional Control and Prediction*
  - *Infinite Horizon Prediction and Filtering Problems*
  - *Exercises*


## 32.1 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 [Sar87] [chs. IX, XIV] and Hansen and Sargent [HS80] 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
```

### 32.1.1 References

Useful references include [Whi63], [HS80], [Orf88], [AP91], and [Mut60].

## 32.2 Finite Dimensional Prediction

Let $(x_1, x_2, \dots, x_T)' = x$ 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 (32.16) 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

$$\hat{\mathbb{E}}\left[x_T | x_{T-j}, x_{T-j+1}, \dots, x_1\right]$$

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 $\varepsilon$ 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 $\varepsilon$ is an orthonormal basis for $x$ because

- $L$ is nonsingular
- $\mathbb{E}\,\varepsilon\,\varepsilon' = L\mathbb{E}xx'L' = I$
- $x = L^{-1}\varepsilon$

It is enlightening to write out and interpret the equations $Lx = \varepsilon$ and $L^{-1}\varepsilon = x$.

First, we'll write $Lx = \varepsilon$

$$\begin{aligned} L_{11}x_1 &= \varepsilon_1 \\ L_{21}x_1 + L_{22}x_2 &= \varepsilon_2 \\ &\vdots \\ L_{T1}\,x_1 \;\ldots\; + L_{TT}x_T &= \varepsilon_T \end{aligned} \tag{32.1}$$

or

$$\sum_{j=0}^{t-1} L_{t,t-j}\,x_{t-j} = \varepsilon_t, \quad t = 1,\,2,\ldots T \tag{32.2}$$

Next, we write $L^{-1}\varepsilon = x$

$$\begin{aligned} x_1 &= L_{11}^{-1}\varepsilon_1 \\ x_2 &= L_{22}^{-1}\varepsilon_2 + L_{21}^{-1}\varepsilon_1 \\ &\vdots \\ x_T &= L_{TT}^{-1}\varepsilon_T + L_{T,T-1}^{-1}\varepsilon_{T-1} \;\ldots\; + L_{T,1}^{-1}\varepsilon_1 \end{aligned}, \tag{32.3}$$

or

$$x_t = \sum_{j=0}^{t-1} L_{t,t-j}^{-1}\,\varepsilon_{t-j} \tag{32.4}$$

where $L_{i,j}^{-1}$ denotes the $i,j$ element of $L^{-1}$.

From (32.2), it follows that $\varepsilon_t$ is in the linear subspace spanned by $x_t,\, x_{t-1}, \ldots,\, x_1$.

From (32.4) it follows that that $x_t$ is in the linear subspace spanned by $\varepsilon_t,\, \varepsilon_{t-1}, \ldots, \varepsilon_1$.

Equation (32.2) 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 $t-1 \geq m \geq 1$

$$\hat{\mathbb{E}}[x_t \mid x_{t-m},\, x_{t-m-1}, \ldots, x_1] = \hat{\mathbb{E}}[x_t \mid \varepsilon_{t-m}, \varepsilon_{t-m-1}, \ldots, \varepsilon_1] \tag{32.5}$$

To proceed, it is useful to drill down and note that for $t-1 \geq m \geq 1$ we can rewrite (32.4) in the form of the **moving average representation**

$$x_t = \sum_{j=0}^{m-1} L_{t,t-j}^{-1}\,\varepsilon_{t-j} + \sum_{j=m}^{t-1} L_{t,t-j}^{-1}\,\varepsilon_{t-j} \tag{32.6}$$

Representation (32.6) 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 $[x_{t-m}, x_{t-m+1}, \dots, x_1]$ 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

$$\hat{\mathbb{E}}[x_t \mid x_{t-m}, x_{t-m-1}, \dots, x_1] = \sum_{j=0}^{m-1} L_{t,t-j}^{-1} \varepsilon_{t-j}$$

### 32.2.1 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):
```

(continues on next page)

(continued from previous page)

```
            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:
            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))
```

(continues on next page)

(continued from previous page)

```
        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

        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)
```

(continues on next page)

(continued from previous page)

```
        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
            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
        """
```

(continues on next page)

(continued from previous page)

```
        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]
        """

        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)
```

(continues on next page)

(continued from previous page)

```
        #-------------------------------------------
        # 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}

        # 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.

### 32.2.2 Example 1

Consider a stochastic process with moving average representation

$$x_t = (1 - 2L)\varepsilon_t$$

where $\varepsilon_t$ 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 (32.21) to compute the linear least squares forecasts $\mathbb{E}[x_{t+j} \mid x_t, x_{t-1}, \ldots]$, 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]]
```

### 32.2.3 Example 2

Consider a stochastic process $X_t$ with moving average representation

$$X_t = (1 - \sqrt{2}L^2)\varepsilon_t$$

where $\varepsilon_t$ 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 (32.21) presented below to compute the linear least squares forecasts $\hat{\mathbb{E}}\left[X_{t+j} \mid X_{t-1}, \ldots\right]$ 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([ 1.41421356e+00, -1.33226763e-15, -1.00000000e+00])
```

```
example.roots_of_characteristic()
```

```
(array([-1.18920712,  1.18920712]),
 -1.4142135623731136,
 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.
```

(continues on next page)

(continued from previous page)

```
   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]]
```

### 32.2.4 Prediction

It immediately follows from the "orthogonality principle" of least squares (see [AP91] or [Sar87] [ch. X]) that

$$
\begin{aligned}
\hat{\mathbb{E}}[x_t \mid x_{t-m}, x_{t-m+1}, \dots x_1] &= \sum_{j=m}^{t-1} L^{-1}_{t,t-j}\, \varepsilon_{t-j} \\
&= [L^{-1}_{t,1}\, L^{-1}_{t,2}, \dots, L^{-1}_{t,t-m}\; 0\; 0 \dots 0] L\, x
\end{aligned}
\tag{32.7}
$$

This can be interpreted as a finite-dimensional version of the Wiener-Kolmogorov $m$-step ahead prediction formula.

We can use (32.7) 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

$$
\hat{\mathbb{E}}[x \mid x_s, x_{s-1}, \dots, x_1] = L^{-1} \begin{bmatrix} I_s & 0 \\ 0 & 0_{(t-s)} \end{bmatrix} Lx \tag{32.8}
$$

This formula will be convenient in representing the solution of control problems under uncertainty.

Equation (32.4) can be recognized as a finite dimensional version of a moving average representation.

Equation (32.2) 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.

## 32.3 Combined Finite Dimensional Control and Prediction

Consider the finite-dimensional control problem, maximize

$$\mathbb{E} \sum_{t=0}^{N} \left\{ a_t y_t - \frac{1}{2} h y_t^2 - \frac{1}{2} [d(L) y_t]^2 \right\}, \qquad h > 0$$

where $d(L) = d_0 + d_1 L + \ldots + d_m L^m$, $L$ is the lag operator, $\bar{a} = [a_N, a_{N-1} \ldots, a_1, a_0]'$ a random vector with mean zero and $\mathbb{E}\, \bar{a}\bar{a}' = V$.

The variables $y_{-1}, \ldots, y_{-m}$ are given.

Maximization is over choices of $y_0, y_1 \ldots, y_N$, 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 (32.7), we can express $\hat{\mathbb{E}}[\bar{a} \mid a_s,\ a_{s-1}, \ldots, a_0]$ as

$$\hat{\mathbb{E}}[\bar{a} \mid a_s,\ a_{s-1}, \ldots, a_0] = \tilde{U}^{-1} \begin{bmatrix} 0 & 0 \\ 0 & I_{(s+1)} \end{bmatrix} \tilde{U}\bar{a}$$

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 (32.8) 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

$$U\bar{y} = L^{-1}\tilde{U}^{-1}\begin{bmatrix} 0 & 0 \\ 0 & I_{(t+1)} \end{bmatrix}\tilde{U}\bar{a} + K\begin{bmatrix} y_{-1} \\ \vdots \\ y_{-m} \end{bmatrix}$$

## 32.4 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 $m^{\text{th}}$ order moving average, covariance stationary stochastic process,

$$Y_t = d(L)u_t \tag{32.9}$$

where $d(L) = \sum_{j=0}^{m} d_j L^j$, and $u_t$ is a serially uncorrelated stationary random process satisfying

$$\begin{aligned} \mathbb{E}u_t &= 0 \\ \mathbb{E}u_t u_s &= \begin{cases} 1 & \text{if } t = s \\ 0 & \text{otherwise} \end{cases} \end{aligned} \tag{32.10}$$

We impose no conditions on the zeros of $d(z)$.

A second covariance stationary process is $X_t$ given by

$$X_t = Y_t + \varepsilon_t \tag{32.11}$$

where $\varepsilon_t$ 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 [Mut60].

### 32.4.1 Problem Formulation

These problems are solved as follows.

The covariograms of $Y$ and $X$ and their cross covariogram are, respectively,

$$\begin{aligned} C_X(\tau) &= \mathbb{E}X_t X_{t-\tau} \\ C_Y(\tau) &= \mathbb{E}Y_t Y_{t-\tau} \qquad \tau = 0, \pm 1, \pm 2, ... \\ C_{Y,X}(\tau) &= \mathbb{E}Y_t X_{t-\tau} \end{aligned} \tag{32.12}$$

The covariance and cross-covariance generating functions are defined as

$$
\begin{aligned}
g_X(z) &= \sum_{\tau=-\infty}^{\infty} C_X(\tau) z^\tau \\
g_Y(z) &= \sum_{\tau=-\infty}^{\infty} C_Y(\tau) z^\tau \\
g_{YX}(z) &= \sum_{\tau=-\infty}^{\infty} C_{YX}(\tau) z^\tau
\end{aligned}
\tag{32.13}
$$

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 [Sar87] [ch. XI], it is true that

$$
\begin{aligned}
g_y(z) &= A(z)A(z^{-1}) + B(z)B(z^{-1}) \\
g_x(z) &= C(z)C(z^{-1}) + D(z)D(z^{-1}) \\
g_{yx}(z) &= A(z)C(z^{-1}) + B(z)D(z^{-1})
\end{aligned}
\tag{32.14}
$$

Applying these formulas to (32.9) – (32.12), we have

$$
\begin{aligned}
g_Y(z) &= d(z)d(z^{-1}) \\
g_X(z) &= d(z)d(z^{-1}) + h \\
g_{YX}(z) &= d(z)d(z^{-1})
\end{aligned}
\tag{32.15}
$$

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,

$$
X_t = c\left(L\right)\eta_t \tag{32.16}
$$

where $c(L) = \sum_{j=0}^{m} c_j\, L^j$, with

$$
c_0\eta_t = X_t - \hat{\mathbb{E}}[X_t | X_{t-1}, X_{t-2}, ...] \tag{32.17}
$$

Here $\hat{\mathbb{E}}$ is the linear least squares projection operator.

Equation (32.17) 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 (32.17) requires that $\eta_t$ 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 (32.17) that $\eta_t$ is a serially uncorrelated random process and that normalization can be imposed so that $\mathbb{E}\eta_t^2 = 1$.

Consequently, an implication of (32.16) is that the covariance generating function of $X_t$ can be expressed as

$$
g_X(z) = c\left(z\right)c\left(z^{-1}\right) \tag{32.18}
$$

It remains to discuss how $c(L)$ is to be computed.

Combining (32.14) and (32.18) gives

$$d(z)\,d(z^{-1}) + h = c\,(z)\,c\,(z^{-1}) \tag{32.19}$$

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**:

$$\left[\sum_{j=-\infty}^{\infty} f_j\,L^j\right]_+ \equiv \sum_{j=0}^{\infty} f_j\,L^j \tag{32.20}$$

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:

$$\gamma_j\,(L) = \left[\frac{c(L)}{L^j}\right]_+ c\,(L)^{-1} \tag{32.21}$$

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

$$b(L) = \left[\frac{d(L)d(L^{-1})}{c(L^{-1})}\right]_+ c(L)^{-1} \tag{32.22}$$

Formulas (32.21) and (32.22) are discussed in detail in [Whi83] and [Sar87].

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 [Mut60].

As an example of the usefulness of formula (32.22), 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 (32.22) to obtain the answer.

Using the standard formulas (32.14), we have that

$$\begin{aligned} g_{yx}(z) &= (1-\delta z^{-1})c(z)c(z^{-1}) \\ g_x(z) &= c(z)c(z^{-1}) \end{aligned}$$

Then (32.22) becomes

$$b(L) = \left[\frac{c(L)}{1-\delta L^{-1}}\right]_+ c(L)^{-1} \tag{32.23}$$

In order to evaluate the term in the annihilation operator, we use the following result from [HS80].

**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

$$\left[\frac{g(z)}{h(z^{-1})}\right]_+ = \frac{g(z)}{h(z^{-1})} - \sum_{j=1}^{n} \frac{\delta_j g(\delta_j)}{\prod_{\substack{k=1\\k\neq j}}^{n} (\delta_j - \delta_k)} \left(\frac{1}{z-\delta_j}\right) \tag{32.24}$$

and, alternatively,

$$\left[\frac{g(z)}{h(z^{-1})}\right]_+ = \sum_{j=1}^{n} B_j \left(\frac{zg(z) - \delta_j g(\delta_j)}{z-\delta_j}\right) \tag{32.25}$$

where $B_j = 1/\prod_{\substack{k=1\\k+j}}^{n} (1-\delta_k/\delta_j)$.

Applying formula (32.25) of the proposition to evaluating (32.23) 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

$$\hat{\mathbb{E}}\left[\sum_{j=0}^{\infty} \delta^j X_{t+j} | X_t,\, x_{t-1}, \ldots\right] = \left[\frac{1-\delta c(\delta) L^{-1} c(L)^{-1}}{1-\delta L^{-1}}\right] X_t \tag{32.26}$$

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

$$\mathbb{E}_0 \lim_{N\to\infty} \sum_{t-0}^{N} \beta^t \left[a_t\, y_t - \frac{1}{2}\, h y_t^2 - \frac{1}{2}\, [d(L) y_t]^2\right] \tag{32.27}$$

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

$$\eta_t = a_t - \hat{\mathbb{E}}[a_t | a_{t-1}, \ldots]$$

The problem is to maximize (32.27) with respect to a contingency plan expressing $y_t$ as a function of information known at $t$, which is assumed to be $(y_{t-1},\ y_{t-2}, \ldots, a_t,\ a_{t-1}, \ldots)$.

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

$$(1 - \lambda_1 L) \ldots (1 - \lambda_m L) y_t = \sum_{j=1}^{m} A_j \sum_{k=0}^{\infty} (\lambda_j \beta)^k \, a_{t+k} \tag{32.28}$$

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 (32.26) and (32.28), we have the following solution

$$(1 - \lambda_1 L) \ldots (1 - \lambda_m L) y_t = \sum_{j=1}^{m} A_j \left[ \frac{1 - \beta \lambda_j \, c(\beta \lambda_j) L^{-1} c(L)^{-1}}{1 - \beta \lambda_j L^{-1}} \right] a_t$$

**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 $z_1, \ldots, z_k$ be the zeros of $\pi(z)$ that are inside the unit circle, $k < m$.

Then define

$$\theta(z) = \pi(z) \left( \frac{(z_1 z - 1)}{(z - z_1)} \right) \left( \frac{(z_2 z - 1)}{(z - z_2)} \right) \ldots \left( \frac{(z_k z - 1)}{(z - z_k)} \right)$$

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.

## 32.5 Exercises

### 32.5.1 Exercise 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 $\varepsilon_t$ 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

$$\hat{\mathbb{E}} X_{t+2} \mid X_t, X_{t-1}, ... = \sum_{j=0}^{\infty} A_{2j} X_{t-j}$$

### 32.5.2 Exercise 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 $\varepsilon_t$ 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 (31.21), only interpret all the objects in (31.21) 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 [Roz67] or [Whi83]) 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 $\eta_t$ 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

THIRTYTHREE

# KNOWING THE FORECASTS OF OTHERS

**Contents**


- *Knowing the Forecasts of Others*
  - *Introduction*
  - *The Setting*
  - *Tactics*
  - *Equilibrium conditions*
  - *Equilibrium with $\theta_t$ stochastic but observed at $t$*
  - *Guess-and-verify tactic*
  - *Equilibrium with one signal on $\theta_t$*
  - *Equilibrium with two noisy signals on $\theta_t$*
  - *Key step*
  - *Comparison of the two signal structures*
  - *Notes on History of the Problem*


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
```

## 33.1 Introduction

Robert E. Lucas, Jr. [REL75], Kenneth Kasa [Kas00], and Robert Townsend [Tow83] showed that giving decision makers incentives to infer persistent hidden state variables from equilibrium prices and quantities can elongate and amplify impulse responses to aggregate shocks in business cycle models.

Townsend [Tow83] noted that such incentives can naturally induce decision makers to want to forecast the forecast of others.

This theme has been pursued and extended in analyses in which decision makers' imperfect information forces them into pursuing an infinite recursion of forming beliefs about the beliefs of other (e.g., [AMS02]).

Lucas [REL75] 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, [Tow83] confronted the forecasting the forecasts of others problem.

To formulate the problem recursively required that Townsend define 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 the hidden Markov component of the demand shock is revealed to all firms after a fixed and finite number of periods.

In this lecture, as yet another instance of the theme that **finding the state is an art**, we show how to formulate Townsend's original model in terms of a low-dimensional state space.

By doing so, 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 culminated in [PS05] that built on [PCL86].

However, rather than deploying the [PCL86] machinery here, we shall rely instead on a sneaky **guess-and-verify** tactic.

- We 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 what it in 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 that firms in the other industry 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.

### 33.1.1 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 [Tow83].

Townsend's is a model of a rational expectations equilibrium in which firms confront the problem **forecasting the 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 start with model components that we shall progressively assemble in ways that can help us to appreciate the structure of a **pooling equilibrium** that ultimately concerns us.

While keeping 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.

## 33.2 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 each of two industries $i = 1, 2$ 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 $y_t^i = fk_t^i$, $f > 0$, acts as a price taker with respect to output price $P_t^i$, and maximizes

$$E_0^i \sum_{t=0}^{\infty} \beta^t \left\{ P_t^i f k_t^i - .5h(k_{t+1}^i - k_t^i)^2 \right\}, \quad h > 0. \tag{33.1}$$

Demand in industry $i$ is described by the inverse demand curve

$$P_t^i = -bY_t^i + \theta_t + \epsilon_t^i, \quad b > 0, \tag{33.2}$$

where $P_t^i$ is the price of good $i$ at $t$, $Y_t^i = fK_t^i$ is output in market $i$, $\theta_t$ is a persistent component of a demand shock that is common across the two industries, and $\epsilon_t^i$ is an industry specific component of the demand shock that is i.i.d. and whose time $t$ marginal distributon is $\mathcal{N}(0, \sigma_\epsilon^2)$.

We assume that $\theta_t$ is governed by

$$\theta_{t+1} = \rho\theta_t + v_t \tag{33.3}$$

where $\{v_t\}$ is an i.i.d. sequence of Gaussian shocks each with mean zero and variance $\sigma_v^2$.

To simplify notation, we'll study a special case of the model by setting $h = f = 1$.

The presence of costs of adjusting their capital stocks imparts to firms an incentives 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 continua of each type of firm, each indexed by $\omega \in [0, 1]$ with $K^i = \int_0^1 k^i(\omega) d\omega$.

In equilibrium, $k_t^i = K_t^i$, but as usual we must distinguish between $k_t^i$ and $K_t^i$ when we pose the firm's optimization problem.

## 33.3 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 information structure that that we are ultimately interested in.

Thus, we shall compute equilibria under the following alternative information structures:

- **Perfect foresight:** future values of $\theta_t, \epsilon_t^i$ are observed in industry $i$.
- **Observed but stochastic** $\theta_t$: $\{\theta_t, \epsilon_t^i\}$ 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** $\theta_t$: Values of $\{\theta_t, \epsilon_t^i\}$ separately are never observed. However, at time $t$, a history $w^t$ of a scalar noise-ridden observations on $\theta_t$ is observed at time $t$.
- **Two noise-ridden observations on** $\theta_t$: 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 $\theta_t$ is observed at time $t$.

Successive computations build one on another.

We proceed by first finding an equilibrium under perfect foresight.

To compute an equilibrium with $\theta_t$ 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 $\theta_t$.

This provides the equilibrium when $\theta_t$ is observed at $t$ but future $\theta_{t+j}$ and $\epsilon_{t+j}^i$ are not observed.

To find an equilibrium when only a history $w_t$ of a single noise ridden observations on $\theta_t$ 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 only a history $w_t$ of a *two* noisy signal on $\theta_t$ 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 $\theta_t$ 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 $\theta$.

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 interested [Tow83]:

- **Firm** $i$**'s noise-ridden signal on** $\theta_t$ **and the price in industry** $-i$, a firm in industry $i$ observes a history $w^t$ of *one* noise-ridden signal on $\theta_t$ and a history of industry $-i$'s price is observed.

With this information structure, the representative firm $i$ sees the price as well as the aggregate 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 firm $i$ 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 eventually verify this assertion by using a guess and verify tactic.[1]

[1] [PS05] verified this assertion using a different tactic, namely, by constructing analytic formulas an equilibrium under the incomplete information structure and confirming that they match the pooling equilibrium formulas derived here.

## 33.4 Equilibrium conditions

It is convenient to solve the 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 for $k_{t+1}^i$. First order conditions for the nonstochastic problem are

$$\begin{aligned} \phi_t^i &= \beta \phi_{t+1}^i + \beta P_{t+1}^i \\ \mu_t^i &= \phi_t^i. \end{aligned} \tag{33.4}$$

Substituting the demand function (33.2) for $P_t^i$, imposing the condition that the representative firm is representative ( $k_t^i = K_t^i$), and using the definition below of $g_t^i$, the Euler equation (33.4), 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

$$k_{t+1}^i = (b+1)k_t^i - \theta_t - \epsilon_t^i + g_t^i \tag{33.5}$$

where we define $g_t^i$ by

$$g_t^i = \beta^{-1}(k_t^i - k_{t-1}^i) \tag{33.6}$$

We can write Euler equation (33.4) as:

$$g_t^i = P_t^i + \beta g_{t+1}^i \tag{33.7}$$

In addition, we have the law of motion for $\theta_t$, (33.3), and the demand equation (33.2).

In summary, with perfect foresight, equilibrium conditions for industry $i$ include the following system of difference equations:

$$\begin{aligned} k_{t+1}^i &= (1+b)k_t^i - \epsilon_t^i - \theta_t + g_t^i \\ \theta_{t+1} &= \rho\theta_t + v_t \\ g_{t+1}^i &= \beta^{-1}(g_t^i - P_t^i) \\ P_t^i &= -bk_t^i + \epsilon_t^i + \theta_t \end{aligned} \tag{33.8}$$

Without perfect foresight, the same system prevails except that the following equation replaces the third equation of (33.8):

$$g_{t+1,t}^i = \beta^{-1}(g_t^i - P_t^i)$$

where $x_{t+1,t}$ denotes the mathematical expectation of $x_{t+1}$ conditional on information at time $t$.

### 33.4.1 Equilibrium under perfect foresight

Our first step is to compute the equilibrium law of motion for $k_t^i$ under perfect foresight.

Let $L$ be the lag operator.[2]

Equations (33.7) and (33.5) imply the second order difference equation in $k_t^i$:[3]

$$\left[ (L^{-1} - (1+b))(1 - \beta L^{-1}) + b \right] k_t^i = \beta L^{-1} \epsilon_t^i + \beta L^{-1} \theta_t. \tag{33.9}$$

[2] See [Sar87], especially chapters IX and XIV, for the principles that guide solving some roots backwards and others forwards.

[3] As noted [Sar87], this difference equation is the Euler equation for the planning problem of maximizing the discounted sum of consumer plus producer surplus.

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 $\lambda$ is the larger root of $(\lambda - 1)(\lambda - 1/\beta) = b\lambda$.

Therefore, (33.9) can be expressed as

$$\tilde{\lambda}^{-1}(L^{-1} - \tilde{\lambda})(1 - \tilde{\lambda}\beta L^{-1})k_t^i = \beta L^{-1}\epsilon_t^i + \beta L^{-1}\theta_t.$$

Solving the stable root backwards and the unstable root forwards gives

$$k_{t+1}^i = \tilde{\lambda}k_t^i + \frac{\tilde{\lambda}\beta}{1 - \tilde{\lambda}\beta L^{-1}}(\epsilon_{t+1}^i + \theta_{t+1}).$$

Recall that we have already set $k^i = K^i$ at the appropriate point in the argument (i.e., *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

$$k_{t+1}^i = \tilde{\lambda}k_t^i + \sum_{j=1}^{\infty}(\tilde{\lambda}\beta)^j(\epsilon_{t+j}^i + \theta_{t+j}). \tag{33.10}$$

Next, we shall investigate consequences of replacing future values of $(\epsilon_{t+j}^i + \theta_{t+j})$ in equation (33.10) with alternative forecasting schemes.

In particular, we shall compute equilibrium laws of motion for capital under alternative assumptions about the information available to decision makers in market $i$.

## 33.5 Equilibrium with $\theta_t$ stochastic but observed at $t$

If future $\theta$'s are unknown at $t$, it is appropriate to replace all random variables on the right side of (33.10) with their conditional expectations based on the information available to decision makers in market $i$.

For now, we assume that this information set $I_t^p = \begin{bmatrix}\theta^t & \epsilon^{it}\end{bmatrix}$, where $z^t$ represents the infinite history of variable $z_s$ up to time $t$.

Later we shall give firms less information.

To obtain an appropriate counterpart to (33.10) under our current assumption about information, we apply a certainty equivalence principle.

In particular, it is appropriate to take (33.10) 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 (33.3) 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

$$k_{t+1}^i = \tilde{\lambda}k_t^i + \frac{\rho}{\lambda - \rho}\theta_t \tag{33.11}$$

where $\lambda \equiv (\beta\tilde{\lambda})^{-1}$.

For future purposes, it is useful to represent the equilibrium $\{k_t^i\}_t$ process recursively as

$$\begin{aligned} k_{t+1}^i &= \tilde{\lambda}k_t^i + \frac{1}{\lambda - \rho}\hat{\theta}_{t+1} \\ \hat{\theta}_{t+1} &= \rho\theta_t \\ \theta_{t+1} &= \rho\theta_t + v_t. \end{aligned} \tag{33.12}$$

### 33.5.1 Filtering

#### One noisy signal

We get closer to a model that we ultimately want to study by now assuming that firms in market $i$ do not observe $\theta_t$, but instead observe a history $w^t$ of noisy signals at time $t$.

In particular, assume that

$$\begin{aligned} w_t &= \theta_t + e_t \\ \theta_{t+1} &= \rho\theta_t + v_t \end{aligned} \tag{33.13}$$

where $e_t$ and $v_t$ are mutually independent i.i.d. Gaussian shock processes with means of zero and variances $\sigma_e^2$ and $\sigma_v^2$, 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 (33.13) is the *innovations representation*

$$\begin{aligned} \hat{\theta}_{t+1} &= \rho\hat{\theta}_t + ka_t \\ w_t &= \hat{\theta}_t + a_t \end{aligned} \tag{33.14}$$

where $a_t \equiv w_t - E(w_t|w^{t-1})$ is the *innovations* process in $w_t$ and the Kalman gain $k$ is

$$k = \frac{\rho p}{p + \sigma_e^2} \tag{33.15}$$

and where $p$ satisfies the Riccati equation

$$p = \sigma_v^2 + \frac{p\rho^2\sigma_e^2}{\sigma_e^2 + p}. \tag{33.16}$$

#### $\theta$-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 (33.13) and (33.14) imply

$$\tilde{\theta}_{t+1} = (\rho - k)\tilde{\theta}_t + v_t - ke_t. \tag{33.17}$$

Now notice that we can express $\hat{\theta}_{t+1}$ as

$$\hat{\theta}_{t+1} = [\rho\theta_t + v_t] + [ke_t - (\rho - k)\tilde{\theta}_t - v_t], \tag{33.18}$$

where the first term in braces equals $\theta_{t+1}$ and the second term in braces equals $-\tilde{\theta}_{t+1}$.

We can express (33.11) as

$$k_{t+1}^i = \tilde{\lambda}k_t^i + \frac{1}{\lambda - \rho}E\theta_{t+1}|\theta^t. \tag{33.19}$$

An application of a certainty equivalence principle asserts that when only $w^t$ is observed, the appropriate solution is found by replacing the information set $\theta^t$ with $w^t$ in (33.19).

Making this substitution and using (33.18) leads to

$$k_{t+1}^i = \tilde{\lambda} k_t^i + \frac{\rho}{\lambda - \rho}\theta_t + \frac{k}{\lambda - \rho}e_t - \frac{\rho - k}{\lambda - \rho}\tilde{\theta}_t. \tag{33.20}$$

Simplifying equation (33.18), we also have

$$\hat{\theta}_{t+1} = \rho\theta_t + ke_t - (\rho - k)\tilde{\theta}_t. \tag{33.21}$$

Equations (33.20), (33.21) describe the equilibrium when $w^t$ is observed.

Relative to (33.11), the equilibrium acquires a new state variable, namely, the $\theta$–reconstruction error, $\tilde{\theta}_t$.

For future purposes, by using (33.15), it is useful to write (33.20) as

$$k_{t+1}^i = \tilde{\lambda} k_t^i + \frac{\rho}{\lambda - \rho}\theta_t + \frac{1}{\lambda - \rho}\frac{p\rho}{p + \sigma_e^2}e_t - \frac{1}{\lambda - \rho}\frac{\rho\sigma_e^2}{p + \sigma_e^2}\tilde{\theta}_t \tag{33.22}$$

In summary, when decision makers in market $i$ observe a noisy signal $w_t$ on $\theta_t$ at $t$, we can represent an equilibrium law of motion for $k_t^i$ as

$$\begin{aligned}
k_{t+1}^i &= \tilde{\lambda} k_t^i + \frac{1}{\lambda - \rho}\hat{\theta}_{t+1} \\
\hat{\theta}_{t+1} &= \rho\theta_t + \frac{\rho p}{p + \sigma_e^2}e_t - \frac{\rho\sigma_e^2}{p + \sigma_e^2}\tilde{\theta}_t \\
\tilde{\theta}_{t+1} &= \frac{\rho\sigma_e^2}{p + \sigma_e^2}\tilde{\theta}_t - \frac{p\rho}{p + \sigma_e^2}e_t + v_t \\
\theta_{t+1} &= \rho\theta_t + v_t.
\end{aligned} \tag{33.23}$$

### 33.5.2 Two noisy signals

We now construct a **pooling equilibrium** by assuming that a firm in industry $i$ receives a vector $w_t$ of *two* noisy signals on $\theta_t$:

$$\begin{aligned}
\theta_{t+1} &= \rho\theta_t + v_t \\
w_t &= \begin{bmatrix} 1 \\ 1 \end{bmatrix}\theta_t + \begin{bmatrix} e_{1t} \\ e_{2t} \end{bmatrix}
\end{aligned}$$

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 $\theta_t$ presented to firms in both industries $i$ and $-i$.

The appropriate innovations representation becomes

$$\begin{aligned}
\hat{\theta}_{t+1} &= \rho\hat{\theta}_t + ka_t \\
w_t &= \begin{bmatrix} 1 \\ 1 \end{bmatrix}\hat{\theta}_t + a_t
\end{aligned} \tag{33.24}$$

where $a_t \equiv w_t - E[w_t|w^{t-1}]$ is a $(2 \times 1)$ vector of innovations in $w_t$ and $k$ is now a $(1 \times 2)$ vector of Kalman gains.

Formulas for the Kalman filter imply that

$$k = \frac{\rho p}{2p + \sigma_e^2} \begin{bmatrix} 1 & 1 \end{bmatrix} \tag{33.25}$$

where $p = E\tilde{\theta}_t \tilde{\theta}_t^T$ now satisfies the Riccati equation

$$p = \sigma_v^2 + \frac{p\rho^2\sigma_e^2}{2p + \sigma_e^2}. \tag{33.26}$$

Thus, when a representative firm in industry $i$ observes *two* noisy signals on $\theta_t$, we can express the equilibrium law of motion for capital recursively as

$$\begin{aligned}
k_{t+1}^i &= \tilde{\lambda} k_t^i + \frac{1}{\lambda - \rho} \hat{\theta}_{t+1} \\
\hat{\theta}_{t+1} &= \rho\theta_t + \frac{\rho p}{2p + \sigma_e^2}(e_{1t} + e_{2t}) - \frac{\rho\sigma_e^2}{2p + \sigma_e^2}\tilde{\theta}_t \\
\tilde{\theta}_{t+1} &= \frac{\rho\sigma_e^2}{2p + \sigma_e^2}\tilde{\theta}_t - \frac{p\rho}{2p + \sigma_e^2}(e_{1t} + e_{2t}) + v_t \\
\theta_{t+1} &= \rho\theta_t + v_t.
\end{aligned} \tag{33.27}$$

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 [Tow83].[4]

## 33.6 Guess-and-verify tactic

As a preliminary step we shall take our recursive representation (33.23) of an equilibrium in industry $i$ with one noisy signal on $\theta_t$ and perform the following steps:

- Compute $\lambda$ and $\tilde{\lambda}$ by posing a root-finding problem and then solving it using `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 (33.23). 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.

## 33.7 Equilibrium with one signal on $\theta_t$

### 33.7.1 Step 1: Solve for $\tilde{\lambda}$ and $\lambda$

1. Cast $(\lambda - 1)\left(\lambda - \frac{1}{\beta}\right) = b\lambda$ as $p(\lambda) = 0$ where $p$ is a polynomial function of $\lambda$.
2. Use `numpy.roots` to solve for the roots of $p$

[4] [PS05] verify the same claim by applying machinery of [PCL86].

3. Verify $\lambda \approx \frac{1}{\beta\tilde{\lambda}}$

Note that $p(\lambda) = \lambda^2 - \left(1 + b + \frac{1}{\beta}\right)\lambda + \frac{1}{\beta}$.

### 33.7.2 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 $p \approx \sigma_v^2 + \frac{p\rho^2\sigma_e^2}{2p+\sigma_e^2}$

Note that:

$$
\begin{aligned}
A &= \begin{bmatrix} \rho \end{bmatrix} \\
B &= \begin{bmatrix} \sqrt{2} \end{bmatrix} \\
R &= \begin{bmatrix} \sigma_e^2 \end{bmatrix} \\
Q &= \begin{bmatrix} \sigma_v^2 \end{bmatrix} \\
N &= \begin{bmatrix} 0 \end{bmatrix}
\end{aligned}
$$

### 33.7.3 Step 3: Represent the system using `quantecon.LinearStateSpace`

We use the following representation for constructing the `quantecon.LinearStateSpace` instance.

$$
\underbrace{\begin{bmatrix} e_{t+1} \\ k_{t+1}^i \\ \tilde{\theta}_{t+1} \\ P_{t+1} \\ \theta_{t+1} \\ v_{t+1} \end{bmatrix}}_{x_{t+1}} = \underbrace{\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{\kappa}{\lambda-\rho} & \tilde{\lambda} & \frac{-1}{\lambda-\rho}\frac{\kappa\sigma_e^2}{p} & 0 & \frac{\rho}{\lambda-\rho} & 0 \\ -\kappa & 0 & \frac{\kappa\sigma_e^2}{p} & 0 & 0 & 1 \\ \frac{b\kappa}{\lambda-\rho} & b\tilde{\lambda} & \frac{-b}{\lambda-\rho}\frac{\kappa\sigma_e^2}{p} & 0 & \frac{b\rho}{\lambda-\rho}+\rho & 1 \\ 0 & 0 & 0 & 0 & \rho & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} e_t \\ k_t^i \\ \tilde{\theta}_t \\ P_t \\ \theta_t \\ v_t \end{bmatrix}}_{x_t} + \underbrace{\begin{bmatrix} \sigma_e & 0 \\ 0 & 0 \\ 0 & 0 \\ \sigma_e & 0 \\ 0 & 0 \\ 0 & \sigma_v \end{bmatrix}}_{C} \begin{bmatrix} z_{1,t+1} \\ z_{2,t+1} \end{bmatrix}
$$

$$
\underbrace{\begin{bmatrix} P_t \\ e_t + \theta_t \\ e_t \end{bmatrix}}_{y_t} = \underbrace{\begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}}_{G} \underbrace{\begin{bmatrix} e_t \\ k_t^i \\ \tilde{\theta}_t \\ P_t \\ \theta_t \\ v_t \end{bmatrix}}_{x_t} + \underbrace{\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}}_{H} w_{t+1}
$$

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

$$
\kappa = \frac{\rho p}{p + \sigma_e^2}
$$

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
from plotly.subplots import make_subplots
import plotly.graph_objects as go
import plotly.express as px
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
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
```

(continues on next page)

(continued from previous page)

```
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]])

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
```

### 33.7.4 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 the result.

```
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
```

### 33.7.5 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 some population regression coefficients.

$$\Sigma_x = \begin{bmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{bmatrix}$$

where $\Sigma_{11}$ is the covariance matrix of dependent variables and $\Sigma_{22}$ 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 $\beta$ that we shall compare to the 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: -3.2755568452197705
\tilde{\theta_t}: -0.964946117047546
P_t: 0.9649461170475461
```

```
# 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 9_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
```

## 33.8 Equilibrium with two noisy signals on $\theta_t$

Steps 1, 4, and 5 are identical to those for the one-noisy-signal structure.

Step 2 requires only a straightforward modification.

For step 3, we use 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 $\theta$ from the other industry that a firm receives directly in a pooling equilibrium on the information that a firm receives in Townsend's original model.

For this purpose, we include equilibrium goods prices from both industries appear 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: -3.1373589171035654
\tilde{\theta_t}: -0.924234396744368
P^{1}_t: -0.037882801627815835
P^{2}_t: 0.9621171983721839
```

```
# Compute R squared
R_squared = reg_coeffs @ Σ_x[2:6, 2:6] @ reg_coeffs  / Σ_x[1, 1]
R_squared
```

```
0.9621171983721838
```

## 33.9 Key step

Now we come to the key step of verifying that equilibrium outcomes for prices and quantities are identical in the pooling equilibrium and Townsend's original model.

We accomplish this by compute 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 receives 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.

## 33.10 Comparison of the two signal structures

It is enlightening side by side to plot impulse response functions for capital in an industry for the two information noisy-signal information structures.

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]).set_
↪subplots(1, 2,
                                                                                 ␣
↪    subplot_titles=("One noisy-signal structure", "Two noisy-signal structure"),
                                                                                 ␣
↪    horizontal_spacing=0.1,
                                                                                 ␣
↪    shared_yaxes=True)
```

(continues on next page)

(continued from previous page)

```
# Export to PNG file
Image(fig_comb.to_image(format="png"))
# fig_comb.show() will provide interactive plot when running
# notebook locally
```

The graphs above show that

- the response of $k_t^i$ to shocks $v_t$ to the hidden Markov demand state $\theta_t$ process is **larger** in the two-noisy=signal structure
- the response of $k_t^i$ to idiosyncratic *own-market* noise-shocks $e_t$ is **smaller** in the two-noisy-signal structure

Taken together, these findings in turn can be shown to imply that time series correlations and coherences between outputs in the two industries are higher in the two-noisy-signals or **pooling** model.

The enhanced influence of the shocks $v_t$ to the hidden Markov demand state $\theta_t$ 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**

$$Onenoisy - signalstructure : 0.403404$$

$$Twonoisy - signalsstructure : 0.25716$$

## 33.11 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 an 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 $\theta_{t-S}$.

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 [PCL86], [PS05] showed that there is a recursive representation of the equilibrium of the perpetually and symmetrically uninformed model formulated but not completely solved in section 8 of [Tow83].

A reader of [PS05] 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 [PS05].

The computational method of [PS05] is recursive: it enlists the Kalman filter and invariant subspace methods for solving systems of Euler equations[5] .

As [Sin87], [Kas00], and [Sar91] 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: seeing 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.

They know those forecasts because they are the same as their own.

[5] See [AHMS96] for an account of invariant subspace methods.

[6] See [AHMS96] for a discussion of the information assumptions needed to create a situation in which higher order beliefs appear in equilibrium decision rules. The way to read our findings in light of [AMS02] is that Townsend's section 8 model has too few sources of random shocks relative to sources of signals to permit higher order beliefs to play a role.

### 33.11.1 Further historical remarks

[Sar91] proposed a way to compute an equilibrium without making Townsend's approximation.

Extending the reasoning of [Mut60], 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 had to guess and verify the appropriate orders of the autoregressive and moving average pieces of the equilibrium representation.

By working in the frequency domain [Kas00] showed how to discover the appropriate orders of the autoregressive and moving average parts, and also how to compute an equilibrium.

The [PS05] 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 [PCL86], [PS05] 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.

---

# Part VIII

# Asset Pricing and Finance

CHAPTER

THIRTYFOUR

# ASSET PRICING II: THE LUCAS ASSET PRICING MODEL

**Contents**


- *Asset Pricing II: The Lucas Asset Pricing Model*
  - *Overview*
  - *The Lucas Model*
  - *Exercises*
  - *Solutions*


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

```
!pip install interpolation
```

## 34.1 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 [Luc78] 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 interpolation import interp
from numba import njit, prange
from scipy.stats import lognorm
import matplotlib.pyplot as plt
%matplotlib inline
```

## 34.2 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.

### 34.2.1 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 $\phi$ 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

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t u(c_t) \tag{34.1}$$

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.

### 34.2.2 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 $\pi_t$ of the claim held in each period.

Thus, the consumer problem is to maximize (34.1) 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 $\pi_t$ 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

$$c + \pi' p(y) \leq \pi y + \pi p(y) \tag{34.2}$$

We can invoke the fact that utility is increasing to claim equality in (34.2) and hence eliminate the constraint, obtaining

$$v(\pi, y) = \max_{\pi'} \left\{ u[\pi(y + p(y)) - \pi' p(y)] + \beta \int v(\pi', G(y, z)) \phi(dz) \right\} \tag{34.3}$$

The solution to this dynamic programming problem is an optimal policy expressing either $\pi'$ or $c$ as a function of the state $(\pi, y)$.

- 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 (34.3) 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 (34.3) with respect to $\pi$, yielding

$$v_1'(\pi, y) = u'(c)(y + p(y))$$

Next, we impose the equilibrium constraints while combining the last two equations to get

$$p(y) = \beta \int \frac{u'[G(y, z)]}{u'(y)} [G(y, z) + p(G(y, z))]\phi(dz) \tag{34.4}$$

In sequential rather than functional notation, we can also write this as

$$p_t = \mathbb{E}_t \left[ \beta \frac{u'(c_{t+1})}{u'(c_t)} (y_{t+1} + p_{t+1}) \right] \tag{34.5}$$

This is the famous consumption-based asset pricing equation.

Before discussing it further we want to solve out for prices.

### 34.2.3 Solving the Model

Equation (34.4) 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

$$f(y) := u'(y)p(y) \tag{34.6}$$

so that (34.4) becomes

$$f(y) = h(y) + \beta \int f[G(y,z)]\phi(dz) \tag{34.7}$$

Here $h(y) := \beta \int u'[G(y,z)]G(y,z)\phi(dz)$ is a function that depends only on the primitives.

Equation (34.7) is a functional equation in $f$.

The plan is to solve out for $f$ and convert back to $p$ via (34.6).

To solve (34.7) 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

$$(Tf)(y) = h(y) + \beta \int f[G(y,z)]\phi(dz) \tag{34.8}$$

In what follows, we refer to $T$ as the Lucas operator.

The reason we do this is that a solution to (34.7) 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

$$\|Tf - Tg\| \leq \alpha \|f - g\|, \qquad \forall\, f, g \in cb\mathbb{R}_+ \tag{34.9}$$

Here $\|h\| := \sup_{x \in \mathbb{R}_+} |h(x)|$.

To see that (34.9) 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 (34.9) with $\alpha := \beta$.

### 34.2.4 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

    @njit(parallel=parallel_flag)
    def T(f):
        """
        The Lucas operator
        """

        # Turn f into a function
        Af = lambda x: interp(grid, f, x)

        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
```

(continues on next page)

(continued from previous page)

```
    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='$p*(y)$')
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.

$\beta = 0.95$
$\beta = 0.98$
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

## 34.3 Exercises

### 34.3.1 Exercise 1

Replicate *the figure* to show how discount factors affect prices.

## 34.4 Solutions

### 34.4.1 Exercise 1

```
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

THIRTYFIVE

# ELEMENTARY ASSET PRICING THEORY

**Contents**


- *Elementary Asset Pricing Theory*
  - *Overview*
  - *Key Equation*
  - *Implications of Key Equation*
  - *Expected Return - Beta Representation*
  - *Mean-Variance Frontier*
  - *Empirical Implementations*
  - *Exercises*
  - *Solutions*


## 35.1 Overview

This lecture summarizes the heart of applied asset-pricing theory.

From a single equation, we'll derive

- a mean-variance frontier
- a single-factor model of excess returns on each member of a collection of assets

To do this, we use two ideas:

- an asset pricing equation
- a Cauchy-Schwartz inequality

For background and basic concepts, 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.

## 35.2 Key Equation

We begin with a **key asset pricing equation**:

$$EmR^i = 1 \tag{35.1}$$

for $i = 1, \dots, I$ 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 returns $R^i$ and the scalar stochastic discount factor $m$ live live in a common probability space.

[HR87] and [HJ91] explain how the existence of a scarlar stochastic discount factor that verifies equation (35.1) is implied by a **law of one price** that requires that all portfolios of assets that end up having the same payouts must have the same price.

They also explain how the **absence of an arbitrage** implies that the stochastic discount factor $m \geq 0$.

## 35.3 Implications of Key Equation

We combine key equation (35.1) 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 [HR87] and [HJ91]. For other important foundations of these ideas, see [Ros76], [Ros78], [HK79], [Kre81], and [CR83].

---

By that remark, Lars Hansen meant 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 key equation (35.1).

Let's do this step by step.

First note that the definition $\text{cov}\left(m, R^i\right) = E(m - Em)(R^i - ER^i)$ of a covariance implies that

$$EmR^i = EmER^i + \text{cov}\left(m, R^i\right)$$

Substituting this result into key equation (35.1) gives

$$1 = EmER^i + \text{cov}\left(m, R^i\right) \tag{35.2}$$

Next note that for a risk-free asset with non-random gross return $R^f$, equation (35.1) 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 (35.2) 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

$$ER^i - R^f = -\text{cov}\left(m, R^i\right) R^f \tag{35.3}$$

Equation (35.3) can be rearranged to display important parts of asset pricing theory.

## 35.4 Expected Return - Beta Representation

We can obtain the celebrated **expected-return-Beta -representation** for gross return $R^i$ simply by rearranging excess return equation (35.3) 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

$$ER^i = R^f + \beta_{i,m} \lambda_m \tag{35.4}$$

Here

- $\beta_{i,m}$ is a (population) least squares regression coefficient of gross return $R^i$ on stochastic discount factor $m$, an object that is often called asset $i$'s **beta**
- $\lambda_m$ is minus the variance of $m$ divided by the mean of $m$, an object that is often called the **price of risk**.

To interpret this representation it helps to provide the following widely used example.

**Example**

A popular model of $m$ is

$$m_{t+1} = \exp(-\rho) \exp(-\gamma(c_{t+1} - c_t))$$

where $\rho > 0$, $\gamma > 0$, and the log of consumption growth is governed by

$$c_{t+1} - c_t = \mu + \sigma_c \epsilon_{t+1}$$

where $\epsilon_{t+1} \sim \mathcal{N}(0, 1)$.

Here

- $\gamma > 0$ is a coefficient of relative risk aversion
- $\rho > 0$ is a fixed intertemporal discount rate

$$m_{t+1} = \exp(-\rho) \exp(-\gamma\mu - \gamma\sigma_c \epsilon_{t+1})$$

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 the representation (35.4), an asset with an $R^i$ that can be expected to be high when consumption growth is low has $\beta_i$ 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, which justifies its low average return

An asset with an $R^i$ that is low when consumption growth is low has $\beta_i$ 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, which justifies its high average return

## 35.5 Mean-Variance Frontier

Now we'll derive the celebrated **mean-variance frontier**.

We do this using a classic method of Lars Peter Hansen and Scott Richard [HR87].

**Note:** Methods of Hansen and Richard are described and used extensively by [Coc05].

Their idea was rearrange the key equation (35.1), namely, $EmR^i = 1$, and then to apply the 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.

Let's apply that idea to deduce

$$1 = E\left(mR^i\right) = E(m)E\left(R^i\right) + \rho_{m,R^i}\frac{\sigma(m)}{E(m)}\sigma\left(R^i\right) \tag{35.5}$$

where $\rho_{m,R^i}$ is the correlation coefficient defined as

$$\rho_{m,R^i} \equiv \frac{\text{cov}\left(m, R^i\right)}{\sigma(m)\sigma\left(R^i\right)}$$

and where $\sigma$ denotes the standard deviation of the variable in parentheses

Equation (35.5) 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

$$\left|ER^i - R^f\right| \leqslant \frac{\sigma(m)}{E(m)}\sigma\left(R^i\right) \tag{35.6}$$

Inequality (35.6) 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

$$\rho_{m,R^i} = \begin{cases} +1 & \implies R^i \text{ is on lower frontier} \\ -1 & \implies R^i \text{ is on an upper frontier} \end{cases}$$

The image below illustrates a mean-variance frontier.

The mathematical structure of the mean-variance frontier described by inequality (35.6) implies that

- all returns on frontier are perfectly correlated.

  Thus,

  - Let $R^m, R^{mv}$ be two returns on frontier.
  - Then for some scalar $a$
  - $R^{mv} = R^f + a\left(R^m - R^f\right)$

  This is an **exact** equation with no **residual**

- each return $R^i$ that is on the mean-variance frontier is perfectly correlated with $m$

  - $\left(\rho_{m,R^i} = -1\right) \Rightarrow \begin{cases} m = a + bR^{mv} \\ R^{mv} = e + dm \end{cases}$ 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:

$$ER^i = R^f + \beta_{i,R^{mv}}\left[E\left(R^{mv}\right) - R^f\right] \tag{35.7}$$

- The special case of a single-beta representation (35.7) with $R^i = R^{mv}$ is

  $ER^{mv} = R^f + 1 \cdot \left[E\left(R^{mv}\right) - R^f\right]$

## 35.6 Empirical Implementations

We briefly describe empirical implementations of multi-factor generalizations of the single-factor model described above.

The single-beta representation (35.7) is a special case with there being just a single factor.

Two representations are often used in empirical work.

One is a **time-series regression** of gross return $R_t^i$ on multiple risk factors $f_t^j, j = a, b, \ldots$ that is designed to uncover exposures of return $R^i$ to each of a set of **risk-factors** $f_t^j, j = a, b, \ldots$,:

$$R_t^i = a_i + \beta_{i,a} f_t^a + \beta_{i,b} f_t^b + \ldots + \epsilon_t^i, \quad t = 1, 2, \ldots, T$$
$$\epsilon_t^i \perp f_t^j, i = 1, 2, \ldots, I; j = a, b, \ldots$$

For example:

- 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.

Model objects are interpreted as follows:

- $\beta_{i,a}$ is the exposure of return $R^i$ to factor $f_a$ risk
- $\lambda_a$ is the price of exposure to factor $f_a$ risk

The other representation entails a **cross-section regression** of **average returns** $ER^i$ for assets $i = 1, 2, \ldots, I$ on **prices of risk** $\lambda_j$ for $j = a, b, c, \ldots$

Here is the regression specification:

$$ER^i = \gamma + \beta_{i,a} \lambda_a + \beta_{i,b} \lambda_b + \cdots$$

**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 $f_t^a, f_t^b, \ldots$

**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**

$$\underbrace{E\left(R^i\right)}_{\text{average return over time series}} = \gamma + \underbrace{\beta_{i,a}}_{\text{regressor}} \underbrace{\lambda_a}_{\text{regressioncoefficient}} + \underbrace{\beta_{i,b}}_{\text{regressor}} \underbrace{\lambda_b}_{\text{regressioncoefficient}} + \cdots + \underbrace{\alpha_i}_{\text{pricing errors}}, i = 1, \ldots, I; \quad \underbrace{\alpha_i \perp \beta_{i,j}, j = a, b, \ldots}_{\text{least squares orthogonality condition}}$$

- estimate $\gamma, \lambda_a, \lambda_b, \ldots$ by an appropriate regression technique, being thoughtful about 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

$$ER^{ei} = \beta_{i,a} \lambda_a + \beta_{i,b} \lambda_b + \cdots + \alpha_i, i = 1, \ldots, I$$

## 35.7 Exercises

Let's start with some imports.

```
import numpy as np
from scipy.stats import stats
import statsmodels.api as sm
from statsmodels.sandbox.regression.gmm import GMM
import matplotlib.pyplot as plt
%matplotlib inline
```

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
```

### 35.7.1 Exercise 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.

### 35.7.2 Exercise 2

Give a formula for the regression coefficient $\beta_{i,R^m}$.

### 35.7.3 Exercise 3

Recall our earlier discussions of a **direct problem** and an **inverse problem**.

- A direct problem is about simulating a particular model.
- An inverse problem is about using data to **estimate** or **choose** a particular model from a manifold of models.

Please assume the parameter values set below and then simulate 2000 observations from the theory specified above for 5

assets, $i = 1, \dots, 5$.

$$
\begin{aligned}
E\left[R^f\right] &= 0.02 \\
\sigma_f &= 0.00 \\
\xi &= 0.06 \\
\lambda &= 0.04 \\
\beta_{1,R^m} &= 0.2 \\
\sigma_1 &= 0.04 \\
\beta_{2,R^m} &= .4 \\
\sigma_2 &= 0.04 \\
\beta_{3,R^m} &= .6 \\
\sigma_3 &= 0.04 \\
\beta_{4,R^m} &= .8 \\
\sigma_4 &= 0.04 \\
\beta_{5,R^m} &= 1.0 \\
\sigma_5 &= 0.04
\end{aligned}
$$

**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:

$$
\begin{aligned}
E[(a + bR_t^m)R_t^m)] &= 1 \\
E[(a + bR_t^m)R_t^f)] &= 1
\end{aligned}
$$

### 35.7.4 Exercise 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

### 35.7.5 Exercise 5

Using the estimates of the parameters that you generated above, compute the implied stochastic discount factor.

## 35.8 Solutions

### 35.8.1 Solution to Exercise 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

$$\begin{aligned} E\left[\sigma_i\epsilon_{i,t}\left(R_t^m-R^f\right)\right] &= E\left[\sigma_i\epsilon_{i,t}\left(\xi+\lambda u_t\right)\right] \\ &= \sigma_i\xi E\left[\epsilon_{i,t}\right]+\sigma_i\lambda E\left[\epsilon_{i,t}u_t\right] \\ &= 0 \end{aligned}$$

### 35.8.2 Solution to Exercise 2

The regression coefficient $\beta_{i,R^m}$ is

$$\beta_{i,R^m}=\frac{Cov\left(R_t^i-R^f,R_t^m-R^f\right)}{Var\left(R_t^m-R^f\right)}$$

### 35.8.3 Solution to Exercise 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
```

(continues on next page)

(continued from previous page)

```
# 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 $\sigma_f$

```
ERf_hat, σf_hat = simple_ols(np.ones(T), Rf)
```

```
ERf_hat, σf_hat
```

```
(0.02000000000000003, 3.123283175179055e-17)
```

Let's compare these with the *true* population parameter values.

```
ERf, σf
```

```
(0.02, 0.0)
```

1. $\xi$ and $\lambda$

```
ξ_hat, λ_hat = simple_ols(np.ones(T), Rm - Rf)
```

```
ξ_hat, λ_hat
```

```
(0.06195715114367803, 0.08004292345366795)
```

```
ξ, λ
```

```
(0.06, 0.08)
```

1. $\beta_{i,R^m}$ and $\sigma_i$

```
βi_hat = np.empty(N)
σi_hat = np.empty(N)
```

(continues on next page)

(continued from previous page)

```
for i in range(N):
    βi_hat[i], σi_hat[i] = simple_ols(Rm - Rf, Ri[i, :] - Rf)
```

```
βi_hat, σi_hat
```

```
(array([0.19738194, 0.38669625, 0.61184757, 0.8067745 , 1.00449198]),
 array([0.03948717, 0.04065713, 0.04082347, 0.03972188, 0.03891566]))
```

```
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?

### 35.8.4 Solution to Exercise 4

$$
\begin{aligned}
a((E(R^f)+\xi)+b((E(R^f)+\xi)^2+\lambda^2+\sigma_f^2)=1 \\
aE(R^f)+b(E(R^f)^2+\xi E(R^f)+\sigma_f^2)=1
\end{aligned}
\tag{35.8}
$$

```
# 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)
```

### 35.8.5 Solution to Exercise 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
```

```
(89.62802368155882, -483.52124407164297, 56.88928261225764)
```

CHAPTER

THIRTYSIX

# TWO MODIFICATIONS OF MEAN-VARIANCE PORTFOLIO THEORY

**Contents**


- *Two Modifications of Mean-Variance Portfolio Theory*
  - *Overview*
  - *Appendix*


## 36.1 Overview

### 36.1.1 Remarks About Estimating Means and Variances

The famous **Black-Litterman** (1992) [BL92] portfolio choice model that we describe in this lecture is motivated by the finding that with high or moderate frequency data, means are more difficult to estimate than variances.

A model of **robust portfolio choice** that we'll describe 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

Let's start with some imports:

```
import numpy as np
import scipy as sp
import scipy.stats as stat
import matplotlib.pyplot as plt
%matplotlib inline
from ipywidgets import interact, FloatSlider
```

### 36.1.2 Adjusting Mean-variance Portfolio Choice Theory for Distrust of Mean Excess Returns

This lecture describes two lines of thought that modify the classic mean-variance portfolio choice model in ways designed to make its recommendations more plausible.

As we mentioned above, 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 contemplated the consequences of adjusting investors' subjective beliefs about mean returns in order to render more sensible decisions.

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.

### 36.1.3 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 $\mu$ and covariance matrix $\Sigma$, 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

$$U(\mu, \Sigma; w) = w'\mu - \frac{\delta}{2} w'\Sigma w \tag{36.1}$$

where $\delta > 0$ is a risk-aversion parameter. The first-order condition for maximizing (36.1) with respect to the vector $w$ is

$$\mu = \delta \Sigma w$$

which implies the following design of a risky portfolio:

$$w = (\delta\Sigma)^{-1}\mu \tag{36.2}$$

### 36.1.4 Estimating the Mean and Variance

The key inputs into the portfolio choice model (36.2) are

- estimates of the parameters $\mu, \Sigma$ of the random excess return vector$(\vec{r} - r_f \mathbf{1})$
- the risk-aversion parameter $\delta$

A standard way of estimating $\mu$ is maximum-likelihood or least squares; that amounts to estimating $\mu$ by a sample mean of excess returns and estimating $\Sigma$ by a sample covariance matrix.

### 36.1.5 The Black-Litterman Starting Point

When estimates of $\mu$ and $\Sigma$ from historical sample means and covariances have been combined with **reasonable** values of the risk-aversion parameter $\delta$ to compute an optimal portfolio from formula (36.2), a typical outcome has been $w$'s with **extreme long and short positions**.

A common reaction to these outcomes is that they are so unreasonable 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 (36.2) 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 $\delta$.
- Leaving $\Sigma$ at its maximum-likelihood value, they push $\mu$ 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 $\Sigma$ and a reasonable value of $\delta$, Black and Litterman reverse engineered a vector $\mu_{BL}$ of mean excess returns that makes the $w$ implied by formula (36.2) equal the **actual** market portfolio $w_m$, so that

$$w_m = (\delta\Sigma)^{-1}\mu_{BL}$$

### 36.1.6 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

$$\mathbf{SR}_m = \frac{r_m - r_f}{\sigma}$$

as the **Sharpe-ratio** on the market portfolio $w_m$.

Let $\delta_m$ 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 (36.2).

Evidently, portfolio rule (36.2) then implies that $r_m - r_f = \delta_m \sigma^2$ or

$$\delta_m = \frac{r_m - r_f}{\sigma^2}$$

or

$$\delta_m = \frac{\mathbf{SR}_m}{\sigma}$$

Following the Black-Litterman philosophy, our first step will be to back a value of $\delta_m$ from

- an estimate of the Sharpe-ratio, and
- our maximum likelihood estimate of $\sigma$ drawn from our estimates or $w_m$ and $\Sigma$

The second key Black-Litterman step is then to use this value of $\delta$ together with the maximum likelihood estimate of $\Sigma$ to deduce a $\mu_{\mathbf{BL}}$ that verifies portfolio rule (36.2) 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 $(\delta_m, \mu_m)$ 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)

fig, ax = plt.subplots(figsize=(8, 5))
ax.set_title(r'Difference between $\hat{\mu}$ (estimate) and $\mu_{BL}$ (market␣
↪implied)')
ax.plot(np.arange(N)+1, μ_est, 'o', c='k', label='$\hat{\mu}$')
ax.plot(np.arange(N)+1, μ_m, 'o', c='r', label='$\mu_{BL}$')
ax.vlines(np.arange(N) + 1, 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()
```

### 36.1.7 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

$$\vec{r} - r_f \mathbf{1} \sim \mathcal{N}(\mu_{BL}, \Sigma) \tag{36.3}$$

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 (36.2) to choose a portfolio.

Suppose that the customer's view is expressed by a hunch that rather than (36.3), 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 $\mu_{\mathbf{BL}}$

$$\tilde{\mu} = (\Sigma^{-1} + (\tau\Sigma)^{-1})^{-1}(\Sigma^{-1}\mu_{BL} + (\tau\Sigma)^{-1}\hat{\mu}) \tag{36.4}$$

Black and Litterman would then advise the customer to hold the portfolio associated with these views implied by rule (36.2):

$$\tilde{w} = (\delta\Sigma)^{-1}\tilde{\mu}$$

This portfolio $\tilde{w}$ will deviate from the portfolio $w_{BL}$ in amounts that depend on the mixing parameter $\tau$.

If $\hat{\mu}$ is the maximum likelihood estimator and $\tau$ 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)

τ_slider = FloatSlider(min=0.05, max=10, step=0.5, value=τ)

@interact(τ=τ_slider)
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()
```

```
interactive(children=(FloatSlider(value=1.0, description='τ', max=10.0, min=0.05,␣
↪step=0.5), Output()), _dom_c…
```

### 36.1.8 Bayes Interpretation of the Black-Litterman Recommendation

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 $\mu$ 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 $\tau$ 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.

### 36.1.9 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

$$\bar{z} \cdot \bar{z} = (z - \mu)'\Sigma^{-1}(z - \mu) = \bar{d} \tag{36.5}$$

where $\bar{d} \in \mathbb{R}_+$ denotes the (transformation) of a particular density value.

The curves defined by equation (36.5) 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) [Lea78]).

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) [Dic75] calls it a *curve decolletage*.

Leamer (1978) [Lea78] 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 $\lambda$, 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 $\mu_{BL}$ and $\hat{\mu}$

$$\vec{r}_e = (\Sigma^{-1} + \lambda(\tau\Sigma)^{-1})^{-1}(\Sigma^{-1}\mu_{BL} + \lambda(\tau\Sigma)^{-1}\hat{\mu}) \tag{36.6}$$

Note that if $\lambda = 1$, (36.6) is equivalent with (36.4) and it identifies one point on the information contract curve.

Furthermore, because $\lambda$ is a function of the minimum likelihood $\bar{d}_2$ on the RHS of the constraint, by varying $\bar{d}_2$ (or $\lambda$ ), 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)
```

(continues on next page)

(continued from previous page)

```
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_slider = FloatSlider(min=.1, max=7, step=.5, value=1)

@interact(λ=λ_slider)
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)
    Z_BL = np.zeros((N_r1, N_r2))
    Z_hat = np.zeros((N_r1, N_r2))

    for i in range(N_r1):
        for j in range(N_r2):
            Z_BL[i, j] = dist_r_BL.pdf(np.hstack([X[i, j], Y[i, j]]))
            Z_hat[i, j] = dist_r_hat.pdf(np.hstack([X[i, j], Y[i, j]]))

    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()
```

```
interactive(children=(FloatSlider(value=1.0, description='λ', max=7.0, min=0.1,␣
↪step=0.5), Output()), _dom_cla…
```

Note that the line that connects the two points $\hat{\mu}$ and $\mu_{BL}$ 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 $\tau I$ instead of $\tau\Sigma$.

This leads to the following figure, on which the curve connecting $\hat{\mu}$ and $\mu_{BL}$ are bending

```
λ_grid = np.linspace(.001, 20000, 1000)
curve = np.asarray([black_litterman(λ, μ_m, μ_est, Σ_est,
                                    τ * np.eye(N)).flatten() for λ in λ_grid])

λ_slider = FloatSlider(min=5, max=1500, step=100, value=200)

@interact(λ=λ_slider)
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)
    Z_BL = np.zeros((N_r1, N_r2))
    Z_hat = np.zeros((N_r1, N_r2))

    for i in range(N_r1):
        for j in range(N_r2):
            Z_BL[i, j] = dist_r_BL.pdf(np.hstack([X[i, j], Y[i, j]]))
            Z_hat[i, j] = dist_r_hat.pdf(np.hstack([X[i, j], Y[i, j]]))

    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()
```

```
interactive(children=(FloatSlider(value=200.0, description='λ', max=1500.0, min=5.
↪0, step=100.0), Output()), _…
```

### 36.1.10 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 $\beta_0$ 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 $\Gamma$ 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, simplify first the equation (36.4) characterizing 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 $\iota_T$ 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, $\hat{\mu}$ that would lead to extreme short-long positions are "shrunk" towards the conservative market view, $\mu_{BL}$, 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.

### 36.1.11 Digression on 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 [HS01], [HS08a] call a $\mathsf{T}$ operator.

Let's define the $\mathsf{T}$ operator as it applies to the problem at hand.

Let $x$ be an $n \times 1$ Gaussian random vector with mean vector $\mu$ 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 $\epsilon$, 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 $\phi$.

**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 $\mathsf{T}$ 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 $\mathsf{T}$ is an indirect utility function for a minimization problem in which an **evil agent** 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 $\theta$ is lowered, more robustness is achieved.

**Note:** The $\mathsf{T}$ operator is sometimes called a *risk-sensitivity* operator.

We shall apply $\mathsf{T}$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 $\epsilon$ 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 $\epsilon$ but not the covariance matrix of $\epsilon$).

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

$$\frac{v'v}{2} = \frac{1}{2\theta^2} w'CC'w$$

### 36.1.12 A Robust Mean-variance Portfolio Model

According to criterion (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

$$\mathsf{T}[w(\vec{r} - r_f \mathbf{1})] = w'\mu - \frac{1}{2\theta} w'\Sigma w$$

that comes from replacing the mean $\mu$ 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 $\mathsf{T}$ 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

$$w'(\mu - \theta^{-1}\Sigma w) - \frac{\delta}{2} w'\Sigma w \tag{36.7}$$

The minimizer of (36.7) 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 $\gamma$ shrinks the portfolio weights toward zero in the same way that an increase in risk aversion does.

## 36.2 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 $\varepsilon > 0$

$$\lim_{N\to\infty} \quad P\left\{\left|\bar{X}_N - \mathbb{E}X\right| > \varepsilon\right\} = 0$$

and

$$\lim_{N\to\infty} \quad P\left\{|S_N - \mathbb{V}X| > \varepsilon\right\} = 0$$

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.

### 36.2.1 A 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.

### 36.2.2 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 $\infty$ 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()
```

### 36.2.3 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 $\mu$.

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

$$\begin{aligned}
\mathbb{V}\left(\bar{X}_N\right) &= \mathbb{V}\left(\frac{1}{N}\sum_{i=1}^{N}X_i\right) \\
&= \frac{1}{N^2}\left(\sum_{i=1}^{N}\mathbb{V}(X_i) + 2\sum_{i=1}^{N-1}\sum_{s=i+1}^{N}\text{cov}(X_i, X_s)\right) \\
&= \frac{1}{N^2}\left(N\gamma(0) + 2\sum_{i=1}^{N-1} i\cdot\gamma\left(h\cdot(N-i)\right)\right) \\
&= \frac{1}{N^2}\left(N\frac{\sigma^2}{2\kappa} + 2\sum_{i=1}^{N-1} i\cdot\exp(-\kappa h(N-i))\frac{\sigma^2}{2\kappa}\right)
\end{aligned}$$

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, the 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 $\exp(-\kappa h)^{N-1}$ 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.

```
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 = stat.norm(0, np.sqrt(s)).rvs((M, N))

    y_path = np.zeros((M, N + 1))
    y_path[:, 0] = stat.norm(mean_uncond, std_uncond).rvs(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 = []
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

THIRTYSEVEN

# IRRELEVANCE OF CAPITAL STRUCTURE WITH COMPLETE MARKETS

**Contents**


- *Irrelevance of Capital Structure with Complete Markets*
  - *Introduction*
  - *Competitive equilibrium*
  - *Code*


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

```
!pip install --upgrade quantecon
!pip install interpolation
!conda install -y -c plotly plotly plotly-orca
```

## 37.1 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 [BCG18].

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 [MM58]
- 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*.

### 37.1.1 Setup

The economy lasts for two periods, $t = 0, 1$.

There are two types of consumers named $i = 1, 2$.

A scalar random variable $\epsilon$ 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 $\epsilon$.

### 37.1.2 Endowments

There is a single consumption good in period $0$ and at each random state $\epsilon$ 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.

### 37.1.3 Technology:

Where $\alpha \in (0, 1)$ and $A > 0$

$$\begin{aligned} c_0^1 + c_0^2 + k &= w_0^1 + w_0^2 \\ c_1^1(\epsilon) + c_1^2(\epsilon) &= w_1^1(\epsilon) + w_1^2(\epsilon) + e^\epsilon A k^\alpha, \quad k \geq 0 \end{aligned}$$

### 37.1.4 Preferences:

A consumer of type $i$ orders period $0$ consumption $c_0^i$ and state $\epsilon$, 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

$$u(c) = \begin{cases} \frac{c^{1-\gamma}}{1-\gamma} & \text{if } \gamma \neq 1 \\ \log c & \text{if } \gamma = 1 \end{cases}$$

### 37.1.5 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.

### 37.1.6 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 $i = 1, 2$.

### 37.1.7 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 $\epsilon$.

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 $\phi_1$ and $\gamma$.

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

$$1 = \beta\alpha A K^{\alpha-1} \int \left(\frac{w_1(\epsilon) + AK^\alpha e^\epsilon}{w_0 - K}\right)^{-\gamma} g(\epsilon) e^\epsilon d\epsilon \tag{37.1}$$

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 $\phi_1$ and the utility curvature parameter $\gamma$.

### Remarks

The relative Pareto weight parameter $\eta$ does not appear in equation (37.1) that determines $K$.

Neither does it influence $C_0$ or $C_1(\epsilon)$, which depend solely on $K$.

The role of $\eta$ 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.

## 37.2 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.

### 37.2.1 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 $\epsilon$

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

$$\begin{aligned} \sum_i w_0^i &= w_0 \\ \sum_i w_1^i(\epsilon) &= w_1(\epsilon) \end{aligned}$$

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 $\epsilon$.

Consumers of types $i = 1, 2$ are endowed with $\theta_0^i$ 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 $\epsilon$ assumes a particular value

Later, we'll allow the firm to issue bonds too, but not now.

## 37.2.2 Objects appearing in a competitive equilibrium

Let

- $a^i(\epsilon)$ be consumer $i$ 's purchases of claims on time $1$ consumption in state $\epsilon$
- $q(\epsilon)$ be a pricing kernel for one-period Arrow securities
- $\theta_0^i \geq 0$ be consumer $i$'s intial share of the firm, $\sum_i \theta_0^i = 1$
- $\theta^i$ 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.

### 37.2.3 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 $\epsilon$ 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.

### 37.2.4 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

$$-\bar{a}^i(\epsilon; \theta^i) = w_1^i(\epsilon) + \theta^i A k^\alpha e^\epsilon \tag{37.2}$$

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 $\epsilon$.

Notice that $-\bar{a}^i(\epsilon; \theta^i)$ defined in (37.2) depends on

- his endowment $w_1^i(\epsilon)$ at time $1$ in state $\epsilon$
- his share $\theta^i$ 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 $\epsilon$

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 $\lambda_0^i$ 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 $\epsilon$, 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

$$q(\epsilon) = \beta\left(\frac{u'(c_1^i(\epsilon))}{u'(c_0^i)}\right)g(\epsilon) \tag{37.3}$$

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

$$w_0^i + \theta_0^i V + \int w_1^i(\epsilon)q(\epsilon)d\epsilon + \theta^i\left[Ak^\alpha \int e^\epsilon q(\epsilon)d\epsilon - \tilde{V}\right] \geq c_0^i + \int c_1^i(\epsilon)q(\epsilon)d\epsilon \tag{37.4}$$

The right side of inequality (37.4) 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 $\theta^i$ shares of equity.

From inequality (37.4), we deduce two findings.

**1. No arbitrage profits condition:**

Unless

$$\tilde{V} = Ak^\alpha \int e^\epsilon q(\epsilon)d\epsilon \tag{37.5}$$

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 $\theta^i$ 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 $\theta^i$ 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:

$$\tilde{V} = \int Ak^{\alpha} e^{\epsilon} q(\epsilon; K) d\epsilon \tag{37.6}$$

Equation (37.6) 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 (37.6) prevails, a consumer of type $i$'s choice $\theta^i$ of equity is indeterminate.

Consumer of type $i$ can offset any choice of $\theta^i$ by setting an appropriate schedule $a^i(\epsilon)$ for purchasing state-contingent securities.

### 37.2.5 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

$$q(\epsilon; K) = \beta \left( \frac{u'\left(w_1(\epsilon) + AK^{\alpha}e^{\epsilon}\right)}{u'(w_0 - K)} \right)^{-\gamma} \tag{37.7}$$

To confirm the guess, we begin by considering its consequences for the firm's choice of $k$.

With Arrow securities prices (37.7), the firm's first-order necessary condition for choosing $k$ becomes

$$-1 + \alpha A k^{\alpha-1} \int e^{\epsilon} q(\epsilon; K) d\epsilon = 0 \tag{37.8}$$

which can be verified to be satisfied if the firm sets

$$k = K$$

because by setting $k = K$ equation (37.8) becomes equivalent with the planner's first-order condition (37.1) 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 $\tilde{V}$:

$$\tilde{V} = \int AK^{\alpha} e^{\epsilon} q(\epsilon; K) d\epsilon \tag{37.9}$$

Let $\tilde{V}$ be the value of equity implied by Arrow securities price function (37.7) and formula (37.9).

At the Arrow securities prices $q(\epsilon)$ given by (37.7) and equity value $\tilde{V}$ given by (37.9), 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 $\eta$. Can you explain why? **Hint:** Think about the portfolio indeterminacy finding above.

### 37.2.6 Modigliani-Miller theorem

We now allow a firm to issue both bonds and equity.

Payouts from equity and bonds, respectively, are

$$
\begin{aligned}
d^e(k,b;\epsilon) &= \max\left\{e^\epsilon Ak^\alpha - b, 0\right\} \\
d^b(k,b;\epsilon) &= \min\left\{\frac{e^\epsilon Ak^\alpha}{b}, 1\right\}
\end{aligned}
$$

Thus, one unit of the bond pays one unit of consumption at time 1 in state $\epsilon$ 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 $\epsilon$ 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 [MM58] 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.

## 37.3 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 $\epsilon$ 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 $\eta$ 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:

- $\chi_1$, $\chi_2$: 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.
- $\theta_0^1$, $\theta_0^2$: Consumers' initial shares of a representative firm. Default values are 0.5.
- $\psi$: CRRA risk parameter. Default value is 3.
- $\alpha$: Returns to scale production function parameter. Default value is 0.6.
- $A$: Productivity of technology. Default value is 2.5.
- $\mu$, $\sigma$: Mean and standard deviation of the log of the shock. Default values are -0.025 and 0.4, respectively.
- $\beta$: 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 pandas as pd
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
%matplotlib inline
```

```
#=========== Class: BCG for complete markets ===========#
class BCG_complete_markets:

    # init method or constructor
    def __init__(self,
                 ?1 = 0,
                 ?2 = 0.9,
                 w10 = 1,
                 w20 = 1,
                 ?10 = 0.5,
                 ?20 = 0.5,
                 ? = 3,
                 ? = 0.6,
                 A = 2.5,
                 ? = -0.025,
                 ? = 0.4,
                 ? = 0.96,
                 nb_points_integ = 10):

        #=========== Setup ===========#
        # Risk parameters
        self.?1 = ?1
        self.?2 = ?2

        # Other parameters
        self.? = ?
        self.? = ?
        self.A = A
        self.? = ?
        self.? = ?
        self.? = ?

        # Utility
```

(continues on next page)

(continued from previous page)

```
        self.u = lambda c: (c**(1-γ)) / (1-γ)

        # Production
        self.f = njit(lambda k: A * (k ** α))
        self.Y = lambda ξ, k: np.exp(ξ) * self.f(k)

        # Initial endowments
        self.w10 = w10
        self.w20 = w20
        self.w0 = w10 + w20

        # Initial holdings
        self.ξ10 = ξ10
        self.ξ20 = ξ20

        # Endowments at t=1
        w11 = njit(lambda ξ: np.exp(-ξ1*ξ - 0.5*(ξ1**2)*(ξ**2) + ξ1*ξ))
        w21 = njit(lambda ξ: np.exp(-ξ2*ξ - 0.5*(ξ2**2)*(ξ**2) + ξ2*ξ))
        self.w11 = w11
        self.w21 = w21

        self.w1 = njit(lambda ξ: w11(ξ) + w21(ξ))

        # Normal PDF
        self.g = lambda x: norm.pdf(x, loc=ξ, scale=σ)

        # Integration
        x, self.weights = np.polynomial.hermite.hermgauss(nb_points_integ)
        self.points_integral = np.sqrt(2) * σ * x + ξ

        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.ξ1, self.ξ2)
            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.⍰1, self.⍰2)).
↪root

        return kk

    #=========== Arrow security price ===========#
    # Function: Compute Arrow security price
    def q(self,⍰,k):
        ⍰ = self.⍰
        ⍰ = self.⍰
        w0 = self.w0
        w1 = self.w1
        fk = self.f(k)
        g = self.g

        return ⍰ * ((w1(⍰) + np.exp(⍰)*fk) / (w0 - k))**(-⍰)


    #=========== Firm value V ===========#
    # Function: compute firm value V
    def V(self, k):
        q = self.q
        fk = self.f(k)
        weights = self.weights
        integ = lambda ⍰: np.exp(⍰) * fk * q(⍰, 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
        ⍰10 = self.⍰10
        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 ⍰: (w1(⍰) + np.exp(⍰)*fk)*q(⍰,k)
        denom = np.sum(weights * c1(self.points_integral)) / np.sqrt(np.pi) + (w0 - k)

        w11q = lambda ⍰: w11(⍰)*q(⍰,k)
        num = w10 + ⍰10 * V(k) + np.sum(weights * w11q(self.points_integral)) / np.
↪sqrt(np.pi)

        ⍰ = num / denom
```

(continues on next page)

(continued from previous page)

```
        # Consumption choices
        c10 = ⍰ * (w0 - k)
        c20 = (1-⍰) * (w0 - k)
        c11 = lambda ⍰: ⍰ * (w1(⍰)+Y(⍰,k))
        c21 = lambda ⍰: (1-⍰) * (w1(⍰)+Y(⍰,k))

        return c10, c20, c11, c21


def k_foc_factory(model):
    ⍰ = model.⍰
    f = model.f
    ⍰ = model.⍰
    ⍰ = model.⍰
    A = model.A
    ⍰ = model.⍰
    w0 = model.w0
    ⍰ = model.⍰
    ⍰ = model.⍰

    weights = model.weights
    points_integral = model.points_integral

    w11 = njit(lambda ⍰, ⍰1, : np.exp(-⍰1*⍰ - 0.5*(⍰1**2)*(⍰**2) + ⍰1*⍰))
    w21 = njit(lambda ⍰, ⍰2: np.exp(-⍰2*⍰ - 0.5*(⍰2**2)*(⍰**2) + ⍰2*⍰))
    w1 = njit(lambda ⍰, ⍰1, ⍰2: w11(⍰, ⍰1) + w21(⍰, ⍰2))

    @njit
    def integrand(⍰, ⍰1, ⍰2, k=1e-4):
        fk = f(k)
        return (w1(⍰, ⍰1, ⍰2) + np.exp(⍰) * fk) ** (-⍰) * np.exp(⍰)

    @njit
    def k_foc(k, ⍰1, ⍰2):
        int_k = np.sum(weights * integrand(points_integral, ⍰1, ⍰2, k=k)) / np.
↪sqrt(np.pi)

        mul = ⍰ * ⍰ * A * k ** (⍰ - 1) / ((w0 - k) ** (-⍰))
        val = mul * int_k - 1

        return val

    return k_foc
```

### 37.3.1 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='Agent 1\'s endowment')
ax[0].plot(epsgrid, mdl1.w21(epsgrid), color='blue', label='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='Agent 1\'s endowment')
ax[1].plot(epsgrid, mdl2.w21(epsgrid), color='blue', label='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 $\chi_i$.

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(⍰1grid)

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 = ⍰1grid[i, j]
            X2 = ⍰2grid[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 2.46 s, sys: 7.69 ms, total: 2.47 s
Wall time: 2.46 s
```

```
%%time
# Second-run
fill_k_grid(kgrid)
```

```
CPU times: user 8.46 ms, sys: 11 µs, total: 8.47 ms
Wall time: 8.2 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=⍰1grid, y=⍰2grid, z=kgrid)])
fig.update_layout(scene = dict(xaxis_title='x - ⍰1',
                               yaxis_title='y - ⍰2',
                               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

THIRTYEIGHT

# EQUILIBRIUM CAPITAL STRUCTURES WITH INCOMPLETE MARKETS

**Contents**


- *Equilibrium Capital Structures with Incomplete Markets*
  - *Introduction*
  - *Asset Markets*
  - *Equilibrium verification*
  - *Pseudo Code*
  - *Code*
  - *Examples*
  - *A picture worth a thousand words*


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

```
!pip install --upgrade quantecon
!pip install interpolation
!conda install -y -c plotly plotly plotly-orca
```

## 38.1 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 [BCG18] 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 [MM58]
- 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, [MM58], 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 we can 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.)

### 38.1.1 Setup

We adopt specifications of preferences and technologies used by Bisin, Clemente, and Gottardi (2018) [BCG18] 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 $\epsilon$ 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 $i = 2$.

### 38.1.2 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 $\epsilon$.

At the start of period $0$, a consumer of type $i$ also owns $\theta_0^i$ shares of a representative firm.

### 38.1.3 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 $\epsilon$

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 $\zeta$ 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.

### 38.1.4 Endowments

Per capital economy-wide endowments in periods 0 and 1 are

$$\begin{aligned} w_0 &= w_0^1 + w_0^2 \\ w_1(\epsilon) &= w_1^1(\epsilon) + w_1^2(\epsilon) \text{ in state } \epsilon \end{aligned}$$

### 38.1.5 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 $f(k) = Ak^\alpha, A > 0, \alpha \in (0,1)$.

### 38.1.6 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.

### 38.1.7 Preferences:

A consumer of type $i$ orders period 0 consumption $c_0^i$ and state $\epsilon$-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

$$u(c) = \begin{cases} \frac{c^{1-\gamma}}{1-\gamma} & \text{if } \gamma \neq 1 \\ \log c & \text{if } \gamma = 1 \end{cases}$$

### 38.1.8 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.

## 38.2 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 $\theta^i$ and $\xi^i$ 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 $e^\epsilon Ak^\alpha < b$ 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 $\epsilon$ are thus

$$\begin{aligned} d^e(k,b;\epsilon) &= \max\left\{e^\epsilon Ak^\alpha - b, 0\right\} \\ d^b(k,b;\epsilon) &= \min\left\{\frac{e^\epsilon Ak^\alpha}{b}, 1\right\} \end{aligned} \tag{38.1}$$

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.

### 38.2.1 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 $\epsilon$ 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 $\theta^i$ 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

$$\begin{aligned} d^e(K,B;\epsilon) &= \max\left\{e^\epsilon AK^\alpha - B, 0\right\} \\ d^b(K,B;\epsilon) &= \min\left\{\frac{e^\epsilon AK^\alpha}{B}, 1\right\} \end{aligned}$$

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:

$$\begin{aligned} \check{q} &\geq \beta \int \frac{u'(c_1^i(\epsilon))}{u'(c_0^i)} d^e(K, B; \epsilon) g(\epsilon)\, d\epsilon \quad (= \text{ if } \theta^i > 0) \\ \check{p} &\geq \beta \int \frac{u'(c_1^i(\epsilon))}{u'(c_0^i)} d^b(K, B; \epsilon) g(\epsilon)\, d\epsilon \quad (= \text{ if } b^i > 0) \end{aligned}$$

These inequalities imply that in a symmetric rational expectations equilibrium consumption allocations and prices satisfy

$$\begin{aligned} \check{q} &= \max_i \beta \int \frac{u'(c_1^i(\epsilon))}{u'(c_0^i)} d^e(K, B; \epsilon) g(\epsilon)\, d\epsilon \\ \check{p} &= \max_i \beta \int \frac{u'(c_1^i(\epsilon))}{u'(c_0^i)} d^b(K, B; \epsilon) g(\epsilon)\, d\epsilon \end{aligned}$$

### 38.2.2 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

$$\begin{aligned} q(k, b) &= \max_i \beta \int \frac{u'(C_1^i(\epsilon))}{u'(C_0^i)} d^e(k, b; \epsilon) g(\epsilon)\, d\epsilon \\ p(k, b) &= \max_i \beta \int \frac{u'(C_1^i(\epsilon))}{u'(C_0^i)} d^b(k, b; \epsilon) g(\epsilon)\, d\epsilon \end{aligned}$$

where the payoff functions are described by equations (38.1).

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.

### 38.2.3 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

$$\begin{aligned}
q(k,b) &= \beta \int \frac{u'(C_1^1(\epsilon))}{u'(C_0^1)} d^e(k,b;\epsilon) g(\epsilon)\, d\epsilon = \beta \int \frac{u'(C_1^2(\epsilon))}{u'(C_0^2)} d^e(k,b;\epsilon) g(\epsilon)\, d\epsilon \\
p(k,b) &= \beta \int \frac{u'(C_1^2(\epsilon))}{u'(C_0^2)} d^b(k,b;\epsilon) g(\epsilon)\, d\epsilon
\end{aligned}$$

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:

$$\begin{aligned}
q(k,b) &= \beta \int_{\epsilon^*}^{\infty} \frac{u'(C_1^i(\epsilon))}{u'(C_0^i)} \left(e^\epsilon A k^\alpha - b\right) g(\epsilon)\, d\epsilon, \quad i = 1, 2 \\
p(k,b) &= \beta \int_{-\infty}^{\epsilon^*} \frac{u'(C_1^2(\epsilon))}{u'(C_0^2)} \left(\frac{e^\epsilon A k^\alpha}{b}\right) g(\epsilon)\, d\epsilon + \beta \int_{\epsilon^*}^{\infty} \frac{u'(C_1^2(\epsilon))}{u'(C_0^2)} g(\epsilon)\, d\epsilon
\end{aligned}$$

#### 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:

$$\frac{\partial q(k,b)}{\partial k} = \beta \alpha A k^{\alpha-1} \int_{\epsilon^*}^{\infty} \frac{u'(C_1^i(\epsilon))}{u'(C_0^i)} e^\epsilon g(\epsilon) d\epsilon, \quad i = 1, 2$$

$$\frac{\partial q(k,b)}{\partial b} = -\beta \int_{\epsilon^*}^{\infty} \frac{u'(C_1^i(\epsilon))}{u'(C_0^i)} g(\epsilon) d\epsilon, \quad i = 1, 2$$

$$\frac{\partial p(k,b)}{\partial k} = \beta\alpha\frac{Ak^{\alpha-1}}{b}\int_{-\infty}^{\epsilon^*}\frac{u'(C_1^2(\epsilon))}{u'(C_0^2)}g(\epsilon)d\epsilon$$
$$\frac{\partial p(k,b)}{\partial b} = -\beta\frac{Ak^{\alpha}}{b^2}\int_{-\infty}^{\epsilon^*}\frac{u'(C_1^2(\epsilon))}{u'(C_0^2)}e^{\epsilon}g(\epsilon)d\epsilon$$

**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:

$$k: \quad -1 + \beta\alpha Ak^{\alpha-1}\int_{-\infty}^{\infty}\frac{u'(C_1^2(\epsilon))}{u'(C_0^2)}e^{\epsilon}g(\epsilon)d\epsilon = 0 \tag{38.2}$$

$$b: \quad \int_{\epsilon^*}^{\infty}\left(\frac{u'(C_1^1(\epsilon))}{u'(C_0^1)}\right)g(\epsilon)\,d\epsilon = \int_{\epsilon^*}^{\infty}\left(\frac{u'(C_1^2(\epsilon))}{u'(C_0^2)}\right)g(\epsilon)\,d\epsilon \tag{38.3}$$

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 $k, b$.

## 38.3 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 $\check{p} = p(K,B)$.

## 38.4 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 $\theta_h^1$ and $\theta_l^1$. 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 $c^i$s 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 $C_0^{1,*} = c_0^1(k^*, b^*),\ C_0^{2,*} = c_0^2(k^*, b^*),\ C_1^{1,*}(\epsilon) = c_1^1(k^*, b^*; \epsilon),\ C_1^{1,*}(\epsilon) = c_1^2(k^*, b^*; \epsilon)$.

## 38.5 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:

- $\chi_1$, $\chi_2$: 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.
- $\theta_0^1$, $\theta_0^2$: initial holding of the firm. Default values are 0.5.
- $\psi$: risk parameter. Default value is 3.
- $\alpha$: Production function parameter. Default value is 0.6.
- $A$: Productivity of the firm. Default value is 2.5.
- $\mu$, $\sigma$: Mean and standard deviation of the shock distribution. Default values are respectively -0.025 and 0.4

- $\beta$: Discount factor. Default value is 0.96.
- bound: Bound for truncated normal distribution. Default value is 3.

```
import pandas as pd
import numpy as np
from scipy.stats import norm
from scipy.stats import truncnorm
from scipy.integrate import quad
from scipy.optimize import bisect
from numba import njit
from interpolation import interp
```

```
class BCG_incomplete_markets:

    # init method or constructor
    def __init__(self,
                 ⍰1 = 0,
                 ⍰2 = 0.9,
                 w10 = 0.9,
                 w20 = 1.1,
                 ⍰10 = 0.5,
                 ⍰20 = 0.5,
                 ⍰1 = 3,
                 ⍰2 = 3,
                 ⍰ = 0.6,
                 A = 2.5,
                 ⍰ = -0.025,
                 ⍰ = 0.4,
                 ⍰ = 0.96,
                 bound = 3,
                 Vl = 0,
                 Vh = 0.5,
                 kbot = 0.01,
                 #ktop = (⍰*A)**(1/(1-⍰)),
                 ktop = 0.25,
                 bbot = 0.1,
                 btop = 0.8):

        #=========== Setup ===========#
        # Risk parameters
        self.⍰1 = ⍰1
        self.⍰2 = ⍰2

        # Other parameters
        self.⍰1 = ⍰1
        self.⍰2 = ⍰2
        self.⍰ = ⍰
        self.A = A
        self.⍰ = ⍰
        self.⍰ = ⍰
        self.⍰ = ⍰
        self.bound = bound

        # Bounds for firm value, capital, and debt
        self.Vl = Vl
        self.Vh = Vh
```

(continues on next page)

(continued from previous page)

```
        self.kbot = kbot
        #self.kbot = (β*A)**(1/(1-γ))
        self.ktop = ktop
        self.bbot = bbot
        self.btop = btop

        # Utility
        self.u = njit(lambda c: (c**(1-γ)) / (1-γ))

        # 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(-σ1*ε - 0.5*(σ1**2)*(ε**2) + μ1*ε))
        self.w21 = njit(lambda ε: np.exp(-σ2*ε - 0.5*(σ2**2)*(ε**2) + μ2*ε))
        self.w1 = njit(lambda ε: self.w11(ε) + self.w21(ε))

        # Truncated normal
        ta, tb = (-bound - μ) / σ, (bound - μ) / σ
        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: interp(ε_range, pdf_range, ε))


    #*************************************************************
    # Function: Solve for equilibrium of the BCG model
    #*************************************************************
    def solve_eq(self, print_crit=True):

        # Load parameters
        σ1 = self.σ1
        σ2 = self.σ2
        γ = self.γ
        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
```

(continues on next page)

(continued from previous page)

```
        # We need to find a fixed point on the value of the firm
        V_crit = 1

        Y = njit(lambda ?, fk: np.exp(?)*fk)
        intqq1 = njit(lambda ?, fk, ?1, ?1, b: (w11(?) + ?1*(Y(?, fk) - b))**(-
↪?1)*(Y(?, fk) - b)*g(?))
        intp1 = njit(lambda ?, fk, ?2, b: (Y(?, fk)/b)*(w21(?) + Y(?, fk))**(-
↪?2)*g(?))
        intp2 = njit(lambda ?, fk, ?2, ?2, b: (w21(?) + ?2*(Y(?, fk)-b) + b)**(-
↪?2)*g(?))
        intqq2 = njit(lambda ?, fk, ?2, ?2, b: (w21(?) + ?2*(Y(?, fk)-b) + b)**(-
↪?2)*(Y(?, fk) - b)*g(?))
        intk1 = njit(lambda ?, fk, ?2: (w21(?) + Y(?, fk))**(-?2)*np.exp(?)*g(?))
        intk2 = njit(lambda ?, fk, ?2, ?2, b: (w21(?) + ?2*(Y(?, fk)-b) + b)**(-
↪?2)*np.exp(?)*g(?))
        intB1 = njit(lambda ?, fk, ?1, ?1, b: (w11(?) + ?1*(Y(?, fk) - b))**(-
↪?1)*g(?))
        intB2 = njit(lambda ?, fk, ?2, ?2, b: (w21(?) + ?2*(Y(?, fk) - b) + b)**(-
↪?2)*g(?))

        while V_crit>1e-4:

            # We begin by adding the guess for the value of the firm to endowment
            V = (Vl+Vh)/2
            ww10 = w10 + ?10*V
            ww20 = w20 + ?20*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**?)
#                     Y = lambda ?: np.exp(?)*fk

                    # Compute integration threshold
                    epstar = np.log(b/fk)
```

(continues on next page)

(continued from previous page)

```
                    #*************************************************************
                    # 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
                    θ1 = 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

                        ## First, compute the constant term that is not influenced by␣
 ↪q
                        ## that is, βE[u'(c^{1}_{1})d^{e}(k,B)]
#                         intqq1 = lambda β: (w11(β) + θ1*(Y(β, fk) - b))**(-θ1)*(Y(β,
 ↪ fk) - b)*g(β)
#                         const_qq1 = β * quad(intqq1,epstar,bound)[0]

                        const_qq1 = β * quad(intqq1,epstar,bound, args=(fk, θ1, θ1,␣
 ↪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)**(-θ1));
                            if (rhs > qq1):
                                qq1l = qq1
                            else:
                                qq1h = qq1
                            diff = abs(qq1l-qq1h)

                        #========
                        # Agent 2
                        #========
                        θ2 = b - θ1
                        θ2 = 1 - θ1
```

(continues on next page)

(continued from previous page)

```
                        # 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, ℰE[u'(c^{2}_{1})d^{b}(k,B)]
#                         intp1 = lambda ℰ: (Y(ℰ, fk)/b)*(w21(ℰ) + Y(ℰ, fk))**(-
 ↪ℰ2)*g(ℰ)
#                         intp2 = lambda ℰ: (w21(ℰ) + ℰ2*(Y(ℰ, fk)-b) + b)**(-ℰ2)*g(ℰ)
#                         const_p = ℰ * (quad(intp1,-bound,epstar)[0] + quad(intp2,
 ↪epstar,bound)[0])
                        const_p = ℰ * (quad(intp1,-bound,epstar, args=(fk, ℰ2, b))[0]\
                                       + quad(intp2,epstar,bound, args=(fk, ℰ2, ℰ2,␣
 ↪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*ℰ2-p*b)**(-ℰ2))
                            if (rhs > p):
                                pl = p
                            else:
                                ph = p
                            diff = abs(pl-ph)

                        # qq2 is the equity price consistent with agent-2 Euler␣
 ↪Equation
#                         intqq2 = lambda ℰ: (w21(ℰ) + ℰ2*(Y(ℰ, fk)-b) + b)**(-
 ↪ℰ2)*(Y(ℰ, fk) - b)*g(ℰ)
                        const_qq2 = ℰ * quad(intqq2,epstar,bound, args=(fk, ℰ2, ℰ2,␣
 ↪b))[0]
                        qq2l = 0
                        qq2h = ww20
                        diff = 1
                        while diff > 1e-7:
                            qq2 = (qq2l+qq2h)/2
                            rhs = const_qq2/((ww20-qq2*ℰ2-p*b)**(-ℰ2));
                            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:
                            ℰ1a = ℰ1
                        else:
```

(continues on next page)

(continued from previous page)

```
                            ?1b = ?1

                        #================
                        # Get consumption
                        #================
                        c10 = ww10 - q*?1
                        c11 = lambda ?: w11(?) + ?1*max(Y(?, fk)-b,0)
                        c20 = ww20 - q*(1-?1) - p*b
                        c21 = lambda ?: w21(?) + (1-?1)*max(Y(?, fk)-b,0) + min(Y(?, fk),
 ↪b)


                        #*************************************************
                        # Compute the first order conditions for the firm
                        #*************************************************

                        #===========
                        # Equity FOC
                        #===========
                        # Only agent 2's IMRS is relevent
#                         intk1 = lambda ?: (w21(?) + Y(?, fk))**(-?2)*np.exp(?)*g(?)
#                         intk2 = lambda ?: (w21(?) + ?2*(Y(?, fk)-b) + b)**(-?2)*np.
 ↪exp(?)*g(?)
#                         kfoc_num = quad(intk1,-bound,epstar)[0] + quad(intk2,epstar,
 ↪bound)[0]
                        kfoc_num = quad(intk1,-bound,epstar, args=(fk, ?2))[0] +␣
 ↪quad(intk2,epstar,bound, args=(fk, ?2, ?2, b))[0]
                        kfoc_denom = (ww20- q*?2 - p*b)**(-?2)
                        kfoc = ?*?*A*(k**(?-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 ?: (w11(?) + ?1*(Y(?, fk) - b))**(-?1)*g(?)
#                     intB2 = lambda ?: (w21(?) + ?2*(Y(?, fk) - b) + b)**(-?2)*g(?)

#                     bfoc1 = quad(intB1,epstar,bound)[0] / (ww10 - q*?1)**(-?1)
#                     bfoc2 = quad(intB2,epstar,bound)[0] / (ww20 - q*?2 - p*b)**(-?2)

                    bfoc1 = quad(intB1,epstar,bound, args=(fk, ?1, ?1, b))[0] / (ww10 -␣
 ↪q*?1)**(-?1)
                    bfoc2 = quad(intB2,epstar,bound, args=(fk, ?2, ?2, b))[0] / (ww20 -␣
 ↪q*?2 - p*b)**(-?2)
                    bfoc = bfoc1 - bfoc2

                    if (bfoc > 0):
```

(continues on next page)

(continued from previous page)

```
                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,
                                                        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
    ⍰1ss = ⍰1


    # 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,
```

(continues on next page)

(continued from previous page)

```
        #                                                   bfoc,
        #                                                   epstar,
        #                                                   Vss,
        #                                                   V_crit]]
        #print(formattedList)

        return kss,bss,Vss,qss,pss,c10ss,c11ss,c20ss,c21ss,𝜆1ss


    #*************************************************************
    # Function: Equity and bond valuations by different agents
    #*************************************************************
    def valuations_by_agent(self,
                            c10, c11, c20, c21,
                            k, b):

        # Load parameters
        𝜆1 = self.𝜆1
        𝜆2 = self.𝜆2
        𝜆 = self.𝜆
        A = self.A
        𝜃 = self.𝜃
        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
        𝜔10 = self.𝜔10
        𝜔20 = self.𝜔20
        w11 = self.w11
        w21 = self.w21
        g = self.g

        # Get functions for IMRS/state price density
        IMRS1 = lambda 𝜀: 𝜆 * (c11(𝜀)/c10)**(-𝜆1)*g(𝜀)
        IMRS2 = lambda 𝜀: 𝜆 * (c21(𝜀)/c20)**(-𝜆2)*g(𝜀)

        # Production
        fk = A*(k**𝜃)
        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]
```

(continues on next page)

(continued from previous page)

```
        # Compute equity valuation with agent 2's IMRS
        intQ2 = lambda ?: IMRS2(?)*(Y(?) - b)
        Q2 = quad(intQ2, epstar, bound)[0]

        # Compute bond valuation with agent 2's IMRS
        intP2 = lambda ?: IMRS2(?)*Y(?)/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
        ?1 = self.?1
        ?2 = self.?2
        ? = self.?
        A = self.A
        ? = self.?
        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
        ?10 = self.?10
        ?20 = self.?20
        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,
```

(continues on next page)

(continued from previous page)

```
                                                        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
```

## 38.6 Examples

Below we show some examples computed with the class `BCG_incomplete markets`.

### 38.6.1 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,?1ss = mdl.solve_eq(print_crit=False)
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.178', '0.503', '0.407', '0.092', '0.000', '-0.000', '-0.568', '0.250', '0.131']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.155', '0.487', '0.381', '0.073', '0.000', '0.000', '-0.518', '0.125', '0.021']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.144', '0.479', '0.368', '0.065', '0.000', '0.000', '-0.492', '0.062', '0.034']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.150', '0.484', '0.374', '0.069', '0.000', '-0.000', '-0.504', '0.094', '0.006']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.153', '0.486', '0.377', '0.071', '0.000', '-0.000', '-0.510', '0.109', '0.008']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.151', '0.484', '0.376', '0.070', '0.000', '0.000', '-0.508', '0.102', '0.001']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.151', '0.483', '0.375', '0.070', '-0.000', '0.000', '-0.507', '0.098', '0.003']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.151', '0.484', '0.375', '0.070', '0.000', '-0.000', '-0.507', '0.100', '0.001']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.151', '0.484', '0.376', '0.070', '0.000', '0.000', '-0.507', '0.101', '0.000']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.151', '0.484', '0.376', '0.070', '0.000', '0.000', '-0.508', '0.101', '0.000']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.151', '0.484', '0.376', '0.070', '-0.000', '-0.000', '-0.507', '0.101', '0.000
↪']
finished
```

```
print(-kss+qss+pss*bss)
print(Vss)
print(?1ss)
```

```
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],
```

(continues on next page)

(continued from previous page)

```
                                    z=[Vss],
                                    mode='markers',
                                    marker=dict(size=3, color='red')),
                        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 $\zeta$ deviates.

So within this equilibrium, there is a *qualified* Modigliani-Miller theorem that asserts that firm $\zeta$'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 $\zeta$'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 $\zeta$ 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 $\xi_1$.

```
def off_eq_check(mdl,kss,bss,e=0.1):
    # Big K and big B
    k = kss
    b = bss + e

    # Load parameters
    ξ1 = mdl.ξ1
    ξ2 = mdl.ξ2
    β = mdl.β
    A = mdl.A
    γ = mdl.γ
    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
    ϕ10 = mdl.ϕ10
    ϕ20 = mdl.ϕ20
    w11 = mdl.w11
    w21 = mdl.w21
    g = mdl.g

    Y = njit(lambda ϵ, fk: np.exp(ϵ)*fk)
    intqq1 = njit(lambda ϵ, fk, ϕ1, ϕ1, ϕ1, b: (w11(ϵ) + ϕ1*(Y(ϵ, fk) - b) + ϕ1)**(-
↪ϕ1)*(Y(ϵ, fk) - b)*g(ϵ))
    intpp1a = njit(lambda ϵ, fk, ϕ1, ϕ1, b: (Y(ϵ, fk)/b)*(w11(ϵ) + Y(ϵ, fk)/b*ϕ1)**(-
↪ϕ1)*g(ϵ))
    intpp1b = njit(lambda ϵ, fk, ϕ1, ϕ1, ϕ1, b: (w11(ϵ) + ϕ1*(Y(ϵ, fk)-b) + ϕ1)**(-
↪ϕ1)*g(ϵ))
    intpp2a = njit(lambda ϵ, fk, ϕ2, ϕ2, b: (Y(ϵ, fk)/b)*(w21(ϵ) + Y(ϵ, fk)/b*ϕ2)**(-
↪ϕ2)*g(ϵ))
    intpp2b = njit(lambda ϵ, fk, ϕ2, ϕ2, ϕ2, b: (w21(ϵ) + ϕ2*(Y(ϵ, fk)-b) + ϕ2)**(-
↪ϕ2)*g(ϵ))
    intqq2 = njit(lambda ϵ, fk, ϕ2, ϕ2, b: (w21(ϵ) + ϕ2*(Y(ϵ, fk)-b) + b)**(-ϕ2)*(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 + ϕ10*V
        ww20 = w20 + ϕ20*V

        # Production
        fk = A*(k**ϕ)
#         Y = lambda ϵ: np.exp(ϵ)*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
        ?1a = 0
        ?1b = b/2
        p = 0.3

        while abs(?1b - ?1a) > 0.001:

            ?1 = (?1a + ?1b) / 2
            ?1a = 0.3
            ?1b = 1

            while abs(?1b - ?1a) > (0.001/b):

                ?1 = (?1a + ?1b) / 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 ?: (w11(?) + ?1*(Y(?, fk) - b) + ?1)**(-?1)*(Y(?,␣
↪fk) - b)*g(?)
#                 const_qq1 = ? * quad(intqq1,epstar,bound)[0]
                const_qq1 = ? * quad(intqq1,epstar,bound, args=(fk, ?1, ?1, ?1, 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*?1-p*?1)**(-?1));
                    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 ?: (Y(?, fk)/b)*(w11(?) + Y(?, fk)/b*?1)**(-
↪?1)*g(?)
#                 intpp1b = lambda ?: (w11(?) + ?1*(Y(?, fk)-b) + ?1)**(-?1)*g(?)
#                 const_pp1 = ? * (quad(intpp1a,-bound,epstar)[0] + quad(intpp1b,
↪epstar,bound)[0])
                const_pp1 = ? * (quad(intpp1a,-bound,epstar, args=(fk, ?1, ?1, b))[0]␣
↪\
                                 + quad(intpp1b,epstar,bound, args=(fk, ?1, ?1, ?1,␣
↪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*β1-pp1*β1)**(-γ1))
                    if (rhs > pp1):
                        pp1l = pp1
                    else:
                        pp1h = pp1
                    diff = abs(pp1l-pp1h)

                #========
                # Agent 2
                #========
                γ2 = b - γ1
                β2 = 1 - β1

                # 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, βE[u'(c^{2}_{1})d^{b}(k,B)]
#                 intpp2a = lambda ε: (Y(ε, fk)/b)*(w21(ε) + Y(ε, fk)/b*γ2)**(-
↪γ2)*g(ε)
#                 intpp2b = lambda ε: (w21(ε) + β2*(Y(ε, fk)-b) + γ2)**(-γ2)*g(ε)
#                 const_pp2 = β * (quad(intpp2a,-bound,epstar)[0] + quad(intpp2b,
↪epstar,bound)[0])
                const_pp2 = β * (quad(intpp2a,-bound,epstar, args=(fk, γ2, β2, b))[0]␣
↪\
                                 + quad(intpp2b,epstar,bound, args=(fk, γ2, β2, γ2,␣
↪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*β2-pp2*β2)**(-γ2))
                    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 ε: (w21(ε) + β2*(Y(ε, fk)-b) + b)**(-γ2)*(Y(ε, fk) -
↪ b)*g(ε)
#                 const_qq2 = β * quad(intqq2,epstar,bound)[0]
```

(continues on next page)

(continued from previous page)

```
                const_qq2 = ℓ * quad(intqq2,epstar,bound, args=(fk, ℓ2, ℓ2, b))[0]
                qq2l = 0
                qq2h = ww20
                diff = 1
                while diff > 1e-7:
                    qq2 = (qq2l+qq2h)/2
                    rhs = const_qq2/((ww20-qq2*ℓ2-p*ℓ2)**(-ℓ2));
                    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:
                    ℓ1a = ℓ1
                else:
                    ℓ1b = ℓ1

                #print(p,q,ℓ1,ℓ1)

            if pp1 > pp2:
                ℓ1a = ℓ1
            else:
                ℓ1b = ℓ1


        #================
        # Get consumption
        #================
        c10 = ww10 - q*ℓ1 - p*ℓ1
        c11 = lambda ℓ: w11(ℓ) + ℓ1*max(Y(ℓ, fk)-b,0) + ℓ1*min(Y(ℓ, fk)/b,1)
        c20 = ww20 - q*(1-ℓ1) - p*(b-ℓ1)
        c21 = lambda ℓ: w21(ℓ) + (1-ℓ1)*max(Y(ℓ, fk)-b,0) + (b-ℓ1)*min(Y(ℓ, 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,ℓ1
```

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,?1e1 = 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,⍰1e2 = off_eq_check(mdl,
                                                                kss,
                                                                bss,
                                                                e=0.1)

# Firm Valuation
kgride2, bgride2, Vgride2, Qgride2, Pgride2 = mdl.eq_valuation(c10e2, c11e2, c20e2,␣
↪c21e2,N=20)
```

(continues on next page)

(continued from previous page)

```
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(1e2))
```

```
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
```

### 38.6.2 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 $\zeta$'s value as a function of $k(\zeta), b(\zeta)$.

### 38.6.3 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,⍰1ss = 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(?1ss)

    # 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)
```

```
0
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.178', '0.502', '0.407', '0.092', '-0.000', '-0.000', '-0.570', '0.250', '0.131
↪']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.155', '0.487', '0.381', '0.073', '-0.001', '0.000', '-0.518', '0.125', '0.022']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.145', '0.480', '0.367', '0.065', '0.000', '-0.000', '-0.490', '0.062', '0.034']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.150', '0.484', '0.374', '0.069', '0.000', '0.000', '-0.504', '0.094', '0.006']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.153', '0.485', '0.378', '0.071', '0.000', '-0.000', '-0.511', '0.109', '0.008']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.151', '0.484', '0.376', '0.070', '-0.000', '-0.000', '-0.508', '0.102', '0.001
↪']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.151', '0.483', '0.375', '0.070', '0.000', '-0.000', '-0.507', '0.098', '0.003']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.151', '0.484', '0.375', '0.070', '-0.000', '0.000', '-0.506', '0.100', '0.001']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.151', '0.484', '0.376', '0.070', '0.000', '0.000', '-0.507', '0.101', '0.000']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.151', '0.484', '0.376', '0.070', '-0.000', '0.000', '-0.507', '0.101', '0.000']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.151', '0.484', '0.376', '0.070', '0.000', '0.000', '-0.507', '0.101', '0.000']
```

(continues on next page)

(continued from previous page)

```
finished
1
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.180', '0.544', '0.404', '0.081', '-0.000', '-0.000', '-0.498', '0.250', '0.130
↪']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.158', '0.531', '0.378', '0.063', '-0.000', '-0.000', '-0.443', '0.125', '0.020
↪']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.148', '0.525', '0.364', '0.055', '0.000', '-0.000', '-0.414', '0.062', '0.036']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.153', '0.528', '0.371', '0.059', '0.000', '0.000', '-0.428', '0.094', '0.008']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.156', '0.530', '0.374', '0.061', '-0.000', '-0.000', '-0.435', '0.109', '0.006
↪']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.154', '0.529', '0.373', '0.060', '0.000', '-0.000', '-0.432', '0.102', '0.001']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.155', '0.529', '0.373', '0.061', '0.000', '0.000', '-0.433', '0.105', '0.002']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.155', '0.529', '0.373', '0.060', '-0.000', '0.000', '-0.433', '0.104', '0.000']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.154', '0.529', '0.373', '0.060', '-0.000', '0.000', '-0.433', '0.103', '0.000']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.154', '0.529', '0.373', '0.060', '0.000', '-0.000', '-0.432', '0.103', '0.000']
finished
2
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.184', '0.590', '0.400', '0.070', '0.000', '0.000', '-0.427', '0.250', '0.128']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.162', '0.581', '0.373', '0.053', '0.000', '-0.000', '-0.366', '0.125', '0.017']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.152', '0.577', '0.359', '0.046', '-0.000', '-0.000', '-0.335', '0.062', '0.039
↪']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.157', '0.579', '0.366', '0.049', '0.000', '0.000', '-0.351', '0.094', '0.011']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.159', '0.580', '0.369', '0.051', '-0.000', '-0.000', '-0.359', '0.109', '0.003
↪']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.158', '0.579', '0.368', '0.050', '-0.000', '-0.000', '-0.355', '0.102', '0.004
↪']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.159', '0.579', '0.369', '0.051', '-0.000', '-0.000', '-0.357', '0.105', '0.000
↪']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.159', '0.580', '0.369', '0.051', '0.000', '0.000', '-0.358', '0.107', '0.001']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.159', '0.580', '0.369', '0.051', '-0.000', '-0.000', '-0.358', '0.106', '0.001
↪']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.159', '0.580', '0.368', '0.051', '-0.000', '-0.000', '-0.357', '0.106', '0.000
↪']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.159', '0.580', '0.368', '0.051', '0.000', '-0.000', '-0.357', '0.106', '0.000']
finished
3
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.187', '0.642', '0.395', '0.059', '0.000', '0.000', '-0.354', '0.250', '0.125']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.166', '0.638', '0.366', '0.044', '-0.000', '-0.000', '-0.289', '0.125', '0.014
↪']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.156', '0.637', '0.351', '0.037', '0.000', '-0.000', '-0.255', '0.062', '0.042']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.161', '0.637', '0.359', '0.040', '-0.000', '-0.000', '-0.273', '0.094', '0.014
↪']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.164', '0.637', '0.363', '0.042', '-0.000', '-0.000', '-0.282', '0.109', '0.000
↪']
```

(continues on next page)

(continued from previous page)

```
finished
4
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.192', '0.702', '0.387', '0.049', '0.000', '-0.000', '-0.281', '0.250', '0.122']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.172', '0.704', '0.357', '0.035', '-0.000', '0.000', '-0.211', '0.125', '0.010']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.162', '0.706', '0.342', '0.029', '-0.000', '-0.000', '-0.173', '0.062', '0.046
↪']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.167', '0.705', '0.350', '0.032', '-0.000', '-0.000', '-0.192', '0.094', '0.018
↪']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.170', '0.705', '0.354', '0.033', '-0.000', '-0.000', '-0.202', '0.109', '0.004
↪']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.171', '0.704', '0.356', '0.034', '0.000', '0.000', '-0.206', '0.117', '0.003']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.170', '0.704', '0.355', '0.034', '-0.000', '-0.000', '-0.204', '0.113', '0.000
↪']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.170', '0.704', '0.355', '0.034', '-0.000', '0.000', '-0.205', '0.115', '0.002']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.170', '0.704', '0.355', '0.034', '-0.000', '0.000', '-0.205', '0.114', '0.001']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.170', '0.704', '0.355', '0.034', '-0.000', '-0.000', '-0.205', '0.114', '0.000
↪']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.170', '0.704', '0.355', '0.034', '0.000', '0.000', '-0.204', '0.114', '0.000']
finished
5
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.198', '0.771', '0.377', '0.039', '-0.000', '-0.000', '-0.205', '0.250', '0.118
↪']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.178', '0.781', '0.346', '0.027', '-0.000', '0.000', '-0.129', '0.125', '0.006']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.169', '0.788', '0.330', '0.022', '0.000', '-0.000', '-0.088', '0.062', '0.050']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.174', '0.784', '0.338', '0.024', '-0.000', '0.000', '-0.109', '0.094', '0.022']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.176', '0.783', '0.342', '0.026', '-0.000', '-0.000', '-0.119', '0.109', '0.008
↪']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.177', '0.781', '0.344', '0.026', '0.000', '-0.000', '-0.125', '0.117', '0.001']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.178', '0.781', '0.345', '0.027', '0.000', '0.000', '-0.127', '0.121', '0.003']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.178', '0.782', '0.345', '0.026', '-0.000', '0.000', '-0.125', '0.119', '0.001']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.177', '0.782', '0.345', '0.026', '-0.000', '-0.000', '-0.125', '0.118', '0.000
↪']
finished
6
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.205', '0.851', '0.366', '0.030', '-0.000', '-0.000', '-0.126', '0.250', '0.114
↪']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.186', '0.872', '0.333', '0.020', '0.000', '0.000', '-0.043', '0.125', '0.001']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.177', '0.885', '0.316', '0.016', '0.000', '0.000', '0.001', '0.062', '0.055']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.181', '0.878', '0.324', '0.018', '-0.000', '0.000', '-0.022', '0.094', '0.027']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.184', '0.875', '0.328', '0.019', '0.000', '-0.000', '-0.033', '0.109', '0.013']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.185', '0.874', '0.330', '0.019', '0.000', '0.000', '-0.038', '0.117', '0.006']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.185', '0.873', '0.332', '0.019', '0.000', '0.000', '-0.041', '0.121', '0.002']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.186', '0.872', '0.332', '0.020', '-0.000', '-0.000', '-0.042', '0.123', '0.001
↪']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.186', '0.872', '0.332', '0.020', '-0.000', '-0.000', '-0.044', '0.124', '0.000
↪']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.186', '0.872', '0.332', '0.020', '-0.000', '0.000', '-0.043', '0.124', '0.000']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.186', '0.872', '0.332', '0.020', '0.000', '0.000', '-0.043', '0.124', '0.000']
finished
7
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.212', '0.945', '0.351', '0.022', '0.000', '0.000', '-0.043', '0.250', '0.108']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.194', '0.980', '0.316', '0.014', '0.000', '0.000', '0.047', '0.125', '0.005']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.203', '0.961', '0.334', '0.018', '0.000', '0.000', '-0.000', '0.188', '0.052']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.199', '0.970', '0.325', '0.016', '-0.000', '0.000', '0.023', '0.156', '0.024']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.196', '0.975', '0.321', '0.015', '0.000', '0.000', '0.034', '0.141', '0.010']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.195', '0.977', '0.319', '0.014', '0.000', '0.000', '0.041', '0.133', '0.003']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.195', '0.979', '0.318', '0.014', '-0.000', '0.000', '0.044', '0.129', '0.001']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.195', '0.978', '0.318', '0.014', '-0.000', '0.000', '0.042', '0.131', '0.001']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.195', '0.979', '0.318', '0.014', '-0.000', '0.000', '0.043', '0.130', '0.000']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.195', '0.978', '0.318', '0.014', '0.000', '-0.000', '0.042', '0.130', '0.000']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.195', '0.978', '0.318', '0.014', '0.000', '-0.000', '0.043', '0.130', '0.000']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.195', '0.978', '0.318', '0.014', '-0.000', '0.000', '0.043', '0.130', '0.000']
finished
8
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.221', '1.056', '0.334', '0.016', '0.000', '0.000', '0.044', '0.250', '0.103']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.204', '1.110', '0.298', '0.009', '0.000', '0.000', '0.143', '0.125', '0.011']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.212', '1.081', '0.316', '0.012', '0.000', '0.000', '0.092', '0.188', '0.046']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.208', '1.095', '0.307', '0.010', '0.000', '0.000', '0.117', '0.156', '0.018']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.206', '1.102', '0.302', '0.010', '-0.000', '0.000', '0.130', '0.141', '0.003']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.205', '1.107', '0.300', '0.009', '-0.000', '-0.000', '0.137', '0.133', '0.004']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.205', '1.105', '0.301', '0.009', '-0.000', '0.000', '0.133', '0.137', '0.000']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.206', '1.104', '0.302', '0.010', '0.000', '0.000', '0.131', '0.139', '0.002']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.205', '1.104', '0.301', '0.009', '0.000', '-0.000', '0.132', '0.138', '0.001']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.205', '1.104', '0.301', '0.009', '0.000', '-0.000', '0.133', '0.137', '0.000']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.205', '1.104', '0.301', '0.009', '-0.000', '0.000', '0.133', '0.137', '0.000']
finished
9
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.231', '1.189', '0.315', '0.010', '0.000', '-0.000', '0.137', '0.250', '0.096']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.214', '1.269', '0.277', '0.005', '0.000', '-0.000', '0.247', '0.125', '0.018']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.222', '1.226', '0.296', '0.008', '-0.000', '0.000', '0.190', '0.188', '0.039']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.218', '1.247', '0.286', '0.006', '0.000', '-0.000', '0.218', '0.156', '0.011']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.216', '1.258', '0.282', '0.006', '-0.000', '-0.000', '0.233', '0.141', '0.003']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.217', '1.252', '0.284', '0.006', '-0.000', '-0.000', '0.225', '0.148', '0.004']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.216', '1.256', '0.283', '0.006', '-0.000', '0.000', '0.229', '0.145', '0.000']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.216', '1.256', '0.282', '0.006', '-0.000', '0.000', '0.231', '0.143', '0.002']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.216', '1.256', '0.282', '0.006', '0.000', '0.000', '0.230', '0.144', '0.001']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.216', '1.255', '0.283', '0.006', '0.000', '-0.000', '0.229', '0.144', '0.000']
```

```
k,b,p,q,kfoc,bfoc,epstar,V,V_crit
['0.216', '1.255', '0.283', '0.006', '-0.000', '0.000', '0.229', '0.144', '0.000']
finished
```

```
# 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
```

(continues on next page)

(continued from previous page)

```
A = mdl.A
γ = mdl.γ
epslist = []
for i in range(len(wlist)):
    bb = blist[i]
    kk = klist[i]
    eps = np.log(bb/(A*kk**γ))
    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()
```

## 38.7 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))

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 IX

# Dynamic Programming Squared

CHAPTER

THIRTYNINE

# STACKELBERG PLANS

**Contents**


- *Stackelberg Plans*
  - *Overview*
  - *Duopoly*
  - *The Stackelberg Problem*
  - *Stackelberg Plan*
  - *Recursive Representation of Stackelberg Plan*
  - *Computing the Stackelberg Plan*
  - *Exhibiting Time Inconsistency of Stackelberg Plan*
  - *Recursive Formulation of the Follower's Problem*
  - *Markov Perfect Equilibrium*
  - *MPE vs. Stackelberg*


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

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

## 39.1 Overview

This notebook 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.

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
%matplotlib inline
```

## 39.2 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 of change

$$\gamma v_{it}^2$$

where $\gamma > 0$, 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.

### 39.2.1 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 $\vec{q}_2$.

### 39.2.2 Abstract Statement of the 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 $a_0, a_1, \gamma$.

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.

### 39.2.3 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 RMT5 chapter 2, appendix A and Macroeconomic Theory, 2nd edition, 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 the second-order difference equation

$$q_{1t} + (1 + \beta + c_1) q_{1t+1} - \beta q_{1t+2} = c_0 - c_2 q_{2t+1} \tag{39.1}$$

Equation (39.1) 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 chapter IX of *Macroeconomic Theory, Second edition (1987)*, difference equation (39.1) 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

$$(1 - \frac{1 + \beta + c_1}{\beta} L + \beta^{-1} L^2) = (1 - \delta_1 L)(1 - \delta_2 L) \tag{39.2}$$

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 (39.2) with $\beta^{-1}$ times this inverse operator gives the follower's decision rule for setting $q_{1t+1}$ in the **feedback-feedforward** form.

$$q_{1t+1} = \delta_1 q_{1t} - c_0 \delta_2^{-1} \beta^{-1} \frac{1}{1 - \delta_2^{-1}} + c_2 \delta_2^{-1} \beta^{-1} \sum_{j=0}^{\infty} \delta_2^j q_{2t+j+1}, \quad t \geq 0 \tag{39.3}$$

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 (39.3) for $t \geq 0$.

We can put a sequence $\{\theta_t\}_{t=0}^{\infty}$ of Lagrange multipliers on the sequence of equations (39.3) and formulate the following Lagrangian for the Stackelberg leader firm 2's problem

$$\begin{aligned}\tilde{L} = & \sum_{t=0}^{\infty} \beta^t \{(a_0 - a_1(q_{1t} + q_{2t}))q_{2t} - \gamma(q_{2t+1} - q_{2t})^2\} \\ & + \sum_{t=0}^{\infty} \beta^t \theta_t \{\delta_1 q_{1t} - c_0 \delta_2^{-1} \beta^{-1} \frac{1}{1-\delta_2^{-1}} + c_2 \delta_2^{-1} \beta^{-1} \sum_{j=0}^{\infty} \delta_2^{-j} q_{2t+j+1} - q_{1t+1}\}\end{aligned} \tag{39.4}$$

subject to initial conditions for $q_{1t}, q_{2t}$ at $t = 0$.

**Comments:** We have formulated the Stackelberg problem in a space of sequences.

The max-min problem associated with Lagrangian (39.4) is unpleasant because the time $t$ component of firm 1's payoff function depends on the entire future of its choices of $\{q_{1t+j}\}_{j=0}^{\infty}$.

This renders a direct attack on the problem cumbersome.

Therefore, below, we will formulate the Stackelberg leader's problem recursively.

We'll put our little duopoly model into a broader class of models with the same conceptual structure.

## 39.3 The 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 physically 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

$$-\sum_{t=0}^{\infty} \beta^t r(y_t, u_t) \tag{39.5}$$

The Stackelberg leader faces the model

$$\begin{bmatrix} I & 0 \\ G_{21} & G_{22} \end{bmatrix} \begin{bmatrix} z_{t+1} \\ x_{t+1} \end{bmatrix} = \begin{bmatrix} \hat{A}_{11} & \hat{A}_{12} \\ \hat{A}_{21} & \hat{A}_{22} \end{bmatrix} \begin{bmatrix} z_t \\ x_t \end{bmatrix} + \hat{B} u_t \tag{39.6}$$

We assume that the matrix $\begin{bmatrix} I & 0 \\ G_{21} & G_{22} \end{bmatrix}$ on the left side of equation (39.6) is invertible, so that we can multiply both sides by its inverse to obtain

$$\begin{bmatrix} z_{t+1} \\ x_{t+1} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} z_t \\ x_t \end{bmatrix} + Bu_t \tag{39.7}$$

or

$$y_{t+1} = Ay_t + Bu_t \tag{39.8}$$

### 39.3.1 Interpretation of the Second Block of Equations

The Stackelberg follower's best response mapping is summarized by the second block of equations of (39.7).

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 (39.5) by choosing sequences $\{u_t, x_t, z_{t+1}\}_{t=0}^{\infty}$ subject to (39.8) 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 (39.8) to manipulate the follower's decisions.

### 39.3.2 More Mechanical Details

For any vector $a_t$, define $\vec{a}_t = [a_t, a_{t+1} \ldots]$.

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 Euler equation is embedded in the system of dynamic equations $y_{t+1} = Ay_t + Bu_t$.

Note that in the definition of $\Omega(y_0)$, $y_0$ is taken as given.

Although it is taken as given in $\Omega(y_0)$, eventually, the $x_0$ component of $y_0$ will be chosen by the Stackelberg leader.

### 39.3.3 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 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 state variables confronting 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 at time $0$, $z_0$.

### 39.3.4 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

$$v(y) = \max_{u, y^*} \{-r(y, u) + \beta v(y^*)\} \tag{39.9}$$

where the maximization is subject to

$$y^* = Ay + Bu$$

and $y^*$ denotes next period's value.

Substituting $v(y) = -y'Py$ into Bellman equation (39.9) 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

$$x_0 = -P_{22}^{-1}P_{21}z_0$$

## 39.4 Stackelberg Plan

Now let's map our duopoly model into the above setup.

We will formulate a state space system

$$y_t = \begin{bmatrix} z_t \\ x_t \end{bmatrix}$$

where in this instance $x_t = v_{1t}$, the time $t$ decision of the follower firm 1.

### 39.4.1 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.

### 39.4.2 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.

## 39.5 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 $\check{z}^t = [\check{z}_t, \check{z}_{t-1}, \dots, \check{z}_0]$ to describe the sequence of the Stackelberg leader's decisions that the Stackelberg follower takes as given.

Thus, we let $\check{y}_t' = [\check{z}_t' \quad \check{x}_t']$ 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 $t \geq 1$

$$x_t = \sum_{j=1}^{t} H_j^t \check{z}_{t-j}$$

where

$$\begin{aligned} H_1^t &= \check{A}_{21} \\ H_2^t &= \check{A}_{22}\check{A}_{21} \\ \vdots \quad & \quad \vdots \\ H_{t-1}^t &= \check{A}_{22}^{t-2}\check{A}_{21} \\ H_t^t &= \check{A}_{22}^{t-1}(\check{A}_{21} + \check{A}_{22}H_0^0) \end{aligned}$$

An optimal decision rule for the Stackelberg'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

$$u_t = -F_z \check{z}_t - F_x \sum_{j=1}^{t} H_j^t z_{t-j} = \sigma_t(\check{z}^t) \tag{39.10}$$

Representation (39.10) confirms that whenever $F_x \neq 0$, the typical situation, the time $t$ component $\sigma_t$ 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 $\check{z}^{t-1}$.

### 39.5.1 Comments and Interpretations

After all, at the end of the day, it will turn out that because we set $\check{z}_0 = z_0$, it will be true 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 (39.10) cast in terms of the history $\check{z}^t$ and **not** a corresponding representation cast in terms of $z^t$.

### 39.5.2 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.

### 39.5.3 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

$$\begin{bmatrix} \tilde{y}_{t+1} \\ q_{1t+1} \end{bmatrix} = \begin{bmatrix} A - BF & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \tilde{y}_t \\ q_{1t} \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} x_t \tag{39.11}$$

This specification assures that from the point of the view of a 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 it's 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 $\vec{q}_2$.

### 39.5.4 Time Consistency of Follower's Plan

Since the follower can solve its problem using dynamic programming its problem is recursive in what for it are the **natural state variables**, namely

$$\begin{bmatrix} 1 \\ q_{2t} \\ \tilde{q}_{10} \\ \tilde{x}_0 \end{bmatrix}$$

It follows that the follower's plan is time consistent.

## 39.6 Computing the Stackelberg Plan

Here is our code to compute a Stackelberg plan via a linear-quadratic dynamic program as outlined above

```
# 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
# 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]
```

(continues on next page)

(continued from previous page)

```
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 Stackelberg leader\n")
print(f"F = {F}")
```

```
Computed policy for Stackelberg leader

F = [[-1.58004454  0.29461313  0.67480938  6.53970594]]
```

### 39.6.1 Implied Time Series for Price and Quantities

The following code plots the price and quantities

```
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()
```

### 39.6.2 Value of Stackelberg Leader

We'll compute the present value earned by the Stackelberg leader.

We'll compute it two ways (they give identical answers – just a check on coding and thinking)

```
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
```

## 39.7 Exhibiting Time Inconsistency of Stackelberg Plan

In the code below we compare two values

- the continuation value $-y_t P y_t$ earned by a continuation Stackelberg leader who inherits state $y_t$ at $t$
- the value of a **reborn Stackelberg leader** who inherits state $z_t$ at $t$ and sets $x_t = -P_{22}^{-1} P_{21}$

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='continuation leader at t', ms=2)
axes[0].set(title=r'Leader control variable $u_{t}$', 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'Follower control variable $x_{t}$', 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'Leader value function $v(y_{t})$', xlabel='t')

plt.tight_layout()
plt.show()
```

## 39.8 Recursive Formulation of the 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

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]))
```

```
6.661338147750939e-16
```

```
# x0 == x0_tilde
yt[:, 0][-1] - (yt_tilde[:, 1] - yt_tilde[:, 0])[-1] < tol0
```

```
True
```

### 39.8.1 Explanation of Alignment

If we inspect the coefficients in the decision rule $-\tilde{F}$, we can spot the reason that 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]
```

```
112.65590740578058
```

```
# Value function with `P_guess`
-(y0_tilde.T @ P_guess @ y0_tilde)[0, 0]
```

```
112.6559074057807
```

```
# 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 \
            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, :]
```

(continues on next page)

(continued from previous page)

```
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
```

## 39.9 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 $3 \times 3$ 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))
```

```
7.771561172376096e-15
```

```
# 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
```

(continues on next page)

(continued from previous page)

```
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
```

## 39.10 MPE vs. Stackelberg

```
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]

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'MPE vs. Stackelberg Value Function')
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]
```

```
-3.970942562087714
```

CHAPTER

FORTY

# RAMSEY PLANS, TIME INCONSISTENCY, SUSTAINABLE PLANS

**Contents**


- *Ramsey Plans, Time Inconsistency, Sustainable Plans*
  - *Overview*
  - *The Model*
  - *Structure*
  - *Intertemporal Influences*
  - *Four Models of Government Policy*
  - *A Ramsey Planner*
  - *A Constrained-to-a-Constant-Growth-Rate Ramsey Government*
  - *Markov Perfect Governments*
  - *Equilibrium Outcomes for Three Models of Government Policy Making*
  - *A Fourth Model of Government Decision Making*
  - *Sustainable or Credible Plan*
  - *Whose Credible Plan is it?*
  - *Comparison of Equilibrium Values*
  - *Note on Dynamic Programming Squared*


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

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

## 40.1 Overview

This lecture describes a linear-quadratic version of a model that Guillermo Calvo [Cal78] used to illustrate the **time inconsistency** of optimal government plans.

Like Chang [Cha98], we use the model as a laboratory in which to explore the consequences of different timing protocols for government decision making.

The model focuses attention on intertemporal tradeoffs between

- welfare benefits that anticipated deflation generates by increasing a representative agent's liquidity as measured by his or her real money balances, and
- costs associated with distorting taxes that must be used to withdraw money from the economy in order to generate anticipated deflation

The model features

- rational expectations
- costly government actions at all dates $t \geq 1$ that increase household utilities at dates before $t$
- two Bellman equations, one that expresses the private sector's expectation of future inflation as a function of current and future government actions, another that describes the value function of a Ramsey planner

A theme of this lecture is that timing protocols affect outcomes.

We'll use ideas from papers by Cagan [Cag56], Calvo [Cal78], Stokey [Sto89], [Sto91], Chari and Kehoe [CK90], Chang [Cha98], and Abreu [Abr88] as well as from chapter 19 of [LS18].

In addition, we'll use ideas from linear-quadratic dynamic programming described in Linear Quadratic Control as applied to Ramsey problems in *Stackelberg problems*.

In particular, we have specified the model in a way that allows us to use linear-quadratic dynamic programming to compute an optimal government plan under a timing protocol in which a government chooses an infinite sequence of money supply growth rates once and for all at time $0$.

We'll start with some imports:

```
import numpy as np
from quantecon import LQ
import matplotlib.pyplot as plt
%matplotlib inline
```

## 40.2 The Model

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 the Cagan [Cag56] demand function:

$$m_t - p_t = -\alpha(p_{t+1} - p_t) \ , \ \alpha > 0 \tag{40.1}$$

for $t \geq 0$.

Equation (40.1) asserts that the demand for real balances is inversely related to the public's expected rate of inflation, which here equals the actual rate of inflation.

(When there is no uncertainty, an assumption of **rational expectations** simplifies to **perfect foresight**).

(See [Sar77] for a rational expectations version of the model when there is uncertainty)

Subtracting the demand function at time $t$ from the demand function at $t+1$ gives:

$$\mu_t - \theta_t = -\alpha\theta_{t+1} + \alpha\theta_t$$

or

$$\theta_t = \frac{\alpha}{1+\alpha}\theta_{t+1} + \frac{1}{1+\alpha}\mu_t \tag{40.2}$$

Because $\alpha > 0$, $0 < \frac{\alpha}{1+\alpha} < 1$.

**Definition:** For a scalar $x_t$, let $L^2$ be the space of sequences $\{x_t\}_{t=0}^{\infty}$ satisfying

$$\sum_{t=0}^{\infty} x_t^2 < +\infty$$

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 (40.2) can be solved forward to get:

$$\theta_t = \frac{1}{1+\alpha}\sum_{j=0}^{\infty}\left(\frac{\alpha}{1+\alpha}\right)^j \mu_{t+j} \tag{40.3}$$

**Insight:** In the spirit of Chang [Cha98], note that equations (40.1) and (40.3) show that $\theta_t$ intermediates how choices of $\mu_{t+j},\ j = 0, 1, \ldots$ impinge on time $t$ real balances $m_t - p_t = -\alpha\theta_t$.

We shall use this insight to help us simplify and analyze government policy problems.

That future rates of money creation influence earlier rates of inflation creates optimal government policy problems in which timing protocols matter.

We can rewrite the model as:

$$\begin{bmatrix} 1 \\ \theta_{t+1} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & \frac{1+\alpha}{\alpha} \end{bmatrix} \begin{bmatrix} 1 \\ \theta_t \end{bmatrix} + \begin{bmatrix} 0 \\ -\frac{1}{\alpha} \end{bmatrix} \mu_t$$

or

$$x_{t+1} = Ax_t + B\mu_t \tag{40.4}$$

We write the model in the state-space form (40.4) even though $\theta_0$ is to be determined and so is not an initial condition as it ordinarily would be in the state-space model described in Linear Quadratic Control.

We write the model in the form (40.4) because we want to apply an approach described in *Stackelberg problems*.

Assume that a representative household's utility of real balances at time $t$ is:

$$U(m_t - p_t) = a_0 + a_1(m_t - p_t) - \frac{a_2}{2}(m_t - p_t)^2, \quad a_0 > 0, a_1 > 0, a_2 > 0 \tag{40.5}$$

The "bliss level" of real balances is then $\frac{a_1}{a_2}$.

The money demand function (40.1) and the utility function (40.5) imply that utility maximizing or bliss level of real balances is attained when:

$$\theta_t = \theta^* = -\frac{a_1}{a_2 \alpha}$$

Below, we introduce the discount factor $\beta \in (0,1)$ that a representative household and a benevolent government both use to discount future utilities.

(If we set parameters so that $\theta^* = \log(\beta)$, then we can regard a recommendation to set $\theta_t = \theta^*$ as a "poor man's Friedman rule" that attains Milton Friedman's **optimal quantity of money**)

Via equation (40.3), a government plan $\vec{\mu} = \{\mu_t\}_{t=0}^{\infty}$ leads to an equilibrium sequence of inflation outcomes $\vec{\theta} = \{\theta_t\}_{t=0}^{\infty}$.

We assume that social costs $\frac{c}{2}\mu_t^2$ are incurred at $t$ when the government changes the stock of nominal money balances at rate $\mu_t$.

Therefore, the one-period welfare function of a benevolent government is:

$$-s(\theta_t, \mu_t) \equiv -r(x_t, \mu_t) = \begin{bmatrix} 1 \\ \theta_t \end{bmatrix}' \begin{bmatrix} a_0 & -\frac{a_1 \alpha}{2} \\ -\frac{a_1 \alpha}{2} & -\frac{a_2 \alpha^2}{2} \end{bmatrix} \begin{bmatrix} 1 \\ \theta_t \end{bmatrix} - \frac{c}{2}\mu_t^2 = -x_t' R x_t - Q\mu_t^2 \tag{40.6}$$

Household welfare is summarized by:

$$v_0 = -\sum_{t=0}^{\infty} \beta^t r(x_t, \mu_t) = -\sum_{t=0}^{\infty} \beta^t s(\theta_t, \mu_t) \tag{40.7}$$

We can represent the dependence of $v_0$ on $(\vec{\theta}, \vec{\mu})$ recursively via the linear difference equation

$$v_t = -s(\theta_t, \mu_t) + \beta v_{t+1} \tag{40.8}$$

## 40.3 Structure

The following structure is induced by private agents' behavior as summarized by the demand function for money (40.1) that leads to equation (40.3) that tells how future settings of $\mu$ affect the current value of $\theta$.

Equation (40.3) 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 the recursion

$$v_t = -s(\theta_t, \mu_t) + \beta v_{t+1}$$

where we have called $s(\theta_t, \mu_t) = r(x_t, \mu_t)$ as above.

Thus, we have a triple of sequences $\vec{\mu}, \vec{\theta}, \vec{v}$ associated with a $\vec{\mu} \in L^2$.

At this point $\vec{\mu} \in L^2$ is an arbitrary exogenous policy.

To make $\vec{\mu}$ endogenous, we require a theory of government decisions.

## 40.4 Intertemporal Influences

Criterion function (40.7) and the constraint system (40.4) exhibit the following structure:

- Setting $\mu_t \neq 0$ imposes costs $\frac{c}{2}\mu_t^2$ at time $t$ and at no other times; but
- The money growth rate $\mu_t$ affects the representative household's one-period utilities at all dates $s = 0, 1, \ldots, t$.

That settings of $\mu$ at one date affect household utilities at earlier dates sets the stage for the emergence of a time-inconsistent optimal government plan under a Ramsey (also called a Stackelberg) timing protocol.

We'll study outcomes under a Ramsey timing protocol below.

But we'll also study the consequences of other timing protocols.

## 40.5 Four Models of Government Policy

We consider four models of policymakers that differ in

- what a policymaker is allowed to choose, either a sequence $\vec{\mu}$ or just a single period $\mu_t$.
- when a policymaker chooses, either at time $0$ or at times $t \geq 0$.
- what a policymaker assumes about how its choice of $\mu_t$ affects private agents' 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, taking into account how $\mu_t$ 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 two other models, there is a sequence of policymakers, each of whom sets $\mu_t$ at one $t$ only

- Each such policymaker ignores effects that its choice of $\mu_t$ has on household one-period utilities at dates $s = 0, 1, \ldots, t-1$.

The four models differ with respect to timing protocols, constraints on government choices, and government policymakers' beliefs about how their decisions affect private agents' beliefs about future government decisions.

The models are

- A single Ramsey planner chooses a sequence $\{\mu_t\}_{t=0}^{\infty}$ once and for all at time $0$.
- A single Ramsey planner 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 $t \geq 0$.
- A sequence of separate policymakers chooses $\mu_t$ for $t = 0, 1, 2, \ldots$
  - a time $t$ policymaker chooses $\mu_t$ only and forecasts that future government decisions are unaffected by its choice.
- A sequence of separate policymakers chooses $\mu_t$ for $t = 0, 1, 2, \ldots$
  - a time $t$ policymaker chooses only $\mu_t$ but believes that its choice of $\mu_t$ shapes private agents' beliefs about future rates of money creation and inflation, and through them, future government actions.

## 40.6 A Ramsey Planner

First, we consider a Ramsey planner that chooses $\{\mu_t, \theta_t\}_{t=0}^{\infty}$ to maximize (40.7) subject to the law of motion (40.4).

We can split this problem into two stages, as in *Stackelberg problems* and [LS18] Chapter 19.

In the first stage, we take the initial inflation rate $\theta_0$ as given, and then solve the resulting LQ dynamic programming problem.

In the second stage, we maximize over the initial inflation rate $\theta_0$.

Define a feasible set of $(\vec{x}_1, \vec{\mu}_0)$ sequences:

$$\Omega(x_0) = \{(\vec{x}_1, \vec{\mu}_0) : x_{t+1} = Ax_t + B\mu_t \ , \ \forall t \geq 0\}$$

### 40.6.1 Subproblem 1

The value function

$$J(x_0) = \max_{(\vec{x}_1, \vec{\mu}_0) \in \Omega(x_0)} -\sum_{t=0}^{\infty} \beta^t r(x_t, \mu_t)$$

satisfies the Bellman equation

$$J(x) = \max_{\mu, x'} \{-r(x, \mu) + \beta J(x')\}$$

subject to:

$$x' = Ax + B\mu$$

As in *Stackelberg problems*, we map this problem into a linear-quadratic control problem and then carefully use the optimal value function associated with it.

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 the optimal decision rule

$$\mu_t = -Fx_t$$

where

$$F = \beta(Q + \beta B'PB)^{-1}B'PA$$

The QuantEcon LQ class solves for $F$ and $P$ given inputs $Q, R, A, B$, and $\beta$.

### 40.6.2 Subproblem 2

The value of the Ramsey problem is

$$V = \max_{x_0} J(x_0)$$

The value function

$$J(x_0) = -\begin{bmatrix} 1 & \theta_0 \end{bmatrix} \begin{bmatrix} P_{11} & P_{12} \\ P_{21} & P_{22} \end{bmatrix} \begin{bmatrix} 1 \\ \theta_0 \end{bmatrix} = -P_{11} - 2P_{21}\theta_0 - P_{22}\theta_0^2$$

Maximizing this with respect to $\theta_0$ yields the FOC:

$$-2P_{21} - 2P_{22}\theta_0 = 0$$

which implies

$$\theta_0^* = -\frac{P_{21}}{P_{22}}$$

### 40.6.3 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 [Cha98]:

$$\begin{aligned} \theta_0 &= \theta_0^* \\ \mu_t &= b_0 + b_1\theta_t \\ \theta_{t+1} &= d_0 + d_1\theta_t \end{aligned} \tag{40.9}$$

To interpret this system, think of the sequence $\{\theta_t\}_{t=0}^{\infty}$ as a sequence of synthetic **promised inflation rates** that are just computational devices for generating a sequence $\vec{\mu}$ of money growth rates that are to be substituted into equation (40.3) to form 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 (40.3), we obtain the same sequence $\vec{\theta}$ generated by the system (40.9).

(Here an application of the Big $K$, little $k$ trick could once again be enlightening)

Thus, our construction of a Ramsey plan guarantees that **promised inflation** equals **actual inflation**.

### 40.6.4 Multiple roles of $\theta_t$

The inflation rate $\theta_t$ that appears in the system (40.9) and equation (40.3) plays three roles simultaneously:

- In equation (40.3), $\theta_t$ is the actual rate of inflation between $t$ and $t+1$.
- In equation (40.2) and (40.3), $\theta_t$ is also the public's expected rate of inflation between $t$ and $t+1$.
- In system (40.9), $\theta_t$ is a promised rate of inflation chosen by the Ramsey planner at time $0$.

### 40.6.5 Time Inconsistency

As discussed in *Stackelberg problems* 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.

The time inconsistency of a Ramsey plan has motivated other models of government decision making that alter either

- the timing protocol and/or
- assumptions about how government decision makers think their decisions affect private agents' beliefs about future government decisions

## 40.7 A Constrained-to-a-Constant-Growth-Rate Ramsey Government

We now consider the following peculiar model of optimal government behavior.

We have created this model in order to highlight an aspect of an optimal government policy associated with its time inconsistency, namely, the feature that optimal settings of the policy instrument vary over time.

Instead of allowing the Ramsey government to choose different settings of its instrument at different moments, we now assume that at time $0$, a Ramsey government at time $0$ once and for all chooses a **constant** sequence $\mu_t = \check{\mu}$ for all $t \geq 0$ to maximize

$$U(-\alpha\check{\mu}) - \frac{c}{2}\check{\mu}^2$$

Here we have imposed the perfect foresight outcome implied by equation (40.2) that $\theta_t = \check{\mu}$ when the government chooses a constant $\mu$ for all $t \geq 0$.

With the quadratic form (40.5) for the utility function $U$, the maximizing $\bar{\mu}$ is

$$\check{\mu} = -\frac{\alpha a_1}{\alpha^2 a_2 + c}$$

**Summary:** We have introduced the constrained-to-a-constant $\mu$ government in order to highlight time-variation of $\mu_t$ as a telltale sign of time inconsistency of a Ramsey plan.

## 40.8 Markov Perfect Governments

We now change the timing protocol by considering a sequence of government policymakers, the time $t$ representative of which chooses $\mu_t$ and expects all future governments to set $\mu_{t+j} = \bar{\mu}$.

This assumption mirrors an assumption made in a different setting Markov Perfect Equilibrium.

Further, a government policymaker at $t$ believes that $\bar{\mu}$ is unaffected by its choice of $\mu_t$.

The time $t$ rate of inflation is then:

$$\theta_t = \frac{\alpha}{1+\alpha}\bar{\mu} + \frac{1}{1+\alpha}\mu_t$$

The time $t$ government policymaker then chooses $\mu_t$ to maximize:

$$W = U(-\alpha\theta_t) - \frac{c}{2}\mu_t^2 + \beta V(\bar{\mu})$$

where $V(\bar{\mu})$ is the time $0$ value $v_0$ of recursion (40.8) under a money supply growth rate that is forever constant at $\bar{\mu}$.

Substituting for $U$ and $\theta_t$ gives:

$$W = a_0 + a_1(-\frac{\alpha^2}{1+\alpha}\bar{\mu} - \frac{\alpha}{1+\alpha}\mu_t) - \frac{a_2}{2}((-\frac{\alpha^2}{1+\alpha}\bar{\mu} - \frac{\alpha}{1+\alpha}\mu_t)^2 - \frac{c}{2}\mu_t^2 + \beta V(\bar{\mu})$$

The first-order necessary condition for $\mu_t$ is then:

$$-\frac{\alpha}{1+\alpha}a_1 - a_2(-\frac{\alpha^2}{1+\alpha}\bar{\mu} - \frac{\alpha}{1+\alpha}\mu_t)(-\frac{\alpha}{1+\alpha}) - c\mu_t = 0$$

Rearranging we get:

$$\mu_t = \frac{-a_1}{\frac{1+\alpha}{\alpha}c + \frac{\alpha}{1+\alpha}a_2} - \frac{\alpha^2 a_2}{\left[\frac{1+\alpha}{\alpha}c + \frac{\alpha}{1+\alpha}a_2\right](1+\alpha)}\bar{\mu}$$

A **Markov Perfect Equilibrium** (MPE) outcome sets $\mu_t = \bar{\mu}$:

$$\mu_t = \bar{\mu} = \frac{-a_1}{\frac{1+\alpha}{\alpha}c + \frac{\alpha}{1+\alpha}a_2 + \frac{\alpha^2}{1+\alpha}a_2}$$

In light of results presented in the previous section, this can be simplified to:

$$\bar{\mu} = -\frac{\alpha a_1}{\alpha^2 a_2 + (1+\alpha)c}$$

## 40.9 Equilibrium Outcomes for Three Models of Government Policy Making

Below we compute sequences $\{\theta_t, \mu_t\}$ under a Ramsey plan and compare these with the constant levels of $\theta$ and $\mu$ in a) a Markov Perfect Equilibrium, and b) a Ramsey plan in which the planner is restricted to choose $\mu_t = \check{\mu}$ for all $t \geq 0$.

We denote the Ramsey sequence as $\theta^R, \mu^R$ and the MPE values as $\theta^{MPE}, \mu^{MPE}$.

The bliss level of inflation is denoted by $\theta^*$.

First, 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, α, α0, α1, α2, c, T=1000, θ_n=200):

        # Record parameters
        self.α, self.α0, self.α1 = α, α0, α1
        self.α2, self.c, self.T, self.θ_n = α2, c, T, θ_n

        # Create β using "Poor Man's Friedman Rule"
        self.β = np.exp(-α1 / (α * α2))

        # Solve the Ramsey Problem #

        # LQ Matrices
        R = -np.array([[α0,             -α1 * α / 2],
                       [-α1 * α/2, -α2 * α**2 / 2]])
        Q = -np.array([[-c / 2]])
        A = np.array([[1, 0], [0, (1 + α) / α]])
        B = np.array([[0], [-1 / α]])

        # Solve LQ Problem (Subproblem 1)
        lq = LQ(Q, R, A, B, beta=self.β)
        self.P, self.F, self.d = lq.stationary_values()

        # Solve Subproblem 2
        self.θ_R = -self.P[0, 1] / self.P[1, 1]

        # Find bliss level of θ
        self.θ_B = np.log(self.β)

        # Solve the Markov Perfect Equilibrium
        self.μ_MPE = -α1 / ((1 + α) / α * c + α / (1 + α)
```

(continues on next page)

(continued from previous page)

```
                * α2 + α**2 / (1 + α) * α2)
        self.θ_MPE = self.μ_MPE
        self.μ_check = -α * α1 / (α2 * α**2 + c)

        # Calculate value under MPE and Check economy
        self.J_MPE  = (α0 + α1 * (-α * self.μ_MPE) - α2 / 2
                      * (-α * self.μ_MPE)**2 - c/2 * self.μ_MPE**2) / (1 - self.β)
        self.J_check = (α0 + α1 * (-α * self.μ_check) - α2/2
                        * (-α * self.μ_check)**2 - c / 2 * self.μ_check**2) \
                        / (1 - self.β)

        # Simulate Ramsey plan for large number of periods
        θ_series = np.vstack((np.ones((1, T)), np.zeros((1, T))))
        μ_series = np.zeros(T)
        J_series = np.zeros(T)
        θ_series[1, 0] = self.θ_R
        μ_series[0] = -self.F.dot(θ_series[:, 0])
        J_series[0] = -θ_series[:, 0] @ self.P @ θ_series[:, 0].T
        for i in range(1, T):
            θ_series[:, i] = (A - B @ self.F) @ θ_series[:, i-1]
            μ_series[i] = -self.F @ θ_series[:, i]
            J_series[i] = -θ_series[:, i] @ self.P @ θ_series[:, i].T

        self.J_series = J_series
        self.μ_series = μ_series
        self.θ_series = θ_series

        # Find the range of θ in Ramsey plan
        θ_LB = min(θ_series[1, :])
        θ_LB = min(θ_LB, self.θ_B)
        θ_UB = max(θ_series[1, :])
        θ_UB = max(θ_UB, 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

        # Find value function and policy functions over range of θ
        θ_space = np.linspace(self.θ_LB, self.θ_UB, 200)
        J_space = np.zeros(200)
        check_space = np.zeros(200)
        μ_space = np.zeros(200)
        θ_prime = np.zeros(200)
        for i in range(200):
            J_space[i] = - np.array((1, θ_space[i])) \
                         @ self.P @ np.array((1, θ_space[i])).T
            μ_space[i] = - self.F @ np.array((1, θ_space[i]))
            x_prime = (A - B @ self.F) @ np.array((1, θ_space[i]))
            θ_prime[i] = x_prime[1]
            check_space[i] = (α0 + α1 * (-α * θ_space[i]) -
            α2/2 * (-α * θ_space[i])**2 - c/2 * θ_space[i]**2) / (1 - self.β)

        J_LB = min(J_space)
        J_UB = max(J_space)
        J_range = J_UB - J_LB
        self.J_LB = J_LB - 0.05 * J_range
```

(continues on next page)

(continued from previous page)

```
        self.J_UB = J_UB + 0.05 * J_range
        self.J_range = J_range
        self.J_space = J_space
        self.θ_space = θ_space
        self.μ_space = μ_space
        self.θ_prime = θ_prime
        self.check_space = check_space
```

We will create an instance of ChangLQ with the following parameters

```
clq = ChangLQ(α=1, α0=1, α1=0.5, α2=3, c=2)
clq.β
```

```
0.8464817248906141
```

The following code generates a figure that plots the value function from the Ramsey Planner's problem, which is maximized at $\theta_0^R$.

The figure also shows the limiting value $\theta_\infty^R$ to which the inflation rate $\theta_t$ converges under the Ramsey plan and compares it to the MPE value and the bliss value.

```
def plot_value_function(clq):
    """
    Method to plot the value function over the relevant range of θ

    Here clq is an instance of ChangLQ

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

    ax.set_xlim([clq.θ_LB, clq.θ_UB])
    ax.set_ylim([clq.J_LB, clq.J_UB])

    # Plot value function
    ax.plot(clq.θ_space, clq.J_space, lw=2)
    plt.xlabel(r"$\theta$", fontsize=18)
    plt.ylabel(r"$J(\theta)$", fontsize=18)

    t1 = clq.θ_space[np.argmax(clq.J_space)]
    tR = clq.θ_series[1, -1]
    θ_points = [t1, tR, clq.θ_B, clq.θ_MPE]
    labels = [r"$\theta_0^R$", r"$\theta_\infty^R$",
              r"$\theta^*$", r"$\theta^{MPE}$"]

    # Add points for θs
    for θ, label in zip(θ_points, labels):
        ax.scatter(θ, clq.J_LB + 0.02 * clq.J_range, 60, 'black', 'v')
        ax.annotate(label,
                    xy=(θ, clq.J_LB + 0.01 * clq.J_range),
                    xytext=(θ - 0.01 * clq.θ_range,
                    clq.J_LB + 0.08 * clq.J_range),
                    fontsize=18)
    plt.tight_layout()
    plt.show()
```

(continues on next page)

(continued from previous page)

```
plot_value_function(clq)
```

The next code generates a figure that plots the value function from the Ramsey Planner's problem as well as that for a Ramsey planner that must choose a constant $\mu$ (that in turn equals an implied constant $\theta$).

```
def compare_ramsey_check(clq):
    """
    Method to compare values of Ramsey and Check

    Here clq is an instance of ChangLQ
    """
    fig, ax = plt.subplots()
    check_min = min(clq.check_space)
    check_max = max(clq.check_space)
    check_range = check_max - check_min
    check_LB = check_min - 0.05 * check_range
    check_UB = check_max + 0.05 * check_range
    ax.set_xlim([clq.θ_LB, clq.θ_UB])
    ax.set_ylim([check_LB, check_UB])
    ax.plot(clq.θ_space, clq.J_space, lw=2, label=r"$J(\theta)$")

    plt.xlabel(r"$\theta$", fontsize=18)
    ax.plot(clq.θ_space, clq.check_space,
            lw=2, label=r"$V^\check(\theta)$")
    plt.legend(fontsize=14, loc='upper left')

    θ_points = [clq.θ_space[np.argmax(clq.J_space)],
                clq.μ_check]
    labels = [r"$\theta_0^R$", r"$\theta^\check$"]
```

(continues on next page)

(continued from previous page)

```
    for θ, label in zip(θ_points, labels):
        ax.scatter(θ, check_LB + 0.02 * check_range, 60, 'k', 'v')
        ax.annotate(label,
                    xy=(θ, check_LB + 0.01 * check_range),
                    xytext=(θ - 0.02 * check_range,
                            check_LB + 0.08 * check_range),
                    fontsize=18)
    plt.tight_layout()
    plt.show()

compare_ramsey_check(clq)
```

The next code generates figures that plot the policy functions for a continuation Ramsey planner.

The left figure shows the choice of $\theta'$ chosen by a continuation Ramsey planner who inherits $\theta$.

The right figure plots a continuation Ramsey planner's choice of $\mu$ as a function of an inherited $\theta$.

```
def plot_policy_functions(clq):
    """
    Method to plot the policy functions over the relevant range of θ

    Here clq is an instance of ChangLQ
    """
    fig, axes = plt.subplots(1, 2, figsize=(12, 4))

    labels = [r"$\theta_0^R$", r"$\theta_\infty^R$"]

    ax = axes[0]
    ax.set_ylim([clq.θ_LB, clq.θ_UB])
```

(continues on next page)

(continued from previous page)

```
    ax.plot(clq.θ_space, clq.θ_prime,
            label=r"$\theta'(\theta)$", lw=2)
    x = np.linspace(clq.θ_LB, clq.θ_UB, 5)
    ax.plot(x, x, 'k--', lw=2, alpha=0.7)
    ax.set_ylabel(r"$\theta'$", fontsize=18)

    θ_points = [clq.θ_space[np.argmax(clq.J_space)],
                clq.θ_series[1, -1]]

    for θ, label in zip(θ_points, labels):
        ax.scatter(θ, clq.θ_LB + 0.02 * clq.θ_range, 60, 'k', 'v')
        ax.annotate(label,
                    xy=(θ, clq.θ_LB + 0.01 * clq.θ_range),
                    xytext=(θ - 0.02 * clq.θ_range,
                            clq.θ_LB + 0.08 * clq.θ_range),
                    fontsize=18)

    ax = axes[1]
    μ_min = min(clq.μ_space)
    μ_max = max(clq.μ_space)
    μ_range = μ_max - μ_min
    ax.set_ylim([μ_min - 0.05 * μ_range, μ_max + 0.05 * μ_range])
    ax.plot(clq.θ_space, clq.μ_space, lw=2)
    ax.set_ylabel(r"$\mu(\theta)$", fontsize=18)

    for ax in axes:
        ax.set_xlabel(r"$\theta$", fontsize=18)
        ax.set_xlim([clq.θ_LB, clq.θ_UB])

    for θ, label in zip(θ_points, labels):
        ax.scatter(θ, μ_min - 0.03 * μ_range, 60, 'black', 'v')
        ax.annotate(label, xy=(θ, μ_min - 0.03 * μ_range),
                    xytext=(θ - 0.02 * clq.θ_range,
                            μ_min + 0.03 * μ_range),
                    fontsize=18)
    plt.tight_layout()
    plt.show()

plot_policy_functions(clq)
```

The following code generates a figure that plots sequences of $\mu$ and $\theta$ in the Ramsey plan and compares these to the constant levels in a MPE and in a Ramsey plan with a government restricted to set $\mu_t$ to a constant for all $t$.

```
def plot_ramsey_MPE(clq, T=15):
    """
    Method to plot Ramsey plan against Markov Perfect Equilibrium

    Here clq is an instance of ChangLQ
    """
    fig, axes = plt.subplots(1, 2, figsize=(12, 4))

    plots = [clq.θ_series[1, 0:T], clq.μ_series[0:T]]
    MPEs = [clq.θ_MPE, clq.μ_MPE]
    labels = [r"\theta", r"\mu"]

    axes[0].hlines(clq.θ_B, 0, T-1, 'r', label=r"$\theta^*$")

    for ax, plot, MPE, label in zip(axes, plots, MPEs, labels):
        ax.plot(plot, label=r"$" + label + "^R$")
        ax.hlines(MPE, 0, T-1, 'orange', label=r"$" + label + "^{MPE}$")
        ax.hlines(clq.μ_check, 0, T, 'g', label=r"$" + label + "^\check$")
        ax.set_xlabel(r"$t$", fontsize=16)
        ax.set_ylabel(r"$" + label + "_t$", fontsize=18)
        ax.legend(loc='upper right')

    plt.tight_layout()
    plt.show()

plot_ramsey_MPE(clq)
```

### 40.9.1 Time Inconsistency of Ramsey Plan

The variation over time in $\vec{\mu}$ chosen by the Ramsey planner is a symptom 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 (40.3).
- A government authority offered the opportunity to ignore effects on past utilities and to reoptimize at date $t \geq 1$ would, if allowed, want to deviate from a Ramsey plan.

**Note:** A modified Ramsey plan constructed under the restriction that $\mu_t$ must be constant over time is time consistent (see $\check{\mu}$ and $\check{\theta}$ in the above graphs).

### 40.9.2 Meaning of Time Inconsistency

In settings in which governments actually choose sequentially, many economists regard a time inconsistent plan implausible because of the incentives to deviate that occur along the plan.

A way to summarize this *defect* in a Ramsey plan is to say that it is not credible because there endure incentives for policymakers to deviate from it.

For that reason, the Markov perfect equilibrium concept attracts many economists.

- A Markov perfect equilibrium plan is constructed to insure that government policymakers who choose sequentially do not want to deviate from it.

The *no incentive to deviate from the plan* property is what makes the Markov perfect equilibrium concept attractive.

### 40.9.3 Ramsey Plans Strike Back

Research by Abreu [Abr88], Chari and Kehoe [CK90] [Sto89], and Stokey [Sto91] 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.

We turn to such theories of **sustainable plans** next.

## 40.10 A Fourth Model of Government Decision Making

This is a model in which

- The government chooses $\{\mu_t\}_{t=0}^{\infty}$ not once and for all at $t = 0$ but chooses to set $\mu_t$ at time $t$, not before.
- private agents' forecasts of $\{\mu_{t+j+1}, \theta_{t+j+1}\}_{j=0}^{\infty}$ respond to whether the government at $t$ **confirms** or **disappoints** their forecasts of $\mu_t$ brought into period $t$ from period $t-1$.
- the government at each time $t$ understands how private agents' forecasts will respond to its choice of $\mu_t$.
- at each $t$, the government chooses $\mu_t$ to maximize a continuation discounted utility of a representative household.

### 40.10.1 A Theory of Government Decision Making

$\vec{\mu}$ is chosen by a sequence of government decision makers, one for each $t \geq 0$.

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** private agents' 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 $\theta_0^A$ for $t+1$. Here $\vec{\mu}^A = \{\mu_j^A\}_{j=0}^{\infty}$ is an alternative government plan to be described below.

### 40.10.2 Temptation to Deviate from Plan

The government's one-period return function $s(\theta, \mu)$ described in equation (40.6) above has the property that for all $\theta$

$$-s(\theta, 0) \geq -s(\theta, \mu)$$

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 $\mu$.

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 $\mu_t$ would bring for continuation government payoffs via alterations in the private sector's expectations.

## 40.11 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 $\mu_t$.

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.

### 40.11.1 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 private agents' 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

$$\begin{aligned} v_j^A &= -s(\theta_j^A, \mu_j^A) + \beta v_{j+1}^A \\ &\geq -s(\theta_j^A, 0) + \beta v_0^A \equiv v_j^{A,D}, \quad j \geq 0 \end{aligned} \tag{40.10}$$

(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 private agents' expectations by following the plan, while the second line tells the consequences of disappointing private agents' 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

$$\begin{aligned} v_j &= -s(\theta_j, \mu_j) + \beta v_{j+1} \\ &\geq -s(\theta_j, 0) + \beta v_0^A \quad \forall j \geq 0 \end{aligned} \tag{40.11}$$

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 ([Abr88]) to construct a self-enforcing plan with low time 0 value.

### 40.11.2 Abreu Carrot-Stick Plan

Abreu ([Abr88]) 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 $\mu_t$ 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.

### 40.11.3 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 $\theta^A$, $\mu^A$ 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 $\theta_t^A$ associated with this **Abreu plan** starts the Ramsey plan from its beginning, i.e., $\theta_{t+10}^A = \theta_t^R \;\; \forall t \geq 0$.

```
def abreu_plan(clq, T=1000, T_A=10, μ_bar=0.1, T_Plot=20):

    # Append Ramsey μ series to stick μ series
    clq.μ_A = np.append(np.full(T_A, μ_bar), clq.μ_series[:-T_A])

    # Calculate implied stick θ series
    clq.θ_A = np.zeros(T)
    discount = np.zeros(T)
    for t in range(T):
        discount[t] = (clq.α / (1 + clq.α))**t
    for t in range(T):
        length = clq.μ_A[t:].shape[0]
        clq.θ_A[t] = 1 / (clq.α + 1) * sum(clq.μ_A[t:] * discount[0:length])

    # Calculate utility of stick plan
    U_A = np.zeros(T)
    for t in range(T):
        U_A[t] = clq.β**t * (clq.α0 + clq.α1 * (-clq.θ_A[t])
                 - clq.α2 / 2 * (-clq.θ_A[t])**2 - clq.c * clq.μ_A[t]**2)

    clq.V_A = np.zeros(T)
    for t in range(T):
        clq.V_A[t] = sum(U_A[t:] / clq.β**t)

    # Make sure Abreu plan is self-enforcing
    clq.V_dev = np.zeros(T_Plot)
    for t in range(T_Plot):
        clq.V_dev[t] = (clq.α0 + clq.α1 * (-clq.θ_A[t])
```

(continues on next page)

(continued from previous page)

```
                        - clq.α2 / 2 * (-clq.θ_A[t])**2) \
                        + clq.β * clq.V_A[0]

    fig, axes = plt.subplots(3, 1, figsize=(8, 12))

    axes[2].plot(clq.V_dev[0:T_Plot], label="$V^{A, D}_t$", c="orange")

    plots = [clq.θ_A, clq.μ_A, clq.V_A]
    labels = [r"$\theta_t^A$", r"$\mu_t^A$", r"$V^A_t$"]

    for plot, ax, label in zip(plots, axes, labels):
        ax.plot(plot[0:T_Plot], label=label)
        ax.set(xlabel="$t$", ylabel=label)
        ax.legend()

    plt.tight_layout()
    plt.show()

abreu_plan(clq)
```

0.10
0.05
0.00
−0.05
−0.10
$\theta_t^A$
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
$t$

0.10
0.05
0.00
−0.05
−0.10
$\mu_t^A$
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
$t$

$V_t^{A,D}$
$V_t^A$
6.6
6.5
6.4
6.3
6.2
6.1
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
$t$

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 (40.10) 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.α0 + clq.α1 * (-clq.θ_series[1, t])
                    - clq.α2 / 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
```

### 40.11.4 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:

$$\begin{aligned} \hat{\mu}_t &= \nu_\mu(v_t) \\ \theta_t &= \nu_\theta(v_t) \\ v_{t+1} &= \nu_v(v_t, \mu_t) \end{aligned} \tag{40.12}$$

In addition to these equations, we need an initial value $v_0$ to characterize a sustainable plan.

The first equation of (40.12) tells the recommended value of $\hat{\mu}_t$ as a function of the promised value $v_t$.

The second equation of (40.12) tells the inflation rate as a function of $v_t$.

The third equation of (40.12) updates the continuation value in a way that depends on whether the government at $t$ confirms private agents' expectations by setting $\mu_t$ equal to the recommended value $\hat{\mu}_t$, or whether it disappoints those expectations.

## 40.12 Whose Credible 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 private agents' forecasts of government actions.

Thus, $\vec{\mu}$ is both a government policy and a collection of private agents' 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 (40.11) that defines a credible policy.

## 40.13 Comparison of Equilibrium Values

We have computed plans for

- an ordinary (unrestricted) Ramsey planner who chooses a sequence $\{\mu_t\}_{t=0}^{\infty}$ at time $0$
- a Ramsey planner restricted to choose a constant $\mu$ for all $t \geq 0$
- a Markov perfect sequence of governments

Below we compare equilibrium time zero values for these three.

We confirm that the value delivered by the unrestricted Ramsey planner exceeds the value delivered by the restricted Ramsey planner which in turn exceeds the value delivered by the Markov perfect sequence of governments.

```
clq.J_series[0]
```

```
6.67918822960449
```

```
clq.J_check
```

```
6.676729524674898
```

```
clq.J_MPE
```

```
6.663435886995107
```

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.

## 40.14 Note on Dynamic Programming Squared

The theory deployed in this lecture is an application of what we nickname **dynamic programming squared**.

The nickname refers to the fact that a value satisfying one Bellman equation is itself an argument in a second Bellman equation.

Thus, our models have involved two Bellman equations:

- equation (40.1) expresses how $\theta_t$ depends on $\mu_t$ and $\theta_{t+1}$
- equation (40.4) expresses how value $v_t$ depends on $(\mu_t, \theta_t)$ and $v_{t+1}$

A value $\theta$ from one Bellman equation appears as an argument of a second Bellman equation for another value $v$.

CHAPTER

FORTYONE

# OPTIMAL TAXATION WITH STATE-CONTINGENT DEBT

**Contents**


- *Optimal Taxation with State-Contingent Debt*
  - *Overview*
  - *A Competitive Equilibrium with Distorting Taxes*
  - *Recursive Formulation of the Ramsey Problem*
  - *Examples*


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

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

## 41.1 Overview

This lecture describes a celebrated model of optimal fiscal policy by Robert E. Lucas, Jr., and Nancy Stokey [LS83].

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
%matplotlib inline
from scipy.optimize import root
from quantecon import MarkovChain
from quantecon.optimize.nelder_mead import nelder_mead
from interpolation import interp
from numba import njit, prange, float64
from numba.experimental import jitclass
```

## 41.2 A Competitive Equilibrium with Distorting Taxes

At time $t \geq 0$ a random variable $s_t$ belongs to a time-invariant set $S = [1, 2, \dots, S]$.

For $t \geq 0$, a history $s^t = [s_t, s_{t-1}, \dots, s_0]$ 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 $\ell_t$ and labor $n_t$:

$$n_t(s^t) + \ell_t(s^t) = 1 \tag{41.1}$$

Output equals $n_t(s^t)$ and can be divided between $c_t(s^t)$ and $g_t(s^t)$

$$c_t(s^t) + g_t(s^t) = n_t(s^t) \tag{41.2}$$

A representative household's preferences over $\{c_t(s^t), \ell_t(s^t)\}_{t=0}^{\infty}$ are ordered by

$$\sum_{t=0}^{\infty} \sum_{s^t} \beta^t \pi_t(s^t) u[c_t(s^t), \ell_t(s^t)] \tag{41.3}$$

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

$$g_t(s^t) = \tau_t(s^t) n_t(s^t) + \sum_{s_{t+1}} p_{t+1}(s_{t+1}|s^t) b_{t+1}(s_{t+1}|s^t) - b_t(s_t|s^{t-1}) \tag{41.4}$$

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

$$c_t(s^t) + \sum_{s_{t+1}} p_t(s_{t+1}|s^t) b_{t+1}(s_{t+1}|s^t) = [1 - \tau_t(s^t)]\, n_t(s^t) + b_t(s_t|s^{t-1}) \quad \forall t \geq 0 \tag{41.5}$$

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 (41.2) 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 (41.3) subject to (41.5) and (41.1) 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 (41.3).

### 41.2.1 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 $\{p_{t+1}(s_{t+1}|s^t)\}_{t=0}^{\infty}$

$$q_{t+1}^0(s^{t+1}) = p_{t+1}(s_{t+1}|s^t) q_t^0(s^t) \quad s.t. \quad q_0^0(s^0) = 1 \tag{41.6}$$

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.

### 41.2.2 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 (41.3) subject to the feasibility constraints (41.1) and (41.2) 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.

### 41.2.3 The Implementability Constraint

By sequential substitution of one one-period budget constraint (41.5) into another, we can obtain the household's present-value budget constraint:

$$\sum_{t=0}^{\infty}\sum_{s^t} q_t^0(s^t)c_t(s^t) = \sum_{t=0}^{\infty}\sum_{s^t} q_t^0(s^t)[1-\tau_t(s^t)]n_t(s^t) + b_0 \tag{41.7}$$

$\{q_t^0(s^t)\}_{t=1}^{\infty}$ 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

$$(1-\tau_t(s^t)) = \frac{u_l(s^t)}{u_c(s^t)} \tag{41.8}$$

and

$$p_{t+1}(s_{t+1}|s^t) = \beta\pi(s_{t+1}|s^t)\left(\frac{u_c(s^{t+1})}{u_c(s^t)}\right) \tag{41.9}$$

where $\pi(s_{t+1}|s^t)$ is the probability distribution of $s_{t+1}$ conditional on history $s^t$.

Equation (41.9) implies that the Arrow-Debreu price system satisfies

$$q_t^0(s^t) = \beta^t\pi_t(s^t)\frac{u_c(s^t)}{u_c(s^0)} \tag{41.10}$$

(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 (41.8) and (41.9) to eliminate taxes and prices from (41.7), we derive the *implementability condition*

$$\sum_{t=0}^{\infty}\sum_{s^t}\beta^t\pi_t(s^t)[u_c(s^t)c_t(s^t) - u_\ell(s^t)n_t(s^t)] - u_c(s^0)b_0 = 0 \tag{41.11}$$

The **Ramsey problem** is to choose a feasible allocation that maximizes

$$\sum_{t=0}^{\infty}\sum_{s^t}\beta^t\pi_t(s^t)u[c_t(s^t),1-n_t(s^t)] \tag{41.12}$$

subject to (41.11).

### 41.2.4 Solution Details

First, define a "pseudo utility function"

$$V\left[c_t(s^t),n_t(s^t),\Phi\right]=u[c_t(s^t),1-n_t(s^t)]+\Phi\left[u_c(s^t)c_t(s^t)-u_\ell(s^t)n_t(s^t)\right] \tag{41.13}$$

where $\Phi$ is a Lagrange multiplier on the implementability condition (41.7).

Next form the Lagrangian

$$J=\sum_{t=0}^{\infty}\sum_{s^t}\beta^t\pi_t(s^t)\Big\{V[c_t(s^t),n_t(s^t),\Phi]+\theta_t(s^t)\Big[n_t(s^t)-c_t(s^t)-g_t(s_t)\Big]\Big\}-\Phi u_c(0)b_0 \tag{41.14}$$

where $\{\theta_t(s^t);\forall s^t\}_{t\geq 0}$ is a sequence of Lagrange multipliers on the feasible conditions (41.2).

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 $\Phi$ 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

$$\begin{aligned}
c_t(s^t)\colon\ &(1+\Phi)u_c(s^t)+\Phi\left[u_{cc}(s^t)c_t(s^t)-u_{\ell c}(s^t)n_t(s^t)\right]-\theta_t(s^t)=0,\quad t\geq 1\\
n_t(s^t)\colon\ &-(1+\Phi)u_\ell(s^t)-\Phi\left[u_{c\ell}(s^t)c_t(s^t)-u_{\ell\ell}(s^t)n_t(s^t)\right]+\theta_t(s^t)=0,\quad t\geq 1
\end{aligned} \tag{41.15}$$

and

$$\begin{aligned}
c_0(s^0,b_0)\colon\ &(1+\Phi)u_c(s^0,b_0)+\Phi\left[u_{cc}(s^0,b_0)c_0(s^0,b_0)-u_{\ell c}(s^0,b_0)n_0(s^0,b_0)\right]-\theta_0(s^0,b_0)\\
&\qquad-\Phi u_{cc}(s^0,b_0)b_0=0\\
n_0(s^0,b_0)\colon\ &-(1+\Phi)u_\ell(s^0,b_0)-\Phi\left[u_{c\ell}(s^0,b_0)c_0(s^0,b_0)-u_{\ell\ell}(s^0,b_0)n_0(s^0,b_0)\right]+\theta_0(s^0,b_0)\\
&\qquad+\Phi u_{c\ell}(s^0,b_0)b_0=0
\end{aligned} \tag{41.16}$$

Please note how these first-order conditions differ between $t=0$ and $t\geq 1$.

It is instructive to use first-order conditions (41.15) 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

$$\begin{aligned}
&(1+\Phi)u_c(c,1-c-g)+\Phi[cu_{cc}(c,1-c-g)-(c+g)u_{\ell c}(c,1-c-g)]\\
&\qquad=(1+\Phi)u_\ell(c,1-c-g)+\Phi[cu_{c\ell}(c,1-c-g)-(c+g)u_{\ell\ell}(c,1-c-g)]
\end{aligned} \tag{41.17}$$

where we have imposed conditions (41.1) and (41.2).

Equation (41.17) is one equation that can be solved to express the unknown $c$ as a function of the exogenous variable $g$ and the Lagrange multiplier $\Phi$.

We also know that time $t=0$ quantities $c_0$ and $n_0$ satisfy

$$\begin{aligned}
&(1+\Phi)u_c(c,1-c-g)+\Phi[cu_{cc}(c,1-c-g)-(c+g)u_{\ell c}(c,1-c-g)]\\
&\qquad=(1+\Phi)u_\ell(c,1-c-g)+\Phi[cu_{c\ell}(c,1-c-g)-(c+g)u_{\ell\ell}(c,1-c-g)]+\Phi(u_{cc}-u_{c,\ell})b_0
\end{aligned} \tag{41.18}$$

Notice that a counterpart to $b_0$ does *not* appear in (41.17), 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 (41.16) 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 $\Phi$.

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 (41.17) 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 $g$.

### 41.2.5 The Ramsey Allocation for a Given Multiplier

Temporarily take $\Phi$ as given.

We shall compute $c_0(s^0, b_0)$ and $n_0(s^0, b_0)$ from the first-order conditions (41.16).

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 $\Phi$.

$\Phi$ 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 $\Phi$.

### 41.2.6 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 $s \in [1, \dots, S]$ and transition matrix $\Pi$, 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.

### 41.2.7 Determining the Lagrange Multiplier

We complete the Ramsey plan by computing the Lagrange multiplier $\Phi$ on the implementability constraint (41.11).

Government budget balance restricts $\Phi$ via the following line of reasoning.

The household's first-order conditions imply

$$(1 - \tau_t(s^t)) = \frac{u_l(s^t)}{u_c(s^t)} \tag{41.19}$$

and the implied one-period Arrow securities prices

$$p_{t+1}(s_{t+1}|s^t) = \beta \Pi(s_{t+1}|s_t) \frac{u_c(s^{t+1})}{u_c(s^t)} \tag{41.20}$$

Substituting from (41.19), (41.20), and the feasibility condition (41.2) into the recursive version (41.5) of the household budget constraint gives

$$\begin{aligned} u_c(s^t)[n_t(s^t) - g_t(s^t)] + \beta \sum_{s_{t+1}} \Pi(s_{t+1}|s_t) u_c(s^{t+1}) b_{t+1}(s_{t+1}|s^t) \\ = u_l(s^t) n_t(s^t) + u_c(s^t) b_t(s_t|s^{t-1}) \end{aligned} \tag{41.21}$$

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 (41.21) while $\beta$ 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 (41.21) as

$$u_c(s)[n(s) - g(s)] + \beta \sum_{s'} \Pi(s'|s) x'(s') = u_l(s) n(s) + x(s) \tag{41.22}$$

where $s'$ denotes a next period value of $s$ and $x'(s')$ denotes a next period value of $x$.

Given $n(s)$ for $s = 1, \dots, S$, equation (41.22) is easy to solve for $x(s)$ for $s = 1, \dots, S$.

If we let $\vec{n}, \vec{g}, \vec{x}$ denote $S \times 1$ vectors whose $i$th elements are the respective $n, g$, and $x$ values when $s = i$, and let $\Pi$ be the transition matrix for the Markov state $s$, then we can express (41.22) as the matrix equation

$$\vec{u}_c(\vec{n} - \vec{g}) + \beta \Pi \vec{x} = \vec{u}_l \vec{n} + \vec{x} \tag{41.23}$$

This is a system of $S$ linear equations in the $S \times 1$ vector $x$, whose solution is

$$\vec{x} = (I - \beta \Pi)^{-1} [\vec{u}_c(\vec{n} - \vec{g}) - \vec{u}_l \vec{n}] \tag{41.24}$$

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

$$\vec{b} = \frac{\vec{x}}{\vec{u}_c} \tag{41.25}$$

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 $\Phi$, then use the first-order conditions and the feasibility conditions to compute $c(s_t), n(s_t)$ for $s \in [1, \dots, S]$ and $c_0(s_0, b_0)$ and $n_0(s_0, b_0)$, given $\Phi$.
   - these are $2(S+1)$ equations in $2(S+1)$ unknowns.
2. Solve the $S$ equations (41.24) for the $S$ elements of $\vec{x}$.
   - these depend on $\Phi$.
3. Find a $\Phi$ that satisfies

$$u_{c,0} b_0 = u_{c,0}(n_0 - g_0) - u_{l,0} n_0 + \beta \sum_{s=1}^{S} \Pi(s|s_0) x(s) \tag{41.26}$$

   by gradually raising $\Phi$ if the left side of (41.26) exceeds the right side and lowering $\Phi$ if the left side is less than the right side.
4. After computing a Ramsey allocation, recover the flat tax rate on labor from (41.8) and the implied one-period Arrow securities prices from (41.9).

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 $\Phi$.

### 41.2.8 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.

### 41.2.9 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) [AMSSeppala02], 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 [LS83]. 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 (41.17) and (41.18) simplify in the case of the CRRA utility function.

They become

$$(1+\Phi)[u_c(c) + u_n(c+g)] + \Phi[cu_{cc}(c) + (c+g)u_{nn}(c+g)] = 0 \tag{41.27}$$

and

$$(1+\Phi)[u_c(c_0) + u_n(c_0+g_0)] + \Phi[c_0u_{cc}(c_0) + (c_0+g_0)u_{nn}(c_0+g_0)] - \Phi u_{cc}(c_0)b_0 = 0 \tag{41.28}$$

In equation (41.27), 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 $b_0$:

$$b_0 + g_0 = \tau_0(c_0 + g_0) + \beta \sum_{s=1}^{S} \Pi(s|s_0) \frac{u_c(s)}{u_{c,0}} b_1(s) \tag{41.29}$$

where $\tau_0$ is the time $t = 0$ tax rate.

In equation (41.29), it is understood that

$$\tau_0 = 1 - \frac{u_{l,0}}{u_{c,0}}$$

### 41.2.10 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
```

(continues on next page)

(continued from previous page)

```
        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):
        '''
        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):
```

(continues on next page)

(continued from previous page)

```
        '''
        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):
        '''
        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
```

(continues on next page)

(continued from previous page)

```
        cn0_arr[:] = c0, n0

        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]
        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
```

(continues on next page)

(continued from previous page)

```
        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]
```

## 41.3 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 (41.21) 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`.

### 41.3.1 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 $t = 1, 2, \ldots$.

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 $0$.

### 41.3.2 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 $W(b, s)$.

### 41.3.3 The Continuation Ramsey Problem

The Bellman equation for a time $t \geq 1$ continuation Ramsey planner is

$$V(x, s) = \max_{n, \{x'(s')\}} u(n - g(s), 1 - n) + \beta \sum_{s' \in S} \Pi(s'|s) V(x', s') \tag{41.30}$$

where maximization over $n$ and the $S$ elements of $x'(s')$ is subject to the single implementability constraint for $t \geq 1$:

$$x = u_c(n - g(s)) - u_l n + \beta \sum_{s' \in S} \Pi(s'|s) x'(s') \tag{41.31}$$

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 (41.30) are $S + 1$ time-invariant policy functions

$$\begin{aligned} n_t &= f(x_t, s_t), \quad t \geq 1 \\ x_{t+1}(s_{t+1}) &= h(s_{t+1}; x_t, s_t),\ s_{t+1} \in S,\ t \geq 1 \end{aligned} \tag{41.32}$$

### 41.3.4 The Ramsey Problem

The Bellman equation of the time 0 Ramsey planner is

$$W(b_0, s_0) = \max_{n_0, \{x'(s_1)\}} u(n_0 - g_0, 1 - n_0) + \beta \sum_{s_1 \in S} \Pi(s_1|s_0) V(x'(s_1), s_1) \tag{41.33}$$

where maximization over $n_0$ and the $S$ elements of $x'(s_1)$ is subject to the time 0 implementability constraint

$$u_{c,0} b_0 = u_{c,0}(n_0 - g_0) - u_{l,0} n_0 + \beta \sum_{s_1 \in S} \Pi(s_1|s_0) x'(s_1) \tag{41.34}$$

coming from restriction (41.26).

Associated with a value function $W(b_0, n_0)$ that solves Bellman equation (41.33) are $S + 1$ time 0 policy functions

$$\begin{aligned} n_0 &= f_0(b_0, s_0) \\ x_1(s_1) &= h_0(s_1; b_0, s_0) \end{aligned} \tag{41.35}$$

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 (41.32) 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.

### 41.3.5 First-Order Conditions

Attach a Lagrange multiplier $\Phi_1(x,s)$ to constraint (41.31) and a Lagrange multiplier $\Phi_0$ to constraint (41.26).

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 (41.30) are

$$\beta\Pi(s'|s)V_x(x',s') - \beta\Pi(s'|s)\Phi_1 = 0 \tag{41.36}$$

for $x'(s')$ and

$$(1+\Phi_1)(u_c - u_l) + \Phi_1\left[n(u_{ll} - u_{lc}) + (n - g(s))(u_{cc} - u_{lc})\right] = 0 \tag{41.37}$$

for $n$.

Given $\Phi_1$, equation (41.37) is one equation to be solved for $n$ as a function of $s$ (or of $g(s)$).

Equation (41.36) implies $V_x(x',s') = \Phi_1$, while an envelope condition is $V_x(x,s) = \Phi_1$, so it follows that

$$V_x(x',s') = V_x(x,s) = \Phi_1(x,s) \tag{41.38}$$

Time $t = 0$: For the time 0 problem on the right side of the Ramsey planner's Bellman equation (41.33), first-order conditions are

$$V_x(x(s_1),s_1) = \Phi_0 \tag{41.39}$$

for $x(s_1), s_1 \in S$, and

$$\begin{aligned}(1+\Phi_0)(u_{c,0} - u_{n,0}) + \Phi_0\big[n_0(u_{ll,0} - u_{lc,0}) + (n_0 - g(s_0))(u_{cc,0} - u_{cl,0})\big] \\ - \Phi_0(u_{cc,0} - u_{cl,0})b_0 = 0\end{aligned} \tag{41.40}$$

Notice similarities and differences between the first-order conditions for $t \geq 1$ and for $t = 0$.

An additional term is present in (41.40) 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.

### 41.3.6 State Variable Degeneracy

Equations (41.38) and (41.39) imply that $\Phi_0 = \Phi_1$ and that

$$V_x(x_t,s_t) = \Phi_0 \tag{41.41}$$

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 $\Phi_0$, this function mapping $s_t$ into $x_t$ can be expressed as a vector $\vec{x}$ that solves equation (41.34) for $n$ and $c$ as functions of $g$ that are associated with $\Phi = \Phi_0$.

### 41.3.7 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 $\Phi$, 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 $b_0$.

### 41.3.8 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):
```

(continues on next page)

(continued from previous page)

```
            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.
        '''

        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()
```

(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
        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)

        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] = interp(self.x_grid, self.c1[:, s], x)
            nHist[t] = interp(self.x_grid, self.n1[:, s], x)
```

(continues on next page)

(continued from previous page)

```
            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] = interp(x_grid, self.c1[:, sprime], x)
                n[sprime] = interp(x_grid, self.n1[:, sprime], x)
            Euc = π[sHist[t-1]] @ Uc(c, 1-n)
            RHist[t-1] = Uc(cHist[t-1], 1-nHist[t-1]) / (self.pref.β * Euc)

            gHist = self.g[sHist]
            yHist = nHist

            if t < T-1:
                sprime = sHist[t+1]
                xHist[t] = interp(self.x_grid, self.xprime1[:, s, sprime], x)

        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):
    '''
```

(continues on next page)

(continued from previous page)

```
    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

    # 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] = interp(x_grid, V[:, sprime], xprime[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
```

## 41.4 Examples

We return to the setup with CRRA preferences described above.

### 41.4.1 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)
```

(continues on next page)

(continued from previous page)

```
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
```

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()
```

(continues on next page)

(continued from previous page)

```
# Solve sequential problem
seq = SequentialLS(crra_pref, π=π, g=g)
sHist_h = np.array([0, 1, 2, 3, 5, 5, 5])
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()
```

```
/tmp/ipykernel_14826/2593351655.py:130: DeprecationWarning: setting an array␣
↪element with a sequence. This was supported in some cases where the elements are␣
↪arrays with a single element. For example `np.array([1, np.array([2])],␣
↪dtype=int)`. In the future this will raise the same ValueError as `np.array([1,␣
↪[2]], dtype=int)`.
  cn0_arr[:] = c0, n0
/tmp/ipykernel_14826/2593351655.py:130: DeprecationWarning: setting an array␣
↪element with a sequence. This was supported in some cases where the elements are␣
↪arrays with a single element. For example `np.array([1, np.array([2])],␣
↪dtype=int)`. In the future this will raise the same ValueError as `np.array([1,␣
↪[2]], dtype=int)`.
  cn0_arr[:] = c0, n0
```

**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()
```

### 41.4.2 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.

### 41.4.3 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.

### 41.4.4 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]):
    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 $t=0$', 'Time $t \geq 1$'))
axes[1].set_xlabel('Initial Government Debt')

fig.tight_layout()
plt.show()
```

```
/tmp/ipykernel_14826/2593351655.py:130: DeprecationWarning: setting an array␣
↪element with a sequence. This was supported in some cases where the elements are␣
↪arrays with a single element. For example `np.array([1, np.array([2])],␣
↪dtype=int)`. In the future this will raise the same ValueError as `np.array([1,␣
↪[2]], dtype=int)`.
  cn0_arr[:] = c0, n0
```

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 $\tau_1^R$ 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()
```

```
/tmp/ipykernel_14826/2593351655.py:130: DeprecationWarning: setting an array␣
↪element with a sequence. This was supported in some cases where the elements are␣
↪arrays with a single element. For example `np.array([1, np.array([2])],␣
↪dtype=int)`. In the future this will raise the same ValueError as `np.array([1,␣
↪[2]], dtype=int)`.
  cn0_arr[:] = c0, n0
```

The tax rates in the figure are equal for only two values of initial government debt.

### 41.4.5 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()
```

```
/tmp/ipykernel_14826/2593351655.py:130: DeprecationWarning: setting an array␣
↪element with a sequence. This was supported in some cases where the elements are␣
↪arrays with a single element. For example `np.array([1, np.array([2])],␣
↪dtype=int)`. In the future this will raise the same ValueError as `np.array([1,␣
↪[2]], dtype=int)`.
  cn0_arr[:] = c0, n0
```

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.

### 41.4.6 Further Comments

A *related lecture* describes an extension of the Lucas-Stokey model by Aiyagari, Marcet, Sargent, and Seppälä (2002) [AMSSeppala02].

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

FORTYTWO

# OPTIMAL TAXATION WITHOUT STATE-CONTINGENT DEBT

**Contents**


- *Optimal Taxation without State-Contingent Debt*
  - *Overview*
  - *Competitive Equilibrium with Distorting Taxes*
  - *Recursive Version of AMSS Model*
  - *Examples*


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

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

## 42.1 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

%matplotlib inline
```

In *an earlier lecture*, we described a model of optimal taxation with state-contingent debt due to Robert E. Lucas, Jr., and Nancy Stokey [LS83].

Aiyagari, Marcet, Sargent, and Seppälä [AMSSeppala02] (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.

## 42.2 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 $s^t = [s_t, s_{t-1}, \dots, s_0]$.

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 $\ell_t$ and labor $n_t$:

$$n_t(s^t) + \ell_t(s^t) = 1 \tag{42.1}$$

Output equals $n_t(s^t)$ and can be divided between consumption $c_t(s^t)$ and $g(s_t)$

$$c_t(s^t) + g(s_t) = n_t(s^t) \tag{42.2}$$

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

$$\sum_{t=0}^{\infty} \sum_{s^t} \beta^t \pi_t(s^t) u[c_t(s^t), \ell_t(s^t)] \tag{42.3}$$

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 [AMSSeppala02] 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) [Bar79] than they do in Lucas and Stokey (1983) [LS83].

### 42.2.1 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]

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

$$\begin{aligned} b_t(s^{t-1}) &= \tau_t^n(s^t)n_t(s^t) - g(s_t) - T_t(s^t) + \frac{b_{t+1}(s^t)}{R_t(s^t)} \\ &\equiv z_t(s^t) + \frac{b_{t+1}(s^t)}{R_t(s^t)}, \end{aligned} \tag{42.4}$$

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 (42.4) yields:

$$b_t(s^{t-1}) = z_t(s^t) + \beta \sum_{s^{t+1}|s^t} \pi_{t+1}(s^{t+1}|s^t) \frac{u_c(s^{t+1})}{u_c(s^t)} \, b_{t+1}(s^t) \tag{42.5}$$

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 (42.5) 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 (42.5) 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:

$$b_t(s^{t-1}) = \sum_{j=0}^{\infty} \sum_{s^{t+j}|s^t} \beta^j \pi_{t+j}(s^{t+j}|s^t) \frac{u_c(s^{t+j})}{u_c(s^t)} \, z_{t+j}(s^{t+j}) \tag{42.6}$$

[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.

Notice how the conditioning sets in equation (42.6) 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

$$z_t(s^t) = \left[1 - \frac{u_\ell(s^t)}{u_c(s^t)}\right] [c_t(s^t) + g(s_t)] - g(s_t) - T_t(s^t) \,. \tag{42.7}$$

If we substitute appropriate versions of the right side of (42.7) for $z_{t+j}(s^{t+j})$ into equation (42.6), we obtain a sequence of *implementability constraints* on a Ramsey allocation in an AMSS economy.

Expression (42.6) at time $t = 0$ and initial state $s^0$ was also an *implementability constraint* on a Ramsey allocation in a Lucas-Stokey economy:

$$b_0(s^{-1}) = \mathbb{E}_0 \sum_{j=0}^{\infty} \beta^j \frac{u_c(s^j)}{u_c(s^0)} \, z_j(s^j) \tag{42.8}$$

Indeed, it was the *only* implementability constraint there.

But now we also have a large number of additional implementability constraints

$$b_t(s^{t-1}) = \mathbb{E}_t \sum_{j=0}^{\infty} \beta^j \frac{u_c(s^{t+j})}{u_c(s^t)} \, z_{t+j}(s^{t+j}) \tag{42.9}$$

Equation (42.9) must hold for each $s^t$ for each $t \geq 1$.

### 42.2.2 Comparison with Lucas-Stokey Economy

The expression on the right side of (42.9) 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.

### 42.2.3 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

$$\mathbb{E}_0 \sum_{j=0}^{\infty} \beta^j \frac{u_c(s^j)}{u_c(s^0)} \, z_j(s^j) \geq b_0(s^{-1}) \tag{42.10}$$

and

$$\mathbb{E}_t \sum_{j=0}^{\infty} \beta^j \frac{u_c(s^{t+j})}{u_c(s^t)} \, z_{t+j}(s^{t+j}) = b_t(s^{t-1}) \quad \forall \, t, s^t \tag{42.11}$$

given $b_0(s^{-1})$.

### Lagrangian Formulation

Let $\gamma_0(s^0)$ be a non-negative Lagrange multiplier on constraint (42.10).

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 (42.11).

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 (42.11), 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.

## 42.2.4 Some Calculations

It is helpful to apply two transformations to the Lagrangian.

Multiply constraint (42.10) by $u_c(s^0)$ and the constraints (42.11) by $\beta^t u_c(s^t)$.

Then a Lagrangian for the Ramsey problem can be represented as

$$\begin{aligned} J &= \mathbb{E}_0 \sum_{t=0}^{\infty} \beta^t \Big\{ u\left(c_t(s^t), 1 - c_t(s^t) - g(s_t)\right) \\ &\qquad + \gamma_t(s^t) \Big[ \mathbb{E}_t \sum_{j=0}^{\infty} \beta^j u_c(s^{t+j}) \, z_{t+j}(s^{t+j}) - u_c(s^t) \, b_t(s^{t-1}) \Big\} \\ &= \mathbb{E}_0 \sum_{t=0}^{\infty} \beta^t \Big\{ u\left(c_t(s^t), 1 - c_t(s^t) - g(s_t)\right) \\ &\qquad + \Psi_t(s^t) \, u_c(s^t) \, z_t(s^t) - \gamma_t(s^t) \, u_c(s^t) \, b_t(s^{t-1}) \Big\} \end{aligned} \tag{42.12}$$

where

$$\Psi_t(s^t) = \Psi_{t-1}(s^{t-1}) + \gamma_t(s^t) \quad \text{and} \quad \Psi_{-1}(s^{-1}) = 0 \tag{42.13}$$

[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 (42.11) takes a positive value, so $\gamma_t(\tilde{s}^t) > 0$.

In (42.12), 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

$$\begin{aligned} u_c(s^t) - u_\ell(s^t) + \Psi_t(s^t)\left\{\left[u_{cc}(s^t) - u_{c\ell}(s^t)\right] z_t(s^t) + u_c(s^t)\, z_c(s^t)\right\} \\ - \gamma_t(s^t)\left[u_{cc}(s^t) - u_{c\ell}(s^t)\right] b_t(s^{t-1}) = 0 \end{aligned} \tag{42.14}$$

and with respect to $b_t(s^t)$ as

$$\mathbb{E}_t\left[\gamma_{t+1}(s^{t+1})\, u_c(s^{t+1})\right] = 0 \tag{42.15}$$

If we substitute $z_t(s^t)$ from (42.7) and its derivative $z_c(s^t)$ into the first-order condition (42.14), 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 (42.14) 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 (42.14) may change over time in response to realizations of the state, while the multiplier $\Phi$ 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.
        '''
```

(continues on next page)

(continued from previous page)

```
        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):
        '''
        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
```

(continues on next page)

(continued from previous page)

```
        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):
        '''
        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, n0

        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
        '''
```

(continues on next page)

(continued from previous page)

```
        # 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]
        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*.

## 42.3 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 (42.8) from the Lucas-Stokey economy, but
- adds measurability constraints (42.6) 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.

### 42.3.1 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 [AMSSeppala02], who restricted transfers to be non-negative).

In this case, the household faces a sequence of budget constraints

$$b_t(s^{t-1}) + (1 - \tau_t(s^t))n_t(s^t) = c_t(s^t) + b_{t+1}(s^t)/R_t(s^t) \tag{42.16}$$

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 $\tau_t$ from budget constraint (42.16) gives

$$b_t(s^{t-1}) + \frac{u_{l,t}(s^t)}{u_{c,t}(s^t)}n_t(s^t) = c_t(s^t) + \frac{\beta(\mathbb{E}_t u_{c,t+1})b_{t+1}(s^t)}{u_{c,t}(s^t)} \tag{42.17}$$

or

$$u_{c,t}(s^t)b_t(s^{t-1}) + u_{l,t}(s^t)n_t(s^t) = u_{c,t}(s^t)c_t(s^t) + \beta(\mathbb{E}_t u_{c,t+1})b_{t+1}(s^t) \tag{42.18}$$

Now define

$$x_t \equiv \beta b_{t+1}(s^t)\mathbb{E}_t u_{c,t+1} = u_{c,t}(s^t)\frac{b_{t+1}(s^t)}{R_t(s^t)} \tag{42.19}$$

and represent the household's budget constraint at time $t$, history $s^t$ as

$$\frac{u_{c,t}x_{t-1}}{\beta \mathbb{E}_{t-1}u_{c,t}} = u_{c,t}c_t - u_{l,t}n_t + x_t \tag{42.20}$$

for $t \geq 1$.

### 42.3.2 Measurability Constraints

Write equation (42.18) as

$$b_t(s^{t-1}) = c_t(s^t) - \frac{u_{l,t}(s^t)}{u_{c,t}(s^t)} n_t(s^t) + \frac{\beta(\mathbb{E}_t u_{c,t+1}) b_{t+1}(s^t)}{u_{c,t}} \tag{42.21}$$

The right side of equation (42.21) 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 (42.21) must equal $b_t(s^{t-1})$.

That implies that it has to be *measurable* with respect to $s^{t-1}$.

Equations (42.21) are the *measurability constraints* that the AMSS model adds to the single time $0$ implementation constraint imposed in the Lucas and Stokey model.

### 42.3.3 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

$$V(x_-, s_-) = \max_{\{n(s), x(s)\}} \sum_s \Pi(s|s_-) \left[ u(n(s) - g(s), 1 - n(s)) + \beta V(x(s), s) \right] \tag{42.22}$$

subject to the following collection of implementability constraints, one for each $s \in S$:

$$\frac{u_c(s) x_-}{\beta \sum_{\tilde{s}} \Pi(\tilde{s}|s_-) u_c(\tilde{s})} = u_c(s)(n(s) - g(s)) - u_l(s) n(s) + x(s) \tag{42.23}$$

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

$$W(b_0, s_0) = \max_{n_0, x_0} u(n_0 - g_0, 1 - n_0) + \beta V(x_0, s_0) \tag{42.24}$$

where maximization is subject to

$$u_{c,0} b_0 = u_{c,0}(n_0 - g_0) - u_{l,0} n_0 + x_0 \tag{42.25}$$

### 42.3.4 Martingale Supercedes State-Variable Degeneracy

Let $\mu(s|s_-)\Pi(s|s_-)$ be a Lagrange multiplier on the constraint (42.23) 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

$$\beta V_x(x(s), s) = \mu(s|s_-) \tag{42.26}$$

Applying an envelope theorem to Bellman equation (42.22) gives

$$V_x(x_-, s_-) = \sum_s \Pi(s|s_-)\mu(s|s_-)\frac{u_c(s)}{\beta \sum_{\tilde{s}} \Pi(\tilde{s}|s_-)u_c(\tilde{s})} \tag{42.27}$$

Equations (42.26) and (42.27) imply that

$$V_x(x_-, s_-) = \sum_s \left( \Pi(s|s_-)\frac{u_c(s)}{\sum_{\tilde{s}} \Pi(\tilde{s}|s_-)u_c(\tilde{s})} \right) V_x(x, s) \tag{42.28}$$

Equation (42.28) 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:

$$\check{\Pi}(s|s_-) \equiv \Pi(s|s_-)\frac{u_c(s)}{\sum_{\tilde{s}} \Pi(\tilde{s}|s_-)u_c(\tilde{s})}$$

**Exercise**: Please verify that $\check{\Pi}(s|s_-)$ 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.

### 42.3.5 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.

### 42.3.6 Digression on Non-negative Transfers

Throughout this lecture, we have imposed that transfers $T_t = 0$.

AMSS [AMSSeppala02] 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 (42.22) subject to (42.23) 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.

### 42.3.7 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)
```

(continues on next page)

(continued from previous page)

```
            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[:]

        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
```

(continues on next page)

(continued from previous page)

```
        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))

        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
```

(continues on next page)

(continued from previous page)

```
        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):
            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:
```

(continues on next page)

(continued from previous page)

```
                    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

    return T_v, T_w
```

## 42.4 Examples

We now turn to some examples.

### 42.4.1 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):
```

(continues on next page)

(continued from previous page)

```
        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 5min 32s, sys: 161 ms, total: 5min 32s
Wall time: 3min 3s
```

```
# 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()
```

```
/tmp/ipykernel_10972/2593351655.py:130: DeprecationWarning: setting an array␣
↪element with a sequence. This was supported in some cases where the elements are␣
↪arrays with a single element. For example `np.array([1, np.array([2])],␣
↪dtype=int)`. In the future this will raise the same ValueError as `np.array([1,␣
↪[2]], dtype=int)`.
  cn0_arr[:] = c0, n0
```

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 9min 42s, sys: 420 ms, total: 9min 42s
Wall time: 5min 5s
```

```
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()
```

```
/tmp/ipykernel_10972/2593351655.py:130: DeprecationWarning: setting an array␣
↪element with a sequence. This was supported in some cases where the elements are␣
↪arrays with a single element. For example `np.array([1, np.array([2])],␣
↪dtype=int)`. In the future this will raise the same ValueError as `np.array([1,␣
↪[2]], dtype=int)`.
  cn0_arr[:] = c0, n0
```

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()
```

```
/tmp/ipykernel_10972/2593351655.py:130: DeprecationWarning: setting an array␣
↪element with a sequence. This was supported in some cases where the elements are␣
↪arrays with a single element. For example `np.array([1, np.array([2])],␣
↪dtype=int)`. In the future this will raise the same ValueError as `np.array([1,␣
↪[2]], dtype=int)`.
  cn0_arr[:] = c0, n0
```

Consumption
Complete Markets
Incomplete Markets
Labor Supply
Government Debt
Tax Rate
Government Spending
Output

CHAPTER

FORTYTHREE

# FLUCTUATING INTEREST RATES DELIVER FISCAL INSURANCE

**Contents**


- *Fluctuating Interest Rates Deliver Fiscal Insurance*
  - *Overview*
  - *Forces at Work*
  - *Logical Flow of Lecture*
  - *Example Economy*
  - *Reverse Engineering Strategy*
  - *Code for Reverse Engineering*
  - *Short Simulation for Reverse-engineered: Initial Debt*
  - *Long Simulation*
  - *BEGS Approximations of Limiting Debt and Convergence Rate*


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

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

## 43.1 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 [AMSSeppala02] model described in *optimal taxation without state-contingent debt* generally differs from a Ramsey allocation and Ramsey policy in the Lucas-Stokey [LS83] 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 [LS83] 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 (42.8) are zero.
- After the implementability constraints (42.8) 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 [LS83] 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
%matplotlib inline
from scipy.optimize import fsolve, fmin
```

## 43.2 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 [BEGS17] (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.

## 43.3 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.

### 43.3.1 Equations from Lucas-Stokey (1983) Model

Although we are studying an AMSS [AMSSeppala02] economy, a Lucas-Stokey [LS83] 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

$$\begin{aligned}(1+\Phi)u_c(c,1-c-g)+\Phi\big[cu_{cc}(c,1-c-g)-(c+g)u_{\ell c}(c,1-c-g)\big]\\ =(1+\Phi)u_\ell(c,1-c-g)+\Phi\big[cu_{c\ell}(c,1-c-g)-(c+g)u_{\ell\ell}(c,1-c-g)\big]\end{aligned} \tag{43.1}$$

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

$$\begin{aligned}(1+\Phi)u_c(c,1-c-g)+\Phi\big[cu_{cc}(c,1-c-g)-(c+g)u_{\ell c}(c,1-c-g)\big]\\ =(1+\Phi)u_\ell(c,1-c-g)+\Phi\big[cu_{c\ell}(c,1-c-g)-(c+g)u_{\ell\ell}(c,1-c-g)\big]+\Phi(u_{cc}-u_{c,\ell})b_0\end{aligned} \tag{43.2}$$

In addition, the time $t=0$ budget constraint is satisfied at $c_0$ and initial government debt $b_0$

$$b_0+g_0=\tau_0(c_0+g_0)+\frac{\bar{b}}{R_0} \tag{43.3}$$

where $R_0$ is the gross interest rate for the Markov state $s_0$ that is assumed to prevail at time $t=0$ and $\tau_0$ is the time $t=0$ tax rate.

In equation (43.3), 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.

### 43.3.2 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 (43.1) and (43.2) simplify in the case of the CRRA utility function.

They become

$$(1+\Phi)[u_c(c) + u_n(c+g)] + \Phi[cu_{cc}(c) + (c+g)u_{nn}(c+g)] = 0 \tag{43.4}$$

and

$$(1+\Phi)[u_c(c_0) + u_n(c_0+g_0)] + \Phi[c_0 u_{cc}(c_0) + (c_0+g_0)u_{nn}(c_0+g_0)] - \Phi u_{cc}(c_0)b_0 = 0 \tag{43.5}$$

In equation (43.4), 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):
```

(continues on next page)

(continued from previous page)

```
        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)
```

## 43.4 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 $2 \times 2$ 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()
```

(continues on next page)

(continued from previous page)

```
    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)) \
                              + 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
```

(continues on next page)

(continued from previous page)

```
    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
        '''
        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)
```

(continues on next page)

(continued from previous page)

```
        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 np.array([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
        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)
```

(continues on next page)

(continued from previous page)

```
        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
        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:]))
```

(continues on next page)

(continued from previous page)

```
        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_
        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]
```

(continues on next page)

(continued from previous page)

```
        return np.array([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:]
            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(
```

(continues on next page)

(continued from previous page)

```
                [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

            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 + \
```

(continues on next page)

(continued from previous page)

```
                [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:
            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]))
```

(continues on next page)

(continued from previous page)

```
    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]
    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]])
```

(continues on next page)

(continued from previous page)

```
    return sHist
```

## 43.5 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 [LS83] 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 [LS83] model presented in *optimal taxation with state-contingent debt*.

We use the following steps.

**Step 1:** Pick an initial $\Phi$.

**Step 2:** Given that $\Phi$, jointly solve two versions of equation (43.4) for $c(s), s = 1, 2$ associated with the two values for $g(s), s = 1, 2$.

**Step 3:** Solve the following equation for $\vec{x}$

$$\vec{x} = (I - \beta\Pi)^{-1}[\vec{u}_c(\vec{n} - \vec{g}) - \vec{u}_l\vec{n}] \tag{43.6}$$

**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

$$\vec{b} = \frac{\vec{x}}{\vec{u}_c} \tag{43.7}$$

**Step 5:** Compute $J(\Phi) = (b(1) - b(2))^2$.

**Step 6:** Put steps 2 through 6 in a function minimizer and find a $\Phi$ that minimizes $J(\Phi)$.

**Step 7:** At the value of $\Phi$ and the value of $\bar{b}$ that emerged from step 6, solve equations (43.5) and (43.3) jointly for $c_0, b_0$.

## 43.6 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
```

(continues on next page)

(continued from previous page)

```
        # 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)
```

(continues on next page)

(continued from previous page)

```
    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], 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
```

```
/tmp/ipykernel_9821/537151386.py:17: DeprecationWarning: setting an array element␣
↪with a sequence. This was supported in some cases where the elements are arrays␣
↪with a single element. For example `np.array([1, np.array([2])], dtype=int)`. In␣
↪the future this will raise the same ValueError as `np.array([1, [2]],␣
↪dtype=int)`.
  return np.array([eq1, eq2], dtype='float64')
```

```
(0.9344994030900681, -1.0386984075517638)
```

Thus, we have reverse engineered an initial $b0 = -1.038698407551764$ that ought to render the AMSS measurability constraints slack.

## 43.7 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
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']
```

(continues on next page)

(continued from previous page)

```
# 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()
```

```
/usr/share/miniconda3/envs/quantecon/lib/python3.8/site-packages/scipy/optimize/
↪optimize.py:282: RuntimeWarning: Values in x were outside bounds during a␣
↪minimize step, clipping to bounds
  warnings.warn("Values in x were outside bounds during a "
```

```
/tmp/ipykernel_9821/428611703.py:24: RuntimeWarning: divide by zero encountered in␣
↪reciprocal
  U = (c**(1 - σ) - 1) / (1 - σ)
/tmp/ipykernel_9821/428611703.py:29: RuntimeWarning: divide by zero encountered in␣
↪power
  return c**(-self.σ)
/tmp/ipykernel_9821/2177315689.py:249: RuntimeWarning: invalid value encountered␣
↪in true_divide
  x * u_c / Eu_c - u_c * (c - T) - Un(c, n) * n - β * xprime,
/tmp/ipykernel_9821/2177315689.py:249: RuntimeWarning: invalid value encountered␣
↪in multiply
  x * u_c / Eu_c - u_c * (c - T) - Un(c, n) * n - β * xprime,
```

```
0.04094445433233126
```

```
0.0016732111461371488
```

```
0.0014846748479253665
```

```
0.0013137721368951308
```

```
0.0011814037130862774
```

```
0.0010559653362150854
```

```
0.00094466661651826316
```

```
0.00084638073175390 53
```

```
0.00075604537895473 34
```

```
0.00067560010350782 82
```

```
0.00060415284548582 47
```

```
0.00053960045190897 69
```

```
0.00048207169093919 55
```

```
0.00043082732100433 19
```

```
0.00038481851385203177
```

```
0.00034383521763016 77
```

```
0.00030724369344304985
```

```
0.00027450091468061093
```

```
0.00024531773293937 29
```

```
0.00021923324291458427
```

```
0.00019593539320936288
```

```
0.00017514303473879235
```

```
0.00015655939845999275
```

```
0.00013996737088209 74
```

```
0.00012514457782858082
```

```
0.00011190070825821409
```

```
0.00010007020010780538
```

```
8.949728532179505e-05
```

```
8.004975327094079e-05
```

```
7.16058524843781e-05
```

```
6.405840593152859e-05
```

```
5.731160522572503e-05
```

```
5.127970136590974e-05
```

```
4.588651728340637e-05
```

```
4.106390488994105e-05
```

```
3.675097001578613e-05
```

```
3.289357329953009e-05
```

```
2.9443322666296015e-05
```

```
2.6356778255166695e-05
```

```
2.359547692616178e-05
```

```
2.1124867627306168e-05
```

```
1.891429243792456e-05
```

```
1.6935989609248822e-05
```

```
1.5165570573885985e-05
```

```
1.3581075134437811e-05
```

```
1.2162766028367938e-05
```

```
1.08932275329519e-05
```

```
9.756678120405687e-06
```

```
8.739234394318832e-06
```

```
7.828320707330869e-06
```

```
7.0126028128616214e-06
```

```
6.282198776164311e-06
```

```
5.628118836991799e-06
```

```
5.04242761207365e-06
```

```
4.517800302064727e-06
```

```
4.048011467209264e-06
```

```
3.627182061383552e-06
```

```
3.2502280080469402e-06
```

```
2.912555247440669e-06
```

```
2.6100632277997536e-06
```

```
2.3390964203934027e-06
```

```
2.096300118130153e-06
```

```
1.8787856325266814e-06
```

```
1.6838896573518092e-06
```

```
1.5092762911985945e-06
```

```
1.3527904691811238e-06
```

```
1.2125869948528265e-06
```

```
1.0869367682883104e-06
```

```
9.743293511950372e-07
```

```
8.734258834193871e-07
```

```
7.829794246706134e-07
```

```
7.01928063865486e-07
```

```
6.292787068089227e-07
```

```
5.641636768488363e-07
```

```
5.058008590671254e-07
```

```
4.5348432258107277e-07
```

```
4.06590671514636e-07
```

```
3.645531968191354e-07
```

```
3.2687007244355e-07
```

```
2.930882555392625e-07
```

```
2.628035059320909e-07
```

```
2.3565299012594116e-07
```

```
2.1131173465782124e-07
```

```
1.8948856108995713e-07
```

```
1.6992249793819623e-07
```

```
1.5237969279958967e-07
```

```
1.3665058124585355e-07
```

```
1.2254732810649932e-07
```

```
1.0990161072750323e-07
```

```
9.85625503930891e-08
```

```
8.839493194179647e-08
```

```
7.927753806402066e-08
```

```
7.11017244295954e-08
```

```
6.377014611321005e-08
```

```
5.7195456771274966e-08
```

```
5.129946121086394e-08
```

```
4.6011950420151866e-08
```

```
4.12702673265941e-08
```

```
3.701791723181564e-08
```

```
3.320422527492953e-08
```

```
2.9783848623773116e-08
```

```
2.6716195267228474e-08
```

```
2.3964833445787186e-08
```

```
2.1497119091155865e-08
```

```
1.928376798030794e-08
```

```
1.7298536372375615e-08
```

```
1.5517881304424567e-08
```

```
1.3920704335644013e-08
```

```
1.2488075472140825e-08
```

```
1.1203024893636727e-08
```

```
1.005033520152302e-08
```

```
9.01635626747282e-09
```

```
8.088851549078394e-09
```

```
7.25684492694046e-09
```

```
6.51049309787113e-09
```

```
5.840969963796187e-09
```

```
5.240358843929338e-09
```

```
4.701559638147992e-09
```

```
4.218204682918539e-09
```

```
3.784583299777964e-09
```

```
3.3955745148222364e-09
```

```
3.0465825598035734e-09
```

```
2.733488019865053e-09
```

```
2.4525946349713553e-09
```

```
2.2005893140584196e-09
```

```
1.9744962382180747e-09
```

```
1.7716511306726106e-09
```

```
1.5896600539440279e-09
```

```
1.4263768091502018e-09
```

```
1.2798787706327043e-09
```

```
1.148436288069456e-09
```

```
1.0305003430129398e-09
```

```
9.247034181554512e-10
```

```
8.297313686371701e-10
```

```
7.44595673801017e-10
```

```
6.681487386772167e-10
```

```
5.996078829877791e-10
```

```
5.380786398072197e-10
```

```
4.82884899929127e-10
```

```
4.3335194495482e-10
```

```
3.889044996494e-10
```

```
3.4902066124403016e-10
```

```
3.1322685773757633e-10
```

```
2.8110758359206e-10
```

```
2.522829728646965e-10
```

```
2.2641401475909507e-10
```

```
2.031999458501809e-10
```

```
1.8236608046589692e-10
```

```
1.6366937394160904e-10
```

```
1.4689077314575126e-10
```

```
1.3183191329777743e-10
```

```
1.183180734470654e-10
```

```
1.0618983159170116e-10
```

```
9.530462934379308e-11
```

```
/tmp/ipykernel_9821/1795147193.py:158: VisibleDeprecationWarning: Creating an␣
↪ndarray from ragged nested sequences (which is a list-or-tuple of lists-or-
↪tuples-or ndarrays with different lengths or shapes) is deprecated. If you meant␣
↪to do this, you must specify 'dtype=object' when creating the ndarray.
  return np.array([cHist, nHist, Bhist, THist, sHist, μHist, RHist])
```

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.

## 43.8 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 $\tau_t$ 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()
```

```
/tmp/ipykernel_9821/1795147193.py:158: VisibleDeprecationWarning: Creating an␣
↪ndarray from ragged nested sequences (which is a list-or-tuple of lists-or-
↪tuples-or ndarrays with different lengths or shapes) is deprecated. If you meant␣
↪to do this, you must specify 'dtype=object' when creating the ndarray.
 return np.array([cHist, nHist, Bhist, THist, sHist, μHist, RHist])
```

### 43.8.1 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.

## 43.9 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 [BEGS17].

BEGS [BEGS17] 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 [BEGS17] expresses the time $t$ state $s$ government budget constraint as

$$\mathcal{B}(s) = \mathcal{R}_\tau(s, s_-)\mathcal{B}_- + \mathcal{X}_{\tau(s)}(s) \tag{43.8}$$

where the dependence on $\tau$ 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 (43.8) 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 $\mathcal{X}_t$

### 43.9.1 Asymptotic Mean

BEGS give conditions under which the ergodic mean of $\mathcal{B}_t$ is

$$\mathcal{B}^* = -\frac{\text{cov}^\infty(\mathcal{R}, \mathcal{X})}{\text{var}^\infty(\mathcal{R})} \tag{43.9}$$

where the superscript $\infty$ denotes a moment taken with respect to an ergodic distribution.

Formula (43.9) 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:

$$\mathcal{B}^* = \text{argmin}_{\mathcal{B}} \text{var}(\mathcal{R}\mathcal{B} + \mathcal{X}) \tag{43.10}$$

The minimand in criterion (43.10) is the measure of fiscal risk associated with a given tax-debt policy that appears on the right side of equation (43.8).

Expressing formula (43.9) in terms of our notation tells us that $\bar{b}$ should approximately equal

$$\hat{b} = \frac{\mathcal{B}^*}{\beta E_t u_{c,t+1}} \tag{43.11}$$

### 43.9.2 Rate of Convergence

BEGS also derive the following approximation to the rate of convergence to $\mathcal{B}^*$ from an arbitrary initial condition.

$$\frac{E_t(\mathcal{B}_{t+1} - \mathcal{B}^*)}{(\mathcal{B}_t - \mathcal{B}^*)} \approx \frac{1}{1 + \beta^2 \text{var}(\mathcal{R})} \tag{43.12}$$

(See the equation above equation (47) in [BEGS17])

### 43.9.3 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 (43.11).

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 (43.11).

After that, we move on to compute $\mathcal{B}^*$ in formula (43.9).

We'll also evaluate the BEGS criterion (43.8) at the limiting value $\mathcal{B}^*$

$$J(\mathcal{B}^*) = \text{var}(\mathcal{R}) \left(\mathcal{B}^*\right)^2 + 2\mathcal{B}^* \text{cov}(\mathcal{R}, \mathcal{X}) + \text{var}(\mathcal{X}) \tag{43.13}$$

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)
```

(continues on next page)

(continued from previous page)

```
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 (43.13) above.

Let's push our luck and compute the mean reversion speed in the formula above equation (47) in [BEGS17].

```
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

FORTYFOUR

# FISCAL RISK AND GOVERNMENT DEBT

**Contents**


- *Fiscal Risk and Government Debt*
  - *Overview*
  - *The Economy*
  - *Long Simulation*
  - *Asymptotic Mean and Rate of Convergence*


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

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

## 44.1 Overview

This lecture studies government debt in an AMSS economy [AMSSeppala02] 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 [BEGS17] (BEGS).

We study an AMSS economy [AMSSeppala02] 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, [BEGS17] (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 [BEGS17] section III.D carry typos that we correct below.

Let's start with some imports:

```
import matplotlib.pyplot as plt
%matplotlib inline
from scipy.optimize import minimize
```

## 44.2 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 $3 \times 3$ 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,
```

(continues on next page)

(continued from previous page)

```
                 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)
```

### 44.2.1 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)
```

## 44.3 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:

    '''
    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])
```

(continues on next page)

(continued from previous page)

```
    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)) \
                              + 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.')
```

(continues on next page)

(continued from previous page)

```
        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
        '''
        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 np.array([cHist, nHist, Bhist, THist, sHist, μHist, RHist])
```

```
import numpy as np
```

(continues on next page)

(continued from previous page)

```
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
        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
```

(continues on next page)

(continued from previous page)

```
        # 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
        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):
```

(continues on next page)

(continued from previous page)

```
        '''
        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_
        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 np.array([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),
```

(continues on next page)

(continued from previous page)

```
                                        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:]
            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):
```

(continues on next page)

(continued from previous page)

```
                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

            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))
```

(continues on next page)

(continued from previous page)

```
        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:
            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):
```

(continues on next page)

(continued from previous page)

```
        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]
    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)
```

(continues on next page)

(continued from previous page)

```
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()
```

```
/usr/share/miniconda3/envs/quantecon/lib/python3.8/site-packages/scipy/optimize/
↪optimize.py:282: RuntimeWarning: Values in x were outside bounds during a␣
↪minimize step, clipping to bounds
  warnings.warn("Values in x were outside bounds during a "
```

```
/tmp/ipykernel_15250/428611703.py:24: RuntimeWarning: divide by zero encountered␣
↪in reciprocal
  U = (c**(1 - σ) - 1) / (1 - σ)
/tmp/ipykernel_15250/428611703.py:29: RuntimeWarning: divide by zero encountered␣
↪in power
  return c**(-self.σ)
/tmp/ipykernel_15250/2177315689.py:249: RuntimeWarning: invalid value encountered␣
↪in true_divide
  x * u_c / Eu_c - u_c * (c - T) - Un(c, n) * n - β * xprime,
/tmp/ipykernel_15250/2177315689.py:249: RuntimeWarning: invalid value encountered␣
↪in multiply
  x * u_c / Eu_c - u_c * (c - T) - Un(c, n) * n - β * xprime,
```

```
0.03826635338765756
```

```
0.0015144378246638572
```

```
0.0013387575050040877
```

```
0.0011833202400557807
```

```
0.0010600307116102315
```

```
0.000950662032490561
```

```
0.0008518776517197146
```

```
0.0007625857031039501
```

```
0.00068195630616799 55
```

```
0.00060940029272314 67
```

```
0.00054430073567929 49
```

```
0.00048599500343786997
```

```
0.00043383959360344373
```

```
0.000387227308658161
```

```
0.00034559541217657 69
```

```
0.00030842870645341445
```

```
0.00027525901875920 07
```

```
0.00024566312919626 54
```

```
0.00021925988534004208
```

```
0.00019570695819679 73
```

```
0.00017469751641664 49
```

```
0.00015595697131384975
```

```
0.00013923987965580792
```

```
0.00012432704762338663
```

```
0.00011102285953507262
```

```
9.915283205416987e-05
```

```
8.856139176880467e-05
```

```
7.910986485602987e-05
```

```
7.067466533409551e-05
```

```
6.314566737386755e-05
```

```
5.642474600655553e-05
```

```
5.0424471419978145e-05
```

```
4.506694212936621e-05
```

```
4.028274354412395e-05
```

```
3.601001917158586e-05
```

```
3.219364287451057e-05
```

```
2.8784481101515106e-05
```

```
2.5738738818558473e-05
```

```
2.3017369740186686e-05
```

```
2.0585562762800045e-05
```

```
1.84122736284722e-05
```

```
1.6470097240068577e-05
```

```
1.4734148076784946e-05
```

```
1.3182214629349998e-05
```

```
1.1794654543098229e-05
```

```
1.0553943002719476e-05
```

```
9.44443615915512e-06
```

```
8.452171125199814e-06
```

```
7.56468159902658e-06
```

```
6.7708366507990055e-06
```

```
6.060699136016447e-06
```

```
5.425387645150192e-06
```

```
4.85697754174042e-06
```

```
4.3483827055120085e-06
```

```
3.893276487084616e-06
```

```
3.4860028892013943e-06
```

```
3.1215110639487147e-06
```

```
2.7952840202921567e-06
```

```
2.503284243329152e-06
```

```
2.241904734316987e-06
```

```
2.007920924637913e-06
```

```
1.7984472189030803e-06
```

```
1.6109041426881386e-06
```

```
1.442988331877059e-06
```

```
1.292635440930743e-06
```

```
1.1580011966873546e-06
```

```
1.0374362227527555e-06
```

```
9.294651312640365e-07
```

```
8.327660657786677e-07
```

```
7.461585785382834e-07
```

```
6.6858664244251e-07
```

```
5.991017361835585e-07
```

```
5.368606435890857e-07
```

```
4.811036963492191e-07
```

```
4.3115434197400447e-07
```

```
3.86404995905041e-07
```

```
3.463127381215619e-07
```

```
3.103914581845975e-07
```

```
2.7820605810944957e-07
```

```
2.4936654001918277e-07
```

```
2.2352416344433042e-07
```

```
2.0036658421647053e-07
```

```
1.7961407210183743e-07
```

```
1.6101613645358078e-07
```

```
1.4434846444957937e-07
```

```
1.2941019296216765e-07
```

```
1.1602139325369497e-07
```

```
1.0402093727029971e-07
```

```
9.326450128271e-08
```

```
8.362279458685582e-08
```

```
7.497999161054085e-08
```

```
6.723237237856022e-08
```

```
6.028699158811175e-08
```

```
5.4060583271401234e-08
```

```
4.8478549604961385e-08
```

```
4.347405134660798e-08
```

```
3.898720067424428e-08
```

```
3.49643347704944e-08
```

```
3.135736953489378e-08
```

```
2.8123214087750785e-08
```

```
2.522325395519629e-08
```

```
2.262288437284662e-08
```

```
2.0291089995716368e-08
```

```
1.8200077047464417e-08
```

```
1.632492929140762e-08
```

```
1.4643321700573224e-08
```

```
1.3135236757142941e-08
```

```
1.1782735357283523e-08
```

```
1.0569736069030654e-08
```

```
9.481822388932894e-09
```

```
8.50607289271253e-09
```

```
7.630899893776282e-09
```

```
6.8459205865908316e-09
```

```
6.141820610160711e-09
```

```
5.510253190209607e-09
```

```
4.943732867154185e-09
```

```
4.435549590733574e-09
```

```
3.979732271899858e-09
```

```
3.5708274231244475e-09
```

```
3.2040002959996427e-09
```

```
2.8749128202557366e-09
```

```
2.579676183036558e-09
```

```
2.314804183013932e-09
```

```
2.0771667394635677e-09
```

```
1.863961377838023e-09
```

```
1.6726701481673408e-09
```

```
1.5010383942057176e-09
```

```
1.347042984624116e-09
```

```
1.2088672079040676e-09
```

```
1.084885740270264e-09
```

```
9.736361312432929e-10
```

```
8.738107452127465e-10
```

```
7.842336709324767e-10
```

```
7.038523531518579e-10
```

```
6.317197710846445e-10
```

```
5.669900992553053e-10
```

```
5.089015058462129e-10
```

```
4.567716585732213e-10
```

```
4.099888914047554e-10
```

```
3.680033712806346e-10
```

```
3.3032275895673613e-10
```

```
2.965058552412645e-10
```

```
2.6615454256237937e-10
```

```
2.389140949083656e-10
```

```
2.14465584304658e-10
```

```
1.9252045686888675e-10
```

```
1.7282487196730194e-10
```

```
1.5514637480665194e-10
```

```
1.392779256507968e-10
```

```
1.2503495539341238e-10
```

```
1.1224947670324972e-10
```

```
1.007730284516321e-10
```

```
9.047218158645764e-11
```

```
/tmp/ipykernel_15250/1795147193.py:158: VisibleDeprecationWarning: Creating an␣
↪ndarray from ragged nested sequences (which is a list-or-tuple of lists-or-
↪tuples-or ndarrays with different lengths or shapes) is deprecated. If you meant␣
↪to do this, you must specify 'dtype=object' when creating the ndarray.
  return np.array([cHist, nHist, Bhist, THist, sHist, μHist, RHist])
```

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 [BEGS17] 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 [BEGS17] 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))

for i, id in enumerate([2, 3]):
```

(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
$\frac{B^*}{Eu_c}$
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)

```
    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

## 44.4 Asymptotic Mean and Rate of Convergence

We apply the results of BEGS [BEGS17] 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 [BEGS17].

As in *Fiscal Insurance via Fluctuating Interest Rates*, we recall that BEGS [BEGS17] used a particular notation to represent what we can regard as their generalization of an AMSS model.

We introduce some of the [BEGS17] 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 [BEGS17] call $\mathcal{X}_t$ the **effective** government deficit and $\mathcal{B}_t$ the **effective** government debt.

Equation (44) of [BEGS17] expresses the time $t$ state $s$ government budget constraint as

$$\mathcal{B}(s) = \mathcal{R}_\tau(s, s_-)\mathcal{B}_- + \mathcal{X}_\tau(s) \tag{44.1}$$

where the dependence on $\tau$ 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 (44.1) 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 $\mathcal{X}_t$

### 44.4.1 Asymptotic Mean

BEGS give conditions under which the ergodic mean of $\mathcal{B}_t$ is approximated by

$$\mathcal{B}^* = -\frac{\text{cov}^\infty(\mathcal{R}_\text{t}, \mathcal{X}_\text{t})}{\text{var}^\infty(\mathcal{R}_\text{t})} \tag{44.2}$$

where the superscript $\infty$ denotes a moment taken with respect to an ergodic distribution.

Formula (44.2) 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:

$$\mathcal{B}^* = \text{argmin}_{\mathcal{B}} \text{var}^\infty(\mathcal{R}\mathcal{B} + \mathcal{X}) \tag{44.3}$$

The minimand in criterion (44.3) measures **fiscal risk** associated with a given tax-debt policy that appears on the right side of equation (44.1).

Expressing formula (44.2) 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

$$\hat{b} = \frac{\mathcal{B}^*}{\beta E(E_t u_{c,t+1})} = \frac{\mathcal{B}^*}{\beta E(u_{c,t+1})} \tag{44.4}$$

where mathematical expectations are taken with respect to the ergodic distribution.

### 44.4.2 Rate of Convergence

BEGS also derive the following approximation to the rate of convergence to $\mathcal{B}^*$ from an arbitrary initial condition.

$$\frac{E_t(\mathcal{B}_{t+1} - \mathcal{B}^*)}{(\mathcal{B}_t - \mathcal{B}^*)} \approx \frac{1}{1 + \beta^2 \text{var}^\infty(\mathcal{R})} \tag{44.5}$$

(See the equation above equation (47) in BEGS [BEGS17])

### 44.4.3 More Advanced Topic

The remainder of this lecture is about technical material based on formulas from BEGS [BEGS17].

The topic involves interpreting and extending formula (44.3) for the ergodic mean $\mathcal{B}^*$.

### 44.4.4 Chicken and Egg

Notice how attributes of the ergodic distribution for $\mathcal{B}_t$ appear on the right side of formula (44.3) for approximating the ergodic mean via $\mathcal{B}^*$.

Therefor, formula (44.3) 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 (44.3)

So the primary use of equation (44.3) 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

### 44.4.5 Approximating the Ergodic Mean

BEGS also [BEGS17] 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 [BEGS17]

- 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 [BEGS17] text that we'll want to correct here

Here is a step-by-step description of the BEGS [BEGS17] approximation procedure.

### 44.4.6 Step by Step

**Step 1:** For a given $\tau$ we compute a vector of values $c_\tau(s), s = 1, 2, \dots, S$ that satisfy

$$(1-\tau)c_\tau(s)^{-\sigma} - (c_\tau(s) + g(s))^\gamma = 0$$

This is a nonlinear equation to be solved for $c_\tau(s), s = 1, \dots, S$.

$S = 3$ 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 [BEGS17] – 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 $c_\tau(s), s = 1, \dots, S$ for a given $\tau$, 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 $s = 1, \dots, S$.

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 $\tau$ 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 $\beta$, 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 [BEGS17] –the left side should be multiplied by $-1$.

- We have made this correction in the above equation.

For a given $\mathcal{B}$, let a $\tau$ that solves the above equation be called $\tau(\mathcal{B})$.

We'll use a Python root solver to find a $\tau$ 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.

$$J(\mathcal{B})(s) = \mathcal{R}_{\tau(\mathcal{B})}(s)\mathcal{B} + \mathcal{X}_{\tau(\mathcal{B})}(s), \quad s = 1, \dots, S$$

**Step 5:** Now that we have a way to compute the random variable $J(\mathcal{B})(s), s = 1, \dots, S$, 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

### 44.4.7 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))
```

```
     fjac: array([[-0.99990816, -0.00495351, -0.01261467],
        [-0.00515633,  0.99985715,  0.01609659],
        [-0.01253313, -0.01616015,  0.99979086]])
      fun: array([ 5.61814373e-10, -4.76900741e-10,  1.17474919e-11])
  message: 'The solution converged.'
     nfev: 11
      qtf: array([1.55568331e-08, 1.28322481e-08, 7.89913426e-11])
        r: array([ 4.26943131,  0.08684775, -0.06300593, -4.71278821, -0.0743338 ,
        -5.50778548])
   status: 1
  success: True
        x: array([0.93852387, 0.89231015, 0.84858872])
```

#### Step 2

```
n = c + u.G   # Compute labor supply
```

### 44.4.8 Note about Code

Remember that in our code $\pi$ is a $3 \times 3$ transition matrix.

But because we are studying an IID case, $\pi$ has identical rows and we need only to compute objects for one row of $\pi$.

This explains why at some places below we set $s = 0$ just to pick off the first row of $\pi$.

### 44.4.9 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 $\pi$

```
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 $\tau$

```
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 $\tau$ is to be computed as a function of B.

Note that 0.2 is the initial value for $\tau$ 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

FORTYFIVE

# COMPETITIVE EQUILIBRIA OF A MODEL OF CHANG

**Contents**


- *Competitive Equilibria of a Model of Chang*
  - *Overview*
  - *Setting*
  - *Competitive Equilibrium*
  - *Inventory of Objects in Play*
  - *Analysis*
  - *Calculating all Promise-Value Pairs in CE*
  - *Solving a Continuation Ramsey Planner's Bellman Equation*


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

```
!pip install polytope
```

## 45.1 Overview

This lecture describes how Chang [Cha98] 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 [APS90] to describe key features of the set of competitive equilibria

Roberto Chang [Cha98] chose a model of Calvo [Cal78] as a simple structure that conveys ideas that apply more broadly.

A textbook version of Chang's model appears in chapter 25 of [LS18].

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 quantecon as qe
import matplotlib.pyplot as plt
%matplotlib inline
```

```
`polytope` failed to import `cvxopt.glpk`.
```

```
will use `scipy.optimize.linprog`
```

### 45.1.1 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 [Cal78] 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 $t \geq 0$.

## 45.2 Setting

### 45.2.1 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

$$\sum_{t=0}^{\infty} \beta^t \left[u(c_t) + v(q_t M_t)\right] \tag{45.1}$$

subject to

$$q_t M_t \leq y_t + q_t M_{t-1} - c_t - x_t \tag{45.2}$$

and

$$q_t M_t \leq \bar{m} \tag{45.3}$$

Here $q_t$ is the reciprocal of the price level at $t$, which we can also call the *value of money*.

Chang [Cha98] 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 (45.2) 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 (45.3) imposes an exogenous upper bound $\bar{m}$ on the household's choice of real balances, where $\bar{m} \geq m^f$.

### 45.2.2 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

$$-x_t = m_t(1 - h_t) \tag{45.4}$$

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:

$$y_t = f(x_t), \tag{45.5}$$

where $f : \mathbb{R} \to \mathbb{R}$ satisfies $f(x) > 0$, is twice continuously differentiable, $f''(x) < 0$, and $f(x) = f(-x)$ for all $x \in \mathbb{R}$, so that subsidies and taxes are equally distorting.

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 (45.1).

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*.

### 45.2.3 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

$$\begin{aligned} u'(c_t) &= \lambda_t \\ q_t[u'(c_t) - v'(M_t q_t)] &\leq \beta u'(c_{t+1}) q_{t+1}, \quad = \text{ if } M_t q_t < \bar{m} \end{aligned}$$

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

$$m_t[u'(c_t) - v'(m_t)] \leq \beta u'(f(x_{t+1}))m_{t+1}h_{t+1}, \quad = \text{ if } m_t < \bar{m} \tag{45.6}$$

Define the following key variable

$$\theta_{t+1} \equiv u'(f(x_{t+1}))m_{t+1}h_{t+1} \tag{45.7}$$

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 (45.7) shows that $\theta_{t+1}$ 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 $\theta_{t+1}$.

The observation that the one dimensional promised marginal utility of real balances $\theta_{t+1}$ 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*.

## 45.3 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.

## 45.4 Inventory of Objects in Play

Chang constructs the following objects

1. A set $\Omega$ of initial marginal utilities of money $\theta_0$
    - Let $\Omega$ denote the set of initial promised marginal utilities of money $\theta_0$ 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 $\theta_0$ and a four-tuple of functions $(h, m, x, \Psi)$ mapping $\theta$ into this period's $(h, m, x)$ and next period's $\theta$, respectively.
- A competitive equilibrium can be represented recursively by iterating on

$$\begin{aligned} h_t &= h(\theta_t) \\ m_t &= m(\theta_t) \\ x_t &= x(\theta_t) \\ \theta_{t+1} &= \Psi(\theta_t) \end{aligned} \tag{45.8}$$

  starting from $\theta_0$

  The range and domain of $\Psi(\cdot)$ are both $\Omega$

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 $\Psi$ determine subsequent $\theta_t$'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 $\theta_t$ associated with the original Ramsey plan to $\theta_0$.
   - The incentive to reinitialize $\theta_t$ associated with this revolution experiment indicates the time-inconsistency of the Ramsey plan.
   - By resetting $\theta$ to $\theta_0$, 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 $s = 0, \ldots, t-1$.

## 45.5 Analysis

A competitive equilibrium is a triple of sequences $(\vec{m}, \vec{x}, \vec{h}) \in E^{\infty}$ that satisfies (45.2), (45.3), and (45.6).

Chang works with a set of competitive equilibria defined as follows.

**Definition:** $CE = \left\{(\vec{m}, \vec{x}, \vec{h}) \in E^{\infty} \text{ such that (45.2), (45.3), and (45.6) are satisfied }\right\}$.

$CE$ 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 [APS90] with insights of Kydland and Prescott [KP80].

**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 (45.2), (45.3), and (45.6).

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 (45.6) 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 $\Omega$ be the set of first period values of $\theta_0$ 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 $\Omega$ 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 $\theta$ 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

$$w(\theta) = \max_{x,m,h,\theta'} \left\{u(f(x)) + v(m) + \beta w(\theta')\right\} \tag{45.9}$$

where maximization is subject to

$$(m, x, h) \in E \text{ and } \theta' \in \Omega \tag{45.10}$$

and

$$\theta = u'(f(x))(m + x) \tag{45.11}$$

and

$$-x = m(1 - h) \tag{45.12}$$

and

$$m \cdot [u'(f(x)) - v'(m)] \leq \beta\theta', \quad = \text{ if } m < \bar{m} \tag{45.13}$$

Before we use this proposition to recover a recursive representation of the Ramsey plan, note that the proposition relies on knowing the set $\Omega$.

To find $\Omega$, Chang uses the insights of Kydland and Prescott [KP80] together with a method based on the Abreu, Pearce, and Stacchetti [APS90] iteration to convergence on an operator $B$ that maps continuation values into values.

We want an operator that maps a continuation $\theta$ into a current $\theta$.

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 (45.11), (45.12), and (45.13) hold.

Thus, $B(Q)$ is the set of first period $\theta$'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 $\Omega$ 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 $\Omega$ as the limit of iterations on $B$ provided that iterations begin from a sufficiently large initial set.

### 45.5.1 Some Useful Notation

Let $\vec{h}^t = (h_0, h_1, \dots, h_t)$ 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, $CE_\pi$ is the set of money growth sequences consistent with the existence of competitive equilibria.

Chang observes that $CE_\pi$ is nonempty and compact.

**Definition**: $\sigma$ 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 $\sigma$ belongs to $CE_\pi$.

Admissibility of $\sigma$ means that anticipated policy choices associated with $\sigma$ 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, $CE_\pi^0$ 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:** $CE_\pi^0 = \{h \in \Pi :$ there is $(m, \theta') \in [0, \bar{m}] \times \Omega$ such that $mu'[f((h-1)m) - v'(m)] \leq \beta\theta'$ 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 $\sigma$, an allocation rule $\alpha$ is called *competitive* if given any history $\vec{h}^{t-1}$ and $h_t \in CE_\pi^0$, the continuations of $\sigma$ and $\alpha$ after $(\vec{h}^{t-1}, h_t)$ induce a competitive equilibrium sequence.

### 45.5.2 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 $(w, \theta)$ 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 $(w', \theta')$ drawn from $Z$ as candidate continuation value, $\theta$ 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

$$w = u(f(x(h))) + v(m(h)) + \beta w'(h) \tag{45.14}$$

$$\theta = u'(f(x(h)))(m(h) + x(h)) \tag{45.15}$$

$$x(h) = m(h)(h-1) \tag{45.16}$$

$$m(h)(u'(f(x(h))) - v'(m(h))) \leq \beta\theta'(h) \tag{45.17}$$

$$\text{with equality if } m(h) < \bar{m}\Big\}$$

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 $(w, \theta)$ pair within this set that attains the highest possible value $w$.

This $(w, \theta)$ 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.

## 45.6 Calculating all Promise-Value Pairs in CE

Above we have defined the $D(Z)$ operator as:

$$D(Z) = \{(w, \theta) : \exists h \in CE_\pi^0 \text{ and } (m(h), x(h), w'(h), \theta'(h)) \in [0, \bar{m}] \times X \times Z$$

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 [JYC03].

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 quantecon as qe
import time

from scipy.spatial import ConvexHull
from scipy.optimize import linprog, minimize, minimize_scalar
from scipy.interpolate import UnivariateSpline
import numpy.polynomial.chebyshev as cheb


class ChangModel:
```

(continues on next page)

(continued from previous page)

```
    """
    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(c)
        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, 9) 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]
```

(continues on next page)

(continued from previous page)

```
            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):
                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]), \
```

(continues on next page)

(continued from previous page)

```
            (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]

        # 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))))
```

(continues on next page)

(continued from previous page)

```
        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
                    # 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]
```

(continues on next page)

(continued from previous page)

```
                                    + 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
            self.c0_c, self.c0_s = np.copy(self.c1_c), np.copy(self.c1_s)

            # Update bounds for w and 9
            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
```

(continues on next page)

(continued from previous page)

```
        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(c)
        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
        # 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

        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
```

(continues on next page)

(continued from previous page)

```
        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]
                res = minimize(p_fun,
                               lb1 + (ub1-lb1) / 2,
                               method='SLSQP',
                               bounds=bnds1,
                               constraints=cons1,
                               tol=1e-10)
                if res.success == True:
                    p_iter1[i] = -p_fun(res.x)
                res = minimize(p_fun2,
                               lb2 + (ub2-lb2) / 2,
                               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))
```

(continues on next page)

(continued from previous page)

```
        # 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]
            res = minimize(p_fun,
                           lb1 + (ub1-lb1) / 2,
                           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]
            res = minimize(p_fun2,
                           lb2 + (ub2-lb2)/2,
                           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[i] = np.dot(cheb.chebvander(scale, order-1), c) - p

        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
```

(continues on next page)

(continued from previous page)

```
        res = minimize(ValFun,
                       (θ_min + θ_max)/2,
                       bounds=[(θ_min, θ_max)])
        θ_series[0] = res.x

        # Simulate
        for i in range(30):
            θ = θ_series[i]
            res = minimize(p_fun,
                           lb1 + (ub1-lb1)/2,
                           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]
            res2 = minimize(p_fun2,
                            lb2 + (ub2-lb2)/2,
                            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)
```

```
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 80.17             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', 'o', zorder=1)
    w_min = min(ext_C[:, 0])

    # Label Ramsey Plan slightly to the right of the point
```

(continues on next page)

(continued from previous page)

```
    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 444.33             seconds
```

```
plot_competitive(ch2)
```

## 45.7 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 $\Omega$.

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 $\Omega$.

The two sets of parameters differ only in the level of $\beta$.

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)
```

```
/tmp/ipykernel_10273/3554151967.py:33: RuntimeWarning: invalid value encountered␣
↪in log
  uc = lambda c: np.log(c)
```

```
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)
```

```
/tmp/ipykernel_10273/3554151967.py:309: RuntimeWarning: invalid value encountered␣
↪in log
  uc = lambda c: np.log(c)
```

```
Convergence achieved after 15 iterations
```

```
/usr/share/miniconda3/envs/quantecon/lib/python3.8/site-packages/scipy/optimize/
↪optimize.py:282: RuntimeWarning: Values in x were outside bounds during a␣
↪minimize step, clipping to bounds
  warnings.warn("Values in x were outside bounds during a "
```

```
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))
```

```
(6.463131208001016e-06, 6.875539781958651e-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)):
    ax.plot(model.θ_grid, model.p_iter)
    ax.set(xlabel=r"$\theta$",
           ylabel=r"$J(\theta)$",
           title=rf"$\beta = {model.β}$")
```

(continues on next page)

(continued from previous page)

```
plt.show()
```

The next figure plots the optimal policy functions; values of $\theta', m, x, h$ for each value of the state $\theta$:

```
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 $\theta'$ chosen by the Ramsey planner quickly hits the upper limit of $\Omega$.

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 $\theta$ (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()
```

$\beta = 0.3$
$\theta$
$m$
$h$
$x$
t

$\beta = 0.8$
$\theta$
$m$
$h$
$x$
t

### 45.7.1 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

FORTYSIX

# CREDIBLE GOVERNMENT POLICIES IN A MODEL OF CHANG

**Contents**


- *Credible Government Policies in a Model of Chang*
  - *Overview*
  - *The Setting*
  - *Calculating the Set of Sustainable Promise-Value Pairs*


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

```
!pip install polytope
```

## 46.1 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 [CK90], also referred to as a *credible public policy* in [Sto89].

Technically, this lecture starts where lecture *competitive equilibria in the Chang model* on Ramsey plans within the Chang [Cha98] model stopped.

That lecture presents recursive representations of *competitive equilibria* and a *Ramsey plan* for a version of a model of Calvo [Cal78] 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 quantecon as qe
import polytope
import matplotlib.pyplot as plt
%matplotlib inline
```

```
`polytope` failed to import `cvxopt.glpk`.
```

```
will use `scipy.optimize.linprog`
```

## 46.2 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 $\vec{z}^t = (z_0, \dots, z_t)$, $\vec{z}_t = (z_t, z_{t+1}, \dots)$.

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.

### 46.2.1 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

$$\sum_{t=0}^{\infty} \beta^t \left[u(c_t) + v(q_t M_t)\right] \tag{46.1}$$

subject to

$$q_t M_t \leq y_t + q_t M_{t-1} - c_t - x_t \tag{46.2}$$

and

$$q_t M_t \leq \bar{m} \tag{46.3}$$

Here $q_t$ is the reciprocal of the price level at $t$, also known as the *value of money*.

Chang [Cha98] 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 (46.2) 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 (46.3) imposes an exogenous upper bound $\bar{m}$ on the choice of real balances, where $\bar{m} \geq m^f$.

### 46.2.2 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

$$-x_t = m_t(1 - h_t) \tag{46.4}$$

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:

$$y_t = f(x_t) \tag{46.5}$$

where $f : \mathbb{R} \to \mathbb{R}$ satisfies $f(x) > 0$, is twice continuously differentiable, $f''(x) < 0$, 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 (46.1).

### 46.2.3 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.

### 46.2.4 Household's Problem

Given $M_{-1}$ and $\{q_t\}_{t=0}^{\infty}$, the household's problem is

$$\begin{aligned}\mathcal{L} = \max_{\vec{c}, \vec{M}} \min_{\vec{\lambda}, \vec{\mu}} \sum_{t=0}^{\infty} \beta^t \{ & u(c_t) + v(M_t q_t) + \lambda_t [y_t - c_t - x_t + q_t M_{t-1} - q_t M_t] \\ & + \mu_t [\bar{m} - q_t M_t] \}\end{aligned}$$

First-order conditions with respect to $c_t$ and $M_t$, respectively, are

$$\begin{aligned} u'(c_t) &= \lambda_t \\ q_t[u'(c_t) - v'(M_t q_t)] &\leq \beta u'(c_{t+1}) q_{t+1}, \quad = \text{ if } M_t q_t < \bar{m} \end{aligned}$$

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

$$m_t[u'(c_t) - v'(m_t)] \leq \beta u'(f(x_{t+1})) m_{t+1} h_{t+1}, \quad = \text{ if } m_t < \bar{m} \tag{46.6}$$

Define the following key variable

$$\theta_{t+1} \equiv u'(f(x_{t+1}))m_{t+1}h_{t+1} \tag{46.7}$$

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 (46.7) shows that $\theta_{t+1}$ 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 $\theta_{t+1}$.

The observation that the one dimensional promised marginal utility of real balances $\theta_{t+1}$ 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*.

### 46.2.5 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.

### 46.2.6 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 $(w_0, \theta_0)$ 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 $w_t, \theta_t$ and in some cases $h_t$ into $\hat{h}_t, m_t, x_t, w_{t+1}$, and $\theta_{t+1}$, respectively.

- Starting from an initial condition $(w_0, \theta_0)$, a credible government policy can be constructed by iterating on these functions in the following order that respects the within-period timing:

$$\begin{aligned} \hat{h}_t &= h(w_t, \theta_t) \\ m_t &= m(h_t, w_t, \theta_t) \\ x_t &= x(h_t, w_t, \theta_t) \\ w_{t+1} &= \chi(h_t, w_t, \theta_t) \\ \theta_{t+1} &= \Psi(h_t, w_t, \theta_t) \end{aligned} \tag{46.8}$$

- 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 $\theta_t$, a promised marginal utility that the Ramsey plan delivers to the private sector.

For a credible government plan, we the two-dimensional state vector $(w_t, \theta_t)$ encodes history dependence.

### 46.2.7 Sustainable Plans

A government strategy $\sigma$ and an allocation rule $\alpha$ are said to constitute a *sustainable plan* (SP) if.

1. $\sigma$ is admissible.
2. Given $\sigma$, $\alpha$ is competitive.
3. After any history $\vec{h}^{t-1}$, the continuation of $\sigma$ is optimal for the government; i.e., the sequence $\vec{h}_t$ induced by $\sigma$ after $\vec{h}^{t-1}$ maximizes over $CE_\pi$ given $\alpha$.

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 $\Theta$.

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 $h \in CE_\pi^0$, 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 $\theta$ 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 [APS90].

Let $Z$ be a nonempty subset of $W \times \Omega$.

Think of using pairs $(w', \theta')$ drawn from $Z$ as candidate continuation value, promised marginal utility pairs.

Define the following operator:

$$\begin{aligned} \tilde{D}(Z) = \Big\{(w, \theta) : &\text{ there is } \hat{h} \in CE_\pi^0 \text{ and for each } h \in CE_\pi^0 \\ &\text{a four-tuple } (m(h), x(h), w'(h), \theta'(h)) \in [0, \bar{m}] \times X \times Z \end{aligned} \tag{46.9}$$

such that

$$w = u(f(x(\hat{h}))) + v(m(\hat{h})) + \beta w'(\hat{h}) \tag{46.10}$$

$$\theta = u'(f(x(\hat{h})))(m(\hat{h}) + x(\hat{h})) \tag{46.11}$$

and for all $h \in CE_\pi^0$

$$w \geq u(f(x(h))) + v(m(h)) + \beta w'(h) \tag{46.12}$$

$$x(h) = m(h)(h - 1) \tag{46.13}$$

and

$$m(h)(u'(f(x(h))) - v'(m(h))) \leq \beta \theta'(h) \tag{46.14}$$

$$\text{with equality if } m(h) < \bar{m} \Big\}$$

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 (46.12) 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 (46.8), which we repeat here for convenience:

$$
\begin{aligned}
\hat{h}_t &= h(w_t, \theta_t) \\
m_t &= m(h_t, w_t, \theta_t) \\
x_t &= x(h_t, w_t, \theta_t) \\
w_{t+1} &= \chi(h_t, w_t, \theta_t) \\
\theta_{t+1} &= \Psi(h_t, w_t, \theta_t)
\end{aligned}
$$

## 46.3 Calculating the Set of Sustainable Promise-Value Pairs

Above we defined the $\tilde{D}(Z)$ operator as (46.9).

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:

$$BR(Z) = \max P(h; Z) \text{ subject to } h \in CE_\pi^0$$

$BR(Z)$ 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:

$$E(Z) = \Big\{(w, \theta) : \exists h \in CE_\pi^0 \text{ and } (m(h), x(h), w'(h), \theta'(h)) \in [0, \bar{m}] \times X \times Z$$

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 quantecon as qe
import time

from scipy.spatial import ConvexHull
from scipy.optimize import linprog, minimize, minimize_scalar
from scipy.interpolate import UnivariateSpline
import numpy.polynomial.chebyshev as cheb


class ChangModel:
```

(continues on next page)

(continued from previous page)

```
    """
    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(c)
        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, 9) 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]
```

(continues on next page)

(continued from previous page)

```
            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):
                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]), \
```

(continues on next page)

(continued from previous page)

```
            (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]

        # 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))))
```

(continues on next page)

(continued from previous page)

```
        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
                    # 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]
```

(continues on next page)

(continued from previous page)

```
                                    + 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
            self.c0_c, self.c0_s = np.copy(self.c1_c), np.copy(self.c1_s)

            # Update bounds for w and 9
            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
```

(continues on next page)

(continued from previous page)

```
        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(c)
        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
        # 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

        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
```

(continues on next page)

(continued from previous page)

```
        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]
                res = minimize(p_fun,
                               lb1 + (ub1-lb1) / 2,
                               method='SLSQP',
                               bounds=bnds1,
                               constraints=cons1,
                               tol=1e-10)
                if res.success == True:
                    p_iter1[i] = -p_fun(res.x)
                res = minimize(p_fun2,
                               lb2 + (ub2-lb2) / 2,
                               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))
```

(continues on next page)

(continued from previous page)

```
        # 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]
            res = minimize(p_fun,
                           lb1 + (ub1-lb1) / 2,
                           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]
            res = minimize(p_fun2,
                           lb2 + (ub2-lb2)/2,
                           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[i] = np.dot(cheb.chebvander(scale, order-1), c) - p

        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
```

(continues on next page)

(continued from previous page)

```
        res = minimize(ValFun,
                       (θ_min + θ_max)/2,
                       bounds=[(θ_min, θ_max)])
        θ_series[0] = res.x

        # Simulate
        for i in range(30):
            θ = θ_series[i]
            res = minimize(p_fun,
                           lb1 + (ub1-lb1)/2,
                           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]
            res2 = minimize(p_fun2,
                            lb2 + (ub2-lb2)/2,
                            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)
```

### 46.3.1 Comparison of Sets

The set of $(w, \theta)$ associated with sustainable plans is smaller than the set of $(w, \theta)$ 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 $\beta$, another with a higher $\beta$

```
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 80.53             seconds
```

The following plot shows both the set of $w, \theta$ pairs associated with competitive equilibria (in red) and the smaller set of $w, \theta$ 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)
```

(continues on next page)

(continued from previous page)

```
    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)

    # 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 446.67             seconds
```

Let's plot both sets

```
plot_equilibria(ch2)
```

Evidently, the Ramsey plan is now sustainable.

# Part X

# Other

CHAPTER

FORTYSEVEN

# TROUBLESHOOTING

**Contents**


- *Troubleshooting*
  - *Fixing Your Local Environment*
  - *Reporting an Issue*


This page is for readers experiencing errors when running the code from the lectures.

## 47.1 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.

## 47.2 Reporting an Issue

One way to give feedback is to raise an issue through our issue tracker.

Please be as specific as possible. Tell us where the problem is and as much detail about your local set up as you can provide.

Another feedback option is to use our discourse forum.

Finally, you can provide direct feedback to contact@quantecon.org

CHAPTER

# FORTYEIGHT

# REFERENCES

CHAPTER

FORTYNINE

# EXECUTION STATISTICS

This table contains the latest execution statistics.

| Document | Modified | Method | Run Time (s) | Status |
|---|---|---|---|---|
| BCG_complete_mkts | 2021-11-24 01:40 | cache | 32.93 | ✅ |
| BCG_incomplete_mkts | 2021-11-24 01:40 | cache | 113.59 | ✅ |
| additive_functionals | 2021-11-24 01:40 | cache | 16.71 | ✅ |
| amss | 2021-11-24 01:40 | cache | 499.59 | ✅ |
| amss2 | 2021-11-24 01:40 | cache | 78.04 | ✅ |
| amss3 | 2021-11-24 01:40 | cache | 347.34 | ✅ |
| arellano | 2021-11-24 01:40 | cache | 21.37 | ✅ |
| arma | 2021-11-24 01:40 | cache | 9.02 | ✅ |
| asset_pricing_lph | 2021-11-24 01:40 | cache | 2.43 | ✅ |
| black_litterman | 2021-11-24 01:40 | cache | 61.02 | ✅ |
| calvo | 2021-11-24 01:40 | cache | 5.55 | ✅ |
| cattle_cycles | 2021-11-24 01:40 | cache | 4.69 | ✅ |
| chang_credible | 2021-11-24 01:40 | cache | 532.05 | ✅ |
| chang_ramsey | 2021-11-24 01:40 | cache | 735.09 | ✅ |
| classical_filtering | 2021-11-24 01:40 | cache | 1.43 | ✅ |
| coase | 2021-11-24 01:40 | cache | 7.82 | ✅ |
| cons_news | 2021-11-24 01:40 | cache | 5.65 | ✅ |
| discrete_dp | 2021-11-24 01:40 | cache | 52.03 | ✅ |
| dyn_stack | 2021-11-24 01:40 | cache | 8.02 | ✅ |
| estspec | 2021-11-24 01:40 | cache | 5.63 | ✅ |
| five_preferences | 2021-11-24 01:40 | cache | 34.64 | ✅ |
| growth_in_dles | 2021-11-24 01:40 | cache | 7.02 | ✅ |
| hs_invertibility_example | 2021-11-24 01:40 | cache | 11.0 | ✅ |
| hs_recursive_models | 2021-11-24 01:40 | cache | 2.02 | ✅ |
| intro | 2021-11-24 01:40 | cache | 2.02 | ✅ |
| irfs_in_hall_model | 2021-11-24 01:40 | cache | 4.91 | ✅ |
| knowing_forecasts_of_others | 2021-11-24 01:40 | cache | 27.02 | ✅ |
| lqramsey | 2021-11-24 01:40 | cache | 6.7 | ✅ |
| lu_tricks | 2021-11-24 01:40 | cache | 2.27 | ✅ |
| lucas_asset_pricing_dles | 2021-11-24 01:40 | cache | 4.93 | ✅ |
| lucas_model | 2021-11-24 01:40 | cache | 7.8 | ✅ |
| markov_jump_lq | 2021-11-24 01:40 | cache | 92.27 | ✅ |
| matsuyama | 2021-11-24 01:40 | cache | 8.02 | ✅ |
| muth_kalman | 2021-11-24 01:40 | cache | 6.02 | ✅ |
| opt_tax_recur | 2021-11-24 01:40 | cache | 50.03 | ✅ |

continues on next page

Table 49.1 – continued from previous page

| Document | Modified | Method | Run Time (s) | Status |
|---|---|---|---|---|
| orth_proj | 2021-11-24 01:40 | cache | 2.02 | ✅ |
| permanent_income_dles | 2021-11-24 01:40 | cache | 4.82 | ✅ |
| rob_markov_perf | 2021-11-24 01:40 | cache | 6.02 | ✅ |
| robustness | 2021-11-24 01:40 | cache | 7.42 | ✅ |
| rosen_schooling_model | 2021-11-24 01:40 | cache | 4.66 | ✅ |
| smoothing | 2021-11-24 01:40 | cache | 4.95 | ✅ |
| smoothing_tax | 2021-11-24 01:40 | cache | 7.53 | ✅ |
| stationary_densities | 2021-11-24 01:40 | cache | 10.8 | ✅ |
| status | 2021-11-24 01:40 | cache | 2.02 | ✅ |
| tax_smoothing_1 | 2021-11-24 01:40 | cache | 12.37 | ✅ |
| tax_smoothing_2 | 2021-11-24 01:40 | cache | 5.59 | ✅ |
| tax_smoothing_3 | 2021-11-24 01:40 | cache | 5.45 | ✅ |
| troubleshooting | 2021-11-24 01:40 | cache | 2.02 | ✅ |
| von_neumann_model | 2021-11-24 01:40 | cache | 2.85 | ✅ |
| zreferences | 2021-11-24 01:40 | cache | 2.02 | ✅ |

These lectures are built on `linux` instances through `github actions` so are executed using the following hardware specifications

# 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.

[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.

[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.

[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.

[Gal89] David Gale. *The theory of linear economic models*. University of Chicago press, 1989.

[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.

[HTW67] Michael J Hamburger, Gerald L Thompson, and Roman L Weil. Computation of expansion rates for the generalized von neumann model of an expanding economy. *Econometrica*, pages 542–547, 1967.

[HS08a] L P Hansen and T J Sargent. *Robustness*. Princeton University Press, 2008.

[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.

[HST99] Lars Peter Hansen, Thomas J. Sargent, and Thomas D. Tallarini. Robust Permanent Income and Pricing. *Review of Economic Studies*, 66(4):873–907, 1999. URL: https://ideas.repec.org/a/oup/restud/v66y1999i4p873-907..html, doi:.

[HS09] Lars Peter Hansen and José A Scheinkman. Long-term risk: an operator approach. *Econometrica*, 77(1):177–234, 2009.

[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.

[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.

[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.

[KMT56] John G Kemeny, Oskar Morgenstern, and Gerald L Thompson. A generalization of the von neumann model of an expanding economy. *Econometrica*, pages 115–135, 1956.

[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, doi:10.1086/296008.

[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.

[Sar91] Thomas J. Sargent. Equilibrium with signal extraction from endogenous variables. *Journal of Economic Dynamics and Control*, 15:245–273, 1991.

[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.

[vN28] John von Neumann. Zur theorie der gesellschaftsspiele. *Mathematische annalen*, 100(1):295–320, 1928.

[vN37] John von Neumann. Uber ein okonomsiches gleichungssystem und eine verallgemeinering des browerschen fixpunktsatzes. In *Erge. Math. Kolloq.*, volume 8, 73–83. 1937.

[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.

# INDEX


## A

AR, 464
ARMA, 462, 464
ARMA Processes, 457

## B

Bellman Equation, 397

## C

Coase's Theory of the Firm, 273
Complex Numbers, 463
Consumption
    Tax, 91
Continuous State Markov Chains, 23
Covariance Stationary, 459
Covariance Stationary Processes, 457
    AR, 460
    MA, 460

## D

Discrete State Dynamic Programming, 53

## E

Elementary Asset Pricing, 593

## F

Fixed Point Theory, 587

## G

General Linear Processes, 459

## L

Linear Markov Perfect Equilibria, 418
Lucas Model, 584
    Assets, 584
    Computation, 588
    Consumers, 584
    Dynamic Program, 585
    Equilibrium Constraints, 586
    Equilibrium Price Function, 586
    Pricing, 585
    Solving, 587

## M

MA, 464
Markov Chains
    Continuous State, 23
Markov Perfect Equilibrium
    Applications, 421
    Overview, 417
Models
    Additive functionals, 495, 721
    Lucas Asset Pricing, 583

## N

Nonparametric Estimation, 484

## O

Optimal Taxation in an LQ Economy, 211
Orthogonal Projection, 5

## P

Periodograms, 479, 480
    Computation, 481
    Interpretation, 480
python, 43, 139, 179, 189, 204, 289, 318, 328, 335, 344, 349, 356, 363, 439, 604

## R

Ramsey Problem
    Optimal Taxation, 211
Robustness, 397

## S

Smoothing, 479, 484
    Tax, 107
Spectra, 479
    Estimation, 479
Spectra, Estimation
    AR(1) Setting, 487
    Fast Fourier Transform, 479
    Pre-Filtering, 487
    Smoothing, 484, 485, 487

Spectral Analysis, 457, 458, 462
Spectral Densities, 463
Spectral Density, 464
    interpretation, 464
    Inverting the Transformation, 469
    Mathematical Theory, 469

## W

White Noise, 459, 464
Wold Representation, 460